Concepts of Programming Languages:
A Unified Approach

Karl Abrahamson

August 2011

Contents

Part I

Fundamental Concepts

Chapter 1

Introduction to Programming Languages

On December 17, 1903, the Wright Brothers achieved something that had, at the time, eluded everyone else who had been brave enough to try it: controlled powered flight in a heavier-than-air vehicle. Their first flight lasted only twelve seconds, but it signified the beginning of a new era in aviation. They did not achieve their feat by having a more powerful engine than anybody else. Their engine was homemade, and would burn itself up if run more than about fifteen minutes. They did not succeed merely because they had the biggest wings. They succeeded where others had failed (and survived where some others did not) largely because they recognized that the fundamental problem in powered flight was not so much power as control, and they put significant effort into solving the problem of how to keep the airplane flying where they wanted it to go.

With computers, we face a similar problem of control. Today's computers are capable of executing billions or trillions of basic instructions per second. Their speed of computation gives them fantastic potential as tools. But a computer is nothing without control over the instructions that it must execute, and in what order they must be done. That control comes from software.

Computer software is written in a variety of different programming languages, which come in several flavors. A language can be very closely tied to the architecture of the computer, as is an assembly language (a symbolic form of machine language), it can be loosely tied to the computer's architecture, as is the language C, or it can be almost completely independent of what a computer actually looks like, as is, for example, Haskell. The language that you choose strongly influences characteristics of the software that you write, including

- Ease of programming, and ease of subsequent modification of your software (which correlates with low development cost);
- Portability of your software to a variety of computers;
- Reliability of your software (freedom from mistakes);
- Efficiency of your software (including time and memory requirements).

When choosing a language, you must decide which of those characteristics is most important to you. You might find that choosing a language that is very easy to use leads to software that is unacceptably slow, or that uses too much memory. On the other hand, choosing a language that offers high efficiency can tend to lead either to low reliability or high cost. Without care, you might well find yourself in a position analogous to the one that faced some of the Wright Brothers' competitors, with an airplane that flies very fast, straight into the ground.

This book discusses different kinds of programming languages, including their desirable and undesirable characteristics and how they relate to one another. But the focus is not strictly on the languages themselves, for a language is nothing if it cannot be used. A major aim of this book is to show the reader how and when to use the different languages and the features that they provide.

How you solve a programming or algorithmic problem depends on the perspective and style that you adopt. This book includes material on problem solving from different perspectives, so that you can make practical use of different kinds of languages.

The remaining sections in this part discuss general issues that affect all (or many) kinds of languages. Later parts turn to specific kinds of languages.

Chapter 2

Language Classification

2.1. Imperative programming

In algebra, you learn that a quadratic equation $ax^2 + bx + c = 0$ generally has two solutions and that they are given by the quadratic formula,

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \tag{2.1.1}$$

How should you express the computation of one of these solutions, in which ± is replaced by +, in a program? One approach is to issue a sequence of commands. This *imperative* approach might lead to a program that, at a conceptual level, would read like the program of Figure 2.1. For historical reasons, the commands that form the basis of imperative programming are called *statements*.

The imperative approach is also called a *dynamic* approach since it views computation as a process in which things are constantly changing, and the intent of a program is to say how things change. Imperative programming is close to the way a computer's processor works, performing one instruction after another, and imperative programming lends itself well to computations that explicitly involve changing things, such as creating or destroying files or modifying what is shown on the monitor.

There is much more to imperative programming than just statements done one after another. A program needs to make decisions, and it needs to perform statements over and over. Controlling how a program selects statements to perform is an important part of any language that employs the imperative approach.

Imperative languages

Languages based on the imperative approach are broken down into subgroups, the two most important being the *procedural languages* and the *object-oriented languages*. Programs written in procedural languages focus on the commands, which are usually encapsulated into subprograms that are typically called procedures. The fundamental unit of a procedural program is the procedure. Pascal and C are examples of procedural languages, and Chapters 14 and 16 discuss procedural languages in more detail.

Programs written in object-oriented languages focus on the data that is being manipulated by commands. Commands and procedures are present in the program, but are

1. Square b, and call the result s.
2. Multiply a by c, and multiply that result by 4, and call the result d.
3. Subtract d from s, and call the result d. (Note that d has now changed.)
4. Take the square root of d, and call the result e.
5. Subtract b from e, and call the result x.
6. Divide x by twice a, and call the result x. (So x has changed.)
7. The answer is x.

Figure 2.1: **An imperative program to solve equation** $ax^2 + bx + c = 0$**.**

of secondary importance in the overall design and program structure. Procedures are tied together with the data that they manipulate, forming collections of variables and procedures that are called *objects*. The fundamental unit of an object-oriented program is the class, describing shared characteristics of related objects. C++, Java and Eiffel are examples of object-oriented languages. Parts XIII and XIV deal with object-oriented languages.

2.2. Declarative programming

Return to the problem of solving the quadratic equation $ax^2 + bx + c = 0$. Rather than saying how to compute the answer, a *declarative* approach tells what the answer is. Except for syntactic issues, a declarative program is just equation (2.1.1), with ± replaced by +. There is no dynamic aspect to this; it simply states a fact, namely that the given expression is a solution to the equation. In addition to being called a declarative approach, this kind of program is called *static*, since it does not call for changing things. Declarative programs are typically based on *expressions*, which simply have a value.

The choice of imperative or declarative style in a program can make a substantial difference in ease of programming, as the quadratic equation solution illustrates. Equation 2.1.1 is shorter and more obvious in its intent than Figure 2.1. But the declarative approach is not well suited to expressing aspects of programs that inherently involve changing things in the computer or its environment.

Declarative languages

The defining feature of a declarative language is that a program is a collection of *facts*. Declarative languages are broken into a few subgroups, the two most common of which are the *functional languages* and the *logic languages*.

In functional languages, facts are stated as equations, such as Equation (2.1.1). You run a functional program by replacing one expression by an equal one, continuing this process until an expression of a desired form (such as just a number) is obtained. For example, if you know that $x = 5$, then you replace expression $x + 4$ by the equal expression $5 + 4$, and in turn replace that by 9. Haskell and Standard ML are examples of functional languages.

In logic languages, the facts that are stated in programs consist of basic facts and implications. A basic fact might state that Rhonda is Larry's mother. An implication would state, for example, that if a person X is person Y's mother, then X is also Y's parent. (That is part of the definition, in a program, of what it means to be a parent.) Logic languages are designed to take over much of the control of how computation is done, leaving the programmer to think only about the facts that need to be stated. For example, the language implementation would be responsible for realizing, from the available facts, that Rhonda is Larry's parent. Prolog and Eclipse are examples of a logic programming languages.

2.3. Language classification

In reality, the line dividing the different kinds of programming languages is fuzzy. Imperative programs are supposed to be based on commands, with declarative programs based on facts. But when you write an imperative program, you are thinking of facts that guide the way you write commands. The quadratic equation solving program is written in a particular way because of a known fact (the quadratic formula), and that fact lies implicitly, below the surface, in the program. In a well-written program, the guiding facts are often written explicitly as comments.

Similarly, a declarative program must have, below the surface facts, a way of performing computation. Since computation is ultimately performed on a computer that executes instructions, there are commands lying beneath the surface of a declarative program. The distinction between the imperative and declarative languages lies in what is clearly visible in programs: commands or facts. In a declarative program, the program itself consists of facts, and the rules for evaluating programs are beneath the surface. In an imperative program, the commands are clearly visible, and the facts are hidden behind them.

As a rule, you will find that few languages fit strictly into one group. The language C++, for example, incorporates procedural aspects and object-oriented aspects, and even a few functional aspects. (You can write complex expressions in C++; you do not need or want to write commands for everything.) C++ is usually called an object-oriented language because that is its focus. Similarly, Standard ML incorporates some imperative features, and Haskell has some characteristics that are similar to object-oriented languages, but both of those languages focus on functional programming, so they are considered to be functional languages. Prolog is considered a logic programming language even though you can write some Prolog programs in an imperative style.

Other classifications of languages are also in common use. For example, some languages have been called *fourth generation languages*. A fourth generation language is typically characterized as being devoted to a special purpose, and having powerful operations that make it easy to use for its intended purpose. Examples are database languages that make it very easy to do relatively complex database queries. Applications such as spreadsheets can be viewed as fourth generation languages, since they incorporate the ability to do some forms of computing that are particular to spreadsheets. Languages such as Mathematica that are used for performing symbolic mathematics fit into this general classification as well. In Mathematica, it is very easy to ask for the derivative or integral of a mathematical expression. Javascript is specialized for performing computations in Web pages.

A Javascript program can, for example, get characteristics of the web page in which it is embedded. Some languages are highly specialized. The programming language Palingol, for example, is used to perform computations concerning the structure of DNA and RNA molecules. Particular fourth generation languages can fit into other classifications as well. Mathematica is mostly functional. Database languages can be designed to fit into almost any classification, including procedural, object-oriented, logic or functional.

For the most part, fourth generation languages are excellent for their intended uses, and very poor for anything else. (Try to imagine writing an operating system using a spreadsheet.) Another kind of language, called a *prototyping language*, is intended for general purpose computing, and offers very powerful operations that make it fairly easy to create programs quickly. The implementations of prototyping languages take over as much as possible from the programmer, making the programming job easy.

Many prototyping languages, however, are not well-principled or carefully thought out. Typically, they offer very little in the way of error checking, opting instead to get out of the programmer's way and let the programmer do things without hindrance. Programs written in prototyping languages typically run slowly. Examples of prototyping languages are Perl, Python, Griffin, shell programming languages and Visual Basic. Almost any language that is suitable to a given task can be used for prototyping, of course, and languages that are suitable for prototyping are occasionally used for fairly large software development, so the line is fuzzy.

2.4. Summary of terminology and concepts

Programming languages come in several different forms. The major groups are the declarative languages and the imperative languages. Other classifications include prototyping languages and fourth generation languages.

Terminology

Declarative language In a declarative language, a program is a collection of facts.

Expression An expression is a part of a program that computes a value.

Fourth generation language A fourth generation language is a specialized language with powerful operations for a particular kind of application.

Functional language A functional language is a declarative language where the facts are equations.

Imperative language In an imperative language, a program is a collection of commands (statements), with additional information to control the order in which commands are performed.

Logic language A logic language is a declarative language characterized by the use of logical concepts and notation, including implications among facts.

Object-oriented language An object-oriented language is an imperative language in which programmers focus on the data that their programs manipulate, and in which procedures are closely tied to that data.

Procedural language A procedural language is an imperative language where the focus is on commands and procedures.

Statement A statement is a command. It performs an action.

2.5. Exercises

2.1. What is the defining characteristic of a declarative programming language?

2.2. What is the defining characteristic of an imperative programming language?

2.3. Is a loop construct, such as a while loop, more likely to be found in a procedural program or a functional program? Why?

2.4. How does a prototyping language differ from a fourth generation language, in terms of how it is typically used?

2.6. Bibliographic notes

Wall et. al. [106] describe the prototyping language Perl. Prolog is covered by Clocksin and Mellish [31], and Meyer [75] gives a detailed description of Eiffel. Bird and Wadler [16] and Thompson [103] explain how to use Haskell, and Paulson [86] covers functional programming using Standard ML. Wolfram [114] describes Mathematica. Billoud, Kontic and Viari [15] describe Palingol.

Chapter 3

Encapsulation and Information Hiding

3.1. Encapsulation and modification

Programs are not created and left alone. A program that is in use is almost always also being modified and improved, and programmers spend most of their time modifying existing programs. So it makes sense for a programming language to provide features that make programs easier to modify.

One way to do that is to provide mechanisms for *encapsulation.* Generally, encapsulation refers to any way of hiding some aspects of something. For example, a function hides details about how it works. You call the sqrt function to compute a square root, and you probably don't care how it works. But, more to the point, if somebody changes the details of how it works, but makes it still compute square roots, you still don't care. Your program works just like it did before, without any modification. An encapsulation is a barrier beyond which certain modifications do not propagate.

3.2. Some kinds of encapsulations

Functions and other subprograms are an obvious encapsulation mechanism. A subprogram hides how it works; any modification to the subprogram that does not affect what it accomplishes cannot affect any other part of the program, and so cannot propagate out into the rest of the program. Encapsulations are also called *abstractions.* A function or procedure is a *procedural abstraction* since it hide procedural details.

Named constants are also a simple form of encapsulation. Suppose that you define a constant in a C++ program by

```
const int maxCookies = 100;
```

so that you can only have 100 cookies at once. The entire rest of the program refers to the constant, maxCookies, without being concerned what its value is. Later, changing maxCookies to 200 does not require modifying any other part of the program.

A type can be an encapsulation. It describes a particular kind of data value, hiding details of how the information about that value are represented in the program. For example, suppose that a program creates a type Stopwatch, where you can start and stop a Stopwatch, and can ask it for the elapsed time. You use a Stopwatch via subprograms. For example,

```
Stopwatch w;
start(w);
...
stop(w);
t = elapsedTime(w);
```

creates and uses a Stopwatch. There is clearly some encapsulation provided by the subprograms, start, stop and elapsedTime. But there is more to it than that. Somehow, a stopwatch needs to remember when it was started and when it was stopped, and possibly some other information. The detailed form of that information is hidden here. For example, there might be no way available to you to get that information directly. If you modify how the information is stored, you might need to change how start, stop and elapsedTime work, but you certainly will not need to change any other part of the program. When thought of as hiding information, types are called *data abstractions.*

You typically break a large program into a collection of modules. Each module has a collection of things such as constants, subprograms and types defined in it. But only some of those are available to other modules. The others are hidden, available just to this one module. Since modules hide things, they constitute a form of encapsulation. Any modification to hidden components that does not affect what the visible components do cannot propagage outside the module.

3.3. Intension and extension

Two useful ideas from logic are the ideas of *intension* and *extension.* The intension of a concept is what it means to you, in a conceptual and, preferably, permanent way. For example, the intension of the concept *the number of planets in the solar system* is just what it sounds like, the concept of the number of planets in the solar system.

The extension is the current value. The ancients counted five planets. (They did not count Earth as a planet, and did not know about planets beyond Saturn.) A few years ago, most people would have said that there are 9 planets. With the demotion of Pluto to a dwarf planet, that number has changed to 8.

The point is that extensions tend to change much more rapidly than intensions. So, to limit the effects of modifications, a programmer should work, wherever possible, in terms of intensions rather than in terms of extensions. That is where encapsulations come into play. The encapsulating concept is the intension. The current (encapsulated) definition is the extension. For example, if maxCookies is defined as it was previously, then the intension is *maxCookies*, the maximum number of cookies that can be held at once. The current extension is 100. No part of a program other than this one line would make use of 100 as the maximum number of cookies. In all other places, you refer to the intension, maxCookies.

3.4. Language support

Of course, all of this only works in programs if the language supports it. If there is no way to define a named constant, you cannot create the encapsulation maxCookies.

Almost all languages provide some kind of procedural abstraction, and most provide support for modules. Of the encapsulations mentioned above, the most difficult to do well is data abstraction. For example, suppose that you want to create type Stopwatch in C. So you define a type, indicating that a Stopwatch carries with it the times at which it started and stopped (each of type double, indicating a real number in C).

```
struct Stopwatch {
   double startTime;
   double stopTime;
};
```

Now comes a problem. If you put this type definition where other modules can use it, those other modules can make direct use of startTime and stopTime; they are not hidden at all, and there is no real data abstraction. To hide startTime and stopTime, you are allowed to replace the definition of Stopwatch by

```
struct Stopwatch;
```

suppressing the details. Unfortunately, doing that makes it impossible for other modules to create a Stopwatch because the compiler does not know how much memory to allocate for one. In C, there is no good way to accomplish data abstraction. C++, an extension of C, provides some ways to achieve genuine data abstraction via object-oriented programming. Those ideas are explored in Part XIII.

3.5. Summary of terminology and concepts

Encapsulations are an imporant tool for facilitating program modification. You use encapsulations by creating intensions (ideas) and extensions (current implementations of those ideas). Kinds of encapsulations include constants, subprograms, types and modules.

Terminology

Abstraction An abstraction is an encapsulation.

Data abstraction The term data abstraction refers to hiding details of how information is represented.

Encapsulation An encapsulation hides some information about something. Ideally, when you see an encapsulation, you see only what the encapsulation's designer wants you to see, and cannot see details that should be irrelevant to you.

Extension The extension of a concept is the current value or implementation of it. The extension of a procedure is the algorithm that it uses. The extension of a type tells exactly how information is represented.

Intension The intension of a concept is the concept itself. It tends to remain the same even as modifications to extensions are made.

Procedural abstraction The term procedural abstraction refers to using subprograms to hide information about how problems are solved.

3.6. Exercises

3.1. How do encapsulations aid modification of programs?

3.2. What is procedural abstraction?

3.3. What is data abstraction?

3.4. Modules are a form of encapsulation. What does a module hide?

3.5. How is a DVD player an abstraction or encapsulation? What does it hide?

3.6. How is a publisher of books an encapsulation? What does it hide?

Chapter 4

Implementation of Programming Languages

4.1. Compilers

The only programming language that runs directly on a computer is that computer's machine language, in which instructions to the processor are expressed. All other languages require some kind of support: the language implementation.

One way to implement a language is to provide a compiler, which translates a program from one language (the *source* language) to another (the *target* language). Compilers have a central position in software design since the vast majority of the machine-language programs in existence today were produced by compilers, as translations of programs written in higher level languages.

Compilers have been written for hundreds of different source languages. Compilers typically translate to machine language or to assembly language, a symbolic form of machine language. But some compilers translate from one higher level language to another. For example, you can find compilers for Pascal or Common Lisp that translate into C. To use them, you must take the C program that is produced and translate that to machine language using a C compiler.

4.2. Linkers

Large programs are rarely written as one large piece, but instead consist of several separate *modules*, each of which is translated separately into machine language by a compiler. So the output of a compiler is not really a full program, but is a piece of a program. You might expect that, to put the pieces together into a complete program, you just put them one after another into a single file. Unfortunately, that does not do the job. The problem is that each module can refer to things that are defined in other modules. For example, suppose that module A needs to use subprogram EXPLODE, created in module B. Module A will have an instruction that needs to contain the memory address where the machine-language instructions for EXPLODE is stored. But when the compiler is translating module A, it does not know that address, since it is defined in another module.

To deal with that, the compiler creates a machine language instruction with the address missing. It also adds information to the translated program indicating where those missing addresses are located, and which addresses they need installed. Machine language programs with this extra *linking* information in them are called *object-language* programs.[1] A tool called a *linker* puts several object-language programs together, replacing missing addresses by their correct values. For example, when the linker reads the translation of module A, it sees that there is a hole that must be filled in with the address of EXPLODE. When it reads module B, the linking information tells the linker where variable EXPLODE is located. The linker goes back to module A and inserts the correct address.

Using a linker introduces yet another stage into the process of translating a program. Now, you must compile and then link. But it allows you to compile each module separately, and, after making changes, to recompile only those modules that have been changed.

Linkers and libraries

A linker can easily combine a program with others that were compiled earlier. Those *library* files provide assorted subprograms that are of general use to programmers, from relatively mundane functions such as one that computes square roots to much more involved ones that render graphics on the screen. When a linker looks at a library, it does not copy the entire library into the final machine-language program, but only copies those subprograms that are used. Additionally, it copies any library subprograms that those library subprograms use, etc., until all references are satisfied, and the program can be run.

Static and dynamic linking

There are two general kinds of linking, static and dynamic. A *static linker* gets all of the subprograms, variables and other things from the libraries and combines them with the rest of the program into a single file. It has the advantage that the final product is a self-contained program, with all of the support that it needs built into itself.

A *dynamic linker* avoids copying functions and procedures into a program, but instead copies them (from the library) either when the program is loaded into memory or by inserting *stubs* in place of each library subprogram. When it runs, the stub causes the library file to be loaded, and replaces itself by a reference to the library function.

Dynamic linking is typically preferred. It has the advantage that, when a dynamically linked library module is loaded, the operating system can store just one copy of it in memory, even if several concurrently running programs need to use it. Also, some library modules, not being needed immediately (or possibly at all if the features that they provide are not used), are not loaded right away, speeding up the startup of a program. There are some disadvantages to dynamic linking. Programs typically rely on a given version of a library module. Loading a new version (possibly for new software to use) can render older software useless, if the new version exhibits any incompatibilities with the old version.

4.3. Interpreters

Another way to implement a programming language is via an *interpreter*, which works very differently from a compiler. It reads one source language instruction at a time and performs

[1] The term *object language* is an old one, and has nothing to do with the more recent notion of object-oriented programming.

the action called for by that instruction immediately. For example, after reading `x = 4`, it finds where it is storing variable x and makes that variable hold 4. No translation to another language is done. An interpreter is in a loop, reading and performing instructions, as follows.

1. Get the current instruction.
2. Determine the meaning of the instruction.
3. Perform the action called for by the instruction.
4. Prepare for getting the next instruction.

Imagine that you have purchased a bicycle that requires assembly. A metaphor for a program is a set of assembly instructions written in Japanese (assuming that you do not read Japanese). In this metaphor, a compiler is a person who translates the instructions from Japanese to English. Once the translation is done, the translator leaves. It is then up to someone who understands English to assemble the bicycle. There are two stages: translation and execution. In the bicycle metaphor, an interpreter is someone who knows Japanese and assembles the bicycle for you, following the instructions. With this approach, there is only one stage: ask for the bicycle to be assembled.

4.4. Comparison of compilers and interpreters

Compilers have several advantages over interpreters. A big one is efficiency. After you have compiled your program, the compiled version will typically run from one or three orders of magnitude faster than will an interpreter running the same program. The interpreter is slowed down because it spends most of its time finding instructions, deciding what to do, finding the appropriate data to work on and checking that everything is going correctly. Only a small percentage of the time is spent doing the actual work. The interpreter might need to perform one hundred of its instructions to simulate one instruction of the source language.

Another advantage is *chaining* of translations. Some compilers translate into languages other than machine language. Older compilers for the language ML compile into into Common Lisp. You must then translate the Common Lisp program to another language. If you translate from Common Lisp to C, you would need to translate the C program to machine language. This chaining of translators may seem awkward, but it works reasonably well, and the program obtained in the end can be quite reasonable in terms of speed. Imagine the same thing with an interpreter. If each interpreter introduces two orders of magnitude slowdown, as compared to machine language, then running an ML interpreter via a Lisp interpreter would introduce four orders of magnitude slowdown.

Yet another advantage of a compiler is that the compiler does not need to be present when the compiled program is run. You can sell a machine language program, translated from C, to somebody who does not have a C compiler. You might even be able to run the machine language program on a computer that is too small to support a C compiler. (Imagine writing a program to control a toaster.) In contrast, dialects of Basic are typically implemented using an interpreter. If you sell a Basic program to somebody, your customer will be unable to run the program without a suitable Basic interpreter to perform the actions called for by the program.

The advantages offered by compilers are compelling. So why would anybody choose to create an interpreter? One reason is that it is typically easier and cheaper to create an interpreter. So if you want to implement a new language, and are on a tight budget, an interpreter might be a good way to go. Another issue is portability. There are two aspects to portability for an interpreter. First, an interpreted program will run on any machine that has the interpreter, but if you sell a machine language program, it will only work on machines that support that machine language. Java is typically (but not exclusively) implemented by an interpreter, making Java programs highly portable. Second, the interpreter itself is portable, and can often be moved from one machine to another by simply recompiling it for the new machine. A compiler must be retargeted to produce machine language for the new machine.

Security can also fall on the side of interpreters. An interpreter can contain instructions to check what it is doing, both for the purpose of ensuring that nothing very bad is being done, and to ensure that a program stays within a set of security guidelines. Implementations of Java, for example, allow you to set security policies, limiting what a Java program is allowed to do.

The prize for efficiency usually goes to compilers, but that is not always the case. The cost of running the compiler needs to be taken into account. If you intend to compile a program once and run it many times, then the cost of compiling is irrelevant. But if you intend to run a program only once, as you might while testing or for prototyping, it can take more time to compile a program than to complete running it using an interpreter. For those purposes, interpreters can be preferable.

4.5. Hybrid implementations

Most interpreters are actually not pure interpreters, but are combinations of interpreters and compilers. A compiler is used to translate a program from the source language into some intermediate language that has been chosen to be easy to implement with an interpreter. Intermediate languages are often called *byte codes* because their instructions typically begin with a byte that tells the kind of instruction. A byte code is thought of as the machine language of an *abstract machine*, whose implementation is not a processor but an interpreter. Examples of abstract machines are the Java abstract machine and the .NET abstract machine.

A relatively recent kind of hybrid is called a *just-in-time* (jit) compiler, which is an interpreter with a kind of compiler available to it. While running a program, the interpreter keeps track of how often it performs each subprogram. When the number of calls to a particular subprogram becomes high enough to justify the cost, the interpreter compiles just that subprogram to machine language, and from then on runs the compiled version rather than using the interpreter for that subprogram. In practice, programs are found to spend a large majority of their time running a small number of subprograms, so this can greatly increase the execution speed of a program at a relatively small cost in compilation time.

The term “just in time” is derived from manufacturing practices in which materials and components do indeed arrive just in time to be used, reducing storage costs. For compilers, “just a little late” might be a more descriptive term, since a subprogram is compiled only after it has already been run in interpretive mode enough times to make it clear that it is worth compiling that subprogram. To compile every subprogram the first time it is called would slow down, not speed up, many executions. The term “on-the-fly compilation” is more descriptive.

4.6. Libraries and run-time support

There is more to a programming language implementation than a compiler or interpreter. One important aspect is a *library*, also sometimes called the language's *application program interface*, or API. Programmers do not want to be forced to write every little thing by themselves. A library provides commonly used operations so that a programmer can get to the job at hand quickly.

The advent of large, standardized libraries has had a profound influence on how software is developed and on programming language design. It is typically in the implementations of libraries that the most sophisticated features of languages are used and the facilities of languages are most heavily stressed. It is, consequently, the development of libraries that provides the most impetus to the development of new programming language features. Indeed, in studying programming languages, it is often useful to imagine how each feature would be used in libraries, and why it would be useful there, rather than how you would use it in specialized applications.

On the other hand, a library should not be thought of as the heart and soul of a programming language. Libraries tend to change rapidly as new tools are added and old ones are discarded. The language in which both your programs and the library are written is usually much more stable. Modern libraries are typically very large, dwarfing the language itself in size and complexity, so it is easy, if you are not careful, just to see the library.

Run-time support

Related to libraries is a body of software called the *run-time support*, which provides some basic services during execution, and of whose existence programmers are typically not even aware. An example of run-time support is a *garbage collector*,, which periodically locates memory that a program has stopped using and recycles it, so that it can be used again. Another example is an event manager. When you press a key on a keyboard, move a mouse, or push a button on the mouse, the event manager recognizes and processes the events. In text-based applications, the event manager typically handles keyboard input by collecting it into a buffer until a carriage return is typed. It is the event manager that understands that a backspace key calls for a deletion of a character from the buffer.

Like a library procedure, a run-time support procedure gets loaded into memory along with your program. But where a program uses a library function explicitly, the run-time support runs without being explicitly invoked by the program.

4.7. Languages are not their implementations

Confusing a language with its implementation can lead you to incorrect conclusions. For example, to understand how efficient programs written in a given language tend to be, an obvious thing to do is to run them to find out. Should the efficiency of an implementation be used to characterize the efficiency of a programming language? Consider Lisp, an old language dating from the 1960s. Lisp was, for a long time, implemented exclusively via interpreters, which are relatively slow, so it was generally accepted that Lisp was a slow language. Today, most Lisp programs are run via compilers, and Lisp programs run very fast.

It comes as no surprise that different implementations exhibit different efficiencies. But what about meaning? To determine the meaning of a program or subprogram, you might just run it to see what it does, letting the language implementation serve as language definition. We might agree, for example, that whatever a particular implementation does is considered, by definition, to be correct. But what if there turns out to be a bug in the implementation? The bug cannot be fixed, or the implementation becomes, by definition, incorrect. Even worse, imagine that you need to learn how to use a language, and, as documentation, someone hands you the listing of a compiler, and tells you to figure it out! Clearly, a language implementation is an awkward definition.

In spite of the obvious difficulties of using an implementation as a definition, programmers often find it convenient to use a language implementation as a definitive source about the language's meaning. For example, if you discover a new function, and you are not sure what that function does (probably due to poor documentation), you might find out by running some experiments. What you find out, of course, is what that particular implementation does. You might expect that all implementations would be required to do the same thing, but that is not always the case. Certain behaviors can be left unspecified in the definition of a language, and different implementations might do different things in those cases. For example, consider procedure test, written in Pascal, with comments written in braces.

```
procedure test;
  var i: integer;
  begin
    for i := 1 to 4 do
      begin {Do nothing}
      end;
    writeln(i) {Print i}
  end;
```

If you run this, something will be printed. But the definition of Pascal says that the value of a for-loop counter is unspecified at the end of the loop, so different implementations might print different values, and you can be misled by a test.[2]

Integrated development environments

An integrated development environment (IDE) is a combination of a specialized text editor, compiler or interpreter and debugger, and possibly other software design tools such as version control and project management tools, and sometimes tools that are designed to write some parts of programs for you, based on menus that you fill out. An IDE can be a tremendous productivity booster, but it should not be confused with the programming language that it implements and supports. It would not be reasonable to say that you like language X because you like the way the text editor makes it easy to find documentation. The distinction between the IDE and the programming language tends to be blurred in the IDE documentation, which understandably is mainly concerned with the features of the IDE itself.

[2] Typically, i will be either 4 or 5. But in Pascal, you can limit the type of variable i to be a value from 1 to 4, by declaring i to have type 1..4. If i has the value 5 after the loop, it would not even have the right type. Pascal allows a for-loop counter to have a value that is not merely unspecified, but is *undefined*, after the loop ends.

4.8. Summary of terminology and concepts

Compilers and interpreters are two common forms of programming language implementations. Compilers produce much more efficient programs than interpreters, but interpreters have advantages of flexibility, portability and security. In reality, most interpreters have an associated compiler that translates a program into a more convenient form, such as a byte code.

Compilers need additional tools, such as linkers, to help them. Libraries and run-time support are also an important part of the implementation, whether a compiler or an interpreter is used.

Be careful in using a language implementation as a complete characterization of the language. The implementation might, for example, make arbitrary choices for language features that are purposely left undefined in the language definition, and other implementations might make different choices.

Terminology

Abstract machine An abstract machine is a made-up machine whose machine language is used as the target language of a compiler. It is chosen both to be convenient for an interpreter to execute and to make compilation relatively easy.

Application program interface The term *application program interface* (API) has, in the past, normally been used for the collection of utilities provided by an operating system for programs to use. So it is the interface between the operating system and application programs. Large libraries can cover so much ground that they rival an operating system interface in complexity. Also, programs tend to use library features, which are more portable, rather than directly interacting with an operating system, so the term API is used for those libraries.

Byte code A byte code is the machine language of an abstract machine.

Compiler A compiler is a program that translates from a higher level language to another language, often an assembly language or machine language or byte code.

Interpreter An interpreter is a program that reads instructions one at a time and processes them as they are read. An interpreter is often combined with a compiler into a hybrid implementation.

Just-in-time compiler A just-in-time compiler is an interpreter with an incremental compiler attached to it. The interpreter compiles parts of a program as needed. Subprograms that are infrequently executed are run via the interpreter, while subprograms that are frequently executed are compiled for speed.

Library A library is a collection of useful functions, procedures and possibly other things for programmers to use.

Linker A linker combines several object-language modules, linking them together by replacing holes in the modules by addresses that are found in other modules. A static linker combines all referenced things into one file. A dynamic linker causes linking to be done when a program is loaded, or when certain functions are used, and results in a program that is incomplete, needing to be combined with dynamically linkable library modules.

Module A large program is usually broken up into separately compiled pieces, called modules.

Object language Object language is a form of machine language in which the machine-language program is combined with tables that indicate the positions of holes where addresses need to be filled in, and tells which addresses need to be put into those holes. Object language is the input language for a linker.

Run-time support Run-time support is software that is combined with a program automatically and that helps the program run, without being invoked directly by the program.

Source language The source language of a compiler is the language of its input. For example, Fortran is the source language of a Fortran compiler.

Target language The target language of a compiler is the language that it produces.

4.9. Exercises

4.1. What are the main differences between a compiler and an interpreter?

4.2. Do all compilers translate to machine language?

4.3. Explain why interpreters tend to be more portable than compilers.

4.4. Explain what a linker does. Provide some details, not just a vague description.

4.5. What is the difference between a static linker and a dynamic linker?

4.6. What is the difference between a library and the run-time support?

4.7. A particular language implementation has a subprogram called DORETURN that accomplishes a return from another subprogram by deallocating that subprogram's local memory and jumping back to the calling subprogram. Is DORETURN a library subprogram or a run-time support subprogram?

4.8. Give two important reasons why an implementation of a programming language is not a good definition of the language.

4.9. What is a just-in-time compiler?

4.10. Suppose that you write an interpreter for a new language called Fruitcake. You write the interpreter in Lisp, with the intent of using a Lisp interpreter to run your Lisp program. Imagine that it takes 100 machine language instructions to perform one Lisp instruction, and it takes 200 Lisp instructions to perform one Fruitcake instruction. How many machine language instructions does it take to simulate one Fruitcake instruction?

4.11. Where are programming language features typically put more to the test, in libraries or in application programs?

4.12. Suppose that all fundamental applications supported by a particular operating system are dynamically linked to a library that supplies subprograms to perform basic jobs. What would happen if you deleted the file that contains that library? Can you put it back? (If you are already running an application, you might have a way to do it, but what if you decide to start up a program to copy the old file back? Many applications perform file copying by starting a new program (or process) that performs the copy.)

4.10. Bibliographic notes

The structure of compilers is described by Aho, Sethi and Ullman [5] and by Grune, Bal, Jacobs and Langendoen [49]. Aycock [9] gives a history of just-in-time compilers. Lindholm and Yellin [70] describe the Java abstract machine and byte code.

Part II

Syntax and Semantics

Chapter 5

Form and Function

5.1. The syntax of a language

The *syntax* of a programming language tells what a program looks like, how a program is broken into words and symbols, how those words and symbols form larger units, and whether or not a given text document is a meaningful program. Syntax is what people tend to notice first about a language. Figure 5.1 shows the same thing written in three different programming languages, each with its own syntax. The differences between the Pascal and C forms are relatively small. For example, C uses a left brace where Pascal uses the word **begin**, and a right brace where Pascal uses **end**. But syntax can vary a great deal, as is apparent from the Scheme form of the same thing.

5.2. Form suggests function

Ideally, the appearance of the program reflects how the program is organized and what the program accomplishes. Most modern programming languages are *free form*, which means that a line break has the same meaning, in most contexts, as a space, that several spaces in a row are equivalent to one space, and that spaces are only required to be placed where they are needed to avoid misunderstanding. That gives programmers the freedom to write programs in a form that conveys structure and meaning. For example, the C program

```
if x < y then
  begin
    w := y;
    z := 0
  end
```

(a)

```
if(x < y)
{
  w = y;
  z = 0;
}
```

(b)

```
(if (< x y)
  (begin
    (set! ’w y)
    (set! ’z 0)
  )
  ()
)
```

(c)

Figure 5.1: A program fragment written in (a) Pascal, (b) C and (c) Scheme.

```
if(x<y)
{ w=y;
  z=0;
}
```

(a)

```
if(x<y){w=y;
z=0;}
```

(b)

Figure 5.2: **Two layouts of a C fragment.**

fragment in Figure 5.2(a) shows, through well chosen line breaks and indentation, which parts are done when $x < y$, as well as the fact that the two steps w = y and z = 0 are done one after the other. But a free-form language does not require a programmer to make form reflect function, and it is just a matter of taste how or whether to indent a program. Figure 5.2(b) shows the same program fragment in a less obvious form. Some languages prefer to make indentation mandatory. Haskell and Python use a form of syntax called *layout* in which indentation is a significant part of the meaning of a program. Haskell program fragment

```
let y = 2 * b
    f(x) = x + y
in f(c) * f(d)
```

computes $(c + 2 * b) * (d + 2 * b)$ by defining $y = 2 * b$ and function $f(x) = x + y = x + 2 * b$. It is indentation, not line breaks, that signal the end of the first part ($y = 2 * b$) and the start of the next ($f(x) = x + y$).

There is much more than indentation to give a programmer hints about a program. Well chosen words help a lot. Ada is a language originally designed for use on U.S. Department of Defense contracts. In Ada, a loop begins, naturally enough, with the word **loop**. A well-designed syntax has regular rules with few exceptions. It uses familiar notation where possible, and avoids notations that are misleading. The language C illustrates both good and bad choices in a single notational decision. To divide two numbers x and y, you write x/y in a C program, which is easy to read and remember because it is similar to familiar notation. But C also uses / to indicate division of integers, with the rule that, when m and n are integers, m/n is also an integer, obtained by dividing and ignoring any fractional part. In C, you create a variable that holds a real number by using the word **double** (odd choice?). Statement

double x = 2/3;

actually makes $x = 0.0$, because the division is between two integers 2 and 3. (The integer result, 0, is automatically converted to 0.0.) That certainly does not correspond with familiar notation. Ada solves that problem by using the word **div** to indicate integer division, at the cost of requiring the programmer to learn one more unfamiliar word.

Ideally, the syntax of a language should take into account that a programmer might make a mistake typing a program. A program that looks like it is saying one thing should not be easily transformed into something altogether different by the slip of a finger on a keyboard. Fortran offers an example of a poor choice in that respect. Figure 5.3 shows two Fortran program fragments that only differ in one place where a comma has been replaced by a period. Fragment (a) is a loop, but fragment (b) is a sequence of two statements.

```
      DO 20 I = 1,100
   20 X = F(X)
```

(a)

```
      DO 20 I = 1.100
   20 X = F(X)
```

(b)

Figure 5.3: **Two Fortran program fragments.** Fragment (a) is a loop, performing statement X = F(X) 100 times. (The number 20 after DO is the label of the last statement in the loop body, and labels are written in the left margin.) Fragment (b) is a pair of assignment statements, the first setting variable DO20I to 1.100. Spaces are ignored in Fortran, and variables, such as DO20I, are created automatically when they are used. Precisely this kind of mistake was discovered by NASA in 1961 in a program that had been in use on spacecraft for navigational calculations, and had been carried on some Mercury orbiters. Fortunately, the mistake only reduced the precision of the calculations, and no disaster occurred as a result of it.

Syntactic choices can influence how easy it is to learn and to use a language, and language designers give careful thought to syntax. Programmers, who have little choice but to use the syntax that is available, need to be aware of pitfalls, and to pay close attention to avoid them.

5.3. Summary of terminology and concepts

Careful choice of syntax can ease the process of developing software by making programs easy to write and to read, and by making mistakes more obvious and so less likely to be made. A guiding principle in the choice of syntax is that a program should be written in such a way that its form is a close reflection of its meaning.

Terminology

Free-form In a free form language, a line break is equivalent to a space in most contexts (excluding special places such as in comments or inside string constants). Several spaces in a row are equivalent to one space.

Layout Layout is a form of syntax where indentation indicates structure, and indentation information is significant to the structure of a program.

Syntax The syntax of a language tells which programs are well-formed, and also provides some information about the structure of programs.

5.4. Exercises

5.1. C++ was developed by augmenting C. A C comment begins with /* and ends with */. When C++ was developed, it was deemed necessary to add another kind of comment that begins with // and ends at the end of a line. Explain why the C style of comments was considered so bad that it needed to be replaced. Also explain how a compiler can warn a programmer about some mistakes using the C comment.

5.2. How could the designers of Fortran have avoided the problem shown in Figure 5.3 without changing the syntax of a do-statement?

5.3. Give arguments for and against the use of layout syntax.

5.4. Take a position either that the C choice of using / for integer division or the Ada choice of the word **div** is preferable. Offer justification for your choice.

5.5. In C, an assignment $x = y$ (copy the value of y into variable x) is an expression whose value is the value of y. Expression $x == y$ is true if x and y are equal. But in C, the integer 0 is used to indicate a false result and 1 is used to indicate true. (There is no *boolean* type.) So a condition, such as $x == y$, is considered to be an integer. Suppose that x and y are integers. Explain what happens if you write a statement that begins `if(`x` = `y`) ...`, accidentally writing = instead of ==. Can you suggest ways in which the language designers could have avoided this problem?

5.6. The syntax of C and Java each allows a statement to be just a semicolon, indicating that nothing should be done. Give an example where an extraneous semicolon (easily overlooked by a programmer) vastly changes the meaning of a C or Java program fragment. (Hint: look at a while loop.)

5.7. The body of a C while loop is one statement. Suppose a program contains

```
while(f(x) > 0)
  x = x + 1;
```

Now you decide to print the value of x each time around the loop, writing

```
while(f(x) > 0)
  x = x + 1;
  print(x);
```

Explain why this does not have the desired effect. Would layout syntax help to avoid this problem? Can you suggest a way to avoid it, or at least to reduce its likelihood, in a free-form language?

5.5. Bibliographic notes

Van der Linden [69] describes the NASA Fortran bug in more detail.

Chapter 6

Describing Syntax and Structure

6.1. Describing the syntax of a language

Even after thinking through design decisions carefully, a language designer is faced with arbitrary choices concerning which words and symbols to use, their order, and other issues of program structure. At that point, the problem changes from one of choosing a "good" syntax to one of describing those arbitrary choices to people who need to use the language, in a way that programmers will find easy to understand.

Algol 60 is an early programming language. In 1958, John Backus, one of the principal designers of Algol 60, circulated a preliminary report describing the new language. To his surprise, he discovered from responses that, not only did readers not understand the meaning of programs, but they did not even understand what programs looked like. That is, they did not understand the rules for creating a syntactically correct program.

Backus, with help from Peter Naur, solved the problem in an inspired way. The revised report included a clear and precise definition of the syntax of Algol 60, so that nobody could ever again be confused about the syntax. That worked so well that it has become standard practice in describing programming languages. The remaining sections show the methods that Backus employed for describing syntax, as well as a few extensions of those ideas.

6.2. Lexical rules

A program is just text, a sequence of characters. But the character level is too detailed for convenience. Instead, a program is broken into a sequence of basic words and symbols called *lexemes*. For example, C program fragment

```
while (turnip != 0) jump();
```

is thought of as a sequence of ten lexemes: **while**, **(**, **turnip**, **!=**, **0**, **)**, **jump**, **(**, **)** and **;**. Punctuation marks and special symbols are lexemes but spaces are not. Spaces, when they are present, just separate lexemes.

The first step in describing the syntax of a language is choice of a set of *lexical rules* that tell how a sequence of characters is broken up and converted to a sequence of lexemes. Typically, the language definition starts by saying what all of the lexemes are. For example, the list of lexemes might start with words such as **while** and **if**, as well as symbols such as **%** and **==**. The form of an *identifier*, or name, is part of this. For example, an identifier might be defined to be any sequence of one or more letters. Then a typical rule is that you start at the beginning of the program, removing one lexeme at a time. To get the next lexeme, you first skip over anything, such as spaces, that does not belong to a lexeme. Then you take the maximum number of characters that make up a valid lexeme of the language. For example, if some number of lexemes have already been obtained, and the rest of the program is " `while(...`", then the next lexeme is **while**, not, for example, **whi**, which might itself be a valid lexeme (such as a variable name). The next lexeme after that is the left parenthesis, and lexemes continue to be obtained by the same rule until the entire program has been broken into lexemes.

Comments

Most languages allow programs to contain comments, which can typically be put almost anywhere in a program, and contribute nothing to the meaning of the program, as far as the language definition is concerned. The easiest way to describe comments is to say that they are similar to spaces, and are not parts of lexemes. For example, one of the lexical rules of C is that, when you are about to get the next lexeme, you should skip over anything that starts with `/*`, up to the next occurrence of character sequence `*/`.

Tokens

The fact that the lexeme **while** is spelled w-h-i-l-e is certainly important for writing programs, but once the lexemes have been obtained, the exact spelling is less important than the fact that **while** is the word that introduces a loop. It is natural to ignore the individual characters, and to think of **while** as an indivisible component of the language, called a *token*. A token is represented by a lexeme, but is not the same thing as a lexeme; how a word is spelled is a property of the lexeme, not of the token.

We have seen lexical rules that tell how to break a program into lexemes. There are additional lexical rules that give a correspondence between lexemes and tokens. For example, a lexical rule might state that lexeme **while** stands for the token **while**, or even the token **loopbegin**. You can call the tokens anything that you like. The remaining syntactic rules are expressed entirely in terms of tokens.

It is important to use tokens instead of lexemes in syntactic rules. At a given place in a program, you might be allowed to write any integer constant. But there are infinitely many integer constants, and you cannot list them all. You do not even want to say, in syntactic rules, what an integer constant looks like, since that is really a lexical issue. Instead, you have just one token, say **number**, that has infinitely many lexemes. The lexical rules can say that lexeme **342** stands for token **number**.

From a syntactic viewpoint, you probably do not care which integer constant is written in a program. Replacing one integer constant by another in a meaningful program should yield another meaningful program. But the meaning has changed, and to account for that, a token that has more than one lexeme usually has an *attribute* that tells the corresponding lexeme. For example, lexeme **95** might correspond to token **number** with attribute "95".

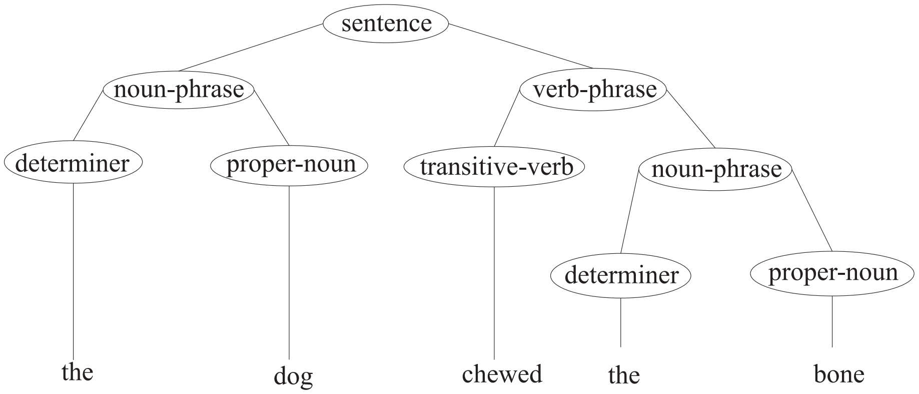


Figure 6.1: **The structure of an English sentence.** Notice that the words are leaves. Concepts such as verb phrase and noun phrase are used to label nonleaves.

A side benefit of using tokens instead of lexemes is that it is easy to modify a language by changing the words and symbols. If you want a Norwegian version of Java, you can just say that the lexeme **hvis** (Norwegian for "if") stands for the token **if**, with no need to make any change any other rules. This was more important in the past when computers were less standardized than they are today. For example, Pascal indicates that token **pointer** can be written ↑. But few computer have an up-arrow character, so lexemes ^ and **ptr** also stand for the same token.

The distinction between lexemes and tokens is an important one, but it can be awkward at times to make that distinction in practice. The most natural way of referring to the token **if** is by using its lexeme, **if**. When a token has just one lexeme, it is common practice to use the lexeme as the name for the token.

6.3. Program structure and parse trees

The structure of an English sentence can be described by a tree. For example, the structure of sentence "The dog chewed the bone" might be as shown in Figure 6.1. The structure of a program, or part of a program, can also be expressed as a tree. The internal nodes of the tree describe abstract units of a program, such as statements and expressions, and the leaves are labeled by tokens. Figure 6.2 shows a plausible structure for expression 4 + 9 * 2. You can read off the sequence of tokens that a tree describes by reading the leaves from left to right. Tree diagrams of programs are called *parse trees*, and the labels of nonleaf nodes are called *nonterminals*. Notice that nonterminals, such as expression, are not really part of the language, but are only part of the syntactic description of the language.

Parse trees actually allow some of the leaves to be labeled by nonterminals, indicating parts of the program that have no corresponding tokens. For example, Pascal allows a statement to be nothing at all, indicating that nothing should be done. Figure 6.3(a) shows a corresponding tree, which is just a leaf that is labeled **statement**. When you read off the sequence of tokens that a tree stands for, skip over leaves that do not contain tokens.

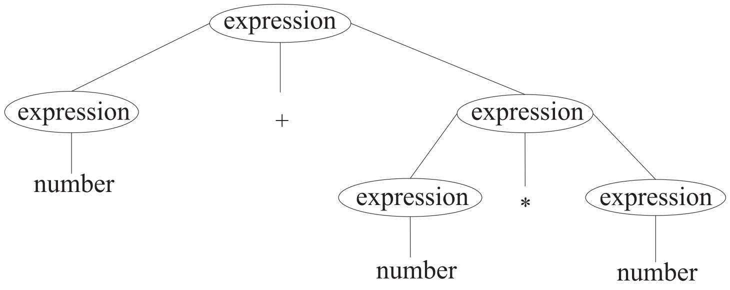


Figure 6.2: **A plausible tree structure for expression 4 + 9 * 2.** The tokens are leaves in the tree. Token **number** can have more than one lexeme, including 4, 9 and 2. This tree is only one way to describe expressions. The actual tree that you get depends on how you describe the syntax.

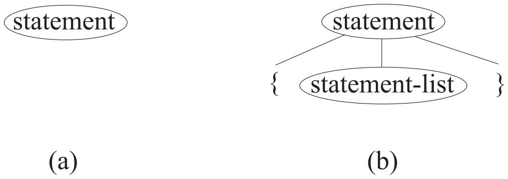
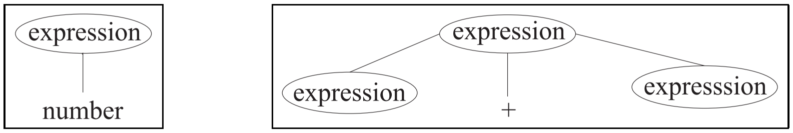


Figure 6.3: **Plausible tree structures for do-nothing statements in (a) Pascal and (b) C.** In Pascal, a statement can contain no tokens. In C, a statement-list can be empty, and a do-nothing statement can be written {}. When you read the tokens in a tree, you skip leaves that are labeled by nonterminals.



⟨expression⟩ ::= **number** ⟨expression⟩ ::= ⟨expression⟩ + ⟨expression⟩

(a) (b)

Figure 6.4: **Two productions.** Production (a) (⟨expression⟩ ::= **number**) indicates that a node labeled ⟨expression⟩ can have just one child, a leaf holding token **number**, as shown in the tree diagram above the production. Production (b) (⟨expression⟩ ::= ⟨expression⟩ + ⟨expression⟩) indicates that a node labeled ⟨expression⟩ can have three children, labeled as shown from left to right. The productions in a grammar tell all of the allowed local structures.

6.4. Using grammars to indicate allowed trees

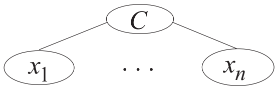
To describe the syntax of a language, you describe exactly what the parse trees for that language look like. That way, not only are you saying which sequences of tokens are allowed (since they are just the sequences of leaves in allowed parse trees), but you also indicate the structure of each program. We will use a notation called Backus-Naur Form (BNF) for describing parse trees.

Assume that the tokens of a language have already been described as part of the language's lexical description. To describe the language's syntax using BNF, you begin by defining a collection of nonterminals. When written outside of tree diagrams, nonterminals are usually written surrounded by angle brackets, to make it clear that they are nonterminals. For example, one of the basic units of a C program is an expression, so we create a nonterminal ⟨expression⟩.

The next step is to describe what a parse tree can look like *locally* at a node that is labeled by a given nonterminal. That is done by writing a collection of *productions* for each nonterminal, where each production for nonterminal C gives one form that the tree structure just beneath a node labeled by C can look like. Figure 6.4 shows two examples of productions and corresponding local tree structures. In general, a production has the form

$$\langle \mathrm{C} \rangle ::= x_1 x_2 \ldots x_n$$

where ⟨C⟩ is a nonterminal, $n \geq 0$ and $x_1, \ldots, x_n$ are each either nonterminals or tokens, indicating that a parse tree can contain local structure



It is allowed for the right-hand side of a production to have no symbols, indicating that this nonterminal can be at a leaf in the parse tree.

A language is described by a *grammar*, with two parts: (1) a collection of productions telling all possible local tree structures, and (2) a nonterminal that is required to be the label of the root of a parse tree. Typically, the root nonterminal is understood to be the one on the left-hand side of the first production. For example, the three productions

$$\begin{array}{lcl} \langle\text{expression}\rangle & ::= & \textbf{number} \\ \langle\text{expression}\rangle & ::= & \textbf{identifier} \\ \langle\text{expression}\rangle & ::= & \langle\text{expression}\rangle + \langle\text{expression}\rangle \end{array} \tag{6.4.1}$$

form a small grammar, with ⟨expression⟩ as the start nonterminal. It is convenient to write several productions for the same nonterminal together, using a vertical bar in place of ::= in all but the first. For example, productions (6.4.1) can be written more succinctly as follows.

$$\begin{array}{lcl} \langle\text{expression}\rangle & ::= & \textbf{number} \\ & \mid & \textbf{identifier} \\ & \mid & \langle\text{expression}\rangle + \langle\text{expression}\rangle \end{array} \tag{6.4.2}$$

Keep in mind that a grammar indicates *all* of the local structure forms that you can use in a tree diagram. Every node labeled by a nontermal *must* have a structure immediately below it that is allowed by a production in the grammar.

6.5. Common grammatical forms

Choice

A statement might have one of several different forms. It might, for instance, be an assignment statement, an if-statement, a while-statement, a procedure call, or any of a large collection of different kinds of things. A grammar deals with that by including a production for each form. An example of productions for statement might include the following.

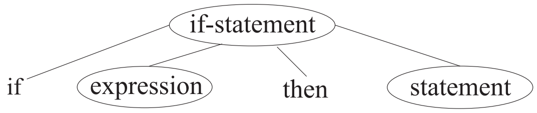
$$\begin{array}{lcl} \langle\text{statement}\rangle & ::= & \langle\text{assignment-statement}\rangle \\ & \mid & \langle\text{if-statement}\rangle \\ & \mid & \langle\text{while-statement}\rangle \\ & \mid & \langle\text{procedure-call}\rangle \end{array} \tag{6.5.1}$$

Recursion

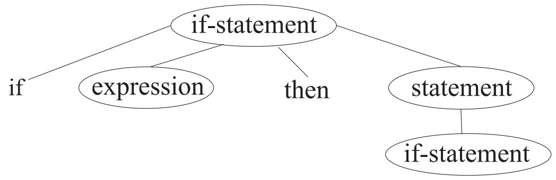
A statement often contains other statements embedded within it. For example, a Pascal if-statement has either the form **if** *expression* **then** *statement* **else** *statement* or the form **if** *expression* **then** *statement* (where there is no else-part). The following productions describe the form of a Pascal if-statement.

$$\begin{array}{lcl} \langle\text{if-statement}\rangle & ::= & \textbf{if } \langle\text{expression}\rangle \textbf{ then } \langle\text{statement}\rangle \textbf{ else } \langle\text{statement}\rangle \\ & \mid & \textbf{if } \langle\text{expression}\rangle \textbf{ then } \langle\text{statement}\rangle \end{array} \tag{6.5.2}$$

From a tree perspective, the second of those production says that a tree can contain the following local structure.



The productions clearly show that an if-statement has a smaller statement as part of it. That statement might be another if-statement, yieiding the following structure.



Recursion, where a nonterminal occurs in a subtree of another node labeled by the same nonterminal, is fundamental to writing grammars.

Repetition

Often you find yourself wanting to describe a sequence of similar things. For example, in the programming language Lisp, one form of expression consists of a sequence of zero or more Lisp expressions, written one after another, enclosed in parentheses. So one production for Lisp expressions should be

$$\langle\text{lisp-expression}\rangle ::= \ \textbf{(}\ \ \langle\text{lisp-expression-list}\rangle\ \ \textbf{)} \tag{6.5.3}$$

(Note the parentheses in the production. They are tokens. There are three things on the right-hand side.) Nonterminal ⟨lisp-expression-list⟩ describes sequences of expressions. Each sequence can either be the empty sequence, containing no expressions, or consists of an expression followed by (zero or) more expressions. So productions for ⟨lisp-expression-list⟩ are as follows.

$$\begin{aligned} \langle\text{lisp-expression-list}\rangle \quad &::= \\ &\ \ \mid \quad \langle\text{lisp-expression}\rangle\ \langle\text{lisp-expression-list}\rangle \end{aligned} \tag{6.5.4}$$

The right-hand side of the first production is empty.

A procedure call in a particular language might allow a list of expressions, separated by commas, as a parameter list. A nonempty parameter list is either a single expression or a single expression followed by a comma followed by more expressions. A grammar for ⟨procedure-call⟩ and ⟨expression-list⟩ goes as follows.

$$\begin{aligned} \langle\text{procedure-call}\rangle \quad &::= \quad \textbf{identifier}\ \ \textbf{(}\ \ \langle\text{expression-list}\rangle\ \ \textbf{)} \\ \langle\text{expression-list}\rangle \quad &::= \quad \langle\text{expression}\rangle \\ &\ \ \mid \quad \langle\text{expression}\rangle\ \ \textbf{,}\ \ \langle\text{expression-list}\rangle \end{aligned}$$

6.6. Ambiguity

The following simple grammar describes expressions with operators +, − and * as well as parentheses.

$$\begin{aligned} \langle\text{expression}\rangle \quad &::= \quad \textbf{number} \\ &\ \ \mid \quad \langle\text{expression}\rangle + \langle\text{expression}\rangle \\ &\ \ \mid \quad \langle\text{expression}\rangle - \langle\text{expression}\rangle \\ &\ \ \mid \quad \langle\text{expression}\rangle * \langle\text{expression}\rangle \\ &\ \ \mid \quad \textbf{(}\ \langle\text{expression}\rangle\ \textbf{)} \end{aligned} \tag{6.6.1}$$

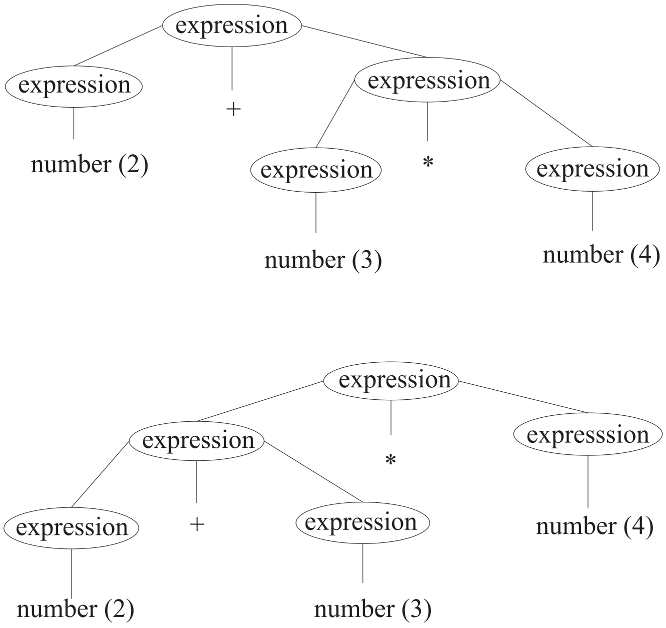


Figure 6.5: **Two parse trees for $2 + 3 * 4$ according to Grammar (6.6.1).** Since there are two different parse trees for the same sequence of tokens, Grammar (6.6.1) is ambiguous.

The syntax of a language is supposed to indicate something about the structure of programs, and, although the syntax does not tell us that + means addition and * means multiplication, it should tell us whether the structure of expression $2{+}3{*}4$ is more like $2{+}(3{*}4)$ or $(2{+}3){*}4$. Unfortunately, Grammar 6.6.1 does not do so. Figure 6.5 shows two different parse trees for expression $2 + 3 * 4$, both allowed by the grammar, but showing different structure. Because of those two parse trees for the same expression, Grammar 6.6.1 is an *ambiguous* grammar.

Definition. A grammar G is ***ambiguous*** if it is possible to produce two different parse trees, both using the rules of G, that have exactly the same sequence of tokens in their leaves.

Fortunately, it is possible to create an unambiguous grammar for simple expressions. A way to overcome ambiguity is to add more nonterminals to the grammar, each representing a level of precedence. It is also important to avoid symmetries that lead to ambiguity. Grammar 6.6.2 does both.

$$\begin{array}{rcl}
\langle\text{expression}\rangle & ::= & \langle\text{term}\rangle \\
 & | & \langle\text{expression}\rangle\ \ \mathbf{+}\ \ \langle\text{term}\rangle \\
 & | & \langle\text{expression}\rangle\ \ \mathbf{-}\ \ \langle\text{term}\rangle \\
\langle\text{term}\rangle & ::= & \langle\text{factor}\rangle \\
 & | & \langle\text{term}\rangle\ \ \mathbf{*}\ \ \langle\text{factor}\rangle \\
\langle\text{factor}\rangle & ::= & \textbf{number} \\
 & | & \mathbf{(}\ \ \langle\text{expression}\rangle\ \ \mathbf{)}
\end{array} \tag{6.6.2}$$

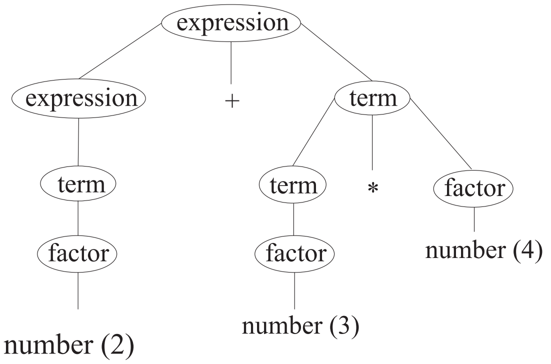


Figure 6.6: The unique parse tree for expression 2+3∗4 **according to Grammar (6.6.2).**

The idea is that ⟨expression⟩ can produce any kind of expression, ⟨term⟩ only allows operator + to occur inside parentheses (but allows * to occur outside parentheses), and ⟨factor⟩ does not allow any operators to occur outside of parentheses. Figure 6.6 shows a parse tree for $2+3*4$ using Grammar (6.6.2). Try to convince yourself that Grammar 6.6.2 describes the same expressions as Grammar 6.6.1, but that it is unambiguous. Why aren't there other ways to derive $2+3*4$?

Associativity

Operator precedence tells how expressions involving operators of different precedence levels are parsed. But that leaves the issue of how to parse expressions involving more than one occurrence of the same operator, or of operators with the same precedence. For example, should $5-2-2$ be thought of as similar to $5-(2-2)$ or to $(5-2)-2$? That is where rules of *associativity* come into play. An operator is left-associative if several occurrences of it are done from left to right, and it is right-associative if operations are done from right to left. If subtraction is left-associative, then $5-2-2$ is understood to have a structure similar to $(5-2)-2$.

Grammar 6.6.2 already says that operators +, − and * are left-associative. Look at the following two productions. The first is taken from Grammar (6.6.2), and the second is a modification of that production.

$$\begin{array}{lcl}\langle\text{expression}\rangle & ::= & \langle\text{expression}\rangle + \langle\text{term}\rangle\\ \langle\text{expression}\rangle & ::= & \langle\text{term}\rangle + \langle\text{expression}\rangle\end{array}$$

According to the first of those two productions, expression $2+3+4$ should be taken to be an expression followed by + followed by a term. Clearly, the expression is $2+3$ and the term is 4 (which is just a factor), since a term does not allow + to occur outside parentheses. So that production indicates that + is left-associative. The second production, on the other hand, indicates that $2+3+4$ should be a term followed by + followed by an expression. In that case, the term must be 2 and the expression $3+4$, so that addition is right-associative. You should include only one of those two productions in a grammar.

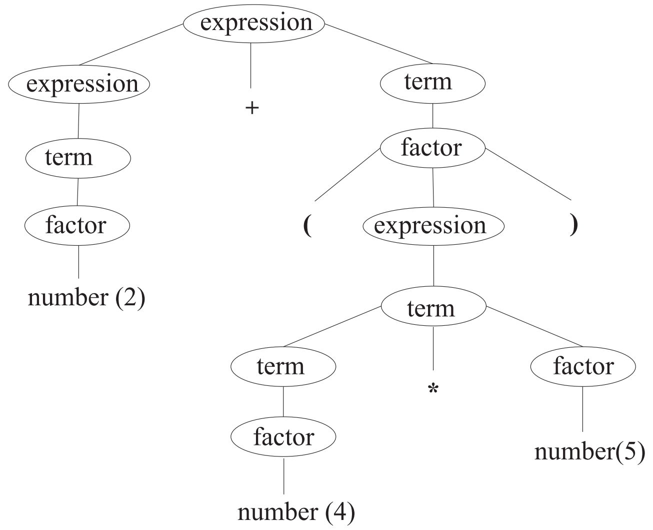


Figure 6.7: **A parse tree for expression 2 + (4 * 5), using Grammar (6.6.2).**

6.7. Syntax is not meaning

When describing a language using BNF, be careful to remember that you are concerned with the syntax of the language, not so much with meaning. Grammar (6.6.2) allows expressions that include parentheses. If you think about what the expression means, the parentheses do not indicate that any operations are to be performed. But from a syntactic point of view, parentheses are certainly tokens, and are part of any program that uses them. Figure 6.7 shows a parse tree for $2 + (4 * 5)$ using Grammar (6.6.2). The parentheses are leaves, since they are tokens that belong to the expression.

Does it matter whether addition is performed from left to right or from right to left? Since addition is an associative operator, you might argue that it does not. But from a syntactic perspective, it definitely does matter. In order to be unambiguous, a grammar needs to ensure that no sequence of tokens can be parsed two different ways, even if those two ways would have the same meaning.

6.8. Extended BNF notation

Some simple extensions make BNF notation a little more convenient to use. To make it clear that we are not using ordinary BNF, any grammar written with the extensions is said to be an Extended BNF (EBNF) grammar.

Placing a sequence of tokens and nonterminals in square brackets indicates that they are optional. For example, production ⟨expression⟩ ::= ⟨term⟩ [+ ⟨expression⟩] is a shorthand for the two productions

$$\begin{array}{rcl}\langle\text{expression}\rangle & ::= & \langle\text{term}\rangle \\ & | & \langle\text{term}\rangle + \langle\text{expression}\rangle\end{array}$$

Placing a sequence of tokens and nonterminals in curly braces indicates zero or more repetitions of that sequence. For example, production $\langle A\rangle$ ::= $\langle B\rangle$ $\{\langle C\rangle\}$ indicates that $\langle A\rangle$ can be $\langle B\rangle$ or $\langle B\rangle\langle C\rangle$ or $\langle B\rangle\langle C\rangle\langle C\rangle$, etc., with as many copies of $\langle C\rangle$ as desired.

You never really need brackets or braces. They are only conveniences. For example, you can derive the same sequences of tokens as are described by production $\langle A\rangle$ ::= $\langle B\rangle$ $\{\langle C\rangle\}$ from the following productions.

$$\begin{array}{rcl}\langle A\rangle & ::= & \langle B\rangle\langle \text{rest}\rangle \\ \langle \text{rest}\rangle & ::= & \\ & | & \langle C\rangle\langle \text{rest}\rangle\end{array}$$

Examples using EBNF

Grammar (6.6.2) can be written as follows using EBNF.

$$\begin{array}{rcl}\langle \text{expression}\rangle & ::= & [\langle \text{expression}\rangle\ \ \mathbf{+}\ ]\ \ \langle \text{term}\rangle \\ & ::= & [\langle \text{expression}\rangle\ \ -\ ]\ \ \langle \text{term}\rangle \\ \langle \text{term}\rangle & ::= & [\langle \text{term}\rangle\ \ \mathbf{*}\ ]\ \ \langle \text{factor}\rangle \\ \langle \text{factor}\rangle & ::= & \textbf{number} \\ & | & \textbf{(}\ \ \langle \text{expression}\rangle\ \ \textbf{)}\end{array}$$

In some places, repetition can be used in place of recursion. For example, the rule for $\langle\text{term}\rangle$ can be replaced by the following extended production, but doing so no longer allows you to express associativity, so you need to add, on the side, that * is left-associative.

$$\langle \text{term}\rangle \quad ::= \quad \langle \text{factor}\rangle \quad \{\mathbf{*}\ \ \langle \text{factor}\rangle\}$$

As shown earlier, you can use two rules to show the possible forms of a Pascal if-statement. With EBNF, you can do the same in a single production.

$$\langle \text{if-statement}\rangle ::= \textbf{if}\ \langle \text{expression}\rangle\ \textbf{then}\ \langle \text{statement}\rangle\ [\textbf{else}\ \langle \text{statement}\rangle]$$

A Pascal compound statement starts with the token **begin** and ends with the token **end**. In between them is a sequence of statements, separated by semicolons. The following EBNF production insists that there must be one or more statements inside the compound statement.

$$\langle \text{compound-statement}\rangle ::= \textbf{begin}\ \langle \text{statement}\rangle\ \{;\ \langle \text{statement}\rangle\}\ \textbf{end}$$

In reality, a Pascal compound statement can contain zero or more statements, separated by semicolons. That is, a compound statement can be just **begin end**. That can be allowed by making the part between **begin** and **end** optional.

$$\langle \text{compound-statement}\rangle ::= \textbf{begin}\ [\langle \text{statement}\rangle\ \{;\ \langle \text{statement}\rangle\}]\ \textbf{end}$$

6.9. Syntax diagrams

Syntax diagrams are a graphical form of grammar. Figure 6.8 shows a diagram of simple expressions. The nonterminals are shown in rectangles, and the tokens are shown in ovals or circles.

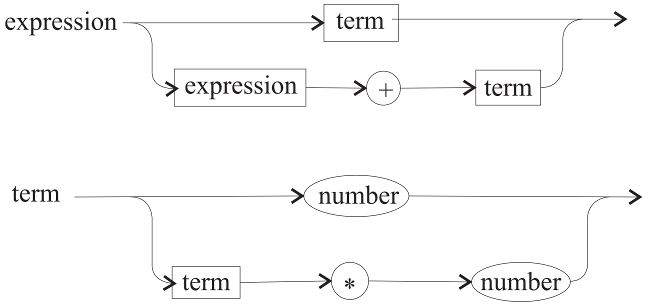


Figure 6.8: **A pair of syntax diagram that describe expressions and terms, with operators + and *.** Nonterminals are shown in rectangular boxes, and tokens are shown in circular or oval boxes. In this grammar, a term directly produces a number; there is no factor nonterminal.

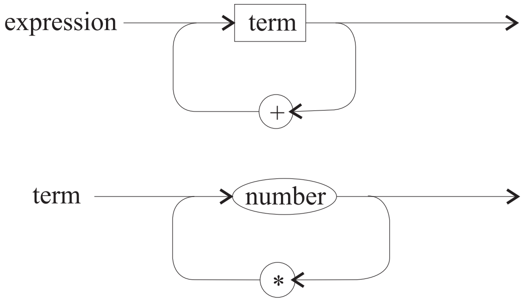


Figure 6.9: **Another pair of syntax diagrams that describe expressions and terms.** In these diagrams recursion has been replaced by cycles in the diagrams.

The idea is that, to derive an expression, you follow the arrows in the diagram for expression, writing things encountered along the way. At a token node, you write the token. At a nonterminal node, you must write any sequence of tokens that can be derived according to the diagram for that nonterminal. When there is a fork in the arrows, you have a choice of either branch. To complete an expression, you must reach the end of the diagram for expressions (the arrow at the right side).

One advantage of syntax diagrams is that it is possible to show loops. Figure 6.9 shows an alternative diagram for expressions. Their graphical nature makes syntax diagrams a little easier to read than BNF notation, so they are a popular way of describing syntax. Unfortunately, syntax diagrams with loops suffer from the same problem as EBNF that associativity information is not present in them, and needs to be indicated by remarks that are not part of the grammar itself.

6.10. Summary of terminology and concepts

Language syntax always incorporates arbitrary choices that need to be conveyed to programmers. The description of syntax is broken into two parts: lexical rules, where the words and symbols of the language are described; and syntactic rules, which describe the allowed sequences of words and symbols, and some aspects of their structure. Grammars and syntax diagrams can clearly and unambiguously define the syntax of programming languages and have become standard components in programming language definitions.

Terminology

[...] Square brackets are used in extended BNF to indicate an optional part of a production.

{...} Curly braces are used in extended BNF to indicate a part of a production that can be repeated zero or more times.

Ambiguous grammar A grammar that allows one sequence of tokens to have two different parse trees is ambiguous.

Associativity Associativity rules tell whether two or more operators of the same precedence are done from left to right or right to left. If + is left-associative, then additions are done from left to right.

BNF BNF stands for Backus-Naur Form. It is a notation for writing grammars.

Comment A comment is a part of a program that is skipped over while breaking a program into lexemes. It is not part of any lexeme.

Extended BNF Extended BNF is an extension to BNF that allows notation [...] and {...}.

Identifier An identifier is a name that you might give to something in a program.

Lexeme A lexeme is a sequence of characters that corresponds to a token.

Lexical rules Lexical rules give the correspondence between lexemes and tokens, and tell how a program (a sequence of characters) is broken up into a sequence of lexemes.

Nonterminal A nonterminal is a symbol that occurs on the left-hand side of a production in a grammar.

Parse tree A parse tree shows the structure that a given grammar attaches to a program or program fragment.

Precedence Precedence rules determine how expressions involving binary operators are parsed. Operators with higher precedence are done before operators with lower precedence. Precedence can be extended to other constructs, such as unary operators and function calls. For example, a typical precedence rule says that, in expression $-x + y$, the unary $-$ operator has higher precedence than the binary + operator.

Production A production consists of a single nonterminal on the left and a sequence of tokens and nonterminals on the right. It indicates an allowed local structure in a parse tree.

Token A token is a basic lexical unit, such as a word, a special symbol, a name or a number. A token can stand for more than one different lexeme.

6.11. Exercises

6.1. What is the difference between a lexeme and a token? Why is it useful to make that distinction?

6.2. Does every token have exactly one corresponding lexeme? Explain.

6.3. Should a string constant such as "the daisy" be considered a single lexeme, even though it contains a space?

6.4. What is the typical rule that is used to break a sequence of characters up into a sequence of lexemes? How do you know where each lexeme begins and ends?

6.5. If a character is not part of a lexeme, can it always be deleted without changing the meaning of a program?

6.6. Using Grammar (6.6.2) draw a parse tree, starting at ⟨expression⟩, for each of the following expressions.

(a) $3 + 4 + 8$

(b) $((5))$

(c) $18 * 2 + 99$

(d) $18 + 2 * 99$

(e) $(4) + 5 * (6 + 2)$

6.7. According to Grammar (6.6.2), is multiplication left-associative or right-associative?

6.8. Show a parse tree for string *aacacab* according to the following grammar, where the start symbol is ⟨S⟩, and the tokens are *a*, *b* and *c*.

$$\begin{array}{rcl} \langle S\rangle & ::= & \langle F\rangle\ a\ \langle S\rangle \\ & | & b \\ \langle F\rangle & ::= & a\ \langle F\rangle \\ & | & c \end{array}$$

6.9. Combine Grammars (6.5.3) and (6.5.4), and add one more production,

$$\langle\text{lisp-expression}\rangle\ ::=\ \text{x}$$

where x is a token. Using that grammar, show a parse tree for each of the following, starting with ⟨lisp-expression⟩.

(a) ( )

(b) (x x)

(c) ((x) x)

(d) ((x x x))

6.10. Show that Grammar (6.5.2) is ambiguous. Assume that there is a simple kind of statement s and a simple kind of expression e, and do not worry about the details of those.

6.11. Show that the following grammar is ambiguous. The start symbol is ⟨S⟩, and a is a token.

$$\begin{array}{lcl} \langle S\rangle & ::= & \langle S\rangle\ a \\ & | & a\ \langle S\rangle \\ & | & a \end{array}$$

6.12. Explain why Grammar (6.6.2) is unambiguous. Do not merely repeat what it means for it to be unambiguous, but argue that it meets the requirements of an unambiguous grammar.

6.13. Lexical rules need to specify collections of lexemes that correspond to a particular token. Sometimes grammars are used for that purpose. A Pascal identifier consists of a letter followed by zero or more letters and digits. Assume that nonterminals ⟨letter⟩ (a single letter) and ⟨digit⟩ (a single digit) are available. Give a BNF grammar for ⟨identifier⟩, describing Pascal identifiers. Do not use extensions.

6.14. Show how EBNF can be used to simplify the grammar from Exercise 6.13.

6.15. Write BNF productions for ⟨while-statement⟩ in a grammar for C or Java.

6.16. Write a BNF grammar that derives all (and only) sequences of the symbols `(`, `)`, `[` and `]` that are well-nested. In a well-nested sequence, parentheses must be balanced and brackets must be balanced, and balanced parts cannot partially overlap. For example, `[]([][()])(())` is a well-nested sequence, but `([)]` and `))((` are not well-nested.

6.17. Write a BNF grammar that derives all (and only) sequences of the letters a, b and c where all of the a's precede all of the c's.

6.18. Write a BNF grammar that derives all (and only) sequences of the letters a and b that have the same number of a's as b's. (This one will take some thought.)

6.19. Draw a syntax diagram or diagrams that describe Pascal identifiers. See Exercise 6.13.

6.20. Draw a syntax diagram that describes a Pascal compound statement. You can use nonterminal ⟨statement⟩.

6.21. Draw a syntax diagram that describes a Pascal if-statement. You can use nonterminal ⟨statement⟩.

6.22. Draw a syntax diagram that describes a C or Java if-statement. Allow a version with an else part, and also a version without an else part.

6.23. A while loop involves repetition. Does that mean you need to show a loop in a syntax diagram for a while statement? Draw a syntax diagram for a C or Java while-loop.

6.12. Bibliographic notes

Algol 58 is described by Backus [10], and BNF is introduced in the Algol 60 report [81]. The Pascal report [62] uses syntax diagrams to describe the syntax of Pascal. Chomsky independently discovered grammars, introducing them in [27] and [28]. Grammars and lexical rules are both used heavily in building compilers, and books on compilers ([5], [49]) provide extensive material on both.

Chapter 7

Semantics

7.1. Introduction to semantics

The semantics of a program tells you exactly what that program does for every possible input. The semantics of a programming language tells you what every syntactically well-formed program does in every possible situation in which it might run. That involves describing every construction and every basic word or action in the language. Semantics is a major part of a programming language description, and a substantial part of what you read in this book concerns it. This chapter only covers broad notions in semantics, with later chapters providing more detail.

7.2. Operational semantics

To describe what an imperative program does when it runs, you explain the sequence of changes that it makes. A program starts in a particular *state*, indicating the initial values of variables, and what is in the input. Each basic step of the program produces another state. For example, suppose that the program of Figure 2.1 starts in state $\{a = 1, b = -4, c = 3\}$, indicating that variable a holds 1, etc. After the first line, it moves to state $\{a = 1, b = -4, c = 3, s = 16\}$. The overall meaning of the program, or its *operational semantics*, is the sequence of states that it goes through. Chapter 46 looks at methods for carefully describing operational semantics.

7.3. Denotational semantics

Instead of saying what an expression *does*, the semantics of an expression can rely on saying what it computes, or what it *denotes*. For example, expression $3 + 4$ denotes 7 since that is the value that it has when it is computed. A *denotational semantics* of a declarative programming language tells what every syntactically correct expression denotes, and suppresses details about how the result is computed.

There is still a notion of state, giving values of variables. For example, in order to obtain the value of expression $y + 2$, you need to know the value of y. But, in a denotational semantics, there is no notion of one state coming after another in the computation. A state just provides static information about variable values. The denotational semantics of an expression is given by a function that takes a state and produces the value of the expression in that state. Chapter 49 discusses denotional semantics.

7.4. Axiomatic semantics

Operational semantics tells what a program does and denotational semantics tells the result that it computes. But there are other approaches to understanding programs. Imagine that you have a programming problem, and that somebody has written a very clever and fast program that solves exactly that problem. That would be great — all you have to do is use that program off-the-shelf. But now imagine that the clever program works, but, unfortunately, neither you nor the author has any idea *why* it works, and you cannot demonstrate to yourself that it should work. You would have to be very reluctant to use it, since you cannot be sure about it. You cannot test it on every possible input. What if it is wrong?

The point is that, for a programmer, what a program does is sometimes less important than what the programmer *knows* about what the program does. If the program has some property that the programmer is unaware of, then the fact that is possesses that property is almost completely useless. To drive the point home, imagine that you need to make some modifications to a program. You will not be able to do it unless you understand how and why the program works.

There is a form of semantics, called *axiomatic semantics*, that concentrates not on what programs do or compute, but on what a programmer can know about what a program does or computes. An axiomatic semantics of a programming language provides a collection of rules for proving that given programs have given characteristics. Axiomatic semantics suppresses all aspects about what a program does or computes, instead concentrating on properties of programs and proofs of those properties. Chapter 47 gives an example of an axiomatic semantics.

7.5. Partial semantics

Axiomatic semantics provides only partial information about what a program does or computes; it tells only what can be proved about a program's behavior. Although denotational and operational semantics usually tell *exactly* what a program computes or does, programming language designers sometimes prefer to provide only partial information for those kinds of semantics as well, in order to give language implementors some flexibility. A common example is computation of pairs, triples, etc. If A and B are expressions then the ordered pair (A, B) can be computed in two obvious ways. The implementation might

1. Compute the value a of expression A.
2. Compute the value b of expression B.
3. Build the ordered pair (a, b).

or it might instead

1. Compute the value b of expression B.
2. Compute the value a of expression A.
3. Build the ordered pair (a, b).

For most programs it does not matter which order is chosen, but sometimes the order influences how efficiently the computation can be done, so language designers prefer to leave the choice of order up to the implementor. The language definition simply does not say which order is used.

7.6. Relational semantics

In some languages an expression can produce more than one result. That is not the same as leaving the result up to the language implementor, nor the same as producing an ordered pair as its result; instead, you say that an expression might produce 1, and might produce 2. The language implementation explores *all* of the potential results, forking the computation into separate branches, with each branch exploring a different option. That is typical of the way logic programming languages work.

A variation on denotational semantics can be used for those languages. Instead of being a function from states to values, the semantics or an expression is a *relation* between a state and a value. A relation is a set of pairs. Imagine an "expression" $\mathrm{betw}(x, y)$ that produces any integer (and all integers) between x and y, including x and y. Let s be the state $\{x = 4,\ y = 6\}$. Then the relation that is the semantics of $\mathrm{betw}(x, y)$ contains $\{(s, 4)$, $(s, 5)$, $(s, 6)\}$ as well as more pairs for other states. That is, if $\mathrm{betw}(x,\ y)$ is evaluated in state s, its value might be 4, might be 5 and might be 6. They must all be tried. Chapter 26 examines the idea of allowing an expression to produce more than one result, and Part IX explores those ideas in the context of logic programming.

7.7. Semantic black holes and infinite loops

In most programming languages it is possible to write a program, or part of a program, that never stops, but keeps computing forever without producing any result. For an operational semantics, that means the program goes through an infinitely long sequence of states.

A denotational semantics has a more difficult problem. Recall that a denotational semantics is supposed to tell the value of every expression in every state. But an expression simply does not have a value when its computation never stops. So there does not appear to be any answer that a denotational semantics can attach to some expressions. The solution is to attach a special *infinite loop* meaning to an expression whose evaluation does not stop.

An infinite loop affects not only one statement or expression, but everything after it, since any action that should have been done after the nonterminating computation will never be performed. The program appears to enter a black hole from which it can never emerge. That significantly complicates the semantics, since every semantic rule must take infinite loops into account. Even the rule for adding two numbers needs to say what the answer is when one of the operands is an infinite loop. (The sum is also an infinite loop.) Some languages, such as Charity, do not allow a programmer to write a program that never terminates. The semantics of Charity is significantly simplified by that.[1]

[1] Unfortunately, there is a curious and subtle result of computability theory: For every (implementable) programming language L where all programs must terminate, there exists a problem that can be solved in a general programming language, by a program that always terminates, but that cannot be solved by any program written in language L. So Charity cannot solve all problems that general purpose languages can, even when you require programs in the general language to terminate on all inputs. Somehow, the ability to loop forever is critical, even when you do not use it!

There is another kind of black hole that does not involve an infinite loop, and that is often more difficult to deal with for a programmer. A few programming languages allow well-formed programs whose semantics are so difficult to understand that the language designers are not able to give any meaning to them at all. Once such a computation is performed, not only is the result of that part of the program unspecified, but everything that the program does from then on is undefined. The program might stop there, but it might continue to run past the offending part, performing more of the program, but with the entire semantics of the programming language invalidated. For example, the semantics of C indicates that, if x is a real number and you perform statement `x = 3.5` then afterwards, variable x will have value 3.5. But if you write

```
a[n] = 0;
x = 3.5;
```

and n is not in the range of indices supported by array a, then there is no guarantee that x will hold 3.5 afterwards. It is possible that the program stores the constant 3.5 into memory somewhere, and performs statement $x = 3.5$ by copying from that memory location into variable x. C programs do not check array indices; $a[n]$ is just the memory location n spots from the beginning of array a in memory, and if it happens that $a[n]$ refers to the memory location where 3.5 is stored, then that value might be overwritten.

Obviously, you want to avoid semantic black holes. With the exception of infinite loops, most language definitions have no black holes at all. C and C++ are examples of languages whose semantics are replete with them. The committee that wrote the definition of the ANSI C++ standard seemed to like them so much that it decided to add some new semantic black holes to the language. Expression `++x` has a side-effect of adding one to variable x, and has the new value of x as its value. Statement `x = ++x` appears to add 1 to x and then to store the result into x again, causing a redundant operation to be performed. But according to the ANSI C++ standard, performing statement `x = ++x` causes a program to enter a semantic black hole! Almost anything could happen.

7.8. Resource limitations and semantics

Every computer has resource limitations. A memory limitation might cause a computation to fail when it tries to allocate memory, even though the language semantics says that the computation should succeed. Generally, resource limitations are not explicitly given as part of the semantics of a programming language since the available resources vary from one computer to another. The semantics of Cinnameg, for example, allows you to compute with arbitrarily large integers, and the language semantics does not put a limitation, such as 1,000,000 digits, on the size of integers. Any particular machine will necessarily impose some limits. When you encounter an error during computation caused by a memory limitation, the language implementation is not really considered to be broken, since there is nothing that it can do about a lack of memory.

7.9. Summary of terminology and concepts

The semantics of a language, which is concerned with what programs mean, is the bulk of a language description, and there are several ways to explain it. Operational semantics tells what a program *does* when it runs, by giving a sequence of states through which the program takes a computer. Denotational semantics concentrates instead on what a program *computes*, ignoring the intermediate steps, and giving only the answer. Axiomatic semantics is concerned with what a programmer can *prove* about what a program does or computes. Relational semantics allows an expression or program fragment to produce more than one answer, or to fork into more than one branch.

Terminology

Axiomatic semantics An axiomatic semantics gives rules that can be used to prove properties of programs.

Denotational semantics A denotational semantics gives the result that each expression in a language computes.

Operational semantics An operational semantics describes the steps that a program makes while it runs.

Relational semantics A relational semantics attaches a relation to each expression or other fragment of a program. The relation gives all possible behaviors of the fragment. When run, the program must be prepared to try all of its possible behaviors, not just one of them.

Semantics The semantics of a programming language tells something about what each well-formed program does or computes. It is the meaning of a program.

Semantic black hole Semantic black holes occur where a program performs an operation or sequence of operations that make it impossible to continue, either at all or in a sane way, with the rest of the computation.

State A state tells values of all variables in a program at a given point during the program's execution.

7.10. Exercises

7.1. If you are interested in how much time a particular computation takes, which kind of semantics would you find most useful, an operational semantics, a denotational semantics or an axiomatic semantics?

7.2. What distinguishes an axiomatic semantics from other kinds of semantics?

7.3. What distinguishes an operational semantics from a denotational semantics?

7.4. Some implementations of Fortran allow a program to change the value of a constant. For example, you can change constant 2.0 to be 3.0, causing statement `X = 2.0` to set variable X to 3.0. (Constants are stored in the memory, and you are allowed to change the memory that holds a constant.) How would this affect the definition of the semantics of Fortran? Would you call Fortran a well-principled language?

7.5. Running out of memory is usually fatal to a program. Should running out of memory be considered a semantic black hole?

7.11. Bibliographic notes

Winskel [110] and Pagan [84] cover different kinds of semantics of programming languages.

Part III

Managing Information

Chapter 8

Data and Data Representation

8.1. Programs and data

A program works by manipulating information. It reads information, stores it in memory, and produces information as its answers. Information, or *data*, comes in various forms; characters, integers, strings, lists and trees are just a few. Anything that might be stored in a variable, passed to a subprogram, returned from a subprogram, or stored in the computer's memory qualifies as a form of data.

Data can be viewed at two levels of detail. There is a conceptual level, where you think of the data in your own terms. For example, you understand integers quite well, since you have used them since childhood. There is also a more concrete view, where you see the data in terms of how it is stored in a computer's memory. The integer 2 might be represented by a sequence of bits, such as 00000010. The conceptual view is the *semantics* of the data, while the representation is a way of achieving the semantics. This chapter explores both views of data.

8.2. Simple values

When a novice programmer is first introduced to a programming language, usually the first kinds of data that he or she sees are numbers. Numbers are well understood from the early years of school, so they provide a familiar setting in which to study the concepts of computation. Fortunately, numbers are also among the most useful of data forms, and provide some excellent examples of computational problems. Numbers are examples of *simple values*, which are any values that are *atomic* because they have no discernible internal structure, and are not intended to be taken apart into smaller pieces. From the point of view of a programming language, a number is (usually) atomic.

Other simple values include truth values (true and false), also called *boolean* values, used as the results of tests; and characters, used to store textual information. Some languages provide the concept of a *symbol*, which is a thing that is described entirely by its name. For example, **robot** and **vegetable** are symbols. A symbol has no structure — it is just a thing. It is left up to the programmer to interpret what that thing means to him or her.

Simple values are typically stored in fixed size chunks of memory. For example, a character is stored as a pattern of bits.[1] For a truth value (true or false) you could make use of a single bit, but you typically use one byte simply because smaller pieces of memory are difficult to use in most computers. A symbol is usually represented as the address of a piece of memory where the name and possibly other information about the symbol are stored.

Integers are also typically stored in fixed size chunks of memory. On a 32-bit computer, an integer is normally stored as a 32-bit binary number in a single four-byte word, which is convenient since the processor handles instructions that manipulate single-word integers. But using 32 bits has the disadvantage that the largest integer that can be stored is $2^{31} - 1$ (about two billion). Since the available representations encode both positive and negative integers, adding 1 to the largest positive integer yields a negative integer.

How a programming language implementation chooses to store integers is a reflection of what the language is trying to provide to the programmer. The language C, for example, attempts to provide a convenient but relatively direct view of the computer hardware. Integers are stored in one word, and when you see two integers added together in a C program, you understand that the addition is to be done by a single machine ADD instruction. But some languages want to provide mathematically correct integers, where adding 1 to a positive integer always yields a larger positive integer. In that case, an integer needs to be stored in several bytes, and arithmetic operations are carried out by software rather than hardware, at a cost in efficiency. Mathematica , which is designed for general mathematical calculations and reasoning, is an example of a programming language that offers this more conceptual, and mathematically correct, view of integers.

Real numbers, such as 2.4, are typically stored in 64 bit words, as binary numbers with limited precision, since processors can perform arithmetic on such numbers. Each real number is stored in *floating point* representation, as a pair of integers called the *mantissa* and the *exponent*, where a number with mantissa 110111001110_2 and exponent e represents the real number $0.110111001110_2 \times 2^e$. Of course, for 64 bits numbers, the mantissa would be longer. The IEEE 64-bit standard uses 11 bits for the exponent and 53 bits for the mantissa (including a sign bit in each). That allows positive numbers as small as 10^{-323} and as large as 10^{308}, stored with about 15 decimal digits of precision, as well as the corresponding negative numbers. A number such as $1/3$ cannot be written exactly in binary notation, so it is stored as an approximation. Even the number 0.1, which can be written exactly in decimal notation, can only be written approximately in binary notation. (It is 0.000110011001100. . . in base 2.)

Since Mathematica wants to present a mathematical view of numbers, it stores real numbers with arbitrary precision, using as many bits as you ask for. In fact, Mathematica can even use a symbolic notation to represent irrational numbers such as π exactly, by simply calling numbers by their names, or by saying how they would be computed to arbitrarily high precision.

8.3. Tuples and records

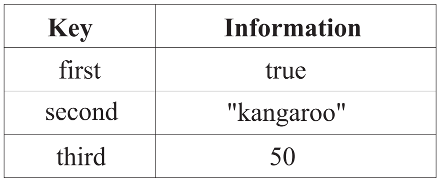
A value that has visible internal structure is called a *complex value*. One kind of complex values is an ordered pair, which should be familiar from mathematics as a point in the Cartesian plane. For example, the point with x-coordinate 3 and y-coordinate 5 is written

[1] ASCII uses one byte (7 or 8 bits), Unicode uses two bytes, and extensions of Unicode use up to 4 bytes per character.

$(3, 5)$, with a left-hand part (3) and a right-hand part (5). The concept of an ordered pair can be extended to an ordered *tuple*, consisting of two or more values. For example, $(3, 5, 7)$ is a tuple of three values. One of the most common uses of tuples is to pass information to subprograms; a subprogram that takes three parameters can be viewed as taking a tuple of three things.

A key feature of a tuple is that it has a known size. If you create a function that takes four parameters, you know, while writing the function definition, that there must be exactly four parameters, so the parameter is a tuple of size four. When a program asks for a part of a tuple, it explicitly specifies which one it wants, such as the third one. It never asks for the k-th component, where k is a variable. A consequence of that for a typed language, where the compiler performs type checking, is that the components of a tuple do not all need to have the same type. If you know that ordered pair (x, y) consists of an integer x and a boolean value y, then you can determine the type of each separate component. A compiler would not be able to determine the type of the k-th component if it does not know, when compiling the program, what k is.

A *record* is a kind of tuple where the parts are identified by names instead of by positions. For example, tuple (true, “kangaroo”, 50), whose components are identified by positions 1, 2 and 3, is equivalent to the record



Key	Information
first	true
second	"kangaroo"
third	50

whose components are called first, second and third. Of course, you would normally choose much more descriptive names. Programmers have a much easier time remembering the names of components (such as **social** for a social security number) than remembering to look in the third position of a tuple. A compiler treats the two in the same way, remembering the position corresponding to each name. Ada and C are examples of languages that provide records; Ada calls them records, while C calls them structures.

Representing tuples

An efficient way to store a tuple is to put the parts in consecutive memory locations. An ordered pair consisting of a four-byte value and an eight-byte value would occupy 12 consecutive bytes. Since the members of the tuple are stored sequentially in memory, this is called *sequential representation* of the tuple.

8.4. Lists and arrays

A tuple is a kind of sequence of a fixed size. A *list* (sometimes called an *array*) is another kind of sequence whose size, usually called its length, can only be determined by examining it; a variable that holds a list could hold a list of any size. A typical operation is to select the k-th member of a list, where k is a variable or other expression. In a typed language, where the compiler needs to know the type of each thing, all members of a list must have the same type, since otherwise it is impossible to know the type of the k-th member. To avoid confusion between the two closely related notions of tuple and list, we write lists inside square brackets. For example, $[2, 3]$ is a list of two numbers, but $(2, 3)$ is a tuple of two numbers.

A list can have any length, including zero; $[\,]$ indicates an empty list. The *singleton* list [2] has only one member, making it tempting to say that it *is* exactly the same as the number 2. But in programming languages, it is usually necessary to keep lists and numbers separate. A list of one thing is not the same thing as its first (and only) member.

You can mix complex values in arbitrary ways. For example, [(2,5), (4,3), (9,7)] is a list of ordered pairs whose first member is the pair $(2, 5)$. Similarly, [[2,3], [ ], [4,8,10], [0]] is a list of lists and ([5,4], [6]) is a tuple of two lists.

Sequential representation of lists

Like a tuple, a list can be stored in memory by putting its members one after another. But unlike tuples, lists require additional length information that needs to be stored with the list. One convention is to store the length at the start of the representation, storing list ['a', 'b', 'c'] in the same way as tuple (3, 'a', 'b', 'c'). That makes it fast and convenient for a program to find the length information. A *string* is a list of characters, and some sequential implementations of strings store the length of the string in the first byte or two bytes.

An alternative way to indicate the length is to put some kind of special marker just after the list in memory. That requires a marker value to be available to use. For example, strings typically do not contain the null character (whose character code is 0, and which shows up as nothing at all when printed), so the null character is available as an end marker. To store list ['a', 'b', 'c'] with an end marker, you represent the list as the tuple ('a', 'b', 'c', **null**).

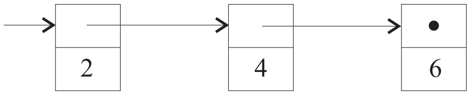
Sequential representation has some desirable characteristics. It is easy to implement, since a computer's memory is already set up as a sequence of bytes, and the computer hardware already performs the basic operations needed to manage sequential lists. A major advantage is that accessing the k-th value in the list, for a given k, is very efficient. If the list begins at memory address p, and each item in the list occupies b bytes, then the k-th value in the list begins at memory address $p + b(k - 1)$. A little arithmetic gets the address of the k-th value, in an amount of time that is the same regardless of how large k is. This *random access* property can be critical to the efficiency of some programs.

Unfortunately, the sequential representation method suffers from some difficulties. Suppose that you have a list $[40, 50, 60]$ of three numbers, represented as tuple $(3, 40, 50, 60)$, and you want to build the list $[30, 40, 50, 60]$ of four values, by adding 30 to the front of what you already have. To build the representation of the new list, you must find an unoccupied place in memory that is large enough to hold the new list and copy the old list into that memory, producing, sequentially, $(4, 30, 40, 50, 60)$. The amount of time and memory required to add a value to the front or back of a list using sequential representation is proportional to the length of the list.

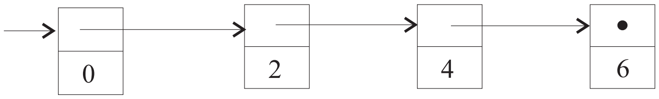
Linked representation of lists

Think again about a computer's memory. Each byte in the memory has an *address*, an integer telling where that byte is located in the memory, and a program can refer to any byte by its address. It can refer to a section of several consecutive bytes by referring to the address of the first byte in the section. But memory addresses are just integers, and so they can be stored in the memory. That allows one data item to refer to another data item by indicating the address where that other item is stored. When a variable or data item contains the address of another variable or data item, the address is usually called a *pointer* or a *reference*.

There is a way of storing lists that can avoid some of the copying that is required by the sequential representation. *Linked representation* stores the list $[2, 4, 6]$ in linked *cells*, each a small block of memory holding one thing and a pointer to the next cell. The last cell holds the *null pointer* 0, indicating the end of the list. (Memory address 0 is generally not an allowed memory address, so is available as a special marker.) In the following diagram of $[2, 4, 6]$ in linked form, the null pointer is shown as a bullet.



To add 0 to the beginning of this list is easy. Just create a new cell holding 0 and a pointer to the cell that is at the start of the previous list. The new list is as follows.



You can view the addition of 0 to the front as a change to the list (provided you use a pointer to the new cell as the new front). But notice that the representation of list $[2, 4, 6]$ is still intact, beginning with the cell that contains 2. So both lists $[2, 4, 6]$ and $[0, 2, 4, 6]$ are present. That allows you to think in terms of creating a new list without destroying the old one. The two lists share memory, which makes it possible to add a value to the front of a list in a constant amount of time, regardless of how long the list is, without destroying the list that was extended. Memory sharing also has important ramifications for programs that change things. Any change to the representation of list $[2, 4, 6]$ will implicitly change the representation of list $[0, 2, 4, 6]$ as well. Either no changes should be allowed, or the programmer should be aware that changes to one list will affect the other.

The linked representation makes use of an important characteristic of nonempty lists. The first cell of a nonempty list contains just two pieces of information: the first member of the list and a pointer to the rest of the list. So the information contained in a nonempty list can always be expressed as an ordered pair consisting of the first member of the list and the list of all members excluding the first member. For example, list $[2, 4, 6]$ can be broken into the ordered pair (2, [4, 6]). If we already have a way to represent ordered pairs (such as sequential representation), and we throw in a way to represent empty lists (such as using a null pointer) then we can represent arbitrary lists. That is exactly what the linked representation does. The first member of a nonempty list is called the *head* of the list, and the list of all remaining members is called the *tail* of the list. A nonempty list is defined by its head and its tail.

The linked scheme has some advantages over the sequential representation scheme, since it allows lists to be extended at their front cheaply. But it suffers disadvantages as well. An obvious one is that it uses more memory. If the members of the list use as much memory as pointers, then about half of the memory is used by the pointers, and the linked representation scheme uses about twice as much memory as the sequential representation scheme to store the same list. The ratio is not so bad when the list members require much

```
i = 0;
while (i < n)
{
  printf("%d ", A[i]);
  i++;
}
```

(a)

```
x = A
while(x is not an empty list)
  print(head(x))
  x = tail(x)
end while
```

(b)

Figure 8.1: **Two program fragments to print members of a list *A*.** Program (a) is written in C and uses indexing as the fundamental method of getting a member of an array. (C begins indexing at 0, so an array A of size 4 has members $A[0]$, $A[1]$, $A[2]$ and $A[3]$.) That style is appropriate for a sequential representation of lists, where indexing is efficient. Program (b) is written in pseudo-code, in a style more appropriate for a linked representation, as is used by Lisp.

more memory than the pointers, but the ratio is very bad when the list members are very small. For example, assume that a character occupies one byte, and a pointer occupies four bytes. Then to store a list of characters (a string) in a linked fashion would require about five times as much memory as to store the same string in sequential bytes of memory.[2]

Locating the k-th member of a linked list is much slower than locating the k-th member of a sequentially stored list, since you need to chain through all of the previous members, which takes time proportional to k. That compares to a constant amount of time, independent of k, for the sequential representation.

Choosing a representation

Most programming language implementations provide some kind of list or array, and offer either the sequential or the linked representation methods as fundamental. C, for example, wants to present the programmer with a direct view of the memory, so it provides arrays with sequential representation. C encourages the use of indexing (selecting the k-th member of an array) as a fundamental operation, and needs for that operation to be efficient. The choice of sequential representation is a good one for the kinds of things that C programmers usually do. Figure 8.1(a) shows a C program fragment to print the first n members of an array A. Notice the use of indexing.

Lisp uses linked representation as the basic method of representing lists. Lisp encourages the programmer to add new values to the front of existing lists, and to compute the head and the tail of a list. So the choice of a linked representation is a good one for the kinds of programs that Lisp programmers tend to write. With linked representation, getting the head and tail of a list are very cheap operations. Figure 8.1(b) shows a program (written in pseudo-code, not in Lisp) that prints all of the members of list A in a way that is suitable for linked representation. Instead of using an integer counter to keep track of an index, it keeps track of the rest of the list (x) that needs to be printed.

[2] In fact, it can be worse than that. As a consequence of memory allocation issues, some systems will actually use 8 or 16 bytes when you ask for a block of 5 bytes, causing the linked representation to use 8 or 16 times as much memory as the sequential representation.

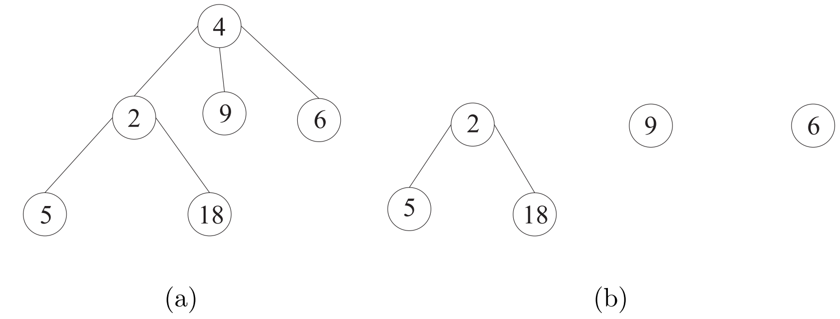


Figure 8.2: (a) An example of a tree data structure and (b) the three subtrees of tree (a).

Some language implementations use other representations of lists. This book uses the language Cinnameg for many of its examples. The current implementation of Cinnameg uses a hybrid representation that combines linked and sequential representation with another representation that uses trees as representations of lists (see the Section 8.6). The hybrid representation makes it possible to concatenate lists in a constant amount of time, something neither the linked nor sequential representations of lists can do. But the scheme requires the use of memory sharing in ways that are difficult to understand. As noted earlier, the programmer must either be aware of all memory sharing or must not be able to make modifications to data structures. Since the programmer cannot be made aware of the memory sharing in its lists, Cinnameg does not allow the programmer to rearrange a list once it is built.

8.5. Sets

Lists are ordered, and allow duplicates. For example, list $[8, 2, 8, 4]$ contains two occurrences of 8, and they occur in the first and third positions. *Sets*, on the other hand, are unordered and do not allow duplicates. Set $\{2, 4, 6\}$ is the same as set $\{6, 4, 2, 2\}$. SETL and Griffin are examples of languages that use sets as a fundamental data structure. Operations include intersection, union, creating a singleton set, etc.

One advantage of using sets is that there are so many choices that can be made for representation. You can use the list $[2, 4, 6]$ as a representation of the set $\{2, 4, 6\}$, but other representation schemes include binary search trees, hash tables and, for sets of small integers, bit arrays, where A[k] is 1 when k is a member of the set and 0 when k is not a member. The range of choices makes room for a powerful compiler to make intelligent choices that speed up program execution.

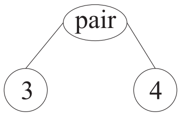
8.6. Trees

One frequently used kind of value is a *tree*, such as the one shown in Figure 8.2(a) A tree has a *root value* and zero or more *subtrees*. The subtrees are ordered from left to right, so there is a first subtree, a second one, etc. The tree shown in Figure 8.2(a) has a root value of 4, and has three subtrees, shown in Figure 8.2(b).

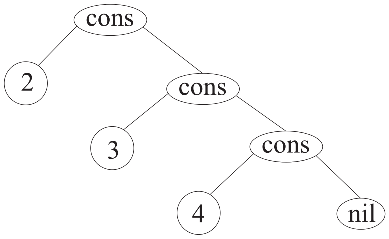
A tree is characterized by its root label and a list of zero or more subtrees, so a nonempty tree with three subtrees can be represented as an ordered pair (root-label, [subtree-1, subtree-2, subtree-3]). As long as we know how to represent ordered pairs and lists, we know how to represent trees.

Using trees to represent other kinds of data

Trees are amazingly versatile in what they can do. You can view ordered pairs and lists as special kinds of trees if you like. Recall that a *symbol* is a kind of simple value that is described by its name. A convenient way to represent an ordered pair as a tree is to use symbol **pair** as the root label, with the two parts of the pair represented by two subtrees. For example, ordered pair $(3, 4)$ might be represented by the tree following tree.



Lists can also be represented as trees in several different ways. One popular way is to assume that there are symbols *cons* and *nil* and to represent list $[2, 3, 4]$ as follows.



A tree with just one node that holds symbol nil stands for the empty list. Notice that it is easy to find the head (in the left subtree of the root) and the tail (in the right subtree of the root). In fact, the tree representation is really just the linked representation turned diagonally.

There are other ways to use trees to represent lists. For example, a tree might, by convention, be traversed in preorder to find the members of the list. (A preorder traversal first selects the root, then does preorder traversals of the subtrees, in left-to-right order.) List $[3, 6, 4, 7, 9, 5]$ could be represented as the tree in Figure 8.3. That probably sounds like a strange choice, but it has the advantage that there are many different ways to represent a list, and that flexibility can be exploited to implement some operations on lists very efficiently. It forms the basis of one of the ways in which lists are represented in the Cinnameg implementation.

You might wonder why lists and tuples would be represented as trees when earlier we did the opposite, representing trees in terms of tuples and lists. But what happens inside a language implementation is invisible to the programmer. The point is that a language that only supplies trees does not really need to supply lists and pairs separately. Prolog is an example of a language that uses trees, and expresses other data structures in terms of them.

Trees are written in programming languages in a variety of ways. Prolog uses notation pair(3,4) for the tree with root label *pair* and two subtrees, 3 and 4. In Cinnameg, you write *r*-<*L* to describe a tree whose root label is *r* and whose subtrees are the trees in list *L*.

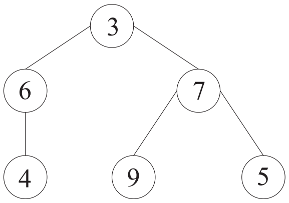
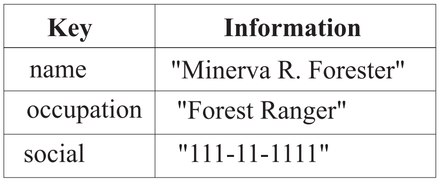


Figure 8.3: **A tree that can be used as a representation of list $[3, 6, 4, 7, 9, 5]$.** The tree is traversed in preorder, where you start with the root label, then traverse the left subtree in preorder, then traverse the right subtree in preorder. The head of the list is easy to get because it is stored at the root of the tree.



Key	Information
name	"Minerva R. Forester"
occupation	"Forest Ranger"
social	"111-11-1111"

Figure 8.4: **A diagram of a table, or dictionary.** A table consists of a collection of rows, where each row has a key and some information associated with that key. No two rows contain the same key.

8.7. Tables

A *table* is a data item that you can think of as a collection of rows, each having two columns called the *key* and the *information.* No two rows have the same key value. Figure 8.4 shows an example of a table. The keys can, in principle, be any kind of values, not just strings or names, but keys that are strings are very common. Operations on tables include lookups (finding a row with a particular key and returning the associated information) adding a new row and removing the row that has a given key. Python and Cinnameg are examples of languages that provide tables as fundamental data types. (Python calls tables *dictionaries.*)

A table is a set of rows. So any representation of sets that allows the members of the sets to be ordered pairs can be used to represent tables. Typical representations are binary search trees and hash tables.

8.8. Tags and tagged values

In order to compute with a value x, it is often necessary to know what kind of item x is. For example, instructions for adding integers are typically different from instructions for adding floating point (real) numbers, and to perform an addition, you need to know whether the numbers are integers or floating point. In many cases the kind of item is clear from context. For example, if a Fortran program contains line `INTEGER N` then it is clear that the value called N will always be an integer, and can be treated as such. Sometimes, however, it is not apparent from the program what kind of value a variable holds, and that information needs to be obtained while the program is running. Typeless languages, such as Lisp, illustrate that; a Lisp program does not say what the type of a variable is, and the compiler cannot determine that information. The program cannot look at a pattern of bits and determine whether it stands for an integer, a real number, or something else, without some kind of help.

To handle this, values are typically *tagged* to make it possible to tell the difference between different kinds of things at run time. For example, suppose that you have a variable that might hold a list of integers, but might also hold just a real number. A tag convention can be adopted; tag 0 indicates a real number, tag 1 a list of integers. Then a tagged number 4.5 can be represented as an ordered pair $(0, 4.5)$, where the left-hand member is the tag and the right-hand member is the value. List $[3, 5]$ would be represented as an ordered pair $(1,$ [3,5]$)$, with tag 1. Any time a tagged value is used, the program consults the tag to see which kind of value is present, and selects a mode of computation based on the tag. We will encounter uses of tags in subsequent chapters.

8.9. First class data items

Numbers are examples of data items that a program can store and manipulate. You can pass a number to a subprogram, return a number from a function, put a number into a variable, etc. Depending on the programming language, you might be allowed to do the same kind of things with strings, lists, arrays, ordered pairs, and so forth. A kind of data item that can be moved around, stored, etc. in all of the usual ways is called a *first class* data item.

Although a language almost surely treats numbers, characters and a few other things as first class items, programming languages differ from one another in terms of which other kinds of things are first class. For example, C allows you to return a record (called a structure in C) from a function, but Pascal does not. C allows you to return some arrays from functions, but not others.

You are probably used to thinking of a function or procedure not as an item, but as a tool that can be used in a program. But a real tool, such as a hammer, is a perfectly good thing, and there are things that you can do with a hammer, such as putting the hammer into a tool box, that do not involve using it as a tool. Some languages allow you to treat functions and procedures as data items. Pascal allows you to pass a function as a parameter to another function. Think of lending your hammer to a friend. But Pascal does not allow a function to return a function as its result, so you cannot ask your friend to make a tool for you and to give you back the new tool. In Pascal, functions are not first class, and they can only accompany requests, not answers to requests. On the other hand, some languages, including Scheme and Cinnameg, treat functions as first class items, and allow them all of the usual privileges that data items have. You can put a function into a variable, return one from a function, and so on, allowing a program to build new functions on the fly.

There is a surprisingly wide variety of things that have been made first class items in programming languages. Examples include variables, functions, procedures, expressions, types (as data items), modules, and even events that occur during the execution of the program, allowing a program to examine what it is doing or even what it might do in the future.

8.10. Objects

If you think of a subprogram as a first class thing, then you can imagine creating a record where some of the things are subprograms. When you do that, what you have is an *object*.[3] Subprograms that are part of an object are usually called *methods*, and they normally produce or manipulate information stored in the same object. The typical way to use an object is to look up one of its methods and run it. Objects are the fundamental kind of data item used in object-oriented and object-based programming, the topic of Part XIII.

8.11. Persistent and ephemeral data

We have seen that the collection of operations that a given type of data supports has a strong influence on how you choose to represent that data. But the available operations can also influence your conceptual view. To illustrate, think of strings (lists of characters).

Suppose your operations only allow you to look at a string, and never to change it; for example, you can get the k-th character of a string, and you can concatenate two strings to build another string. Then you think of a string as a sequence of characters, and you can completely characterize a particular string by saying that it is “frog”; there is no extra hidden information about it. If variables x and y both contain “frog”, then they must contain equal values, since the only information that you have about them is the sequence of characters.

But now suppose another operation is added that changes a string, modifying the memory that it occupies. For example, you might be allowed operation addChar(s, c) that adds character c to the end of string s. Then it is no longer adequate to describe a string by a particular sequence of characters; you have to say something more about it. For example, you can describe a string as an object that currently holds information “frog”, but that might hold a different string in the future. Two objects that both currently contain “frog” might not be the same object. The conceptual view of a string has to adapt to the available operations.

Finally, suppose that you not only have operations that modify the characters in strings, but you also have the ability to use memory sharing. For example, suppose that strings are represented sequentially, with an end marker, and that function tail(s) just gives you the memory address one character after the beginning of s, which is a representation of the tail of string s. Then sequence of operations

```
x = "frog";
y = tail(x);
addchar(x, 'g');
```

leaves variable y referring to string “rogg”. A change to x has also changed y, because they share memory. The conceptual view of strings that you use must be accurate, and it must predict the behavior of programs like this. You are forced to adopt a view of strings that is very close to the way they are represented in the memory, including the memory sharing; the simple view that a string is a sequence of characters is untenable.

Generally, types of data that only support operations that look at the data are called

[3] In an object, some of the components can also be marked as being usable only by subprograms that are part of the same object. We ignore that for now.

persistent[4] types of data, and they tend to enjoy relatively simple conceptual views. Types of data that allow changes are called *ephemeral* types of data. Declarative languages typically only allow persistent data types, while imperative languages typically employ ephemeral data types. But there are exceptions. For example, Java, an imperative language, provides strings in a persistent way; no operations are available that change a string.

8.12. Summary of terminology and concepts

Data is the foundation for computation, since there can be no computation without something to compute. A large part of a program typically consists of passing data from one subprogram to another. Simple data items have no visible structure, while complex data items have apparent structure.

There are two ways to view data. The *semantic* or *conceptual* view is the basis for the language definition, and tells only enough conceptual information about the data to allow a programmer to understand what the data items mean and what operations on them do. The *representational* view is the basis for the language implementation. Data can be represented in a wide variety of ways, some of them far from obvious, and a language implementation derives part of its efficiency or lack thereof from how it chooses to represent its data.

Terminology

Boolean values There are two boolean values, true and false.

Character A character is a value such as ’a’ and ’b’ that can, for example, be read from a keyboard or a file, or shown on a monitor.

Complex value A complex item is a data item that has visible internal structure, and whose parts can be extracted by operations.

Data The term data refers to all of the items that a program manipulates, including things that can be put in variables, passed to subprograms, etc.

Ephemeral data type An ephemeral type of data allows operations that change values in-place, so that what was formerly stored with that data is replaced.

First class item A data item is first class if it can be moved around and stored in the usual ways of other data items. For example, it should be possible to pass it as a parameter to a function, return it from a function and store it in a variable, if those capabilities are provided for most data items.

Head The head of a list x is the first member of x. For example, the head of $[5, 4, 3, 2]$ is 5. The empty list has no head.

[4] This definition of the word *persistent* is common to some fields, including analysis of algorithms. It is different from the way the same term is used in object-oriented programming, where a persistent object is one that can change over time, but that can be modeled at any instant as if it holds a particular value, so that it has meaning outside the context of a particular program. (An object that contains references to other objects is not generally persistent, since it is tied to the context where those other objects exist.) The same term is also used for data that is stored in persistent memory, such as on a disk, having nothing to do with the definition used here.

List A list is a sequence of zero or more items. In a typed language, the members of a list must all have the same type.

Ordered pair An ordered pair, such as $(2,4)$, has a left-hand member and a right-hand member. An ordered pair is a special case of a tuple, where there are just two members.

Persistent data type A persistent type of data only allows operations that look at the data, possibly including operations that build new things from old ones without destroying the old ones.

Pointer A pointer is a memory address. Some languages make pointers explicitly available to be used by programs, but some languages hide their use in the implementation.

Record A record is a tuple whose parts have names rather than positions.

Reference The term *reference* is used interchangeably with the term *pointer*.

Set A set is an unordered collection of things without duplicates. Sets have the advantage that they can be represented in many ways, allowing a language implementation to make intelligent choices.

Simple value A simple value is one that has no visible internal structure.

Symbol A symbol is a simple value that is entirely characterized by its name.

Table A table associates information with keys. You can look up a key and find the associated information.

Tag A tag is a value (often an integer in practice) that indicates the kind of another object. It is used when the kind cannot be determined until the program runs.

Tail The tail of a list x is the list that you get by removing the first member of x. For example, the tail of $[5,4,3,2]$ is $[4,3,2]$. The empty list has no tail.

Tree A tree is a data structure that has a root value and zero or more subtrees.

Tuple A tuple is a finite sequence of items, such as $(2,4,2)$. Normally, when you work with a tuple in a program, you know while writing the program how many items the tuple will contain when the program runs. In a typed language, the components of a tuple can have different types.

8.13. Exercises

8.1. What are the head and tail of each of the following lists? If the list does not have a head or tail, say so.

(a) $[2,4,6,8,10]$

(b) $[2, 7]$
(c) $[5]$
(d) $[\,]$

8.2. What is the difference between a simple value and complex value?

8.3. Integers written in decimal notation appear to be sequences of digits. Why are integers not usually considered to be complex values?

8.4. What is the difference between a symbol and a string?

8.5. Some languages allow variables to be first-class items. What does it mean to pass a variable to a subprogram? Give an example where it might make sense to use a variable as a data item in a program.

8.6. Lisp is a typeless language, which means that all type checking is done at run time. How can a lisp program (or interpreter) tell what kind of data it is looking at? What mechanism does it use?

8.7. Give two important advantages of sequential representation of lists over linked representation.

8.8. Give two important advantages of linked representation of lists over sequential representation.

8.9. When a data type is persistent, the presence of memory sharing does not affect your conceptual view of the data. Why not?

8.10. Suppose that lists are represented sequentially in memory, and suppose for simplicity that lists are persistent, so that, once created, you will never change what is in a list. In order to pass a list to a subprogram, is it necessary to copy the entire list into the subprogram, or is there a more efficient way to pass the list? If there is a better way, what is it?

8.11. Java does not provide any operations that modify strings. (You can compute new strings, but never change an existing one. For example, computing $s + t$, the concatenation of strings s and t, does not change s or t.) But Java has one unusual operator, ==, which asks whether two strings are stored in the same place in memory. A variable that holds a string actually holds the memory address where the string is stored. So if variables x and y both hold string "frog", then $x == y$ might be true and might be false, depending on whether the occurrences of "frog" referred to by x and y are in the same memory location. Explain how the presence of the == operator affects the conceptual view of a string. Is it adequate to explain a string solely as a sequence of characters?

8.12. For this question, assume that lists are persistent (a list cannot be modified once it is created), as is the case in Cinnameg. Memory sharing is allowed. Computing the length of a linked list normally requires counting the number of links that need to be traversed before the null pointer at the end of the list is seen. Describe a modification of linked lists where computing the length takes only a constant amount of time, regardless of how long the list is. Ensure that the operations of taking the head and tail, and of adding a value to the front of a list (producing a new, longer list), still take a constant amount of time.

8.13. Describe a scheme for representing lists such that the operations of indexing (taking the k-th member), getting the length, and computing the tail of a list each take a constant amount of time, independent of the size of the list. Notice that you do not need to build larger lists in constant time. Assume that memory sharing is allowed.

8.14. Describe a hybrid representation of lists that is linked, but allows an arbitrary number of items in each cell. Both standard linked representations and sequential representations are special cases of this. How would you find the k-th member of such a list?

8.15. Suppose that lists are persistent. Memory sharing is allowed. Describe a version of the hybrid representation of the previous question that allows you to take the tail of a list in constant time, regardless of how many items are stored in the first cell of the list. How is the tail computed?

8.16. What motivation is there for using a hybrid representation for lists. What are the advantages? What are the pitfalls?

8.17. How long, as a function of n, does it take to run the loop of Figure 8.1(a)? Give the answer to within a constant factor (big-O notation).

(a) Assume arrays are stored using sequential representation.

(b) Assume arrays are stored using linked representation.

8.18. How long, as a function of n, does it take to run the loop of Figure 8.1(b)? Give the answer to within a constant factor.

(a) Assume that lists are stored using linked representation.

(b) Assume that lists are stored in sequential representation, with the size stored at the beginning. (How can the tail of such a list be computed? How long does it take?)

(c) Assume that lists are stored in sequential representation, with an end marker. (How can you take the tail? Assume that memory sharing is allowed.)

8.19. A sequence can be stored as a linked list with pointers going both directions, both from a cell to the next one, and from a cell to the previous one. Can you see any advantage to this representation? What kind of operation might be easier to implement?

8.20. Neither the linked representation nor the sequential representation is good for representing a list that can be extended cheaply by adding a member to its end. Describe a representation that does allow extending a list at its end cheaply (in a constant amount of time, regardless of the length of the list), while still allowing rapid access to the beginning of the list, and rapid removal of the first member. Assume that lists are ephemeral, and adding an item to the end of the list changes the list, destroying the information that was formerly held in the list.

8.21. Does your solution to Exercise 8.20 work when lists are required to be persistent, and adding a member to the end of a list creates a new list, leaving the old one intact? Did you need to destroy the old list in order to create the new one?

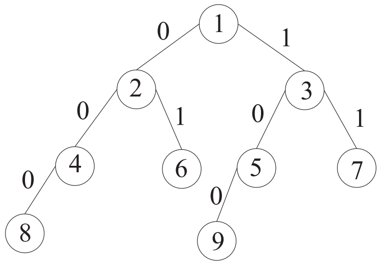


Figure 8.5: **A tree representation of a list.** This tree represents a list of nine values, with the k-th value stored at the node labeled k, for $k = 1, \ldots, 9$. (As shown, it represents the list $[1, 2, 3, 4, 5, 6, 7, 8, 9]$.)

8.22. Modify your solution to Exercise 8.20 so that, in addition to extending the list, deleting the last member of the list is cheap, and getting the list of all members except the last is also cheap. Assume that lists can be modified.

8.23. You can represent a list as a pair (N, T) where N is the length of the list and T is a binary tree. In the tree, label each edge by either 0 or 1, choosing label 0 for each left edge and 1 for each right edge. See Figure 8.5. Each node in the tree has an associated binary number. To get it, start by writing 1. Now, starting at the given node, read the labels upwards to the root, writing each 0 and 1 encountered after the 1 that you started with. For example, if you start at the node labeled 6 in the tree shown in Figure 8.5, then you write 110, starting with 1 and then writing 10. Notice that 110 is the binary representation of 6. You store the sixth member of the list at the node labeled 6 in the figure. In general, you store the first member at the node labeled 1, the second member at the node labeled 2, etc.

Suppose that lists are persistent, so you are not allowed to destroy one list in order to build another one. Show how to use this tree representation to implement the following three operations in time proportional to $\log_2(n)$, where n is the length of the list. The operations are (1) indexing (finding the k-th member), (2) adding a member to the end of the list (producing a new list), and (3) removing the last member of the list (again, producing a new list). Explain how each operation is implemented by writing it in pseudo-code. Be sure to check that, after you do any given operation, the previous lists still exists.

8.14. Bibliographic notes

IEEE floating point numbers are described in [61]. Okasaki [83] shows how to implement lists efficiently as trees. Books on data structures, such as Carrano and Prichard [25], cover representations of sequences, trees and other data types, and efficient algorithms that work on those representations.

Wolfram [114] explains Mathematica. Schwartz et. al. [94] describe SETL. You can find information about Griffin at http://cs.nyu.edu/cs/faculty/goldberg/griffin.html. The language Tahiti [1] treats events as first-class data items.

Exercise 8.23 describes an implementation of *persistent arrays.* Aasa et. al. [2] compare some representation schemes for persistent arrays, and Paulson [86] describes implementing them in Standard ML.

Chapter 9

Data Types

9.1. Types

When a program runs, it needs to know what kind of data it is working on. For example, a program probably should not try to multiply two strings. If it is asked to add numbers, it needs to know whether the numbers are integers or real numbers. Although tags suffice to check types while the program is running, many languages employ type information to allow a compiler to check types without running the program at all. This chapter examines some of the most basic ideas about types. Parts XI and XII explore more detail on types.

9.2. Sources of type information

A *dynamically typed* language (also called a *typeless* language) does all of its type checking while the program runs. It keeps tags on data items and checks the tags whenever it performs one of its basic operations. Scheme, Prolog and Python are examples of dynamically typed languages.

A *statically typed* language requires that some type checking be done by a compiler while it translates a program. Information about types can be used to check that the program is sensible from a type standpoint, and also to generate the correct kinds of instructions, adding two integers using an integer add instruction, etc.

Information used by a compiler can come from three sources. The first concerns built-in operations. A compiler might know that the addition operator can either take two integers and produce an integer, or can take two real numbers and produce a real number. Information about built-in things is loaded into the compiler when it starts.

A second source of information is explicit in a program. If a C program contains declaration `int x`, then the compiler knows that variable x holds an integer. A language might require that the types of all variables, and of other identifiers such as those that name functions, be given explicitly in the program. The C *function prototype*

```
int f(int,int);
```

is a way of telling the compiler that function f takes two integers as parameters and produces an integer result.

```
fun
    len([]) = 0
  | len(h::t) = 1 + len(t);
```

Figure 9.1: **Definition in SML of a function that computes the length of a list.** Each line is an equation. The first says that the length of the empty list is 0. Expression $h :: t$ stands for the list whose head is h and whose tail is t, so the second line says that the length of a nonempty list one larger than the length of its tail. This style of definition is discussed in Chapter 18.

The last source of information is inference. Suppose that the compiler has already determined that x and y are integers. When the compiler sees expression $x + y$, it can determine that $x + y$ must produce an integer. Larger expression $z * (x + y)$ then clearly involves a second operand that is an integer. All compilers for typed languages do type inference to some degree.

Some languages, including Standard ML (SML) and Cinnameg, carry the idea of type inference to an extreme, requiring type information only when it cannot be inferred by the compiler. Figure 9.1 shows a definition in SML of a function that computes the length of a list. Notice that the definition contains no explicit type information. But the compiler knows that [ ] is an empty list, and the presence of expression len([ ]) tells the compiler that len must be a function that takes a parameter that is a list. Similar inferences at other expressions lead the compiler to conclude, based on the way things are used and the context in which they occur, that len is a function that takes a list as a parameter and produces an integer result. Chapter 37 gives more detail on how an SML compiler performs type inference.

9.3. Type checking

A *type error* is any error where the programmer is using an operation on a type of data to which it does not or should not apply. For example, taking the square root of a string should be a type error. Adding two characters is a type error, since 'a' + 'b' is surely a meaningless concept on characters. The definition of a type error depends on the language, but it should be defined from a logical point of view. A language is *type-safe* if every type error will be detected, either at compile time or at run time. If it is not type-safe, so that some type errors go entirely undetected, say that a language is *type-unsafe*.

Dynamically typed languages can be type-safe, but you will not learn about a type error until the error occurs during execution of a program. Finding type errors can be a matter of tedious debugging.

Static type checking

Static type checking offers the programmer a much easier way to learn about type errors; the compiler reports them, without the program being run at all. Static type checking offers the programmer a kind of security whose worth cannot be overestimated, since a type checker can assure the programmer, once type errors are no longer reported, that certain kinds of errors cannot happen in his or her program.

Say that a syntactically correct program is *well typed* if a compiler detects no type errors in it, and is *poorly typed* if the compiler reports a type error. A *strongly statically typed language* is one that is type-safe, and where all type errors are detected at compile time. In a strongly statically typed language, a well typed program cannot encounter runtime errors due to type mismatches, or due to a programmer's faulty reasoning about types. In fact, there is no need for any type checking to be done at run time. A strongly statically typed language is at the opposite extreme from a dynamically typed language.

An example of a strongly statically typed language is Standard ML. Languages of the ML family grew out of work on automatic theorem provers, and the type system is critical to ensuring the correctness of the theorem provers. Clearly, they require a rock solid type system, without any loopholes or back doors that allow type errors to slip by.

Most programming languages are neither dynamically typed nor strongly statically typed, but are somewhere in between, with some type errors reported at compile time, some reported at run time, and (possibly) some escaping detection altogether. Any language that is neither dynamically nor strongly statically typed is *weakly statically typed.* C[1] is an example of a weakly statically typed language. In C, variables must be given types explicitly within a program. If you perform an operation on incorrectly typed arguments, you (usually) get a compile error, and cannot run the program. However, there are some rather large loopholes in the type system that allow type errors to occur when a program runs.

C is type-unsafe. The type errors that are not reported at compile time will never be reported, not even when the running program encounters them. An example concerns the treatment of pointers. In C, pointers occur explicitly in programs. If T is a type, then a value of type $T*$ is a pointer to a variable of type T. For example, a pointer of type int* points to, or refers to, a variable of type int. If you have a pointer of type $T*$, you can *cast* it to another type $S*$, telling the compiler to pretend the memory pointed to contains something of type S rather than type T. There is no checking that the conversion is sensible, and it can cause arbitrarily bad type errors at run time.

Java is another weakly statically typed language.[2] It does not allow the serious unchecked type errors that a C program can make, but a Java program needs to perform some type checking while the program is running. Java has a notion of typing that allows, for example, a horse to be considered a special kind of mammal. Suppose that you have a variable m of type Mammal, which, of course, you know holds a mammal. Based on what you think is going on in your program, suppose you reason that m must actually contain a horse, not a giraffe, or some other kind of mammal, and you would like to put this horse into a variable h of type Horse. In Java, you can write

```
h = (Horse) m;
```

It is impossible for the compiler to determine whether you are right about m holding a horse, and so it allows that. At run time, the Java program will check whether you are right, and if not, it will cause the program to fail.[3] If it turns out that you were mistaken, and that

[1] There are several versions of C, some more weakly typed than others. We will assume that C means ANSI C, a standardized version of C that is better than most about type checking, but is still weakly typed.

[2] You will find books about Java that say that Jave is strongly typed. What they generally mean is that type information about variables is required to be written in Java programs. Here, we are less concerned with how type information is conveyed than with the reliability of the type checking and whether some type checking is done while a program runs. We have avoided the term *strongly typed* to avoid confusion.

[3] It will throw an exception. Exception handling is discussed in Chapter 25.

m really holds a giraffe, then you, the programmer, have made an error in reasoning about types that went uncaught by the compiler.

Ideally, the Java run-time system would detect all type errors not reported by the compiler. Unfortunately, there are a few that are ignored, so Java is not entirely type-safe. Chapter 44 discusses how type errors can escape detection in a Java program.

9.4. Types as sets

We have talked about types, but have not said just what a type is. The simplest explanation is that a type is a set of values. For example, type Boolean is the set {false, true} and type Char is the set of all characters in a particular alphabet. Saying that a type is a set is a kind of *model theory.* In general, a model theory describes something unfamiliar by modeling it by something that is familiar. The type Integer in a programming language is modeled by the set of values $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$, with which you are already familiar.

The viewpoint that a type is a set will serve us well for the remainder of this chapter. But there is more to be said about just what a type is; Chapter 35 examines an alternative viewpoint.

9.5. A notation for types

Assume that a few basic types, such as Boolean, Char and Integer, are available. Complex types, whose members are complex data items, can be built up from those. Complex types are also called *structured types.*

Lists

In a statically typed language, a list is a sequence of items of the same type. Some languages have a concept of an *array*, which is a list that is stored sequentially in memory, making some operations efficient. We will treat all kinds of lists in a similar way here, regardless of what they are called or how they are represented. A list such as $[3, 5, 7]$ whose members are integers has type "list of integers", and a list such as [[2], [4,5]] whose members are lists of integers has type "list of lists of integers". Lists can typically be of arbitrary length; the length is not part of the type. So a variable of type "list of integers" might hold a list of any length.

Type "list of integers" is written in different ways in different languages. We will adopt the Cinnameg notation [Integer]. In general, $[T]$ is the type "list of T", where T is a type. That looks similar to the notation for writing a list, with one important difference. If x is an item, then $[x]$ is a list that has just one member, x. But if T is a type, then $[T]$ is the type of *all* lists whose members have type T, regardless of length. List ["mouse","rabbit","capybara"], for example, has type [String], and [[2,4], [5]] has type [[Integer]].

Tuples and Cartesian products

In a statically typed language, a list is a way to group together values of the same type. But you probably also want to be able to group values of different types, and some other mechanism needs to provide for that.

```
Define big(x: Integer): Boolean = (x > 100).
```

Figure 9.2: **Function big, written in Cinnameg, with *domain* Integer and *codomain* Boolean.** Cinnameg allows you to indicate that expression E must have type T by writing $E : T$. Any types that you do not explicitly indicate are determined by the compiler.

The Cartesian product $S \times T$ of two sets S and T is the set of all ordered pairs (a, b) where a is a member of S and b is a member of T. Cartesian products in mathematics lead to the notion of Cartesian product types, or simply *product types*. Type Integer × Real contains ordered pairs (x, y) where x is an integer and y is a real number. Larger tuples can be formed from ordered pairs. For example, ordered triples can be handled by letting a triple (a, b, c) be thought of as an abbreviation for $(a, (b, c))$, where the second member is itself an ordered pair. The type of $(2, ('j', 4))$ is Integer×Char×Integer, where × associates to the right.

For a programming language, it is convenient to choose a notation for types that mimics the notation for the members of the type. Since an ordered pair is written (a, b), we will write the Cartesian product $S \times T$ as (S, T), as is done in Cinnameg. Cartesian product $S \times T \times W$ is written (S, T, W). So (Integer, String, Boolean) is the type of a triple where the first member has type Integer, the second member type String and the third member type Boolean. But, since it is equivalent to Integer×(String×Boolean), type (Integer, String, Boolean) is thought of as a shorthand for (Integer, (String, Boolean)).

Records

You talk about the members of an ordered pair as the left-hand member and the right-hand member. In Cinnameg, for example, there are two functions for getting the members of a pair: $\text{left}((x, y)) = x$ and $\text{right}((x, y)) = y$.

Records are similar to tuples except that their parts are referred to by name rather than by position. The type of a record needs to tell the name and type of each component. We will use notation {shoesize: Integer, height: Real, social: String} to indicate a record type with three components, each having a name and a type. The order of the components in a record is irrelevant. For example, {height: Real, name: String} is exactly the same type as {name: String, height: Real}.[4]

Functions

Figure 9.2 show a definition of function big, written in Cinnameg, taking a single integer parameter and producing a boolean result. Each function has a *domain*, the type of the parameter that the function can take, and a *codomain*, the type of the result that it produces. For example, function big has domain Integer and codomain Boolean. If function f has domain type A and codomain type B, then f has type $A \to B$. So big has type Integer → Boolean. If function f has type $A \to B$ and x has type A, then expression $f(x)$ has type B. Expression big(20), for example, has type Boolean (and value false).

[4] Cinnameg does not support records in this form, but some other languages, such as SML, do.

```
double max(double x, double y)
{
  if(x > y) return x;
  else return y;
}
```

Figure 9.3: **A C function that takes the maximum of two real numbers.** The type of max is (double, double) → double in our notation for types. (The C language uses its own notation for types. In C notation, max has type double(*)(double,double). We will stick with our notation, regardless of the language being described.)

Mixing types

Notations for list, product, record and function types are called *type constructors*, since they are used to build types. Type constructors can be combined in arbitrary ways to make more complex types. The addition function (+) for integers has type (Integer, Integer) → Integer. That is, it takes a pair of integers and produces an integer. Since $(4, 7)$ has type (Integer, Integer), expression $(+)(4, 7)$ has type Integer. (Of course, $(+)(4, 7)$ is usually written in the form 4+7, but that is only a syntactically convenient notation, and has no bearing on the meaning.) Figure 9.3 shows function max, written in C, which computes the maximum of two real numbers (type **double** in C). Its type can be expressed as (double, double) → double. Function g defined in Cinnameg by

```
Define g(x: Integer, y: Integer) = (x + y, x - y).
```

takes a pair of integers and produces a pair of integers. It has type (Integer, Integer) → (Integer, Integer).

What about functions head and tail on lists? Since head($[2, 3, 4]$) = 2, the head function for lists of integers has type [Integer] → Integer. Since tail($[2, 3, 4]$) = $[3, 4]$, it is apparent that the tail function for lists of integers has type [Integer] → [Integer].

9.6. Subtypes

Constrained types

Since we are thinking of a type as a set of values, it makes sense to ask for a subset of a type, or to say that the value of a particular variable is constrained to lie within some given subset. Some languages allow the programmer to specify subsets of types in limited ways. For example, in Ada, if you write

```
X: Integer range 2..5;
```

you indicate that variable X must contain an integer, and that the value of X must be one of the integers 2, 3, 4 or 5. Variable X is said to be of the *subtype* Integer range 2..5 of type Integer.

Subtypes, also called constrained types, are useful for two reasons. One is that the compiler can tell, from the range of values that need to be represented, how much memory a variable needs. For example, a compiler might use only one byte for a variable of type Integer range 2..5, since all of the integers from 2 to 5 can be stored in a byte. An alternative way to obtain the same information (as exemplified by the approach taken by C and Java) is to provide a collection of basic types such as byte, short (two bytes), int (four bytes) etc., indicating the amount of storage needed in machine-specific form. But asking a programmer to tell just which values need to be stored (conceptual information relevant to the programmer) is preferable to asking the programmer to give low level information such as how many bytes are needed for the variable.

A second benefit of constrainted types is error checking. If variable X must lie in the range 2..5, then any attempt to make X hold a value not in that range will be caught (either at run time or compile time) as an error.

Typically, if you write an assignment $X := Y$ in Ada, you expect X and Y to have exactly the same type, or an error is reported by the compiler. With constrained types, it is unrealistic to be so restrictive. Suppose that X has type (Integer range 1..10), and Y has type (Integer range 2..6). Then the assignment is clearly valid, even though the types of X and Y are different. Cases can arise where an assignment might be correct and might not. If, for example, X has type (Integer range 3..7) and Y has type (Integer range 5..10), it might make perfectly good sense to allow assignment $X := Y$. The compiler generates code to check the value of Y.

Subtypes of records

A subtype is a special case. For example, an integer in the range from 1 to 9 is a special case of an integer. Record types also have natural special cases. Suppose that SHS is record type {shoesize: Integer, height: Real, social: String} and SH is type {shoesize: Integer, height: Real}. You can argue that a value X of type SHS should be considered to be a special kind of value of type SH, and that the former type is a subtype of the latter, since it has all of the required components (and more). So you should be allowed store X into a variable of type SH. (Notice that you get a subtype of a record type by *adding* components, not by removing them.) You should also be able to pass X to a function that expects a parameter of type SH. The effect is that functions can work on several different types of parameters. (A function that expects a parameter of type SH can handle any type of parameter that is a record having at least those components.) Generally, a subprogram that works on more than one type of parameter is said to be *polymorphic*. Part XII explores polymorphism, and Part XIII looks at polymorphism in object-oriented programming, where subtypes are similar in spirit to subtypes of records.

9.7. Summary of terminology and concepts

The types that we have seen in this chapter can be modeled by sets. For example, type Boolean is modeled by the set {false,true} of values that a variable of type boolean can hold.

Complex, or structured, types can be built up from simple types and other complex types. Examples of complex types are list, product, record and function types.

Some languages support subtypes, allowing the programmer close control over the values of variables and parameters and results of functions.

Terminology

Cartesian product The Cartesian product of sets A and B is the set of all ordered pairs (x, y) where x is a member of A and y is a member of B. The usual mathematical notation for the Cartesian product of A and B is $A \times B$. We have used notation (A, B) when A and B are types.

Codomain The codomain of function type $A \rightarrow B$ is type B.

Constrained type A constrained type is a subset of another type that is characterized by a constraint, a condition that the members are required to satisfy. Most programming languages that allow constrained types only allow constraints of very limited forms.

Domain The domain of function type $A \rightarrow B$ is type A.

Function type A function type describes functions from a given domain to a given codomain. The domain tells the type of the parameter to the function, and the codomain tells the type of the result of the function. The type of functions with domain A and codomain B is written $A \rightarrow B$.

List type A list type is a type, such as $[A]$, of lists all of whose members have the same type, A.

Model theory You employ a model theory when you model one thing by another. Treating a type as a set of values is a kind of model theory where types are modeled by sets.

Product type A Cartesian product type is usually just called a product type.

Record A record is a kind of product type where the components are referred to by name rather than by position. For example, instead of asking for the left-hand, or first, component of a pair, you might ask for the component called *size*.

Structured type A structured type is a type whose members are complex values.

Subtype In the model-theoretic view, type S is a subtype of type T if S is a subset of T. Conversions from S to T can be done safely, and conversions from T to S can be done if a run-time type check is performed to ensure that the conversion is sensible.

In a more general view, a subtype is a special case. For example, record type {size: Integer, weight: Real} is a subtype of {size: Real}, since a value with both an integer size and a real weight is a special kind of value with an integer size.

Type A type is an indication of the kind of data that an item can be, or that a variable can hold. In the model-theoretic view, a type is a set of values.

9.8. Exercises

9.1. C allows expression 'a' + 'b'. Should that be considered a type error that goes undetected, or should it be considered a normal operation that is not erroneous. Explain.

9.2. Give three ways that a compiler can determine type information.

9.3. List all of the values in the set that models type (Boolean, Boolean).

9.4. Does the Java type **double** have a model-theoretic semantics, or is it too complicated for that? Explain. (In Java, a value of type double is a real number, represented to a given prescribed precision. There is a finite, but large, set of *model numbers* that can be represented.)

9.5. List all of the members of the set that models type Boolean → Boolean. What do such things look like, and how can you write one down?

9.6. Does type Integer → Integer have a model-theoretic semantics? Explain.

9.7. What is an advantage of record types over Cartesian product types?

9.8. (Requires knowledge of C or C++) C and C++ support records. How would you write type {name: String, size: Integer} in C or C++? Are you required to give the type a name, or can you write it directly as an anonymous record type where it is needed?

9.9. What is the type of each of the following functions? How can you tell? Assume that the addition operation + can only add two integers.

(a) $f(x) = x + x$.

(b) $g(x, y) = \text{head}(x) + y$.

(c) $h(x) = \text{head}(x) + \text{head}(\text{tail}(x))$

(d) $i(x, y) = (y + 1, x + 1)$.

(e) $j(x) = (\text{head}(x), \text{head}(x) + 1)$.

9.10. You can imagine very general constrained types. For example, if T is a type and p is a function of type $T \rightarrow$ Boolean (a *predicate*), it might be possible to create a variable of type (T where p), consisting of all items x of type T such that $p(x)$ is true. Would you advocate allowing an arbitrary predicate to be used as the constraint of a constrained type? Why or why not?

9.11. Java uses four different representations (with four different types) for integers, depending on the number of bytes used to represent the integer. Options are 1, 2, 4 or 8 bytes. What is an advantage of providing four different types of integers? What is a disadvantage as compared with having only one type of integer?

9.9. Bibliographic notes

Pierce [87] covers most aspects of types. Modula-3 [24, 50] is a language with extensive support of subtypes.

Chapter 10

Memory Management

10.1. Memory allocation

As discusssed in Chapter 8, a programming language implementor needs to make choices about representation schemes for basic data structures, such as lists. But regardless of the representation scheme, which determines *how* to represent data, an implementation of a programming language needs to decide *where* in memory to store data, and how to determine when that data is no longer needed, so that the memory that it occupies can be reused. Three schemes for that are *static allocation*, *stack allocation* and *heap allocation.* Each of those allocation schemes has advantages and disadvantages.

10.2. Static allocation

The simplest method of managing memory is to place all data items in fixed memory locations, and never to move them. The memory occupied by each variable or constant is determined when the program is loaded into memory, and cannot change while the program runs. That method, called *static* allocation, was used by the first high level language, Fortran. It had the advantage of simplicity. Also, statically allocated variables were efficient to access, since direct addressing modes provided by processors could be used, and addresses did not need to be computed on the fly.

But with static allocation, the size of a program, and the number and sizes of items stored in the memory, cannot change while the program runs. That severely limits the kinds of data structures and algorithms that can be used. Arrays can be used, as long as it is clear exactly which arrays are needed, and how large they are, when the program is loaded into memory. Linked representation of sequences, on the other hand, relies on the ability to create new list cells on the fly, and so cannot be used without some special mechanism for storing the list cells into statically allocated memory. Fortran is very successful for certain kinds of numerical computing that need speed and that rely mainly on simple data structures such as arrays, but it is not recommended for programs that need sophisticated data structures, or that need to get more memory as the program runs.[1]

[1] Even early versions of Fortran did provide for a primitive way to get more memory, but all memory management was done by the program itself on what appeared to be a blank array of memory.

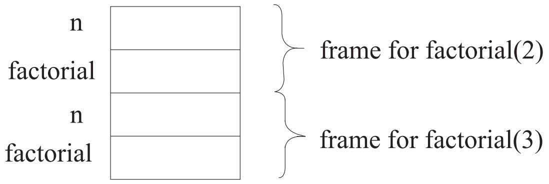
10.3. Dynamic allocation: the run-time stack

Early versions of Fortran did not allow the use of recursion, where a subprogram can use itself. To do that, the subprogram must actually use a copy of itself, where each copy has different variables, to prevent one copy from interfering with what another copy is doing. As more and more copies of a subprogram are created, more and more variables must also be created. Clearly, static allocation cannot be used for that.

Algol 60 does allow recursion, and that required its implementors to find a way to allocate memory on the fly. The solution that they used is to store variables in an area of memory called the *run-time stack*, or simply the stack for short. The stack is broken into *frames* of various sizes, each frame holding variables for one copy of a subprogram. (The instructions of the subprigrams are not themselves copied, since they never change.) Consider the following Algol 60 function, which computes factorials. It is called a procedure because all subprograms are called procedures in Algol 60. The result is stored into a variable that has the same name as the subprogram.

```
integer procedure factorial(n);
  integer n;
  begin
    if n = 0
      then factorial := 1
      else factorial := n * factorial(n-1)
  end
```

When computation of factorial(3) begins, a frame is allocated for it in the run-time stack, holding variables n and factorial. Then computation of factorial(2) is started, since its result is needed by factorial(3). A new frame is created for that copy of factorial, and the run-time stack looks like this, with the more recently created frame shown on the top.



Next factorial(1) is required, and then factorial(0). At its peak, the run-time stack holds four frames for copies of function factorial. When a subprogram is finished, it no longer needs its variables, so its frame can be removed from the stack. Frames are added to the top of the stack, and always deleted from the top.

Use of a stack for holding variables can be a considerable advantage over static allocation, even when recursion is not used. Suppose several subprograms are defined, each requiring an array of 1000 bytes. With static allocation, 1000 bytes must be given permanently to each subprogram, and those bytes will be allocated even when the subprogram is not running. With stack allocation, memory is only allocated for each array when the subprogram that needs that array starts running, and that memory is recovered for reuse as soon as the subprogram is done. So the run-time stack allows the memory for those arrays to be shared.

But there is another advantage that is sometimes even more important. With static allocation, a program is intimately tied to the variables that it uses. If you want to run two copies of a program at once, you have no choice but to create two copies of both the program and the data. But when variables are stored in a stack, subprograms get their data from their own stack frames. To have two copies of a program running at once only requires that there be two copies of the data; the memory occupied by the programs themselves can be shared. For example, a large piece of software might be in use simultaneously by two users, or in two windows by one user. The operating system only needs to load one copy of its machine code into memory. Libraries similarly can be loaded only once, and shared by several concurrent programs. Programs or subprograms that can be run by several different programs at once are called *reentrant*.

Modern operating systems allow a given process to run several *threads* that are similar to several concurrently executing programs, but that share some memory, and that contribute to one overall goal. Different threads have separate run-time stacks, so that they do not interfere with one another. More and more, computers are being designed to take advantage of multithreading, and use of threads is constantly growing. Even modern versions of Fortran no longer rely on static allocation because of the advantages offered by the run-time stack.

10.4. Dynamic allocation: the heap

The use of a run-time stack allows a program to grow and shrink, but it has a fairly severe limitation. The memory allocated by a subprogram is always lost when that subprogram finishes, because the subprogram's frame is removed from the stack when the subprogram is done. It is impossible for a subprogram to create something that needs to be stored in newly allocated memory, and to let that thing continue to exist when the subprogram returns to its caller. Using stack allocation, it is impossible to write a function that takes a list, such as $[2, 3, 4]$, and returns a longer list, such as $[1, 2, 3, 4]$, with another value added to the front, since the longer list must make use of newly allocated memory.

Heap allocation solves that problem. The *heap* is an area of memory that is initially unused by a program. It is managed by a heap memory manager, which can either be part of the library or part of the run-time support, depending on whether it is called explicitly or implicitly. A heap memory manager accepts requests to allocate memory, and *might* also accept requests to deallocate memory, so that the memory can be recycled and allocated again. An example of a heap memory manager is the one provided in the C standard library. To allocate a block of n bytes of memory, you call function malloc with n as a parameter. Expression malloc(n) allocates n bytes of memory from the heap and returns the address of the first of those n bytes. To recycle those bytes, you call procedure free, passing it the memory address given to you by malloc.

Managing the heap

Malloc not only needs to allocate memory, but it must keep track of which parts of memory are available to be allocated. Different versions of malloc behave a little differently, but all have some similarities, and we give a rough approximation of what malloc does.

One approach is to keep several *free space lists* of free blocks of memory. They are linked lists, where the pointers that link the lists together are stored in the memory blocks themselves. A list is kept for memory blocks of 16 bytes, another for blocks of 32 bytes, another for 64 bytes, etc., using higher and higher powers of two. A simple version of malloc always allocates a block whose size is a power of two, allocating 64 bytes when 40 are requested.

When malloc is called, it checks for a block in a free space list of an appropriate size. If it finds a block, it allocates that block, removing it from the front of the list. If it does not find a block of the appropriate size, it needs to get a block from a free space list with larger pieces. It takes that larger block and divides it in half. If no larger blocks are available, malloc asks for more memory from the operating system, adding one large block to one of its free space lists.

A call to malloc(n) actually allocates a little more than n bytes. For our simple version of malloc, a number of bytes m is selected such that $m \geq n + 4$ and m is a power of 2. Malloc places the number m in four bytes of memory just before the block whose address is returned. When a program calls free to return this memory to the heap manager, the parameter passed to free is the memory address that was returned by a call to malloc. Free looks in the memory immediately before that address, and finds the number m of bytes that were allocated. It then places this block at the front of the appropriate free space list.

Fragmentation

This simple memory allocator suffers from a problem. Occasionally, it must chop a large block into smaller blocks. If it never does the opposite operation, combining smaller blocks into larger blocks, then over time the average size of the available blocks tends to shrink. The memory allocator can end up in a state where it has a great deal of memory available, but almost all of it is in very small blocks. A request to allocate a large block might have to be denied, causing the program to run out of memory. This problem is called *fragmentation* of the memory, and is a serious one.

Fragmentation can be partially remedied by schemes that combine small blocks into larger blocks. For example, if free space lists are kept in ascending order by address, then it is easy to discover when two adjacent memory blocks are both in the same free space list. When they are, they can be combined, and moved into a list of larger blocks. That scheme somewhat alleviates, but does not fully solve, the problem of fragmentation. It is still possible to have a great deal of free space cut up into small blocks, with used space in between the free blocks, which makes combining adjacent free blocks impossible. Fortunately, fragmentation can be eliminated by a more sophisticated approach to memory management: garbage collection, discusses in the next section.

Difficulties with heap allocation

Suppose that a program mistakenly frees a block of memory that it still needs to use. What happens to the program? The potential problems are endless. The freed block will be placed into a free space list. Later, it might be allocated again. At that time, the same memory will be in use for two different purposes: the original purpose, for which it was not really finished, and the new purpose, for which it has just been allocated. The two uses interfere with one another, and chaos ensues. On the other hand, before the block is allocated again, it is both in use by the program and part of a free space list. If the program alters the memory block before it is allocated again, the change might disrupt the structure of a free space list (remember that the pointers that link the free space list are in the free memory), causing eventual chaos when the memory allocator attempts to use the damaged list. It is so difficult to predict what will happen if memory is freed prematurely that the only reasonable thing for a language definition to do is to say that a program enters a semantic black hole, with all future behavior undefined, if it ever deallocates a block of memory prematurely, and later tries to make use of that block of memory.[2]

A pointer that points to memory that has been deallocated is called a *dangling pointer*. Experienced programmers who make use of the heap are well aware of the problems presented by dangling pointers. Freeing memory prematurely is a surprisingly easy mistake to make. What is worse, when it is made, this kind of error is typically very difficult to localize and fix. A bug that shows up in one module might have been caused in almost any other module of the program. The error in logic occurs at the point where the memory is deallocated prematurely, but the program enters a semantic black hole only at the point (possibly much later) where it uses the deallocated memory. Even worse, it is a general characteristic of semantic black holes that entering one can yield very strange results that are difficult to understand, and that only show up far from the point where the black hole was entered.

10.5. Garbage collection

Most modern programming language implementations avoid the memory deallocation problem altogether by not allowing a program to free memory that it has allocated. Instead, the system takes on the responsibility for identifying and freeing memory that is no longer in use. The part of the run-time support responsible for that is called a *garbage collector*. One problem faced by a garbage collector is determining where the unused memory is. There are two commonly used schemes for doing that.

Using reference counts

One scheme is to use *reference counts*. Each block of memory has some extra memory allocated with it, holding a count of the number of pointers currently held by the program that point to this block. Each time a new pointer to a block is created, the reference count of that block is increased, and each time a pointer is destroyed, the reference count is decreased. When the reference count becomes zero, the block is inaccessible, and it is moved to a free space list.

Reference counts are relatively straightforward to implement, but they suffer from some difficulties. One is that reference count management is *pervasive*. It must be done every time a pointer is created or destroyed. A program that makes frequent use of pointers will be slowed down by the need to manage reference counts. A more serious difficulty is that cyclic structures cannot be recovered. If two blocks contain pointers to one another, then their reference counts will remain positive, even when both blocks become inaccessible to the program. Finally there is the problem of the need to store the reference counts. Extra memory must be used, which increases the overall memory utilization of the program.

Mark/sweep garbage collection

An alternative approach is a *mark/sweep garbage collector*, which is run periodically, when available memory is low. Each block of data has an associated *mark bit*, which is used

[2] A careful definition needs to distinguish a *logical* block of memory, which is allocated and deallocated only once, from a *physical* block of memory, which might be allocated several times, each time holding another logical block of memory. The program enters a semantic black hole at the point where it makes use of a logical block of memory that has been deallocated.

to indicate whether that block is in use. Initially, all of the mark bits are set to 0. The garbage collector begins with a *mark phase*, during which it scans through all memory that is accessible to the program, marking it all by setting the mark bits to 1 for the blocks that are encountered. To get going, the garbage collector marks all variables that are kept in the run-time stack. When a variable refers to a data structure via a pointer, that data structure is marked. The mark phase follows pointers, working its way through data structures. Of course, if a block of memory is encountered that has already been marked, it is not necessary to look inside that block again, or to follow any of the pointers that it contains.

After the mark phase, the garbage collector does a *sweep phase*, also called as *collect phase*, in which it scans the entire heap, freeing all blocks that have not been marked.

Mark/sweep garbage collectors have some advantages over other schemes. They can collect cyclic structures. There is no need to keep track of reference counts while the program runs. The program just runs as if it does not care whether unused memory is ever deallocated. Finally, only one bit per block is needed for the marking, so memory requirements are low. A disadvantage is that the garbage collector needs to be able to recognize pointers (and to distinguish them from integers) so that it knows to follow them during the mark phase. Tags on data items will do the job.

Avoiding fragmentation

One of the big advantages of mark/sweep garbage collectors is that they can be designed to be very effective at avoiding fragmentation. A *relocating* garbage collector can move blocks to new locations, *compacting* them so that there is no unused memory between the used blocks. Then the unused portion is a single, contiguous block of memory. The biggest technical problem in performing the compactification is *relocation*: when a block of memory is moved, all pointers to it that are in other blocks must be changed to point to the new location. That can be done during the sweep phase, as long as information is available that tells the garbage collector how to change each pointer. The main issue is how the relocated pointers are found.

A simple relocating garbage collector keeps two heaps. At any given time, one of the heaps is in use and the other is fallow. Suppose that, at the start of a given garbage collection, heap A is in use and heap B is fallow. During the mark phase, the garbage collector copies each marked block from heap A to heap B, putting the blocks next to one another to eliminate holes. It leaves behind, in the old location of a block (in heap A), the address of its new location (in heap B). Think of it as like a forwarding address telling where each block now resides. At the end of the mark phase, the blocks in heap B contain the same bit patterns as they did when they were in heap A, since they are exact copies. Any pointers in those blocks are memory addresses in heap A. Each pointer in heap B needs to be changed to point the correct block in heap B. The garbage collector performs a *relocation* phase, during which it sweeps over heap B, replacing each pointer that points into heap A by the new location of the block to which the pointer refers. That new location is found in heap A, where it was put earlier. (It is the forwarding address.) Relocation is also done on the run-time stack, and all other places where pointers might be stored. Finally, heap A is marked fallow and heap B is marked in-use.

Of course, having two heaps, one of which is always not being used, doubles the memory requirements. Both time and memory can be saved by some simple improvements. Things stored in the heap tend to be of two kinds: long-lived things that are created early and stick around, and short-lived things that are created, used and abandoned quickly. A memory manager can keep a few blocks of memory for the heap. As values are moved during relocation (from one block to another), they are preferentially moved into lower numbered blocks. The result is that long-lived things tend to move into those blocks and stay there, leaving higher-numbered blocks for the short-lived things. During a given garbage collection, the lower numbered blocks might not be scanned at all under the presumption that almost all things in those blocks will still be accessible. Occasionally, the low numbered blocks need to be examined. But not every block needs to be compactified during every garbage collection; some can simply be left as is, so that only a few extra fallow block will suffice to manage relocation.

Time out for garbage collection

Mark/sweep garbage collectors have one drawback. During garbage collection, the program cannot run. So there are brief periods during which nothing appears to happen. For most applications, that is not a problem. It is certainly no worse than the time taken by a multiprogrammed processor switching to another process or thread. But for time-critical applications, such as real-time applications that require immediate responses to requests, a mark/sweep garbage collector would be unacceptable.

Impact of garbage collectors

Programs that need to free memory explicitly can have a significant fraction of their lines (and programmer time) devoted to determining when to free each item, making memory management an unpleasant and difficult job for software developers. Languages whose implementations provide automatic garbage collectors tend to be much easier to use.

Lisp and Java are examples of languages all of whose implementations are equipped with garbage collectors. Typically, relocating mark/sweep garbage collectors are used. Most new languages require garbage-collecting implementations because of the enormous convenience that they provide.

10.6. Summary of terminology and concepts

A language implementation must provide some form of memory management. Very early implementations provided only static allocation, done when the program is loaded into memory. Static allocation does not allow a program to grow, and does not allow recursion. Stack allocation is more versatile, and allows recursion, but is still restrictive in how allocated memory is used. Heap allocation is very versatile, but needs special care for deallocation. Usually, the preferred method of doing deallocation in the heap is for the language implementation to provide a garbage collector.

Terminology

Compactification Compactification is a way of avoiding fragmentation. It moves data items that are in use to contiguous blocks of memory, making the available memory also contiguous.

Fragmentation Fragmentation is a problem that occurs when the available blocks of memory are small, so that no large blocks can be allocated. Without care, fragmentation grows over time, and can paralyze a system.

Garbage collector A garbage collector periodically locates memory that is not being used and recycles it for use again.

Heap allocation Data that is allocated in the heap lives for as long as the program needs it. Memory in the heap is returned to the system for reuse either explicitly by a program or via the action of a garbage collector.

Mark/sweep A mark/sweep garbage collector works in two phases. During the first phase, it marks all memory that is in use by traversing all of the data structures that the program and the language implementation are maintaining. During the second phase, it recovers all unmarked memory.

Reentrant A reentrant program can be run simultaneously by more than one thread, with the machine code shared.

Reference counts Reference counts are one way of determining when memory is no longer in use. But they cannot be used to recover cyclic structures.

Relocation A relocating garbage collector compactifies memory. Since this requires moving blocks to new memory addresses, pointers must be changed. The garbage collection phase that changes the pointers is called the relocation phase.

Run-time stack The run-time stack is an area of memory where local variables of subprograms are stored, and where data and variables are created and destroyed automatically. Variables allocated in the stack are only available as long as the subprogram that created them is still running.

Stack allocation Stack allocation is a scheme that stores variables in a run-time stack. It allows an arbitrary number of copies of a subprogram to be running at the same time.

Static allocation Statically allocated memory is allocated once, when the program is loaded. It is easy to do and efficient to use, but not very flexible, and does not yield reentrant programs.

10.7. Exercises

10.1. Why can't Algol 60 implementations use static allocation like Fortran implementations?

10.2. Some Fortran programmers were unhappy with the introduction of recursion into Fortran. Why do you think they did not want recursion? Do you pay for it even if you do not use it?

10.3. What was the motivation for introduction of the heap?

10.4. What are the advantages of a relocating mark/sweep garbage collector over a garbage collector that uses reference counts?

10.5. What is one piece of software that probably needs to be able to free memory in the heap explicitly, rather than letting a garbage collector do that?

10.6. What is the main advantage of a garbage collector over letting a program manage its own memory?

10.7. What is fragmentation? Why is it a serious problem?

10.8. Our simple relocating garbage collector uses two heaps, one in use and the other fallow. That doubles the memory requirements of a program. What is the reason for keeping two heaps? Why not just (1) mark all accessible blocks, and (2) move all marked blocks to the beginning of the heap? Then only one heap is needed. Can you see any difficulties with that?

10.9. Language implementations that provide a heap and a garbage collector typically still use a run-time stack. Why do you think they don't just dispense with the stack and do all allocation in the heap?

10.8. Bibliographic notes

Boehm [18] describes a garbage collector for C++ programs that allows C++ programmers to avoid freeing memory explicitly. Aho, Lam, Sethi and Ullman [5] provide extensive information on garbage collection techniques.

Part IV

Names, Scope and Modules

Chapter 11

Names and Name Binding

11.1. Giving things names

Effective communication of any form appears to require the ability to give things names. When names exist, we use them. When names do not exist, we invent them. Names are important partly because they allow you to hide details behind a name. It is much easier to refer to a baseball as a baseball rather than as a sphere of tightly wrapped string with a stitched cover. Programming languages are no different, and mechanisms for naming things are among their most important features. You can give names to a wide variety of things in a program, including data items, variables, subprograms, types, modules or your entire program. This chapter explores mechanisms for giving things names. The next chapter looks at ways of controlling what names mean in different parts of a program, so that you can use the same name in more than one place.

11.2. Identifiers

In programming languages, names are usually called *identifiers*. The forms that an identifier can have depend on the language, but typically an identifier can begin with a letter and can contain letters and digits. For example, *abc2* could be an identifier. But there are other conventions for the form of identifiers. Lisp, for example, allows *-e##9aq* to be an identifier. In Cinnameg, the symbol + is an identifier, and can be used to name something. You must consult the documentation of each language to see what an identifier can look like in that language.

11.3. Naming data items

When you give an item a name, you *bind* the name to that item. For example, in Cinnameg you write[1]

[1] Cinnameg uses *full bracketing*. A statement that begins with Let normally ends with %Let, an end-of-Let marker. Similarly, a statement that begins with If ends with %If. But you can use a period to stand for an arbitrary end marker. We use a period in displayed program text and an explicit %Let in program text within a paragraph.

Let i = val.

to indicate that item *val* should be computed, and that identifier i should be bound to that item. So

```
Let root = n^2 - 4.
Let frog = root*(root+1).
```

computes $n^2 - 4$, and calls the result root. Identifier root can then be used in many places; there is no need to recompute the value each time it is used.

11.4. Naming and creation

Sometimes, creating things and giving them names are tied together. For example, in Ada, you write

```
function Sqr(X: Integer) return Integer is
begin
  return X*X;
end;
```

to define a function named Sqr that returns the square of its parameter. You both create the function (and define what it does) and give it a name, Sqr.

What if you want to give a new name to an existing function, rather than creating a new function? Then some other mechanism is required. To make *Square* be another name for *Sqr* in Ada, you write

```
function Square(X: Integer) return Integer renames Sqr;
```

Cinnameg exemplifies a different approach to naming functions that does not tie naming and creation together. Cinnameg expression `(x |-> x*x)` produces, as its value, the function that returns the square of its parameter. The function is anonymous; it is not necessary to give it a name in order to create it. To give that function a name, you use the same mechanism as is used to name any other kind of item, such as `Let sqr = (x |-> x*x) %Let`. To give another name *square* to function *sqr* you write `Let square = sqr %Let`, indicating that identifier square should be bound to the same thing as identifier sqr. By separating creation and naming of subprograms, Cinnameg avoids the need for introducing special notation that separates them.

But giving names to anonymous functions is not the most natural way to write function definitions, and most programmers would prefer, most of the time, to use a notation that looks something like the Ada notation. That can be accomplished by introducing abbreviations. In Cinnameg, definition `Let f(`x`) = `e` %Let` abbreviates `Let f = (`x` |-> `e`) %Let`. So an alternative definition of the sqr function is `Let sqr(x) = x*x %Let`. Abbreviations are merely a convenience, and you are free to avoid them if you prefer to.

Naming variables and arrays

Variables are also typically created and given a name in one statement. For example, C++ statement `int frog;` creates a new variable and names it frog. Suppose that you want, for some reason, to have a new name for an old variable.[2] A first attempt is

```
int kermit = frog;
```

But, since naming and creation of variables are tied together, that actually creates a new variable named kermit, and just copies the value of frog into kermit. Now you have two different variables. To make kermit and frog name exactly the same variable in C++, you can write the following, using another mechanism that is provided to suppress creation of a new variable.

```
int& kermit = frog;
```

For variables, Cinnameg again uses the standard naming mechanism together with an expression that creates a new variable. In Cinnameg, a variable is called a *box*, since it can be thought of as a box that can hold something inside it. Expression `[:0:]` yields, as its value, a new box with initial content 0, and statement `Let frog = [:0:] %Let` binds frog to that new box. Then `Let kermit = frog %Let` makes kermit another name for the same box, just as it would for any other kind of thing.

11.5. Names in the implementation

We have seen that binding identifiers is an important aspect of programming languages. But where, in an implementation, are those bindings actually stored? First, consider compilers. A compiler typically translates a program from a higher level language into machine language. In machine language, variables and subprograms do not require names: you refer to them by their memory addresses. A compiler maintains a table, called the *symbol table*, in which each name is associated with some information that tells where its value will be stored in the memory. Each time the compiler sees a name, it looks up that name in the symbol table and uses the associated address in the machine language program. The upshot of that is that, when the program is running, most information about names is gone.

That can cause tremendous difficulty for debugging programs. What if, during the course of running a program, a programmer wants to inspect the contents of a particular variable? With very primitive systems, he or she cannot refer to the variable by name, since the name is not anywhere in the program. Instead, the programmer must try to guess where the compiler placed the variable. Linkers can be asked to provide *maps* to the locations of variables and subprograms, but they are clumsy to use. Most modern systems provide a way to put information about name bindings back into machine language programs, so that programmers can find variables and subprograms by name. The machine language program is augmented by a symbol table that is similar in intent to the compiler's symbol table and to the linker map. It tells where each thing is located. A symbolic debugger can then locate each thing by its name.

Interpreters face a similar problem. Some interpreters carry symbol tables with them, looking up each variable, by name, when it is used. But that tends to be inefficient, and most interpreters use an abstract machine language that shares the characteristic with machine language that things are referred to by where they are stored, not by names. Those interpreters typically carry a symbol table with them, allowing the programmer to find the location and value of something by giving its name.

[2] Having two names for the same variable is called *aliasing*, and is usually discouraged. But there can be situations where it makes sense to have two names for the same thing, and you might want to have the ability to do it.

11.6. Overloading

My dictionary lists 67 definitions for the word “play”. A word that has more than one meaning is said to be *overloaded*. Some programming languages similarly give more than one meaning to a single symbol. Even the earliest high level language, Fortran, allows overloading to a very limited extent. The symbol + is used in Fortran for two purposes: it can indicate addition of integers or addition of floating-point (real) numbers. To a Fortran compiler, the two operations are quite different, since they are implemented by different machine-language instructions. To the programmer, the two concepts are really the same, so it is convenient to have a single name for them. Overloading is the most useful to a programmer in cases where the programmer thinks of two or more ideas as being the same (and hence deserving of the same name) but a program must treat those ideas differently. For example, the length of a string, the length of a linked list and the length of an array of integers might need to be implemented differently from one another. But it is convenient for a programmer to call them all *length*, since they all represent the same concept.

Fortran does not allow the programmer to bind one identifier in more than one way at a time. The overloading is restricted to the primitive operations provided by the language. Some languages, however, allow the programmer to overload identifiers. C++ and Java, for example, allow one identifier to be used to name many different subprograms. It is required that the compiler be able to tell which binding is intended at any given use of an identifier. That is accomplished by insisting that each subprogram with the same name have a different list of parameter types, since then the desired definition can be inferred by looking at the parameters. For example, in C++ you can define two different functions, each called *sqr*, one squaring an integer and the other squaring a real number (of type **double** in C++).

```
int sqr(int x) {return x*x;}
double sqr(double x) {return x*x;}
```

Now, if you write expression sqr(34.5), it is clear from the parameter which definition of identifier sqr is being used.

Cinnameg allows overloading, with the only requirement being that you are not allowed to use the same name for two items of the exact same type. The compiler determines which definition you want based on the type, and it determines the type based on how you use the identifier.

11.7. Pattern matching

Suppose that a mathematician wants to convey the information that variable x must have value 3. An obvious and simple approach is to say directly that $x = 3$. But sometimes it is advantageous to use an indirect definition, such as $2x + 5 = 11$, where x is understood to be a solution to this equation. Both definitions indicate that x is 3, but the second is implicit because you need to solve the equation.

Cinnameg’s **Let** mechanism is explicit: you compute or create a data item and give that item a name. But some languages also provide implicit ways of giving things names, called *pattern matching*, typically involving solving simple equations. An example of a pattern matching feature is Cinnameg’s match statement. Statement

```
Match n + 1 = m.
```

tells the compiler to solve equation $n + 1 = m$ for n. The solution, of course, is $n = m - 1$. If m is 5 then n is bound to 4.

Sometimes you have a complex item and you want to get some of its parts, and give them names. For example, suppose you have an ordered pair p, and you want to give names to the left-hand and right-hand members of p. Pattern matching provides a convenient way of doing that. Cinnameg statement

```
Match (a,b) = p.
```

binds identifiers a and b to the left-hand and right-hand members of p, respectively. For example, if ordered pair p is $(3, 4)$ then equation $(a, b) = p$ clearly has exactly one solution, $a = 3$ and $b = 4$. The same kind of thing can be used for lists. For example, suppose that x is the list $[3, 6, 2]$. Then

```
Match [fst,snd,thrd] = x.
```

must bind fst to 3, snd to 6 and thrd to 2.

Pattern matching is often the method of choice for extracting parts of a complex item. It can be much easier to read (and hence less likely to be written incorrectly) than any other method of achieving the same end.

Symmetry and asymmetry

The pattern matching facility provided by Cinnameg is asymmetric, and only solves for variables on the left-hand side of an equation. That simplifies solving equations, but it limits generality. Some languages, such as Prolog, support a symmetric form of pattern matching called *unification*. For example, unification can solve equation $(2, n) = (m, 3)$ by binding $n = 3$ and $m = 2$. Notice that variables to be solved for occur on both sides of the equation. Unification is covered in more detail in Chapter 28.

11.8. Summary of terminology and concepts

Naming things is fundamental to programming languages. It is one of the information hiding features that allows you to suppress irrelevant details. Instead of referring to an item by what it is, or by how you create it or find it, you just call it by its name.

There is more than one way to give a name to something. A simple approach is explicitly to bind a name to a value. Pattern matching is an implicit naming mechanism, and relies on letting the language implementation solve an equation and bind names to the equation's solution.

Some languages provide mechanisms where you simultaneously create a new item and give it a name. Examples are variable creation/naming and subprogram creation/naming. Other languages prefer to provide a basic creation mechanism that creates anonymous items, and to use a separate mechanism for attaching a name to an anonymous item, as the Cinnameg example showed.

It is convenient to be able to overload names, since two things that a programmer thinks of as the same might need to be handled differently by a program.

Terminology

Binding When you give an item i a name n, you are said to bind name n to item i.

Identifier An identifier is a name. Different languages have different requirements on the form of an identifier.

Overloading An identifier that has more than one binding is overloaded.

Pattern matching Pattern matching finds solutions of simple equations, and is typically used to extract parts of a complex item and give names to the parts.

Symbol table The symbol table is a table where information about identifiers is stored in a compiler or interpreter.

Unification Unification is a symmetric form of pattern matching. The idea is to bind variables, possibly on both sides of an equation, to make the equation true.

11.9. Exercises

11.1. Explain how the ability to give things names is one of a programming language's encapsulation, or information-hiding, mechanisms.

11.2. What is an advantage of separating naming from creation of things like subprograms and variables?

11.3. After performing C++ program fragment

```
int x = 1;
int& y = x;
y = 2;
```

what is the value of x? Explain.

11.4. How can a debugger know that the second variable in a function's frame of the run-time stack is called peach? Where is that information stored?

11.5. Give motivation for allowing programmers to overload identifiers in a programming language. Can you think of a reason why a language designer might prefer not to allow overloading?

11.6. What is the solution to pattern match equation $[x, (y + 1, z)] = [7, (99, 41)]$?

11.7. Write a Cinnameg pattern match that binds r to the right-hand member of an ordered pair p. Cinnameg allows you to use a question mark for an anonymous variable, which is one whose value you don't want to name.

11.8. Write a Cinnameg pattern match that binds t to the third member of a quadruple called q. (So q has the form (a, b, c, d) for some a, b, c and d.) Use a question mark for an anonymous, or don't-care, variable.

11.9. In Cinnameg, binary operators are an abbreviation. Expression $a + b$ abbreviates $(+)(a, b)$, an application of function + to the ordered pair (a, b). Using that abbreviation, show what definition

```
Let x + y = add(x,y).
```

abbreviates. Carry the expansion out to a point where there is just an identifier on the left-hand side of the = sign.

11.10. Bibliographic notes

Pattern matching is used in functional languages such as Standard ML [86] and Haskell [103] to support a simple way of defining functions. SNOBOL [46] and its descendent Icon [48] use pattern matching as a central part of the way they compute.

C++ [102] allows operators and functions to be overloaded, and makes heavy use of that in its standard library. Operator `<<` has about 25 definitions.

Chapter 12

Scope, Visibility and Modules

12.1. Limiting scope

You do not want to be able to use variable name x in only one place in a large program. That would be very unpleasant for one programmer, and it would be unbearable for a team of programmers working on a large piece of software. Some mechanism needs to be provided by the programming language to limit where each name binding is visible. That part of a program where a binding is visible is called the *scope* of the binding.

Languages typically have two different mechanisms for controlling scope, one for dealing with scope on a small scale, within individual subprograms or other small units, and a separate mechanism for controlling scope on a larger scale, for example on the scale of the entire program. This chapter explores both small-scale and large-scale mechanisms.

12.2. The scope of a binding: small scale

Details of the rules for determining scope differ from one language to another, but there are some general similarities. Most languages have some notion of a scope unit, consisting of a section of a program that we will call a *block*. The *lexical scope rule* indicates that, when an identifier is bound, its binding is visible only within the innermost block that contains the binding. Typically, the binding is further restricted to be visible only from the point where the binding is made to the end of the innermost block that contains the binding; but some languages allow a binding to be visible, in some circumstances, before the point where the binding is made. In either case, visibility under lexical scoping is determined by the (static) structure of the program.

Java and C++, for example, have a similar notion of a block; it is either an entire subprogram (or method definition) or any part of a subprogram that is enclosed in braces ({...}).[1] Figure 12.1 shows a definition of subprogram *sort* written in C++. In sort, five local identifiers, a, n, i, j and t, are bound; a is bound to (or names) an array and each of the others to a variable. None of the local bindings are visible outside the definition of sort, since the entire subprogram is a block. The scope of the bindings of a and n is the entire subprogram. The binding of i is visible in the entire inside of the definition. The scope of

[1] There is an additional rule concerning for-loops. A for-loop constitutes a block, so that variables declared in the heading of a for-loop are not visible outside the loop.

```
void sort(int a[], int n)
{
  int i = 1;
  while(i < n) {
    int j = i;
    while(j > 0 && a[j] < a[j-1]) {
      int t = a[i];
      a[i] = a[j];
      a[j] = t;
      j = j - 1;
    }
    i = i + 1;
  }
}
```

Figure 12.1: **A C++ definition of a procedure that sorts an array, given the array and its size.** There are four blocks, one being the entire definition, and the three others the sections delimited by $\{\ldots\}$. The blocks overlap one another, since one block is inside another.

the binding of j is the body of the outer while-loop, and the scope of the binding of t is the body of the inner while-loop.

It is common convention to refer to the scope of a binding of an identifier as the scope of the identifier itself. So you might say that the scope of t in procedure sort is the inside of the inner while-loop. Of course, it is the *binding* whose scope you are really talking about. Such twists of terminology are common when they make the terminology less cumbersome, but you need to realize what is really being said.

Nested subprograms

Some languages allow a subprogram definition to be written inside another subprogram definition. Figure 12.2 shows an example in Pascal. Notice that the nested subprograms can make use of bindings that are in scope at the point where they are defined. For example, the definition of swapA occurs in the scope of A. So the scope of A extends across the definition of swapA, and swapA can use A.

Shadowing

Suppose that you attempt to rebind an identifier within its scope. That is not the same as changing the value of a variable. Think, instead, of binding an identifier to a different variable. What should happen? Typically, the old binding is not replaced, but instead is hidden, or *shadowed*, by the new binding, leaving a hole in the scope of the old binding. The C++ program fragment in Figure 12.3 illustrates that. The inner block contains a binding of x that shadows the binding of x in the outer block, and, within the inner block, identifier x is presumed to refer to the variable called x in that block. So the scope of the outer x has a hole in it, where the shadowing occurs.

```
type intarray = array[1..100] of integer;
procedure sort(A: intarray);
  var k: integer;
  procedure swapA(var i,j: integer);
  var temp: integer;
  begin
    temp := A[i];
    A[i] := A[j];
    A[j] := temp
  end;
  procedure insert(n: integer);
  var i: integer;
  begin
    i := n;
    while (i > 1) and (A[i] < A[i-1]) do
    begin
      swapA(i, i-1);
      i := i - 1
    end
  end;
  begin
    for k := 2 to 100 do insert(k)
  end;
```

Figure 12.2: **An example of nested subprograms, written in Pascal.** Procedure sort sorts an array of 100 integers. It contains definitions of procedures swapA and insert, whose scope is restricted to the interior of the sort procedure. Notice that the nested procedures can make use of *A*, since it is in scope at the point where they are defined. (The word var in the heading of swap indicates call-by-reference, discussed in Chapter 16.)

```
{
  int x = 3;
  int y;
  {
    int x;
    ...
    x = 2;
    ...
  }
  cout << x; // comment:  print x
}
```

Figure 12.3: **A program fragment written in C++ that illustrates shadowing.** The binding of x in the inner block shadows the binding of x in the outer block. The scope of the binding of x in the outer block has a hole in it. The last line prints 3, since it is not in the hole, and so prints the value of the outer x.

Shadowing is usually not a good idea. If you have two variables with the same name, you can easily become confused. When a person reads a program, he or she cannot be expected to see every aspect of the program, and might not notice that a binding becomes shadowed. So shadowing can be misleading. Some languages, including Java and Cinnameg, disallow shadowing of one local binding by another.

Neither Java nor Cinnameg forbids shadowing a global binding (one done outside a subprogram) by a local one. Forbidding all shadowing would create a problem. Most programs make use of libraries, and a programmer rarely knows all things that belong to a library. If shadowing of the bindings in the library is not permitted, then a programmer must be sure to avoid using names of things that he or she is not even aware exist. Even worse, a library might grow over time as new functions and procedures are added, so the programmer would need to avoid names that might be added to the library in the future! For precisely this reason, most languages allow some shadowing. Extensive use of shadowing, however, is usually strongly discouraged when it can be avoided, so that human readers will not be confused.

12.3. The scope of a binding: large scale

Figure 12.2 illustrates how the lexical scope rule can be used to limit where subprograms are visible. Some older languages, such as Pascal and Algol 60, use the lexical scope mechanism to control scope for an entire program. The program is written as one large main procedure with other procedures written inside it. Procedures are further hidden by embedding them inside other procedures. The idea is that you put the definition of one procedure inside another procedure that needs to use it.

Unfortunately, large-scale lexical scoping does not work as well as would be hoped. The problems associated with managing names on a program-wide level are quite different from those that occur on a small scale, within a single subprogram definition. Imagine an implementation of a data structure, such as a linked list, that supports procedures *insert* and *remove*. In order to implement those two subprograms, you write a third (helper)

```
public class Example {
  private int var1;
  public static final int size = 4;
  ...
}
```

Figure 12.4: **The beginning of a Java class.** Each name is marked according to visibility (public or private).

subprogram, *locate*, that should only be used by insert and remove. Using the local scoping rule, there is no natural place to write locate. If you write it inside insert, for example, then it is only accessible to insert, not to remove. If you define it beside insert and remove, so that it is accessible to both, then it becomes accessible to all parts of the program where insert and remove are accessible, and you have not hidden it at all. You cannot solve this by putting insert, remove and locate all inside one larger procedure, because then they can only be used by calling the larger procedure, and that does not make any sense. The idea is to perform insertions by calling insert and to perform removals by calling remove.

Failure to control where a subprogram can be seen can be problematic in a large piece of software. If you make a subprogram definition available everywhere, then when you change that subprogram, you need to check everywhere to see how that change affects the rest of the program. You need to do that check even if, in fact, the subprogram is not used anywhere else, just because, until you do the check, you do not know that it is not used.

Since lexical scoping works poorly, the scope of a definition on a large scale is usually handled differently. Most languages have some variation on the following scheme for managing names at a large scale. A program is broken into *modules*, or *packages*. Each module has two forms of control over the scope of identifier bindings: import control and export control. Import control allows module A to control which bindings from other modules are visible within module A, and export control allows it to control which identifier bindings in module A are allowed to become visible in other modules.

Controlling exports

Exports can be controlled by marking certain things as *private*, or not exportable, and other things *public*, or exportable. That is the approach taken in Java, for example. Each nonlocal identifier binding is marked according to its visibility. A Java definition might begin as shown in Figure 12.4. Export controls allow you to know what can and cannot be used in other modules. If you make a change to a private variable or to the definition of a private function or procedure, you know that doing so cannot affect how other modules work, unless you indirectly affect something else that is public.

There is an alternative approach to indicating export controls that does not require an explicit visibility mark on each definition. A module can be broken into two parts, the *interface* and the *implementation*. The interface describes exactly what is exported, and anything that is written in the implementation but not mentioned in the interface is not exported. That is the approach taken by Ada. Figure 12.5 shows an example of an Ada package broken into an interface and implementation (called the *body* in Ada). The example package exports only one function, but a package can, in general, export many things. Notice that function DivisibleBy is not exported simply because it is not mentioned

```
-- interface:

package Prime is
  function IsPrime(N: Positive) return Boolean;
end Prime;

-- implementation:

package body Prime is

  function DivisibleBy(N,K: Positive) return Boolean is
  begin
    return N mod K = 0;
  end DivisibleBy;

  function IsPrime(N: Positive) return Boolean is
    K: Positive;
  begin
    If N < 2 then return False;
    else
      K := 2;
      while K < N loop
        if DivisibleBy(N,K)
          then return False;
        end if;
        K := K + 1;
      end loop;
      return True;
    end if;
  end IsPrime;
end Prime;
```

Figure 12.5: **A module in Ada.** The module has two parts: the interface, which tells what is exported, and the implementation. The binding of identifier DivisibleBy to a function is not exported by this module, since it only occurs in the implementation. Lines beginning -- are comments, and := indicates an assignment to a variable.

in the interface part of the package.

Controlling imports

There can be, and usually are, many modules in a piece of software. Module A might need to know public definitions made in modules B and C, but not those made in D or E. It makes good sense for module A to see only those definitions that it needs. What is usually done is that module A explicitly *imports* those modules that it needs. In Ada, you can import the Prime module by writing

```
with Prime;
```

Some languages provide for close control over what gets imported. In Cinnameg, for example,

```
Import select, selectSuffix from "collect/search".
```

imports only functions select and selectSuffix from module collect/search.

12.4. Naming on a large scale

Large libraries consist of many modules, and can be built up over time by the accumulated contributions of different authors. It is entirely possible, even likely, that two different modules in a library each use the same name, for different things. For examples, modules A and B might each have a function called getSize. What happens if a program wants to use both of those modules, and needs to use the getSize function from each one? Allowing overloading can help, but only if getSize has different types in the two modules. What if getSize has the same type in both?

Ada offers a way out that is typical of several other languages as well, including Java and Cinnameg. The idea is that, by default, you must explicitly state from which module each name is drawn. The binding of *fruit* in module *cake* is called *cake.fruit*, but the binding of *fruit* in module *salad* is called *salad.fruit.* There is no confusion between the two. A name that has both parts is called a *qualified* name.

When you import Ada module Prime, what you actually get is a function whose name is Prime.IsPrime. So a module that wants to know whether an integer is prime might look like the example shown in Figure 12.6(a).

But programmers find it awkward to refer to things by such long names, and the names can get long indeed. To set Y to the square root of X in an Ada program, using the standard square root function, you write

```
Y := Ada.Numerics.Generic_Elementary_Functions.Sqrt(X);
```

That is just intolerable. Ada solves this problem by adding a mechanism to allow programmers to use abbreviated names. In Ada, you write

```
use Ada.Numerics.Generic_Elementary_Functions;
```

to allow all names, such as Sqrt, defined in the indicated package to be used without the package name. To use the Prime package, you would normally write something like what is shown in Figure 12.6(b). You only do that, of course, when there is no conflict between the name bindings of different packages.

```
package body Test is
  with Prime;
  procedure Main is
    N: Positive;
  begin
    ...
    if Prime.IsPrime(N) then
    ...
  end Main;
end Test;
```

(a)

```
package body Test is
  with Prime;
  use Prime;
  procedure Main is
    ...
    if IsPrime(N) then
    ...
  end Main;
```

(b)

Figure 12.6: **An Ada module making use of the IsPrime function from module Prime (Figure 12.5).** Version (a) uses the full name of IsPrime, and (b) uses the shortened name, enabled by the `use` directive.

Naming can be used as a way of simulating partial imports. For example, Java uses the word **import** for the same purpose as Ada uses **use**. In Java, directive `import java.util.*` says to allow short names for all things in package java.util. If you only want to use Random from that package, you include directive `import java.util.Random`. Now you can write Random as an abbreviation for java.util.Random. Of course, you can still use the long names for other things in that package, but you have limited what is visible via a shortened name.

12.5. Summary of terminology and concepts

The scope of a name binding is that part of a program that can see the binding. Rules for determining scope at the level of a single unit, such as a subprogram definition, are fairly simple, based on the notion of a block. A binding is typically visible from the point of its definition in a block until the end of the block. Shadowing, however, can introduce holes into the scope of a binding.

The local scope rules are inadequate for larger units, such as the entire program. Rules for determining the scope of a name on a large scale typically involve modules, export controls and import controls.

Because software can contain components written by different organizations, it is important to use long names. An insert procedure written by the Widget division of the Acme software firm might be called Acme.Widget.insert. But programmers find long names cumbersome, so an additional mechanism needs to be provided for shortening the names (for example, to insert) when that does not lead to conflicts.

Terminology

Block A block is a unit of a definition or other section of a program that delimits the scope of bindings.

Export control A module can control which bindings it exports. That makes it easier to modify a module, since you can be sure that certain changes cannot affect what happens in other modules.

Implementation The implementation of a module is the section where definitions are made, and details on how things work are provided.

Import control A module can control which bindings it imports from other modules. That makes it impossible to use something by mistake from a module that you do not want to use at all, and so reduces the number of errors that you can make.

Interface The interface of a module tells which bindings made in the module can be made visible in other modules.

Lexical scoping Lexical scoping is the name of the scope rule for the local level, where a binding is visible within the innermost block where the binding is made.

Module A module is a basic unit of a program in which some identifier bindings are kept private and others are made public.

Package Package is another word for module.

Private definition A private definition is a binding that is made inside a module, and that is only visible within that module.

Public definition A public definition is a binding that is made inside a module, and that can be made visible outside of the module.

Qualified name A name that includes its module, such as Queue.insert, is called a qualified name.

Scope The part of a program where a given identifier binding is known, or visible, is called the scope of that binding.

Shadowing When a binding of an identifier i is replaced by a different binding of i in a context where the first binding would normally be visible, the second binding shadows the first binding. The first binding is not destroyed, but is merely hidden within the scope of the shadowing binding.

12.6. Exercises

12.1. Briefly explain the lexical scoping rule.

12.2. What is shadowing, and how can it occur?

12.3. (This question assumes familiarity with either C++ or Java or C#.) Some languages, such as Algol 60, allow you to write one subprogram definition inside another one. Do C++ or Java or C# (choose just one) allow one method to be defined inside another, in such a way that it can use bindings that are in scope at the point of the definition? If you do not know, try it to see if it works.

12.4. From the Cinnameg documentation, determine the definition of a block that delimits scope. (Try searching for scope in the index.)

12.5. Give two reasons why the lexical scope rule does not work well on the scale of an entire program.

12.6. What is an important motivation for exercising export control in a module? Why not just export everything?

12.7. If a module has just one part (not separate interface and implementation parts) how can export control be managed?

12.8. Why is it important to use long names for things defined in modules?

12.9. Is it also important to use long names for things defined inside subprograms, such as local variables? Explain. Would you recommend names such as sqrt.x inside a function called sqrt?

12.7. Bibliographic notes

Wirth [111] discusses reasoning behind the need for and structure of modules. Early languages that support modules include Modula-2 [112] and CLU [72]. Successors of Modula-2 include Modula-3 [24, 50] and Oberon [113].

Chapter 13

Mechanics of the Run-Time Stack

13.1. Activations

Chapter 10 introduces the run-time stack, or simply the *stack*, an area of memory where local variables of subprograms are stored. The stack is broken into frames, each holding the variables for one call to a subprogram, as well as other information such as an instruction counter indicating which instruction a subprogram is about to perform. The information in a stack frame is called an *activation* of a subprogram.

This chapter explores more detail about the information that is kept in an activation and how a language implementation manages frames and the structure of the stack. Although this chapter fits best with memory management, it requires material on scoping. In some places it requires material covered in later chapters, and you might prefer to come back to it after studying issues in procedural and functional programming.

13.2. Pushing and popping frames

A program needs to be able to add and remove frames in the run-time stack and to locate the top frame, which belongs to the subprogram that is currently running. Since frames are of different sizes, to say that you know where a frame is in memory, you must know both its top and bottom boundaries. So the program evaluator keeps two pointers, the *stack pointer*, pointing to the top of the top frame, and the *frame pointer*, pointing to the bottom of the top frame.

Imagine that subprogram P calls another subprogram Q. A frame holding an activation of Q needs to be pushed on top of the stack, just above a frame holding an activation of P, as shown in Figure 13.1. As part of the calling process, P stores some information into the frame, such as the instruction counter and parameters, so that Q has enough information to start. After Q starts running, it enlarges the frame to make room for local variables, since only Q knows how many local variables it needs. Suppose that, after running for a while, Q is ready to return an answer to P. The evaluator must restore the stack pointer and the frame pointer to the top and bottom of the frame for P, so that P can continue. Setting the new stack pointer is easy: the top of P's frame is the same as the bottom of Q's

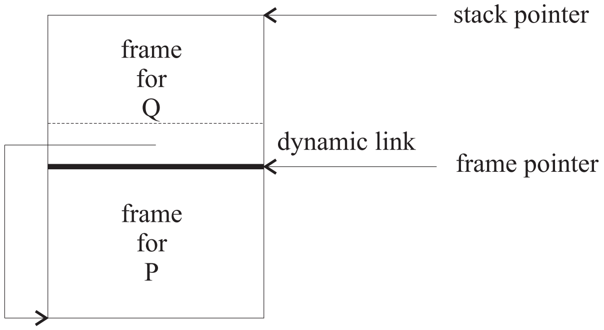


Figure 13.1: **Two frames on the run-time stack.** The stack pointer points to the top of the topmost frame, and the frame pointer points to the bottom of the topmost frame. The dynamic link in Q's frame points to the bottom of P's frame.

```
Define f(x,z) = g(z) |
  Define g(y) = x + y.
%Define
```

Figure 13.2: **Nested function definitions in Cinnameg.** The result returned by $f(x, z)$ is the value of expression $g(z)$ after defining function g as shown. (Read the bar as *where*.) For example, $f(1, 5) = g(5) = 1 + 5$. Notice that g is allowed to use variable x that is in scope at the point where g is defined.

frame, so it suffices to copy the frame pointer into the stack pointer. But how is the new value of the frame pointer, the bottom of P's frame, determined?

A solution is to store the address of the bottom of P's frame at the bottom of Q's frame, in a variable called the *dynamic link*, and to copy that into the frame pointer. Figure 13.1 shows Q's frame holding a dynamic link pointing to the bottom of P's frame.

13.3. Scope and closures

Figure 12.2 in Chapter 12 shows an example of nested subprogram definitions, allowing the inner subprogram to use variables defined in the outer one. Figure 13.2(a) shows a similar thing, written in Cinnameg, where function g is written inside function f. The scope of x is the entire definition of function f. Since the definition of g lies within the definition of f, g can (and does) refer to x. But x is bound in the definition of f, not within g, so the value of x is stored in f's frame, not g's. Somehow, g needs to find f's frame. Notice that, when g is running, the stack frame for f will be immediately beneath g's in the run-time stack, since g is only called by f, so an obvious idea is to follow the dynamic link in g's frame to find the stack frame of f.

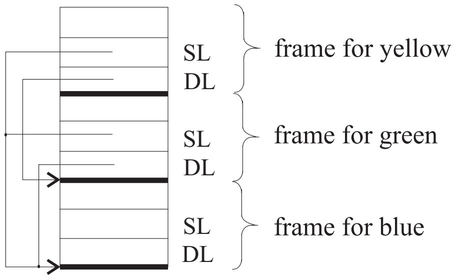
But that assumes that the only function that can call g is f, and that is not always the case. Figure 13.3 shows another example, a definition that contains two embedded definitions. Since blue calls green, which calls yellow, a frame for green will lie between the

```
Define blue(x,z) = green(z) |
  Define yellow(y) = x + y.
  Define green(w) = yellow(w+1).
%Define
```

Figure 13.3: **Function blue, with functions yellow and green defined inside it.** The value of x is stored in blue's frame. Blue calls green, which calls yellow. To find the value of x, function yellow needs to skip over the intervening frame for green.

frames for yellow and blue. But yellow needs to find the value of x, which is stored in blue's frame. The dynamic link does not point to the frame where x is located.

A solution to this problem is to add another link to each frame, called the *static link*, also called a *scope link* or *access link*. The static link in a frame for subprogram S points to the bottom of the frame where S was created, which is where S can find variables that were in scope when it was created. For example, since yellow is created inside blue, the static link in a frame for yellow will point to a frame for blue. When yellow is running, the run-time stack looks as follows, where SL is the static link and DL the dynamic link.



It is often the case, however, that solving one problem creates another, and this is no exception. Once you have decided to add a static link to each frame, you need to think about how you know what to store into it. When green calls yellow, for example, how does it know what to put into the static link of the new frame for yellow? It is tempting to let green pass its own static link on to yellow, but there is no reason to think that a subprogram can only call other subprograms that were created in the same place.

To illustrate what can happen, let's look at an example where a function is used far from the point where it is created. Some languages, including Cinnameg, allow you to pass a function as a parameter to another function. An example of that is function mapf(f, $[a, b, c]$) $= [f(a), f(b), f(c)]$. In general, mapf(f, x) computes f of each value in list x. For example, if add1(y) $= y + 1$, then mapf(add1, $[2, 5, 7, 3]$) $= [3, 6, 8, 4]$. Figure 13.4 shows a definition of mapf and of function h that uses mapf. When h runs, it will call mapf, which will need to call g (several times). But mapf and g might be defined in different modules, so mapf has no idea where g was created unless it is told. The obvious thing to do is to tell mapf where g was created. What is passed to mapf, representing function g, is a pair called a *function closure* that contains (1) the instruction address where evaluation of g will start and (2) a pointer to the activation where g was created. That pointer will become the static link when g is applied. For example, suppose that expression $h(3, [4, 5])$ is evaluated, using the definition of h in Figure 13.4. When h defines g, it builds a function closure for it, with

```
Define
  case mapf(f, [])    = []
  case mapf(f, x::xs) = f(x) :: mapf(f, xs)
%Define

Define h(x,z) = mapf(g,z) |
  Define g(y) = x + y.
%Define
```

Figure 13.4: **Definition of mapf and a function h that uses mapf, in Cinnameg.** $h(3, [4, 5, 6, 7]) = [3+4, 3+5, 3+6, 3+7] = [7, 8, 9, 10]$. Function h calls mapf, which needs to call g. When mapf pushes a frame for g, it needs to know what to install as the static link in the new frame. It gets the pointer to install from the representation of function g as a function closure, as shown in Figure 13.5.

The definition of mapf uses a feature of Cinnameg where you can write a definition in cases. The cases are tried in the order written. The first case tells how to perform mapf when its second parameter is an empty list, and the second case tells the answer when the second parameter is a nonempty list whose head is x and whose tail is xs. As in SML (Figure 9.1), x::xs is the list whose head is x and whose tail is xs.

a static link pointing to the bottom of the current stack frame. (That is, h copies its own frame pointer into the new function closure.) Just before calling mapf, the activation of h is as shown in Figure 13.5(a). The value of g, stored in the activation of h, is a pointer to a function closure, and the closure points back to the activation where g was created, and also holds the instruction address where g will begin its computation. We write g:1 for the address where g starts, and h:i for some address in function h.

Now h calls mapf, and mapf calls $g(4)$, installing information from the function closure for g into a new frame, yielding the run-time stack shown in Figure 13.5(b). It is a simple matter for g to locate x by following the static link.

It is possible to define subprograms within subprograms within subprograms. To obtain a binding, it might be necessary to follow a static link to another activation, and then to follow that activation's static link yet again to find a binding. The compiler can see just how long a chain of static links must be followed to obtain a given binding, and can compile that information into the instructions that implement a subprogram.

13.4. Tail recursion and tail calls

Normally, computation of a function by recursion requires storing several activations of the function simultaneously in the run-time stack. For example, suppose that the factorial function is defined in Cinnameg by cases as follows.

```
Define
  case factorial(0) = 1
  case factorial(n) = n*factorial(n-1)
%Define
```

Then to compute factorial(n), a total of $n + 1$ frames for factorial need to be present in the run-time stack at once. Each is waiting for the one above it to finish, so that it can

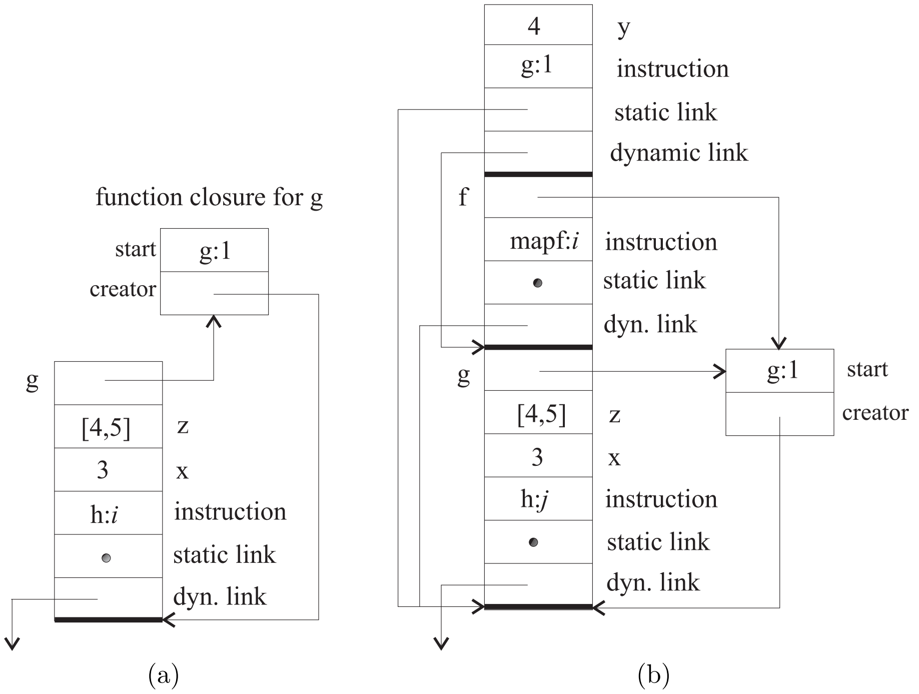


Figure 13.5: **Evaluation of** $h(3, [4, 5])$**.** Diagram (a) shows the frame for h in the run-time stack just before h calls mapf. Notice that the value of g in that frame is a pointer to a function closure. Diagram (b) shows the stack after h has called mapf and mapf has called g. The frame for mapf is abbreviated, showing only the value of f. Mapf gets the static link to use for g from the closure that represents g.

multiply the result by its own value of n. But sometimes creating many frames can be avoided. Suppose that factorial is defined recursively as follows, using a helper function factTimes$(n, x) = x * n!$. (Check that the equations for factTimes are consistent with this definition.)

```
Define
  case factTimes(0,x) = x
  case factTimes(n,x) = factTimes(n-1, n*x)
%Define
Define factorial(n) = factTimes(n,1).
```

A small improvement to stack management avoids letting the stack grow when evaluating a call to factTimes. Start with factorial(3), which calls factTimes(3,1), which in turn calls factTimes(2,3). But instead of placing the frame for factTimes(2,3) above the frame for its caller, the frame for factTimes(2,3) can *replace* the frame for factTimes(3,1). The reason is that factTimes(3,1) wants to return exactly the result of factTimes(2,3) to factorial(3). There is no need for it to be around waiting to pass the result back to factorial; we can just ask factTimes(2,3) to give its result directly back to factorial(3).

Whenever the result of one function call is defined as exactly the same as the result of another, with no computation to be done with the result, we say that the function call is a *tail call*, and when the call is recursive, it is *tail recursive*. (In the case where a subprogram has no result, a tail call is one where nothing more is done after the call.) The altered stack management rule can be used for tail calls, with the new frame replacing the frame of the caller.

Recursive definitions that use tail recursion have the potential to be much more efficient than general recursive definitions, as long as a compiler or interpreter is prepared to recognize them, and to use the altered stack manipulation rule. But not all tail calls are recursive. For example, the definition of factorial does a tail call to factTimes, since it produces exactly the same result as factTimes produces. So the frame for factorial(3) can overwritten by the frame for factTimes(3,1).

Doing your own tail recursion improvement

Figure 13.6 shows an implementation of the Quicksort algorithm to sort an array, written in C++. There are two recursive calls to quicksort. The second call is tail recursive, since nothing is done after that call returns. It can be replaced by (1) an adjustment of the parameters (in this case, just changing low to mid+1), and (2) a jump back to the beginning of the function. Figure 13.7 shows a modified definition of quicksort that does the jump explicitly, using the goto feature of C++. You should see how the goto can be replaced by a while-loop. Try writing this version of quicksort without an explicit goto.

13.5. Summary of terminology and concepts

Bindings of local variables in subprograms are generally kept in the run-time stack, which is broken into frames, each frame holding an activation of a function. Each frame points to the one beneath it via its dynamic link.

In order to permit access to nonlocal variables, each activation has a static link, pointing to the activation where the corresponding subprogram was defined. The requirement

```
int partition(int A[], int low, int high)
{
  int i,j;
  i = j = low + 1;
  while(j <= high) {
    if(A[j] < A[low]) {
      swap(A[i], A[j]);
      i = i + 1;
    }
    j = j + 1;
  }
  swap(A[i-1], A[low]);
  return i-1;
}

void quicksort(int A[], int low, int high)
{
  if(low < high) {
    int mid = partition(A, low, high);
    quicksort(A, low, mid-1);
    quicksort(A, mid+1, high);
  }
}
```

Figure 13.6: **Implementation of the Quicksort algorithm in C++.** If low $\leq$ high, then a call to quicksort$(A, \text{low}, \text{high})$ sorts the portion of array A from indices low to high. If low $>$ high, then quicksort$(A, \text{low}, \text{high})$ does nothing. The partition function rearranges array A around a pivot value, A[low], returning the index where the pivot ends up. We assume that statement swap(u, v) swaps the contents of variables u and v. The second recursive call to quicksort is tail recursive, but the first is not.

```
void quicksort(int A[], int low, int high)
{
start:
  if(low < high) {
    int mid = partition(A, low, high);
    quicksort(A, low, mid-1);
    low = mid+1;
    goto start;
  }
}
```

Figure 13.7: **A modified definition of quicksort that replaces the second recursive call by a change of the parameter (low) and a jump back to the start.** The effect is to modify the frame for quicksort to look like the frame for its recursive call. This is just the modification that an intelligent compiler can make for you, effectively avoiding creating a new stack frame for the second recursive call that overwrites the current frame.

for putting the static link in an activation means that functions and procedures must hold their own static links. The combination of a static link and an instruction address is called a function or procedure closure.

Tail recursion can be implemented in an especially efficient way, requiring much less memory than general recursion. An intelligent compiler can replace tail recursion with a loop.

Terminology

Activation An activation is the contents of a frame in the run-time stack, describing one call of a subprogram. There can be several activations for the same subprogram in the run-time stack simultaneously when recursion is used.

Dynamic link The dynamic link is a link from a stack frame to the next one down from it in the run-time stack. It is used when returning from a subprogram call to find the bottom of the stack frame of the activation to which the current activation returns.

Frame pointer The frame pointer is a pointer to the bottom of the topmost frame in the run-time stack.

Function closure A function closure for function f is a combination of an instruction address and a pointer to the frame that was active when f was created. That frame, and possibly others found by following more static links from there, hold nonlocal bindings that f might need.

Run-time stack The run-time stack is the area in memory where stack frames are placed.

Stack pointer The stack pointer is a pointer to the top of the top frame of the run-time stack.

Static link A static link is the link from a stack frame to the frame where non-local bindings are found. The static link for a frame that is running subprogram g points to the frame that was active when g was created.

Tail call A tail call is a special kind of subprogram call where the result of the called subprogram is returned by the caller unmodified, or where nothing is done after performing the call. Tail calls can be implemented by allowing the stack frame for the called subprogram to replace the frame for the caller, preventing the run-time stack from growing. In the case where a tail call is a recursive call, it is called tail recursion. Tail recursion can be replaced by a “goto” in the program, so that recursion is replaced by a loop.

13.6. Exercises

13.1. What is the purpose of a dynamic link in the run-time stack?

13.2. When a new stack frame is pushed, how does the evaluator know what to put in the new frame’s dynamic link? How does it update the frame pointer and the stack pointer for the new frame?

13.3. When a subprogram returns to its caller, how are the frame pointer and stack pointer adjusted?

13.4. What is the purpose of the static link in a frame of the run-time stack?

13.5. What information is stored in a function closure?

13.6. When a function is called, how does the evaluator know what to put in the static link of the new frame?

13.7. In what ways is tail recursion more efficient than general recursion?

13.8. Using the definition of factorial that does not use factTimes, show the steps in evaluating factorial(3) using a run-time stack by showing a snapshot of the stack after each step. Perform steps leading up to a recursive call as a single step, to avoid too many snapshots of the stack.

13.9. Using the definition of factorial that uses factTimes, show the steps in evaluating factorial(3) using a run-time stack by showing a snapshot of the stack after each step. Perform tail calls by replacing stack frames. Perform steps leading up to a recursive call as a single step, to avoid too many snapshots of the stack.

13.10. It is easy to show that the first equation for factTimes is consistent with the assertion that $\text{factTimes}(n, x) = x * n!$.

$$\begin{array}{lcll} \text{factTimes}(0, x) & = & x & \text{from the definition of factTimes} \\ & = & x * 0! & \text{since } 0! = 1 \end{array}$$

Show that the second equation for factTimes is also consistent with the assertion that $\text{factTimes}(n, x) = x * n!$.

13.11. The definition of quicksort in Figure 13.7 uses a goto. Rewrite it to use a while-loop instead, avoiding use of a goto.

13.12. Quicksort needs to do two recursive calls, to sort two regions of the array, and it does not matter in which order they are done. You can keep the stack depth small by always performing the recursive call on the smaller region of the array, using the goto for the larger region. Modify the definition of quicksort so that it uses the genuine recursive call on the smaller region.

13.13. The definition of function *partition* in Figure 13.6 uses a loop. A loop can always be replaced by tail recursion. Show how to rewrite partition using tail recursion, without using a loop or using any kind of goto statement. (In a language that supports loops, that transformation does not make much sense. But the point is that loops can be simulated by tail recursion in languages that offer no direct support for loops.)

13.14. Show how to replace an arbitrary while-loop by tail recursion. You will need to create a helper function that represents the computation carried out by the loop body.

13.15. C# is an object-oriented language. A function (also called a method) can exist in the context of an object, where it has access to the components, such as variables, of that object. A *delegate* is a function that can be removed from the object and passed as a parameter to another function. But the delegate still has access to the object from which it was removed. Explain why a delegate is similar to a function closure.

13.7. Bibliographic notes

The version of quicksort shown in Figure 13.6 is from Bentley [14]. The GNU C/C++ compiler is one that will use the improved implementation of tail recursion in some places, provided it is asked to optimize.

Part V

Imperative Programming

Chapter 14

From Machine Language to Structured Programming

14.1. Imperative programming

Chapter 2 introduces imperative programming, based on the use of commands that change things. From the earliest days of programming in machine language to today's more advanced languages, imperative programming has held a dominant spot in software design. This chapter explores fundamental ideas behind imperative programming, including the evolution of those concepts over time.

If you are well-versed on the architecture of a particular kind of computer, then you can write software in machine language, making direct use of the computer hardware. Unfortunately, that requires a great deal of specialized knowledge, yields nonportable software, and is so difficult that mistakes in machine-language programs written by humans are commonplace and difficult to track down.

Instead, most people write in a higher level programming language, which provides the programmer with a particular view of the computer and computation. Some kinds of languages use highly abstract views, quite far from the nature of the underlying computer hardware, relying on an interpreter or a sophisticated compiler to bridge the gap between the two. Imperative languages tend to be closer, in the view that they present, to the way a computer really works, and imperative programs can be highly efficient, since the programmer describes what is to be done in the way that the computer naturally works, and there is little between the programmer and the computer to disrupt efficiency. A good C programmer, for example, can often visualize the machine language program into which his or her C program will be translated. Programs written in C can be as efficient as programs written directly in machine language. They are just a lot easier to write.

Imperative programming languages also enjoy the advantage that they are relatively easy to implement, since the basic operations are already provided by the computer hardware. As a consequence, language implementors have time available to work on fine-tuning their implementations to produce very efficient machine language programs, further improving efficiency.

```
v = expr;
```

(a)

```
V := expr;
```

(b)

```
Let v = expr.
or
Relet v = expr.
```

(c)

```
(set! 'v expr)
```

(d)

Figure 14.1: **Statements that assign the value of expression *expr* to variable *v* written in (a) C, (b) Ada, (c) Cinnameg and (d) Scheme.** In Cinnameg, you use Let to create a variable and give it a value, and Relet to change its value after it has already been created.

14.2. Statements, variables and assignment

The basic unit of computation in an imperative program is the *statement*, which is a command that performs an action, such as changing the value of a variable within the program or changing something external to the program, such as the contents of a file on a disk. A statement that changes the value of a variable is called an *assignment statement.* Figure 14.1 shows assignment statements in a few different languages. For uniformity, we will write $var := expr$ to indicate an assignment.

14.3. Control: early approaches

Each statement potentially affects the actions that are done by future statements. Assignment $x := 4$, for example, affects the value later computed by assignment $y := x - 2$. A program achieves its ends not only by choosing which statements to do but also by controlling the order in which they are done, or how control flows through the program.

An early method of specifying the flow of control was to use *flowgraphs.* Restricting attention to assignments, a flowgraph is a diagram made from a start node, a stop node and computational nodes of two kinds, linked by directed edges (arrows between the nodes). The first kind of computational node is an assignment node, usually drawn as a rectangle, and holding an assignment statement. When the assignment node is reached during computation, the assignment statement inside it is performed. The node has one edge leading out of it, pointing to the node to continue with.

The second kind of computational node is a choice node, usually drawn as a diamond, and containing an expression that must produce a boolean value. It has two directed edges leaving it, one labeled true, the other labeled false. The next node to continue with is found by following the edge with the label that matches the value of the expression.

We have ignored issues concerning inputs to a flowgraph and outputs that it produces. For simplicity, presume that certain variables represent parameters, having values when the flowgraph starts, and that other variables represent results. Figure 14.2 shows an example

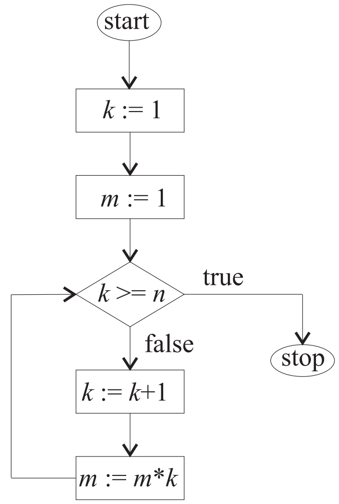


Figure 14.2: A flowgraph computing m = n!.

of a flowgraph that computes $m = n!$. Variable n is the parameter, presumed to have a value when the flowgraph starts, and m is the result.

There is little doubt that flowgraphs are simple. Excluding the start and stop nodes, there are only two kinds of nodes, with simple rules for connecting them. Loops in the program are immediately evident as cycles in the flowgraph. Among other things, Occam's razor tells us to choose simple solutions when there is no compelling reason to select more complex ones. So why not program with flowgraphs?

Early programmers did just that, at least in their initial designs. Unfortunately, flowgraphs do not lend themselves well to typewriters, and early computers had little graphics capabilities. (Even on modern computers, programming directly with flowgraphs is cumbersome.) So programmers encoded a flowgraph in a more convenient form, where each node was given a name or number, and the statements in the flowgraph were written as a program, with their numbers attached as *labels*. Links between the nodes were described by "goto" statements. The number of gotos was reduced by adopting a rule that, in the absence of an explicit goto, control flows to the next statement in the program. This approach was adopted in early versions of Fortran. Figure 14.3 shows a Fortran IV program fragment that implements the factorial flowgraph. It assumes that N already has a value, and it does not do anything with the result M except compute it.

That style of programming flourished for several years, but it was found to have two serious drawbacks. First, the program is much more difficult to read and follow than the flowgraph from which it is derived. Seeing a goto, the reader finds himself or herself searching up and down for the label. A second drawback is that there is almost no direct correspondence between the structure of the program and the structure of the computation that it describes. For example, a simple choice looks a lot like a loop to a quick glance. Two loops coming one after another can look a lot like two nested loops or even a program that just does choices and contains no loops, even though the structures of the computations are very different. Some programs made such complex use of gotos that they were nearly impossible to understand.

```
    K = 1
    M = 1
10  IF(K .GE. N) GOTO 20
    K = K + 1
    M = M * K
    GOTO 10
20  CONTINUE
```

Figure 14.3: **Computation of M = N! in Fortran IV.** This is a translation of the flowgraph shown in Figure 14.2. Statement labels are in the left margin; it is only necessary to label a statement that is the target of a goto. The CONTINUE statement is a "do nothing" statement that is just used as a place to put a label. The .GE. operator is Fortran's way of saying $\geq$.

14.4. Structured programming constructs

To overcome the difficulties with direct translation of flowgraphs using gotos, a new form of programming language, a *structured programming language*, was developed. Flowgraphs and gotos were replaced by a small collection of *control structures* that indicate, in clear terms, how computation in a program flows. There are three fundamental kinds of control structures of structured programming: sequencing, choice and looping.

Sequencing

A sequence of statements is executed one after another, in the order written. For example, sequence

```
x := 0;
y := x + 1;
z := 2;
```

first sets x = 0, then sets y = 1 and last sets z = 2. This was, of course, already in use in Fortran IV. Structured programming simply recognized the importance of sequencing as a fundamental way of ordering computational steps.

Some languages, such as Ada and Cinnameg, allow sequencing in almost any context; where you can write one statement, you can write several in a row. Both Ada and Cinnameg have a syntax that makes it is clear where every sequence of statements begins and ends. But some languages only allow sequencing inside a statement called a *compound statement*. In C[1], for example, a compound statement has the form $\{s_1 \ldots s_n\}$ where s_1, ..., s_n are statements. The braces indicate the beginning and end of the sequence of statements. In Pascal, a compound statement has the form **begin** s_1; ...; s_n **end**, where the statements are separated by semicolons.[2]

[1] Java, C# and C++ borrow a lot of their syntax from C, so if you are familiar with any of those languages, you should recognize the syntax.

[2] Syntactically, C and Pascal take slightly different points of view here. In C, a simple statement such as an assignment statement ends on a semicolon, and the semicolon is part of the statement. In Pascal, the semicolon is not syntactically part of the statement, but separates the statement from the next one. You can think of the semicolon as a "sequencing operator," where $A; B$ says to do A then B. There is no semicolon after the last statement in a Pascal compound statement.

```
if (expression)
   statement1
else
   statement2
```

(a)

```
If expression then
   statements1
else
   statements2
%If
```

(b)

Figure 14.4: **Two syntactic forms of a conditional statement, (a) in C and (b) in Cinnameg.** Flowgraph 14.5 shows a corresponding flowgraph.

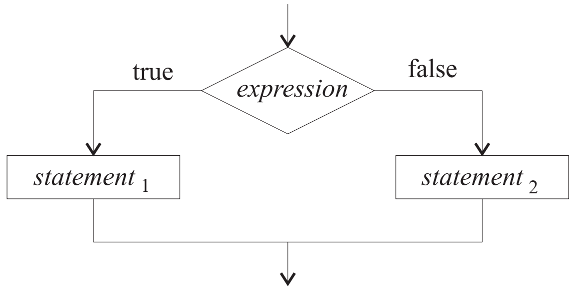


Figure 14.5: **A flowgraph of C statement** `if (`*expression*`)` *statement*$_1$ `else` *statement*$_2$ **.**

Choice

A *conditional* statement allows a program to choose between alternatives. The most basic conditional statement in C has the form shown in Figure 14.4(a). Syntactic variations on conditional statements are common, but the general idea is similar regardless of the syntax. For example, Cinnameg uses the form shown in Figure 14.4(b).

Loops

A loop, also called an *iterative* statement, calls for performing a statement or a collection of statements repeatedly. The most common form of loop is a while loop, illustrated by Figure 14.6. There are syntactic variations on the while-loop, but the concept is the same in different languages.

Contextual independence

Assigment statements, choices and loops all share the characteristic that they have a single entry point and a single exit point. So all can participate in sequencing. An important principle of structured programming is that any kind of statement can occur where any other kind can occur. There are no statements that have special privileges. The body of a while-loop can be (or can include) any kind of statement, whether it is an assignment

```
while(expression)
   statement
```

(a)

```
While expression do
   statements
%While
```

(b)

Figure 14.6: **While loops written in (a) C and (b) Cinnameg.** The meaning is the same as the flowgraph in Figure 14.7.

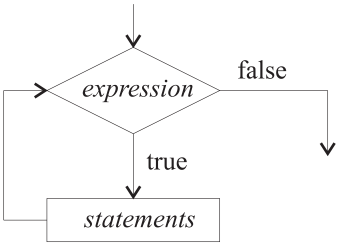


Figure 14.7: **A flowgraph of Cinnameg statement** `While` *expression* `do` *statements* `%While`**.**

statement, a compound statement, a conditional statement or another while loop. Also, the meaning of a construct does not depend on where it occurs. A while loop does not change its nature just because it occurs in a particular place in a program. Figure 12.1 shows a loop inside a loop.

14.5. Variations on choice

Multiway choice

It is common for a choice to have more than two alternatives. For example, consider the problem of computing the sign of a number, where the sign of x is defined by

$$\text{sign}(x) = \begin{cases} -1 & \text{when } x < 0 \\ 0 & \text{when } x = 0 \\ 1 & \text{when } x > 0 \end{cases}$$

There are naturally three alternatives in the definition of the sign function. The choices can be performed using if-then-else, by letting the else-part of the first if-statement be another if-statement. Figure 14.8(a) shows a C statement that sets variable *sign* to the sign of x. A language with end markers, such as %If, can make multiway choices awkward because each two-way choice needs an end marker. Typically, a special provision is made for multiway choices. Ada, for example, allows the word **elsif** to stand for **else if**, but without a matching end marker, as shown in Figure 14.8(b). Cinnameg provides a construct that is illustrated in Figure 14.8(c) for doing multiway choices.

```
if(x < 0)
  sign = -1;
else if (x == 0)
  sign = 0;
else
  sign = 1;
```

(a)

```
if X < 0 then
  Sign := -1;
elsif X = 0 then
  Sign := 0;
else
  Sign := 1;
end if
```

(b)

```
open Choose
  case x < 0  => Let sign = -1.
  case x == 0 => Let sign = 0.
  case x > 0  => Let sign = 1.
%Choose
```

(c)

Figure 14.8: **A three-way choice written in (a) C, (b) Ada and (c) Cinnameg.** The Cinnameg form needs to be marked *open* to allow the bindings of sign to be visible outside the choose construct, which normally would be a block, as described in Section 12.2.

Multiway choice based on a value

Sometimes, instead of testing several Boolean expressions, you want to base a decision on the value of some variable or expression. For example, you could do one thing if x is 1, another if x is 2, and yet anoter if x is 3. Figure 14.9 shows an example written in C (a switch statement) and Cinnameg (a variation on the choose construct). There is an important difference, beyond mere syntax, between the Cinnameg choose construct and the C switch statement that tells you something about the design philosophies of those two languages. The C switch statement requires that the value being tested (n in Figure 14.9(a)) is an integer, or something that is represented by an integer, such as a character. It also requires the case labels to be constants, not variables or more general expressions. You are not allowed, for example, to use a case of the form `case k:` where k is a variable. Those restrictions allow the compiler to exploit features of computer architecture to perform the switch efficiently. The compiler can create an array of instruction addresses and select an address to jump to from that array based on the value being tested. For example, the array entry at index 1 would be the address of the machine-language code for the instructions to be performed when n is 1. That is consistent with the C philosophy of providing efficient and direct access to machine resources.

The Cinnameg **choose** construct has no such restrictions. The philosophy of Cinnameg is to provide a more logical language, and allowing any kind of value to be tested is consistent with that view. That does not preclude using an efficient implementation when the compiler recognizes that the value being tested happens to be an integer, and the labels happen to be constants. It just forces the compiler to work a little harder to determine how to compile a program, if it wants to do an efficient job. Even if such an improvement is made, however, the programmer might not be aware of it. Where C tends to push the programmer toward more efficient ways of doing things, Cinnameg leaves the programmer to take the approach that he or she understands best.

```
switch(n)
{
case 0:
  printf("You are lucky\n");
  break;
case 1:
  printf("You will be happy\n");
  break;
case 2:
  printf("You will be rich\n");
}
```

(a)

```
Choose matching n
  case 0 =>
    Displayln("You are lucky").
  case 1 =>
    Displayln("You will be happy").
  case 2 =>
    Displayln("You will be rich").
%Choose
```

(b)

Figure 14.9: **A statement in (a) C and (b) Cinnameg that prints a fortune based on the value of variable** ***n*****.**

14.6. Variations on loops

Programs written using structured programming constructs tend to be much easier to read than programs written using gotos, because they clearly state what they are trying to do. The (static) structure of the program reflects the (dynamic) structure of the computation.

But the common structured programming constructs do not do a good job of indicating what is being done in all cases. The while loop, for example, has an important defect; the decision about whether to continue the loop is made only at one place, the top of the loop. In practice, loops often do not fit that structure. For example, imagine a program that reads and processes a sequence of values, until it reads a value that marks the end of the input. The loop is most naturally expressed as follows.

```
Loop
  1. Read a value.
  2. If the value is the end marker then exit the loop.
  3. Process the value.
End loop
```

Notice that the decision about whether to continue the loop is made in the middle of the loop body, not at the top. When that is written as a while loop, the decision must be forced to the top of the loop. That can be done by duplicating the line that reads a value, and forcing each loop iteration to prepare for the next one, as follows.

```
Read a value.
While(the value is not the end marker)
  Process the value.
  Read a value.
End loop
```

The goal of structured programming is to make the form of a program clearly reflect its intent. By forcing the loop test to be made at the top, the while-loop has caused the program to migrate away from the intent of the program.

```
Loop
  do Read a value
  exitcase the value is the end marker => ()
  loopcase else =>
    Process the value
%Loop
```

Figure 14.10: **A read/process loop in Cinnameg.** The loop has three kinds of cases: those that request computation to performed (**do**); those that exit the loop (**exitcase**); and those that continue the loop for another iteration (**loopcase**). There can be any number of cases, in any order, so the programmer can decide when to exit the loop.

Solutions to this problem have been provided in most languages. In C, for example, there is a **break** statement that can be used to terminate a while-loop at any point. But the while loop is intended to convey information to the reader that the exit test is at the top; a break statement makes a lie out of that. A better solution is a loop that is normally intended to exit at any point, and some languages provide those loops. Figure 14.10 shows the read/process loop in Cinnameg.

Managing the loop control variables

One goal of programming languages is to reduce the frequency of errors in programs. Among the constructs used in structured programming, loops tend to be among the most error-prone. In recognition of that, most programming languages that support loops offer a kind of loop that can reduce the error rate.

Typically, a loop has a small collection of variables, called the loop *control variables*, that are initialized at the start of the loop and are updated each time through. Common mistakes include failure to initialize and failure to update one or more of the control variables. A *for loop* offers a solution when there is one control variable, or one primary control variable. The for-loop syntax either has special places for the initialization, test and update code, so that none of them can be accidentally left out (illustrated by C), or performs all initialization and updates automatically (illustrated by Cinnameg). Figure 14.11 shows an algorithm to compute $n!$ written in C and Cinnameg using a for loop. It is almost impossible to forget to initialize or update the control variable k.

Unfortunately, it is still possible to forget to initialize or update m. Loops often have more than one control variable, and it would be nice if all of them could be managed in the loop heading.[3] Cinnameg's general loop allows you to specify several control variables, in a way that makes it difficult to forget to initialize or update any of them. The factorial loop in Cinnameg is shown in Figure 14.12.

14.7. Summary of terminology and concepts

Imperative programming is based on statements, which are commands that perform actions. Typically, the actions change the values of variables.

[3] It is possible to have more than one control variable for a C for-loop. See exercise 14.3.

```
m = 1;
for(k = 2; k <= n; k = k + 1)
{
  m = m * k;
}
```

(a)

```
Let m = 1.
For k from [2, ..., n] do
  Relet m = m * k.
%For
```

(b)

Figure 14.11: **Computation of $m = n!$ expressed in (a) C and (b) Cinnameg using a for-loop.** The C form contains an initialization part, a test to determine whether to keep going, and an update part, all together in the heading rather than spread around the loop. The Cinnameg for-loop performs a sequence of statements for each member of a list.

```
Loop (k,m) = (1,1)
  exitcase k >= n  => m
  loopcase else    => Continue(k+1, m*(k+1)).
%Loop
```

Figure 14.12: **The factorial computation expressed in Cinnameg.** The loop control variables are k and m. They are both initialized to 1 in the loop heading, which is a pattern match, as described in Section 11.7. In Cinnameg, a loop is an expression; each exitcase tells the value that the expression produces. When $k \geq n$, the loop exits and produces value m. Otherwise, the loop continues back to its top, replacing the values of k and m by $k + 1$ and $m * (k + 1)$, respectively. The loop syntax makes it impossible to omit the initialization or the update.

Imperative programming is close to the architecture of computers, so it allows the programmer to express computation in a way that is very close to the way it will actually be performed on the computer. Allowing (and requiring) the programmer to work at a level close to the computer tends to make language implementations easy to create and tends to make programs efficient.

Early approaches to imperative programming were based on direct translations of flowgraphs, but that approach was found to lead to programs that were difficult to understand. Structured programming is an alternative where programs reflect, in the way they are written, what is going on in the computation. A loop looks like a loop, and a choice looks like a choice.

Terminology

Assignment An assignment statement computes a value and puts that value into a variable.

Compound statement A compound statement consists of a sequence of statements collected together into one large statement that is performed by doing the statements inside the compound statement one after another, in the order in which they are written. Compound statements are found in languages where sequencing is restricted to the body of a compound statement.

Conditional statement A conditional statement, such as an if-statement, allows a programmer to choose between two or more alternatives.

Control flow Imagine pointing your finger to the box in a flowgraph that is currently being performed by a program, and moving it through the flowgraph, following the arrows, as the program runs. The motion of your finger is the control flow in the program, representing how control moves through the flowgraph. You can also understand control flow in a structured program by following the sequence of statements that the program performs.

Control structure A control structure is a kind of statement that is used to control execution of other statements. Examples of control structures are conditional and looping statements.

Flowgraph A flowgraph is a graphical diagram of a computer program, employing basic computation nodes (such as assignments) and decision nodes linked by directed edges.

Imperative programming An imperative style is any style of programming where a program is a collection of commands (or statements) together with constructs or statements that control the order in which the commands are performed.

Iterative statement Loops are called iterative statements.

Loop A loop is a kind of control structure that calls for performing another statement or sequence of statements repeatedly.

Sequencing Sequencing is the operation of performing one statement after another, in a specific order.

Statement A statement is a command that calls for an action to be performed. Unfortunately, the programming-language definition of a statement does not agree well with the usual definition of a statement, such as "The car is green," that states a fact, but this term was used for early languages, and it has stuck.

Structured programming Structured programming is a form of imperative programming based on control structures such as if- and while-statements, where gotos are not used (or are used only in very limited ways).

Variable An imperative program typically keeps track of information by changing what is stored in variables.

14.8. Exercises

14.1. Explain why flowgraphs were replaced by structured programming constructs.

14.2. How does Cinnameg's **Loop** construct differ from a while-loop, in ways other than obvious syntactic ones. What is the important difference to a programmer?

14.3. The C for-loop can be used with multiple control variables. In the loop heading for(A;B;C) you can use a comma to separate steps in individual parts. For example, for(A,B; C; D,E) says to do both A and B initially, in that order, and to perform both D and E at the end of each iteration.

(a) Show how to write the factorial function (Figure 14.11) in C with all of the computation done in the for-loop heading. Both k and m should be initialized and updated in the heading. The body of the loop should be an empty set of braces, indicating that nothing should be done in the body.

(b) Does this feature of C make it more difficult to forget to initialize or update one of the control variables? Explain.

(c) Do you believe this style of programming makes the loop easier to understand or more difficult to understand?

(d) The comma can actually be used in any expression; A,B says to compute A then B, with the value of B used as the expression's value. Novice C programmers sometimes try to return two values from a function by writing

```
return x,y;
```

What does this do? In your opinion, is this a difficulty with the language or just a difficulty with novice programmers?

14.4. Each case in a C switch statement is normally ended by a break statement. If the break statement is omitted, then control flows from one case directly into the body of the case that follows it. So, without the break, more than one case body is performed.

(a) Show how to write a C program that prints the first five verses of "The Twelve Days of Christmas" using a switch without any breaks. (Feel free to substitute a different but similar song that involves repetition of lines from prior verses.)

(b) Do you find the requirement for a break to be a desirable feature? Why or why not? Can you suggest a way to modify C so that the breaks do not normally need to be written, but so that control flow from one case into the next is still possible?

14.5. Write a program fragment involving a loop that computes $m = n^k$, where n and k are integer variables that have been given values and $k > 0$. Do not change n or k in the process. Write the fragment in C and in Cinnameg.

14.6. Draw a flowgraph of your loop from question 14.5.

14.7. Give an example of a looping computation that naturally exits at the bottom of the loop. If you know C, C++, C# or Java, give an example of a loop control structure that has an exit test at the bottom instead of at the top.

14.8. Some languages that support structured programming constructs, such as C, retain the ability to do a direct goto. In C, you label a statement by writing the label name and a colon in front of the statement. Then statement `goto lbl;` jumps to label lbl. For example,

```
here:
printf("I am happy\n");
goto here;
```

prints I am happy over and over until you stop the program.

(a) Translate the factorial fragment of Figure 14.3 into C, using gotos, and avoiding structured programming constructs.

(b) Your solution to part (a) is probably considerally more difficult to read than an implementation that uses a while loop. But can you think of any ways in which a goto might improve the readability of a program, and actually be preferable to using only structured programming concepts? If you cannot, you might consult Knuth [64].

(c) It can be difficult to say just what a goto means when it is combined with structured programming constructs. Give an example where a goto jumps from outside a construct into the interior of it, where the meaning is unclear. Just where does the confusion arise? Do you think a goto should be allowed to jump into a construct from outside of it?

14.9. Bibliographic notes

Dahl, Dijkstra and Hoare [35] advocate the use of structured programming, and Dijkstra [38] recommends abandoning the idea of a goto statement. Knuth [64] offers an opinion that gotos can be combined reasonably with structured programming constructs.

Chapter 15

Objects and First Class Variables

15.1. Variables as data items

Variables, as we have used them up to now, are not first class items. Our programs talk about the value of a given variable, but not the variable itself. Our variables have just been names, and assigning a value to a variable is nothing but binding the name to a new value. Now, we turn to the possibility of letting a variable be a data item, where you can store a variable inside another variable, pass a variable as a parameter to a subprogram, or return a variable from a subprogram.

We will call a first class variable a *box.* You can think of it as like a shoe box that is capable of holding one thing. You can open the box and see what is inside, or you can take what is in it out, and put a new thing in it. But you can also just treat the box itself as a thing. You can, for example, give the box to someone else.

The box analogy suffers from a defect: in the real world, there is just one of each box. But in a program, you can put the same box in several different places. That defect can be eliminated by imagining a rack of numbered shoe boxes. When you say that you have a box, you actually only have its number, and when you lend the box to somebody else, you just give that other person the box number. Anybody can go to the rack and open a given box, either just looking at what is inside or replacing the contents of the box. You can imagine two boxes A and B that hold the same box C inside them. What they really hold is the number of box C. The rack of boxes is analogous to the computer's memory, and the box numbers are memory addresses, also called *pointers.*

Objects

Looking ahead to object-based programming, a box is a very simple kind of *object.* Generally, an object holds some information, and has some capabilities for manipulating that information. A box holds just one item. Its capabilities include an operation to get the item, and one to replace the item with another one.

Cinnameg treats a box as an object. Expression `[:` x `:]` produces a new box, initially holding x. Operator @ gets the content of a box, and statement `Assign` b `=!` v`.` stores v

```
Define factorial(n) = @m |
  Let k = [:1:].
  Let m = [:1:].
  While @k < n do
    Assign k =! @k + 1.
    Assign m =! @m * @k.
  %While
%Define
```

Figure 15.1: **A factorial function written in Cinnameg, using boxes.** The end (`@m |`) of the heading line indicates that the value to be returned is the content of box *m*, after the function body is finished. Notice that the parameter, *n*, is not a box, but is a number.

into box *b*. For example,

```
Let b = [:0:].
Let x = @b.
Assign b =! 5.
Let y = @b.
```

makes *b* name a box, *x* name the number 0 and *y* name the number 5. Fetching the content of a box is called *dereferencing* the box, and @ is the Cinnameg dereference operator.

Notice that a box is anonymous; it has no name associated with it. But you often want to use a name to refer to a particular box, and it is common to create a box and give it a name in a single statement, as is done for *b* above. Figure 15.1 shows a definition of the factorial function, using boxes.

15.2. Expression context: lvalue and rvalue

C++ supports boxes, but with an important difference from Cinnameg. Consider C++ fragment

```
int num = 0;
num = num + 1;
```

When we viewed variables as names, we would say that this makes the name num refer to 0, and then changes the binding of num to 1. In reality, that is not what happens in a C++ program. The first line, `int num = 0`, creates a new box, initially holding 0, and names that box num. The second line, `num = num + 1`, changes the content of the box, not the binding of num (which still refers to the same box).

But if num is a box, then expression `num+1` appears to make no sense. You want to add 1 to the content of the box, not to the box itself. If C++ required the @ operator, then a more sensible assignment statement might be `num = @num + 1`. (Notice that it would not make sense to write `@num = @num + 1` since, if box num currently contains 24, you do not want to perform statement `24 = 24 + 1`. On the left-hand side of the assignment statement, you really want the box.)

Programmers generally find the concept of rebinding names appealing, because there is only one level of binding (a name to a value) to think about, rather than two (a name to a box and a box to a value). When a C++ programmer writes $k = k + 1$, he or she usually thinks in terms of changing the value of k — that is, in terms of changing what the name k is bound to. That is not what really happens, but the language designers want to make it *appear* that the name k is being rebound. To achieve that fiction, the language definition must call for the dereference operation to be inserted automatically sometimes, but not all of the time. A C++ compiler must realize that the statement written $k = k+1$ really means $k\ =\ @k + 1$ where @ has been inserted before the second occurrence of k, but not before the first occurrence.

The language definition needs to include a notion of the context in which an expression occurs. There are two contexts: left context, which is like the left-hand side of an assignment statement; and right context, which is like the right-hand side of an assignment. Each expression has two values. The *lvalue* of expression E is the value that E has when it occurs in a left context. For example, if b is the name of a box, then the lvalue of b is the box itself. The *rvalue* of E is the value of expression E when it occurs in a right context; when E is the name of a box, the rvalue of E is the content of that box.

Context is not always obvious. For example, $A[k]$ is the k-th variable in array A, and in C++ assignment statement `A[k] = r`, k occurs in a right context, because you want its value, even though k occurs on the left-hand side of the assignment statement. For example, if box k currently holds 2, then $A[k] = r$ is equivalent, *at that particular use*, to $A[2] = r$.

We have seen that Cinnameg requires you to insert explicit dereference (@) operators. Cinnameg generally avoids the ideas of left and right context since it is simpler for each expression to have just one value, not two, and for the value not to depend on where the expression is written. But Cinnameg does provide one statement that introduces context. Statement Make @b =! E %Make is defined to mean the same thing as Assign b =! E %Assign. If all you do is fetch values from boxes and assign them values (using Make), then you can put @ in front of a box in every context. So

```
Make @b =! @b + 1
```

adds one to the content of box b.

15.3. Extending context to aggregates

The concepts of assignment, left context and right context can be extended to structured items that contains boxes within them. For example, if u and v are boxes then assignment (u, v) := (1,2) seems to make sense; it makes u hold 1 and v hold 2.

C++ and Java both have the notion of an object, which is a collection of variables along with procedures and functions (called methods) that work with those variables. For our purposes here, we only care about the variables, not the methods, so (for now) we model an object as a tuple of boxes.

C++ and Java treat assignments involving objects differently. Look at Java first. It treats objects as tight encapsulations. Suppose that object U is tuple (u_1, u_2) and object V is tuple (v_1, v_2), where u_1, u_2, v_1 and v_2 are all boxes. In Java, you refer to an object by putting it into a variable. So suppose that variable u contains object U, and variable v contains object V. Assignment $u = v$ just puts the object V held in variable v into u. It does not change any of u_1, u_2, v_1 or v_2, but instead just changes the object to which u refers.

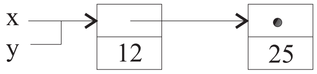


Figure 15.2: **List x = [12, 25] and the result of setting $y = x$ by copying pointers.**

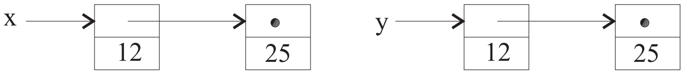


Figure 15.3: **The list x = [12, 25] and the result of setting $y = x$ by copying list structure.**

But C++ treats objects as looser encapsulations, making their structures more visible. In C++, assignment statement $u = v$ has the same effect as doing two assignments, $u_1 = v_1$ and $u_2 = v_2$. That is, not only is v in a right context, so that it stands for its content (v_1, v_2), but the component variables v_1 and v_2 are also in a right content, and their contents are obtained. Similarly, putting u in a left context implicitly puts its component variables u_1 and u_2 into a left context. The C++ rule for assignments on objects is called *coherent assignment.*

An array is, conceptually, a list of boxes. Although Java and C++ treat assignments involving objects differently, they treat arrays in the same way. Suppose that u and v are variables that contain arrays. In both Java and C++, assignment $u = v$ makes variable u hold the same array as v. It does not change what is in either of the arrays. After doing $u = v$, if you change the content of $u[1]$, you will find that the content of $v[1]$ has also changed, since u and v are the same array. But Pascal treats assignments involving arrays more like the way C++ treats assignments involving objects, using coherent assignment. If u and v are (different) arrays, then Pascal statement $u := v$ copies the content of each box of v into the corresponding box in u. Afterwards, u and v are still different arrays, but they have the same values in them.

Cinnameg relies on the kind of statement that you perform to indicate how to treat things. If v is a pair of boxes, then statement `Let` *u* `=` *v*`.` makes u be the same pair of boxes. To request a coherent assignment, you use a Copy statement. Generally, Copy u =! v %Copy copies from boxes in v to corresponding boxes in u, looking inside tuples and lists to find the correspondences.

Assignments involving more complex data structures

A linked list is more involved than either a tuple or an array, and it is not clear what assignment $y := x$ should mean when x is a linked list. The most obvious definition is to make y hold the same linked list as x, with memory sharing, as shown in Figure 15.2. Then, because of the memory sharing, any change to the cells that represent list x will implicitly change list y too. An alternative would be to copy the entire list, as shown in Figure 15.3, avoiding issues of memory sharing, and that might be what the programmer prefers.

It should be clear that, no matter what the standard or default definition of assignment is in a language, the programmer would ideally like to decide, on a case-by-case basis, just what assignment should mean. We have said that C++ performs assignment on objects in a coherent manner. But that is really just the default. C++ allows you, the programmer, to define assignment in a different way for each type of object, by treating the assignment operator = as a function that the programmer can redefine, with a different definition for each type. (So operator = is *overloaded*.) By defining $y = x$ to be the same as $y = \text{copyList}(x)$, when x and y are linked lists, you make assignments on linked lists copy the list. The following C++ definition does the job.

```
List& operator=(List& y, List& x)
{
  y = copyList(x);
  return y;      // Return the variable that was just changed.
}
```

15.4. Summary of terminology and concepts

Allowing variables to be first-class data items gives a programmer a great deal of flexibility, but at some cost in simplicity. In most imperative languages, the value that an expression computes depends on the context in which it occurs, a left context indicating that a box is desired, and a right context indicating that the content of the box is desired.

Not all languages agree on what it means to copy a data structure that contains boxes. C++ allows the programmer to define what copying means on a case-by-case basis.

Terminology

Box A box is an object that can hold an item, called the content of the box. It supports two operations, getting its content and replacing its content by something else.

Coherent assignment Suppose that u and v are structured, or complex, items that contain boxes, and that have the same structure, but with different boxes. Assignment $u := v$ is coherent if it copies from each box in v into the corresponding box in u.

Dereference A deference operation gets the content of a box. Dereferences are often inserted automatically into programs, based on context, but in some languages they must be explicit.

Left context In a left context, the name of a box refers to the box itself. No implicit dereference is done.

Lvalue The lvalue of an expression is that expression's value when it occurs in a left context. The lvalue of a box is the box itself.

Right context In a right context, the name of a box implicitly refers to the content of that box. An implicit dereference is done.

Rvalue The rvalue of an expression is that expression's value when it occurs in a right context. The rvalue of a box is the content of that box.

15.5. Exercises

15.1. Like C++, C uses boxes and implicit dereferencing, with left and right context. Rewrite the C program fragment of Figure 14.11(a) as it would be if there were no notion of left and right context, and all dereference operations were required to be explicit. Since @ indicates logical negation in C, use @ as the name of the dereference operator.

15.2. Rewrite the procedure definition of Figure 12.1 as it would be if there were no notion of left and right context, and all dereference operations were required to be explicit. Use @ as the name of the dereference operator. In C++, function parameters are boxes, and the function body can change their contents. Parameter a of function sort is actually a box that holds, as its content, an array. (If you want to, you can store a different array in that box, though few programmers would do that.) So, each time a is mentioned, you will need to do a dereference to get the array stored in that box.

15.3. The meaning of a fundamental language unit such as an expression or a statement should not depend on the context in which it occurs. From a syntactic standpoint, you can write $x + y$ in any context where an expression is expected, and, from a semantic standpoint, you should expect $x + y$ to have the same meaning in every context.

Having left and right context can be seen as a violation of this rule. Choose a point of view, and argue either that having left and right context really does violate that rule, or that having those two contexts does not cause any problems. Use examples from a familiar programming language to make your point.

15.4. C++ allows the programmer to redefine the meaning of assignment for some types, but not for all. For example, you cannot redefine how copying is done for integers or pointers. Do you consider that a reasonable restriction, or should it be done away with? Why do you think that restriction was made?

Chapter 16

Procedures and Parameter Passing

16.1. Procedural programming

A procedural programmer is encouraged to invent his or her own kinds of statements and expressions. That is not to say that the programmer introduces new syntax into the language; very few languages allow that. Rather, the programmer creates *procedures* that have the effect of being new kinds of statements, and *functions* that have the effect of being new kinds of expressions.

Suppose that a programmer wants the ability to sort an array of numbers into ascending order. Obviously, it would be nice to have a **sort** statement in the language, as COBOL does, but most languages do not have that. Instead, the programmer creates a sort procedure, and uses standard *procedure-call* syntax to use it. For example, to sort array A, you might write

```
sort(A);
```

16.2. Information flow

Flowgraphs are a way of describing *control flow* in programs, namely, how a thread of control weaves its way through the statements, performing them in a specific order. In a flowgraph, control can flow in arbitrary ways, since any node in the graph can conceivably point to any other node. Structured programming is much more restrictive (and, of course, structured) in terms of how it allows control to flow through a program. Each basic construct has a single entry point and a single exit point, and it is partly the restricted flow of control that makes structured programs easier to understand than flowgraphs with arrows pointing in arbitrary directions, obscuring what the flowgraph does.

Another kind of flow that occurs during the execution of a program is a flow of information. For example, a line of a program that stores a value into variable x is implicitly passing information to another line that uses that value of x. Understanding the flow of information in a program is just as important to a programmer as understanding the flow of control.

```
int factIn, factOut;
void computeFactorial()
{
  int k = 1;
  factOut = 1;
  while(k < factIn)
  {
    k = k + 1;
    factOut = factOut * k;
  }
}

void PrintFactorials()
{
  int k = 1;
  while(k <= 10)
  {
    factIn = k;
    computeFactorial();
    printf("%i %i", k, factOut);
    k = k + 1;
  }
}
```

(a)

```
int factorial(int n)
{
  int k = 1;
  int m = 1;
  while(k < n)
  {
    k = k + 1;
    m = m * k;
  }
  return m;
}

void PrintFactorials()
{
  int k = 1;
  while(k <= 10)
  {
    printf("%i %i", k,
           factorial(k));
    k = k + 1;
  }
}
```

(b)

Figure 16.1: **Two implementations of a subprogram that prints the factorials of the numbers from 1 to 10, both written in C.** Version (a) uses global variables to pass information to and from the helper procedure, and version (b) passes information in a more structured way. The printf lines print two integers using formatted prints. Like the other subprograms here, printf is not built into the language, but is a procedure.

An obvious and simple way to pass information is to store it into a variable in one place and get it out of that variable in another place. On the scale of a single subprogram definition, that is generally how information flows. On a larger scale, that approach leads to the idea of shared, or *global*, variables, that can be used by more than one subprogram. Figure 16.1(a) shows a definition, written in C, of a procedure that prints the factorials of the numbers from 1 to 10, using a helper procedure that computes factorials. Information is passed through global variables factIn and factOut. But use of global variables allows unrestricted information flow, just as a flowgraph allows unrestricted control flow. When global variables are used, it is not clear, except after careful reading, how information flows in a program. In large programs it can be too complex to understand.

A more structured approach to information flow is to use parameters of subprograms, and sometimes to return results from functions. Figure 16.1(b) uses parameter passing to accomplish the same effect as Figure 16.1(a), but avoids global variables. With parameter passing, information flow is clear and controlled, and you can see just what is being passed to or from a subprogram by reading the program. So the (static) structure of the program tells you something about how information flows (dynamically) in the program while it runs, in a way that is similar to how the (static) structure of a loop or conditional statement tells you something about how control flows (dynamically) during program execution.

A big problem with use of global variables concerns whether a subprogram is *reentrant.* There are times when you need to use two copies of a given subprogram simultaneously, either because your program has more than one thread of control or because a subprogram either directly or indirectly calls itself. Using global variables for communication with a subprogram makes it difficult to have two copies of that subprogram running simultaneously without those copies interfering with one another, since the two will want to store different information in those shared variables. The same problem does not occur when parameters are passed because different copies of a subprogram store their information in different places.

In modern programming, parameter passing (and use of return values from functions) is considered the preferred way to send information either to or back from a subprogram. Chapter 13 discusses implementation of parameter passing using the run-time stack.

16.3. Recursion

Subprograms were introduced to programming languages as a way of providing procedural abstractions, allowing a programmer to add, in effect, new kinds of statements and expressions to the language that are suitable for his or her program. Subprograms also make it possible to break up a large program into smaller units that are easier to understand, and make it possible to provide prewritten libraries of subprograms.

But the introduction of subprograms also opens up the possibility of a new control mechanism, *recursion.* A subprogram can call itself, and the ability to do that can be used as a substitute for loops. Figure 16.2 shows two implementations of a function that computes the sum $1 + 2 + \cdots + n$, one using a loop and one using recursion. Recursion is explored in more detail in later chapters. But it is worth noting that, in order to make use of recursion, a subprogram must be reentrant, since two or more copies (the caller and the callee) will both be active at once. One will be waiting for the other to return, but it remains alive, even if it is paused.

16.4. Syntactic approaches to parameter passing

Procedures typically require parameters given to them by writing them in a specified order; there is a first parameter, a second, and so on. Suppose, for example, that Ada procedure *Simulate* has parameters called *Startval*, *Duration* and *Stepval*, all integers, in that order. Then to call Simulate, you would write

```
Simulate(0, 100, 2);
```

indicating that the Startval is 0, the Duration is 100 and the Stepval is 2. Unfortunately, programmers are rarely good at remembering the correct order, and it is easy to make a mistake, and write the parameters in the wrong order. Ada offers a way around that. You can include the parameter names in the call:

```
Simulate(Startval => 0, Duration => 100, Stepval => 2);
```

```
Define sum(n) = ans |
  open Loop (s, k) = (0, 0)
    exitcase k == n =>
      Let ans = s.
    loopcase else =>
      Continue (s + k, k + 1).
  %Loop
%Define
```

(a)

```
Define sum(n) = ans |
  open If n == 0
    then Let ans = 0.
    else Let ans = sum(n-1) + n.
  %If
%Define
```

(b)

Figure 16.2: **Two implementations of a function sum(n) =** $1+2+\cdots n$**, written in Cinnameg.** Function (a) uses a loop and function (b) achieves similar control using recursion. If you use a run-time stack to allow multiple copies of a subprogram to be running simultaneously, as discussed in Chapter 13, then there is really no difference between recursion and subprogram calls in general. You know, for example, that sum(3) should produce 1 + 2 + 3 = 6. A call to sum(4) produces answer sum(3) + 4 = 6 + 4 = 10.

There are two advantages to that. One is that it is clear which parameter is which. A second is that the programmer does not need to remember the order of parameters. The compiler will reorder them if necessary. So

```
Simulate(Duration => 100, Stepval => 2, Startval => 0);
```

works perfectly well. The programming language Smalltalk takes the point of view that parameter names should always be given, and even drops the procedure name entirely, instead letting the parameter names play the role of procedure name. In Smalltalk, you might ask a particular object sim to perform procedure startval:duration:stepval: as follows.

sim startval: 0 duration: 100 stepval: 2.

Smalltalk does not allow you to reorder the parts. The procedure name is startval:duration:stepval:, not stepval:startval:duration:.

16.5. Parameter passing modes

When a subprogram is defined, the definition needs a way to refer to the parameters that are passed to it. Usually, it gives names to the parameters. Those names, occurring in the subprogram definition, are called the *formal parameters* of the subprogram. If sqr is defined by sqr(x) = x*x, variable x is the formal parameter.

When a subprogram is used, particular items, called the *actual parameters*, or sometimes the *arguments*, are passed to it as its parameters. If you compute sqr(6), then 6 is the actual parameter that is passed to sqr (and referred to by sqr as x).

```
void valincrement(int* p)
{
  *p = *p + 1;
}
```

(a)

```
void refincrement(int& x)
{
  x = x + 1;
}
```

(b)

Figure 16.3: (a) C function valincrement increments a variable, using call-by-value. Type int* is the type of an address of an integer variable, or of a box that holds an integer, and * is the C dereference operator. A similar function (b) in C++ uses call-by-reference, indicated by type int&. The return type of both is void, indicating that no value is returned.

Call-by-value

When a parameter is passed to a subprogram, the question arises whether you intend to pass the rvalue or the lvalue of the parameter expression. For example, when the parameter is the name of a box, do you want to send the content of the box or the box itself? In C, the answer is simple, since only one option is available: you pass the rvalue of the expression. That is called *call-by-value.* For example, C function sqr defined by

```
int sqr(int x)
{
  return x * x;
}
```

takes an integer x that is passed to it using call-by-value, and produces an integer result. Function sqr treats formal parameter x as a variable that receives an initial value at the beginning of the function call. If you write

```
z = sqr(y);
```

then the rvalue of y is placed into sqr's variable x and function sqr runs.

If you want to pass the lvalue of a variable to a subprogram in C, you must explicitly force the lvalue to be used. Operator & does that; it changes the context from right to left, requesting the lvalue of the expression. For example, procedure valincrement, shown in Figure 16.3(a), adds one to a variable. Formal parameter p is a pointer (the box). To pass it variable n (not the value of n), you can write

```
valincrement(&n);
```

Inside valincrement you need to remember that p itself is a variable, and when you get the value of p, what you really get is the box (such as n) that was passed to valincrement. To get the content of n, you need to add a second, explicit, dereference (* in C). When you want to change n, you need to change *p, not p. (Changing p would make p hold a different box.) So the extra * occurs in both left and right contexts.

```
void alias(int& x, int& y)
{
  x = 1;
  y = 1;
  x = x + 1;
  y = y + 1;
}
```

Figure 16.4: **A C++ procedure that illustrates aliasing.** Statement alias(z,z) causes formal parameters x and y to be the same variable, and at the end of alias, both x and y will hold 3.

16.6. Call-by-reference

An alternative convention is to pass the lvalue of the parameter expression implicitly. Fortran IV, for example, always does that. Passing the lvalue of the parameter is called *call-by-reference*.

C++ supports call-by-reference as an option, indicated by &, and Figure 16.3(b) shows an example. When refincrement is called, the lvalue of its actual parameter is passed, not the rvalue. Since the parameter now contains a box, extra dereferences need to be done. But those dereferences are implicitly inserted by the compiler when call-by-reference is used.

The extra dereference is a very real thing in a program; it signifies a machine instruction that performs a memory fetch, and makes access to reference parameters a little slower than access to local variables.

Aliasing

Suppose that statements

```
x = 1;
y = 1;
x = x + 1;
y = y + 1;
```

occur in a C++ program, and that variables x and y each hold integers. Can you say, regardless of the context in which these statements occur, what the values of x and y are after these statements? It turns out that the answer is no. The problem is that, in some contexts, x and y might have been declared to be references, and it is possible for them to refer to the same box. Implicit dereferencing hides this phenomenon, known as *aliasing*. Figure 16.4 shows a procedure, alias, that illustrates.

Aliasing can be a serious problem, primarily because programmers are accustomed to adopting the convenient, but fictional, view that it is the names that are being bound, in which case aliasing can never occur. Programmers are likely to assume that there will be no aliasing, even in contexts where aliasing is a possibility.

```
procedure Sumdiff(X,Y: in integer; Sum,Diff: out integer) is
begin
  Sum := X + Y;
  Diff := X - Y;
end sumdiff;
```

Figure 16.5: **An Ada definition of a procedure to compute the sum and difference of two numbers.** The parameter passing modes are expressed in logical terms **in** and **out**. The actual modes used are typically copy-in and copy-out.

16.7. Copying in and out

When a parameter is passed by value, the value is copied into a subprogram's formal parameter variable when the subprogram is called. For that reason, call-by-value is also called *call-by-copy-in.* For example, when sqr(5) is called, the value 5 is copied into the parameter x of sqr.

There is a complementary mode, *call-by-copy-out*, that is used to copy a value out of a subprogram and into a box owned by another subprogram. Like call-by-reference, it can only be used when the actual parameter is a box. When the subprogram is finished, the value of the formal parameter variable is copied out into that box.

Ada provides an example. Figure 16.5 shows an Ada procedure that computes the sum and difference of two integer values. Ada provides three parameter passing modes, **in**, **out** and **in out**, to tell how parameters are to work, where **in** parameters are only copied in, **out** parameters are only copied out, and **in out** parameters are copied in at the start and out at the end. When statement

```
Sumdiff(A,B,S,D);
```

is performed, the values of A and B are copied in to parameters X and Y, respectively. When Sumdiff is finished, the values of variables Sum and Diff are copied out to variables S and D, respectively.

Mechanism and intent

Notice that call-by-reference could be used instead of copying in or out. Both mechanisms get information into and out of a subprogram. Call-by-reference is more efficient than copying for large data structures, such as arrays, since all that is actually passed is the memory address of the data structure. But call-by-reference is less efficient than copying for small things, such as integers, because of the extra implicit dereferences that must be done each time the parameter is used.

It would be possible for Ada to provide call-by-reference as an alternative, since it is more efficient sometimes. But why should the programmer have to be concerned with which is more efficient? That really should be the compiler's responsibility. What the programmer wants to concentrate on is *how* the information flows; is it just into the subprogram, just out, or both in and out? The *mechanism* for accomplishing the programmer's intent is a separate issue.

An obvious thing to do would be to let the compiler decide to use copying for small things and call-by-reference for large things. Ada does just that. More precisely, the definition of Ada gives the compiler implementor the option to use copying or call-by-reference for *all* parameter passing modes, even mode **in**. Typically, implementors choose copying for small things, but they are not required to. Nor are they required to use references for large things, but they typically do. Ada distinguishes the semantics of parameter passing (the intent) from its implementation (the mechanism).

16.8. Summary of terminology and concepts

Use of subprograms makes it possible to introduce what are at least semantically, if not syntactically, new kinds of statements and expressions.

Parameter passing adds structure to information flow in a way that is similar to how structured programming constructs add structure to control flow.

Having two different values for each expression, an lvalue and an rvalue, suggests allowing more than one parameter passing mode. With call-by-value, the rvalue of the parameter is passed to a procedure. With call-by-reference, the lvalue of the parameter expression is passed. Other parameter passing modes involve copying an item either into the procedure's variables or out of them or both.

Subprograms enable a new control structure, recursion, which can make some algorithms easier to describe than if they need to be written using a loop.

Terminology

Actual parameter An actual parameter is what a program passes to a subprogram that it calls. It tells what the parameter is during a particular call.

Argument The terms actual parameter and argument are synonymous.

Call-by-reference Call-by-reference is a parameter passing mode in which the lvalue of the actual parameter is passed.

Call-by-value Call-by-value is a parameter passing mode in which the rvalue of the actual parameter is passed.

Formal parameter A formal parameter is a placeholder for the actual parameter, and gives a subprogram a way to refer to the actual parameter when the subprogram is called.

Function A function is a kind of subprogram that produces an answer when it is run. It is used to create what is conceptually (but not syntactically) a new kind of expression.

Information flow Information flows from a place in a program where it is produced to other places where it is used. Variables are used for implicit information flow. Parameter passing and returning values from functions make information flow explicit, so that you can see just what information is being sent from one subprogram to another.

Parameter A parameter represents a piece of information that is passed to a subprogram. Using parameters instead of shared variables to send information makes subprograms more general, easier to use and to understand, and reentrant.

Procedure A procedure is, conceptually but not syntactically, a new kind of statement. It performs a specific action.

Recursion Recursion is an implicit control structure that comes out of the ability of subprogram to call itself.

Reentrant A reentrant subprogram can have more than one copy active at once, without those copies interfering with one another.

Subprogram A subprogram is either a procedure or a function.

16.9. Exercises

16.1. What are advantages of Ada's syntax for labeling actual parameters in a procedure call?

16.2. Explain why passing information to or from a subprogram by putting it into global variables is usually considered poor practice. Why do you think languages like C++ allow global variables?

16.3. (This requires some experience with object-oriented programming) Can you pass information implicitly between methods that belong to an object without using global variables or parameters? Explain how. Do you consider this a good way to pass information? Why or why not?

16.4. Using function valincrement from Figure 16.3, carefully show an evaluation of valincrement(&frost) where frost is a variable that holds 4 just before running this statement.

16.5. Using function refincrement from Figure 16.3, carefully show an evaluation of refincrement(frost) where frost is a variable that holds 4 just before running this statement.

16.6. Fortran IV uses only call-by-reference. How can a compiler deal with a parameter that does not have an lvalue? For example, how can a statement JUMP(2) be compiled, using call-by-reference?

16.7. A C++ subprogram is allowed to alter a parameter that is passed by value. The parameter is just a local variable that happens to be initialized to hold the actual parameter, and the subprogram can change what is in that local variable. An Ada subprogram, on the other hand, is not allowed to alter the value of a parameter whose calling mode is **in**. Explain why it is important for Ada subprograms not to allow modification of in-parameters. What could happen if an Ada procedure were allowed to modify a parameter that is only passed into the procedure?

16.8. The definition of Ada does not tell the programmer whether parameters will be passed by copying or by reference. It is possible for the behavior of a program to depend on which is done. Clearly, a programmer should avoid writing anything that depends on unspecified aspects of the semantics. Show how to write an Ada subprogram whose behavior depends on the mechanism that is used for parameter passing. Do not worry about the details of Ada syntax, just use generic syntax. You will need to use parameters that are passed both in and out.

16.10. Bibliographic notes

Wulf [115] argues that global variables can be harmful to program design.

Part VI

Elementary Functional Programming

Chapter 17

Introduction to Functional Programming

17.1. Functional programming

John Backus was one of the leaders in the development of the first higher-level programming language, Fortran. Soon after the development of Fortran, Backus also took a leadership role in the development of another programming language, Algol 60. Taken together, Fortran and Algol 60 have had a tremendous influence on the design of programming languages. The Algol 60 report was the first to use Backus-Naur Form (Chapter 6) to give a formal description of the syntax of a programming language, and similar descriptions of syntax have become common practice for new languages. In recognition of his contributions to programming languages, Backus was awarded the Turing Award in 1978. In his Turing Award lecture, you might have expected Backus to discuss the legacy of what he had done. And, in a sense, he did. He said that languages in the families of Fortran and Algol 60 represented the wrong way to program, and advocated a very different style: functional programming.

Functional programming is a declarative style based on defining functions using equations. The foundational ideas are very close to familiar mathematics. Instead of two concepts, expressions and statements, there is only one, expressions. There are no variables that can be changed. In fact, nothing can ever be changed. Instead, you compute new values from old. Errors due to changing variables or data structures at inappropriate times, which are very common in imperative programs, can never happen in functional programs.

Functional programming tends to give very short and simple solutions to a wide variety of programming problems, and programs tend to work with a minimal amount of debugging. Functional programming consequently offers an economical way to design reliable software.

17.2. Functional programming in this book

Although it has tremendous advantages for some kinds of problems, functional programming is more like what most students have studied in mathematics courses than like what they have used for computing, and it takes some work to understand what functional programming is and how to use it. This part introduces elementary ideas of functional programming in Chapter 18 in a mathematical style that is independent of any particular programming language, keeping the ideas clear and simple. But without a programming language to support those ideas, they are sterile. Chapter 20 casts functional ideas in Cinnameg, with very little change from their mathematical forms.

The style that Backus advocated, called higher-order functional programming, carries programs into a realm where the parameters of functions can be other functions, and where functions are not just written in a static form by a programmer, but can be built by the program itself on the fly. That opens up the possibility of very powerful functions that do not just implement one specific algorithm, but implement an entire class of algorithms. For example, the idea of linear search can be employed to find a value in a list or array with a given property. In higher order functional programming, not only the list but also the property that is sought can be a parameter to a function that performs linear search, making that function highly reusable. Chapter 21 explores those ideas.

Declarative programs, of which functional programming is variation, need to be implemented on a processor that inherently works in an imperative, command-driven way. Although Chapter 18 describes the basics of how functional programs are run in a step by step fashion, details are left open. Chapter 22 explores some important ideas for running functional programs.

Given the level of success that the ideas of functional programming enjoy for a wide variety of problems, it is surprising that it is not used more extensively. The most obvious reason is that programmers, having learned to express algorithms in an imperative style where programs are expressed in terms of command, have difficulty converting those ideas into a style where the commands are hidden behind facts. Chapter 19 shows how common imperative algorithms are converted into functional specifications that can be simpler than the original algorithms.

17.3. Functional programming languages

This part uses Cinnameg to express functional programs in a concrete, executable form. But it is important to see how the same ideas look in other languages. Chapter 23 covers the basics of Scheme, a dialect of Lisp, and Chapter 24 discusses aspects of Haskell, a language that supports only functional ideas, and nothing else.

17.4. Summary

Functional programming is a declarative style of programming based on equations. Programs written in a functional style tend to be short and simple. It is common to find that they work immediately, without any time-consuming debugging.

17.5. Bibliographic notes

Backus describes his views on functional programming languages in his Turing Award address [11]. Hudak [59] gives a history of the ideas of functional programming languages. Appel [6] shows how to write a compiler in a functional style using ML. Wiger [109] describes an industrial project that is controlled by about two million lines of functional code, and that has yielded extremely high reliability.

Chapter 18

Computing with Equations

18.1. Functions and equations

A *function* is a central concept of both computation and mathematics. In mathematics, you typically define a function by an equation, such as

$$f(x) = 3(x) + 2, \tag{18.1.1}$$

telling you, for example, that $f(6) = 3(6) + 2 = 20$. From a computational standpoint, a function is a tool that takes an input (called its argument or parameter) and produces an output, or result, just as f takes argument 6 and yields result 20.

Equational programming is a form of declarative programming where a program consists of equations stating facts similar to Equation 18.1.1. Computation is performed by doing substitutions, using the available facts. For example, to determine $f(10)$, where f is defined by Equation 18.1.1, substitute 10 for x, yielding $f(10) = 3(10) + 2 = 32$.

This chapter explores equational programming in a mathematical style, without the burden of any particular programming language. Later chapters look at how these ideas can be exploited by a programming language.

18.2. Making decisions

Some functions can be defined by a single equation, but others, such as the function $\max(x, y)$ that produces the maximum of x and y, appear to need a mechanism for making decisions; you would expect a rule for computing $\max(x, y)$ to involve comparing x and y to decide which is larger. A first step toward making decisions is to allow expressions to produce boolean values, not just numbers. For example, expression $x > y$ produces value **true** if x is greater than y, and **false** otherwise. Then, to make decisions, you can use a *conditional* function $\mathrm{cond}(a, b, c)$ that yields b if a is true, and yields c if a is false. For example, $\mathrm{cond}(\mathrm{true}, 4, 2) = 4$ and $\mathrm{cond}(\mathrm{false}, 4, 2) = 2$. So max is defined by equation $\max(x, y) = \mathrm{cond}(x \geq y, x, y)$. A slightly more involved function is $\mathrm{sign}(x)$, defined to be -1 when x is negative, 0 when $x = 0$ and 1 when x is positive. It is defined by the following equation.

$$\mathrm{sign}(x) = \mathrm{cond}(x > 0, 1, \mathrm{cond}(x < 0, -1, 0)) \tag{18.2.1}$$

Figure 18.1 shows computation of sign(4) by doing repeated substitutions.

$$\begin{aligned}
&\mathrm{sign}(4)\\
&\quad = \mathrm{cond}(4 > 0, 1, \mathrm{cond}(4 < 0, -1, 0)\\
&\quad = \mathrm{cond}(\mathrm{true}, 1, \mathrm{cond}(\mathrm{false}, -1, 0))\\
&\quad = \mathrm{cond}(\mathrm{true}, 1, 0)\\
&\quad = 1
\end{aligned}$$

Figure 18.1: **Evaluation of** sign(4) **by substitution, using Equation 18.2.1.** (Was it really necessary to evaluate cond$(4 < 0, -1, 0)$, or could it have been left as an unevaluated expression?)

Using more than one equation

Although using a conditional function is all that is required to make decisions, an alternative is to write more than one equation, with a proviso indicating when each equation is true. Two equations suffice to define max.

$$\begin{array}{lcll}
\max(x, y) & = & x & \text{when } x \geq y\\
\max(x, y) & = & y & \text{when } x < y
\end{array}$$

To compute max(4, 6), you notice that only the second equation is true, since only its proviso is true, so that is the equation that you use. There can be any number of equations. For example, the sign function involves three equations.

$$\begin{array}{lcll}
\mathrm{sign}(x) & = & 1 & \text{when } x > 0\\
\mathrm{sign}(x) & = & 0 & \text{when } x = 0\\
\mathrm{sign}(x) & = & -1 & \text{when } x < 0
\end{array}$$

18.3. Recursion

Some functions, such as factorial$(n) = n \cdot (n-1) \cdot \cdots \cdot 2 \cdot 1$, need more steps to evaluate as their arguments become larger. Equational programming handles that using recursion; a function can appear on the right-hand side of its defining equations. For example, notice that factorial$(n) = n \cdot$ factorial$(n-1)$ for $n > 1$. In fact, if we define factorial(0) = 1, which is the usual rule, then factorial$(n) = n \cdot$ factorial$(n-1)$ for all $n > 0$, leading to the following definition of factorial.

$$\begin{array}{lcll}
\mathrm{factorial}(n) & = & 1 & \text{when } n = 0\\
\mathrm{factorial}(n) & = & n \cdot \mathrm{factorial}(n-1) & \text{when } n > 0
\end{array} \qquad (18.3.1)$$

Computing factorial(3) involves doing repeated substitutions, as shown in Figure 18.2.

18.4. Pattern matching

Look more carefully at evaluation of factorial(3). Equations 18.3.1 tell you how to compute factorial(n), but what you want is factorial(3). To make them the same, you solve equation $n = 3$ to determine a suitable value for n. It seems silly to consider this task as “solving” an equation, but looking at the substitution process as a matter of solving equations opens

$$\begin{aligned}
&\text{factorial}(3)\\
&\quad = 3 \cdot \text{factorial}(3-1)\\
&\quad = 3 \cdot \text{factorial}(2)\\
&\quad = 3 \cdot (2 \cdot \text{factorial}(1))\\
&\quad = 3 \cdot (2 \cdot (1 \cdot \text{factorial}(0)))\\
&\quad = 3 \cdot (2 \cdot (1 \cdot 1))\\
&\quad = 6
\end{aligned}$$

Figure 18.2: **Evaluation of** factorial(3) **using Equations 18.3.1.** At each step, the leftmost expression is found that can be evaluated, and that does not contain any subexpressions that can be evaluated. For example, the equation for factorial is not used for factorial($3-1$) since it contains the smaller expression $3-1$, which can be evaluated to 2. In later steps, arithmetic simplifications are performed immediately for brevity.

the possibility of slightly more involved equations. For example, suppose that you are told the following equation (with a proviso) about the factorial function.

$$\text{factorial}(n+1) = (n+1) \cdot \text{factorial}(n) \quad \text{when } n \geq 0 \tag{18.4.1}$$

Now to compute factorial(3), you want $n+1=3$, or $n=2$, and substituting $n=2$ into Equation 18.4.1 tells you that factorial(3) = $(2+1) *$ factorial(2). There is not much advantage here, but we will encounter situations shortly where this kind of thing significantly simplifies equations.

When an expression such as $n+1$ is used on the left-hand side of an equation, it is called a *pattern*. Patterns are allowed in equational programming when it is clear how to solve the simple equations that arise from using them, so that their solutions can be done automatically in the course of performing substitutions. The mechanisms for solving them must be part of the language implementation, or must be specified somehow in a program. We will assume that, at a minimum, the following equations can be solved for x and y, given a and b.

$$\begin{aligned}
x+1 &= a\\
(x,y) &= (a,b)
\end{aligned}$$

(The only solution to equation $(x,y)=(a,b)$ is $x=a$ and $y=b$.) Pattern forms can be mixed. For example, equation jump$(x, y+1) = x \cdot y$ can be used to evaluate jump(5,9) by solving pattern match equation $(x, y+1) = (5,9)$, with solution $x=5$ and $y=8$.

Another kind of pattern is a constant. For example, the basis case for factorial can be simplified to

$$\text{factorial}(0) = 1, \tag{18.4.2}$$

where the pattern is just the constant 0. To evaluate factorial(0), you solve equation $0 = 0$, which is trivially true. Since there are no variables, just use Equation 18.4.2 as is. Since equation $0=1$ has no solution you cannot use Equation 18.4.2 to compute factorial(1). In that case, select a different equation.

18.5. The form of an equational program

Up to now, we have introduced new ideas as they were needed. But it is important to settle on the rules of what an equational program looks like and how it is used in computation. An equational definition of a function f consists of a collection of one or more equations, each of the form

$$f(p) = r \text{ when } c$$

or

$$f(p) = r$$

where p is a pattern expression, c is a condition (an expression that produces a boolean value) and r is an expression that gives the answer. The pattern expression p must have a form such that an algorithm is available that solves equation $p = v$, where v is an abitrary given value. The algorithm either gives a value for each variable that occurs in p or says that no solution to equation $p = v$ exists.

18.6. The mechanics of substitution

Mathematically, equations are symmetric; saying that $A = B$ is the same as saying that $B = A$. But from the standpoint of computation, equations are not symmetric; you always replace the left-hand side of an equation by its right hand side, not the other way around. So, during computation, Equation 18.1.1 allows you to replace $f(x)$ by $3(x) + 2$, but never $3(x)+2$ by $f(x)$. Other than that restriction, any kind of substitution can be done. Suppose, for example, that g is defined by $g(x, y) = x + x \cdot y$. Substituting (5+4) for x and $(9 + 2)$ for y tells you that $g((5 + 4), (9 + 2)) = (5 + 4) + (5 + 4) \cdot (9 + 2)$.

Be careful when you substitute expressions for variables. A naive substitution of $5+4$ for x and $9+2$ for y yields $g(5+4, 9+2) = 5+4+5+4\cdot 9+2$. Because of precedence rules, that is not correct. In general, put parentheses around expressions that are substituted, unless the parentheses are not required to maintain the correct meaning.

To evaluate an expression, you perform substitution steps until a form is reached that does not need further evaluation. Each step proceeds as follows.

1. Select a subexpression of the form $f(e)$, where e can be any expression, for evaluation. How the selection is made is discussed in the next section under evaluation policies.

2. Select an equation [$f(p) = r$ when c] or [$f(p) = r$] that is part of the definition of f. How this choice is made is also a policy decision.

3. Solve pattern match equation $p = e$ for all of the variables in pattern p, treating expression e as a unit not to be changed. (That is, if e is not just a value, do not try to get its value here.) But not all equations have solutions. If there is no solution, then select a different equation, since this one cannot be used. If there is a solution, proceed to the next step.

4. If there is a proviso [when c], substitute the solution found in Step 3 into expression c, yielding expression c'. Evaluate c'. If it yields false, then this equation cannot be used; select another equation. If c' is true, continue with the next step.

5. Substitute the solution found in step 3 into expression r, yielding modified expression r'. Replace $f(e)$ by r' to complete the substitution.

For example, suppose that function $s(x, y)$ is defined for integers (positive and negative) x and y by

$$\begin{array}{rcll} s(0,y) & = & y & \\ s(x,y) & = & s(-x,y) & \text{when } x < 0 \\ s(x+1,y) & = & s(x,y+1) & \text{when } x \geq 0 \end{array} \tag{18.6.1}$$

and you need to evaluate s(1,7). An attempt to use the first equation requires solving pattern match equation $(0, y) = (1, 7)$, which has no solution. The second equation has a successful pattern match (solution: $x = 1$ and $y = 7$), but its proviso is false. The third equation has both a successful pattern match (solution: $x = 0$ and $y = 7$), and a true proviso. Using it yields replacement $s(1, 7) = s(0, 7 + 1)$. Continue doing substitutions on $s(0, 7 + 1)$.

Your mathematical experience encourages you to use some well-known facts about arithmetic operations, such as the associative law of addition, $x + (y + z) = (x + y) + z$. But when performing substitutions, all you have is a set of defining equations for each function, and the associative law is not one of those equations. Suppose that $r(n)$ is defined by

$$\begin{array}{rcll} r(0) & = & 0 & \\ r(n) & = & n + r(n-1) & \text{when } n > 0 \end{array}$$

Evaluation of $r(3)$ begins as follows.

$$\begin{array}{rcl} r(3) & = & 3 + r(3-1) \\ & = & 3 + r(2) \\ & = & 3 + (2 + r(2-1)) \end{array}$$

At this step, it is tempting to rearrange the parentheses and convert this to $5 + r(1)$. But that is not how evaluation is done. You must compute in the original order, as dictated by the parentheses.

18.7. Evaluation policies

When you go to evaluate an expression, there can be two or more substitutions available to you. For example, to evaluate $f(g(2)) + h(10)$, you could use an equation for g or for h or even for f. An evaluation policy determines which one to choose, usually with the goal of performing an efficient or simple evaluation.

A common evaluation policy is *left-to-right inside-out* evaluation, commonly abbreviated to *inside-out evaluation.* At each step, an expression is chosen for replacement that does not contain any smaller subexpressions that could also be replaced. If there are several choices, the leftmost one is chosen. Figure 18.2 shows an inside-out evaluation. Inside-out evaluation usually works well, but sometimes it needs to be modified. For example, to evaluate cond(a, b, c), it does not make sense to evaluate all of a, b and c right away. Instead, it suffices to evaluate a, and then to continue either by evaluating b (if a is true) or c (if a is false). Figure 18.1 shows an evaluation that can be shortened by that rule.

There are other evaluation policies that might be chosen. One is pure left-to-right evaluation, which selects the expression that begins at the leftmost place, and that has an available rule. That rule evaluates $f(g(x))$ by selecting an equation for f rather than one for g. Evaluation policies are examined more in Chapter 22.

Choosing among equations

After a subexpression is chosen, an evaluation policy also needs to select an equation that can be used for it. A typical policy is to try the equations in the order in which they are written until one that can be used is found. But the equations can, in principle, be tried in any order; the next chapter has something to say about that.

Sometimes more than one equation gives an answer for a particular expression. For example, if the max function is defined by

$$\begin{array}{rcl} \max(x, y) & = & x \text{ when } x \geq y \\ \max(x, y) & = & y \text{ when } y \geq x \end{array}$$

then both equations can be used to compute max(4,4). There is nothing wrong with that, as long as both equations give the same answer. For example, if the equations are tried in the order written, the first one will be used.

18.8. Equational programming with lists

Up to now we have only defined functions on numbers, but equational programming shows its real power when used with data structures such as lists and trees, where functions often have short and simple definitions. This section illustrates functions involving lists.

We need some basic operations on lists to get going. It suffices to have one constant, $[\,]$, standing for an empty list, and four functions: head(x) is the head of list x (its first member); tail(x) is the tail of list x (all but its first member); null(x) is true if $x = [\,]$, and false otherwise; and $h :: t$ is the list whose head is h and whose tail is t. The empty list does not have a head or a tail, so both head$([\,])$ and tail$([\,])$ are undefined. We will write $x = [\,]$ in programs to mean null(x). But we only use this to test whether a list is empty. A general list equality test is not one of the basic functions.

The :: operator needs some explanation. The definition is that $h :: t$ is the list whose head is h and whose tail is t. Since head$([1, 2, 3, 4]) = 1$ and tail$([1, 2, 3, 4]) = [2, 3, 4]$, it must be the case that $1 :: [2, 3, 4] = [1, 2, 3, 4]$. The :: operator adds one value to the beginning of a list. Do not expect it to do more than that. It does not add a value to the end of a list, and it does not concatenate lists. For example, $[1, 2, 3] :: [4, 5, 6] = [[1, 2, 3], 4, 5, 6]$, not $[1, 2, 3, 4, 5, 6]$. By convention, operator :: associates to the right, which means that $a :: b :: c$ is understood to mean $a :: (b :: c)$. So $5 :: 3 :: [\,] = 5 :: [3] = [5, 3]$. Notice the empty list at the end of $5 :: 3 :: [\,]$; it is necessary. Expression $a :: b$ is not the same as $[a, b]$.

If lists are represented in linked form, then each of the four operations (null, head, tail and ::) takes constant time to perform. For example, computing $h :: t$ just requires building one list cell, and computing tail(x) just involves getting the "next" pointer.

Computing the length of a list

Suppose you need a function length(x) that tells how many members list x has. For example, length$([5, 6, 7]) = 3$ and length$([\,]) = 0$. Notice that the length of a nonempty list is one greater than the length of its tail, suggesting the following definition of length.

$$\begin{array}{rcll} \text{length}(x) & = & 0 & \text{when } x = [\,] \\ \text{length}(x) & = & 1 + \text{length}(\text{tail}(x)) & \text{when } x \neq [\,] \end{array} \qquad (18.8.1)$$

Pattern matching with lists

Pattern matching equations that involve the :: operator are straightforward to solve. For example, equation $h :: t = [2,\ 5,\ 7]$ has exactly one solution, $h = 2$ and $t = [5, 7]$. In general, equation $h :: t = v$ has solution $h = \text{head}(v)$ and $t = \text{tail}(v)$. Equation $h :: t = [\,]$ has no solution, so pattern $h :: t$ can only match a nonempty list. Using patterns yields a simpler definition of length.

$$\begin{array}{rcl} \text{length}([\,]) & = & 0 \\ \text{length}(x :: xs) & = & 1 + \text{length}(xs) \end{array} \tag{18.8.2}$$

Note that the provisos are no longer needed; pattern $x :: xs$ can only match a nonempty list.

Is a given list in ascending order?

Suppose that ascend(xs) is supposed to be true if list xs is in ascending order. For example, ascend($[2, 4, 6, 7]$) = true, but ascend($[2, 4, 3, 5]$) = false. For long lists, you check whether the order is ascending by checking whether the first two members are in the correct order, and then checking whether the tail of the list is also in ascending order. But that only makes sense if the list has at least two members. Smaller lists need to be handled as special cases.

$$\begin{array}{rcl} \text{ascend}([\,]) & = & \text{true} \\ \text{ascend}([a]) & = & \text{true} \\ \text{ascend}(a :: b :: r) & = & a < b \text{ and ascend}(b :: r) \end{array} \tag{18.8.3}$$

Membership test

Now let's define function member(x, ys), which yields true if x is a member of list ys. For instance, member(3, $[2, 5, 3, 6]$) = true, but member(3, $[2, 5, 7]$) = false. The empty list has no members, so member(x, $[\,]$) is certainly false. Evidently, x is in $y :: ys$ if either $x = y$ or x is a member of ys, leading to the following definition of member.

$$\begin{array}{rcl} \text{member}(x, [\,]) & = & \text{false} \\ \text{member}(x, y :: ys) & = & x = y \quad \text{or} \quad \text{member}(x, ys) \end{array} \tag{18.8.4}$$

Taking a prefix of a list

Suppose that you want to extract the first n members of a given list. Say that take(3, $[2, 4, 6, 8, 10]$) = $[2, 4, 6]$, and, in general, take(n, x) yields the length n prefix of list x. There is a special case where take is asked to take more than is available. In that case, by convention, say that it takes the entire list. For example, take(20, $[200, 400]$) = $[200, 400]$. Equations for take are straightforward to derive.

$$\begin{array}{rcll} \text{take}(0,\ xs) & = & [\,] & \\ \text{take}(n,\ [\,]) & = & [\,] & \\ \text{take}(n,\ x :: xs) & = & x :: \text{take}(n-1,\ xs) & \quad \text{when } n > 0 \end{array} \tag{18.8.5}$$

The last equation uses the :: operator to build the list. Notice that take(3, $[a,\ b,\ c,\ d]$) = $[a,\ b,\ c] = a :: [b,\ c] = a :: \text{take}(2,\ [b,\ c,\ d])$. Figure 18.3 shows an inside-out evaluation of take(2, $[a,\ b,\ c,\ d]$).

$$\begin{aligned}
&\text{take}(2,\ [a,\ b,\ c,\ d])\\
&\quad = a :: \text{take}(1,\ [b,\ c,\ d])\\
&\quad = a :: b :: \text{take}(0,\ [c,\ d])\\
&\quad = a :: b :: [\,]\\
&\quad = a :: [b]\\
&\quad = [a,\ b]
\end{aligned}$$

Figure 18.3: **An inside-out evaluation of expression take(2, [a, b, c, d]) using Equations 18.8.5.**

$$\begin{aligned}
&\text{cat}([2,\ 3,\ 4],\ [5,\ 6])\\
&\quad = 2 :: \text{cat}([3,\ 4],\ [5,\ 6])\\
&\quad = 2 :: 3 :: \text{cat}([4],\ [5,\ 6])\\
&\quad = 2 :: 3 :: 4 :: \text{cat}([\,],\ [5,\ 6])\\
&\quad = 2 :: 3 :: 4 :: [5,\ 6]\\
&\quad = 2 :: 3 :: [4,\ 5,\ 6]\\
&\quad = 2 :: [3,\ 4,\ 5,\ 6]\\
&\quad = [2,\ 3,\ 4,\ 5,\ 6]
\end{aligned}$$

Figure 18.4: **Inside-out evaluation of cat([2, 3, 4], [5, 6]) using Equations 18.8.6.** Be careful to remember that :: associates to the right, so evaluation must be done in the order shown. (It would not make sense to try to evaluate $2 :: 3$, since that is a meaningless expression).

Concatenating lists

Suppose that you want to concatenate, or join, two lists. Say that cat([2, 3], [4, 2]) = [2, 3, 4, 2] and cat([7, 4, 3], [8, 9, 10]) = [7, 4, 3, 8, 9, 10]. Notice that cat([2, 3, 4], [6, 5, 2]) = 2 :: 3 :: 4 :: [6, 5, 2], and

$$\begin{aligned}
\text{cat}([a,\ b,\ c,\ d], y) &= a :: b :: c :: d :: y\\
&= a :: (b :: c :: d :: y)\\
&= a :: \text{cat}([b,\ c,\ d], y).
\end{aligned}$$

In general, $\text{cat}(x :: xs,\ ys) = x :: \text{cat}(xs, ys)$. That handles the case where the first parameter is not an empty list. But the case of an empty first parameter is obvious, leading to Equations 18.8.6.

$$\begin{aligned}
\text{cat}([\,], ys) &= ys\\
\text{cat}(x :: xs, ys) &= x :: \text{cat}(xs, ys)
\end{aligned}\tag{18.8.6}$$

Figure 18.4 shows an inside-out evaluation of expression cat([2, 3, 4], [5, 6]).

18.9. Summary of terminology and concepts

Equational programming is a declarative programming style where a function is defined by a small collection of equations. Computation is performed by doing substitutions.

Fundamental design tools for equational programming are: (1) choice (using provisos where needed) among different equations to deal with the need for making decisions; (2) recursion to handle the need to perform arbitrarily many operations, and (3) pattern matching to make equations simpler. Equational programming works well on lists, yielding short and simple definitions of functions.

Terminology

$[\,]$ — $[\,]$ is the empty list.

$h :: t$ — $h :: t$ is the list whose head is h and whose tail is t.

Conditional function — The conditional function $\text{cond}(a, b, c)$ is defined to yield b when a is true, and c when a is false.

Equation — In an equational program, an equation is a basic fact about a function f in which the left-hand side has the form $f(\ldots)$, and the right-hand side is an expression. It is used during computation to replace the left-hand side by the right-hand side, after suitable specialization.

Evaluation mechanism — Substitution is an evaluation mechanism. It is the means by which basic steps of evaluation are carried out.

Evaluation policy — An evaluation policy chooses the next substitution to do from all that are available.

Head — The head of a list is the first member of a list. The head of $[\,]$ is undefined.

Inside-out evaluation — This is a policy for evaluating an expression that chooses an innermost expression to evaluate first. Left-to-right inside-out evaluation additionally specifies that, if there are two or more innermost expressions, the leftmost one should be chosen.

Pattern matching — Pattern matching is used both to restrict the circumstances under which an equation can be used and to obtain values related to a value of interest. For example, equation $\text{length}([\,]) = 0$ indicates a restriction: the parameter must be $[\,]$ in order for this equation to be used. Equation $\text{length}(x :: xs) = 1 + \text{length}(xs)$ is partly restrictive (it indicates that the parameter must be a nonempty list for this equation to be used), but it also extracts the head x and tail xs from the list.

Proviso — A proviso is an extra condition that must be true in order for an equation to be used.

Recursion — Recursion is a fundamental tool for equational programming in which a function can be used in its own definition.

Substitution — In order to evaluate an expression, you must perform substitutions. For example, if you have concluded that $g(2) = 5$ and you need to evaluate $(g(2) + 1)$, you substitute to get $5 + 1$.

Tail — The tail of a list x is the list obtained by removing the first member of x. The tail of $[\,]$ is undefined.

18.10. Exercises

18.1. What is the value of each of the following expressions? Read them carefully.

(a) 2 :: [ ]

(b) 3 :: [4,7]

(c) [3 :: [4,7]]

(d) 2 :: 4 :: 5 :: [ ]

(e) head([2 :: [3], 4 :: [5]])

(f) tail([3])

(g) tail([4,5])

18.2. Show a left-to-right inside-out evaluation of length($[1, 2, 3, 4]$), using Equations 18.8.1. Be careful not to make use of the associative law of addition.

18.3. Using Equations 18.8.5, show a left-to-right inside-out evaluation of take(2, $[4, 3, 2]$).

18.4. Using Equations 18.8.5, show a left-to-right inside-out evaluation of take(5, $[4, 3, 2]$). Why was no special case needed to check whether the list has length less than n?

18.5. Given the definition

$$\begin{aligned} f([\,]) &= [\,] \\ f(x :: xs) &= (x * x) :: f(xs) \quad \text{when } x > 10 \\ f(x :: xs) &= f(xs) \quad \text{when } x \le 10 \end{aligned}$$

show an inside-out evaluation of expression f($[4, 12, 15, 6, 11]$). Show the entire expression at each step. Assume that arithmetic is done as soon as possible.

18.6. Show that the number of substitution steps needed to compute cat(x, y) is $2n + 1$, if n is the length of list x.

18.7. The cat function uses memory sharing. Show a linked list diagram of lists $[1, 2]$ and $[3, 4]$, as well as the result of computing cat($[1, 2]$, $[3, 4]$), showing the memory sharing. Is memory sharing safe for equational programming, or is it likely to cause trouble?

18.8. Write an equational definition of function sum(x) that produces the sum of a list of numbers. For example sum($[1, 2, 3, 4]$) $= 1 + 2 + 3 + 4 = 10$.

18.9. For a positive integer k, say that next(k) $= k/2$ when k is even, and next(k) $= 3k + 1$ when k is odd.

(a) Write an equational definition of next.

(b) The *hailstone sequence* starting at n is a list where each number k is followed by next(k), until the list ends on 1. For example, the hailstone sequence starting at 13 is $[13, 40, 20, 10, 5, 16, 8, 4, 2, 1]$. Write an equational definition of hailstone(n), yielding the hailstone sequence starting at n.

(c) Define a function hailstoneLength(n) that returns the length of the hailstone sequence that starts at n. For example, hailstoneLength(13) $= 10$. Give a definition that uses the length and hailstone functions.

(d) Give a direct definition of hailstoneLength that does not use the length or hailstone functions.

18.10. Write an equational definition of function digitsum, where digitsum(n) is the sum of the digits in decimal number n. For example, digitsum(554) = 5 + 5 + 4 = 14. Use operators **div** and **mod** where x **div** y is the integer quotient when x is divided by y, and x **mod** y is the remainder.

18.11. Using your solution to Exercise 18.10, show an inside-out evaluation of digitsum(3942).

18.12. Write an equational definition of function seven, where seven(n) returns true if decimal number n has any 7's in it. For example, seven(372) = true and seven(29) = false.

18.13. Using your solution of Exercise 18.12, show an inside-out evaluation of expression seven(9725).

18.14. Write an equational definition of function dup where dup(n, x) produces a list of n copies of x. For example, dup(3, 20) = $[20, 20, 20]$.

18.15. Write an equational definition of a function stutter(x) that takes a list x and produces a list with each member of x duplicated. For example, stutter([a, b, c, d]) = [a, a, b, b, c, c, d, d].

18.16. A *predicate* is a function that produces a boolean result. Write an equational definition of a predicate called prefix so that prefix(x, y) is true just when list x is a prefix of list y. For example, prefix($[1, 2]$, $[1, 2, 4, 6]$) is true, but prefix($[1, 2]$, $[1, 5, 2, 6]$) is false. The empty list is a prefix of every list, and every list is a prefix of itself.

18.17. Show an inside-out evaluation of prefix($[1, 2]$, $[1, 2, 3, 4]$) using your definition of prefix from the preceding question. Show the entire expression at each step.

18.18. The zip function takes a pair of lists and produces a list of pairs. For example, zip($[a, b, c]$, $[d, e, f]$) = $[(a, d), (b, e), (c, f)]$. (Compare that to the action of a zipper.) If the two lists have different lengths, zip goes only to the length of the shorter list, so zip($[a, b, c]$, $[d]$) = $[(a, d)]$. Write a definition of zip.

18.11. Bibliographic notes

Equational programming got its start with recursive function theory [34], before there were any modern computers. Languages that support an equational style include ML [86], Haskell [16, 103] and Miranda [104]. Lisp [45, 100] is an early language that supports functional programming, but its unusual syntax tends to hide the equational nature of function definitions.

Chapter 19

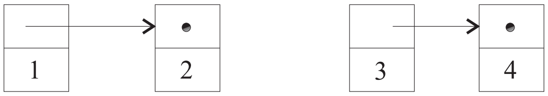
Are Imperative Constructs Necessary?

19.1. Introduction

Although the equational style has some advantages over the imperative style, it is an acquired taste. This chapter compares and contrasts equational programming with imperative programming, not only to see where the differences lie, but to help understand how to modify imperative ideas so that they work in the equational setting.

19.2. Persistent data structures

If you are accustomed to working with data structures such as linked lists in an imperative or object-oriented programming language, your natural tendency will be to think in terms of modifying a data structure. Take, for example, the problem of concatenating two lists. If you think in terms of a linked representation of lists, you imagine two lists $[1, 2]$ and $[3, 4]$ represented as follows.



You will be tempted to perform the concatenation by changing the null link in the list node that contains 2 so that it points to the list node that contains 3. There are two serious problems with that for the setting of functional programming. One is that it changes what is stored in a list node. That is, it treats the list node as a *variable*. In equational programming, there are no variables. You must compute new values from old ones without modifying the old values. A second problem is that no operation has been provided for modifying a list. Only the constant [ ] and the four functions null, ::, head and tail are available. The list operations are pure functions; they do not call for changing anything.

Data structures that support operations that modify the data are called *ephemeral* data structures because information is lost when an update is performed. A data structure that has only pure functions defined on it is called a *persistent* data structure, since anything that is created persists, and cannot be destroyed, except when it becomes inaccessible to the program, and can be recycled by a garbage collector. Functional programming uses only persistent data structures. For example, Equations 18.8.6 define concatenation as as a function, computing the new list from the old ones.

Example: Sorting a list

A sorting function takes a list and reorders it into ascending (or, more generally, nondescending) order. Of course, in an equational program, you cannot actually change the list; instead, a sorting function produces a new list. For example, $\text{sort}([5,2,7,1]) = [1,2,5,7]$.

Sorting an empty list is easy: $\text{sort}([\,]) = [\,]$. So all that is left is to see how to sort a nonempty list; an example helps to see what $\text{sort}(x :: xs)$ is. To sort list $[6,7,5,4,3]$, first sort the tail $[7,5,4,3]$, yielding $[3,4,5,7]$. Now insert 6 where it goes, yielding $[3,4,5,6,7]$. The insertion can be done by another function, insert, were $\text{insert}(6,\ [3,4,5,7]) = [3,4,5,6,7]$. More generally, $\text{insert}(x,\ L)$ produces a sorted version of $x :: L$, but requires that L is already sorted. (The behavior of $\text{insert}(x,\ L)$ is unspecified when L is not sorted.) That suggests the following definition of sort, which implements an algorithm called *insertion sort.*

$$\begin{aligned} \text{sort}([\,]) &= [\,] \\ \text{sort}(x :: xs) &= \text{insert}(x,\ \text{sort}(xs)) \end{aligned}$$

Three cases suffice to define $\text{insert}(x,\ L)$, one for $L = [\,]$, one for the case where x goes at the beginning of the result list, and one for the case where x does not belong at the beginning.

$$\begin{aligned} \text{insert}(x,\ [\,]) &= [x] && \\ \text{insert}(x,\ y :: ys) &= x :: y :: ys && \text{when } x \le y \\ \text{insert}(x,\ y :: ys) &= y :: \text{insert}(x,\ ys) && \text{when } x > y \end{aligned}$$

19.3. Loops and equational definitions

Programmers who are accustomed to programming with loops sometimes have a difficult time adapting to a form of programming that uses recursion in place of loops. But, with an improved understanding of what a loop is accomplishing, conversion to recursion is usually not only simple, but sometimes yields a shorter and clearer definition than a loop would.

Example: List reversal

Suppose that you would like to define function $\text{reverse}(x)$ to yield the reversal of list x, so that reverse([2,5,3,6]) = [6,3,5,2]. A fairly simple recursive definition, called *naive reverse*, is as follows. We use operator ++ to indicate concatenation of lists (function cat from Chapter 18).

$$\begin{aligned} \text{reverse}([\,]) &= [\,] \\ \text{reverse}(x :: xs) &= \text{reverse}(xs) +\!+ [x] \end{aligned} \tag{19.3.1}$$

Notice that $\text{reverse}([a,\ b,\ c]) = [c,\ b,\ a] = [c,\ b] +\!+ [a] = \text{reverse}([b,\ c]) +\!+ [a]$. The second equation above generalizes that observation.

Unfortunately, naive reverse can be very time-consuming (justifying its name). Exercise 18.6 asks you to show that concatenating two linked lists using function cat takes time proportional to the length of the first list. Reversing $[a, b, c, d]$ requires doing concatenations

[*d*] ++ [*c*], [*d*, *c*] ++ [*b*] and [*d*, *c*, *b*] ++ [*a*]. In general, the total amount of work used to reverse a list of length n is proportional to $1 + 2 + \cdots (n-1) = n(n-1)/2$.

There is no reason to believe that reversing a list should be so expensive, and it is not, if you just adopt a better approach. Think about reversing a deck of cards. You start by holding the deck in your hand. You remove the top card, and place it in a new stack on a table. Then you successively remove cards from the original deck in your hand and place them on top of the stack on the table, until eventually you have the entire deck on the table, in reverse order. That algorithm is naturally expressed as a loop; the following loop sets variable *answer* to the reversal of list x.

```
$y$ = $x$
$r$ = [ ]
while $y \neq [\,]$ do
  $r$ = (head $y$) :: $r$
  $y$ = tail $y$
end while
answer = $r$
```

Typically, a loop has some *loop control variables* that change as the loop runs. Our reversal loop has two loop control variables, y and r, where you can think of y as the deck of cards in your hand and r the deck on the table. Look at the values that y and r take on each time the top of the loop is reached, for the example where $x = [1, 2, 3, 4]$.

y	r
$[1, 2, 3, 4]$	$[\,]$
$[2, 3, 4]$	$[1]$
$[3, 4]$	$[2, 1]$
$[4]$	$[3, 2, 1]$
$[\,]$	$[4, 3, 2, 1]$

A *loop invariant*[1] is a property of the loop control variables (and possibly some other variables that do not change during the loop) that is true each time the loop reaches its top, and a well-chosen loop invariant can help a programmer understand what a loop is accomplishing. It expresses an unchanging aspect of a process that is constantly changing variables. In terms of the table for the reversal loop, a loop invariant is an assertion about y and r that is true for every line of the table. A useful loop invariant is that $\text{reverse}(x) = \text{reverse}(y) +\!\!+ r$. That is, the reversal of y followed by r yields the reversal of the original list x, at every line of the table. The line where $y = [3, 4]$ and $r = [2, 1]$ satisfies the loop invariant because reverse([1,2,3,4]) = reverse([3,4]) ++ [2,1].

When converting a loop to a recursive definition, there is a concept related to a loop invariant called an *invariant function.* If the control variables are y and r, then an invariant function is a function $\text{inv}(y, r)$ that produces the same value at every line of the table. For reverse, you should discover that $\text{inv}([1, 2, 3, 4], [\,]) = \text{inv}([2, 3, 4], [1]) = \text{inv}([3, 4], [1, 2])$, etc. A function that has the desired property is *shunt*, defined by $\text{shunt}(y, r) = \text{reverse}(y) +\!\!+ r$. At every line in the list reversal loop table, $\text{shunt}(y, r)$ yields the same answer, $\text{reverse}(x)$.

To convert a loop to a recursive function definition, write a definition of an invariant function for the loop. The loop offers advice on how to define its invariant function. Suppose that the loop table has two consecutive lines,

[1] Loop invariants are discussed in more detail in Chapter 47.

$$\begin{aligned}
&\text{reverse}([2,5,3,6])\\
&\quad = \text{shunt}([2,5,3,6],[\,])\\
&\quad = \text{shunt}([5,3,6],[2])\\
&\quad = \text{shunt}([3,6],[5,2])\\
&\quad = \text{shunt}([6],[3,5,2])\\
&\quad = \text{shunt}([\,],[6,3,5,2])\\
&\quad = [6,3,5,2]
\end{aligned}$$

Figure 19.1: Evaluation of a list reversal using a more efficient definition of reverse, Equations (19.3.2), (19.3.3) and (19.3.4).

y	r
y'	r'

Then, clearly, the invariant function definition should be written so that $\text{inv}(y, r) = \text{inv}(y', r')$. For example, $\text{shunt}([1, 2, 3, 4], [\,]) = \text{shunt}([2, 3, 4], [1]) = \text{shunt}([3, 4], [2, 1]) = \text{shunt}([4], [3, 2, 1]) = \text{shunt}([\,], [4, 3, 2, 1])$. In general,

$$\text{shunt}(x :: xs, r) = \text{shunt}(xs, x :: r). \tag{19.3.2}$$

Another equation handles the case of an empty list.

$$\text{shunt}([\,], r) = r \tag{19.3.3}$$

That equation corresponds to the step at the end of the loop that computes the answer from the control variables. (Remember that we made answer = r in the loop.) You should be able to check not only that these equation describe what is happening in the loop, but also that they are correct according to the definition $\text{shunt}(y, r) = \text{reverse}(y) +\!+\, r$.

Loops usually also have some initialization. For example, the reverse loop initializes $y = x$ and $r = [\,]$. That is taken into account by adding another function that calls shunt with specific parameters. The reverse function is defined as follows, in terms of shunt.

$$\text{reverse}(x) = \text{shunt}(x, [\,]). \tag{19.3.4}$$

Evaluation of reverse([2,5,3,6]) by this new definition of reverse is shown in Figure 19.1. If you look carefully, you can see the playing card method being performed. The first of the two lists is what is in your hand, and the second is what is on the table. The algorithm is played out in the evaluation, and so is implicit in the equational definition. The definition explicitly states facts about shunt, and inside-out evaluation turns those facts into an algorithm.

The number of steps to reverse a list of length n by this method is $n + 2$, which is certainly much better than the $n(n - 1)/2$ steps required by naive reverse. Another thing to notice is that, in the new reversal algorithm, the size of the expression being evaluated stays the same throughout the computation. The reason is that the definition of shunt is tail recursive (see Chapter 13), and that can lead to a substantial saving in memory.

What is shunt really doing for us?

Naive reverse computes all prefixes of the desired answer. For example, using Equations 19.3.1, you find that computing $\text{reverse}([a, b, c, d])$ computes each of $[d]$, $[d, c]$, $[d, c, b]$ and

$[d, b, c, a]$. If you look at each of those as a linked list diagram, you see that each one is obtained from the previous one by replacing a null pointer by a nonempty list. What makes naive reverse so slow is that it is producing the wrong lists, and then patching them. A more efficient algorithm would produce the correct list the first time. The source of the problem is the equation that introduces the null pointers, namely

$$\text{reverse}([\,]) \quad = \quad [\,].$$

Instead, it would be better to tell the function what you want at the end, after the reversal. That is, pass it another list. Calling the modified function shunt, you get

$$\text{shunt}([\,], r) \quad = \quad r.$$

That is, when the reversal is done, put r in the linked list, not $[\,]$. That suggests to you that shunt(x, y) should compute reverse(x) ++ y. For a nonempty list, you want

$$\begin{array}{rcl} \text{shunt}(x :: xs, ys) & = & \text{reverse}(x :: xs){+}{+}ys \\ & = & (\text{reverse}(xs){+}{+}[x]){+}{+}ys \\ & = & \text{reverse}(xs){+}{+}([x]{+}{+}ys) \\ & = & \text{shunt}(xs, [x]{+}{+}ys) \\ & = & \text{shunt}(xs, x :: ys) \end{array}$$

using the facts that ++ is associative and that $x{::}ys = [x]$ ++ ys. We arrive at the same equations by a different thought process.

19.4. Summary of terminology and concepts

Equational programming can only be used to define functions that do not change anything, so it is used for persistent data structures.

Loops can be simulated using tail recursion, and understanding an invariant property of a loop helps you to choose the function to define.

Terminology

Invariant function An invariant function of a loop is a function of the control variables of the loop, and possibly of other variables that do not change in the loop, that has the same value each time the loop reaches its start.

Persistent data structure A data structure is persistent if none of the allowed operations on it can change anything. They only compute new data structures from old ones without altering the old ones.

19.5. Exercises

19.1. Suppose that function sum is supposed to add the numbers in a list. For example, (sum $[2, 4, 6]$) $= 2 + 4 + 6 = 12$.

(a) Using pseudo-code, write a loop that sums the members of a loop by adding the head of a list to a variable, and taking the tail of the loop, each time around the loop.

(b) Show a table of the values of the loop control variables each time control reaches the top of the loop, for the example (sum $[2, 4, 6]$).

(c) Find a useful invariant function of your loop. (If your loop control variables are s (a number) and y (a list), and the orignal list is called x, how are s, sum(y) and sum(x) related?)

(d) Write an equational definition of the invariant function.

(e) Write a definition of sum in terms of the invariant function.

(f) Your definition of sum is more involved that the one that you would have produced for Exercise 18.8. Does this new definition have any advantage over the shorter, more direct, one?

19.2. Chapter 21 defines function decimal, where decimal(”247”) = 247, using a fold. For this exercise, you will define decimal without a fold.

(a) Using pseudo-code, write a loop that computes decimal(s) for a string s of decimal digits. (A string is just a list of characters.)

(b) Show the values of the loop control variables each time the loop reaches its top when computing decimal(”2468”).

(c) Derive an invariant function from the table in the example. What useful thing is true at each line? (Hint: your invariant function might involve taking the digits of a given number and following them by the digits in a given string. For example, the digits of 24 followed by the digits in string ”68” would be number 2468.)

(d) Convert the invariant function to a function definition.

(e) Define decimal in terms of your invariant function. If you cannot, then you might have made an inappropriate choice of invariant function.

19.3. Suppose that you already have a function called *get* that gets a decimal integer from a string. It produces an ordered pair holding the integer and the rest of the string, after the integer. For example, get(”123 456 78”) = (123,” 456 78”), where string ”123” has been converted into the integer 123. Before getting digits, get skips over leading blanks. Using get, write an equational definition of a function getAll that takes a string of decimal digits and blanks, and converts it into a list of all of the blank-separated numbers in the string. For example, getAll(”123 456 78”) = $[123, 456, 78]$. Assume that there are no spaces after the last digit.

19.4. A binary search tree is either an empty tree or is a node with a label and two subtrees, called the left subtree and the right subtree. If a node v has label x, then all labels in the left subtree of v are smaller than x, and all labels in the right subtree of v are greater than x.

Suppose that you have a binary search tree, and you want to produce a list of all of the labels in the tree in ascending (left-to-right) order. You have the following operations available: empty(t) is true if t is an empty tree; left(t), right(t) and label(t) are the left subtree, right subtree and label, respectively, of a nonempty tree t.

(a) Write an implementation of members(t), which produces a list of the members of tree t in ascending order. You can use concatenation (++).

(b) Write an implementation of members(t) that does not use concatenation. Instead, use a helper function membersHelp(t, r) that produces a list of all members of tree t, in ascending order, followed by list r. Define members in terms of membersHelp.

(c) What is the advantage of using membersHelp and avoiding concatenation?

Chapter 20

Equational Programming in Cinnameg

20.1. Defining functions in Cinnameg

Although you can translate a simple equational definition into just about any programming language, it is best to have a language that supports equations directly. In this chapter, we look at how you can write equational function definitions in Cinnameg with very little modification from their mathematical forms, making equations a viable and practical way of defining functions. That gives you an opportunity to try out your ideas with minimal modification to fit them into what a language requires. Also, later chapters, which introduce more advanced ideas, will rely on Cinnameg to make those ideas concrete.

To apply a function in Cinnameg, just write the function followed by its argument. Parentheses are not required. For example, expression sqrt 4.0 evaluates to 2.0. Often, we write parentheses for clarity. (Parentheses are required around tuples, and where needed to override precedence rules. Function application has higher precedence than binary operators, so $f(x + y)$ is not the same as $f x + y$.)

To express a definition with one equation in Cinnameg, introduce the equation with the word Define, and end it with %Define, as in the following definition of function sqr, where $\text{sqr}(x) = x^2$.

```
Define sqr(x) = x * x %Define
```

The syntax of Cinnameg requires a construct that begins with a capitalized word, such as Define, to have an end marker, %Define. But you can use a period to stand for an arbitrary end marker, as long as the period is no more than one line after the beginning of the construct that it ends. So you can write

```
Define sqr(x) = x * x.
```

The max function can be written with one equation or two. When you use more than one equation, introduce each equation with the word **case**. Figure 20.1 shows both approaches. Cinnameg supports lists using the same notation that we have been using. Figure 20.2 shows a definition of cat.

```
Define max(x,y) =
  If x >= y
    then x
    else y
  %If
%Define
```

(a)

```
Define
  case max(x,y) = x when x >= y
  case max(x,y) = y when y >= x
%Define
```

(b)

Figure 20.1: **Two definitions of the max function written in Cinnameg.** The conditional expression cond(a, b, c) is written `If` a `then` b `else` c `%If` in Cinnameg. To use more than one equation, write **case** in front of each equation. Notice that there is no period at the end of a case. A period stands for an end marker, and only words that begin with upper case letters have end markers.

```
Define
  case cat( [], ys)    = ys
  case cat(x::xs, ys) = x :: cat(xs, ys)
%Define
```

Figure 20.2: **A definition of the cat function in Cinnameg.** Compare to Equations 18.8.6. You do not really need this function since concatenation is already available in Cinnameg as operator ++. For example, $[2,4,6]$ ++ $[8,10] = [2,4,6,8,10]$.

20.2. Writing examples

Cinnameg allows you to write examples with a function definition. Examples serve three purposes. They augment the documentation, so that a human reader can understand what the function does. They let the compiler know something about the types that you intend a function's parameter and result to have. And they help to check that a function definition is at least partially working; examples are run when a program starts, and any that fail to work are reported. The following defines sqr with two examples by grouping the examples and the definition together in a *team*.

```
Team
  Example sqr(3) = 9.
  Example sqr(5) = 25.
  Define sqr(x) = x * x.
%Team
```

20.3. Search order

When you define a function by more than one equation in Cinnameg, the normal evaluation policy is to try the equations in the order in which they are written until one that succeeds is found. Only the first one that works will be used, so you can presume that each equation will only be tried when all of the preceding equations have failed. To emphasize that to a reader, write the word **first** just before the first case. The following definition of max illustrates.

```
Define first
  case max(x,y) = x when x >= y
  case max(x,y) = y
%Define
```

Of course, the second equation is only true when $x \le y$, but its proviso can be omitted since it will only be tried in the case where $x < y$. That makes a more efficient definition, since the second proviso does not need to be checked. But it has an unpleasant characteristic as well. Since the second equation relies on an *implicit proviso* that the preceding case does not apply, you can only check it by first understanding the prior case. If there are several equations, checking each one can be difficult. Sometimes, you prefer to write a definition so that each case can be checked by itself, independently of the other cases. To do that in Cinnameg, write the word **one** just before the first case, indicating that the cases can be tried in an arbitrary order, not necessarily in the order written. Each case must stand on its own. The definition of function max can be written as follows.

```
Define one
  case max(x,y) = x when x >= y
  case max(x,y) = y when y >= x
%Define
```

20.4. Giving things names

Sometimes you want to give names to intermediate results in computations either because you intend to use those values more than once (and only want to compute them once) or because you find that the names make the program more readable. To illustrate, function pow(x, n) = x^n can be defined, for a positive integer power n, by three equations.

$$\begin{aligned}\text{pow}(x,1) &= x\\ \text{pow}(x,2*n) &= \text{pow}(x,n)*\text{pow}(x,n)\\ \text{pow}(x,n+1) &= x\cdot\text{pow}(x,n)\end{aligned}\tag{20.4.1}$$

The second equation uses pattern 2*n, which, since n is an integer, can only match an even integer. Notice that the second equation uses pow(x,n) twice. Since the value will be the same both times, it makes sense to compute it just once, and to give it a name. Figure 20.3 shows a definition of that does that.

20.5. Types

Cinnameg is a typed language, and the compiler performs type checking to ensure that you have not made errors such as passing a list to a function that needs to receive an integer. It is important to have some understanding of types so that you can see what is wrong when you make a type error.

You are not required to write information about types in programs because the compiler infers information from the way the definition is written. For example, if you write

```
Define
  case pow(x, 1)   = x
  case pow(x, 2*n) = p*p |
                       Let p = pow(x, n).
  case pow(x, n+1) = x*pow(x, n)
%Define
```

Figure 20.3: **Definition of pow in Cinnameg illustrating naming.** Expression E | S calls for performing S, in this case defining p, and then producing the value of expression E. Pattern $2*n$ only matches an even number, and solving equation $2*n = k$ makes $n = k/2$.

head(x), then the compiler can infer that x must be a list. But you are allowed to write explicit type information if you want to, and doing that can help you to understand your definitions. This section explains how to add type information to Cinnameg programs.

Chapter 9 explains Cinnameg notation for types. Simple types have names, such as Integer, Boolean and Char, that begin with upper case letters. (Integer, Boolean) is the type of an ordered pair containing an integer and a boolean value, in that order, and [Integer] is the type of a list of integers. All members of a list are required to have the same type. A string is a list of characters, so has type [Char]. (String constant ”abc” is an abbreviation for [’a’, ’b’, ’c’].)

Functions also have types. A function that takes a parameter of type A and produces a result of type B has type $A \to B$. For example, the addition function for integers takes a pair of integers and produces an integer. Its type is (Integer, Integer) → Integer. (In programs, you write → as `->`.)

Adding type information to programs

You can add type information to a name or expression by writing :T after it, where T is its type. For example, a function that squares a real number can be defined by

```
Define sqr(x: Real): Real = x*x.
```

indicating that x has type Real and sqr(x) also has type Real. Alternatively, you can separate the type information from the definition. After Define, write the left-hand side, with type information, then the word **by** followed by the definition. For example,

```
Define sqr(x: Real): Real by
  sqr(x) = x*x
%Define
```

Since a function has a type, you can omit the details of the heading, and directly give the type of the function. In that style, the max function might look as follows, when restricted to work on integers.

```
Define max: (Integer, Integer) -> Integer by first
  case max(x,y) = x when x >= y
  case max(x,y) = y
%Define
```

Polymorphism

The cat function should, ideally, be able to work on any kind of list, as long as the two lists being concatenated have the same type. For example, the concatenation of two lists of characters is a list of characters, and the concatenation of two lists of integers is a list of integers. Functions that work on more than one type of thing are called *polymorphic* functions.

If you do not provide any type information, the Cinnameg compiler will try to make a function as polymorphic as possible, within the constraints of what the function does. When it reports a type, it will use *type variables* to stand for arbitrary types, where a type variable is a name enclosed in angle brackets (less-than and greater-than signs). For example, after compiling the definition of cat, the compiler will respond that cat has type ([<a>], [<a>]) → [<a>]. That is, cat takes a pair of lists (of the same type) and produces a list (of that same type). You can include type variables in your definitions too, as shown below.

A practical issue having to do with polymorphism concerns examples. Normally, the compiler assumes that an example provides implicit type information, and it takes examples into account when it is looking at how things are used. If it sees

```
Team
  Example cat([1,2,3], [4,5]) = [1,2,3,4,5].
  Define cat: ([<a>], [<a>]) -> [<a>] by
    ...
%Team
```

then the compiler assumes that cat can only work on lists of numbers, because that is how it is used in the example. (The polymorphic type given for cat is taken as an upper bound, not as a requirement for polymorphism; more information might restrict what variable <a> can stand for. Here, it can only stand for a numeric type.) If you use the word Sample instead of Example, the compiler will not use type information in your example as an indication of the type of the function. So you write

```
Team
  Sample cat([1,2,3], [4,5]) = [12,3,4,5].
  Sample cat("ham", "burger") = "hamburger".
  Define cat: ([<a>], [<a>]) -> [<a>] by
    ...
%Team
```

20.6. List notations

Programming languages make some jobs easier by providing convenient notations for common tasks. This section describes some notation for lists that are available in Cinnameg.

Counting lists

Notation $[m, \ldots, n]$ stands for the list $[m, m+1, \ldots, n]$. For example, $[1, \ldots, 5]$ is the list $[1, 2, 3, 4, 5]$. If $y < x$ then $[x, \ldots, y]$ produces $[\,]$.

```
Define
  case quicksort [] = []
  case quicksort (pivot :: rest) =
    quicksort(smaller) ++ [pivot] ++ quicksort(larger) |
      Let smaller = [x | x from rest, x <= pivot].
      Let larger  = [x | x from rest, x > pivot].
  %Define
```

Figure 20.4: **Quicksort using list comprehensions.** Operator ++ concatenates lists. Compare this to an imperative definition of Quicksort, such as the one in Figure 13.6.

List comprehensions

Expression `[x*x | x from [1,...,5]]` stands for list $[1, 4, 9, 16, 25]$, the list of squares of the members if list $[1, \ldots, 5]$. More generally, a *list comprehension* describes a list by giving a source list ([1,. . . ,5] in the example) and an expression telling what is put in the result list (x*x in the example). A list comprehension can also contain a *filter*, a condition indicating which members of the source list to look at. (Only those that make the condition true are considered.) For example, odd?(n) is true if n is odd, and `[x*x | x from [1,...,5], odd?(x)]` evaluates to list $[1, 9, 25]$ and `[n | n from [2, ..., 20], prime?(n)]` produces the list $[2, 3, 5, 7, 11, 13, 17, 19]$ of all prime numbers between 2 and 20.

List comprehensions can make some algorithms easy to express. Suppose that a function sum has already been defined, where sum(x) is the sum of the numbers in list x. Then the sum $\sum_{k=1}^{n} f(k)$, for some function f, can be expressed easily and clearly using a list comprehension and the sum function.

```
sum [f(k) | k from [1, ..., n]]
```

List comprehensions also make it easy to express the Quicksort algorithm for sorting a list. Quicksort says to sort a nonempty list L by using the head of L as a *pivot.* Separate the tail of L into two lists A and B, where A consists of all members of tail(L) that are less than or equal to the pivot and B consists of all members of tail(L) that are greater than the pivot. Then sort lists A and B, using Quicksort, yielding sorted lists A' and B'. Finally, create the result list by concatenating A', the pivot and B'. For example, to sort list $L = [6, 3, 8, 9, 2, 4, 7]$, the pivot is 6, $A = [3, 2, 4]$ and $B = [8, 9, 7]$, $A' = [2, 3, 4]$ and $B' = [7, 8, 9]$. Concatenating A', the pivot and B' yields [2,3,4,6,7,8,9]. Figure 20.4 defines Quicksort using list comprehensions to get the two lists A and B.

20.7. Testing your definitions

To test your function definitions, you will need to write a complete program, including a request to run the functions. Examples and samples will run automatically, but you might want to do more than just run examples. Figure 20.5 shows a complete package that tests the insertion sort function described in Section 19.2.

```
Package tester
  Team
    Sample insert(5,[2,4,6,8]) = [2,4,5,6,8].
    Define
      case insert(x, [])    = [x]
      case insert(x, y::ys) = x::y::ys        when x <= y
      case insert(x, y::ys) = y::insert(x, ys)
    %Define
  %Team
  Team
    Sample sort([5,2,4,7,1]) = [1,2,4,5,7].
    Define
      case sort([])    = []
      case sort(x::xs) = insert(x, sort(xs))
    %Define
  %Team
  Execute
    Displayln(sort([3,21,8,2,9,15])).
    Displayln(sort("word")).
  %Execute
%Package
```

Figure 20.5: **A complete package that defines and tests a sorting function.** The package is named tester. The Execute part steps aside from functional programming. It contains statements to be performed, in this case displaying the results of sorting two given lists.

20.8. Exercises

20.1. Translate each of the following to Cinnameg and test it using a Cinnameg compiler.

(a) Equations 18.4.1 and 18.4.2.

(b) Equations 18.8.3.

(c) Equations 18.8.4.

20.2. The second equation in Equations 20.4.1 for the pow function is not really necessary, since the other equations cover all cases. Does the second equation help at all, or should it be removed? (**Hint.** How many multiplications does pow use to compute 3^{16}?)

20.3. Function sublist is defined so that sublist(x, y) takes a list x and an *increasing* list of positive integers y and produces the list of members of x in the positions indicated by y, where 1 means the first member, 2 the second member, etc. For example, sublist($[a, b, c, d, e, f, g]$, $[2, 3, 5]$) = $[b, c, e]$, since it asks for the second, third and fifth members of list $[a, b, c, d, e, f, g]$. If an index is requested that does not exist or that is out of order, then sublist should ignore that index.

(a) Write a definition of sublist in Cinnameg in an equational style, with type information and samples. You can use operator _, where x_n is the n-th member of list x.

(b) A more efficient approach for linked lists is to define a more general function sublistFrom(x, y, i) that is similar to sublist, but it assumes the first member of list x is numbered i, the next $i + 1$, etc. For example, sublistFrom($[b, c, d]$, $[3, 4]$, 3) = $[b, c]$. Write a definition of sublistFrom, then define sublist in terms of sublistFrom.

20.4. Say that list x is a sublist of list y if x can be obtained by removing zero or more members from list y. The empty list is a sublist of every list, and every list is a sublist of itself. A sublist preserves order. For example, $[1, 3]$ is a sublist of $[1, 2, 3, 4]$, but $[3, 1]$ is not. Write a Cinnameg definition of function isSublist(x, y) that returns true just when x is a sublist of y. Include type information and samples.

20.5. You do not always need to use recursion in a function definition. Without using recursion, write a Cinnameg definition of function rotate, which moves the first member of a list to the end. For example, rotate($[a, b, c, d]$) = $[b, c, d, a]$. Assume that rotate($[\,]$) = $[\,]$. You can use concatenation.

20.9. Bibliographic notes

Quicksort was discovered by Hoare [53] and further explored by Sedgewick [96].

Part VII

Exploring Functional Programming

Chapter 21

Higher-Order Functional Programming

21.1. Functions as values

Until now, we have treated functions as tools to be defined and used. But you can do more with a tool than use it. You can put it in a tool box, lend it to a friend, or make improvements to it. In a program, that means treating a function as a first class value.

There is an important notational issue. You are probably accustomed to showing a parameter when discussing a function. For example, you might draw a graph of function $\sin(x)$. But in our notation, $\sin(x)$ is the result that the sin function produces on parameter x. The function itself is just called sin.

Suppose that you have a function called longfunctionname, and you find it convenient to give this function a new, shorter name, s. So you write

```
Define s(x) = longfunctionname(x).
```

But if you realize that a function is just a value, you can write a slightly shorter definition.

```
Define s = longfunctionname.
```

For this chapter, it is important to begin thinking of functions a values on the same footing as numbers and lists.

21.2. Function expressions

If a function is a value, then it should be possible to write an expression whose value is a function. In Cinnameg, if p is a pattern and e is an expression, then expression (p `|->` e) has a function as its value, namely the function f defined by $f(p) = e$. For example, a function that squares its input can be written (x `|->` $x*x$). Figure 21.1 shows two equivalent definitions of sqr, one using a function expression.

The process of running a function on a given parameter is called *applying* the function, and expression $f\,x$ produces the result of applying function f to parameter x. The function can be given by any expression that yields a function. For example, expression (x `|->` $x * x$)

```
Define sqr = (x |-> x*x).
```

(a)

```
Define sqr(x) = x*x.
```

(b)

Figure 21.1: **Two equivalent definitions of the sqr function.** The Cinnameg compiler treats definition (b) as an abbreviation for definition (a). In general, if A, B and C are expressions, then `Define` A B `=` C `%Define` is an abbreviation for `Define` A `= (`B `|->` C`) %Define`

5 yields the result, 25, of applying function (x `|->` $x * x$) to 5. You can add parentheses freely. Expressions $f\,x$, $f(x)$, $(f\,x)$ and $(f)(x)$ all have the same meaning.

21.3. The mechanics of function application

The *body* of a function expression is the part after `|->`. For example, the body of function (x `|->` $x{*}x$) is expression $x{*}x$. Evaluation of expression ((p `|->` b) v) is done in three steps: (1) solve pattern match equation $p = v$; (2) substitute the solution of that equation into the body b; and (3) evaluate the body with those substitutions. For example, expression ((x `|->` $x * x$) 5) starts by solving the trivial pattern match equation $x = 5$ for x. Then it substitutes 5 for x into the body, yielding 5*5. Finally, evaluation of 5*5 yields the result, 25.

21.4. Function transformers

A *function transformer* takes a function as a parameter and produces another function as its result. It is a tool for creating new functions. As an example, suppose that you have a function p that yields a Boolean answer, and you would like one that produces the opposite answer. A function that produces a Boolean answer is called a predicate and predicates are, by convention, given names that end on a question mark. For example, Cinnameg library predicate odd? takes an integer n and produces answer true if n is odd. You need a predicate that tells whether a number is even. An obvious definition of even? is

```
Define even?(n) = not(odd?(n)).
```

But another approach is to create a function transformer that does this kind of thing on any predicate, and to apply the transformer to the odd? predicate. Here is a definition based on that. Notice that parameter p of notf is a predicate.

```
Define notf(p) = (x |-> not(p(x))).
Define even?  = notf(odd?).
```

The advantage is that notf is available for other predicates. In this simple case, the advantage is slight. But function transformers yield a new way of writing programs. Instead of building each tool (or function) by hand, you can use tool-building tools to write function definitions for you.

21.5. Curried functions

Suppose that function add is defined as follows.

```
Define add = (x |-> (y |-> x+y)).
```

Clearly, add is a function. What do you get if you evaluate add(2)? To find out, follow the rules for applying a function. The body of (x |-> (y |-> $x + y$)) is (y |-> $x + y$), and substituting $x = 2$ into the body yields expression (y |-> $2 + y$). So add(2) is a function, namely the function that returns a number that is 2 larger than its parameter. For example, $(\text{add}(2))(3) = 5$ and $(\text{add}(2))(7) = 9$. To define a function called shove so that shove(x) $= x + 2$, it suffices to write

```
Define shove = add(2).
```

You can use parentheses wherever it makes expressions more readable, so you can write add 2, add(2), (add 2), (add) 2, etc. Juxtaposition (putting things next to one another) associates to the left, so expression add 2 3 is implicitly understood to be the same as ((add 2) 3).

Figure 21.1 shows equivalent function definitions; `Define` A B `=` C `%Define` is an abbreviation for `Define` A `= (`B `|->` C`) %Define`. Using that idea, you can abbreviate the definition of add to

```
Define add x = (y |-> x + y).
```

Now, using the same idea again, you get

```
Define add x y = x + y.
```

Function add appears to have two parameters. But *all* functions in Cinnameg really have just one parameter. The add function takes one parameter and returns a function. Initially, add is waiting for its first parameter, x. When you evaluate add(2), you get a function that is remembering that x is 2, and is waiting to get the value of y. Functions defined in this way are called *curried* functions, after Haskell Curry[1], one of the early researchers in the foundations of functional programming.

There is an analogy between curried functions and multi-dimensional arrays. Some languages, including C++ and Java, treat a two-dimensional array as an array of arrays. If you declare

```
int A[5][5];
```

in C++ to create a 5×5 array, then you can use two indices, such as $A[2][3]$, to get a particular integer from the two-dimensional array, or you can use just one index, such as $A[2]$, to get the entire row at index 2. A is a two-dimensional array of integers, $A[2]$ is a one-dimensional array of integers and $A[2][3]$ is an integer. Analogously, add is a function that takes two numbers and produces a number, (add 2) is a function that takes one number and produces a number, and (add 2 3) is a number.

You can define any function that you think of as taking more than one parameter in a curried form. For example, function take from Chapter 18 gets a length n prefix of a given list x. It naturally has two parameters, n and x. Definition of take as a curried function is shown in Figure 21.2. Now (take 3) is a function that extracts a length 2 prefix of a given list, and (take 2) $[2, 4, 6, 8] = [2, 4]$.

[1] Curry used this style, but claimed that it was invented by others earlier

```
Define one
  case take 0 xs          = []
  case take n []          = []
  case take (n+1) (x::xs) = x :: take n xs
%Define
```

Figure 21.2: **A definition of take as a curried function.** Parentheses around $(n+1)$ and $(x :: xs)$ are important, since juxtaposition has higher precedence than binary operators.

21.6. Types of functions

A function that takes a parameter of type A and produces a result of type B has type $A \to B$. So the polymorphic type of the take function is Integer → ([<a>] → [<a>]). The curried add function, when used on real numbers, has type Real → (Real → Real). By convention, the → operator is right-associative, so you can say that add has type Real → Real → Real. Expression add(2.4) has type Real → Real.

21.7. Some elementary tool-building tools

A higher-order function is one that either takes another function as a parameter or produces another function as its result, or both. Higher order functions typically do not implement just a single algorithm, but an entire class of algorithms. They make it possible to write the details of the entire class just once and reuse the implementation many times.

Mapping a function onto a list

Higher order function *map* is defined so that (map f [a,b,c]) = [$f(a)$, $f(b)$, $f(c)$]. In general, map applies f to each member of a list separately. A simple definition, using a list comprehension, is

```
Define map f x = [f(y) | y from x].
```

For example, map sqr $[2, 3, 4, 5] = [4, 9, 16, 25]$ and map (+) [(2,3), (4,5)] = $[5, 9]$ and map (take 2) $[[a, b, c, d], [e, f, g]] = [[a, b], [e, f]]$. Figure 21.3 shows how map works in a graphical form. Map captures all algorithms that work by modifying each member of a list independently of the other members. Examples of functions that are maps are (1) converting all letters in a string to upper case, (2) replacing each file in a list by the length of the file, and (3) decoding each string in a list of strings.

Checking whether any member of a list satisfies a predicate

Another useful higher order function is *someSatisfy*, where (someSatisfy f $[a, b, c]$) is true if at least one of $f(a)$, $f(b)$ or $f(c)$ is true. For example, (someSatisfy (x |-> x $>$ 100) $[20, 300, 5]$) is true, but (someSatisfy (x |-> x $>$ 100) $[1, 2, 3, 4]$) is false. Function someSatisfy captures all algorithms that search a list looking for a particular kind of value. For example, asking whether a value is a member of a list is a search problem. The following defines a

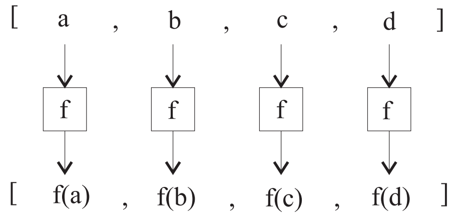


Figure 21.3: **Diagram of a map.** The diagram shows computation of map f $[a, b, c, d]$.

curried member predicate, where (member k x) is true if k is a member of list x, using someSatisfy.

```
Define member k = someSatisfy (y |-> y == k).
```

You might find a slightly longer definition easier to understand.

```
Define member k x = someSatisfy (y |-> y == k) x.
```

Definition of someSatisfy is straightforward.

```
Define
  case someSatisfy f []      = false
  case someSatisfy f (x::xs) = f(x) or someSatisfy f xs
%Define
```

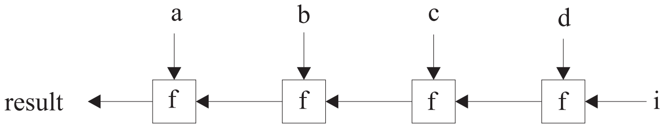
21.8. Folding, or scanning, a list

Some algorithms on lists work by scanning once over the list, either from left to right or from right to left, and producing an answer at the end of the scan. While scanning the list, the algorithm remembers some information, such as the largest value seen, or the sum of the values seen. Scan algorithms are called *folds* of lists.

Two higher order functions will capture the two classes of scan algorithms; foldRtoL scans from right to left and foldLtoR scans from left to right. Each function needs three parameters: an initial value, representing the initial information that the algorithm remembers; a function that updates the information as new list members are encountered; and a list to scan. There are arbitrary decisions to be made in defining foldRtoL and foldLtoR: in what order, and in what way, should the parameters be given to these functions? We will use a definition of foldRtoL where foldRtoL i f [a,b,c] $= f(a, f(b, f(c, i)))$. The initial value is i and the update function is f. It is easier to see what foldRtoL does when the function f is replaced by a binary operator, such as $+$. Then foldRtoL i (+) [a,b,c] $= a + b + c + i$, and foldRtoL i (*) [a,b,c,d] $= a * b * c * d * i$, where in both cases the operations are performed from right to left. So, for example, foldRtoL 0 (+) [2,3,4,5] = 2+3+4+5+0 = 14. A function to sum the members of a list is as follows.

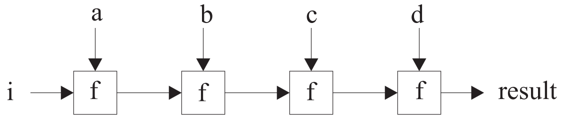
```
Define sum = foldRtoL 0 (+).
```

Figure 21.4 shows a definition of foldRtoL and a diagram of its action. Function foldLtoR is similar to foldRtoL, but it scans the list from left to right. Figure 21.5 shows a diagram of the action of foldLtoR i f $[a, b, c, d]$, along with the definition.



```
Define one
  case foldRtoL i ? []      = i
  case foldRtoL i f (x::xs) = f(x, foldRtoL i f xs)
%Define
```

Figure 21.4: **Definition of foldRtoL and a diagram showing computation of foldRtoL** i f **[**a**,**b**,**c**,**d**].** Pattern ? matches anything, and is a don't-care value. The computations done in the diagram are $f(d, i)$, $f(c, f(d, i))$, $f(b, f(c, f(d, i)))$ and the final result is $f(a, f(b, f(c, f(d, i))))$.



```
Define one
  case foldLtoR i ? []      = i
  case foldLtoR i f (x::xs) = foldLtoR (f(i,x)) f xs
%Define
```

Figure 21.5: **Computation and definition of a left-to-right fold.** The diagram shows computation of foldLtoR i f [a,b,c,d] by computing $f(i, a)$ then $f(f(i, a), b)$ then $f(f(f(i, a), b), c)$ and finally the result $f(f(f(f(i, a), b), c), d)$. Notice that the definition of foldLtoR is tail-recursive.

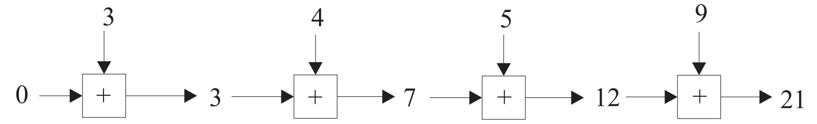


Figure 21.6: **Summing the numbers in a list.** Notice that the partial result, as the algorithm scans the list, is the sum of the numbers seen so far. It is a typical characteristic of scan algorithms that the partial results are just what the final results would be if the scan had reached the end of the list.

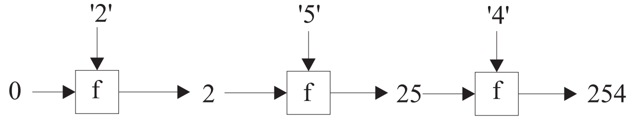


Figure 21.7: **Converting a string of decimal digits to an integer using a left-to-right fold.**

Understanding folds

There is a general rule for defining functions using folds. As you scan a list, you remember some information about the values that you have seen so far. At each stage of the fold, the value to remember is the answer that would be produced if the part of the list looked as so far were the entire list. For example, you know that sum[3,4,5] = 12, so the sum algorithm will remember 12 after reading prefix [3,4,5] of a longer list. Figure 21.6 shows the action of sum as a left-to-right fold.

As another example, suppose that you would like to convert a string of decimal digits to a number. Say, decimal(”254”) = 254. A scan, from left to right, should do the job. After reading no digits, the result should be 0. After reading ”2”, the result should be 2, and after ”25”, number 25 should be produced, as shown in Figure 21.7. The only question is how to write function f. Examining the diagram, you can see that $f(0, \text{‘2’}) = 2$, $f(2, \text{‘5’}) = 25$ and $f(25, \text{‘4’}) = 254$. The algorithm needs to convert a character such as ‘2’ to its associated number, 2. Cinnameg function digitValue will do that, and $f(x, c) = 10 * x + \text{digitValue}(c)$ will work in the fold. Function decimal can be defined as follows.

```
Define decimal = foldLtoR 0 ((x,c) |-> 10*x + digitValue(c)).
```

Often, definitions that make use of higher order functions need helper functions, as does the definition of decimal, and it can be convenient to give those helper functions names. Figure 21.8 shows a definition of decimal with f defined inside it.

21.9. Write-only programs?

Powerful operations such a map and folds take quite a bit of thought to use. If you use them extensively, and especially if you use several of them in one line, you can get a definition that is nearly impossible to read. For example,

```
Team
  Example decimal "2531" = 2531.

  Define decimal = foldLtoR 0 f |
    Define f(x,c) = 10*x + digitValue(c).
  %Define
%Team
```

Figure 21.8: **Definition of decimal with embedded function definitions.** Expression $e|d$ produces the value of e after performing definitions d.

```
Define isPrime n =
  n > 1 and notf someSatisfy (k |-> n ‘mod‘ k == 0) [2,...,n-1]
%Define
```

defines a function isPrime that tells whether an integer is prime. But that is certainly not obvious. Because of this tendency toward unreadability, programs written in this style have been called write-only programs. To avoid the problem, use only one, or occasionally two, powerful operations per line. Carefully document what you have done at each line. Show, with examples, what is happening. Choose descriptive names. Then definitions can be made simple and readable.

```
Team
  %% (factorOf n) k is true if k is a factor of n.
  Example factorOf 10 5 = true.
  Define factorOf n k = (n ‘mod‘ k == 0).

  %% isPrime(n) is true if n is prime.
  Example isPrime(13) = true.
  Example isPrime(9) = false.
  Define isPrime n =
    n > 1 and not(someSatisfy (factorOf n) [2,...,n-1])
  %Define
%Team
```

Computer programs should be written with careful thought to each line, not by typing as fast as you can. Higher order functions tend to concentrate more thought into each line.

21.10. Summary of terminology and concepts

Higher order functions can capture entire classes of algorithms, enhancing reusability of components. Examples of higher order functions are maps and folds.

Terminology

Curried function A function that takes one parameter at a time, by returning a function after it gets each parameter except possibly the last, is called a curried function.

Fold A fold performs a single scan over a list, either from left to right or from right to left. As it scans, it remembers something about the members of the list that it has seen so far.

Higher-order function A function that takes functions as parameters or produces a function as its result is called a higher-order function.

Map A map is an operation on a list that operates on each member of the list independently, and produces a list of the modified values.

21.11. Exercises

21.1. Determine the value of each of the following Cinnameg expressions.

(a) map (x |-> x + 3) [3, 7, 5]

(b) map (add 1) [2, 4, 6, 8]

(c) map (map (x |-> x + 1)) [[2,3,4], [ ], [5, 6]]

(d) map tail [[2, 5], [1, 3, 5, 7]]

(e) someSatisfy (x |-> x) [false, true, false]

(f) map (someSatisfy (y |-> y > 10)) [[2, 3, 4], [ ], [10, 20, 30]]

(g) foldRtoL [ ] (::) [6, 3, 1]

(h) foldRtoL [ ] (++) ["choc", "o", "late"]

(i) foldLtoR 1 (+) [2, 3, 4]

21.2. What is the type of each of the following expressions? If the expression is polymorphic, give the polymorphic type, with type variables. For this exercise, assume that all numbers have type Integer.

(a) map (x |-> x + 1) [2, 4, 6]

(b) map (x |-> x + 1)

(c) map tail [[2,5], [1, 3, 5, 7]]

(d) map tail

(e) foldLtoR 1 (+) [2, 3, 4]

(f) foldLtoR 1 (+)

(g) foldLtoR true (and)

(h) foldLtoR

21.3. Given definition

Define f x y = (x+1)::y.

(a) What kind of thing is x (a number, a list or a function)?

(b) What kind of thing is y (a number, a list or a function)?

(c) What kind of thing is $f\,x\,y$ (a number, a list or a function)?

(d) What kind of thing is $f\,x$ (a number, a list or a function)?

21.4. Given definition

Define f x y z = y x (x::z).

(a) What kind of thing is parameter z? (A number, a list, a function, or something else)?

(b) What kind of thing is parameter y? (A number, a list, a function, or something else)?

(c) What kind of thing does expression $f(2)$ yield? (A number, a list, a function, or something else? Is this expression allowed?)

21.5. Cinnameg library function select is written so that (select p x) returns the first member m of list x such that $p(m)$ is true. Notice that p is a predicate. Predicate odd? returns true on an odd number and false on an even number. What is the value of Cinnameg expression map(select odd?)[$[2, 3, 4, 5]$, $[1, 2, 3, 4]$, $[2, 4, 6, 7, 8]$]?

21.6. Function select of the preceding exercise has the difficulty that it might not have anything to select. For example, it is not clear what (select (x `|->` x < 0) [2,3]) should yield. Suppose that (selectD d p L) yields the same result as (select p L) as long as some member of list L makes predicate p produce true, and yields d otherwise. Write a definition of selectD.

21.7. Show how to write a definition of function doubleAll(L), which takes a list L of numbers and produces a list of the doubles of those numbers. For example, doubleAll($[5, 2, 19, 3]$) = $[10, 4, 38, 6]$. For this definition, do not use any direct recursion. Use map to do the work.

21.8. Show how to write the map function without using a list comprehension, but using recursion.

21.9. Cinnameg library function filter selects all members of a list on which a given predicate yields true. For example, filter (x `|->` x > 20) $[30, 2, 7, 85, 6, 23]$ = [30, 85, 23].

(a) Write a definition of filter using a list comprehension.

(b) Write a definition of filter without using a list comprehension, using recursion instead.

21.10. Show how to implement each of the following as a fold, either from left to right or from right to left (your choice). Do not use recursion.

(a) The factorial function.

(b) The list concatenation function, cat. (Does it matter whether you do a right-to-left fold or a left-to-right fold?)

(c) The map function.

(d) The someSatisfy function.

(e) The list reversal function.

(f) The sort function of Figure 20.5. (Just define sort as a fold, not insert.)

(g) The function maxlist m L that produces the largest member of list $m :: L$.

21.11. The inner product ip(x, y) of two lists of numbers x and y is defined so that ip($[a, b, c]$, $[d, e, f]$) $= a * d + b * e + c * f$.

(a) Write a definition of function sum, defined in Exercise 18.8, using a fold function.

(b) Write a definition of function zip, defined in Exercise 18.18, using recursion.

(c) Write a definition of function products, where products($[a, b, c]$, $[d, e, f]$) $=$ $[a * d, b * e, c * f]$. In general, products takes a pair of lists and makes a list of the componentwise products. Use zip and map to do this.

(d) Write a definition of ip using sum and products. It should be clear from the definition that ip is a sum of products.

21.12. In a purely functional language, is it ever possible to compute the same expression twice in the same context and get different values? For example, if E is an expression, and you compute E two times, one right after another, could the first computation yield a different result from the second computation? Why or why not?

21.13. Answer the preceding question for an imperative language such as Java or C++. Explain your answer.

21.14. The composition $f \circ g$ of two functions f and g is defined by $(f \circ g)(x) = f(g(x))$. Suppose that you would like to compute the composition of a list of functions. For example, composeAll $[f, g, h] = f \circ g \circ h$. Show how to do this without using recursion using a fold. Does it matter whether the fold is from left to right or from right to left? What function should you use as the initial value of the fold? That function determines what composeAll $[\,]$ is.

21.15. Sorting can be done with respect to any comparison operation. If $\ll$ is a given comparison relation, the goal is to rearrange a list into an order $[a_1, \ldots, a_n]$ where $a_1 \ll a_2 \ll \cdots \ll a_n$. For example, if you sort using comparison operation $\leq$, then you sort into ascending (or nondescending) order. If you sort with respect to comparison $\geq$, you sort into descending (or nonascending) order. Modify the definition of quicksort shown in Figure 20.4 so that, in addition to the list, it takes a comparison predicate as a parameter, and sorts with respect to that comparison predicate. The comparison predicate takes an ordered pair and returns a boolean result. (Assume that the comparison predicate returns true on equal values.) Make your function curried, with the comparison function first, so that it is easy to define functions that sort in specific orders.

21.16. Two useful concepts that you have learned about mathematics are *substitution* and *specialization*. If you know that $z = y + u$, the substitution rule says that you can replace z by $y + u$, and that you can also replace $y + u$ by z. If you know that $x + 1 = g(x)$ for every x, then the specialization rule says that you can replace x by anything in that equation and get another true equation. For example, you know that $5 + 1 = g(5)$.

Suppose that you are given the following equations, the first true for all values of x and y.

$$\begin{aligned} \text{add } x\ y &= x + y \\ \text{inc} &= \text{add } 1 \end{aligned}$$

Using only substitution and specialization, show that inc 3 = 1 + 3. Be careful to remember that function application (juxtaposition) associates to the left, and that it is not associative. That is, $(a\,b)\,c \neq a\,(b\,c)$.

21.17. A directed graph has vertices numbered 1, ..., n. The graph is represented by a list $[L_1, \ldots, L_n]$ where L_i is a list of all vertices v such that there is a directed edge from vertex i to vertex v. For example, list [[2], [1,3], [2,3]] describes a directed graph with vertices 1, 2 and 3 and with directed edges (1,2), (2,1), (2,3), (3,2) and (3,3).

Write a definition of function accessible(G, v) that takes a graph G and a vertex number v, and produces a list of all vertices u such that there is directed path from v to u.

21.18. A group of people needs to walk across a rickety bridge. The people walk at different speeds, and you are given a list of the number of minutes each person takes to cross the bridge. For example, list [2,3,6,10] indicates that one person can cross in two minutes, the next in three minutes, etc.

Because the bridge is rickety, only two people can be on the bridge at once. To make matters worse, it is night, and anyone crossing the bridge needs to carry a flashlight. But there is only one flashlight. So two people will cross the bridge with the flashlight, then one will carry the flashlight back, then two people will cross, and so on, until everyone is across. When two people cross together, they go at the speed of the slower person.

You would like to determine the least number of minutes that it takes for all of the people to cross the bridge. If the list is [1,100,100,100], it takes 302 minutes for everyone to cross the bridge. If the list of times is [1,2,5,10], then all can get across in 17 minutes. Do not proceed until you see how each of those times is achieved. This problem is much more subtle that it looks initially, and a naive algorithm will not work.

(a) Write a function (minOver f L) in Cinnameg so that minOver f [a,b,c,d] = min[$f(a)$, $f(b)$, $f(c)$, $f(d)$]. But make minOver work for any nonempty list, not just for lists of length four. Cinnameg provides a standard function, min, that produces the smallest member of a given nonempty list. For example, min[2,4,1,5] = 1.

(b) Suppose that the people start on the left side and need to cross to the right side. Write a function that takes a pair (L_1, L_2), where L_1 is a list of the crossing times of the people on the left side, and L_2 is a list of the crossing times of the people on the right side, and returns the total amount of time necessary for everyone represented in list L_1 to cross the bridge. First, work out the times for lists L_1 of length no more than two. Then, for longer lists, try all possible pairs of people to walk across the bridge together, and select the pair that leads to the least total time. Two calls to minOver (one to select the first person, the other to select the second person), plus a recursive call, will do the job. (To select a person to walk back, it can be shown that you can always select the fastest walker who is on the right side of the bridge. But be careful to select the fastest one after two people have crossed from left to right.)

Arrange only to pass your function lists that are sorted into nondescending order. You will find standard operator $-/$ useful ($L - /y$ is list L with the first occurrence of y removed), and you will also find the insert function from Figure 20.5 useful. Test your function.

(c) You will find that your function from part (b) is *very* slow on lists of size about 7 or 8. Try it to see. The problem is that it computes the same result over and over. Read about the *memoize* function in the Cinnameg documentation, and use it to improve your function. (This is why you should keep the lists of times sorted. You do not want to recompute the cost of list [2,7,4,1] when you already have a result for [1,2,4,7].) Test your modified function. Does it run faster?

21.12. Bibliographic notes

Darlington Henderson and Turner [36] describe use of higher order functions, as do the references from the preceding chapter.

The analogy between curried functions and arrays is from Paulson [86].

Chapter 22

Lazy Evaluation

22.1. Evaluation can be done in different orders

We have seen, in Chapter 18, that expressions can be evaluated in different orders, and that an evaluation policy chooses from among the possible orders. Inside-out evaluation, also called *eager* or *strict* evaluation, performs evaluations of all arguments of a function before evaluating the function call. For example, to evaluate $f(g(x))$ by eager evaluation, you first compute $v = g(x)$, then compute $f(v)$. Inside-out evaluation also goes by the name *call-by-value*, since the argument in a substitution is always a precomputed value.

Eager evaluation is a natural way to do things, but it is not always the best idea. For example, the conditional function, $\text{cond}(a, b, c)$, naturally wants to have a evaluated first, followed by evaluation of either b or c, but not both. Boolean operations have similar improvements. For example, (x and y) is equivalent to $\text{cond}(x, y, false)$, suggesting that y should only be evaluated when x is true. These special functions seem to prefer outside-in evaluation, where arguments are kept unevaluated until they are needed.

It is not hard to think of other functions that only need some of their arguments some of the time, and might benefit from outside-in evaluation. But to employ outside-in evaluation *always* also has problems. If function sqr is defined by $\text{sqr}(x) = x{*}x$, then computation of $\text{sqr}(g(4))$ by outside-in evaluation leads to expression $g(4) * g(4)$, and you find yourself evaluating the same expression $g(4)$ twice.

22.2. Lazy evaluation

Inside-out evaluation has the advantage that it avoids repeated evaluation of the same expression. Outside-in evaluation, on the other hand, has the advantage that it avoids doing even one evalation of expressions whose values are not really needed. You might ask whether those advantages can be combined into a single policy that has the best characteristics of both. It turns out that there is a policy that *almost* does that. It goes by several names, including *lazy evaluation*, *nonstrict evaluation*, and *call-by-need.* The idea is that an expression should not be evaluated until its value is needed. An unevaluated expression is stored as a function, called a *promise*, that can compute the value of the expression when and if it is needed. For example, expression $\text{head}(x)$ is stored as a parameterless function p where $p(\,) = \text{head}(x)$. The value x is built into function p.

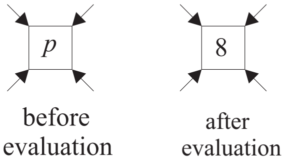


Figure 22.1: **Memoization of a promise.** All copies of a promise point to the same place. If any of the copies need to evaluate the promise, then the result of evaluation is stored where the promise used to be stored, and all copies see the computed value. As a result, a promise only needs to be computed at most once. (Sometimes it is not needed, so is not computed at all.)

To use this idea, the expression evaluator must look at each value, when it is needed, to see whether it is a promise. If so, then the evaluator *forces* the promise; it runs the function stored in the promise to find the expression value. The evaluator then performs a step, called *memoizing*, that is critical to the efficiency of lazy evaluation; it replaces the promise by the value that was computed. If the same value is used again, no evaluation of a promise will be needed — the value will be there to get.

A promise is never copied. Instead, if it needs to be duplicated, two *references*, or *pointers* to the promise are created. That way, if any of the references need to evaluate the promise, all of them will see the answer after it is memoized. The memory holding the promise is shared. Figure 22.1 shows several references to a promise.

You can think of evaluating an expression as a way of transferring information from the point where it is produced to another point in a program where it is consumed. In computer networking, information transfers are classified according to who initiates them. A *push* operation is initiated by the sender. For example, cable television is a push since it is sent out to subscribers whether they are viewing it or not. Push transfers are similar to eager evaluation. A *pull* transfer is initiated by the receiver. An example is a request for the content of a web page. Lazy evaluation is the analog of a pull transfer in evaluation of programs.

22.3. Lazy evaluation in programming languages

With respect to evaluation mode, there are two classes of languages. Nonstrict languages, such as Haskell, use lazy evaluation as the normal mode of evaluation. Strict languages, including most programming languages, instead uses eager evaluation as the normal mode of evaluation. Imperative languages are almost always strict, because, when expressions can have side effects, delaying evaluation can cause the value produced by an expression to depend on when the expression is forced.

Some strict languages provide a way for you to request delayed evaluation. Cinnameg, for example, is strict, but you can request delayed evaluation by enclosing the expression in *smiles*, as (:e:). Evaluating expression (:e:) yields a promise. Definition

```
Define hd(x) = (:head x:).
```

makes hd(x) the same as head(x), but it waits until the head is needed to perform the evaluation. (That might avoid an error, in the case where x is an empty list, if the value is never used in that case.) You can also wrap smiles around the cases in a definition by cases. For example, definition

```
Define (:
  case take 0      ?         = []
  case take ?      []        = []
  case take (n+1) (x :: xs) = x :: take n xs:)
%Define
```

defines a function similar to the take function defined in Chapter 18, but evaluation of (take n x) produces a promise. If you compute head(take 3 [2,4,6,8,10]), that promise will be forced, producing a list whose head is 2 and whose tail (computed by (take 2 [4,6,8,10])) is another promise.

22.4. Some uses for lazy evaluation

Lazy evaluation is a powerful tool, and this section illustrates with a few of its uses. The examples are written in Cinnameg, using smiles to request lazy evaluation.

Infinite lists

Lazy evaluation lets you produce infinitely long lists. Suppose that you would like to create the infinite list [1,2,3,4,...] holding all of the positive integers. You cannot store the entire list in explicit form. But you can store it implicitly, using lazy evaluation, as long as you never try to look at the entire list. To build a list of the positive integers, start by writing a function ints, where ints(n) = [n, $n+1$, $n+2$, ...]. Then define positives to be ints(1).

```
Define ints(n) = (: n :: ints(n+1) :).
Define positives = ints(1).
```

At first, list positives is represented by a promise p_1 that will compute it, if it is ever asked for. Suppose you ask for the head of positives. Forcing promise p_1 yields list 1::p_2 where p_2 is a promise that will compute list [2,3,...], if it is ever asked for. Asking for the second member of list positives replaces p_2 by a little bit more, so that list positives has the form 1::2::p_3, containing yet another promise p_3. List *positives* appears to be changing each time an evaluation of one of its promises is performed. But that is not really what is happening. Chapter 8 showed that there can be many different representations for the same value. A promise is a kind of representation of a value, and all that happens when a promise is forced is that the representation is changed. The value remains the same. The value of positives is constant, the infinite list [1,2,3,...]. (Cinnameg provides a notation [n, ...] for the infinite list [n, $n+1$, ...], so you can define positives = [1,...].

List comprehensions in Cinnameg (Section 20.6) are lazy, and they can also be used to produce infinitely long lists. List comprehension

```
[2*n | n from [2, ...], prime?(n)]
```

produces the infinite list [4,6,10,14,...] of the doubles of all prime numbers.

Explicit lists for implicit searches

Suppose that you want to find the smallest prime number that is greater than or equal to n. In C, you write a loop. (Operator ! is the negation (*not*) operator in C.)

```
k = n;
while(!prime(k)) k = k + 1;
```

What that loop is doing is searching through an *implicitly* defined sequence, given by an initial value (n), and a way to obtain the value that follows any given value (just add 1). The program walks through the sequence without ever writing the whole sequence down at once. An alternative is to search an explicit list. For example, function nextprime defined by

```
Define nextprime(n) = select prime? [n,...].
```

does the job. (Expression (select p xs) produces the first member x of list xs such that $p(x)$ is true.) Only as much of the list as is needed is produced.

Object-oriented languages that provide collections such as sets and tables typically also provide objects called *iterators* that allow you to look at each thing in the collection. A simple iterator has a function init( ), which produces the initial value, a function next(x), which produces the value that follows x in the sequence, and a predicate last(x), which tells whether x is the last value in the sequence. A search for a value in a collection that has property p might look as follows, in general.

```
k = init();
while(!last(k) && !p(k)) k = next(k);
```

From a computational standpoint, the three functions init, next and last are adequate to define the search sequence implicitly. But, from a logical standpoint, you might ask whether this is the most pleasant way to provide a sequence to the programmer. Typically, for every kind of collection object, there is also a kind of iterator, increasing the size of the library. Also, everything that you want to do with an implicit sequence needs to be defined, typically with a loop.

An alternative is to produce an explicit list of all members of the collection, making all list functions in the library available for working with the sequence, without the need to rewrite them for iterators. Lazy evaluation is critical for doing that, since you do not want to produce the entire list if only a small part of it is going to be examined. You can view the presence of iterators in a language as compensation for the language's inability to handle lazy lists.

Memory and pipes

Some operating systems provide a feature called a *pipe* for communicating between processes or threads. A pipe is a channel into which one process writes, and from which another reads. If the pipe is empty, then a process that tries to read from the pipe is suspended until another process writes something into the pipe. If the pipe gets too much information in it, then any process that tries to write into the pipe is suspended until another process consumes some characters from the pipe. That way, it is possible for two processes to exchange a great deal of information using a relatively small memory buffer to hold what is currently in the pipe.

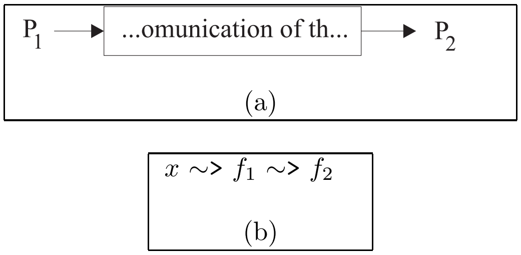


Figure 22.2: **(a) Two processes P_1 and P_2 communicating over a pipe, and (b) two functions f_1 and f_2 communicating by sending the result of f_1 as the argument of f_2.** The pipe only holds information that has been written by P_1 but not yet read by P_2. If f_1 produces a list, and if it is designed to be lazy, then communication between f_1 and f_2 behaves similarly to the pipe, and requires little memory, even if the list is long. The ~> notation is from Cinnameg: x ~> f is the same as $f(x)$, and is done from left to right.

Lazy lists provide a similar mechanism within a program. Suppose function P (the producer) creates a lazy list — one where only as much as is needed is computed. Suppose function C (the consumer) consumes that list, possibly searching for a value, or accumulating some value as it scans the list. As function C runs, more and more of the list is computed by P, and, because the results of lazy evaluation are memoized, what has been computed occupies memory. But eventually the garbage collector runs, the memory occupied by that part of the list that C has already passed over can be reclaimed, and that memory can be used again to produce more of the list. You can scan a long list with a small amount of memory.

22.5. The robustness of substitution

A fundamental principle of mathematics is that two equal expressions are interchangeable. If you have established that $x = y + z$, then you are allowed to substitute $y + z$ for x, or the other way around, in any other expression, as long as you are careful to parenthesize to avoid problems with operator precedence and associativity. Functional programming is designed to follow that principle as well. The only issue should be one of efficiency. But that principle suggests that expression `(Let x = f(1). 0)` must produce result 0, the result of replacing all occurrences of x by $f(1)$ in expression 0. But in Cinnameg, Let statements are eager, and expression `(Let x = f(1). 0)` evaluates $f(1)$ before moving on; if evaluation of $f(1)$ runs forever, the expression will never produce result 0. So, with eager evaluation, performing a substitution can change the result of an expression from an infinite computation to an actual result and the other way around.

Lazy evaluation comes to the rescue. Cinnameg offers another kind of definition, `Define` x `=` e `%Define`, which is lazy; it creates a promise for e, binding x to that promise. Expression `( Define x = f(1). 0)` does have value 0, since $f(1)$ will not be computed at all.

You can see the same thing at work when a conditional expression is used. From a mathematical standpoint, you should be able to give names to common subexpressions that must have the same value. The substitution rule suggests that expression `If r == 0 then f(z) else If r > 0 then 2*f(z) else 0 %If %If` should always produce the same result as `Define y = f(z). If r == 0 then y else If r > 0 then 2*y else 0 %If %If` And it does, but if the (lazy) Define is replaced by an (eager) Let, then the two are not always the same, since an infinite computation of $f(z)$ will be trouble when $r < 0$.

When all computations are as lazy as possible, and expressions with side effects are not allowed, the substitution rule is always correct, and the only effect that substitutions can have are on efficiency, program size and readability.

22.6. Summary of terminology and concepts

An equational program does not always specify all aspects of how evaluation should be done. The language implementation has some flexibility in determining how and when to use equations as substitutions.

Some languages include nonstrict, or lazy, aspects that allow you to define a value now but evaluate it later, if it is ever needed. Some languages do all evaluation in a lazy way.

Lazy evaluation allows you create infinite lists and to traverse long lists with a small amount of memory, since only part of the list needs to be in memory at once.

Terminology

Call-by-need Call-by-need is another term for lazy evaluation.

Call-by-value Call-by-value is another name for inside-out evaluation. The argument of a function is evaluated before calling the function, and the computed value is passed to the function.

Eager evaluation Inside-out evaluation is also called eager evaluation. Each expression that *might* be needed is evaluated, on the presumption that it *will* be needed.

Force Evaluating the function that is held in a promise is called forcing the promise.

Lazy evaluation Evaluation in which each expression is evaluated only the first time its value is needed is called lazy evaluation.

Memoizing When a promise is evaluated its result is memoized (written down), to prevent repeated evaluation.

Nonstrict evaluation Nonstrict evaluation is the same as lazy evaluation.

Promise A promise is a function that is used for lazy evaluation. When a lazy expression is first encountered, a promise is created for it. When the value of this expression is needed, the function is performed to find the value.

Strict evaluation Strict evaluation is another term for eager evaluation.

22.7. Exercises

22.1. Write a Cinnameg definition of function infinitedup(x) that produces the infinite list $[x, x, x, \ldots]$.

22.2. Write a Cinnameg definition of function primes(n) that produces a list of all prime numbers that are greater than n.

22.3. Consider the following definition written in Cinnameg.

Define f(n) = (:n*n :: f(n+1):).

What value does expression f(2) compute? Give the full value, as it would be if it were examined. (It suffices to give a clear description of the full value.)

22.4. Suppose that an iterator for a collection is defined by functions init( ), next(x) and last(x). Show how to examine the sequence backwards, finding the last value y that makes $p(y)$ true. How would you do the same job if given an explicit list?

22.5. Write a lazy list concatenation function. It should be able to produce the concatenation of two lists, but should not spend more time than is necessary computing it. If the list A is infinitely long, then the concatenation of A and B should be the same as A; your concatenation function must not get stuck evaluating A. (The Cinnameg operator ++ is lazy. For this exercise, do not use it.)

22.6. Write a lazy filter function, where filter p L produces a list of all members x of list L such that $p(x)$ is true. (So p is a predicate.) It should be possible to filter an infinitely long list.

22.7. Suppose that L is a list, and that function f takes a list and produces another list. You intend to compute $f(L)$, and you know you will need to use the entire result list. Is there any point in function f being lazy, or would it always make more sense for f to be eager? Think of how the result of $f(L)$ might be used, and the memory requirements for the entire computation.

22.8. Give an example where the substitution rule fails dramatically in an imperative language by showing an expression E that contains two identical subexpressions, but if you derive E' from E by giving a name to the result of the common subexpression and using the name twice, then E' produces a different result from E. (Both E and E' should produce a result, not loop forever.)

22.9. Some compilers, both for functional languages and imperative languages, perform an improvement called common subexpression elimination, where a subexpression that occurs more than once is only computed once for efficiency. Improvements must be sure not to change the meaning of a program. What tests does a common subexpression eliminator for an imperative language need to make? What about for a functional language?

22.8. Bibliographic notes

Haskell [16, 103] and Miranda [104] are examples of languages that use lazy evaluation throughout. Algol 60 [92], an imperative language, uses a calling mechanism referred to as call-by-name for passing parameters to functions. With call-by-name, each time you use a parameter, you re-evaluate the expression that was passed as that parameter, so it is akin to outside-in evaluation, without memoization.

Chapter 23

Scheme: Functional Programming with Lists

23.1. The Lisp family

The Lisp (LISt Processing) family of languages was begun by John McCarthy, with the first implementation of Lisp available in 1960. All of the languages in the Lisp family have a few common characteristics.

1. Lists are the basic data structure. The operations that we have called head, tail and :: are called *car*, *cdr* and *cons*, respectively, in Lisp. (Cdr is pronounced could-er.) For the remainder of this chapter we use terms car and cdr instead of head and tail.

2. The major paradigm of the language is functional, and recursion is the main control mechanism, but other paradigms are supported to some extent. It is possible, for example, to write a loop, and to change a variable.

3. The syntax is extremely simple. Lists are written in parentheses. The members of a list are written after one another, without any additional syntax such as commas.

4. Data and programs are of the same form. A program is a list. As a result, it is easy to treat data as a program, or to treat a program as data.

5. There is no concept of a type when a program is compiled. All type checking is done at run time.

During the 1960s and 1970s, Lisp became popular as a language for symbolic computing. Several different dialects were developed. In an attempt to consolidate the common features of these dialects, and to standardize the language to some extent, Common Lisp was defined. Today, Common Lisp is a popular language for symbolic computing.

The idea of Common Lisp was not to restrict dialects, but to suggest a core language that all dialects of Lisp should support, and possibly extend. But, even though it is intended to be a common core, Common Lisp is still a large language. An alternative Lisp dialect, Scheme, does not contain the Common Lisp core, and instead is designed to be small and elegant. The philosophy of Scheme is expressed well in the first paragraph of the Sheme report: “Programming languages should be designed not by piling feature on top

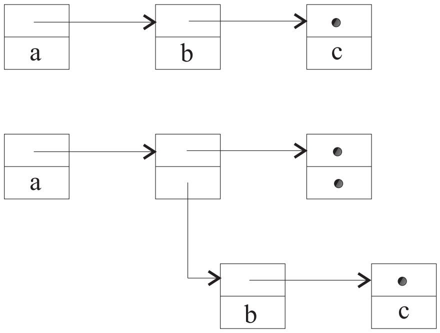


Figure 23.1: **Diagrams of the representation of lists (a b c) and (a (b c) ( )).**

of feature, but by removing the weaknesses and restrictions that make additional features appear necessary."

23.2. The syntax of Scheme and the meaning of lists

An identifier, or name, can be any sequence of letters, digits and certain special characters, provided it does not begin with a digit or a period or the special symbols +, − or #. (The single-character names + and − are exceptions; they are identifiers.) The special symbols that can occur in identifiers are

```
! $ % & * + - . / : < = > ? @ ^ _ ~
```

For example, `this-identifier!` and `*+/` can both be used to name something. There are a few reserved words, such as **if**, **let** and **define**, that cannot be used for names.

The syntax of Scheme is very simple. To write a list in Scheme, enclose the members of the list in parentheses, and separate them by spaces. For example, (a b c) is a list of three things. The empty list is written ( ). A list can contain lists; (a (b c) ( )) is a list of three things, the second and third of which are lists. Scheme syntax is free-form, so a line break is equivalent to a space. An exception to free-formness is a comment, which begins with a semicolon and ends at the end of the line.

Scheme implementations represent lists in linked form, with the empty list represented by a null pointer. The representations of lists (a b c) and (a (b c) ( )) are shown in Figure 23.1.

23.3. Programs and evaluation

A Scheme program is a list, so the syntax of programs is identical to the syntax of lists. A program is just a list that is passed to the Scheme evaluator to be computed. Computation is done as follows. First, the car of the list is extracted, and evaluated. It must be a function or one of the special words such as **if** or **define**. The cdr of the list contains the parameters of the function, and the function is run on those parameters. For example, when list (+ 2 3) is evaluated, the Scheme evaluator extracts function + and parameter list (2 3). Since the function called + performs addition, the result is 5. To avoid confusion between a list and its value as an expression, we write $\langle L\rangle$ for the value produced when list L is evaluated. So, for example, $\langle(+\ 2\ 3)\rangle = 5$. We also say that evaluation of (+ 2 3) yields 5.

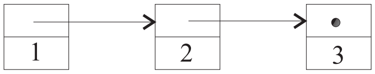
Scheme does not have any special syntactic forms such as binary operators. If you want to compute the product of x and y, write (* x y), not (x * y). The arithmetic functions +, *, − and / all take arbitrarily many parameters, with operations done from left to right. For example, $\langle(+\ 3\ 5\ 7\ 9)\rangle = 24$.

You can have arbitrarily complicated expressions, with parameters computed by subexpressions. For example, evaluation of (+ (+ 4 5) (+ 2 6)) yields 17. Scheme normally uses inside-out evaluation, so each subexpression (+ 4 5) and (+ 2 6) is evaluated before doing the first +. For a few special functions, such as **if**, **cond** and **define**, Scheme deviates from inside-out evaluation, and instead passes the function its arguments as unevaluated lists. The function decides what to do with those lists. The actions of **if**, **cond** and **define** are discussed below.

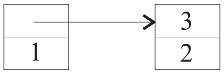
Creating lists in programs

Be careful about writing lists in programs. For example, suppose that you want the list (1 2 3). If you write (1 2 3) *as a program*, and evaluate it, the Scheme evaluator will presume that you intend the car of this list, the number 1, to be a function. That will cause no problem when the program is compiled, since the compiler does not have any concept of type, and so does not recognize that 1 is not a function. When you run the program, however, you will get a run-time type error.

In a program, the empty list is written '( ). To use the list (1 2 3) in a program, you need to write an expression whose value is the list (1 2 3). An expression that does the job is (cons 1 (cons 2 (cons 3 '( )))). Recall that cons is the Scheme equivalent of the :: operator of Cinnameg. Expression (cons 1 (cons 2 (cons 3 '( )))) is the Scheme equivalent of Cinnameg expression 1::2::3::[ ]. Be sure to notice the empty list ( ) at the end. What you want, in terms of a representation diagram, is



If you forget the empty list and write expression (cons 1 (cons 2 3)), you get the following diagram instead.



This is called an *improper list* in Scheme, since it does not end on ( ). Although you can build it, you need to be careful when you use it, since most functions on lists require proper lists.

Several cons operations in a row can be done using the **list** function. Expression (list 1 2 3) evaluates to the list (1 2 3), and $\langle(\text{list }(+\ 1\ 1)\ (+\ 3\ 4))\rangle = (2\ 7)$. Another approach to building list (1 2 3) is to *quote* the list, by putting a single quote mark in front of it. Expression '(1 2 3) evaluates to the list (1 2 3). Quoting a list gives you a literal list, *exactly* what you write after the quote, without evaluating *anything* in it. So $\langle$'(+ (+ 2 3) (+ 1 2))$\rangle$ is the list (+ (+ 2 3) (+ 1 2)) (a list of three things: the symbol +, the list (+ 2 3) and the list (+ 1 2)). Symbols are discussed in the next section.

23.4. Values in Scheme

In addition to lists, Scheme offers a collection of primitive values. Numbers, such as 34 and 16.23, are supported. Boolean values are #t (true) and #f (false). The character *a* is written #\a, the space character is written #\space, and the newline character is #\newline. Strings are enclosed in double quotes, such as "abc". In Scheme, a string is not considered to be a list, and does not have a car or cdr.

Symbols

One of the most useful kinds of data in Scheme is a *symbol*. A symbol is just a name. Symbols are similar to strings, but have an advantage over strings that equality testing is very efficient for them. A symbol is stored as an integer symbol-identity, and there is a table that converts between symbol names and identities. To ask whether two symbols are the same, you just compare their identities, rather than comparing strings character-by-character.

To get a symbol in a Scheme program, quote it. For example, 'camel is the symbol *camel*. If you quote a list, as in '(the fox hid in the bushes), then each symbol in the list is implicitly quoted. Do not write '('the 'fox 'hid 'in 'the 'bushes), since then the quotes themselves are quoted, and they become part of the list.[1]

23.5. Defining functions

To define a function *f*, use **define**. It appears like a function whose first parameter is a list that looks like an application of *f* to some arguments and whose second parameter is an expression that computes the value of the function. For example, definition

```
(define (sqr x) (* x x))
```

indicates that (sqr x) is the same value as (* x x), for any x. You can think of this definition as an equation stating that (sqr x) and (* x x) are equal. But there is no equal sign in the syntax. All Scheme constructs use the syntax of lists, without any wasted words or extra symbols.

23.6. Making choices

Scheme does not support definition by cases in the way that we have used in prior chapters. Instead, a single definition must be augmented by a conditional expression that embodies the choice.

Binary choices: if

The basic conditional function is **if**. The Scheme system evaluates expression (if a b c) by first evaluating a. If a comes out to be #f (false), then the value of expression (if a b c) is the same as the value of expression c. If a comes out to be anything except #f, then the

[1] Notation 'x abbreviates (quote x). (The word **quote** is one of the reserved words.) If '('the 'fox 'hid 'in 'the 'bushes) is evaluated, it yields list ((quote the) (quote fox) (quote hid) (quote in) (quote the) (quote bushes)), where **quote** itself is a symbol.

value of *b* is taken instead. For example, here is a definition of a function that takes the length of a list. It uses built-in function null?, where evaluating (null? '( )) yields #t, and (null? x) yields #f when x is not an empty list.

```
(define (length x) (if (null? x) 0 (+ 1 (length (cdr x)))))
```

That definition of the length function is difficult to read. Typically, Scheme programs are presented in multiple lines, with indentation showing structure. The length function would normally be written as follows to improve readability.

```
(define (length x)
  (if (null? x)
    0
    (+ 1 (length (cdr x)))
  )
)
```

Multiway choices: cond

There is a more versatile conditional function called **cond**. Expression (cond (t1 v1) (t2 v2) ... (else v)) begins by testing condition t1. If ⟨t1⟩ is #t (or anything other than #f) then the cond expression returns ⟨v1⟩. But if ⟨t1⟩ is #f, then the cond expression tests ⟨t2⟩. Testing continues until a pair is found whose car evaluates to something other than #f, or until the else-pair is encountered. Reserved word **else** is treated like #t, so it is always true. The value of the conditional expression is given by the second member of the first pair whose condition is true.

For example, here is a definition of (drop *n* *L*), which yields the result of removing the first *n* members of list *L*, or ( ) if *L* has fewer than *n* members. It uses function =, which does an equality test.

```
(define (drop n L)
  (cond
    ((null? L)  '())
    ((= n 0)    L)
    (else       (drop (- n 1) (cdr L)))
  )
)
```

Boolean operators

Boolean functions include **not**, **or** and **and**. For example, ⟨(not #t)⟩ = #f, and ⟨(or #f #t)⟩ = #t. The **and** and **or** symbols are not ordinary functions; their evaluation is short-circuited, and they only evaluate enough of their parameters to determine the answer. For example, evaluation of (or #t *E*) will not evaluate *E* at all.

Scheme takes full advantage of its typeless nature. All of the standard boolean functions allow parameters that are neither #t nor #f, with anything other than #f treated as a true value. The **or** function, for example, produces the first of its parameters that evaluates to a value other than #f, or returns #f if there are no such values. So ⟨(or '(3 4) 6)⟩ = (3 4).

```
Define
  case append ([], y)   = y
  case append (h::t, y) = h :: append(t,y)
%Define
```

(a)

```
Define append (x,y) =
  If x == []
    then y
    else (head x) :: append(tail x, y)
  %If
%Define
```

(b)

```
(define (append x y)
  (if (null? x) y
    (cons (car x) (append (cdr x) y))
  )
)
```

(c)

Figure 23.2: Definitions of list concatenation (called append here) in (a) Cinnameg (using definition by cases), (b) Cinnameg (with an *if* to handle the cases) and (c) Scheme. The append function is standard in Scheme.

23.7. More examples in Scheme

It is straightforward to translate simple equational function definitions from Cinnameg into Scheme. Just remember that Scheme has no pattern matching (of the form that we have been using) so you must break things down using basic functions such as car and cdr. Also remember that Scheme has no definition by cases (again, of the form that we have used) so all definitions must be written as a single equation, using conditional functions to make choices. Remember to use Scheme syntax to call functions, not common notation such as binary operators.

Figure 23.2 shows definitions of a list concatenation function *append* in Cinnameg and Scheme. Append is one of the standard functions provided by Scheme. Chapter 19 introduces the naive reverse algorithm to reverse a list. Figure 23.3 shows definitions of naive reverse in Cinnameg and Scheme.

```
Define
  case reverse []       = []
  case reverse (h :: t) = reverse t ++ [h]
%Define
```

(a)

```
(define (reverse x)
  (if (null? x) '()
    (append (reverse (cdr x)) (list (car x)))
  )
)
```

(b)

Figure 23.3: Naive reverse written in (a) Cinnameg and (b) Scheme.

23.8. Run-time type checking and typeless programming

In Scheme, all type checking is done at run time. By simply omitting type checks, you can treat different types of data in a uniform way. You automatically get *polymorphism*, which allows a function to work on many different kinds of data. For example, the length function will take the length of any list; it does not care what is in the list.

Occasionally, you need to check that the data in hand is of the correct type. A *predicate* is a function that returns a boolean result, and Scheme provides predicates that test for membership in specific "types". Typically, predicates have names that end on a question mark. For example, predicate **number?** is true of any number, **boolean?** is true just when its parameter is either #t or #f, and predicate **list?** is true of any list. Other standard predicates include **pair?** (true when its parameter is a nonempty list), **symbol?** (true when its parameter is a symbol), and **procedure?** (true when its parameter is a function, called a procedure in Scheme).

Run-time type checking is versatile, since it allows the program to respond dynamically to whatever kind if data it sees. The philosophy is to get out of the programmer's way, and let the programmer do whatever he or she wants to do. To illustrate, Figure 23.4 uses type predicates to flatten a list, removing all internal structure from it.

The down side of run-time type checking is that almost anything that you write, no matter how full of nonsense, will compile. The main requirement is that the parentheses balance. That forces the programmer to discover mistakes via debugging that would have been pointed out by a compiler that performed reasonable type checking.

23.9. Programs as data: a simple interpreter

Since a program has the same form as a list, it is very easy to treat programs as data and data as programs. A Scheme interpreter just takes a linked list and evaluates it. The only

```
(define (flatten x)
  (cond
    ((null? x) '())
    ((list? x) (append (flatten (car x)) (flatten (cdr x))))
    (else (list x))
  )
)
```

Figure 23.4: **A definition of function flatten, which removes internal structure from a list.** Evaluating expression (flatten '((a b) '( ) ((c) (d)) e)) produces list (a b c d e), where all list structure has been removed, and only the non-list components are left. Flatten uses predicate list? to tell whether its parameter is a list. Expression (list x) produces a singleton list containing only x.

role of a compiler in such an implementation is to convert a list from textual form into linked list form.

One of the built-in functions that a program can use is the Scheme interpreter itself. Evaluating expression (eval e v) runs the Scheme interpreter on list $\langle e \rangle$. That is, it feeds the result of e back into the interpreter to be evaluated again. Using eval, a program can construct another program on the fly and then run it immediately. (The second parameter of eval provides bindings of identifiers to be used in doing the evaluation. Expression (eval e (interaction-environment)) evaluates Scheme expression $\langle e \rangle$ in the same environment as would be used for interaction with a user, which includes definitions made by the current program.)

Since programs and data have the same form, it is easy to write your own interpreter. An interpreter is also called an *evaluator*, since it evaluates an expression. Figure 23.5 shows an evaluator called eval-expr that only allows two functions, plus (addition) and times (multiplication), and that also allows numeric constants. For example, (eval-expr '(plus (times 2 3) 5)) yields value 11. Scheme allows the addition and multiplication functions to have an arbitrary number of parameters. For example, (+ 5 2 8 4) evaluates to 19. Our evaluator also allows an arbitrary number of parameters. The cases in the definition of eval-expr are as follows.

1. If n is a number then (eval-expr n) should just produce n. There is nothing to evaluate.

2. Suppose E is the list (plus a b c d). Then (eval-expr E) must first evaluate each of a, b, c and d, since they might be complex expressions. Function eval-params will do that job. The cdr of E is (a b c d), and (eval-params (cdr E)) should yield list (a', b', c', d') where a' is $\langle$(eval-expr a)$\rangle$, b' is $\langle$(eval-expr b)$\rangle$, etc.

   Since the example expression E requests an addition, the result of (eval-expr E) is the same as the result of (+ a' b' c' d'), where + is the built-in Scheme addition function. Using eval-params, we have computed the list $r = (a'\ b'\ c'\ d')$. Unfortunately, evaluating (+ r) yields an error. The addition function cannot take a list of numbers as a parameter; the parameters must be in the same list that contains the + function itself. That problem is overcome by using the built-in function **apply**, where $\langle$(apply f (list a b c))$\rangle$ produces the same thing as $\langle$(f a b c)$\rangle$. So if E is a list whose car is the symbol plus, then $\langle$(eval-expr E)$\rangle$ is the same as $\langle$(apply + (eval-params (cdr E)))$\rangle$.

```
(define (eval-expr E)
  (cond
    ((number? E) E)
    ((eqv? (car E) ’plus)  (apply + (eval-params (cdr E))))
    ((eqv? (car E) ’times) (apply * (eval-params (cdr E))))
    (else ’())             ; confused - return ()
  )
)

(define (eval-params E)
  (if (null? E) ’()
    (cons (eval-expr (car E)) (eval-params (cdr E)))
  )
)
```

Figure 23.5: **Definition of an evaluator for simple arithmetic expressions.** This evaluator allows only plus and times and numbers in expressions. Expression (apply *f* args) has the effect of running function *f* on the parameters in list args. For example, (apply + (list 2 3)) is treated like (+ 2 3). Predicate eqv? is used to perform equality tests of symbols, as opposed to =, which works for numbers. See Section 23.13.

3. Multiplication is handled similarly to addition.

Handling variables

Function eval-expr is very simple, and does not handle much at all. For example, it does not allow any variables. Evaluating (eval-expr '(plus r 2)) does not make any sense, since eval-expr does not know what to do with symbol *r*. (Notice that, due to the quoting, the Scheme interpreter does not replace *r* by its value. What eval-expr sees is the symbol *r* itself.)

How might variables be handled? You would need to add another parameter to eval-expr: the environment in which evaluation takes place. For example, (eval-expr-with-env *expr env*) should produce the result of evaluating expression *expr* in an environment given by *env*. The environment can be a list of pairs, called an *association list*, where pair (*x* 3) indicates that variable *x* has value 3. Evaluating (eval-expr-with-env '(plus *x* *y*) '((*x* 3) (*y* 6))) should yield 9, since the environment indicates that *x* has value 3 and *y* has value 6. Definition of eval-expr-with-env is left as an exercise.

23.10. Higher order programming in Scheme

You can create an anonymous function using the built-in function constructor **lambda**, where, for example, the value of expression (lambda (*x*) (* *x* *x*)) is the function that returns the square of its argument. Function (lambda (*a* *b*) (list *a* *b*)) takes two parameters and produces a list containing those values.

Scheme is a higher order language, with functions as first class values. The **map** function is standard. If expression *L* evaluates to list (*a* *b* *c*), then expression (map *f*

```
(define (map f L)
  (if (null? L) '()
    (cons (f (car L)) (map f (cdr L)))
  )
)
```

Figure 23.6: **A definition of map in Scheme.** This definition is less general than the standard Scheme map function.

L) evaluates to the same thing as expression (list (*f a*) (*f b*) (*f c*)). Figure 23.6 show a definition of map in Scheme. Expression (map (lambda (*x*) (* *x x*)) (list 3 4 5)) evaluates to list (9 16 25). A little thought shows that function eval-params of Figure 23.5 can be expressed easily using map. Doing so is left as an exercise.

The standard Scheme **map** function is actually a little more versatile than the one written above. It allows any number of parameters, where the first is a function and the rest are lists. If there are two or more lists, then they must have the same length. Evaluating (map *f* (list *a b c*) (list *u v w*)) yields the same thing as evaluating (list (f *a u*) (f *b v*) (*f c w*)). It applies *f* to the two first members of the lists, then to the two second members, etc.

23.11. Giving things names

Naming using lambda

Cinnameg expression (Let z = 5 * 5. (z + z)) evaluates 5*5 and gives a name, *z*, to the result. But suppose that you could not use **Let** to bind *z*, and had to accomplish the same thing by another mechanism. You could use expression ((z |-> z + z)(5*5)), letting the parameter passing bind *z* to the result of evaluating 5*5. You can do the same thing in Scheme; ((lambda (z) (+ z z)) (* 5 5)) evaluates to 50.

Naming using let, let* and letrec

The need to use function application to simulate name binding is unpleasant, and Scheme provides a better approach. Expression (let ((*x v*)) *e*) indicates that identifier *x* should be bound to the value of expression *v* and then expression *e* should be evaluated. The value of *e* is used as the value of this let expression. For example, Scheme expression

```
(let ((z (* 5 5)))
  (+ z z)
)
```

has value 50. Expression (let ((*x v*)) *e*) just abbreviates for ((lambda (*x*) *e*) *v*).

A let expression allows you to bind several names, by providing more than one (variable value) pair in the binding list. For example, expression

```
(let ((x u)
      (y v))
  e
)
```

abbreviates ((lambda (x y) e) u v). But you need to be careful with this. Translating expression (let ((x 2) (y (+ x 1))) (* x y)), where the binding of y uses x, into its lambda form yields ((lambda (x y) (* x y)) 2 (+ x 1)). But that is a problem. The value of x is not known outside of the function, but it is used in expression (+ x 1). The bindings are only in effect inside the expression that gives the value of the let expression, not in expressions that perform other bindings.

Scheme provides a way around the scope problem of **let**. Expression (let* ((x u) (y v)) e) abbreviates

```
(let ((x u))
  (let ((y v))
    e
  )
)
```

so that each binding can use previous bindings.

There is another difficulty with let expressions; they cannot be used to define recursive functions, since the identifier being defined is not available at the point where you would want to use it. Let* expressions do not help, since they only allow each binding to use prior bindings, not itself. For local recursive definitions, you can use **letrec**. Expression

```
(letrec ((len
           (lambda (x)
             (if (null? x) 0 (+ 1 (len (cdr x))))
           )
         ))
  (len (list 20 30 40)))
)
```

evaluates to 3.

23.12. Imperative programming in Scheme

Although functional programming is encouraged in Scheme, Scheme is not a purely functional language. It also has some imperative aspects.

Rebinding variables

You can change the binding of an identifier using set!. For example, (set! 'goose '( )) rebinds symbol goose to ( ). Notice that goose needs to be quoted. The arguments to set! are evaluated before being passed to set!. If you write (set! goose '( )), then the value of variable goose is passed to set! as the name of the symbol to be rebound. For that to make sense, goose would need to hold a symbol.

Expression (begin e_1 e_2 $\ldots e_n$) evaluates each of e_1, e_2, $\ldots$, e_n, in that order, and then produces the value of e_n as its own value. Typically, expressions e_1, $\ldots$, e_{n-1} perform actions, using features like set!, and e_n computes a value.

```
(define (sum L)
  (do ( (x     L (cdr x))
        (total 0 (+ total (car x)))
    ((null? x) total)
  )
)
```

Figure 23.7: A function that computes the sum of the members of a list of numbers using a loop.

Loops

Scheme offers a kind of loop called a *do loop*. In simplified form it is as follows.

```
(do ((v1 i1 s1) (v2 i2 s2) ...)
   (test r)
   c1 ... cn
)
```

Each of v_1, v_2, ... is a control variable for the loop. Expression i_1 gives the initial value of v_1, i_2 is the initial value of v_2, etc. Each time around the loop expression *test* is evaluated. If the test expression yields #t, then the loop is ended and the value of the loop expression is the value of expression r. If the test yields #f then each of commands $c_1, \ldots, c_n$ is evaluated, in sequence. (It is allowed for n to be 0, so there aren't any commands to do.) After the loop body is done, each control variable is updated, with v_1 set to value of expression s_1, v_2 set to s_2, etc., where the updates are done in parallel, as for a let expression. The loop then runs for another iteration, with expression *test* being tried again, and so on. For example, function sum of Figure 23.7 computes the sum of the members of a list of numbers using a loop.

Be wary before embracing loops too quickly. Remember that Scheme is designed to be a functional language.

23.13. Equality tests

Early versions of Lisp were implemented on computers that, by today's standards, were extraordinarily slow, and efficiency of computation was a critical concern. Equality tests needed to be done frequently during execution of Lisp programs, so it was important to make them efficient. That meant using more than one equality testing function. Scheme has inherited that legacy.

In some cases, equality tests can be done by comparing pointers or integers. Scheme function eq? does that. That works for asking whether something is equal to a symbol, for example, since a symbol is represented by its integer identity number.

Function eq? works for comparing integers, but it does not recognize that the integer 0 is equal to the real number 0.0. Function = remedies that, but it only works correctly for comparing numbers. Function eqv? combines = and eq?. It yields true when either eq? or = would.

None of those functions works for comparing lists, unless one of the lists is empty. Predicate equal? is a general test that works correctly on a large variety of things, including lists. Evaluation of (equal? '(a b c) '(a b c)) yields #t, but evaluation of (eq? '(a b c) '(a b c)) yields #f.

Offering the programmer the flexibility to choose among equality tests might sound like a good idea, but it has a difficulty. With several equality tests to choose from, it is not obvious which one should be used for standard functions that need to perform equality tests, such as testing for membership in a list. To ask whether x occurs in list L, should you ask whether (eq? x (car L)) yields true, or (equal? x (car L)), or something else? Scheme's solution is to provide three different membership testing predicates. Function memq uses eq?, memv use eqv? and member uses equal?.

23.14. Dot notation and cons cells

The basic data structure of Scheme is the *cons cell*, which has two parts, the car and the cdr. Scheme has a way to describe a cons cell directly; notation $A.B$ stands for a cons cell whose car is A and whose cdr is B. So list (x y z) can also be written $A.B.C.(\,)$.

Scheme's dot operator is almost identical to the :: operator of Cinnameg. There is an important difference, however. When the dot operator occurs in a program, it is only used to construct the program itself. It is *not* equivalent to performing the cons function. If you evaluate expression '(goat.sheep), you get a cons cell

sheep	(cdr)
goat	(car)

But if you try to evaluate expression ('goat.'sheep), you get an error because the car of this list, the symbol goat, is not a function. But evaluation of (cons 'goat 'sheep) yields the same cons cell as ⟨'(goat.sheep)⟩. When the interpreter evaluates (cons 'goat 'sheep), it extracts the car of this list, the function cons, and runs that function to produce the desired cons cell.

Using dot notation, it is easy to construct improper lists. Be careful about that, since most functions expect lists to be proper.

23.15. Summary of terminology and concepts

Scheme is designed in the tradition of the Lisp family of programming languages. Programs and data have the same form, making it easy to treat a data value as if it is a program or to treat a program as data. The syntax of Scheme is very simple. There is just one fundamental kind of data structure, linked lists.

Scheme is dynamically typed. Type errors are only reported at run-time.

Most dialects of Lisp are large languages due to the large collection of functions and options that they provide. Scheme has instead chosen simplicity, with relatively few built-in concepts.

Scheme is a higher-order programming language, supporting functions as values. It encourages a functional style of programming, but also supports imperative programming.

Terminology

Association list An association list is a list of pairs, each indicating an association or relationship between two things. You normally look up information in an association list by looking for a pair that has a given value as its car.

Car The car of a list is the head of the list.

Cdr The cdr of a list is the tail of the list.

Cons The cons function is the equivalent of the Cinnameg :: operator, which builds a list with a given head and tail.

Cons cell A cons cell is an individual data cell with two parts, the car and the cdr. All data structures in Scheme are constructed by linking cons cells together.

Dot notation Scheme expression A.B stands for a cons cell whose car is A and whose cdr is B. Be careful about using this in programs, since it only describes the program itself, by building a cons cell that is part of the program. To build a cons cell at run time, use cons.

Evaluator An interpreter is also called an evaluator.

Improper list A list is improper if it does not end on ( ).

Lambda Lambda is used to build an anonymous function. It is one of the most basic aspects of Scheme.

Lisp Lisp is a family of related languages that are based on lists and a simple syntax similar to that of Scheme. Scheme is in the Lisp family.

List The basic data structure of Scheme is a linked list.

Symbol A symbol is a name. When a program is compiled into a linked list, each identifier in it is replaced by a symbol. Symbols in programs normally stand for variables; to get a literal symbol, add a single-quote mark.

23.16. Exercises

23.1. What is the value of Scheme expression (car (cdr (cons ’horse (cons ’zebra ’( )))))?

23.2. Write a Scheme definition of function **member?**, where (member? x L) is #t when x is a member of list L, and is #f otherwise. For this function you need to do equality tests. Use function equal? to do those tests.

23.3. Write a Scheme definition of function **member**, where (member x L) is the first suffix of list L that has x as its car, and is #f if x does not occur in L. For example, ⟨(member 3 (list 2 3 4 5))⟩ = (3,4,5). Use equal? for equality tests. This is what the standard Scheme member function does.

23.4. Write a Scheme definition of function **prefix**, where (prefix x y) is #t if list x is a prefix of list y, and is false otherwise. The empty list is a prefix of every list, and a list is considered to be a prefix of itself.

23.5. Write a definition of function **list-equal?**, which compares two lists by comparing their structures. Assume that non-lists can be compared using function eqv?. Be careful to handle list members that are themselves lists using your list-equal? function. For example, evaluating (list-equal? '((a b) (c)) '((a b) (c))) should yield #t.

23.6. Write a Scheme function **descending?** so that (descending? L) is true for a list of numbers L if L is in descending order. You can use Scheme expression ($<$ x y), which yields true if $x < y$. The empty list is in descending order by definition, as is a singleton list.

23.7. Write a definition of the foldRtoL function in Scheme.

23.8. Function eval-params of Figure 23.5 is really performing a map operation. Show how to replace it in the definition of eval-expr with a use of the built-in Scheme map function.

23.9. Write function eval-expr-with-env, where (eval-expr-with-env e B) evaluates expression e in environment B. The environment is an association list. Be careful about evaluation time. Use inside-out evaluation. So if pair (x v) occurs in the environment, then v should be a number, not an unevaluated expression. You will find standard Scheme function **assoc**, described in Exercise 23.14, to be useful in looking up bindings in the environment.

23.10. The eval-expr-with-env function of Exercise 23.9 uses eager evaluation, storing a fully computed number with each name. What would be involved in using lazy evaluation instead? Would it be enough simply not to evaluate an expression, and to store the unevaluated expression in the environment, or is more than that required? Think about some examples.

23.11. Scheme supports functions set-car! and set-cdr!, where (set-car! x v) changes the car of list x to be v, and (set-cdr! x v) changes the cdr of list x to be v. The existence of those functions has a bearing on the semantics of lists in Scheme. Argue that it is not adequate to say that a list is a sequence of things. What is a viable semantics of lists in Scheme? Why do you think Cinnameg does not support analogs of set-car! and set-cdr!?

23.12. As indicated in the preceding question, functions set-car! and set-cdr! force you to take the representation of lists into account in the semantics of the language. Argue that the eq? predicate has the same effect. Remember that eq? compares memory addresses. If L is a list, then evaluating (eq? L L) yields #t since L is stored at the same memory address as itself.

23.13. Scheme provides some abbreviations for extracting parts of lists. Expression (cadr L) is equivalent to (car (cdr L)), and (cdar L) is equivalent to (cdr (car L)). You can have up to 6 letters between c and r. For example, (caddr L) is equivalent to (car (cdr (cdr L))). Say which of these functions should be used for each of the following jobs.

(a) Get the second member of a list.

(b) Get the third member of a list.

(c) Get the first member of the second member of a list. (That is, get x from L, where L is (a (x ...) ...)

23.14. The Scheme function **assoc** is defined so that (assoc x L) returns the first member of list L whose car is equal to x (using equality function equal?). If there is no such member of L, then (assoc x L) returns #f.

(a) Assoc is typically used for getting bindings out of lists. For example, (assoc 'x '((y 2) (x 5) (z 9))) evaluates to (x 5), telling you that symbol x is bound to 5. How can you extract the binding of x from the result? Can you use one of the functions described in Exercise 23.13?

(b) You can also use cons cells (improper lists) for the members of L. Evaluation of (assoc 'x '((y.2) (x.5) (z.9))) yields (x.5). How can the binding of symbol x be extracted from the result?

(c) Explain why a programmer might want to use improper lists as the members of list L instead of (proper) lists of length two.

23.15. Write a Scheme definition of the assoc function described in the preceding question.

23.17. Bibliographic notes

Scheme has been revised over time, and is described in various revisions of the Scheme report [88, 89]. Abelson and Sussman [4] use Scheme as an introductory programming language. The Scheme report is available from www.schemers.org/Documents/Standards/R5RS/HTML/. An implementation of Scheme is available from www.drscheme.org.

Steele [99] and Graham [45] describe Common Lisp.

Chapter 24

Haskell: Pure Functional Programming

24.1. Introduction to Haskell

Haskell is a polymorphically typed purely functional programming language whose development began in the late 1980s and continued through the 1990s and 2000s. It is undergoing constant improvement, but its core has remained quite stable. The following are some of its interesting features.

1. Haskell is nonstrict, meaning that it uses lazy evaluation as its normal mode of evaluation.

2. Haskell uses the idea of layout to achieve a very compact syntax, and generally employs short and simple notation that makes programs easy to write. For example,

   ```
   if x > y
     then n + 1
     else (m + 4)
            - 5*n
   ```

   uses indentation to show structure. The 'else' part has been broken into two lines for illustration. (Although you typically use layout, you can also use explicit braces, $\{a;$ $b;$ $c\}$ to group things, separating them by semicolons. In fact, the Haskell compiler just inserts braces and semicolons based on the layout.)

3. Haskell has a rich type system, providing support for abstractions and type variables, with type inference, as studied in Chapter 37.

4. Because Haskell is purely functional, it needs to find ways to do such usually imperative features as input and output without the need for side effects in the language.

This chapter gives a brief introduction to Haskell. It does not attempt to cover all aspects of Haskell. The references at the end of the chapter offer far more detail.

+, −, ∗, /	arithmetic binary operators
‘div‘, ‘mod‘	integer quotient and remainder operators
ˆ, **	power operators (ˆ raises to integer power, ** to real power)
>, <, >= <=	comparisons
==, /=	equality and inequality tests
&&, ||, not	Boolean ’and’ and ’or’ (binary) and negation (unary)
:	$h : t$ is the list whose head is h and whose tail is t.
++	list concatenation operator
head, tail	get the head or tail of list
length	get the length of a list
!!	$x!!n$ is the n-th member of list x, counting from 0. For example, $x!!0$ is the head of x.
fst	get the first member of an ordered pair. (Unlike Cinnameg, Haskell does not consider an ordered triple to be a special kind of ordered pair, so you cannot use fst to give the first member of a triple.)
snd	get the second member of an ordered pair
.	$f.g$ is the composition of functions f and g. That is, $(f.g)(x) = f(g(x))$.

Figure 24.1: **Some standard functions and operators.** The standard functions are typically curried. Expression x ‘op‘ y abbreviates (op x y), and $x + y$ abbreviates $(+)(x, y)$. Haskell is case-sensitive, and these function names need to be written exactly as shown.

24.2. Lexical issues

Identifiers begin with a letter and can contain letters, digits and special characters _ and ’. Symbolic operator names are one or more of the special symbols in string “!#$%*+−./<>=?\^|:~”. Comments can either begin with `--` and extend to the end of a line or be enclosed in {- and -}.[1]

24.3. Expressions and values

Haskell supports typical values such as integers (32), real numbers (32.5) and characters (’a’). Boolean constants are True and False. Be sure to capitalize them. Haskell supports ordered pairs such as (5, True) and list such as $[1, 2, 3]$ in the same form as Cinnameg. A string is a list of characters; ”red” abbreviates [’r’, ’e’, ’d’].

Figure 24.1 shows a few standard functions and operators. Juxtaposition indicates function application, and associates to the left, just as in Cinnameg. For example, `length` x is the result of applying function length to x. Expression `if` a `then` b `else` c is b if a is

[1] There is a *literate* style of Haskell in which all lines are presumed to be comments except those that begin with >. We will use the ordinary style.

True and is c if a is False. Expression `let` x `=` A `in` B yields the value of B, where x has value A. For example,

```
let amphibian = "frog" in
  amphibian ++ "s and more " ++ amphibian ++ "s"
```

defines amphibian to be ”frogs and more frogs”. Since Haskell is always lazy, the value of a variable is only computed when it is used.

Expression (\x -> E) is the function that takes parameter x and produces the value of expression E. For example, the value of (\y -> y + 1) is a function that returns one larger than its parameter.

List [1 . . 5] is the list $[1, 2, 3, 4, 5]$. List comprehensions (Section 20.6) allow any number of generator parts (such as x <- ys) and filter parts (such as isEven x), separated by commas. For example, [2*n | n <- [2..5]] yields $[4, 6, 8, 10]$, [x | x <- [1..8], isEven x] yields [2,4,6,8] and [(x,y) | x <- [2..4], y <- [2..4], x < y] yields [(2,3), (2,4), (3,4)].

24.4. Definitions and patterns

A definition has the form of an equation where the left-hand side is a name or a function name applied to one or more variables or patterns. Looking ahead toward types, a stand-alone definition (not part of a larger definition) is typically (but not necessarily) preceded by an indication of its type. You can include more than one equation; they are tried in the order written. Here are some examples.

```
num :: Int -- num has type Int
num = 43 + 19

double :: Int -> Int
double n = 2*num

reverse :: [Int] -> [Int]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]

max :: Int -> Int -> Int
max x y =
  if x > y then x else y
```

The left-hand side of an equation can be a pattern. For example,

$$(x,y) = p$$

makes x the first member of ordered pair p and y the second member. Each variable can occur only once, so no equality checks are performed. You can use pairs and lists, or expressions of the form h:t, or constants. You can also use patterns such as $n + 1$. An underscore is a “don’t care” variable.

You can try several pattern matches against one value using a **case** construct. Each case consists of a pattern, an arrow (->) and a value that the expression takes on if the pattern match succeeds. For example, expression

```
case x of
  []  -> 1
  _:_ -> 2
```

produces value 1 when x is an empty list and 2 when x is a nonempty list. Cases are tried in order.

To use provisos (called *guards* in Haskell) for equations, use a single left-hand side, but precede each right-hand side by a bar, the condition under which it works, and an equal sign. For example, an alternative definition of max is as follows.

```
max x y
  | x > y     = x
  | otherwise = y
```

As shown, the last case can be labeled **otherwise** to select it whenever it is reached. (If none of the cases have true guards, the next equation or group of equations is tried.) After an equation or list of guarded right-hand sides you can include a **where** clause that defines variables that can be used throughout that equation. For example, maxabs(x,y) = max($|x|$, $|y|$) can be defined as follows.

```
maxabs x y
  | ax > ay   = ax
  | otherwise = ay
    where
      (ax, ay) = (absval x, absval y)
      absval z
        | z >= 0    = z
        | otherwise = -z
```

Notice that the definitions of ax and ay use absval before it is defined. Haskell allows definitions to be written in any order and used before they are defined. Of course, definition

```
maxabs x y = max(abs x, abs y)
```

using the standard abs function is much simpler.

24.5. Types

Haskell is a strongly statically typed language. Some concepts used here are covered in later chapters of this book.

Types have names that begin with an upper case letter, and include Int, Float, Bool, Char. Ordered pair (2,True) has type (Int, Bool). List [1,2,3] has type [Int] and "jump" has type [Char]. Type name String abbreviates [Char].

You can give a new name to an existing type using a **type** definition. For example, definition

```
type Age = Int
```

defines Age to be another name for Int. Haskell also allows you to define new types as discussed in Chapter 34. You can create a new type using a **data** definition. The left-hand side is the name of the new type and the right-hand side has a constructor name followed by zero or more types, where the type and constructor names are required to start with upper case letters. For example,

```
data Name = MakeName String
```

defines type Name to be represented as a string. Constructor MakeName has type String → Name. To get the string that a given Name represents, use pattern matching. For example,

```
plural(MakeName str) = MakeName(str ++ "s")
```

makes one Name from another by adding an 's'.

A type can have variants, as discussed in Section 34.4. A simple example is an enumerated type, such as

```
data Flower = Orchid | Rose | Gladiolus
```

Types can be recursive. For example, the following defines a binary tree type with nodes labeled by integers.

```
data BinaryTree = EmptyTree |
                  Node Int BinaryTree BinaryTree
```

The second variant says that a binary tree constructed by Node contains an integer and two other binary trees; Node x L R builds a tree with root label x, left subtree L and right subtree R. The number of nodes in a tree is easy to compute using primitive recursion.

```
numNodes :: BinaryTree -> Int
numNodes EmptyTree      = 0
numNodes (Node _ t1 t2) = numNodes t1 + numNodes t2
```

24.6. Polymorphism

Haskell is polymorphically typed in the sense discussed in Chapter 37. A type variable has a name that is an identifier. For example, the head function has polymorphic type [a] -> a.

Types can be parameterized as discussed in Section 37.6. Here is a parameterized version of the binary tree type; type Tree t is a tree whose nodes are labeled by values of type t.

```
data Tree t = EmptyTree |
              Node t (Tree t) (Tree t)
```

Now to sum the numbers in the tree:

```
sumTree :: Tree Int -> Int
sumTree EmptyTree      = 0
sumTree (Node x t1 t2) = x + sumTree t1 + sumTree t2
```

A more interesting function numbers the nodes of a tree in preorder, replacing each node label x by a pair (n, x), where n is the number that this node would have in a preorder traversal of the tree, starting the numbering at 1. The helper takes an extra parameter that is the label to attach to the root, and it returns pair (m, r) where r is the labeled tree and m is one larger than the last label that was used in r (or is just n if r is an empty tree).

```
numberTreeHelp :: Int -> Tree a -> (Int, Tree(Int, a))
numberTreeHelp n EmptyTree = (n, EmptyTree)
numberTreeHelp n (Node x t1 t2)
  = (m2, Node (n,x) nt1 nt2)
    where
      (m1, nt1) = numberTreeHelp (n+1) t1
      (m2, nt2) = numberTreeHelp m1 t2

numberTree :: Tree a -> Tree (Int, a)
numberTree t = snd(numberTreeHelp 1 t)
```

Haskell uses the word *class* for a concept similar to a genus in Cinnameg. Think of a class as a set of types. For example, standard Haskell class Eq contains those types that support equality (==) and inequality (/=) tests. Standard class Ord contains types on which you can perform order tests (<, >, etc.).

To require a type variable to be a member of a given class, write *conditions* `=>` *type*, where each condition has the form $C\ v$, indicating that variable v must belong to class C. If there two or more conditions they are required to be in parentheses and separated by commas. For example, the Quicksort algorithm is as follows.

```
sort :: Ord a => [a] -> [a]
sort [] = []
sort x:xs = sort small ++ [x] ++ sort large
  where
    small = [y | y <- xs, y <= x]
    large = [y | y <- xs, y > x]
```

To create a new class, give a class definition along with types of associated functions (or other things), as illustrated in Figure 24.2. The definition provides default definitions that can be overridden elsewhere. When you add a type to a class, you typically say how to perform the operations that are part of the class signature. For example,

```
instance Eq Bool where
  -- Definition of ==:
  True == True   = True
  False == False = True
  _ == _         = False
```

says that type Bool belongs to class Eq, and defines the == operator for it. (The /= operator uses the default given with the definition of Eq.)

24.7. Modules

A module has the form

```
module name (exports) where
  ...
```

where *exports* tells the names that this module exports and the body (. . .) gives definitions. The things that can be exported are as follows

```
class Eq t where     -- Definition of class Eq.
                     -- If type t belongs to class Eq,
                     -- then the following operations
                     -- are defined for type t.
  (==), (/=) :: t -> t -> Bool

  -- The following are default definitions

  x == y  = not(x /= y)
  x /= y  = not(x == y)

class (Eq t) => Ord t where    -- Definition of Ord
                               -- All types in Ord also
                               -- belong to Eq.
  compare              :: t -> t -> Ordering
  (<), (<=), (>), (>=) :: t -> t -> Bool
  min, max             :: t -> t -> t

  -- The following are default definitions

  x < y  = compare x y == LT
  x > y  = compare x y == GT
  x <= y = compare x y /= GT
  x >= y = compare x y /= LT
  max x y
    | x > y     = x
    | otherwise = y
  min x y =
    | x > y     = y
    | otherwise = x
```

Figure 24.2: **Definitions of classes Eq and Ord.** The definition gives the class name and a variable that represents an arbitrary type in the new class. The variable is used in the polymorphic types of functions (and possibly other kinds of things) that make up the signature of the class. You can include default definitions as well. Definitions made elsewhere will override them.

Ordered types must also possess equality tests, so class Ord is a subclass of Eq. Adding a type to Ord automatically adds it to Eq as well. Type Ordering is defined by

```
data Ordering = LT | EQ | GT
```

```
module Set (Set, set, insert, member) where
  type Set = Tree
  data (Ord a) => Tree a = EmptyTree | Node a (Tree a) (Tree a)
  set :: [a] -> Tree a
  set [] = EmptyTree
  set (h:t) = insert h (set t)
  member :: a -> Tree a -> Bool
  member x EmptyTree = False
  member x (Node k lft rgt)
    | x < k = member x lft
    | x > k = member x rgt
    | otherwise = True
  insert :: a -> Tree a -> Tree a
  insert x EmptyTree = Node x EmptyTree EmptyTree
  insert x (Node k lft rgt)
    | x < k = Node k (insert x lft) rgt
    | x > k = Node k lft (insert x rgt)
    | otherwise = Node k lft rgt
```

Figure 24.3:

1. Just a name indicates that the binding of that name is exported.
2. Form T`(..)`, where T is a type, indicates that T is exported along with its constructors (including the ability to use the constructors in patterns). If you omit (..), then only type T is exported, without its constructors.
3. Form `module` M indicates that all exported bindings in module M are exported to any module that imports the current module. If M is the name of the current module, then all bindings in the current module are exported.

Figure 24.3 shows a sample module that provides sets, represented as binary search trees. To import the Set module, use

```
Import Set
```

You can import only selected things or all but selected things. For example,

```
Import Kangaroo(Kangaroo(..), hop)
Import Wallaby hiding (jump, goOnAllFives)
```

imports only type Kangaroo (with constructors) and hop from the Kangaroo module, and imports everything except jump and goOnAllFives from Wallaby.

Haskell uses names that include the module. When you create module Set, for example, what you really get are Set.Set, Set.insert and Set.member. Typically, you prefer to use short names Set, insert and member, and that is what you normally get. If you want to use the long names, then import the module *qualified*, as follows.

```
Import qualified Set
```

Now you will need to write Set.insert to use the insert function from the Set module.

24.8. Monads and imperative computation

Since Haskell is a pure functional language, it cannot allow a program to perform commands, such as changing the value of a variable or creating a file. But there is a back door available. There is nothing to forbid a program from creating a procedure that has side effects, as long as it cannot run the procedure!

In Haskell, you perform imperative actions by writing a program whose result is a procedure that performs the action. The result, obtained by an agent external to the functional program, can be run by that agent. The effect is that the language itself does not experience any of the difficulties that would result from allowing imperative aspects into it.

How all of this happens is covered in Chapter 50.

24.9. Exercises

24.1. Write a definition in Haskell of each of the following functions.

(a) The factorial function.

(b) The Haskell ++ operator. You can use binary operator notation in definitions, so define what x ++ y is. To test this, you will need to hide the standard ++ operator. Write

```
import Prelude hiding (++)
```

(c) Function dup n x, which produces a list of n copies of x.

(d) removeLeading r xs, which produces the list obtained from list xs by removing all initial occurrences of r. (For example, removeLeading 'z' "zzzazn" = "azn".)

(e) The map function, map f [a,b,c] = [$f\,a$, $f\,b$, $f\,c$].

(f) Function prefix x y, which yields True just when list x is a prefix of list y.

24.2. Some imperative aspects can be simulated by higher-order functions. Standard Haskell function until p f x computes the sequence of values $y_0 = x$, $y_1 = f(y_0)$, $y_2 = f(y_1)$, etc. until it hits the first value y_i such that y_i is true, and returns that value. It is similar to a while loop where predicate p tells when to stop the loop and function f tells how to update.

(a) Write a definition of until.

(b) Write a definition of reverse using until to perform the looping. Use the algorithm that shunts values from one list to another.

(c) Write a definition of factorial using until to do looping.

24.3. Suppose that search x s is intended to yield True if string x occurs as a contiguous substring of string s, and False otherwise.

(a) Write a definition of search, with a type.

(b) Suppose that you declare search to have polymorphic type [a] -> [a] -> Bool. Does the definition of search that you wrote for part (a) work for this polymorphic type, or do the equations need to be changed?

24.4. A directed graph on n vertices can be represented as a list of the form $[L_1, L_2, \ldots, L_n]$, where, for $i = 1, \ldots, n$, L_i is a list of all vertices j so that there is an edge from i to j.

Write a Haskell definition of a function reachable u v g, which returns true if there is a directed path from u to v in g (where g is given as a list of lists, as explained).

24.5. It is important to be able to convert algorithms that are described in an imperative way into a functional style. This exercise has you do that. First, the following is an imperative description of an algorithm.

It is easy to modify the representation of a directed graph to include weights on the edges (indicating distances to be traveled). Just let each list member be a pair (w, j) telling the weight of the edge to j. For example, [[(30,2), (25,3)], [(110,1)], [ ]] describes a graph with three weighted edges: from 1 to 2 (weight 30), from 1 to 3 (weight 25) and from 2 to 1 (weight 110).

There is a well-known algorithm, Dijkstra's algorithm, to find the length of the shortest directed path between two given vertices. It performs a simulation. Imagine sending a signal along an edge. If the edge has weight w, then it takes w seconds to travel from one end to the other of that edge. To find the distance from u to v, start the simulation by sending a signal on every edge that comes out of u.

An *event* is the arrival of a signal at a vertex. The idea is to keep a list of events, each holding the time (from the beginning of the simulation) at which it will occur and the vertex where the signal will arrive. Initially, the list of events should be [(0,u)], forcing a signal to arrive at the start vertex u at time 0. Now process the events in chronological order. To process event (t, x), where t is a time and x is a vertex number, do the following.

(a) If $x = v$ (the end vertex) then the simulation is done. The distance from u to v is t.

(b) If x has not previously been reached by a signal and $x \neq v$, then send a signal along each edge coming out of x by scheduling more events, indicating the arrivals of those signals at the other ends of the edges. If there is an edge from x to y of weight w, then add event $(t + w, y)$ to the event list.

(c) Record that a signal has reached vertex x.

Write the shortest distance function in Haskell. A simple way to simulate a variable that a function wants to change is to pass that variable to the function (giving the variables "current value") and to return an updated value from the function. A function can return several values by returning a tuple. You will probably want some helper functions. Give a clear description of what each one is intended to do. Do not make someone reverse engineer you definitions to understand what they are doing.

24.6. Write a Haskell module that provides type Queue T, a queue whose members have type T. Provide functions to insert a value, remove the first value, look at the first value and ask whether the queue is empty. How can you insert or remove a value in a purely functional language where no data structures can be changed?

24.7. A formula of propositional logic is *valid* if it is true for all values that its variables could take on. For example, $P \Rightarrow P$ and $P \Rightarrow (Q \Rightarrow P)$ are both valid. This exercise is to write a definition, in Haskell, of function valid(r) that returns true if r is a valid propositional formula.

(a) A *formula* can be a variable name (a single letter), or have the form $\neg A$ or $A \vee B$ or $A \wedge B$ or $A \Rightarrow B$, where A and B are formulas. Write a Haskell definition of type Formula. Provide an equality testing function for formulas. Two formulas are equal if they are identical in form. Provide a function that takes a string and produces a Formula

(b) A *sequent* has the form $L \vdash R$ where L and R are lists of formulas. It means, if all of the formulas in list L are true, then at least one of the formulas in list R is true. Either side can be an empty list. (An empty list on the left means that there are no requirements. An empty list on the right guarantees a false sequent, since it is surely not the case that one of no things is true.) The order of formulas in the lists does not matter; you are free to reorder them, and all that matters is whether it is on the left-hand side or the right-hand side.

A sequent is valid if it is true for all values of the variables. Define type Sequent. (Ignore symbol $\vdash$. A sequent is just given by a pair of lists of formulas.)

(c) A sequent is *basic* if there is a formula that occurs in both the left list and the right list. For example, sequent $(P \vee Q, P \vdash P \vee Q)$ is basic since $P \vee Q$ occurs on both sides. Every basic sequent is valid. (Why?) Define a function basic(s) that yields True if s is a basic sequent, and False if not.

(d) Table 24.4 shows rules of inference that allow you to prove sequents are valid, assuming some others are already known to be valid.

To prove that a sequent is valid, keep a goal list of sequents that need to be proved. To prove that formula A is valid, start the goal list holding just one sequent, $(\vdash A)$, where the left-hand side is empty.

Remove a sequent from the goal list and examine it. If the sequent is basic, you are finished with it; move on the the remaining goals. Otherwise, if the sequent has no formula on either side that has an operator (so all formulas are just variables) then it is not valid. Conclude that your original sequent is not valid.

If the sequent is not basic, but it has at least one formula that is not just a variable, select a formula, either on the left or right side, that has an operator ($\neg$, $\vee$, $\wedge$ or $\Rightarrow$). There is an inference rule that allows you to introduce that formula. Determine what other sequents would need to be proved valid in order to use that inference rule. Add those sequents to your goal list. (There will be one or two new sequents.) Continue, selecting a sequent to prove.

For example, to prove that $(P \vdash Q \Rightarrow P)$ is valid, select formula $Q \Rightarrow P$ from the right-hand side. Rule ($\Rightarrow$ R) says to add new sequent $(Q, P \vdash P)$ to the goal list; if you prove that sequent, you can conclude $(P \vdash Q \Rightarrow P)$. Since $(Q, P \vdash P)$ is basic, you are done.

$$(\wedge L):\quad \frac{(A,\, B,\, \alpha \vdash \beta)}{(A \wedge B,\, \alpha \vdash \beta)} \qquad\qquad (\wedge R):\quad \frac{(\alpha \vdash A,\, \beta),\ (\alpha \vdash B, \beta)}{(\alpha \vdash A \wedge B,\, \beta)}$$

$$(\vee L):\quad \frac{(A,\, \alpha \vdash \beta),\ (B,\, \alpha \vdash \beta)}{(A \vee B,\, \alpha \vdash \beta)} \qquad\qquad (\vee R):\quad \frac{(\alpha \vdash A,\, B,\, \beta)}{(\alpha \vdash A \vee B,\, \beta)}$$

$$(\neg L):\quad \frac{(\alpha \vdash A,\, \beta)}{(\neg A,\, \alpha \vdash \beta)} \qquad\qquad (\neg R):\quad \frac{(A,\, \alpha \vdash \beta)}{(\alpha \vdash \neg A,\, \beta)}$$

$$(\Rightarrow L):\quad \frac{(\alpha \vdash A,\, \beta),\ (B,\, \alpha \vdash \beta)}{(A \Rightarrow B,\, \alpha \vdash \beta)} \qquad\qquad (\Rightarrow R):\quad \frac{(A,\, \alpha \vdash B, \beta)}{\alpha \vdash A \Rightarrow B,\, \beta}$$

Figure 24.4: **Rules of inference for propositional logic.** These rules work with sequents, allowing you to prove that the sequent below the line is valid, assuming you have already proved that all of the sequents above the line are valid. Greek letters α and β stand for arbitrary lists of formulas. You are free to reorder the lists so that the formula that you are interested in is first.

There are two rules for each operator ($\wedge$, $\vee$, $\neg$ and $\Rightarrow$). One is used to introduce that operator into a formula on the left-hand side of a sequent, and the other introduces the operator into a formula on the right-hand side. The names of the rules express what they introduce. For example, rule ($\wedge L$) adds a new $\wedge$ operator into a formula on the left-hand side of a sequent.

24.10. Bibliographic notes

Thompson [103] and Bird [16] are introductory texts in functional programming using Haskell. Hudak, Hughs, Jones and Wadler [60] give a history of Haskell. You can find information on Haskell, including information on getting implementations, tutorials, and the Revised Haskell 98 Report at http://www.haskell.org/.

Part VIII

Handling and Exploiting Failure

Chapter 25

Handling Failure: Exceptions

25.1. Failure

What should a program do when it attempts to divide by 0? An obvious (but naive) answer is that the program should be stopped immediately, since it has clearly done something wrong. But that is a heavy-handed way of dealing with the problem. The person using the program might not be the one who wrote it, and might have no interest in hearing about what caused the error. It is true that something is wrong, but the program should really be given some opportunity to recover from the error. Software that suddenly stops running without warning tends to be unpopular.

One approach to allowing recovery is to design each subprogram or operation that might encounter trouble so that it returns a special value, or sets a flag, indicating an error. Division by 0 is an example. If IEEE floating point arithmetic is used, division by 0 yields a special value NaN (*Not a Number*), and the program can test the outcome of an operation to see whether the result is NaN. Unexpected situations can occur in a large variety of subprograms, and status values or flags are common. Reading information from a file, for example, can fail for a variety of reasons.

A carefully written program using status values must make frequent checks to see that no errors have occurred, and a significant portion of the program has to be devoted to checking and dealing with status values. Programs can be cluttered with this extra code that is rarely executed, and whose existence is really peripheral to the logic of the program. The main flow of the program is obscured by it, and the program is difficult to read.

But not all programs are carefully written. Programmers get tired of all that checking, and tend to become lazy and to skip some tests that should really be done, leading to serious failures due to errors that a program could have avoided. In security-critical applications, such as network services, an uncaught error can result in a breach of security.

Some programming languages are equipped with a feature, called *exception handling*, that alleviates both the problem of a program becoming cluttered with rarely executed error-checking code and the problem of programmers failing to check for error conditions. With exception handling, code that is concerned with rarely occurring errors can be removed from the main part of a subprogram definition, making the logic of the program more apparent. Also, a program can be designed to recover from many kinds of errors, even errors that are made in untrustworthy and poorly written modules that contain no error checks in them.

25.2. Exception handling

Exception handling is provided by two related features. The first feature is the ability to cause the program to fail. When a program fails, it is said to cause, or to *raise* (or sometimes to *throw*), an exception. The second feature is the ability to catch, or to *handle*, an exception when one occurs. We begin by looking at how to catch an exception. The language used is Cinnameg. Other languages provide similar mechanisms, and some are explored later in this chapter.

Catching exceptions

In Cinnameg, you can create a **Try**-expression that catches, or handles, exceptions generated within it. A try-expression or statement is similar in form to an if-expression or statement, and has the form

```
Try A
  then B
  else C
%Try
```

It starts by evaluating A. If that evaluation is successful, then expression B is evaluated, and its value is the value of the try-expression. But if an exception is raised while evaluating A, then expression C is evaluated instead, and its value is the value of the try-expression. Expression C is the *exception handler*, which decides what to do to recover from an exception. (Either or both of phrases `then` B and `else` C can be omitted. Both default to "do nothing.")

Identifying exceptions

When status values are used instead of exceptions, there must be a separate test for each use of a function that returns a status value, and for each error that might occur. But with exception handling, an entire paragraph of code, possibly using several different subprograms that might raise exceptions, can be covered by a single exception handler at the end of the paragraph.

But in order to handle many possible errors, an exception handler usually needs to know what kind of error occurred. When a program fails, it raises some particular exception. For example, in Cinnameg, attempting to read an integer where there is no integer in the input raises exception *conversionX*. An array bounds error, where the program tries to get the value at a nonexistent index in an array, raises exception *subscriptX*. Division by 0 raises an exception of kind domainX, indicating that a value was passed to a function that the function does not allow, and having an associated string indicating the particulars. Specifically, dividing by 0 raises exception domainX("divide by zero").

An exception is a value, and can be examined like other values. In the program, the word **exception** stands for the exception that occurred. For example, Figure 25.1 shows a program fragment that checks for conversionX or domainX.

Sometimes, you arrange for an exception handler to resume computation, possibly after doing something to recover from the error. If you do not know how to recover from the failure, an exception handler can *propagate* the exception by failing again, letting some other exception handler deal with it.

```
Try
    section whose failure will be caught
  else
    Choose matching exception
      case conversionX => handle conversionX
      case domainX(s)  => handle domainX
      else             => handle other exceptions
    %Choose
%Try
```

Figure 25.1: **A Cinnameg fragment that catches an exception and tests the exception.** Reserved word **exception** stands for the exception that was raised when and if computation failed. A Choose statement lists cases; in this case the first one whose pattern matches the exception will be chosen.

Example

Imagine that you would like to compute and print the inner product of two vectors, where the inner product of $[a, b, c]$ and $[d, e, f]$ is $ad + be + cf$. You are only allowed to compute the inner product of two vectors that have the same length; otherwise, the inner product is undefined. Figure 25.2 shows a function that prints the inner product of two vectors, or prints a message that the inner product is undefined. The exception handling code is at the end, and does not clutter the main logic.

25.3. Raising exceptions

Causing an exception is called *raising* the exception, or sometimes *throwing* the exception (since it is *caught* by the exception handler). Sometimes, a computation is simply said to *fail.* In Cinnameg, you can raise exception e using expression fail(e). At a failure, normal processing stops, and the system begins searching for an exception handler.

Another way to cause failure using Cinnameg is to use an *assertion.* If E is a condition (an expression of type Boolean) and x is an exception, then `MakeSure` E `else` x `%MakeSure` does nothing if E is true, and fails with exception x if E is false. You use it to prevent computation from continuing when a condition is false. For example,

```
MakeSure lst =/= [] else emptyListX.
```

says only to continue this line of computation when lst is not empty. If you omit the **else** part, the exception is assumed to be standard exception testX. Assertions will come in useful in the next chapter, on backtracking.

25.4. Exception handling in C++

In C++ you can raise an exception using a **throw** statement, of the form

`throw` e;

```
Define
  case ip([],[])       = []
  case ip(x::xs, y::ys) = (x*y) :: ip(xs,ys)
  else ip(?)           = fail(sizeX)
%Define

Define PrintInnerProduct(v1,v2). =
  Try
    Let inner = ip(v1,v2).
    Writeln[$(v1), ".", $(v2), " = ", $(inner)].
  else
    Writeln["Cannot compute the inner product of ",
             $(v1), " and ", $(v2)].
  %Try
%Define
```

Figure 25.2: **Catching an exception in Cinnameg.** PrintInnerProduct prints the inner product of vectors v1 and v2, handling the case where they have different lengths by printing an error message. Notice that the error handling code is out of the way of the main logic. Function ip is written in a functional style, using recursion. There is no need for an initial check of the vector lengths. Instead, a length difference is detected deep inside the recursion of ip. Expression `fail sizeX` causes a failure with exception sizeX. Function $ converts a value to a string, for printing.

where *e* is the exception. The exception can be any value, of any type. You are encouraged to use classes as exception types, but to avoid that level of complexity, we will use simpler values. For example, statement `throw 2;` throws a value of type int, and `throw "whoa, this is bad!";` throws an exception that is a string.

C++ has an exception handling mechanism that is similar in spirit to that of Cinnameg. You create a try block, followed by exception handlers. In C++, you must test the exception by checking its type. Statement

```
try {
  P(x);
}
catch(int k) {
  // comment: do nothing
}
catch(string s) {
  // comment: print a message
  cout << "Exception " << s << " happened\n";
}
```

performs statement P(x). If it succeeds, then the try statement is finished. If P(x) throws an exception, then the exception handlers are tried, in the order written. The first one that has a type that matches the type of the exception is performed. If none of the exception handlers has a correct type, then another exception handler (in another try) is searched for.

```
(define (zipit x y)
  (lambda (exit)
    (if (null?  x)
      (if (null?  y) () (exit 'exception))
      (cons (list (car x) (car y))
            ((zipit (cdr x) (cdr y)) exit)
      )
    )
  )
)
(define (zip x y)
  (call-with-current-continuation (zipit x y))
)
```

Figure 25.3: **Implementation of zip in Scheme, using a continuation to implement exception handling.** Expression (zipit x y) yields a function that takes parameter exit, the continuation. In the initial call, the exit parameter is provided by call-with-current-continuation. In the recursive calls, zipit must pass the exit function explicitly.

25.5. Exception handling in Scheme

This section requires knowledge of Scheme. Scheme does not provide exception handling as a separate mechanism, but instead offers a versatile feature that can be used for exception handling and much more.

Scheme takes an unusual approach to exception handling. It builds a pseudo-function called a *continuation*, which, when run, abandons the current computation and resumes computation at a prior place. Normally, when a function is called, a frame for it is pushed onto the run-time stack. But a continuation remembers the entire run-time stack as it was at a prior time. When the continuation is called, the current run-time stack is thrown away and the old one is brought back.

Scheme provides that feature through a built-in function called *call-with-current-continuation*. Evaluation of expression (call-with-current-continuation f) builds another function (let's call it *exit*) and then evaluates expression (f exit). Evaluating (exit x) will cause evaluation of (call-with-current-continuation f) to stop what it is doing and return x. If (f exit) returns without calling exit, then evaluation of (call-with-current-continuation f) returns the same result as (f exit).

This is not a simple concept by any means, but an example should help to clarify it. Imagine that you want to define function zip, where evaluating (zip '(a b c) '(d e f)) yields list ((a d) (b e) (c f)). But, if the two lists to be zipped have different lengths, you want zip to return symbol **undefined**. Figure 25.3 shows how to use call-with-current-continuation to do that.

Scheme allows continuations (such as function exit in Figure 25.3) to be kept as long as desired and used in any way desired. It is even possible to use the exit function after call-with-current-continuation has returned successfully. That is, you can change your mind later, and back up to where you used to be. What happens is that computation picks up again at the point indicated by the continuation, backing up to a prior spot and pretending that the call-with-current-continuation evaluation had failed. The next chapter, on backtracking, explores the idea of keeping continuations around after the expressions that they cover have already returned successfully.

Unfortunately, Scheme's exception handling mechanism has the weakness that it requires you to adopt one of two unpleasant approaches. In the first approach, when a function encounters an exceptional circumstance, it returns some special value, as zip does. But that takes you back to status values, with the requirement of constant checking. The second approach is to determine the correct behavior of a function at the point where the exception occurs, and just produce the answer there. For example, if you decide to make (zip x y) return an empty list when lists x and y have different lengths, then it suffices to use (exit ( )) instead of (exit 'undefined). Unfortunately, you typically do not know what to produce. For example, if you are writing a library for general purpose use, you probably do not know what some application that uses the library needs to do to recover from a failure that occurs inside one of the library functions.

25.6. When to use exceptions

Exception handling provides the possibility of an alternative control mechanism to the usual ones such as choices, loops and recursion, and it is possible to write a program that exploits exception handling as the normal mode of computing. For example, you might design a program that normally expects to divide by 0, and relies on an exception handler to recover from that apparent error. Or a program can read a list of numbers from a file until an attempt to read a number fails. You recover from the failure, recognizing it as the end of the file. That is not an error, of course, since you normally expect to see the end of the file eventually.

One school of thought is that this is an acceptable way to envision computation and to write programs. That is how Cinnameg handles multiple equations in a functional program. The compiler translates multiple equations into a program that tries the first equation inside a try expression. If that equation fails (for example, because its pattern match fails), the program catches that failure and tries the next equation, and so on. The idea is that an exception does not really indicate an error, but suggests that a particular way of solving a problem does not work, and some other way should be tried instead. If used carefully and clearly, that can lead to simple and elegant programs, and it is heavily used in logic programming.

On the other hand, overuse of exception handling, or use in awkward ways, can create programs that are truly difficult to understand. There is a second school of thought that exception handling should only be used to recover from situations that are not expected to happen during normal computation. That viewpoint is exemplified by Java, where you are discouraged from throwing exceptions unless something is really wrong.[1] There are a few reasons for that.

First, the very idea of an exception is of something that is unusual, and that does not normally happen. Accordingly, you might argue that an exception should only occur under truly exceptional circumstances.

Second, exception handling can be an unusually difficult control mechanism to understand. When an exception occurs, functions begin to be terminated abnormally, and

[1] Exception handling in Java is covered in Chapter 44.

control immediately transfers to a position that could be remote from the place where the exception was raised. Using exceptions as the usual control mechanism has the potential to yield programs that are, you might say, exceptionally difficult to understand.

Finally, and least importantly, exception handling typically takes much more time than tests using if-statements. An efficiently written program would probably like to avoid overuse of exception handling.

Both schools of thought have their adherents. Whichever school of thought you find more compelling, be cautious with exception handling. Use it only when you genuinely understand why you are using it and what it is doing.

25.7. Summary of terminology and concepts

Exception handling is a control mechanism that allows a program to recover from unusual circumstances or errors. Using exception handling, parts of a program that are responsible for error recovery can be removed from the normal flow of control, making the normal processing of the program easier to understand. Exception handling also allows a programmer to worry about erroneous circumstances less often, and for a program to recover from errors that were not thought about during the programming process.

Terminology

Catching an exception A program catches an exception when it resumes normal computing at an exception handler.

Exception An exception is an error, or exceptional condition, that must be dealt with before a program can continue with normal processing. An exception might indicate an error in the program or an error by the program's user, but sometimes an exception indicates a failure to proceed in a particular direction, and suggests that another direction be tried. An exception can be a data item that indicates the reason why a failure occurred, and which a program can inspect.

Exception handler An exception handler is run when an exception is caught. It can recover from the error and return the program to normal computation.

Failure A program fails when it cannot continue for some reason. Typically, it has encountered unusual, or erroneous, conditions.

Raising an exception A program raises an exception when it fails. The exception indicates why it failed.

Throwing an exception Throwing an exception is another term for raising an exception.

25.8. Exercises

25.1. Explain how exception handling can make it possible to run an unreliable function (that might cause errors) without the worry that those errors will cause the entire program to stop running. Assume that the function does not run forever.

25.2. Java allows a function to produce a special **null** value, indicating nothing, or no result. Instead of adding exception handling, why doesn't Java just let you use a null result to indicate an error? Does exception handling have any advantages over that?

25.3. Write a Cinnameg function that tries to read an integer from the standard input and to return that integer, but that produces answer -1 if no integer could be found. Library function read( ) returns an Integer. It will read an integer from the standard input and return it, but will fail with exception conversionX if there is no integer to read. (The type of the result of read( ) is determined from context. You will want to supply a tag, saying (read( ): Integer).

25.4. Write a Cinnameg subprogram that reads several integers from the standard input. It should continue reading until there are no more integers, and return a list of all of the integers that were read.

25.5. Write a Cinnameg statement that binds x to $f(y)$ provided $f(y)$ succeeds, and binds x to 0 otherwise. Be careful about scoping. The binding of x should be visible after your expression.

25.6. The map function, described in Chapter 21, is defined so that map f [a, b, c] = $[f(a), f(b), f(c)]$. Write a version of map called mapIgnoreFail that can handle cases where evaluation of f fails, putting only results that succeed in the result list. For example, if $f(a)$ and $f(c)$ succeed but $f(b)$ fails, then mapIgnoreFail f [a, b, c] = $[f(a), f(c)]$.

25.7. Common Lisp has an exception handling mechanism that is roughly as follows. Expression (catch X E) sets up an exception handler that can handle exception X, and evaluates E with this exception handler in place. Expression (throw X V) is evaluated by searching for an active catch expression that will catch exception X, and causing it to return V as its value. Does this mechanism have the same difficulties as the Scheme mechanism, or can you simulate a typical try block with it?

25.9. Bibliographic notes

Goodenough [42] introduces some ideas on exception handling. For exception handling in Scheme, see the Scheme report [89].

Chapter 26

Backtracking

26.1. The concept of backtracking

Exception handling offers one approach to dealing with failure that is typically used for kinds of failure that are not expected to occur during the normal course of computation, but instead indicate errors from which the program must recover. There is another approach to handling failure, called *backtracking*, that works a little differently from exception handling, and that is expected to be used as a control mechanism during normal computation rather than only for handling extraordinary circumstances. This chapter explores backtracking using Cinnameg as the vehicle. Cinnameg expression

```
Backtrack A else B %Backtrack
```

is evaluated by first computing expression A. If evaluation of A fails, then B is evaluated, and its value is used instead.

But suppose that evaluation of A succeeds. Then the value of the backtrack expression is *tentatively* taken to be A, and computation continues. But if the continuation of this computation, beyond the backtrack expression, fails, then it is as if A has failed, and the backtrack expression catches the failure and instead computes B as its value. For example, the program of Figure 26.1 prints 2. The first value $x = 1$ causes the MakeSure statement to fail, raising exception testX. That causes computation to back up to the Backtrack expression and to try making $x = 2$ instead, a choice of that makes the assertion successful. Notice that the failure occurs after finishing the Backtrack expression.

```
Let x = Backtrack 1 else 2 %Backtrack.
MakeSure x > 1.
Writeln[$x].
```

Figure 26.1: **A program fragment that employs backtracking.** This fragment prints 2, since the choice of $x = 1$ cannot make it past the assertion that $x > 1$.

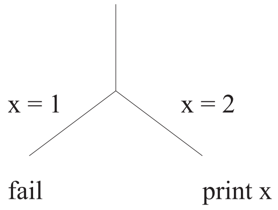


Figure 26.2: **A diagram showing computation of the program of Figure 26.1.** The fork is a branch point where the program can try either direction. Since the left hand branch leads to failure, the right-hand branch is tried instead.

```
Let article = Backtrack "a" else "the" %Backtrack.
Let noun = Backtrack "cat" else "mouse" %Backtrack.
Writeln[article, " ", noun].
(fail textX)
```

Figure 26.3: **A slightly more involved example of backtracking.** Figure 26.4 shows the branches that this computation takes.

Branching computations

Imagine that you are exploring a maze. When you enter the maze, you tie a string to a tree, and as you walk you unroll the string behind you. In your pocket you carry a supply of brightly colored pebbles. For simplicity, suppose the maze has no cycles, so you cannot walk forward and come back to where you were.

Suppose you come to a fork in the maze. Since you don't know which way to go, you choose a passage to explore, placing a pebble on the maze floor to show that you have tried that way. Eventually, you might hit a dead end. When you do, you begin backing up, rolling up the string, until you return to the fork. Then you proceed down another passage, also marking it by placing a pebble on the floor.

You might reach several forks and have to make a choice at each one before hitting a dead end. After backing up several times, you can back up to a fork where the pebbles tell you that you have already tried all of the passages. If that happens, you continue backing up, rolling up the string, and looking for a passage that you have not yet tried.

Backtracking works similarly to exploring a maze. A Backtrack expression represents a fork in the maze. As the program runs, imagine it unrolling a string behind it, keeping track of where it has been. At a failure, the program backs up along the string, undoing its previous actions, until it gets to the fork. Then it tries the other branch. Figure 26.2 illustrate computation of the program in Figure 26.1 as a fork.

A Backtrack expression is a branch point for the needle, or a fork in the maze. Figure 26.3 shows another example whose computation is diagrammed Figure 26.4. It prints four lines,

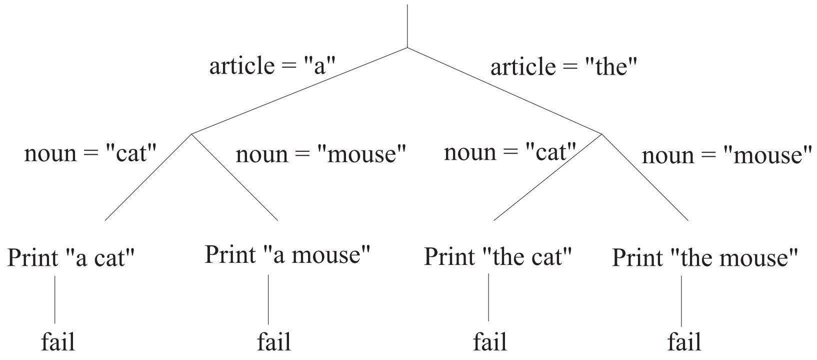


Figure 26.4: **Computation of the program in Figure 26.3.**

```
a cat
a mouse
the cat
the mouse
```

and then fails, backing up to look for prior branch points to explore.

The difference between Try and Backtrack

A try-expression `Try` *A* `then` *B* `else` *C* `%Try` is used to catch a failure that happens within a limited part *A* of the execution of the program. Once that part has succeeded, the try is no longer needed. The try commits to the decision of success. But a backtrack-expression `Backtrack` *A* `else` *B* `%Backtrack` does not commit to its decisions so readily. If *A* succeeds, the backtrack expression still considers it a possibility that *B* should be tried later, due to a failure that occurs *after A* in the computation.

26.2. Variables and branching computations

Suppose a computation forks into two branches. In the Chapter 27, we will see that backtracking involves making a copy of the run-time stack, which is where local variables are stored. Since each branch works with its own copy of the run-time stack, anything that one branch of the computation does to the run-time stack, such as changing the binding of a local variable, does not affect any other branch.

But there can be other variables, called boxes in Cinnameg, that live longer than a single subprogram execution. A box is an object that can hold one value. Cinnameg lets you create, initialize and name a box b using statement `Var` b `=!` i `%Var`; and to replace the value in box b by v using statement `Make @`b `=!` v `%Make`. Expression @b stands for the value currently stored in box b.

For backtracking, boxes present a problem: if one branch changes the content of a box, then it might interfere with what another branch is doing. For example, program fragment

```
Var{--nonshared--} z =! "ben".
Backtrack
  Make @z =! @z ++ "d".
else
  Make @z =! @z ++ "t".
%Backtrack
Writeln[@z].
(fail testX)
```

Figure 26.5: **An Cinnameg program using a nonshared box.** The Var line creates a nonshared box called z, and initializes it to hold string “ben”. Cinnameg does not automatically get the content of a box, so you need to do that explicitly. Expression @z produces the current content of box z. This program prints bend and bent on separate lines. The two branches do not interfere with one another.

```
Try
  Var b =! 10.
  Let x = Backtrack 1 else 2 %Backtrack.
  Make @b =! @b + 1.
  Displayln( (x, @b) ).
  MakeSure false.
%Try
```

prints (1,11) and (2,12) since the first branch changes what is in box b. The second branch's failure is caught by the Try.

There are times when sharing boxes is desirable, since, sometimes, one branch needs to convey information to another. (Imagine using a counter to count how many different branches the program explores.) But most of the time, you would prefer for the second branch to behave exactly as it would have behaved if the first branch had not been run at all. We really need two different kinds of boxes: *shared* boxes, which are shared by different branches, and allow one branch to affect what another does; and *nonshared* boxes, which behave as if the other branches did not run at all. Cinnameg provides both kinds of box. By default, a box created by Var is shared. But if you create a box n via `Var{--nonshared--} n %Var`, then you get a box that is private to each branch. Any change made to the content of this box in one branch is not seen by parallel branches. Figure 26.5 shows a program that uses nonshared boxes. It prints

bend
bent

and then fails. The first branch adds 'd' to the end of the string in box z, yielding “bend”. But when computation backtracks, it undoes that binding, putting “ben” back into z, so that the second branch adds the 't', yielding “bent”.

26.3. A backtracking example

Backtracking is useful for solving problems that involve searching or planning. An example is the problem of a farmer who owns a wolf, a goat and a cabbage. The farmer needs to

```
Define farmer = ’f’; wolf = ’w’; goat = ’g’; cabbage = ’c’.
Define somethingIsEaten(bank) =
  not(farmer ‘in‘ bank) and
  (goat ‘in‘ bank) and
  (cabbage ‘in‘ bank or wolf ‘in‘ bank)
%Define
Define admissible (lft,rgt) =
  not(somethingIsEaten(lft)) and not(somethingIsEaten(rgt))
%Define
```

Figure 26.6: **Implementation of** ***admissible*****.** Expression admissible(s) yields true just when s is an admissible state for the farmer-wolf-goat-cabbage problem. The parameter of admissible is an ordered pair. Helper function somethingIsEaten checks one bank.

get across a river in a boat, taking his possessions with him. The boat only has room for the farmer and one of his possessions. The animals will not run away, so it is acceptable to leave them alone. The difficulty is that, if the goat and the cabbage are left alone, the goat will eat the cabbage; and if the wolf and the goat are left alone, the wolf will eat the goat. How can the farmer cross the river without any of his possessions being eaten?

We will design a program that finds a solution based on a *generate-and-test* approach, where solutions are generated and explored by separate branches of the computation. Each branch checks itself to make sure that it is not exploring a disallowed solution (where, for example, some possession is being eaten). If a branch sees that something is wrong, it simply fails. That causes the program to backtrack and to try a different, possibly more promising, branch. Only those branches that are exploring acceptable solutions will make it to the end.

Sets

Cinnameg supports sets, which we will use for showing what or who is on each bank. Explicit sets are enclosed in braces, such as {1,2,3}. Expression x ‘in‘ s is true if x is a member of set s;[1] $s + /x$ computes $s \cup \{x\}$ and $s - /x$ computes $s - \{x\}$,

States

A *state* tells who and what are on each bank of the river. For a state we will use a pair of sets (L,R), where set L tells what is on the left bank and set R tells what is on the right bank. For simplicity, we use characters ’f’ for the farmer, ’w’ for the wolf, ’g’ for the goat and ’c’ for the cabbage. The boat is always where the farmer is, so there is no need to show it. For example, ({’f’,’g’}, {’w’,’c’}) shows the farmer and goat on the left bank and the wolf and cabbage on the right bank.

Say that a state is *admissible* if nothing is being eaten. Function admissible, shown in Figure 26.6, returns true on an admissible state, false on an inadmissible state.

[1] Binary operators that are words are surrounded by left-single quote marks in Cinnameg.

```
Define next(lft,rgt) = st |
  open If farmer ‘in‘ lft then
    Let possession = each lft.  %% possession can be the farmer
    Let st = (lft -/ farmer -/ possession, rgt +/ farmer +/ possession).
  else
    Let possession = each rgt.
    Let st = (lft +/ farmer +/ possession, rgt -/ farmer -/ possession).
  %If
%Define
```

Figure 26.7: **Implementation of function** ***next*****.** Expression next(s) returns a state that might follow state s in a solution of the farmer-wolf-goat-cabbage problem. It uses backtracking to produce a branch for each possible next state. Library function *each* takes a list as a parameter and backtracks over the members of that list, one branch returning each member. The farmer takes nothing when possession is selected to be the farmer. The word **open** allows the definition of st to be in scope outside the If statement.

Finding the next state to try

There are up to four things that the farmer might conceivably do from a state; he might row across the river with any one of his three possessions, or he might row across alone. It is not our concern yet whether any of those actions leads to a solution, or even to an admissible state; that will be tested elsewhere. Right now, we only want to know what options the farmer has. Let next(s) produce, by backtracking, all possible states that might follow state s. (It produces only one of them in each branch, and can produce several branches.) For example, if state s is ({'f','w','g','c'},{ }), then computation of

```
Let t = next(s).
```

yields four branches

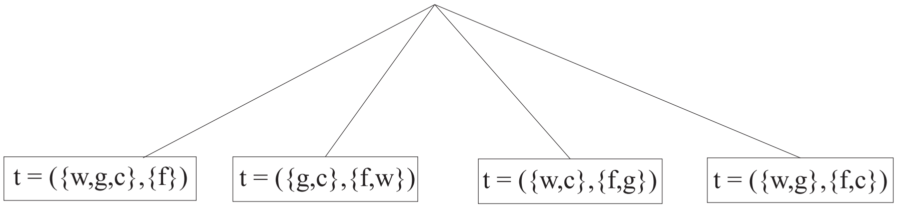


Figure 26.7 shows an implementation of the *next* function.

Building up a solution

A solution to the farmer-wolf-goat-cabbage problem can be described by a list of states, starting with the initial state (all on the left bank) and giving, in sequence, the states that occur as the farmer rows back and forth across the river. The list should end on a state where all are on the right bank. Getting from one state in the list to the next one should involve the farmer crossing the river once.

```
Define addState(soln) = newState :: soln |
  Let newState = next(head(soln)).
  MakeSure admissible(newState).
  MakeSure not (newState ‘in‘ soln).  %% no loop
%Define
```

Figure 26.8: **Implementation of function** ***addState*****.** Operator ‘in‘ is overloaded, and here tests whether a state is a member of a list rather than a member of a set.

It is convenient to build the solution up in reverse order, so that the more recently added states are near front of the list. Just read the lists backwards. At each point during the search, you have a partial solution that starts (at the end of the list) with the initial state ({’f’,’w’,’g’,’c’},{ }), and *might* be a good beginning to the full solution. The initial partial solution is list [({’f’,’w’,’g’,’c’}, { })], a list holding only the initial state. New states are added to the front of the list to explore slightly longer partial solutions.

To generate a solution, you start with an initial solution and extend it, ensuring after each extension that (1) the state just reached is admissible, and (2) the state just reached has not been reached before (to prevent looping). Function addState, shown in Figure 26.8, adds a state to a sequence of states, checking for consistency, and returning the slightly longer sequence of states. If the new state should not be tried, then addState fails.

Look at the structure of function addState. It is linear; there is no apparent looping or recursion. The body of addState just says to find the next state and move to it, as long as it is a good state. You are not concerned with the process of searching through all of the possible solutions, but only with describing what a correct solution looks like. Backtracking allows simple and elegant solutions to problems like this one. It allows you to linearize a computation that is, in reality, branching.

Finishing the solution

All that is needed to finish the farmer-wolf-goat-cabbage problem is a function that builds the entire solution. Function findsoln, shown in Figure 26.9, generates an extension of a given partial solution that reaches a given goal state, by adding as many states as necessary. In more detail, expression findsoln(partialPlan, goal) returns a list L such that head(L) = goal, partialPlan is a suffix of L, and L is a good plan in the sense that all of the states in L are admissible and all of the moves between consecutive states are allowed. (Function findsoln does not check whether partialPlan itself is good. It presumes that test was done earlier.) Now

```
Execute
  Let everything = {farmer, wolf, goat, cabbage}.
  Let soln = findsoln([(everything, { })], ({ }, everything)).
  Writeln["Solution:  ", $(reverse soln)].
%Execute
```

finds a solution and writes it (in reverse order, since it was built up backwards.) In Cinnameg, an Execute block keeps running until all of its control threads are done, so it will print all solutions.

```
  Define first
    case findsoln(goal::rest, goal) = goal::rest
    case findsoln(rest,goal)        = findsoln(addState(rest), goal)
  %Define
```

Figure 26.9: **Implementation of function findsoln.** Expression findsoln(rest, goal) extends list rest by adding zero or more states to its beginning until the list begins with the goal state, and returns the final list, which is the plan. The first case applies when the list already begins with the goal. The second case uses addState to add one more state, ensuring that added states are admissible ones that are not already in the list.

26.4. Pattern matching in lists

One use of backtracking is in searching for patterns in lists (such as strings). Some languages, such as Icon, are based on pattern matching in strings. For example, suppose that you would like to check whether a string called *text* contains any of the substrings “the cat”, “the mouse”, “a cat” or “a mouse”. Cinnameg pattern a ‘bor‘ b uses backtracking to try matching a, and if that fails, to match b. Using the string concatenation operator ++ in a pattern, a pattern match can be formulated that will succeed if *text* contains any of the four listed strings.

```
Match
  ?  ++
  ("the " ‘bor‘ "a ") ++
  ("cat" ‘bor‘ "mouse") ++ ?  = text
%Match
```

Matching pattern a ++ b against a target list L tries all possible ways of breaking up list L into two lists. For example, matching pattern a ++ b against list $[1, 2, 3]$ creates a computation with four branches. The first binds a to $[\,]$ and b to $[1, 2, 3]$, and the second binds a to $[1]$ and b to $[2, 3]$. All four possible values of a and b such that a ++ $b = [1,\ 2, 3]$ are tried.

The list membership testing function can be expressed as a pattern match. The idea is to try to do a match and to see whether it succeeds. We use Cinnameg expression Succeeds S %Succeeds, which yields true if statement S succeeds and false if it fails.

```
Define member(m,x) =
  Succeeds
    Match ? ++ [m] ++ ? = x.
  %Succeeds
%Define
```

Suppose that, instead of testing whether m is a member of list x, you want to get a member of x, as function *each* does. Pattern matching can be used for that as well, yielding a definition of each whose structure is almost identitical to the pattern match in the membership testing function.

```
Define each(x) = m |
  Match ? ++ [m] ++ ? = x.
%Define
```

```
Try
  Let x = each[1, ..., 50].
  Let y = each[1, ..., 50].
  Let z = each[1, ..., 50].
  MakeSure x*x + y*y == z*z.
  Displayln(x,y,z).
  MakeSure false.
%Try
```

Figure 26.10: **A program employing backtracking to find all Pythagorean triples** (x, y, z) **where** x**,** y **and** z **are all between 1 and 50.** Notice the apparent straight-line structure of the program.

The similarity exemplified here between (1) testing whether a given value has a given property and (2) finding something that has a given property (such as membership in a list) is exploited heavily in logic programming, the topic of the Part IX.

26.5. Using recursion to simulate backtracking

Relatively few programming languages provide direct support for backtracking. But backtracking can be simulated using more common mechanisms such as loops and recursion. To illustrate, suppose that you want to find all *Pythagorean triples* (x,y,z) where x, y and z are integers between 1 and 50. Triple (x, y, z) is a Pythagorean triple if $x^2 + y^2 = z^2$. A backtracking solution is shown in Figure 26.10.

One way to simulate the backtracking uses a separate function to choose each of x, y and z. Figure 26.11 shows an implementation of that idea. Function ChooseX chooses x and then calls the next function, ChooseY, for each x. The chain of calls eventually leads to the function that prints the answer. The same effect could by achieved by three nested loops.

Backtracking can always be simulated if it is not directly available in the programming language. But the simulation needs to take control of the searching process. Because of that, the natural structure of the computation is hidden behind the machinery that manages the search. Instead of just saying choose x, choose y, choose z and test, looping or recursion needs to be used to manage the choices. If a goal of a programming language is to make the logical structure of algorithms more evident and easy to follow, then backtracking can be seen as a success story, at least for some kinds of algorithms.

26.6. Backtracking and substitution

Allowing an expression to produce more than one result, in different branches, comes at a cost in mathematical elegance. One of the rules of mathematics (and of pure functional programming) is that, if $A = B$, you should be able to replace A by B or B by A without changing what is computed. But suppose you write program

```
Let x = Backtrack 1 else 2 %Backtrack.
Let y = x + x.
```

```
Define ChooseX. =
  For x from [1, ..., 50] do
    ChooseY(x).
  %For
%Define
```

```
Define ChooseY(x). =
  For y from [1, ..., 50] do
    ChooseZ(x,y).
  %For
%Define
```

```
Define ChooseZ(x,y). =
  For z from [1, ..., 50] do
    If x*x + y*y == z*z then
      Displayln(x,y,z).
    %If
  %For
%Define
```

Figure 26.11: **Another program that finds all Pythagorean triples** (x, y, z) **where** x**,** y **and** z **are all between 1 and 50.** This program simulates backtracking by using loops to make the choices and function calls to make the steps between choices. The linear structure is lost, but this does not require language support for backtracking.

That creates two branches, one where $y = 2$, the other where $y = 4$. But, replacing x with the expression that computes it yields

```
Let y = Backtrack 1 else 2 %Backtrack +
        Backtrack 1 else 2 %Backtrack.
```

which produces four branches, with values $y = 2$, $y = 3$, $y = 3$ (again) and $y = 4$. Logic programming, discussed in the next part, allows a language implementation to perform automatic backtracking, but restores the correctness of some elementary forms of mathematical reasoning.

26.7. Summary of terminology and concepts

Backtracking is a control structure that allows the language implementation automatically to control searching among several different branches of computation. When one branch fails, another is taken instead. Backtracking has the advantage that a computation that is in reality branching and trying different alternatives can be expressed in a program in a linear form, describing only the branch or branches that do not fail. That makes it possible to write very simple and elegant programs to solve some search problems.

Terminology

Backtracking Backtracking is a control structure that relies on setting up multiple computation paths. When one path fails, the program backs up and tries another branch instead.

Nonshared box Modifications of the content of a nonshared box on one branch of a computation do not affect what other branches see as the content of that box. Each branch has a private copy of the box. That is important to prevent different branches of a computation from interfering with one another.

Shared box When a shared box has its contents changed on one branch of a branching computation, other branches can see the modification.

26.8. Exercises

26.1. Backtracking can be simulated using recursion. So why bother to provide backtracking at all? Explain why backtracking can be useful to a programmer.

26.2. If the language does not support nonshared boxes, can they be simulated? If so, how?

26.3. The n-queens problem asks how (or whether) n queens can be put on an $n \times n$ chessboard so that no queen is attacking any other queen. Two queens attack one another if they are in the same row, same column or same diagonal. Write a solution to the n-queens problem in Cinnameg using backtracking. Make your program produce all answers.

26.4. The knight's tour problem asks how a knight placed in one square of an $n \times n$ chessboard can move around the board, hitting each square exactly once and returning to the start square. Write a solution to the knight's tour problem in Cinnameg using backtracking.

26.5. The knight's tour problem described in the preceding exercise can be solved efficiently without backtracking. A square is available if the knight has not visited it yet. At each step, the knight moves to an available square that has the most moves to available squares after that. Implement this algorithm, and compare it to the backtracking algorithm for speed.

26.6. Three missionaries and three cannibals are on one bank of a river. They want to get across to the other bank. They have a boat that can hold up to two people at a time. To go across the river, the boat must have at least one person in it to paddle. The paddler can be a missionary or a cannibal. The difficulty is that, if at any time the cannibals outnumber the missionaries on either bank, they will eat the missionaries. (A missionary cannot hide in the boat. You are on the bank even if in the boat on that bank.) Write a program that finds all ways that all six people can cross the river without anyone being eaten, and that do not go through any state more than once.

26.7. The obvious backtracking algorithm is not always an efficient algorithm to solve a problem. A more efficient way to find pythagorean triples is to choose positive integers a, b and c, where $a > b$, and to let $x = c(a^2-b^2)$, $y = 2abc$ and $z = c(a^2+b^2)$. It is easy to verify that $x^2 + y^2 = z^2$, and it turns out that all pythagorean triples can be found by selecting suitable values for a, b and c, if you take into account that, when (x, y, z) is a pythagorean triple, then so is (y, x, z).

Define a function in Cinnameg that produces pythagorean triples (x, y, z) by selecting positive integers a and $b < a$, and defining $x = a^2 - b^2$, $y = 2ab$ and $z = a^2 + b^2$. Design the function to produce one triple at a time, but to produce a new triple each time it is backtracked into. (That will not produce all pythagorean triples, but it will produce infinitely many, and others can be found by scaling and flipping x and y.) You will want to loop over pairs (a, b) without getting stuck on one value of a while you try all (infinitely many) values for b. You might find the Cinnameg library function *dovetail* useful. How can you avoid producing the same triple more than once? You are not required to use backtracking explicitly, and might find recursion more a more convenient way to do the search.

26.8. Figure 26.11 simulates backtracking to choose three values. But suppose that, instead of choosing just three variables, you need to select of list of n values, where n is only known at run time. You cannot right n separate functions. Nor can you write n nested loops. Rewrite the pythagorean triple program, but instead of using a separate function for each value to be chosen, write just one that selects a list of three values. You might find it useful to add an extra parameter that tells which variable (1, 2 or 3) is to be chosen next.

26.9. Bibliographic notes

SNOBOL [46] is an early language that employed backtracking. Icon [47, 48] is a more recent language in the SNOBOL family. Logic programming languages, such as Prolog [31, 32], also employ backtracking to control computation. Logic programming languages are covered in the next part.

Chapter 27

Mechanics and Practical Issues of Handling Failure

27.1. Implementation of exception handling

When a subprogram is called, a frame for it is pushed onto the run-time stack. Similarly, when a try expression is entered, a *control frame* is pushed onto the run-time stack, containing information about where to resume when an exception is raised.

When the program raises an exception, an *exception manager* runs. It searching down the run-time stack for the topmost control frame, discarding any subprogram frames that it encounters (causing those subprograms to stop immediately). When a control frame is encountered, it is removed, and computation resumes with the exception handler that is indicated by the control frame.

For example, suppose that function f contains a Try expression. Inside the try, f calls g, which in turn calls h. The run-time stack is as follows.

frame for h
frame for g
control frame
frame for f

Now suppose h raises an exception. The result is that the frames for h and g are removed, as is the control frame, and f resumes at the exception handler for the try expression, whose location is remembered in the control frame.

A program can enter a try while it is inside another try. In that case, the run-time stack contains more than one control frame. When the program raises an exception, the topmost control frame, belonging to the most recently entered try, is the one that is found. Try expressions are nested *dynamically*, not statically. Suppose that function f enters a try, and, while processing the body of the try, calls function g. If g enters a try, then g's try is dynamically nested inside f's try.

Of course, computation normally does not fail, and all goes well. When the body of a Try expression succeeds, the control frame is removed without invoking the exception handler.

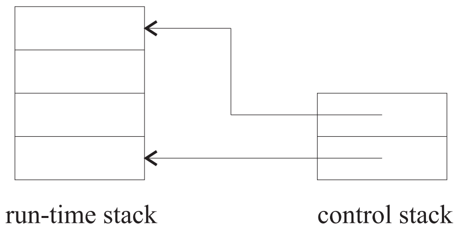


Figure 27.1: **Placing control frames in a separate control stack.** Notice that each frame in the control stack indicates where it belongs in the run-time stack.

27.2. Implementation of backtracking

Exception handling can be implemented by putting control frames in the run-time stack. But to deal with backtracking, we will need to make a modification to how control frames are handled. Instead of putting them in the run-time stack, we put them in a separate *control stack*. Each control frame points to the frame in the run-time stack that created it. The two stacks (the control stack and the run-time stack) are pictured in Figure 27.1. When a failure occurs, the top frame on the control stack tells where to continue execution.

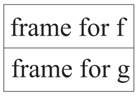
For simplicity, we will imagine a language that supports, such as Prolog, backtracking but not exception handling. Exercises ask you to explore what is required to provide both.

A Backtrack expression differs from a try in the following important way. When expression A succeeds in `Backtrack` A `else` B `%Backtrack`, the control frame is not removed from the control stack. Instead, it is kept there, ready to catch an exception when one is raised, and to resume by computing B.

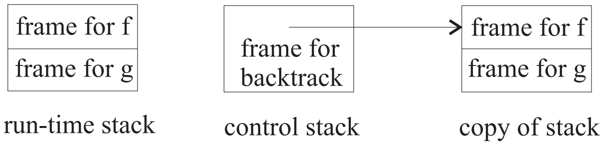
That leads to a serious implementation problem. The control frame points to a frame in the run-time stack. Since the control frame is not removed when the backtrack expression is exited, it can continue pointing to a frame even after the subprogram that owned that frame has returned. Because of that, it appears not to be possible to remove frames from the run-time stack. That can be avoided by making a control frame point to a copy of the run-time stack. For example, consider the following, written in Cinnameg.

```
Define f() = Backtrack 1 else 2 %Backtrack.
Define g() =
  Let x = f().
  Displayln(x).
  (fail testX)
%Define
```

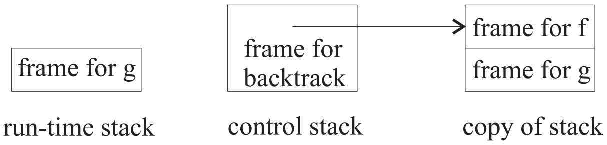
Imagine computing $g(\,)$. The first thing g does is evaluate $f(\,)$. When $f(\,)$ begins to run, the run-time stack looks as follows, with the frame for f on top of the frame for g.



When $f(\,)$ enters the backtrack expression, a control frame is pushed onto the control stack, and the situation changes to the following.



Notice that the entire run-time stack has been copied, and the control frame points to the copy. Now the backtrack construct finishes, and $f()$ returns with value 1. The situation looks like this.



Now $g()$ prints the value 1, and then fails. When a failure occurs, the top control frame on the control stack tells where to resume computation. In the example, the control frame indicates that computation should resume in the backtrack expression, at the point where it produces the alternative value 2. The entire run-time stack is thrown out, and replaced by the copy that is remembered with the control frame. That restores the run-time stack to the point where it was when the backtrack expression was started, and $f()$ can resume computing. This time, $f()$ returns 2.

Copying the entire run-time stack is expensive, and it can be avoided if some care is taken. All that is really needed is an implementation that works *as if* it copies the stack. The idea is to copy parts of the stack only when they genuinely need to be copied. One way to do that is to store the run-time stack as a linked list of frames, letting the dynamic link (Section 13.2) be the list link. Using reference counts (Section 10.5), you can ensure that a frame is not destroyed or modified prematurely. If you try to modify the information in a frame that has more than one reference to it, you need to copy just that frame first.

27.3. Implementation of nonshared boxes

A typical way to implement nonshared boxes is to store *trail* frames in the control stack. Each trail frame holds a box and its content, and any time a nonshared box has its content changed, that box and its old content are pushed on the control stack. When the program backtracks, it restores the old contents of the boxes found in the trail frames that are above the topmost control frame, making it appear that the box was not changed.

27.4. Pruning the search using commits

Introduction of a backtrack control frame leads to copying of at least part of the run-time stack. The program needs to remember every place that it has performed a branch, just in case there is a need later to back up. That can result in a vast amount of memory being used, and can be a serious problem for programs that employ backtracking.

Sometimes, a program gets to a point where it knows it will not backtrack, or it knows that any backtracking that it would do would not lead to any useful or desired computations. In that case, the program should cause the backtrack control frames to be removed, and the

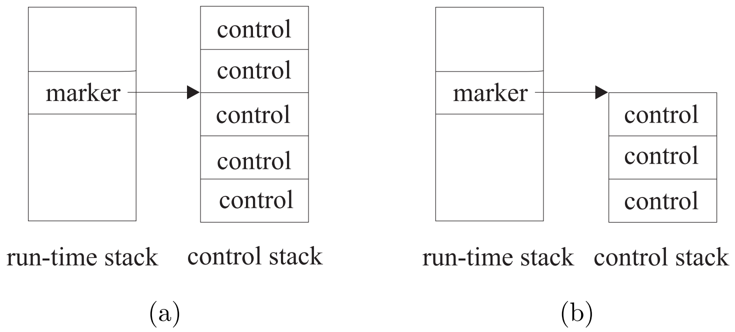


Figure 27.2: **The action of a commit.** The run-time stack before a commit (a) holds a marker frame pointing to a position in the control stack. The commit removes all backtrack frames above the position indicated by the topmost marker frame (b).

memory that they are using to be recovered. The operation that does that is called a *cut* or a *commit*. We will use the term *commit*, and describe how commits are done in Cinnameg.

There are two parts to the implementation of a commit. The first part concerns management of *marker frames*. Sometime before the commit is done, a marker frame is pushed onto the run-time stack, holding a pointer to the current position of the top of the control stack. In Cinnameg, CommitBarrier constructs control pushing and popping of marker frames. On entry to `CommitBarrier ... %CommitBarrier`, a marker frame is pushed, and on exit from that construct, the marker frame is removed. But keep in mind that creating a branch at a backtrack construct copies the run-time stack. Exiting `CommitBarrier ... %CommitBarrier` only removes the marker frame from the currently running copy of the run-time stack.

The next part is the commit itself. Statement `Commit.` tells the interpreter to find the top marker frame in the current run-time stack and to remove all backtrack control frames that are above the indicated position in the control stack. Figure 27.2 shows the action of a commit.

The farmer-wolf-goat-cabbage problem has two solutions. Having found the first one, suppose that you want to ignore the other one. Figure 27.3 shows a way of doing that by removing the backtrack control frames that would lead to finding the second solution.

27.5. Summary of terminology and concepts

Exception handling can be managed by adding control frames to the run-time stack. For backtracking, it is preferable to move the control frames into a separate control stack, and to make each control frame point to a private copy of the run-time stack.

Nonshared boxes can also be handled by adding frames to the control stack, but those frames tell how to undo changes when the program backs up.

Backtracking has the disadvantage that it prevents normal handling of the run-time stack, making the language implementation more complicated and less efficient, and possibly causing a program that employs backtracking to use a large amount of memory. Commits are a means of reducing the memory requirements by removing control frames.

```
Execute
  CommitBarrier
    Let everything = {farmer,wolf,goat,cabbage}.
    Let soln = findsoln([(everything, { })],
                        ({ }, everything)).
    Commit.
    Writeln["Solution:  ", $(reverse soln)].
  %CommitBarrier
%Execute
```

Figure 27.3: **A program that prints only the first solution that is found to the farmer-wolf-goat-cabbage problem.** It uses function findsoln from Figure 26.9. Once a solution is found, other solutions are ignored by removing the backtrack control frames that have been pushed onto the control stack while running findsoln.

Terminology

Control frame A control frame is put on the run-time stack to allow the language implementation to know how to handle an exception.

Control stack The control stack holds control frames that are used to find places to continue when a failure occurs. Each frame points (effectively) to a complete run-time stack.

Commit A commit removes frames from the control stack, thus destroying the computation branches to which those frames refer. Commits are used to reduce both the memory and time requirements of programs.

Trail frame A trail frame stores information about nonshared boxes that allows changes to boxes to be undone during backtracking.

27.6. Exercises

27.1. What is the key difference between the implementations of try control frames and backtrack control frames? Ignore where they are placed. Discuss the timing of adding and removing them.

27.2. In logic programming, variables are like nonshared boxes, but a variable can have no value (or be an empty box). Only variables without a current value can be changed; once a variable has a value in a given branch of the computation, the value is permanent for that branch. How does that simplify trail frames?

27.3. Imagine a language, such as Cinnameg, that allows both exception handling and backtracking. Derive a way to manage the two together. Explain how you intend to keep track of control information, how you find where to pick up after a failure, and how control frames are added and removed both for exception handling and for backtracking. Control frames can be tagged as being either for backtracking or for exception handling, and they can contain extra information.

27.4. When are commits most important in backtracking computations? If you view a branching computation as a tree, are commits more important for computations that are wide but shallow trees, or for computations that are narrow but deep trees?

27.5. Commits can save both memory and time.

(a) Explain how commits save memory.

(b) Explain how commits save time.

27.6. Can you see any danger in using commits? What surprising problems might emerge with their use?

Part IX

Logic Programming

Chapter 28

Trees and Unification

28.1. Trees and data

Logic programming is based on a variation of (first order) mathematical logic, and a good starting point is examination of the data structures and variables that underly it.

Logic programming starts with a *universe of discourse*, consisting of values whose properties and relationships you express in logical terms. The universe of discourse contains numbers, strings, as well as a few other kinds of data. One useful kind of constant is a *symbol*, which just stands for itself; it is a name. A typical logic programming convention, which we will adopt here, is that words that start with lower case letters are symbols, so symbols might include *a*, *alice*, *orion*, and so on. Often, a few special characters or sequences of characters, such as + and ::, are also treated as symbols.

Trees

Structured values in logic programming have the form of trees, where the nonleaf nodes are labeled by symbols, and the leaf nodes can be labeled by any kind of value supported by the language. Figure 28.1 shows examples of trees. Lists are considered to be special cases

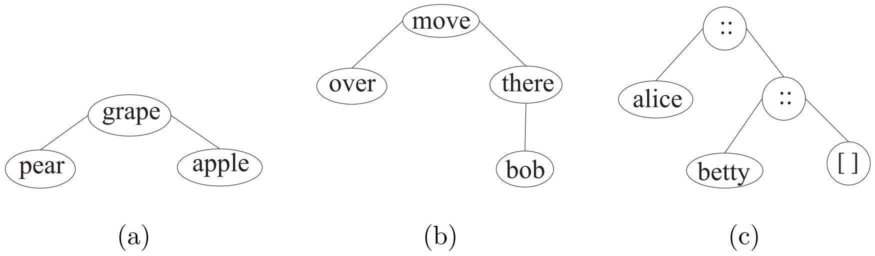
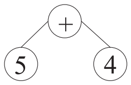


Figure 28.1: **Three trees that can be values in a logic program.** Tree (a) is written grape(pear,apple) and tree (b) is written move(over,there(bob)). Binary operator syntax, if supported by a language, allows you to write x::y instead of ::(x,y). So tree (c) is written alice::betty::[ ]. Syntax [alice, betty] is an abbreviation for alice::betty::[ ].

of trees whose internal nodes are labeled by a list construction symbol (which we will call :: here), as is shown in Figure 28.1(b). The end of a list is marked by another symbol, often simply called [ ], since it is really the empty list. (So the sequence of two characters "[ ]" is another special name for a symbol.)

Writing trees in programs

It is awkward to draw trees in programs, so you write them using an expression notation. Expression $r(a, b, c)$ stands for a tree whose root is labeled r, having three subtrees a, b, and c. The order of the subtrees is important; they are given from left to right. A leaf having label q is just written q, without showing any subtrees. When a node label is a binary operator, it is more convenient to use binary operator syntax. Expression $5 + 4$ stands for the tree



as if it were written $+(5, 4)$.

28.2. Unknowns

A logic program computes by finding values for unknown quantities. For example, to find the factorial 6!, you state that the unknown quantity X is equal to 6!, and let the program work from the facts that you have provided to determine the value of X. One of the basic concepts of logic programming is, therefore, a variable, or an *unknown*, representing some value that has not yet been determined.

An unknown starts life having no known value; it is truly unknown. At some point during computation information might be obtained that gives a value to the unknown. (So it becomes known, or *bound*.) An unknown differs from a variable in an imperative program in one key characteristic; once a value is given to the unknown in a given branch of a computation, that value cannot be changed in that same branch. The unknown is permanently bound. That makes logical sense; once you know the value, you need to stick with what you say, and cannot change your mind later.

We will adopt the convention of using a name that starts with an upper case letter for an unknown. For example, X and Who are unknowns. But be careful. An unknown can become bound shortly after it is created. It is still referred to by its name, such as X, but after becoming bound it is no longer unknown at all.

Unknowns in data structures

A complex value, such as a list, can have some parts that are unknown. For example, a list might have the form $[X, Y]$. It is clear that this is a list of two things, but the members are not known. Figure 28.2(a) shows $[X, Y]$ as a tree. In general, any subtree of a tree can be an unknown. For example, $H :: T$ is a list that must be nonempty, but whose head and tail are not known.

An unknown in a tree can stand for a simple value (a leaf in the tree), or for another tree. As unknowns become bound, more structure becomes known. For example, suppose the program knows that L is list $H :: T$. Later, as the program runs, T becomes bound to

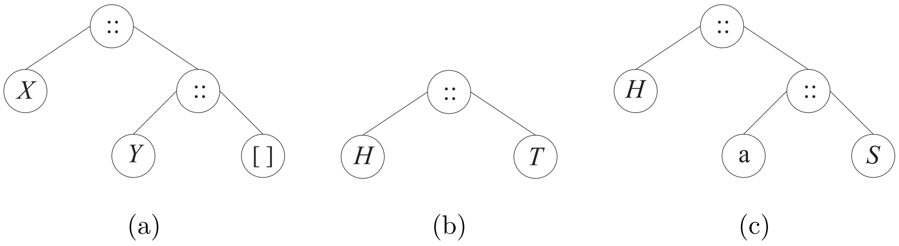


Figure 28.2: **Three trees containing unknowns.** Tree (a) is written $[X, Y]$, a list of two unknown things. Tree (b) is $H :: T$. Unknowns can only be leaves of trees, but they stand for unknown subtrees, not necessarily unknown leaves. If, later, unknown T becomes known to be $a :: S$, then tree (b) will be changed into tree (c), representing the extra knowledge about T.

$a :: S$, where a is a constant and S is another unknown. Then the program knows that L is $H :: a :: S$. The leaf that was occupied by unknown T has been replaced $a :: S$, as shown in Figure 28.2(c).

28.3. Unification: active pattern matching

Chapters 11 and 18 introduce pattern matching, a tool for solving simple equations. In equational programming, pattern matching solves for *names* that are part of the pattern, and binds those names to the solutions that are found. For example, you solve equation $[x, y] = [5, 7]$ by binding x to 5 and y to 7. Logic programming also requires a way of solving simple equations, but it employs a different form of pattern matching called *unification.* Instead of binding names, unification binds unknowns. For example, equation $H :: T = [a, b, c]$ is solved by creating the two data structures $H :: T$ and $[a, b, c]$, containing unknowns H and T, and then trying to bind unknowns to make those data structures identical. The solution binds H to a, and T to $[b, c]$. An unknown is bound by replacing it. For example, to bind H to a, you replace every occurrence of H by a.

There can be unknowns on both sides of an equation. For example, solving equation $[A, m] = r :: X$ yields bindings $A = r$ and $X = [m]$, as shown in Figure 28.3. That contrasts with pattern matching for equational programs, where variables being solved for can only occur on one side of the pattern match equation, and the other side must be known.

An unknown can occur more than once. When it is bound, all of its occurrences are replaced, so that it disappears. For example, imagine solving equation $[A, A] = [X, c]$. That will certainly require binding $A = c$. Doing so yields pattern match $[c, c] = [X, c]$; continuing to solve the equation, X is bound to c.

Unknowns can be bound to one another

Allowing unknowns to occur on both sides of the pattern match equation has an interesting consequence; unification can bind two unknowns to one another, without binding either of them to anything in particular. Equation $X = Y$ tells you that X and Y must have the same value, but it does not say what that value is. That is easy to handle; just replace all occurrences of one of the unknowns by the other one. For example, unknown X might

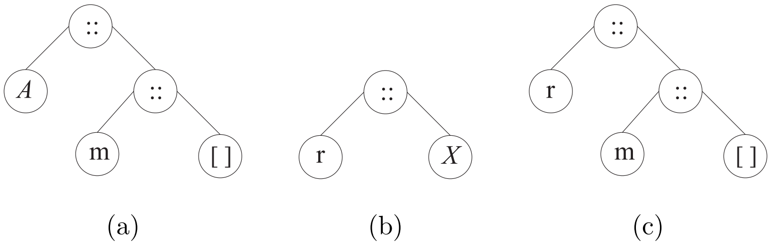


Figure 28.3: **An example of unification.** By convention, A and X are unknowns, since their names start with upper case letters, but r and m are symbols, just things called "r" and "m", respectively. Unification of tree (a) and tree (b) is achieved by binding $A = r$ and $X = [m] = m :: [\,]$. Performing those replacements yields tree (c).

replaced by Y. If you perform several unifications $X = Y$, $Y = Z$, $X = R$, and $R = c$, then afterwards you must find that each of X, Y, Z and R has been replaced by symbol c.

Unification is active

The pattern matching that is performed in equational programming only binds names, and has no effect on the data values that are being examined. It is a *passive* form of pattern matching. Unification, in contrast, is *active*: it can bind unknowns that occur inside data structures, causing those data structures to be modified by the addition of information to them. But the active nature of unification is restricted to filling in information that was previously left unspecified. Unification can replace unknown X by 2, but it cannot replace 2 by 3, or symbol horse by symbol zebra.

Unification can fail

Not all equations have solutions, and it is possible for unification to fail. When it does, a particular branch of a computation comes to a halt, and another branch is tried. For example, equation $[A, A] = [u, v]$ has no solution. Remember that u and v are constants, since their names start with lower case letters. So u and v cannot be changed to be anything else. The only unknown is A, and no substitution for A can possibly make list $[A, A]$ be the same as list $[u, v]$.

28.4. Implementation of unification

It is instructive to see how unification is done in a typical logic programming implementation. Each unknown is represented by a variable that can hold a pointer to a value, but that initially holds a null pointer. When you bind unknown X to T, you just store a pointer to T into variable X. If you bind X to another unknown Y, then you put a pointer to Y into X. (So a pointer can point to another pointer.) Figure 28.4 shows an example where X has become bound to W and Z has been bound to the tree rooted at **robin**.

Using pointers avoids doing actual substitutions; each variable is stored in just one place, so there is only one thing to update when a variable becomes bound. But you have to be careful dealing with bound unknowns (nonnull pointers). Each time a bound unknown

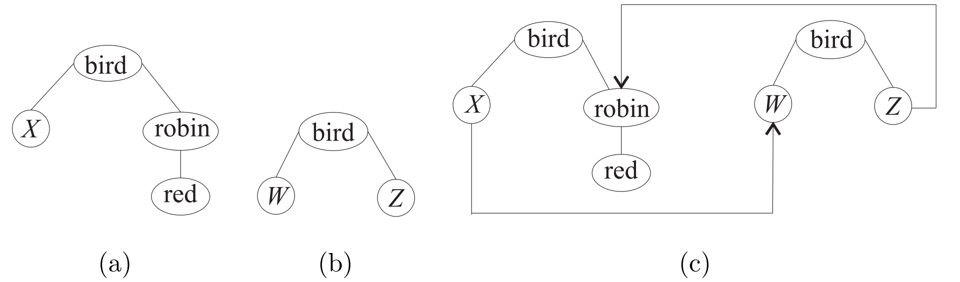
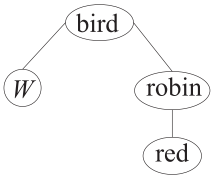


Figure 28.4: **Unification by binding boxes.** Unification of the two trees (a) and (b) yields tree (c).

is encountered, skip across its pointer to see what its value really is. If you do that, and you ask what the result of a unification is, you get the same result regardless of which tree you look at. The result of unifying the two trees in Figure 28.4 is the following tree.



The basic rules for carrying out unification on trees are as follows.

1. If either tree is a bound unknown, follow the pointer(s), until neither tree is a bound unknown. This might require several hops over bound unknowns.

2. If neither tree is an unbound unknown, then check that the roots of the trees have the same label, and that each has the same number of subtrees. If not, then the unification fails. If so, then unify corresponding subtrees. The leftmost subtree in each is unified with the leftmost subtree in the other, etc. The unification succeeds only if all of the pairs of subtrees can be unified.

   If there are no subtrees, then there are no recursive unifications to be done. The unification simply succeeds if the two leaves have the same label, and fails if they are different.

3. If one of the values is an unbound unknown and the other is not, then bind the unbound unknown to the other tree.

4. If both values are unbound unknowns, then bind one unknown to the other. There is an important exception, however; to unify an unknown with itself, do nothing.

The occur check

Unification as presented so far is called *naive unification.* Figure 28.5 shows an example where X is unified with plus(X,X) using naive unification. The result is a "tree" that

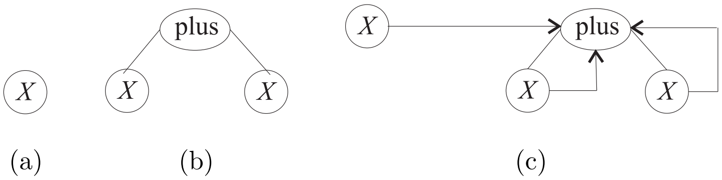
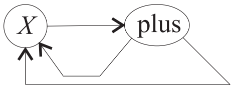


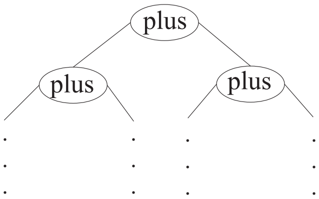
Figure 28.5: **Naive unification can yield cycles.** Unifying the two trees (a) and (b) yields "tree" (c).

contains cycles, and allowing that seems to create a value that is not in the usual domain of discourse.

It is important to keep in mind that each occurrence of unknown X is really the same unknown, and is only stored in one place in the memory. Binding one occurrence binds them all. X is shown in the diagram in three places only to make the diagram more readable. A more realistic picture of the two trees after unification is as follows.



To understand the result, you need to skip through bound unknowns when they are encountered. The result is an infinite tree.



Infinite structures can be useful in computation. But in mathematical logic, they cause difficulty, partly because they introduce things that have not been admitted into the domain of discourse, and so are not supposed to exist, and partly because you simply cannot write down an infinitely long expression. There is a modification to naive unification that disallows cycles, and that yields the *full unification* algorithm. The modification is called the occur check, and is a proviso added to step 3.

> (**Occur check**) When binding an unbound unknown X to a tree T that is not an unbound unknown, check whether X occurs inside T. If it does, then the unification fails. If T does not contain X, then bind X to T; the unification succeeds.

Using the occur check ensures that cycles are never created, and that trees remain finitely large.

Although the occur check is relatively simple to state, it is fairly expensive to perform. The entire tree T must be searched for occurrences of X. Unification is the only mechanism that is available in a logic program for binding an unknown, or even for simply giving a tree a name. It would be expensive indeed if, just in order to name a list, you had to search the list to ensure that the new name did not already occur in that list. Most logic programming languages use naive unification rather than full unification because naive unification is so much faster to do. You must be careful though; you can get infinite trees. ISO Prolog, a relatively recent addition to the Prolog family, uses the occur check.

The occur check does not have to be exceedingly costly. It is possible for an ISO Prolog compiler to recognize typical situations where the check can be skipped. For example, binding a newly created unknown to a previously created tree never requires doing the occur check, and that is the most common way that unification is used.

28.5. A careful treatment of unification

If you feel that you understand unification, you probably do not need to read this section, which gives a careful definition of unification. In this section, we treat trees strictly in their expression forms, such as $f(X, Y)$. In logic, expressions are called *terms* and unknowns are called *variables*, and we adopt that terminology here.

Substitutions

A *substitution* is a collection of variable bindings. We will write substitutions in braces. For example, $\{X = \text{joe}, Y = \text{table(chair)}\}$ is a substitution that binds variable X to joe and Y to the term table(chair). We will use Greek letters such as θ and α to name substitutions.

If θ is a substitution and A is a term, write $\theta(A)$ for the term that results by performing substitution θ on term A, replacing each variable that is bound in θ by the term to which it is bound. For example, if A is term flower(Y, rose) and $\theta = \{Y = \text{four}\}$, then $\theta(A)$ is term flower(four, rose).

Suppose that $\phi = \{X = Y, Y = \text{eighty}\}$ is a substitution. Then $\phi(\text{five}(X, Y))$ yields five(Y, eighty). But that still contains variable Y, which ϕ says should be replaced by eighty. You could use ϕ again, but it is better to use a substitution that only needs to be performed once. Say that a substitution θ is *simple* if no variable that occurs on the left-hand side of an equation in θ also occurs on the right-hand side of an equation. Substitution ϕ is not simple, since Y occurs on both the left-hand side and the right-hand side of an equation. But substitution $\{X = \text{eighty}, Y = \text{eighty}\}$ is simple. If θ is a simple substitution then $\theta(\theta(A)) = \theta(A)$. The second application of θ cannot have any effect.

Unifiers

When you perform a successful (full) unification, the outcome is a simple substitution indicating the bindings of variables that should be done to make the two terms identical.

Definition. A simple substitution θ is a *unifier* for A and B if $\theta(A) = \theta(B)$.

For example, $\theta = \{X = \text{eighty}, Y = \text{eighty}\}$ is a unifier for terms five(X, eighty) and five(eighty, Y). Say that terms A and B are *unifiable* if a unifier for them exists.

Principal unification

An important feature of unification concerns how variables are unified with one another. Suppose that you want to unify terms bird(X, Y) and bird(W, Z). You must ensure that X

and W are the same, and that Y and Z are the same. There is a myriad of ways to do that. For example, to ensure that X and W are the same, you could bind each of them to symbol rocket. But that approach has a serious drawback. What would happen if, later, you tried to unify one of those two terms with term bird(feeder, Q). If you have already committed to X being rocket, then the later unification is impossible. But if you had seen ahead, and instead bound X and W each to feeder, the later unification would have succeeded.

That problem is overcome by performing unification in a most general way, allowing as many subsequent unifications to take place as possible. A unifier that is most general in this sense is called a *principal unifier*. Instead of replacing X and W each by a fixed value, a principal unifier chooses one of those variables to stand for both. Either X can be replaced by W or W can be replaced by X.

The definition of a principal unifier as a "most general" unifier can be made much more precise. After performing the substitutions called for by a principal unifier θ, the effects of any other (possibly nonprincipal) unifier α can be accomplished by performing more substitutions (β) after the substitutions called for by θ. For example, $\alpha = \{X = \text{rocket}, W = \text{rocket}\}$ is a nonprincipal unifier for X and W. But it is equivalent to principal unifier $\theta = \{X = W\}$ followed by substitution $\beta = \{W = \text{rocket}\}$ since, for any term t, $\alpha(t) = \beta(\theta(t))$.

> **Definition.** Let A and B be two unifiable terms. A unifier θ for A and B is *principal* if, for every unifier α for A and B, there exists a simple substitution β such that $\beta(\theta(A)) = \alpha(A)$.

If two terms A and B can be unified, then there is always a principal unifier for them, and there is only one principal unifier, up to renaming of variables. (For example, a principal unifier might replace X by Y or Y by X, but the only difference is the choice of name, X or Y, for the variable.)

Principle unification algorithm

Figure 28.6 shows an equational definition of principal unification of terms. Notice that it uses the occur check, which is critical. Terms that can be unified without the occur check, but that cannot be unified with the occur check, have no unifier, according to our definition of a unifier.

28.6. Summary of terminology and concepts

Logic programming uses trees as the fundamental data structure, with internal nodes labeled by symbols and leaves that can be symbols, numbers, strings, etc. Leaves can also be unknowns, or variables, that stand for yet-to-be-determined subtrees.

The values of unknowns are found as information becomes available during execution of a logic program. The information typically takes the form of an equation, indicating that two given trees must be the same. Unification, a form of pattern matching, is the tool that incorporates the equational information into the data, binding unknowns appropriately.

Terminology

Full unification Full unification is the unification algorithm that employs the occur check.

$$
\begin{array}{rcl}
\mathrm{unify}(v, v) & = & [\,] \\
\mathrm{unify}(v, A) & = & [(v, A)] \;\; \text{provided } v \text{ does not occur in } A \\
\mathrm{unify}(A, v) & = & \mathrm{unify}(v, A) \\
\mathrm{unify}(f(), f()) & = & [\,] \\
\mathrm{unify}(\; f(A_1, \ldots, A_n), & = & \theta_1 +\!+\, \theta_2 \;\; \text{where} \\
f(B_1, \ldots, B_n)) & & \quad \theta_1 = \mathrm{unify}(A_1, B_1) \\
 & & \quad \theta_2 = \mathrm{unify}(\; f(\theta_1(A_2), \ldots, \theta_1(A_n)), \\
 & & \qquad\qquad\qquad f(\theta_1(B_2), \ldots, \theta_1(B_n)))
\end{array}
$$

Figure 28.6: **An equational definition of principal unification.** Given two terms S and T, unify(S, T) produces a principal unifier of S and T as a list of pairs $[(v_1, t_1), \ldots, (v_m, t_m)]$, indicating substitution $\{v_1 = t_1, \ldots, v_m = t_m\}$, where each v_i is a variable and t_i is a term. If no rule applies, then there is no unifier; similarly, if any of the recursive calls to unify fail to produce a unifier, then there is no unifier. In the equations, v is an unbound variable, A is a term that is not a variable, f is an arbitrary symbol, and $\theta(A)$ is the result of applying substitution θ to term A. A leaf is equivalent to a function of no parameters, shown as $f()$ in the equations.

Naive unification Naive unification is the unification algorithm that does not employ the occur check, and allows creation of cyclic structures.

Occur check The occur check is a provision stating that an unbound variable X cannot be unified to a (nonvariable) tree that contains X.

Principal unification Principal unification is a most general way of unifying terms, allowing as many subsequent unifications to be performed as possible.

Substitution A substitution is a binding of unknowns to terms.

Simple substitution A substitution θ is simple if no variable occurs on both the left-hand side of an equation and on the right-hand side of a (possibly different) equation in θ.

Term Expressions in logic are called terms. In logic programming, a term refers to a tree, possibly containing unknowns, and possibly being just a single node or a single unknown.

Tree A tree is the basic data structure of a logic program. An internal node is labeled by a symbol, and can have any number of children, ordered from left to right. A leaf can contain any kind of data item supported by the language, possibly including symbols, numbers, characters, etc.

Unification Unification is a form of pattern matching that is symmetric. It tries to bind unknowns in order to make two trees (or terms) identical.

Unifier A unifier is a simple substitution that makes two given terms identical. That is, θ is a unifier for A and B if $\theta(A) = \theta(B)$.

Universe of discourse The universe of discourse of a logic program consists of all of the values that unknowns can stand for. In typical logic programming languages, the universe of discourse contains trees, including trees that are just leaves, so they appear to be simple values such as numbers.

Unknown An unknown is a quantity that has not yet been determined. It is an unbound variable. An unknown can only be bound once on any given branch of a program; binding an unknown is permanent for a given branch, and you cannot change your mind and bind it to something else on the same branch. (But different branches can choose different values for an unknown.)

Variable In logic programming, a variable is the same thing as an unknown.

28.7. Exercises

28.1. Show each of the following as a tree diagram.

(a) bear(black)

(b) fruit(mango, citrus(orange, tangerine))

(c) $f(X,\, g(Y))$

28.2. Unification is a form of pattern matching. Which of the following is *not* a characteristic of unification?

(a) Unification never changes the binding of a bound variable.

(b) Unification is symmetric; unifying A with B has exactly the same effect as unifying B with A.

(c) Unification performs arithmetic, so binding X to $5 + 6$ will bind X to 11.

(d) Unification can bind unbound variables.

28.3. Show tree diagrams for each of the following pairs, and show how unknowns become bound when the pair is unified, by showing the pointers. There might be more than one way to do some of them, since to unify X with Y, you can either replace X by Y or Y by X.

(a) $f(X, Y)$ and f(apple, orange)

(b) $f(X, Y)$ and f(apple, X)

(c) $f(g(X, Y), Z)$ and $f(Z, W)$

(d) $r(g(X), Y, h(Z))$ and $r(Y, W, X)$

28.4. Give a principle unifier for each pair of terms in problem 28.3.

28.5. What happens if you unify terms $f(X, Y)$ and $f(g(Y), X)$ using naive unification? Show the result in a tree diagram, with pointers. Can those terms be unified using full unification?

28.6. Give a principal unifier and two different examples of non-principal unifiers of terms $f(X, Y)$ and $f(Z, W)$. Then show how to accomplish the same effect as each of those non-principal unifiers in two steps, first by using a principal unifier, then by performing another simple substitution.

28.7. Why do you think an unknown is only allowed as a leaf of a tree? (Remember that an unknown can be bound to a tree.)

28.8. Bibliographic notes

Robinson [90] studies automatic theorem proving, and introduces unification as part of the resolution principle, which underlies logic programming as studied in the next chapter.

Chapter 29

Concepts of Logic Programming

29.1. What is logic programming?

Chapter 26 explores backtracking. Although backtracking is a powerful control mechanism, it can be somewhat difficult to understand, and convincing yourself that a program that employs backtracking is correct can be difficult. Logic programming is a form of declarative programming that is partly a way of putting reins on backtracking, making it more manageable. Where, in a functional program, facts are expressed as equations, a logic program might express "facts" such as "All apples are good to eat" or "If I can see Orion, it must be Winter." To convince yourself that a logic program can only produce correct answers, it suffices to make sure that your stated facts are, indeed, facts.

A computer program can be thought of as having two main ingredients. One ingredient is the *logic* of the program, consisting of the facts that the programmer knows, and upon which the correctness of the program is based. The other ingredient is the *control*, the knowledge of how to use those facts to perform computations. For example, you might know that $0! = 1$ and $n! = n(n-1)!$ for $n > 0$. Those are the facts that you work with. To compute 6!, you employ those facts in a particular way. You might, for example, write a loop to accumulate the factorial of 6, using the second fact repeatedly to get higher and higher factorials.

A logic programming implementation consists of an *inference engine*, typically an interpreter, that is used to perform logical inferences of new facts from old, and that provides most of the control. The engine finds proofs, based on the given facts, and it is the action of this engine that is responsible for the computation of a logic programming system. The programmer provides the facts, and the inference engine determines how to use those facts to perform specific computations.

There are several forms of logic programming, some with relatively simple engines and some with very sophisticated engines. The nature of the engine influences how facts are written, and how they are used. We will focus on just one form of logic programming, exemplified by the language Prolog. It is the first form of logic programming that was implemented, and is still a popular one today.

29.2. Clauses and formulas

A *logical formula*, or simply a *formula*, is an expression that is either true or false. This section explores the structure of formulas.

Predicates and atoms

A predicate is a function that returns a boolean result. It captures properties of things or relationships among things. When you say that an apple is green, you capture the notion of "greenness" in a predicate, saying that green(apple) is true. The relationship "A is the mother of B" can be expressed as mother(A, B), where predicate mother is true of an ordered pair (A, B) just when A is the mother of B. A predicate can have any number of parameters, so that it can express complex relationships among several things.

An *atom* is an application of a predicate to an argument or arguments. For example, mother(X,Y) is an atom. The arguments can be constants or variables (unknowns), or possibly more complex expressions that describe trees. For example, if *greater* is a predicate, then greater(5+2, 2+1) is an atom, claiming that the "greater" relationship holds between the two data items $5 + 2$ (a tree with $+$ at its root) and $2 + 1$ (another tree). In some cases, a predicate has no arguments, and the predicate by itself is an atom, standing for the value of the predicate (true or false).

Boolean operations and literals

Atoms can be combined using boolean operations. The usual boolean operations are as follows.

1. **Conjunction.** $A \wedge B$ is read "A and B", and is true just when both A and B are true.
2. **Disjunction.** $A \vee B$ is read "A or B", and is true when either A or B or both are true.
3. **Negation.** $\neg A$ read "not A", and is true just when A is false.
4. **Implication.** $A \Rightarrow B$ is equivalent to $(\neg A) \vee B$. Notice that, if A is true, then $A \Rightarrow B$ can only be true if B is also true. $A \Rightarrow B$ is read "A implies B",

The usual rules of precedence give $\neg$ the highest precedence, followed by $\wedge$, then $\vee$, with $\Rightarrow$ having the lowest precedence.

A *literal* is either an atom or the negation of an atom. For example green(apple) and ¬green(apple) are literals. A literal such as green(apple) that is not negated is called a *positive literal*, and a literal such as ¬green(apple) is called a *negative literal*.

Formulas and Horn clauses

Logical *formulas* are built by using boolean operations on atoms and other formulas in arbitrary ways. For example, $p(X, Y) \Rightarrow r(X) \vee q(Y)$ is a formula. But logic programming does not work with arbitrary formulas. Instead, it uses restricted kinds of formulas called Horn clauses that make the inference engine easier to define and easier to understand.

A *clause* is a disjunction of literals. That is, it has the form $L_1 \vee L_2 \vee \ldots \vee L_n$, for some $n \geq 1$, where each of $L_1, \ldots, L_n$ is a literal. (A literal by itself is considered to be a clause of one literal.) A *Horn clause* is a clause that contains exactly one positive literal. For example, suppose that mother and parent are predicates. Then ¬mother(X,Y) ∨ parent(X,Y) is a Horn clause, since it has exactly one positive literal. The positive literal is called the *head* of the Horn clause.

A Horn clause is not required to have any negative literals. A clause consisting of nothing but one positive literal states a basic fact. For example, mother(betty,bob) states the fact that betty is the mother of bob. If a Horn clause has at least one negative literal, then it can be expressed as an implication. For example,

1. $\neg A \vee B$ is equivalent to $A \Rightarrow B$, from the definition of implication.

2. $\neg A \vee \neg B \vee C$ is equivalent to $(A \wedge B) \Rightarrow C$. That is, C is true if both A and B are true. You can understand why by imagining the case where both A and B are true, and asking how $\neg A \vee \neg B \vee C$ might also be true. Since neither $\neg A$ nor $\neg B$ is true, the only way to make $\neg A \vee \neg B \vee C$ true is to make C true. That is, whenever A and B are both true, C must be true.

3. $\neg A \vee \neg B \vee \neg C \vee D$ is equivalent to $(A \wedge B \wedge C) \Rightarrow D$.

Notice that, when a Horn clause is expressed as an implication, there is always just one positive literal on the right-hand side of the implication, and all of the literals on the left-hand side of the implication are also positive.

Horn clause $(A \wedge B) \Rightarrow C$ can be thought of as saying "if you want to show that C is true, try proving that both A and B are true." As we will see shortly, logic programming is based on proofs; proving something and computing it are the same thing. In programming languages, it is common to write what you want to compute on the left hand side of a fact and to write information on how to compute it on the right hand side. For example, equation $f(x) = x * x$ tells you to compute $f(x)$ by computing expression $x * x$. To keep this familiar order, it is common practice to write implications using the right-to-left implication symbol $\Leftarrow$, where $B \Leftarrow A$ is the same as $A \Rightarrow B$.

Also, in a Horn clause, the $\wedge$ operator is usually replaced by a comma. So $C \Leftarrow A, B$ says, to prove that C is true, try proving that both A and B are true. The remainder of this section uses the modified notation for Horn clauses.

Programs

A logic program is a collection of Horn clauses, called the *axioms* of the program, stating facts that the inference engine can use. A fact that contains variables is understood to be true for all values of the variables. Computation of a logic program is the process of the inference engine deriving new facts from the axioms stated in the program.

29.3. Logic programs and computation

This section begins to look at how the inference engine works.

Goals

A *goal* is a question that you ask the inference engine to answer. The simplest kind of question is a yes/no question; you ask the engine whether it can prove a particular atom.

In general, a goal is a list of atoms, and it is the inference engine's responsiblity to prove *all* of the atoms in the list. Each atom in a goal list is called a *subgoal*.

For example, if the goal list holds (parent(betty, bob), mother(betty, ron)), then the engine needs to find a way to prove both that parent(betty,bob) is true and that mother(betty, ron) is true. Think of the goal list as a to-do list of things that need to be proved. The first thing that the inference engine does is select one of the atoms to work on. The simplest rule is to select the first subgoal in the list, and that is the rule that we will use.

Having selected a subgoal to work on, such as parent(betty, bob), the engine looks for a way to prove that subgoal. It examines the axioms, selecting all axioms that *might* work. An axiom is interpreted as follows for the purposes of proving an atom.

1. A clause that contains no negative literals states a fundamental fact. For example, if the engine is trying to prove that mother(betty, ron) is true, and mother(betty, ron) is stated as an axiom in the program, then the job is done. In this case, the subgoal is erased from the list of subgoals and the engine selects another subgoal to work on.

2. A Horn clause of the form $B \Leftarrow A$ says that, if you want to prove B, try proving A. That is, it suggests replacing subgoal B by subgoal A. For example, if the engine is currently trying to prove parent(betty, bob), and one of the axioms is parent(betty, bob) $\Leftarrow$ mother(betty, bob), then the engine can choose to replace subgoal parent(betty, bob) by subgoal mother(betty, bob). The engine modifies its list of subgoals, and again selects a subgoal to work on.

   When the engine removes a subgoal from its list and replaces that subgoal by another one, it needs to make a choice about where to put the newly inserted subgoal — should it go at the front of the list or at the back? The usual rule it to put the new subgoal at the front of the list. That way, after replacing parent(betty, bob) by mother(betty, bob), the engine will select mother(betty, bob) next to work on, continuing on this same line of reasoning.

3. A Horn clause of the form $(C \Leftarrow A, B)$ says that, if you want to prove C, try proving both A and B. So you remove C from the list of subgoals and insert A and B. In general, if there are n literals on the right-hand side of $\Leftarrow$, then you add all n of those literals to the subgoal list. The usual rule is to insert them so that they end up in the same order as they are in the axiom. When axiom $(C \Leftarrow A, B)$ is used to prove C, subgoal A will end up at the front of the new subgoal list, and it will be the next subgoal worked on.

The role of unification

When evaluating an expression in an equational program, pattern matching has to be used when performing substitutions. The reason is that the equations that a program states are more general than you need at a particular step in the computation. For example, if an equation states that $f(x) = x * x$, and you want to compute $f(3)$, then you need to find a special form, $f(3) = 3 * 3$, of the more general equation, and pattern matching does that.

Pattern matching (by unification) is also used when evaluating a subgoal in a logic program, for the same reason. It needs to find a specialized form of an axiom that is suitable for a particular step in reasoning. For example, suppose that the program states, as an axiom,

$$\text{parent}(X, Y) \Leftarrow \text{mother}(X, Y)$$

When an axiom contains variables, the programmer is asserting that the axiom is true for *all* values of those variables. So, for any X and Y, if X is Y's mother, then X is also Y's parent. To prove goal parent(betty, bob), a pattern match is done, matching parent(X,Y) against parent(betty,bob). Clearly, X must be betty and Y must be bob. The specialized axiom is

$$\text{parent}(\text{betty}, \text{bob}) \Leftarrow \text{mother}(\text{betty}, \text{bob})$$

and subgoal parent(betty, bob) is therefore replaced by new subgoal mother(betty, bob).

In general, suppose that your current list of subgoals is g_1, …, g_n, and that you want prove g_1 using axiom $H \Leftarrow L$, with head H (an atom) and right-hand side L (a list of atoms). You first try to unify g_1 with H. If that unification succeeds, then you perform the substitutions that are required by the unification on all of the atoms in list L and in list g_2, …, g_n, yielding modified lists L' and g'_2, …, g'_n. The new list of subgoals is L' followed by g'_2, …, g'_n.

For example, suppose that you want to prove goal list (mother(Z, curly), mother(Z, moe)), and that the first axiom is mother(sally,curly). Working on goal mother(Z, curly), start by unifying pair (Z,curly) with (sally,curly), yielding unifier $\{Z = \text{sally}\}$. In this case, since the right-hand side of the axiom is empty, no new goals are added, and the new list of goals is (mother(sally, larry)). Notice that variable Z has been replaced in the remaining goal. If you decide that Z = sally, then you must include that decision in all of the subgoals, not just in the one you are working on.

The role of backtracking

Suppose that a logic program contains two axioms,

$$\begin{array}{rcl} \text{parent}(X, Y) & \Leftarrow & \text{mother}(X, Y) \\ \text{parent}(X, Y) & \Leftarrow & \text{father}(X, Y) \end{array}$$

where the first says that X is a parent of Y if X is the mother of Y, and the second says that X is a parent of Y if X is the father of Y. A goal is to prove parent(a,b) for two particular individuals a and b. There are two ways to do the proof; you can either replace goal parent(a,b) by new goal mother(a,b), or you can replace it by new goal father(a,b). It is not clear which axiom to use. Certainly, you do *not* want to replace it by both new subgoals, putting both into the list of subgoals to be proved. That would say that, to prove that a is a parent of b, you must prove *both* that a is b's mother and that a is b's father. But what you want to do is to prove *either* that a is b's mother or that a is b's father.

A solution is to use backtracking. Each axiom is tried, but each is tried in a separate branch of the computation. If the first branch succeeds (so proving mother(a,b) succeeds), then parent(a,b) is proved. But if the first branch fails, then the second branch, which tries to demonstrate father(a,b), is tried instead.

Unknowns can become bound during the course of computation. They must be nonshared variables. When computation forks into two or more branches, the unknowns are, effectively, copied, so that each branch sees its own copies of the unknowns, and different branches do not interfere with one another. Chapter 27 discusses how that copying can be simulated, at lower cost than actual copying, by maintaining *trail* frames that allow bindings to be undone when the inference engine backtracks.

1. mother(marcy, cathy).
2. mother(sally, bertrand).
3. father(bertrand, marcy).
4. father(william, cathy).
5. father(jordan, william).
6. parent(X,Y) $\Leftarrow$ mother(X,Y).
7. parent(X,Y) $\Leftarrow$ father(X,Y).

Figure 29.1: **A sample logic program.** Each line is a Horn clause. Predicate mother(X,Y) means "X is the mother of Y," father(X, Y) means "X is the father of Y" and parent(X, Y) means "X is a parent of Y."

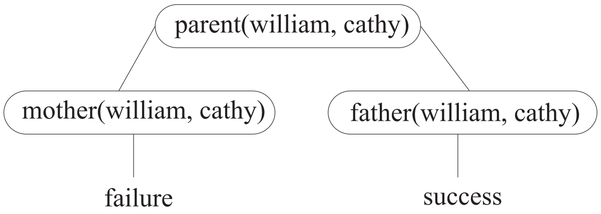


Figure 29.2: **Computation of goal parent(william, cathy), using the axioms from Figure 29.1.** The failure of mother(william, cathy) is because there is no Horn clause whose head unifies with mother(william, cathy). Goal father(william, cathy) is replaced by no goals, so the computation terminates successfully.

Example

Figure 29.2 shows computation of goal parent(william, cathy), using the axioms shown in Figure 29.1. Each node of the tree contains a "to do" list of subgoals to be proved. The computation is shown as a tree, since it uses backtracking; each branch of the tree represents one way in which the goals might be proved. Notice that subgoal parent(william, cathy) is replaced by mother(william, cathy) in one branch and by father(william, cathy) in the other branch. *Failure* indicates that none of the axioms can be used to prove the first subgoal. *Success* indicates that the list of subgoals is empty; all have been proved.

29.4. Using unknowns

Goals involving unknowns

Imagine trying to prove goal parent(william, X), containing an unknown X, using the axioms of Figure 29.1. Computation of that goal is shown in Figure 29.3. During the process of the computation, unknown X becomes bound by unification. If you read backwards from the success to the root of the tree, and examine the bindings that are done, you can see what was proved; it was not really parent(william, X) that was proved, but the special case parent(william, cathy). Think of a goal containing unknown X not just as a yes/no question, but as a request that the engine find a value for X that makes the goal true, or

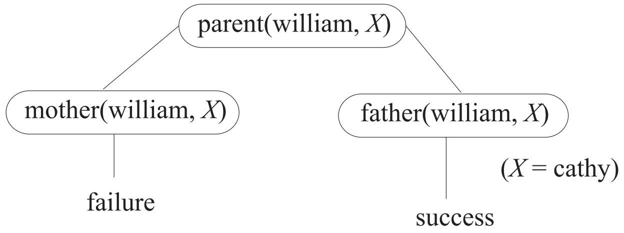


Figure 29.3: **Computation of goal parent(william, X), using the axioms from Figure 29.1.** Unification of goal father(william, X) and axiom father(william, cathy) causes unknown X to become bound to cathy, as is shown in the computation. The computation finds the value of X.

to tell you that there does not exist any such value. In the example, you are asking for a child of william, and the engine tells you that cathy is a child of william.

In general, when a goal contains an unknown, the inference engine is asked to find a value for that unknown that makes proof of the goal possible, and unification is the tool that finds that value. So unification is not only responsible for specializing axioms to work for particular subgoals, but also for specializing subgoals so that they can be proved by particular axioms.

Notice the distinction between a goal and an axiom. An axiom is thought of as expressing a *property of a predicate*, namely, that the given fact is true for all values of the variables. A goal, on the other hand, is thought of as stating a *property of the variables that the goal contains*, and the problem is to find values for the variables that have the desired property.

Auxiliary variables

Suppose that you want to define a new predicate grandparent, where grandparent(X, Y) is true when X is a grandparent of Y. You can express what it means to be a grandparent in (at least) two equivalent ways.

1. X is a grandparent of Y if there is some individual Z such that X is a parent of Z and Z is a parent of Y.
2. For any three individuals X, Y and Z, if it turns out that X is a parent of Z and Z is a parent of Y, then you can conclude that X is a grandparent of Y.

The second formulation avoids having to say that something exists, instead saying that a particular statement is true for all X, Y and Z. It leads to the following Horn clause defining predicate grandparent.

8. grandparent(X, Y) $\Leftarrow$ parent(X, Z) $\wedge$ parent(Z, Y).

If you think about using axiom (8) to prove a goal such as grandparent(bertrand, cathy), you see unknown Z popping up. The program does not say how to determine what Z should be, but that is all right, since it is the inference engine's responsibility to find things, and the engine should be able to find Z, without any help. The combined effects of backtracking

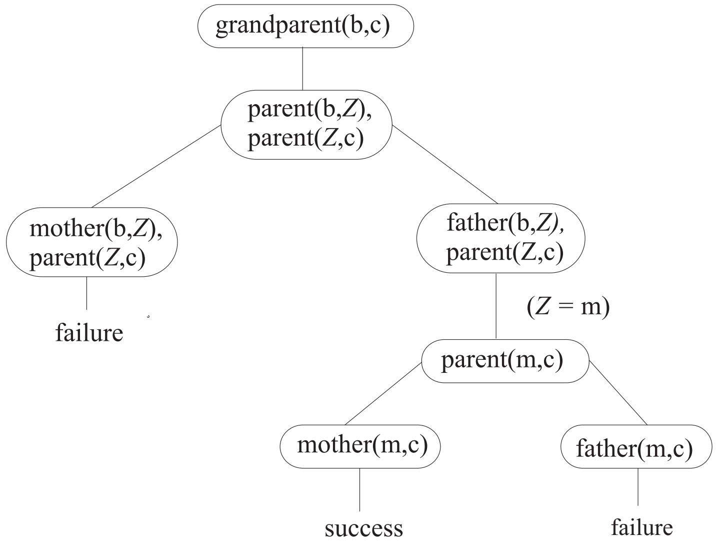


Figure 29.4: **Computation of goal grandparent(bertrand,cathy).** To reduce the size of the figure, each person's name is replaced with its first letter. For example, *b* stands for bertrand.

and unification will find an appropriate Z automatically. Figure 29.4 shows computation of goal grandparent(bertrand, cathy).

To prove a list of goals, you start with the first one. Notice that replacing parent(bertrand, Z) by mother(bertrand, Z) does not affect the other goal, parent(Z, cathy), since that goal must still be proved later. But when Z is bound to marcy, it is replaced in the remaining goal, parent(Z, cathy), yielding goal parent(marcy, cathy).

Figure 29.5 shows computation of goal grandparent(X, marcy). Notice that both X = sally and Z = bertrand are found automatically. The program (or, if you prefer, the programmer) is not concerned with where they come from.

Renaming variables

Think, for the moment, of a procedural or functional language where you have a function $f(n)$. When you call this function the first time, you might choose $n = 2$. The next time, you could choose $n = 6$. Each time you use the function, you begin a new copy of the function, with a different variable n, and it is important to realize that there is more than one variable called n in the program.

The same kind of thing happens in a logic program. A given axiom might be used several times during a computation. Unifications that are done in one use of the axiom must not affect another use of the same axiom. If an axiom that includes variables is used more than once, a copy of the axiom should be made, with brand new variables that have not been used before. For example, if axiom (8) is used twice, the first time it might have the form (grandparent(X_1,Y_1) $\Leftarrow$ parent(X_1,Z_1), parent(Z_1,Y_1)), but the second time it could be (grandparent(X_2,Y_2) $\Leftarrow$ parent(X_2,Z_2), parent(Z_2,Y_2)). Renaming variables is especially important in logic programming because the variables that you create can remain unbound

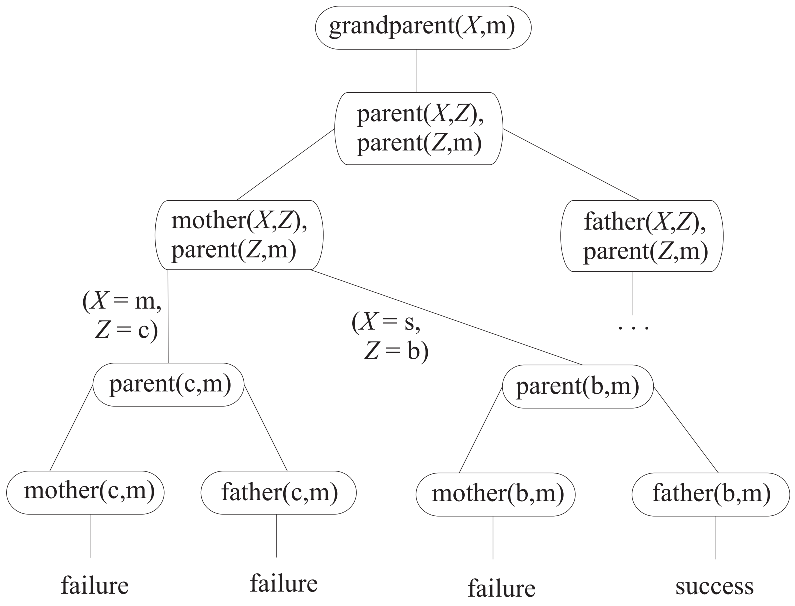


Figure 29.5: **Computation of goal grandparent(X,marcy).** Names are abbreviated to their first letters.

even after evaluation of a goal is finished. That is, a local variable can still exist, as an unbound unknown, even after the predicate that created it has been removed from the goal list.

Summary of computational rules

Here is a summary of the rules that you should keep in mind when building proof trees.

1. When a subgoal is replaced with a list of new subgoals, only that subgoal is replaced. Any other subgoals that do not participate are left in place, and must still be proved later.

2. When an unknown is bound, it must be replaced in *all* of the subgoals in the current list. For example, if the goal list is (mother(X, Z), parent(Z, marcy)), and you bind of Z to cathy in order to prove mother(X, Z), you need to bind Z to cathy in the second subgoal, parent(Z, marcy), as well.

3. Each time an axiom is used, you need to create a new copy of that axiom, with new variables, to keep different uses of an axiom from interfering with one another.

4. A successful proof is found when the goal list becomes empty. The bindings of the unknowns can be found by tracing up toward the root of the tree, reading off the bindings that were done by unification.

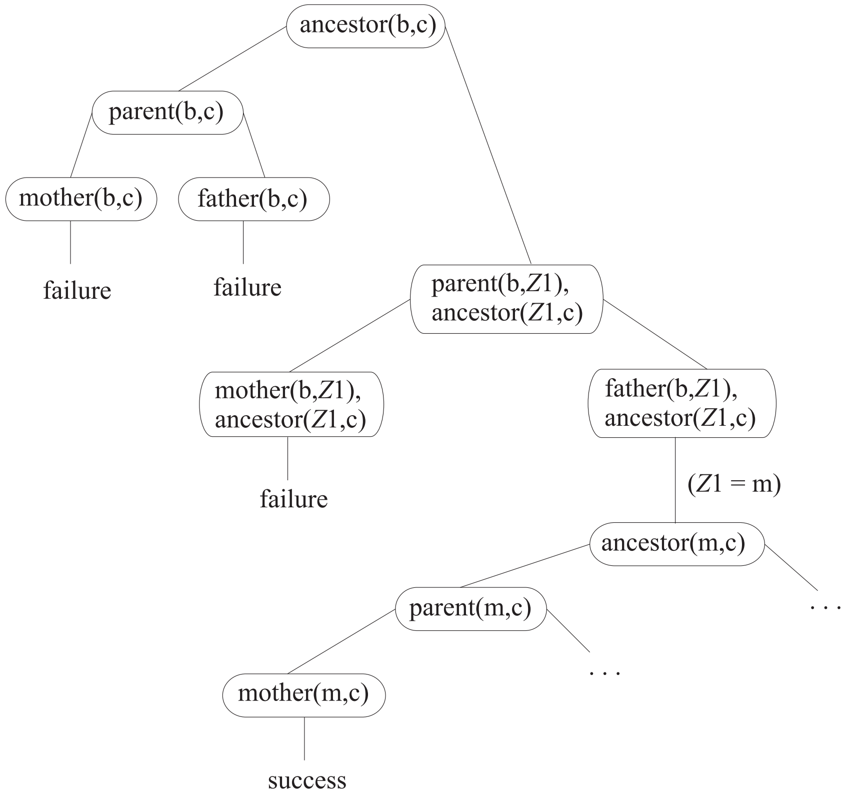


Figure 29.6: **Computation of goal ancestor(bertrand, cathy).** Unification performs quite a few variable bindings, but only the binding of Z_1 to m is shown explicitly.

29.5. Recursion

Suppose you want ancestor(X, Y) to be true if X is an ancestor of Y. Two facts are evident.

9. ancestor(X, Y) $\Leftarrow$ parent(X, Y).
10. ancestor(X, Y) $\Leftarrow$ parent(X, Z), ancestor(Z, Y).

Figure 29.6 shows a proof of goal ancestor(bertrand, cathy).

29.6. Summary of terminology and concepts

Logic programming is a form of declarative programming. It separates the logic (facts) on which a program is based from the control of the computation. The logic consists of a collection of Horn clauses that tell properties of predicates. The control is achieved by an inference engine that uses the axioms to prove new facts. The engine searches for a proof by backtracking through all possible ways of doing proofs. Proofs are found by starting at a goal and reasoning backwards to subgoals, until there are no more subgoals to prove.

The inference engine finds values that makes it possible to prove the desired facts. For example, the process of proving mother(X, bertrand) finds a value for X that makes mother(X, bertrand) true. The engine uses unification to find appropriate values of unknowns.

Terminology

Atom An atom is a predicate applied to an argument or arguments. An atom is also called a positive literal, or sometimes an atomic formula.

Axiom A fact that is part of a program is called an axiom. An axiom is understood to be true for all values of the variables that it mentions.

Clause A clause is a disjunction (*or*) of literals.

Control The control of a program tells how the underlying facts are used to reach a conclusion.

Goal A goal is a formula that a logic programming system is asked to prove. Any variables in the goal are intended to be found by the inference engine, which finds particular values for the variables that make the goal true.

Horn clause A Horn clause is a clause that has exactly one positive literal. Equivalently, a Horn clause can be described as a formula of the form $A_1 \wedge A_2 \wedge \cdots \wedge A_n \Rightarrow B$, where $n \geq 0$ and all of $A_1, A_2, \ldots, A_n$ and B are positive literals.

Inference engine An inference engine forms the core of an implementation of a logic programming language. The inference engine is responsible for finding proofs of lists of goals.

Literal A literal is either an atom (a positive literal) or the negation of an atom (a negative literal).

Logic of a program The logic of a program consists of the facts upon which computation is based, without regard for how those facts are used to reach conclusions. The axioms of a logic program express those facts.

Logic programming Any form of programming where a program is a collection of facts written using mathematical logic, and the language implementation consists of some form of inference engine, is called logic programming.

Predicate A predicate is a function that returns a boolean result, so that it classifies things as “good” (true) or “bad” (false). Since a predicate can take several parameters, it can also express relationships among things.

29.7. Exercises

29.1. What is the definition of a Horn clause? What is the head of a Horn clause?

29.2. Does logic programming, as described in this chapter, allow an axiom to be any formula at all, or is there a restriction on kinds of formulas that can be used? Explain.

29.3. When an unknown X occurs in an *axiom*, is the engine being told that the axiom holds for all values of X, or for some value of X?

29.4. When an unknown X occurs in a *goal*, is the engine being asked whether the goal holds for all values of X, or for some value of X?

29.5. Using Horn clauses, write a definition of predicate sameParent(X, Y), which is true if individuals X and Y have at least one parent in common. (This indicates that X and Y are either (half) siblings or are the same individual.)

29.6. Using Horn clauses, write a definition of predicate greatGrandparent(X, Y), which is true if X is Y's great grandparent.

29.7. Using Horn clauses, write a definition of predicate cousinOrCloser(X, Y), which is true if X and Y are either cousins or siblings or the same individual. (Cousins share a common grandparent.)

29.8. Using Axioms (1–8), show a proof tree for goal grandparent(jordan, X). Where does the first success occur?

29.9. Using Axioms (1-10), show a proof tree for goal ancestor(A, cathy), showing all successes.

29.10. Can the ancestor predicate be used to find descendants? Using Axioms (1–10), show a proof tree for goal ancestor(sally, X), and show all of the branches that lead to success. Label each success by the value of X that it finds.

29.11. Using Axioms (1–8), show a proof tree for goal parent(A, B), showing all successes, not just the first one. Does it make sense to ask the engine to provide more than one value?

29.12. Using Axioms (1–8), show a proof tree for goal grandparent(X, Y), up to the point of the first success. What are X and Y? What does goal grandparent(X, Y) ask for?

29.8. Bibliographic notes

Colmerauer and Rousel [32] discusses the history of the development of logic programming and the Prolog programming language. See the next chapter for more notes.

Chapter 30

Understanding Logic Programming

30.1. Modes and information flow

If you think of a procedure in a procedural language, typically some of the parameters represent information coming into the procedure and some represent information coming out. For example, Ada procedure Sumdiff of Figure 16.5 has two in-parameters and two out-parameters. The procedure gets information from the in-parameters and stores information into the out-parameters.

Logic programming goals can have some parameters that are specific values and others that are unbound variables; the values represent information coming into the goal and the variables information that comes out (or that is left unknown). For example, in subgoal mother(marcy, X), the first parameter is coming into the mother predicate and the second parameter is coming out.

The *mode* of a goal indicates which parameters are inputs and which are outputs. In goal grandparent(X, cathy), for example, the first parameter is an output and the second is an input, so the mode is grandparent(out, in), but goal grandparent(cathy, X) has mode grandparent(in, out). It is possible that there are no outputs, as in the pure test grandparent(bertrand,cathy), (mode grandparent(in, in)) or even that there are no inputs, as in grandparent(X,Y) (mode grandparent(out, out)). The grandparent predicate can be used in four modes: to compute a grandparent of an individual; to compute a grandchild of an individual; to test whether one individual is a grandparent of another; or to provide pairs (X,Y) where X is a grandparent of Y. In procedural programming, the mode is a property of a procedure, and a given procedure can only be used in one mode. In logic programming, the mode is a property of the goal, and a predicate can often be used in more than one mode.

The multimodal nature of predicates reduces the number of different predicates that need to be written. For example, the ancestor predicate can be used to compute descendants. Imagine that predicate sum(A, B, C) is defined so that sum(A, B, C) is true if $C = A + B$; that predicate can be used to add, of course, but it can also be used to subtract. If X and Y are known, then to compute $Z = X - Y$, simply try to prove sum(Z, Y, X), that is, $Z + Y = X$.

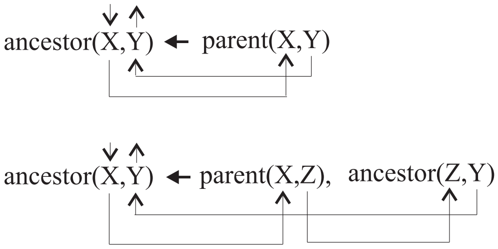

ancestor(X,Y) ← parent(X,Y)
ancestor(X,Y) ← parent(X,Z), ancestor(Z,Y)

Figure 30.1: **A definition of predicate ancestor(X,Y), meaning that X is an ancestor of Y.** The arrows show the flow of information in the case where ancestor is used in mode ancestor(in, out). In the second axiom, the parent predicate is used in mode parent(in, out) and the recursive call to ancestor is also used in mode ancestor(in, out).

Not all predicates can be used in every possible mode. For example, the sum predicate might reasonably be asked to solve equation $Z = X + Y$ for any one of the variables, given the other two. But if more than one of the variables is unknown, then it becomes much more difficult to give a good definition to sum.

Typically, when you define a predicate, you also indicate the modes in which it will work. The sum predicate has four modes, sum(in, in, out), sum(in, out, in), sum(out, in, in) and sum(in, in, in). That is, at least two of its parameters must be known, and sum can determine the third. Out-parameters can always be replaced by in-parameters, since it is always acceptable to have a little more information than you really need. For example, since the sum predicate allows mode sum(in, in, out), it must also allow mode sum(in, in, in), indicating that it is just checking whether the sum is correct. The grandparent predicate can handle any mode at all, so it suffices to say that its mode is grandparent(out, out).

You can understand predicate modes by drawing diagrams that show how information flows into and out of predicates during computation of a predicate in a given mode. Figure 30.1 shows a diagram for a definition of the ancestor predicate when it is being used in mode ancestor(in, out). Drawing the information flow arrows requires that you think about the order in which the literals will be evaluated. Imagine computing a goal using axiom

$$\text{ancestor}(X, Y) \Leftarrow \text{parent}(X, Z), \text{ancestor}(Z, Y)$$

in mode is ancestor(in, out). You start out knowing X. The next subgoal that is evaluated is parent(X, Z); here, X is known, so parent needs to find Z. Then subgoal ancestor(Z, Y) knows Z (provided by parent), and needs to find Y, which is the value produced by goal ancestor(X, Y).

The ancestor predicate can be defined in several ways and the information flow can depend strongly on which definition is chosen. Figure 30.2 shows an alternative definition of ancestor and shows the information flow for the same mode, ancestor(in, out), as for the previous definition. Now, the parent predicate is used in mode parent(out, out), forcing it to find all pairs (A, B) where A is a parent of B, and the recursive call to ancestor is used in mode ancestor(in, in), a pure test.

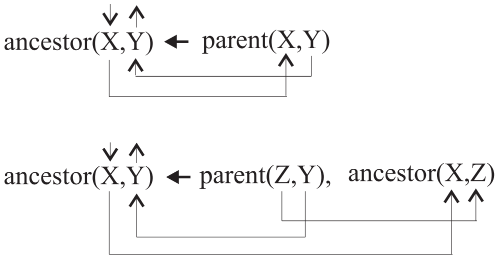

ancestor(X,Y) ← parent(X,Y)
ancestor(X,Y) ← parent(Z,Y), ancestor(X,Z)

Figure 30.2: This definition of the ancestor predicate is different from the definition in Figure 30.1. The arrows show the information flow for the same mode as the arrows of Figure 30.1, ancestor(in, out). But in this case, the recursive call is not used in the same mode. Notice that, in the second axiom, the parent predicate mode is parent(out, out), which requires that all parenthood relations be tried. This definition is much less efficient than the former one *in this mode*.

30.2. Understanding the meaning of a predicate

Every axiom in a logic program really has two meanings. In the *logical* interpretation, the axiom is taken as a statement of a fact about the predicate. But the axiom also makes a suggestion to the inference engine about how to use the axiom in proofs; it should replace the left-hand side (the head of a Horn clause) by the list of atoms on the right-hand side. So there is also a *procedural* interpretation of the axiom, where you think in terms of how the inference engine will use the axiom. If a predicate is intended to run in a restricted mode, you can find yourself tempted to think of the predicate and its axioms in procedural terms. For example, suppose that a list reversal predicate is intended to run in mode reverse(in, out). A procedural description of reverse is "reverse(X, Y) sets Y to the reversal of list X," since that is what the inference engine will do when it uses the reverse predicate in mode reverse(in, out).

But logic programming is not really intended to be procedural, and the semantics of a predicate should be logical, stated in terms of the values that make it true. When describing a predicate, think of its meaning in the mode where all parameters are inputs. For example, reverse(X, Y) is true when Y is the reversal of X (or, equivalently, when X is the reversal of Y). But the *mode of the description*, where all parameters are inputs, does not necessarily indicate the computational mode. The semantic description can be augmented by mode information, indicating how the predicate is intended to be used. A comment that the mode is reverse(in, out) suffices. Logic programming tries to separate logic from control, and you should maintain that separation in your descriptions.

30.3. Computing on data structures

The predicates that we have written so far have only been concerned with simple data values, such a symbols and numbers. But logic programming generally works on trees, including the special case of lists. A good example is the append predicate, where append(A, B, C) is true if list C is A ++ B, and ++ is list concatenation. For example, append($[2, 3]$, $[5, 7]$,

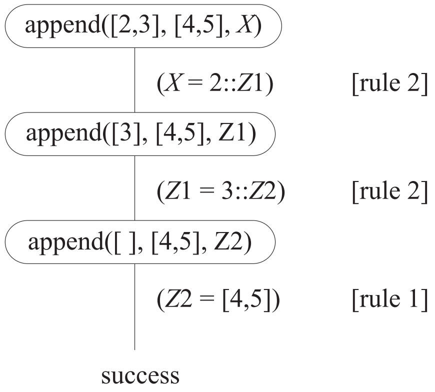


Figure 30.3: **Computation of goal append$([2,3], [4,5], X)$ using Axioms 30.3.1.** It is critical to this computation that, each time an axiom is used, a new copy of it, with new variables, is created. The first time Rule 2 is used, its variable Z is called $Z1$, and the second time it is called $Z2$.

$[2,3,5,7]$) is true. Axioms are as follows.

$$\begin{array}{l}\text{append}([\,],\ X,\ X).\\ \text{append}(H :: T,\ Y,\ H :: Z) \Leftarrow \text{append}(T,\ Y,\ Z).\end{array} \tag{30.3.1}$$

The first axiom states the obvious fact that $[\,] \mathbin{++} X = X$. The second axiom states that $(H :: T) \mathbin{++} Y = H :: (T \mathbin{++} Y)$, or, equivalently, if $Z = T \mathbin{++} Y$ then $H :: Z = (H :: T) \mathbin{++} Y$.

It should come as no surprise that the definition of append can be used to concatenate lists. Goal append$([2,3],\ [4,5],\ X)$ asks for X to be bound to $[2,3]$ ++ $[4,5]$, and the computation that leads to that is shown in Figure 30.3. Putting the bindings together, you see that $X = 2 :: Z1 = 2 :: 3 :: Z2 = 2 :: 3 :: [4,\ 5]$. That is, $X = [2,\ 3,\ 4,\ 5]$.

But append can be used in a variety of modes. For example, goal append$(X,\ [4,5],\ [2,3,4,5])$ yields result $X = [2,\ 3]$. That goal is not using the append predicate to concatenate lists, but instead is using it to pull a list apart. Figure 30.4 shows how the inference engine processes it. As expected, it binds $X = 2 :: T1 = 2 :: 3 :: T2 = 2 :: 3 :: [\,] = [2,\ 3]$.

Try computing goal append$([2,3],\ X,\ [2,3,4,5])$. You should find that X becomes bound to $[4,5]$. Goal append$(X, Y, [2,3,4,5])$ binds $X = [\,]$ and $Y = [2,\ 3,\ 4,\ 5]$, and reaches a success. If backtracked into because that solution does not satisfy some later subgoal, it binds $X = [2]$, $Y = [3,\ 4,\ 5]$ for its second solution. In all, it produces five solutions, giving all pairs X and Y such that $X \mathbin{++} Y = [2,\ 3,\ 4,\ 5]$. You can even use goals where all of the parameters are unknown; the append predicate works in mode append(out, out, out).

When used in conjunction with the ability of the inference engine to find solutions via unification, the append predicate is remarkably powerful. For example, suppose that you want to write a predicate member$(A,\ L)$ that is true just when A is a member of list L. You can write the member predicate using append.

$$\text{member}(A, L) \Leftarrow \text{append}(X,\ A :: Y,\ L). \tag{30.3.2}$$

Any time a variable occurs on the right-hand side of an axiom, but not on the left-hand side, you are asking the inference engine whether an appropriate value for that variable

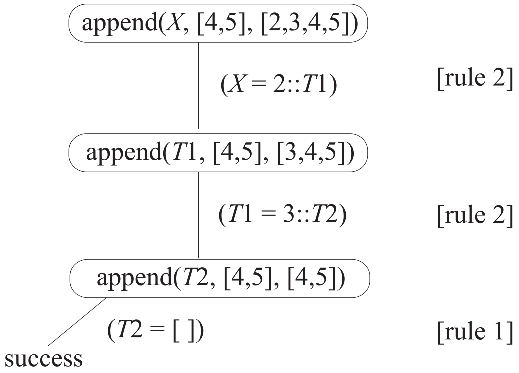


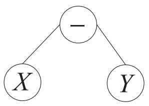
Figure 30.4: Computation of goal append(X, $[4, 5]$, $[2, 3, 4, 5]$) using Axioms 30.3.1.

exists. Axiom 30.3.2 says that A is a member of list L if there exist lists X and Y such that $L = X{+}{+}(A :: Y)$. It is the responsibility of the inference engine to find X and Y, and unification will lead to them relatively quickly. Compare this with the definition of member using pattern matching from Section 26.4.

Difference lists and more general modes

Unification is so versatile that it is not necessary for every parameter to be entirely input or entirely output, even in a particular subgoal. For example, a parameter that is a list might contain some members that are known and some that are unknown. A predicate can examine the known members (making them inputs to the predicate), bind some unknown members (making them outputs) and just leave some of the unknown parts unknown, leaving them for other predicates to bind. Also, when a predicate binds an unknown, it might bind it to another partially known data structure.

A *difference list* is a tool that makes it easier to understand some computations on lists. If X and Y are lists where Y is a suffix of X, define the difference list $X - Y$ to be the list obtained from X by removing suffix Y from it. For example, [2, 4, 6, 8, 10] − [8, 10]) = [2, 4, 6]. If Y is not a suffix of X, then $X - Y$ is undefined. We are usually not interested in computing difference lists, but in using them in symbolic form as a way of understanding what a predicate does. Remember that $X - Y$ stands for a tree



so writing $X - Y$ is just a convenient way of combining variables X and Y.

For example, suppose that L is a (known) list of characters, representing a string, and that you want to get a decimal number from the front of L. What you want to find is the longest prefix N of L consisting entirely of decimal digits. But there might be additional useful information in L after the initial number, and you want to get that too, so you need to get the suffix S of L, after the numeric prefix. If L is ['2', '6', '1', ' ', '3', '2'], then the

extractNumber$([\,] - [\,], [\,])$.
extractNumber$((H{::}T) - R, H{::}N) \Leftarrow$ digit(H), extractNumber$(T - R, N)$.
extractNumber$((H{::}T) - (H{::}T), [\,]) \Leftarrow$ nondigit(H).

Figure 30.5: **Definition of predicate extractNumber.** The semantics is that extractNumber$(L - S, N)$ is true if S is a suffix of L and $L - S = N$ and N is the longest prefix of L that consists entirely of decimal digits. The intended mode is extractNumber(in − out, out). The first parameter is partially input and partially output. Predicate digit(C) is presumed to be true when character C is a digit, and nondigit(C) is true when C is not a digit.

$$\begin{array}{rcl}
\langle\text{article}\rangle & ::= & \text{a} \mid \text{the} \\
\langle\text{propernoun}\rangle & ::= & \text{larry} \mid \text{curly} \mid \text{moe} \\
\langle\text{noun}\rangle & ::= & \text{parrot} \mid \text{kangaroo} \\
\langle\text{verb}\rangle & ::= & \text{meets} \mid \text{slaps} \\
\langle\text{nounphrase}\rangle & ::= & \langle\text{article}\rangle\ \langle\text{noun}\rangle \mid \langle\text{propernoun}\rangle \\
\langle\text{verbphrase}\rangle & ::= & \langle\text{verb}\rangle\ \langle\text{nounphrase}\rangle \\
\langle\text{sentence}\rangle & ::= & \langle\text{nounphrase}\rangle\ \langle\text{verbphrase}\rangle
\end{array}$$

Figure 30.6: **A grammar of simple sentences.**

decimal number at the beginning of L is $N =$ ['2', '6', '1'] (later to be converted to 261) and the rest of L is $S =$ [' ', '3', '2']. Notice that $L - S = N$.

Figure 30.5 defines predicate extractNumber, where the idea is that extractNumber$(L - S, N)$ should be true just when S is a suffix of L and $L - S = N$ and N is the longest prefix of L that consists entirely of decimal digits. In terms of mode, extractNumber is intended to be used as extractNumber(in − out, out). That is, extractNumber$(L - S, N)$ takes in L and produces the prefix N and suffix S. For example, extractNumber(['2', '3', 'x', 'y'] $- R$, K) binds $R =$ ['x', 'y'] and $K =$ ['2', '3'].

There are other ways to write extractNumber. For example, it works just as well to have three parameters (L, S and N) rather than combining L and S into one parameter, but the idea of a difference list helps you see that you are breaking a list into a prefix and suffix, and makes the semantic intent clear.

Using difference lists for parsing

Chapter 6 introduces context-free grammars, and Figure 30.6 shows a small context-free grammar describing very simple English sentences. Suppose that you want to be able to recognize lists, such as [the, kangaroo, meets, larry], that are sentences according to this grammar. A natural plan is to write a predicate sentence(L) that is true just when L is a sentence. But suppose that, instead of taking the list directly, the sentence predicate takes a difference list. Define sentence$(A - B)$ to be true if $A - B$ is a sentence (and, of course, B is a suffix of A so that $A - B$ is meaningful). Then the sentence predicate does more than simply recognize a sentence; it can be used to extract a prefix of a list that is a sentence.

article((a::R) − R).
article((the::R) − R).
propernoun((larry::R) − R).
propernoun((curly::R) − R).
propernoun((moe::R) − R).
noun((parrot::R) − R).
noun((kangaroo::R) − R).
verb((meets::R) − R).
verb((slaps::R) − R).
nounphrase($L - R$) ⇐ article($L - A$), noun($A - R$).
verbphrase($L - R$) ⇐ verb($L - A$), nounphrase($A - R$).
sentence($L - R$) ⇐ nounphrase($L - A$), verbphrase($A - R$).

Figure 30.7: **Implementation of the grammar of Figure 30.6.** Each predicate describes what it means for a difference list to be derivable from a given nonterminal. Upper case letters are variables and words that start with lower case letters are (constant) symbols. A grammar written in this style is called a *definite clause grammar.*

For each nonterminal N, you define a predicate $N(A - B)$ that is true just when list $A - B$ can be derived starting at nonterminal N. For example, in our small grammar, a sentence can be a noun phrase followed by a verb phrase, so, if $L - R$ is a sentence, then $L = X \mathbin{++} Y \mathbin{++} R$ where X is a nounphrase and Y is a verbphrase. But we need to express the noun phrase X and verb phrase Y as difference lists, since that is the form that our predicates expect. Since $L = X \mathbin{++} Y \mathbin{++} R$, list X can be defined as $X = L - (Y \mathbin{++} R)$. In logic programming, you generally need to give things names, so define $A = Y \mathbin{++} R$. Then the noun phrase X is $L - A$, and the verb phrase Y is $A - R$ (since $Y = (Y \mathbin{++} R) - R$). Expressing that X is a noun phrase and Y is a verb phrase yields the following axiom.

$$\text{sentence}(L - R) \Leftarrow \text{nounphrase}(L - A), \text{verbphrase}(A - R) \qquad (30.3.3)$$

Figure 30.7 shows axioms for the entire grammar. Goal sentence([larry, slaps, curly] − [ ]) should succeed.

30.4. Positive thinking

It is possible to define the member predicate directly, without using append. Here it is.

$$\begin{array}{lcl} \text{member}(A, A :: X) & & \\ \text{member}(A, B :: X) & \Leftarrow & \text{member}(A, X) \end{array} \qquad (30.4.1)$$

Compare that with an equational definition of member.

member(a, [ ]) `= false`
member(a, a :: x) `= true`
member(a, b :: x) `= member(`a`, `x`) when` $b \neq a$

Notice that the two cases of the logic programming definition correspond roughly to the last two cases of the equational definition. (Why is it possible to drop the proviso that

$b \neq a$?) There is no case that corresponds to the equation member(a, [ ]) = false. In logic programming, you say what is true. Anything that does not follow logically from what you have said is presumed to be false, a presumption that is captured in the *closed world hypothesis*.

> **Closed World Hypothesis.** The program says everything that is relevant. Anything that is not said, and that does not follow logically from what was said, is false.

As another example, consider the problem of telling whether a list is a palindrome, a list that is its own reversal. One approach is as follows. Notice that there is no negative information. Only the positive cases are stated.

$$\begin{array}{l}\text{palindrome}([\,]).\\ \text{palindrome}([A]).\\ \text{palindrome}(A{::}X) \Leftarrow \text{append}(Y,\ [A],\ X),\ \text{palindrome}(Y).\end{array}$$

The third fact states that, if Y is palindrome and $X = Y +\!+ [A]$, then $A :: X$ is a palindrome. That is, $A :: (Y +\!+ [A])$ is a palindrome.

30.5. Computing functions

In imperative or equational programming, list concatenation is normally provided as a function (such as operator ++) taking two parameters. In logic programming, concatenation is provided as a predicate (append) having three parameters, where an extra parameter is added for the result. You generally add an extra parameter any time you need to compute a function via a predicate. For example, to compute the N-th member of list L, you need a third parameter to hold the result, so you define predicate nth(N, L, R), which is true just when R is the N-th member of L. An exception is a function that produces a boolean result, since predicates naturally produce true or false already. For example, there is no need to add an extra parameter to the member predicate.

In a style that uses expressions, you set $a = x$ ++ y ++ z as follows.

```
Let a = x ++ (y ++ z).
```

The intermediate result y ++ z is implicitly passed to the next use of ++. But using predicates, you need to introduce a variable to stand for the intermediate result, and explicitly pass it from one predicate to the next. For example,

$$\begin{array}{l}\text{append(Y, Z, R),}\\ \text{append(X, R, A).}\end{array}$$

states that $R = Y +\!+ Z$ and $A = X +\!+ R$.

30.6. Summary of terminology and concepts

Predicates can be used in several different modes, allowing each goal to decide which parameters it needs to be inputs and which should be outputs. Information flow in a program is determined dynamically rather than being coded into the definitions of predicates. The source of this flexibility is the symmetry of unification.

Logic programming is a good tool for working with trees, allowing you to write versatile predicates that manipulate data structures.

Terminology

Closed world hypothesis The closed world hypothesis is the supposition that anything that a program does not say, and that does not follow logically from what was said, is false.

Definite clause grammar A definite clause grammar is a collection of axioms that expresses the requirements of a context-free grammar, and that can be used for parsing lists.

Difference list A difference list has the form $A - B$, where B is a suffix of A. It stands for the prefix of A obtained by removing suffix B from A. The difference $A - B$ is not typically computed from A and B. Rather a program typically solves an equation $A - X = C$, given A and some property that list C must have. It obtains C and X.

Mode The mode of a predicate indicates which of its parameters are inputs and which are outputs.

30.7. Exercises

30.1. In what modes can predicate member defined by Axiom (30.3.2) be used?

30.2. Draw an information flow diagram for the definition of append (Axioms 30.3.1) in mode append(in, in, out).

30.3. Draw an information flow diagram for the definition of append in mode append(in, out, in). Is the recursive call done in the same mode?

30.4. Show the information flow for the ancestor predicate in mode ancestor(out, in)

(a) using the definition of ancestor in Figure 30.1.

(b) using the definition of ancestor in Figure 30.2.

30.5. Show the information flow for Axiom (30.3.3) when used in mode sentence(in− out, out).

30.6. Write axioms for predicate even(L), which is true if list L has an even number of members. In mode even(out), it should bind its parameter to a list of unbound variables, first of length 0, then 2, then 4, etc.

30.7. Write axioms for predicate samelength(A, B), which is true when A and B are lists of the same length. Make it usable in mode samelength(out, out). If both parameters are unknown, samelength should bind each of them to a list of zero or more unknowns, of the same length. (Try length 0, then one, then two, etc.)

30.8. Write axioms for predicate allsame(L), which is true just when all members of list L are the same. For example, allsame($[5, 5, 5]$) is true. By definition, allsame($[\,]$) and allsame($[x]$) are true.

30.9. Write axioms for predicate prefix(X, Y), which is true if list X is a prefix of list Y. For example, prefix([a, b], [a, b, c, d]) is true. The empty list is a prefix of every list, and every list is a prefix of itself.

30.10. Write axioms for predicate suffix(X, Y), which is true if list X is a suffix of list Y. For example, suffix([c, d], [a, b, c, d]) is true. A list is a suffix of itself. For this definition, do not use recursion, but use the append predicate to do the work for you.

30.11. Convert the definition of naive reverse (Equation 19.3.1) to a logic programming style. Use append to perform concatenations.

30.12. Naive reverse is inefficient. Write a more efficient implementation of reverse, as a predicate, following the plan of Equations 19.3.2, 19.3.3 and 19.3.4.

30.13. Write a goal, using the append predicate, to bind X to the last element of list L, assuming that L is nonempty. Do not write any new predicate definitions. Just write a goal.

30.14. Using the goal from the preceding exercise, write a definition of predicate last(L, X) that is true if L is a nonempty list whose last member is X. Do not use recursion, just use the append predicate to do the job.

30.15. Write axioms for predicate twice(X, L), which is true if L has the form X ++ X. For example, twice([3, 5], [3, 5, 3, 5]) is true. It should be usable in mode twice(out, out).

30.16. Write a definition of predicate perm(A, B) that is true if list B is a permutation of list A. Determine the modes in which your definition can be used.

30.17. Show the computation tree for goal list (member(X, [3, 4, 5]), member(X, [4])), where X is an unbound variable, up to the point where the first success is found, using Axioms 30.4.1 as the definition of member.

30.18. Suppose that list membership is defined by Axioms 30.4.1. Can member be used in mode member(in, out) to produce a list that has a given value as a member? Show a computation of member(3, L) to illustrate. What are the first to results for L?

30.19. Suppose that the two axioms of the preceding exercise are written in the opposite order,

$$\begin{array}{l}\mathrm{member}(A, B :: X) \Leftarrow \mathrm{member}(A, X).\\ \mathrm{member}(A, A :: X).\end{array}$$

Can this definition of member be used in mode member(in, out)? Try computation of member(3, L). Does the order in which axioms are written matter?

30.20. Using the style described in Section 30.3, write a parser for expressions using grammar (6.6.2) in Chapter 6. For simplicity, presume that a number is a single digit.

30.8. Bibliographic notes

Hogger [56], Clocksin and Mellish [31] and Kowalski [65] explain principles and uses of logic programming.

Part X

Logic Programming in Prolog and Cinnameg

Chapter 31

Elementary Prolog

31.1. Introduction to Prolog

Prolog is a typeless programming language, or really a family of related programming languages, that is dedicated to logic programming. This chapter looks primarily at ISO Standard Prolog, simply called Prolog here. ISO Prolog has some differences from typical other dialects, and this chapter mentions a few of them.

Basic syntax

In Prolog, each axiom or goal must be followed by a period. The right-to-left implication $\Leftarrow$ is written `:-`. For example, an axiom defining the grandparent predicate is written as follows in Prolog.

```
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
```

Prolog is mostly free-form, so an axiom can extend over several lines, with a period used to indicate the end of the axiom.

In ISO Prolog, you write a comment inside /*. . . */. Many Prolog dialects use a comment form that begins with % and continues to the end of the line.

In ISO Prolog, symbols can be written in three forms: (1) a name, such as chihuahua, that starts with a lower case letter and can contain letters, digits and the underscore character; (2) a sequence of special characters, such as + and +@$, consisting of one or more special characters in string "`+-*/\^<>:.?@#$&`"; and (3) any sequence of characters enclosed in single quotes, such as 'the man in the moon'. Type (3) can only be constants or leaves of trees; types (1) and (2) can occur anywhere in a tree. Predicate names are required to have the same form as type (1) symbols. Different dialects of Prolog have different requirements on symbols. It is a good idea to restrict yourself to names that contain only letters, as well as a few short symbolic names such as + and −.

Prolog is case-sensitive. Unknowns are called *variables*, and variable names start with an upper case letter or underscore, and can contain letters, digits and the underscore character. So, while kangaroo is a symbol or predicate or function, Kangaroo is a variable. A single underscore is an anonymous, or don't-care, variable; each underscore is a different variable.

```
append([], X, X).
append([H|T], Y, [H|Z]) :- append(T, Y, Z).

member(A, [A|_]).
member(A, [_|X]) :- member(A, X).
```

Figure 31.1: **Axioms for the append and member predicates in Prolog notation.** Recall that append(X, Y, Z) is true if Z is list X followed by list Y, and member(X, L) is true if X is a member of list L. An underscore is an anonymous variable, and is commonly used for a variable that occurs only once in an axiom.

You should be able to put a space between any two tokens. Some Prolog dialects are picky about spaces in certain places. In particular, a space between a predicate name and the left parenthesis just after that name confuses some Prolog compilers.

Functors and operators

Functions do not perform computation in Prolog. They are only used to describe trees. For example, $g(X, a)$ indicates a tree with root labeled g, and two subtrees X and a. To emphasize the distinction between the Prolog view of functions, as tags in trees, and functions in most programming languages, which perform computations, Prolog uses the term *functor*. So $g(X, a)$ is a tree with functor g.

Prolog allows you to use binary operators such as $+$, $-$, $*$ and /, but $X + Y$ has the same meaning as $+(X, Y)$, which describes a tree with root labeled + and two subtrees, X and Y.

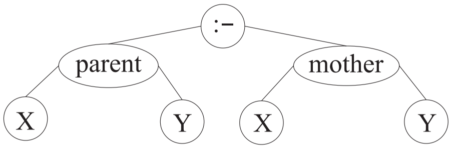
Lists

The empty list is written [ ]. The Cinnameg :: operator is written as a dot in Prolog, and you can write $.(X, Y)$ for the list whose head is X and whose tail is Y. Lists are special cases of trees, and $.(X, Y)$ is just a tree whose root is labeled by a dot, with left subtree X and right subtree Y.

Prolog provides a notation, $[H|T]$, that has the same meaning as $.(H, T)$, and that is the notation that we will use. (The bar notation is more universal among different dialects than the dot notation, so it is a good choice.) You must use brackets with this notation; Prolog does not allow you to write $H|T$ without brackets around it. Figure 31.1 shows definitions of the append and member testing predicates. An extension of the list notation is $[A\,B,\ C \mid R]$, which is the list that begins $[A,\ B,\ C, \ldots]$, with the rest $(\ldots)$ being R. So $[A,\ B,\ C \mid R]$ stands for $.(A, .(B, .(C, R)))$. You can list any number of members before the bar, but just one (the rest of the list) after the bar.

31.2. Prolog implementations

Prolog implementations come in three forms. One is an interpreter that works on a direct description of the axioms. The axioms are read and converted into a tree structure, and the interpreter works on the trees. For example, axiom parent(X,Y) :- mother(X,Y) is represented as follows by the interpreter.



A second kind of implementation translates the axioms into instructions of an abstract machine, and uses an interpreter to run the abstract machine. So instead of seeing an axiom, such as append($[\,]$, X, X) for the append predicate, the interpreter is given instructions that, in conceptual form, are something like the code shown in Figure 31.2. Basic operations such as unification and operations on the run-time stack and the control stack, as well as handling of the trail, are built into the implementation of the abstract machine.

A third kind of implementation is a compiler that compiles to native (machine) code, or to some language such as C that can be compiled into native code. We will restrict attention here to implementations that employ interpreters.

Interaction with the Prolog interpreter

Prolog systems typically interact with the user, asking for goals. A goal begins with ?-. So, to ask whether list [a, b, c] contains b, you type goal

```
?- member(b, [a,b,c]).
```

The system responds `yes`. Notice that a, b and c are symbols, not variables, since they begin with lower case letters.

When a goal contains one or more variables, the system responds with the values of the variables that make the goal true. A semicolon typed by the user asks for yet another solution. It forces backtracking by causing an artificial failure. An interaction with a Prolog system might go as follows. The goal and the semicolons are typed by the user. The values of X are written by the interpreter.

```
?- member(X, [a,b,c]).
X = a;
X = b;
X = c;
no
```

You will usually want to load axioms into the interpreter from a file. Pseudo-goal

```
consult('myfile.pl')
```

loads the axioms found in file myfile.pl. Many Prolog implementations allow a goal to be a list of symbols. All of the files named in the list are consulted. Also, some versions of Prolog require file names to be strings enclosed in double quotes rather than symbols.

31.3. Unification

Chapter 28 discusses two forms of unification, one with an *occur check*, the other without. Most dialects of Prolog do not employ the occur check, preferring naive unification because the occur check is costly to perform. But omitting the occur check allows a program to

```
PUSH-MARK-FRAME
PUSH-BACKTRACK-FRAME (pointing to label 1)
V1 = new variable (X)
UNIFY param1, [ ]
UNIFY param2, V1
UNIFY param3, V1
POP-MARK-FRAME
RETURN
LABEL 1
V1 = new variable (H)
V2 = new variable (T)
V3 = new variable (Y)
V4 = new variable (Z)
V5 = new variable (T)
V6 = V1.V2
V7 = V1.V4
UNIFY param1, V6
UNIFY param2, V3
UNIFY param3, V7
CALL append(V5,V3,V4)
POP-MARK-FRAME
RETURN
```

Figure 31.2: **A translation of the definition of append into a generic abstract machine language.** The backtrack frame will start the second axiom if the first fails, due to failed unifications. The marker frame is used to limit how far back a cut goes. You can see that the call to append in the second axiom is tail-recursive, since, ignoring the marker frame pop, it is immediately followed by RETURN, so an efficient call instruction that will only work for tail-recursive calls could be used instead of a normal call. The last three instructions (starting at the call) can be replaced by

```
POP-MARK-FRAME
TAIL-CALL append(V5,V3,V4)
```

The POP-MARK-FRAME needs to be done before the tail-call because TAIL-CALL will not return; it destroys the frame for this predicate, so it has nothing to return to.

create a cyclic structure, which represents an infinite tree, as shown in Chapter 28. ISO Prolog requires the occur check to be performed, and disallows cyclic structures.

Goal $X = Y$ is performed by unifying X and Y. You could write your own predicate eq(X, Y) to perform the same action, by writing a single axiom, `eq(X,X)`, and writing eq(A, B) instead of $A = B$.

31.4. Disjunction

On the right-hand side of an axiom, a comma indicates conjunction; to say that you have proved the right-hand side, you need to prove all of the goals that are listed. Prolog also provides a disjunction operator, a semicolon, indicating an *or* of parts. For example,

```
parent(X, Y) :- mother(X, Y) ; father(X, Y)
```

says that parent(X, Y) is true when either mother(X, Y) is true or father(X, Y) is true. Prolog implementations deal with semicolons by translating to more than one axiom. The axiom above is equivalent to

```
parent(X,Y) :- mother(X,Y)
parent(X,Y) :- father(X,Y)
```

31.5. Definite clause grammars

Section 30.3 describes definite clause grammars, which use logic programming with difference lists to parse sequences of words according to a context-free grammar. Some Prolog dialects have a special notation that makes writing definite clause grammars easy and convenient. ISO Prolog provides notation

```
sentence --> nounphrase, verbphrase.
```

which stands for

```
sentence(A, R) :- nounphrase(A, B), verbphrase(B, R).
```

No explicit difference list is built, but goal sentence(A, R) is thought of as meaning that $A - R$ is a sentence. There can be any number of parts on the right-hand side of `-->`; they are chained together, letting each extract a prefix of a list and pass the remainder to the next predicate. There is also a notation for rules that produce fixed words. Notation

```
noun --> [horse].
```

stands for

```
noun([horse|X], X).
```

and fixed parts (in brackets) can be mixed with nonterminals in rules, as shown in Figure 31.3.

Sometimes you need to add more parameters to the parsing predicates. For example, rather than merely checking whether a list is a sentence, you might like to produce a parse tree of a sentence. Prolog's grammar notation allows you to add parameters, which are just added to the parsing (difference list) parameters. For example, rule

```
expression(e(T))     --> term(T).
expression(e(T,+,E)) --> term(T), [+], expression(E).
term(t(N))           --> num(N).
term(t(N,*,T))       --> num(N), [*], term(T).
num(N)               --> [N], {number(N)}.
```

Figure 31.3: **A parser for simple expressions.** Goal expression(T, L, R) succeeds if difference list $L - R$ is an expression according to this grammar and T is a parse tree for expression $L - R$. Parameters L and R are not written explicitly. They are added when these grammar rules are translated into axioms. Goal expression(T,L,R) normally takes L in and produces T and R. For example, goal expression(T, [2,+,3,*,4], [ ]) succeeds, binding T to $e(t(2), +, e(t(3, *, t(4))))$. Standard predicate number(N) succeeds when N is a number, and {number(N)} is just a test ensuring that N is a number.

```
sentence(sent(NP,VP)) --> nounphrase(NP), verbphrase(VP).
```

is translated to

```
sentence(sent(NP,VP), A, R) :- nounphrase(NP, A, B),
                               verbphrase(VP, B, R).
```

Additionally, you can add tests that are not part of the grammar rule by enclosing parts in braces; those parts are copied directly into the translation, without change, and without affecting the difference-list parameters. Figure 31.3 shows a Prolog parser for expressions that employs all of these features.

31.6. Reading and writing

In ISO Prolog, a character is just a symbol whose name has length 1. For example, symbol x is thought of also as the character x. Some dialects just use character codes (integers), and dispense with the notion of a character.

Goal read(X) reads a Prolog term from the standard input, and unifies X with that term. To read just one character, use get_char(X). (In some dialects, goal get0(X) binds X to the next character code, an integer.)

You can write a term X to the standard output using write(X). Goal put_char(X) writes character X. (Some Prolog dialects use put(N) instead of put_char(X), where N is an integer code.) Goal nl ends a line on the standard output.

31.7. Summary of terminology and concepts

Prolog is a dedicated logic programming language. Programs are collections of axioms. A Prolog interpreter performs inferences based on the axioms, in the style that was discussed in prior chapters.

ISO Prolog departs from most dialects of Prolog in a few important ways. The names of some of the standard predicates have been changed, and unification employs the occur check.

Prolog notation and terminology

`:-` `:-` is the backwards implication operator, $\Leftarrow$.

`?-` An interpreter typically prompts the user with `?-`, indicating that the user should type a goal.

`;` A semicolon indicates *or* on the right-hand side of an axiom.

`-->` `-->` is used to ask the Prolog compiler to write an axiom for you using the ideas of definite clause grammars.

$|$ Prolog notation $[H|T]$ is equivalent to $.(H,T)$, or to the Cinnameg notation $H :: T$, the list whose head is H and whose tail is T.

Functor Functions are called functors in Prolog. Functors are simply labels of nodes in trees.

Variable A variable is, at least initially, unknown. It can become bound either immediately after creation or much later during evaluation of a program.

31.8. Exercises

31.1. Write a Prolog definition of predicate prefix(X, Y), which is true if list X is a prefix of list Y. X is a prefix of Y if there exists a list Z such that $X \mathbin{++} Y = Z$, where $++$ is concatenation. (A list is a prefix of itself.)

31.2. Write a Prolog definition of predicate suffix(X, Y), which is true if list X is a suffix of list Y.

31.3. Write a Prolog definition of predicate csublist(X, Y), which is true if list X is a contiguous sublist of list Y, that is, if there exist lists A and B such that $A \mathbin{++} X \mathbin{++} B = Y$.

31.4. Write a Prolog definition of predicate sublist(X, Y), which is true if list X is a (not necessarily contiguous) sublist of list Y, that is, if it is possible to obtain list X by removing zero or more members of list Y. For example, sublist([1,3,5], [1,2,3,4,5,6,7]) is true. Order must be preserved, so sublist([1,2], [3,2,1]) is false.

31.5. (Requires elementary calculus) Think about how to represent expressions in Prolog. Allow the expressions to use addition, subtraction, multiplication, the symbol x, and numbers. Write a predicate derivative(E, D) that is true if D is the derivative of E with respect to x. Your definition should be usable at least in the mode derivative(in, out). Does it work in other modes? Is derivative(2*x, 2) true? Be careful.

31.6. Write a program to solve the n-queens problem (Exercise 26.3) in Prolog.

31.7. Write a definite clause grammar using ISO Prolog to determine whether a list is a sentence according to the grammar in Figure 30.6.

31.8. Modify your solution to the preceding problem so that your parser produces a parse tree of a sentence.

31.9. Definite clause grammars allow you to perform computations during the process of parsing. Typically, those computations handle issues that are difficult to deal with in a context-free grammar. Without changing the underlying context-free grammar, write an ISO Prolog definite clause grammar for the grammar of Figure 30.6, but ensure that any sentence that uses the word **meets** is not allowed to have identical subject and object. For example, [moe, meets, moe] should not be an acceptable sentence.

31.9. Bibliographic notes

Clocksin and Mellish [31] describe ISO Prolog. Colmerauer and Roussel [32] give a history of Prolog. Marriott and Stuckey [74] describe languages that extend Prolog by allowing more kinds of relationships among values than equality (unification).

Warren [107] describes an abstract machine suitable for use in Prolog implementations.

Chapter 32

Practical Issues

32.1. Arithmetic in Prolog

Arithmetic creates a problem for Prolog. All computation is done by predicates, not by functors. Goal `2 + 3 = 3 + 2` fails, since 2+3 and 3+2 are different trees. Prolog deals with arithmetic by introducing a special operator, **is**, where goal A **is** B is performed by evaluating B using the usual rules of arithmetic and then unifying A with the result. For example, goal list

```
X is 2 + 3, Y is 3 + 2, X = Y.
```

succeeds, since X and Y are each bound to 5. Operators that **is** understands include $+$, $-$, $*$, / (real division), // (integer division) and mod (integer remainder). Be careful; goal (2 + 3 is 3 + 2) still fails, since only the right-hand side, $3+2$, is evaluated by the **is** operator.

Goal (K is X + Y) causes an error if either of X or Y is unknown, since arithmetic cannot be performed on unknown numbers. So you need to be careful about mode when using the **is** operator. For example, goal ($X = 3$, $Y = 4$, 7 is $X{+}Y$) succeeds, but evaluating goal (7 is $X{+}Y$, $X = 3$, $Y = 4$) causes an error, since X and Y are not known at the point where arithmetic is requested.

You can compare two (known) numbers using operators =:= (equality), =\= (inequality), <, >, =< and >=. (Notice that Prolog uses =<, not <= for a less-than-or-equal test.) For example, goal $3 < 6$ succeeds.

Example

Figure 32.1 shows a definition of a factorial predicate in Prolog, where factorial(N, Y) is true if $Y = N!$. Goal factorial(3, N) binds $N = 6$. But goal factorial(M, 6) yields an error, since the > operation is used with an unbound variable. Prolog is obviously not well-suited to performing arithmetic calculations. It is designed for symbolic computing.

Notice that the second axiom for factorial explicitly states that X must be greater than 0. Is that really required? You know that the axioms are tried in the order written, so you might argue that if the first parameter is 0, then the first axiom will be taken. But in a logic program, *all* of the axioms are tried, by backtracking, and more than one can succeed. For example, what if you compute goal list (factorial(0, N), $N = 12$)? Clearly, that goal must fail. Initially, factorial binds $N = 1$, which leads unification $N = 12$ to fail. But

```
factorial(0,1).
factorial(X,Y) :-
  X > 0,
  N is X-1,
  factorial(N,M),
  Y is X*M.
```

Figure 32.1: **Definition of factorial in Prolog.**

```
take([], _, []).
take([i|N], [H|T], [H|R]) :- take(N, T, R).
```

Figure 32.2: **Definition of predicate take using jailbird notation.** The meaning is that take(N, X, Y) is true if list Y is the length N prefix of X. Number N is represented as a list for versatility, allowing take to be used in different modes. If N is larger than the length of X, then take(N, X, Y) fails. It is no surprise that take([i, i], [a, b, c, d], L) binds L to [a, b]. But this version of take will also handle goal take(N, [a, b, c, d], [a, b]), where N is unknown; it binds N to [i, i].

then factorial is backtracked into, and the second factorial axiom is tried, asking factorial whether it wants to produce yet another answer. Without the test that $X > 0$, that leads to computation of the factorial of -1, and an infinite recursion.

Using jailbird notation

If you only need small nonnegative integers, you can use a representation that exploits Prolog's data structure capabilities to get more versatile operations on numbers. Instead of using the built-in numbers, use a list of length n to stand for the number n. For example, list [i, i, i] stands for 3. The advantage is that you can perform arithmetic and comparisons on unknown numbers. To add two numbers, just concatenate them, using the append predicate. To subtract, just use the append predicate in a different mode. For example, saying that $X = Y - Z$ is equivalent to saying that $Y = X + Z$, which is expressed as append(X, Z, Y). That will only succeed when the difference $Y - Z$ is nonnegative, since lists have nonnegative length. You can express $X + 1$ as [i |X]. To compare numbers, compare the lengths of lists. For example, define ge(X, Y) to be true when X is at least as long as Y, as follows.

```
ge(_,[]).
ge([i|A], [i|B]) :- ge(A,B).
```

Figure 32.2 shows an implementation of the take function, which takes a prefix having a given length from a list, as a predicate. Because numbers are represented in jailbird notation, the take predicate can be used in several modes. Had usual arithmetic been used, the mode would have been restricted.

```
factorial(0,Y) :- !, Y = 1.
factorial(X,Y) :-
  N is X-1,
  factorial(N,M),
  Y is X*M.
```

Figure 32.3: **An alternative Prolog definition of the factorial predicate.** Instead of testing whether $X > 0$ in the second axiom, the first axiom cuts away the branch that would have tried the second one, ensuring that the second axiom will not be tried for $X = 0$. (This definition assumes that X is not negative.)

32.2. Pruning the proof tree using cuts

The backtrack control mechanism is highly demanding of memory, and practical Prolog programs usually need to employ commits (Section 27.4) to destroy control frames. Prolog calls a commit a *cut*, and uses an exclamation point to indicate a cut; the cut removes all backtrack frames back to one indicated by the topmost marker frame. As Figure 31.2 shows, a marker frame is pushed each time evaluation of a predicate begins.

The factorial predicate defined in Figure 32.1 requires an explicit test $X > 0$ in its second axiom to prevent that axiom from being used when only the first axiom should apply. Alternatively, you can omit the test by using a cut to remove the branch that would try the second axiom, as shown in Figure 32.3.

If-then-else

Cuts allow you to simulate an if-then-else construct. To make $P(X)$ simulate **if** $A(X)$ **then** $B(X)$ **else** $C(X)$, define $P(X)$ as follows.

```
P(X) :- A(X), !, B(X).
P(X) :- C(X).
```

When A(X) succeeds, the cut removes that branch that would have done C(X).

Cuts limit mode

The membership testing predicate can also, at least apparently, benefit from a cut. Once a value x is found in a list, there appears to be no point in looking for another copy of it, since you only want to know whether there are any occurrences of x. Axioms

```
member(A, [A|_]) :- !.
member(A, [_|X]) :- member(A, X).
```

accomplish that. That works well in mode member(in, in). But what if it is used to generate members of a list, in mode is member(out, in), as in the following goal?

member(X, [1,2,3,4])
$X = 2$

If the cut were not present, that goal would succeed and bind $X = 2$. But the presence of the cut means that, after binding X to 1, the second member axiom will not be tried. So the possibility that $X = 2$ will never be tried, and the goal fails.

As a rule, using a cut limits the mode in which a predicate can be used, so be cautious. When justifying the use of a cut, always consider the modes in which you want your predicate to work.

32.3. Negation

Suppose that you want to write a predicate sibling(X,Y) that is true when X and Y are (half) siblings, with the intended mode sibling(in, in). A first attempt is as follows.

```
sibling(X,Y) :- parent(Z,X), parent(Z,Y).
```

Unfortunately, according to this definition, everybody who has a parent is his or her own sibling, since there is nothing to prevent X and Y from being the same individual. Somehow, you need to say that $X \neq Y$. Unification can be used to state that two variables are equal, but not to state that they are not equal. Remember that, when a Horn clause is written in the usual form, as an implication, all literals are positive, and there is no provision for saying that anything is false. Let's try adding a special case to handle equality, explicitly saying that sibling(X, X) is false. Standard predicate **fail** takes no arguments and always fails.

```
sibling(X,X) :- fail.
sibling(X,Y) :- parent(Z,X), parent(Z,Y).
```

But that does not do the job; the first axiom always fails, and the second axiom is tried anyway. A cut will prevent the second axiom from being tried.

```
sibling(X,X) :- !, fail.
sibling(X,Y) :- parent(Z,X), parent(Z,Y).
```

The cut effectively implements an inequality test.

Negation as failure

In general, if you have any predicate p(x), you can define another predicate notp(x), where notp(x) is true just when p(x) is false, as follows.

```
notp(X) :- p(X), !, fail.
notp(X).
```

If p(X) succeeds then notp(X) fails by the first rule. It also cuts away backtracking so that the second rule cannot make notp(X) succeed. But if p(X) fails, then the cut in the first rule will not be done, and the second rule will be backtracked into, causing notp(X) to succeed. This approach to simulating negation is called *negation as failure*.

ISO Prolog provides negation as failure as a unary operator, \+; goal \+G succeeds just when an attempt to prove goal G fails. So sibling(X, Y) can be defined as follows.

```
sibling(X,Y) :- parent(Z,X), parent(Z,Y), \+(X = Y).
```

Some other dialects of Prolog offer the same thing as a pseudo-predicate not(A), and the sibling predicate in those dialects has the following form.

```
sibling(X,Y) :- parent(Z,X), parent(Z,Y), not(X = Y).
```

Negation and mode

You should be wary of negation as failure. It is implemented using a cut, and cuts usually limit the mode in which a predicate works. Consider the following goal.

```
?- \+(X = 2).
```

This goal contains a variable X, so it appears to ask for some value X such that $X \neq 2$. Of course, there are many such values X. But the Prolog system will not find any of them. First, it tries to prove $X = 2$. That succeeds, binding X to 2. But because goal $X = 2$ succeeds, goal \+($X = 2$) fails! The \+ operator is only a correct implementation of the concept of negation if no variables (that exist when \+ starts) are bound during its computation. Goal list ($X = 4$, \+($X = 2$)) succeeds, with $X = 4$, since X was already bound when the negated goal was tried. But goal list (\+($X = 2$), $X = 4$) fails, because X is unbound when the negated goal is tried.

If you think about what negation as failure does when variables become bound, you can see that it is performing a kind of nonexistence test. If X is unbound when goal \+(X $= 2$) is tried, then this goal is asking whether it is true that there does not exist an X such that $X = 2$. Of course, that is false, since there does exist an X such that $X = 2$, so goal \+($X = 2$) fails. You can exploit this behavior of \+ to perform existence tests. Goal \+ \+(A) is true if there exist values of the currently unbound variables that make A true. The difference between \+ \+(A) and just A is that bindings done while testing \+ \+(A) are undone before proceeding.

32.4. An example

Section 26.3 illustrates backtracking by solving a planning problem where a farmer needs to cross a river with his goat, wolf and cabbage. This section redoes that example using Prolog.

One issue is representation of sets. We store a set as a list of its members. But it is important to keep the lists in a standard order, since otherwise it will be awkward to tell whether two sets are the same. The four things (cabbage, farmer, goat and wolf) are represented as symbols, and sets are stored in alphabetical order. Figure 32.4 shows operations on sets.

Next, as in Section 26.3, we need a way of representing a state, and a way of testing whether a state is admissible. A good choice for a state is a tree of the form state(L,R) where L is a set of the things on the left bank and R is the set of things on the right bank. Figure 32.5 shows operations on states.

Predicate 'follows' of Figure 32.6 chooses a state to go to next, and Figure 32.7 shows predicates that find a solution to the problem as a list of states.

32.5. Modifying a Prolog program on the fly

Prolog allows you to alter the axioms as the program runs. To add a positive literal C to the program, as a new axiom, perform goal asserta(C) (to add C as the first axiom) or assertz(C) (to add it as the last axiom). For example, asserta(animal(kangaroo)) adds axiom animal(kangaroo) to the program. You can remove C by performing goal retract(C). When C contains variables, retract(C) retracts all axioms that can be unified with C. The

less(X, Y) is true if X is alphabetically before Y.
Mode: less(in,in).

```
less(cabbage, farmer).   less(farmer, goat).
less(cabbage, goat).     less(farmer, wolf).
less(cabbage, wolf).     less(goat, wolf).
```

greater(X, Y) is true if X is alphabetically after Y.
Mode: greater(in,in).

```
greater(X,Y) :- \+(X = Y), \+(less(X,Y)).
```

add(XS, Y, ZS) is true if $ZS = XS \cup \{Y\}$.
Mode: add(in,in,out).

```
add([], X, [X]).
add([X|Xs], X, [X|Xs]).
add([X|Xs], Y, [Y,X|Xs]) :- less(Y,X).
add([X|Xs], Y, [X|R]) :- greater(Y,X), add(Xs, Y, R).
```

remove(XS, Y, ZS) is true if $ZS = XS - \{Y\}$.
Mode: remove(in,in,out).

```
remove([], _, []).
remove([X|Xs], X, Xs).
remove([X|Xs], Y, [X|Xs]) :- less(Y,X).
remove([X|Xs], Y, [X|R]) :- greater(Y,X), remove(Xs, Y, R).
```

Figure 32.4: **Operations on sets.** Sets are assumed to be lists in alphabetical order. Since we plan to use cuts and negation, the intended modes of predicates are important. Predicate less is defined directly, but in ISO Prolog you can also use operator @< to compare two symbols. For example, cabbage @< farmer succeeds.

admissibleBank(Bank) is true if nothing is being eaten in Bank.
Mode: admissibleBank(in).

```
admissibleBank(Bank) :-
  member(farmer, Bank), !.
admissibleBank(Bank) :-
  \+(member(goat, Bank)), !.
admissibleBank(Bank) :-
  \+(member(wolf,Bank)),
  \+(member(cabbage,Bank)).
```

admissible(S) is true if nothing is being eaten on either bank in state S.
Mode: admissible(in).

```
admissible(state(LeftBank,RightBank)) :-
  admissibleBank(LeftBank),
  admissibleBank(RightBank).
```

Figure 32.5: **Testing whether a state is admissible.** The cuts in the definition of admissibleBank are important. There are three ways that a bank can be admissible, and we do not want to try a given state three times just because of that.

follows(S, T) is true if state T can immediately follow state S in a solution.
Mode: follows(in,out).

```
follows(state(Left1, Right1), state(Left2, Right2)) :-
  member(farmer, Left1),
  member(Possession, Left1),
  remove(Left1, farmer, A),
  remove(A, Possession, Left2),
  add(Right1, farmer, B),
  add(B, Possession, Right2).
follows(state(Left1, Right1), state(Left2, Right2)) :-
  member(farmer, Right1),
  member(Possession, Right1),
  remove(Right1, farmer, A),
  remove(A, Possession, Right2),
  add(Left1, farmer, B),
  add(B, Possession, Left2).
```

Figure 32.6: **Finding a following state.** This is similar to the next function in Section 26.3, but it is a predicate.

lengthen(Sol, Longer) is true if Longer = [S | Sol] where S is an admissible state that does not belong to Sol and that can follow the head of Sol. That is, Longer is a slightly longer partial solution.
Mode: lengthen(in,out).

```
lengthen([St | Rest], [NewSt, St | Rest]) :-
  follows(St, NewSt),
  admissible(NewSt),
  \+(member(NewSt, [St | Rest])).
```

A list of states L is a partial solution if, (1) no state occurs twice in L, (2) each state in L is admissible, and (3) If $L = [\ldots, S, T, \ldots]$, then follows($T$, S) is true. (So the state sequence is backwards.) Suppose that Partial is a partial solution. Then solution(Partial, G, Sol) is true if Sol = A ++ Partial for some list A, Sol is also a partial solution, and the head of Sol is G.
Mode: solution(in, in, out)

```
solution([Goal | Rest], Goal, [Goal | Rest]).
solution(PartialSolution, Goal, FullSolution) :-
  lengthen(PartialSolution, LongerSolution),
  solution(LongerSolution, Goal, FullSolution).
```

```
showSolution([]).
showSolution([state(Left, Right) | Rest]) :-
  write(Left), write(' '), write(Right), nl,
  showSolution(Rest).
```

```
run :-
  EmptyBank = [],
  FullBank = [cabbage, farmer, goat, wolf],
  FirstState = state(FullBank, EmptyBank),
  GoalState = state(EmptyBank, FullBank),
  solution([FirstState], GoalState, ASolution),
  reverse(ASolution, FwdSolution),
  showSolution(FwdSolution),
  nl.
```

Figure 32.7: **Solution of the Farmer-Wolf-Goat-Cabbage problem.** This is similar to the direct backtracking approach in Section 26.3.

```
dynamic seed/1.
seed(25).

random(N) :-
  seed(N),
  retract(seed(N)),
  R is (91*N + 3) mod 4096,
  asserta(seet(R)).
```

Figure 32.8: **A Prolog predicate random(N)**, which sets N to a pseudo-random number from 0 to 4095. It uses the seed predicate to simulate a global variable.

actions of asserts and retracts are not undone by backtracking; no information about them is written into trail frames.

One use of asserts is to add *lemmas* during computation. If a difficult goal is proved, it might be added to the axioms so that it will not need to be proved again. That use of asserts is quite benign and reasonable.

Asserts and retracts can also be used, however, to treat the program itself as a shared data structure through which a program can keep track of where it is, or to allow one part of a program to communicate with another part. As you can imagine, unfettered use of the capability to change a program while the program runs can lead to chaos. Therefore, ISO Prolog does not allow you to modify just any predicate on-the-fly. Before you try to add or remove an axiom for predicate p, you need to perform pseudo-goal `dynamic` p/n, where n is the number of parameters that p takes.

Figure 32.8 shows a simple pseudo-random number generator. It wants to update a global variable each time a new number is requested. But, since Prolog does not support global variables, it uses predicate seed to simulate one, by having just one axiom, seed(n), when the value of the seed is n. Goal seed(N) binds N to the current value of the seed by matching that axiom. Changing the seed value requires retracting the old axiom and asserting a new one.

A pseudo-random number generator is a reasonable application of a global variable. But you can imagine other uses. Figure 32.9 shows an example of a predicate called printnums that uses asserts and retracts to treat predicate count as a global variable. In an imperative program you are encouraged to avoid global variables. Using the program itself to simulate global variables can only be worse, and, as expected, the definition of printnums is very difficult to understand. Try working out a more principled definition of printnums.

32.6. Summary of terminology and concepts

Arithmetic in Prolog is performed by the **is** operator. Since expressions evaluated using **is** cannot contain variables, use of **is** limits the modes in which predicates can be used.

Commits are called cuts in Prolog. Using cuts requires careful thought, but you often cannot avoid them.

Logic programs only express positive facts. Negative facts are implicit by their absence. Since you write only positive literals and positive facts, it can be difficult to express a requirement that a particular formula must be false. But negation as failure simulates

```
dynamic count/1.
printnums(N) :-
  asserta(count(1)),
  repeat,
  body,
  count(N),
  !.
body :-
  count(K),
  write(K), nl,
  next is K+1,
  retract(count(K)),
  asserta(count(next)).
repeat.
repeat :- repeat.
```

Figure 32.9: **A Prolog predicate printnums(N), which prints the numbers from 1 to N, one per line.** It uses asserts and retracts to treat the count predicate as a global variable that can be modified. Predicates body and repeat have no parameters. Repeat acts as a springboard; each time computation backtracks into it, repeat pushes it back forward again. Each iteration or printnums encounters goal count(N), which fails (back to repeat) if the count is not yet N. After successfully completing the loop, the cut removes the backtrack control frame that was created by repeat. This is intended to show what you *can* do, not to advocate this as a good way of programming in Prolog. (The repeat predicate is actually built into ISO Prolog, but its definition is shown for clarity.)

negation and can be used as long as no old variables are bound during the course of the negated computation.

Notation and Terminology

`!` An exclamation point in a Prolog program indicates a cut.

\+ \+ is the negation operator in ISO Prolog, implemented using negation as failure. Most dialects of Prolog use not(A) for the negation of A.

Assert You can add axioms while a program is running using asserta and assertz. You can remove axioms using retract.

Negation as failure When using negation as failure, you demonstrate that an atomic formula A is false by trying to prove A, and concluding that $\neg A$ is true when the proof cannot be done. This only works correctly when no variables are bound that existed when you started to work on goal A.

Retract See assert.

32.7. Exercises

32.1. Write a Prolog definition of predicate pow(X, N, P), which is true when $X^N = P$, and which is intended to be used in mode pow(in, in, out). Assume that N is a positive integer.

32.2. Write a predicate sort(A, B) that is true when B is the result of sorting list A into ascending order. Assume that A and B are lists of numbers. The mode is sort(in, out).

32.3. Show the information flow in the axioms for factorial shown in Figure 32.1, in the mode factorial(in, out).

32.4. Show the information flow in the first axiom of follows in Figure 32.6.

32.5. Suppose that the first axiom for factorial in Figure 32.3 is replaced by

```
factorial(0,1) :- !.
```

without changing the second axiom.

(a) Does goal list (factorial(0, X), X = 2) fail, as it should?

(b) Does goal factorial(0, 2) correctly fail, as it should?

32.6. Prolog predicate var(X) succeeds when X is an unbound unknown and nonvar(X) succeeds when var(X) fails. Write a definition of predicate sum(X, Y, Z) that is true when $Z = X + Y$, and that can be used in any of the modes sum(in, in, out), sum(in, out, in) and sum(out, in, in).

32.7. Write axioms for predicate cousin(X, Y), which is true if X and Y are cousins. That is, X and Y must have a grandparent in common, but they must not also have a parent in common, or be the same person. Use predicates parent(X, Y) (X is Y's parent) and grandparent(X, Y) (X is Y's grandparent). Design this predicate to run in mode cousin(in, in).

32.8. Does your definition of cousin(X, Y) from the preceding exercise work in mode cousin(in, out)? If not, can you make it work in that mode? Either give a definition that works in that mode, or argue that doing so would be very difficult.

32.9. Write a Prolog program to solve the missionary/cannibal puzzle (Exercise 26.6). Use jailbird notation to represent numbers.

32.10. How are the concepts of negation as failure and the closed world hypothesis related? Is one based on the other, or are they completely unrelated ideas?

32.11. Explain why using negation as failure typically limits the mode of a predicate. Does negation as failure always work when all of the parameters are in-parameters, containing no variables? Explain why or why not.

32.12. Show how to translate Prolog axiom `P(X) :- A(X), (Q(X) ; R(X))` into a form that does not use a semicolon in two ways, one using an auxiliary predicate that stands for the disjunction (Q(X) ; R(X)), and another that does not introduce any auxiliary predicate. Does it make a difference which one is used? What if there is a cut, as in `P(X) :- A(X), (Q(X), ! ; R(X))`? Try this in a version of Prolog that is available to you and try to determine which one is used in that version.

32.8. Bibliographic notes

Clark [30] discusses negation as failure, and Naish [80] shows how to make it mathematically correct by deferring negations until they can be done. Constraint programming languages [39, 74] do much more, allowing a variety of conditions to be asserted and held until it is possible to test them.

Chapter 33

Logic Programming in Cinnameg

33.1. Writing predicate definitions in Cinnameg

To write predicate definitions in Cinnameg, use the same form as for defining equations by cases, but with the following differences.

1. To request that choices be made by backtracking instead of by selecting just one case, write the word **all** before the cases. (But if, for a particular predicate, you only want to use the first successful case, write **first**.)

2. $B \Leftarrow A$ is written B `<-` A in Cinnameg. Writing `<-` in place of = in a definition requests that pattern matching should be done with (naive) unification instead of in the usual (equational) way. Writing `<--` instead requests full unification. You will need to import logic/unify.cmg to use unification.

3. For symbols, use string constants. For example, write ”marcy” instead of marcy.

4. For variables, use names that start with lower case letters, instead of upper case letters. You need to create your auxiliary variables before you use them. Write `Exists x,y,z | A %Exists` to create unknowns x, y and z for use within A.

5. A predicate produces a boolean result. Use **and** for the logical *and* operator and ‘bor‘ for an *or* operator that backtracks. Logic-programming predicates often only yield a true result. Instead of yielding false, they fail.

Figure 33.1 shows a few examples of predicate definitions converted to Cinnameg and Appendix B shows a longer example.

To run a goal g, use statement `MakeSure` g `%MakeSure`. For example,

```
Execute
  Exists p |
    MakeSure grandparent(p, "cathy").
    Display p.
  %Exists
%Execute
```

```
Define
  all
  case mother("marcy", "cathy") <-
  case mother("sally", "bertrand") <-;
  all
  case father("bertrand", "marcy") <-
  case father("william", "cathy") <-
  case father("jordan", "william")<-;
  all
  case parent(x,y) <- mother(x,y)
  case parent(x,y) <- father(x,y);
  grandparent(x,y) <-
    Exists z | parent(x, z) and parent(z, y).;
  all
  case ancestor(x,y) <- parent(x,y)
  case ancestor(x,y) <-
    Exists z | parent(x,z) and ancestor(z,y).
%Define
```

Figure 33.1: **Predicate definitions in Cinnameg.** A semicolon ends one definition and starts another. If there is nothing after `<-`, then the right-hand side of the case is assumed to be **true**.

shows all grandparents of Cathy. To show just one answer, wrap Unique...%Unique around the part that generates the solution.

33.2. Exercises

33.1. Express the Prolog append predicate in Cinnameg.

33.2. Solve each of the following exercises from Chapter 31, but write the definitions in Cinnameg instead.

(a) Exercise 31.1

(b) Exercise 31.2

(c) Exercise 31.3

(d) Exercise 31.4

Part XI

Types

Chapter 34

Programmer-Defined Types

34.1. Introduction

The early programming language Fortran IV was designed for scientific computing with real numbers. It provides types DOUBLE PRECISION (real numbers), INTEGER and a few other simple types, plus arrays. But the types that are provided by the language are all that can ever be used. No new types can ever be defined. Algol 60, another early programming language, similarly provides a few basic types such as integer, real and character, without offering the possibility of creating new types.

A programmer can often get by without any types except the basic ones. For example, if you need to use a point such as (2.0, 3.0) in the cartesian plane, an array of two real numbers will do the job. But an important feature provided by a programming language is the ability to abstract, or to hide details. The programmer might prefer to use a type Point, rather than a more detailed type such as (Real, Real). The type (Real, Real) is merely the *representation type* of type Point — the type under which points are stored.

This chapter adopts the model-theoretic view of a type as a set of values, as discussed in Chapter 9. Because of that viewpoint, we avoid operations that change things. The next chapter discusses how imperative ideas force a change of viewpoint on what a type is.

34.2. Defining data types

Programmers make their works easier to understand by adding new concepts to them. For example, new functions and procedures implement new computational concepts. New types of data are a natural extension of that idea.

We use Cinnameg to illustrate definitions of types. Other programming languages exhibit similar characteristics. In Cinnameg, you create a new type using a type declaration, indicating that a new type is represented by a previously known type. For example

```
Type Distance = Real.
```

defines a new type Distance to be represented by type Real. Think of type Real as a set of numbers. For each member of type Real, there is a corresponding member of type Distance, which you might understand to represent a given distance, in meters. So, corresponding to the real number 3.4 is the distance (3.4 meters).

```
Pattern meters(x) =>
  Match distance(x) = target.
%Pattern
```

Figure 34.1: **Definition of patterns that use meters in Cinnameg.** This indicates that matches of the form Match meters(p) = d %Match are converted into equivalent form Match distance(p) = d %Match. The target d of the pattern match is called **target**.

Constructors and unconstructors

Distance is not the same as Real, in spite of the close correspondence between the two. In order to use Distance, you need a way to convert from Real to Distance and from Distance to Real. When you define Distance, the Cinnameg compiler automatically provides two functions for those conversions: a *constructor* called *distance* that converts from Real to Distance, and an *unconstructor* called *undistance* that converts from Distance to Real. So distance(4.0) is (4.0 meters), the member of type Distance that corresponds to real number 4.0, and undistance(distance(4.0)) is 4.0 (a real number). You could rename the constructor *meters* and the unconstructor to *numberOfMeters*, as follows.

```
Define meters = distance.
Define numberOfMeters = undistance.
```

Then you can write meters(4.0) for the distance 4 meters, and numberOfMeters(d) for the real number that tells how many meters are in distance d.

Rather than using unconstructors directly, you typically use pattern matching to convert from Distance to Real. Pattern match

```
Match distance(r) = d.
```

binds r to the real number that corresponds to distance d. What you are asking here is, how many meters is distance d? The answer, r, is a real number. If you want to use the meters function in patterns, you need to tell the compiler how to deal with it, as shown in Figure 34.1.

Defining operations on new types

When you define a new type, only a few operations are provided for your new type, including a constructor and an unconstructor and possibly a few other operations. It is up to you to provide additional operations. For example, if you want to be able to add two distances (producing another distance), you define + for distances.

```
Define distance(x) + distance(y) = distance(x+y).
```

It is usually a good thing that other operations are not defined automatically. For example, you would not want an automatic definition of multiplication, where the product of two distances is a distance, since the product of two distances should be an area. A more sensible thing is to define type Area, and then multiplication of distances as follows.

```
Define distance(x) * distance(y) = area(x*y).
```

34.3. Structured types and selectors

When you define a new type that is represented by a structured type, you typically also define operations that select parts of the representation. In Cinnameg, for example, if type Computer is defined by

```
Type Computer = (
  manufacturer ~: String,
  model        ~: String,
  serial       ~: Integer)
%Type
```

then it is as if Computer were defined to be represented by a tuple

```
Type Computer = (String, String, Integer).
```

but functions manufacturer, model and serial are automatically defined as follows.

```
Define manufacturer(computer(m,?,?))  = m.
Define model(computer(?,m,?))  = m.
Define serial(computer(?,?,s)) = s.
```

If those functions had not been provided, you could (and probably would) write them yourself. That is, for example, how a "type" is defined in a typeless language such as Scheme. If a computer is represented by a list of three quantities, then you could define

```
(define (manufacturer c) (car c))
(define (model c) (cadr c))
(define (serial c) (caddr c))
```

in Scheme, where (car c) is the fist member of list c, (cadr c) is the second member and (caddr c) is the third member.

Selection for list and array types

A list has a first member, a second member, and so on, and the position of a member is called its *index*. Using the idea of an index, you can view a list as a small function, mapping indices to values. For example, list $L = [5, 90, 23]$ implicitly represents the following function, with domain $\{1, 2, 3\}$.

x	$L(x)$
1	5
2	90
3	23

So $L(2) = 90$, and in general, the value at index x in list L is the value $L(x)$, treating L as a function. Typically, a list has a small range of integers as its domain, but you can imagine other domains. For example, Ada allows the indices to be any type that is represented by a small range of integers. Ada declaration

```
X: array(Integer range 4..7) of Integer;
```

```
Type Clothing =
    socks (size   ~: Integer)
  | pants (waist  ~: Integer,
           inseam ~: Integer)
  | shirt (chest  ~: Integer,
           neck   ~: Integer,
           sleeve ~: SleeveType)
%Type
```

Figure 34.2: A data type for describing articles of clothing, written in Cinnameg. There is one constructor for each kind of clothing.

creates array X so that its domain is $\{4,5,6,7\}$. The first member of X is $X(4)$. But Ada also supports *enumerated types*, where the programmer lists the members of a type. A type of allowable traffic light colors can be written as follows in Ada.

```
type TrafficLightColor is (Red, Amber, Green);
```

Type TrafficLightColor has just three possible values, called Red, Amber and Green, and they are represented by small integers, with Red = 0, Amber = 1 and Green = 2. An array that has TrafficLightColor as its domain type, and that holds values of type Integer, is defined as follows in Ada.

```
X: array (TrafficLightColor) of Integer;
```

Now it makes sense to talk about X(Red), X(Amber) and X(Green).

34.4. Choice in data

Sometimes you need a new type whose values have several different kinds, or variants. You can represent such data types by having more than one representation type and more than one constructor. For example, suppose that you need to represent data about different kinds of articles of clothing, with different characteristics for each kind. Figure 34.2 shows a data type for clothing, written in Cinnameg, with three kinds, socks, pants and shirts. Now expression `socks(11)` creates a pair of socks (of type Clothing) with size 11. You can also write `socks(size ~: 11)`, showing the name of the parameter. If `shortSleeve` is a member of `SleeveType`, then expression `shirt(chest ~: 40, neck ~: 15, sleeve ~: shortSleeve)` creates a shirt, also of type Clothing,

Data types with multiple forms are quite common in software. A program that works on representations of people will probably need to store different kinds of data about people who play different roles. Data about a shipment might depend on the status of the shipment, whether it is in transit, arrived at its destination, etc. A list has two forms, an empty list and a nonempty list. An example comes up in compiler design. The initial stage of a compiler, called the lexical analyzer, reads individual characters of a program and packages them up into *tokens*. Examples of tokens are a reserved word, such as **if**; an identifier such as **size**; a number such as 24; and a special character such as **(**. The lexical analyzer tells the next stage of the compiler what kind of token it has read, and provides an *attribute* that gives additional information about the token, if necessary. The attribute of an identifier might

```
Type Token =
    identifierToken(String)
  | integerConstantToken(Integer)
  | realConstantToken(Real)
  | ...
%Type
```

Figure 34.3: **A type for tokens.**

```
Define
  case weft(socks(size ~: sz))        = sz
  case weft(pants(waist ~: w, ?))     = w
  case weft(shirt(chest ~: c, ?, ?))  = c
%Define
```

(a)

```
Define
  case PrintToken(identifierToken(name)). =
    Write["Identifier:  ", name].
  case PrintToken(integerConstantToken(i)). =
    Write["Integer:  ", $(i)].
  case PrintToken(realConstantToken(r)). =
    Write["Real:  ", $(r)].
%Define
```

(b)

Figure 34.4: **Two functions defined using choice.**

be the name of the identifier, a string. The attribute of a token such as 334 might be an integer, and the attribute of 2.53 might be a real number. Clearly, the token must have several forms. Figure 34.3 shows a definition of type Token.

Computation on data types with variants

To perform computations on types that involve variants, you typically use programs that involve choices that match the variants. For example, suppose that you want to compute the "weft" of a piece of clothing. where the weft of a pair of socks is its size, the weft of a pair of pants is the waist size, and the weft of a shirt is the chest measurement. Figure 34.4(a) shows a definition of weft. Notice that there is one case for each kind of clothing article. Similarly, suppose that you would like to print a value of type Token. You will need a case for each kind of token, as is shown in Figure 34.4(b). Choice in data types is mimicked by choice in computation.

```
Type Clothing =
    hairpin
  | socks (size  ~: Integer)
  | pants (waist ~: Integer,
           inseam ~: Integer)
  | shirt (chest ~: Integer,
           neck  ~: Integer,
           sleeve~: SleeveType)
%Type
```

(a)

```
Define
  case category(=hairpin) = 0
  else category(?)  = 1
%Define
```

(b)

Figure 34.5: **A modified clothing type, written in Cinnameg.** Part (a) shows type new type definition and part (b) defines a function for the new type. Notice that hairpin is a constant of type Clothing. Pattern (=hairpin) only matches hairpin.

Constants in data types with variants

A shirt has three features, its chest and neck measurements and its kind of sleeve. But some variants of a type can be featureless, having no extra information. For example, suppose that a new article of clothing, a hairpin, is added, and that there is only one size of hairpin, so no additional information needs to be stored with one. The definition of Clothing from Figure 34.2 can be modified as shown in Figure 34.5(a), where there is no data associated with the hairpin. Now hairpin is a constant of type Clothing. Figure 34.5(b) shows a definition of a function called category, where the category of a hairpin is 0, and the category of all of the other kinds of clothing is 1.

When all of the members of a type are featureless the type is called an *enumerated type*. For example, Figure 34.6 defines two enumerated types, ColorType and SleeveType. Again, the obvious way to compute with these things is by cases. Figure 34.6 shows a definition of a function that tells whether a color is "warm".

Implementation of data types with variants

Values with variants are represented using *tags*. When you create value identifierToken("size"), for example, the value contains, within its representation, a tag indicating that it was constructed using constructor identifierToken. The tags are typically integers; you might use 0 for identifierToken, 1 for integerConstantToken and 2 for realConstantToken.

A value of an enumerated type can be represented by just its tag, so an enumerated type has a representation type that is a (small) range of integers. Type ColorType of Figure 34.6 uses the representation red = 0, orange = 1, yellow = 2, green = 3 and blue = 4. Of

```
Type ColorType = red | orange | yellow | green | blue.
Type SleeveType = shortSleeve | longSleeve.

Define warm: ColorType -> Boolean by
  case warm(=red)    = true
  case warm(=orange) = true
  case warm(=yellow) = true
  case warm(=green)  = false
  case warm(=blue)   = false
%Define
```

Figure 34.6: **Definitions of two enumerated types in Cinnameg,** and definition of a function on colors.

course, that is only the representation; ColorType and Integer are different types.

Untagged unions

A few languages, such as C, offer an alternative way of creating data types with choice that avoids tags. An attribute type in C can be written as follows.

```
typedef union AttributeType AttributeType;
union AttributeType {
  string identifierAttr;
  int    integerAttr;
  double realAttr;
};
```

A value of type AttributeType[1] is either a string, an integer or a real number (type double), but there is no indication in the value itself of which kind it is. All of the different kinds of values occupy the same memory, and when you store one, you replace (or invalidate) the others. The programmer must know, by other means, which kind value is stored. An obvious way to do that is to create your own tag, and to define type Token as follows.

```
typedef struct Token Token;
struct Token {
  int tag;
  AttributeType attr;
};
```

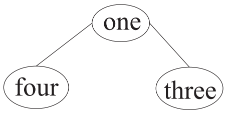
The code for a compiler might contain

```
if(token.tag == integerToken) {
  int tokenValue = token.attr.integerAttr;
  ...
}
```

[1] The type created in the union is called **union AttributeType** in C. The typedef line allows you to use the shorter name, AttributeType.

```
Type BinaryTree =
    leaf (leafLabel ~: String)
  | tree (treeLabel    ~: String,
          leftSubtree  ~: BinaryTree,
          rightSubtree ~: BinaryTree)
%Type
```

Figure 34.7: A recursive definition of a type of binary trees, written in Cinnameg.



(a)

```
tree("one", leaf("four"), leaf("three"))
```

(b)

Figure 34.8: A tree (a) and an expression whose value is that tree (b) using type BinaryTree of Figure 34.7.

using token.attr.integerAttr to select the integerAttr interpretation of the attribute.

Untagged unions are flexible, but they represent a serious breach of type safety. If a programmer makes a mistake, his or her program can treat a value of one type (for example, string) as if it were the representation of another type (for example, double), with unpredictable results.

34.5. Recursion in data

Recursion is a phenomenon that occurs frequently in data types. A large list contains a smaller list (its tail) as part of it. A large tree contains smaller subtrees. Figure 34.7 shows a definition of a type of binary trees where a tree is either a leaf that holds a string or is a nonleaf that holds a string and two subtrees. Figure 34.8 shows a tree and how that tree can be created in a program. Function treeSize tells how many strings are in a tree.

```
Define
  case treeSize(leaf(?))         = 1
  case treeSize(tree(?,lft,rgt)) = 1 + treeSize(lft) + treeSize(rgt)
%Define
```

Notice two things about the definition of function treeSize. First, there are two kinds of trees, leaves and nonleaves. Corresponding to that, there are two cases in the definition of treeSize. Second, recursion in the definition of treeSize matches recursion in the definition of the BinaryTree type. A leaf is the basis of the recursion in both the type and the function

```
Type BinarySearchTree =
    emptyTree
  | node(key   ~: String,
         left  ~: BinarySearchTree,
         right ~: BinarySearchTree)
%Type
```

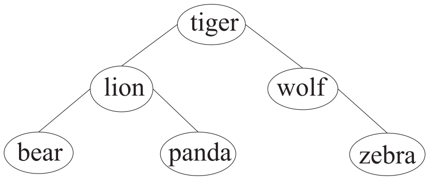
Figure 34.9: Cinnameg definition of a type for binary search trees, used to represent sets of strings.

definition. A nonleaf is defined in the type using recursion, and the definition of treeSize has matching recursion. A recursive definition whose choices and recursion matches the form of the type definition is called a definition by *primitive recursion*. Function treeFlatten, which performs an inorder traversal of a tree, converting it into a list of strings, is another example of primitive recursion.

```
Define
  case treeFlatten(leaf(s)) = [s]
  case treeFlatten(tree(s,lft,rgt)) =
    treeFlatten(lft) ++ [s] ++ treeFlatten(rgt)
%Define
```

Binary search trees

A binary search tree is a kind of tree that is used for organizing sets and tables. It has the characteristic that, for each node N holding value v, all of the nodes in the left subtree of N hold values that are less than v, and all of the nodes in the right subtree of N hold values that are greater than v. For example, the following is a binary search tree that represents the set of strings {"bear", "lion", "panda", "tiger", "wolf", "zebra"}, using alphabetical order.



Type BinaryTree is not quite right for binary search trees, since a binary search tree needs some nodes to have just one subtree. One way to fix that is to introduce an empty tree, letting a leaf be a tree with two empty subtrees. Figure 34.9 shows type BinarySearchTree that uses that idea.

Figure 34.10 shows a definition of a function that tests for membership of a given string in a binary search tree, using the principles of function design. There is one case for the empty tree, but the case for a nonempty tree is broken into three parts for convenience.

A function that inserts a value into a binary search tree can be designed using the same principles. The insert function takes a string x to insert and a tree t in which to do

```
Define
  case member?(x, (=emptyTree)) = false
  case member?(x, node(k, lft, rgt))
    = true            when x == k
    = member?(x,lft)  when x < k
    = member?(x,rgt)  when x > k
%Define
```

Figure 34.10: **A membership tester binary search trees.** (The question mark is part of the name.)

```
Define
  case insert(x, (=emptyTree)) = node(x, emptyTree, emptyTree)
  case insert(x, node(k, lft, rgt))
    = node(x, lft, rgt)            when x == k
    = node(k, insert(x,lft), rgt)  when x < k
    = node(k, lft, insert(x,rgt))  when x > k
%Define
```

Figure 34.11: Insert(x,t) produces the result of inserting x into binary search tree t. This function is consistent with a persistent implementation of binary search trees, the kind of type that we are restricting ourselves to in this chapter. It does not modify the tree, but computes a new tree.

the insertion, and returns the tree that results from inserting x into tree t. Figure 34.11 shows a definition of insert.

Implementation of recursion in data types

Recursion in types needs to be implemented with care. Typically, a given type of data occupies some fixed number of bytes. If you try to put a tree inside a tree, you find that a tree must occupy more bytes than a tree, which is impossible. To avoid that problem, a recursive use of a data type is usually implemented as a pointer to the actual representation. In languages that support explicit pointers, such as C++, recursion must be done via an explicit pointer.

Data types with recursion typically require variants since otherwise there would be no way to break the recursion. The nonrecursive cases correspond to basis cases in recursive algorithms. Using pointers allows you to take advantage of an implicit variant in the pointers to break recursion; a pointer can either be null or nonnull. A binary tree type can be defined as follows in C++, relying on null pointers to represent termination of the recursion. Type Tree* is the type of a pointer to a tree.

```
struct Tree {string data; Tree* left; Tree* right; };
```

With this definition of type Tree, a leaf is a tree node whose left and right pointers are both null. There is no need to use an explicit union type to create an empty tree, and there are no extra tags to keep track of. In C++, you are not allowed to write

```
struct Tree {string data; Tree left; Tree right; };
```

since that indicates that a tree structure contains (all of the memory of) two other tree structures inside it.

Even some languages that keep pointers hidden beneath the surface allow you to take advantage of null pointers. For example, in Java, all objects are represented by pointers, and an object can be **null**.

34.6. Type compatibility

Programmers who work in statically typed languages learn to rely on type checking. If, for example, you pass the wrong type of parameter to a function, then you expect to be told so.

Sometimes more than one conceptual type has the same representation type. Suppose, for instance, that you need two data types; type Point represents a point in the cartesian plane, and type Complex represents a complex number. Each is represented by a pair of real numbers. What happens if you accidentally pass a value of type Point to a function whose domain type is Complex?

Say that two types A and B are *equivalent* if an item of type A can be used where an item of type B is expected, and vice versa. Two common rules for defining when types are equivalent are the *structural equivalence* rule and the *name equivalence* rule.

Two types are structurally equivalent if they have the same representation type. Under the structural equivalence rule, types Point and Complex are equivalent, since both are represented by type (Real, Real). With structural equivalence, you do not really create new types at all, but just gives new names to old types.

The name equivalence rule states that two types are equivalent just when they have the same name. So, for example, Point and Complex cannot be equivalent under name equivalence, since they have different names. With name equivalence, each new type is a genuinely new type. Name equivalence allows for very stringent type checking, and so has become the rule of choice.

Not all programming languages employ just name equivalence or just structural equivalence. Some use a mixture of the two. Pascal provides a particularly bizarre example. If you write

```
type MyInt1 = integer;
     MyInt2 = integer;
```

in Pascal, it appears that you have defined new types. But you have actually just created two new names MyInt1 and MyInt2 for built-in type integer. Structural equivalence is used; all three of MyInt1, MyInt2 and integer are equivalent types. But if you write

```
type MyArray1 = array [1..4] of integer;
     MyArray2 = array [1..4] of integer;
```

in Pascal then you have created two genuinely new and inequivalent types called MyArray1 and MyArray2. For array types, Pascal uses name equivalence. Irregular rules such as that are strongly discouraged in language design. Ada is roughly based on Pascal but is more recent, and it adopts the name equivalence rule regardless of the form of the representation type.

34.7. Summary of terminology and concepts

A type can be defined by choosing another type as its representation. A constructor for type T converts from the representation type of T to T. and an unconstructor converts back.

Two fundamental notions for controlling computation are choice and recursion. Those same concepts can be used in data types, where a type can involve choices, and can refer to itself. Typically, computation emulates the structure of the type. Choice in data is handled by choice in computation, and recursion in data is handled by recursion in computation.

Terminology

Constructor A constructor is a function that creates an item of a new type. It might attach a tag to indicate which constructor was used. A constructor might be used in patterns to accomplish the reverse conversion.

Enumerated type An enumerated type is a type all of whose variants are featureless. It has a finite number of members, each given a name in the definition of the type.

Name equivalence Two types are equivalent under the name equivalence rule if they have the same name.

Primitive recursion A function definition by primitive recursion has a case for each case in the definition of a type, and has recursion that matches the recursion that occurs in the definition of the type. For example, if one variant of type Tree has the form node(Char, Tree), then a primitive recursive definition of function f on a parameter of type Tree would contain a case f(node(c,t)) $= \ldots\ f(t)\ \ldots$, using the same function on the subtree t.

Representation type The representation type of a new type T is a type that tells the details of how information of type T is stored. A new type is typically defined as being a copy of its representation type, or as being a tagged union of copies of more than one representation type.

Structural equivalence Two types are equivalent under the structural equivalence rule if they have the same representation.

Unconstructor A unconstructor is the inverse of a constructor. It converts an item into its representation.

Union type A union type is a data type with choices.

34.8. Exercises

34.1. Does the C or C++ typedef directive create a new name for a type or a genuinely new type? Try an experimental program to see how it is treated by one compiler. Should all compilers be consistent here?

34.2. Using the Distance type, write a function in Cinnameg that doubles a distance. That is, dbl(d) should produce a distance that is twice as long as distance d.

34.3. Would it make sense to have a function, of type (Distance, Real) → Distance, that adds a distance and a number? Why or why not? If so, what would the definition look like?

34.4. Would it make sense to have a function, of type (Real, Distance) → Distance, that multiplies a distance by a real number? Why or why not? If so, what would the definition look like?

34.5. Not every function that works on type Clothing needs to be written by cases. Write a definition of a function bigger such that bigger(a, b) is true if a has a larger weft than b, and is false otherwise. Figure 34.4 defines weft.

34.6. Write a Cinnameg function that produces the alphabetically smallest string in a nonempty binary search tree, using the type in Figure 34.9.

34.7. Using type BinarySearchTree from Figure 34.9, write two functions in Cinnameg.

(a) removeMin(t) should return a pair (x, t') where x is the smallest number stored in t and t' is a tree that holds all numbers in t except x. Presume that t is not empty. Make sure that t' is a binary search tree.

(b) remove(x, t) should return the tree that results from removing x from tree t. If x is not a member of tree t, then remove(x, t) should return t. (**Hint.** Start by thinking about trees with at least one empty subtree. Then deal with trees where both subtrees are nonempty. Use removeMin to help in that case.)

34.8. A *selection tree* is used to store a set of numbers so that the operation of finding the k-th smallest member of the set is efficient. A selection tree is the same as a binary search tree, but at each node an extra piece of information is stored telling the number of values in the subtree rooted at that node.

Define a type for selection trees. Define persistent versions of insert(x, t) and find(k, t), where insert(x, t) produces the result of inserting x into tree t, and find(k, t) returns the k-th smallest value in tree t. Use the sizes to determine where to find the k-th smallest. Be sure that your insert functions puts the correct size in a node.

34.9. A queue is a first-in-first-out list. A persistent implementation of queues has a constant emptyQueue (an empty queue), function isEmpty(q) (which returns true when q is empty), function insert(x, q) (which produces the queue that results when you add x to the end of queue q) and delete(q) (which produces ordered pair (x, q') consisting of the first member x of q and the queue q' that results when your remove x from q). The delete function should fail on an empty queue.

One way to represent a queue is via a pair of lists, (f, r). The contents of the queue, from front to back, consists of list f followed by the reversal of r. For example, pair ([1,2], [5,4,3]) represents a queue that contains [1,2,3,4,5], from front to back. To insert a value at the back, you just add it to the front of r. To remove a member when f is not empty, just remove the first member of f. If f is empty, then switch the representation from ([ ], r) to (reverse(r), [ ]).

Give a definition in Cinnameg of type Queue, representing a queue of strings, as well as definitions of emptyQueue, isEmpty, insert and delete, using this idea for representing queues.

Chapter 35

A Formal View of Types

35.1. The formal view of a type

Chapter 34 adopts a *model theoretic* viewpoint that a type is a set of values. But an alternative *formal* viewpoint avoids thinking about the data items are, and instead concentrates on what you can do with them. We will start by looking at the formal view of types that can also be described as sets, so that they end up with two consistent definitions. In the final section we turn to types for which model theory fails, and only the formal approach yields a sensible definition.

The signature of a type: type syntax

The first part of the formal view of a data type is the *signature* of the type, which is collection of constants and functions, each with a given type. For example, a natural number is (by our definition) a nonnegative integer. The signature of type Natural might include function +, which takes two natural numbers and produces a natural number, and a constant called 0 that is a natural number, and a few other components. A fairly sparsely populated natural number type would have a signature (0: Natural, 1: Natural, +: (Natural, Natural) → Natural, *: (Natural, Natural) → Natural, ==: (Natural, Natural) → Boolean), consisting of constants 0 and 1, the addition and multiplication operations, and a test to see whether two natural numbers are equal.

Type semantics in the formal view

The signature does not say anything about meaning, just as the syntactic specification of a programming language does not say what programs mean. The second part of the formal view of a type introduces semantics.

In the formal view, there is no model, so we cannot say what 0 stands for, or what it means to add two numbers. Instead, we state some properties that the functions and constants in the signature must have. Here, for example, are some of the relationships that are part of the formal view of the natural-number type.

$$\begin{aligned} x + 0 &= x \\ x + (y + 1) &= (x + y) + 1 \end{aligned}$$

$$\begin{aligned}
x * 0 &= 0 \\
x * 1 &= x \\
x * (y + 1) &= (x * y) + x \\
(0 == 0) &= \text{true} \\
(x + 1 == 0) &= \text{false} \\
(x + 1 == y + 1) &= (x == y)
\end{aligned}$$

The formal view is a useful one for reasoning about programs, since it provides, in the properties, the fundamental facts from which to reason. For example, you can use the operations in the signature of Natural to write a definition of a function that exponentiates a number to a positive integer power. (Just write a loop.) Using the properties of the integer operations, and reasoning methods such as the logic of Chapter 47, you can prove that your definition of exponentiation is correct. The formal view gives an *axiomatic semantics* to a type, in the same way that the Hoare logic of Chapter 47 is an axiomatic semantics of a small programming language.

The formal view is useful for specifying the intended semantics of a data type without giving an implementation. Any implementation that has the right signature and satisfies the properties of the type is, by definition, correct.

35.2. Abstract data types and encapsulation

A type that is viewed using the formal view is called an abstract data type. The idea is that it only supports the operations given in the signature, and no other operations can be used on it unless they are defined in terms of the ones in the signature. A language that supports abstract data types should enforce the requirement that only operations in the type's signature can be used. That can put some stress on a programming language.

When you define operations on a type, you need to know the type's representation, which includes features that are not part of the signature. It is not possible, for example, to define the details of how points in the plane work unless you can make use of the representation so that you can extract the components (the x- and y-coordinates). To implement a set of strings as a binary search tree, you need to know details of binary search trees, which certainly are not part of the signature for a set of strings. Typically, you produce one module that implements operations on your new type. In all other modules, you would like to ensure that the representation type is completely hidden; those other modules should be restricted to the type's signature.

A new type is said to be *fully abstract* in modules where all aspects of its representation are hidden, and only operations that are part of the type's signature can be used. Those modules are not allowed to know, for example, that a point is a record that has a component called x, or even that it is a record type at all. A fully abstract type is a kind of encapsulation, and a big advantage of any kind of encapsulation is ease of modification. Changes to the representation of a type cannot affect any modules where that type is fully abstract, as long as the changes respect the signature and semantics.

It is impossible to create a fully abstract type using the structural equivalence rule, since then the representation type is an inherent characteristic of the type. You might expect the name equivalence rule to make representations of types hidden, encapsulated behind the name of the type. But even with name equivalence, you can get some unexpected dependence on representation. For example, C uses name equivalence for structures (records). If you define two structure types

```
typedef struct {double x,y;} Point;
typedef struct {double re,im;} Complex;
```

then, as name equivalence suggests, types Point and Complex are different types, and you cannot pass a value of type Point to a function that expects a parameter of type Complex. But if you write

```
Point p;
p.x = 1.0;
```

then you are relying on the knowledge that a point has a component called x, which you can only know by looking at the representation. Name equivalence *enables* fully abstract types, but is not enough. You must be sure to hide the existence of any operations, such as component selectors, that are specific to the representation.

There can be other operations that also need to be hidden to achieve a fully abstract type. One is assignment. In C, for example, you can perform assignments on structures, as in the following example.

```
Point p,q;
...
p = q;
```

The definition of C says that assignment on structures is done coherently, by copying each component from one structure into the other. That is, the definition of the assignment operator is determined by the representation. Sometimes this gives you what you want, and sometimes it does not. If you change the representation later, then you might find that the C compiler's idea of assignment is not what you want, and changing the representation has the effect of changing what assignment does. For example, if you decide to represent the components of a point as a linked list, then the standard definition of = will not cause the list to be copied; it only copies the pointer to the first cell in the linked list. For a fully abstract type, you need to remove *all* dependence on the representation.

Avoiding automatic definitions

Sometimes even the most basic operations are not appropriate for new types, and should not be defined automatically. In addition to the assignment operator, compilers sometimes automatically define an equality testing operator for new types. C does that, for example, when the type is represented by a pointer. But an equality operator might not be appropriate for your type; maybe you do not want to compare two Widgets to see whether they are equal, so an equality test is not part of the signature. You do not want it added for you by the compiler.

Ada allows a programmer to create a type whose details are completely hidden within a single module. Definition

```
package Points is
  type Point is private;
  ...
  private
    type Point is record
      X, Y: Float;
    end record
end Points;
```

makes Point available to other modules, but does not define any operations on it other than those explicitly mentioned in the (...) part. Ada packages are broken into interface parts (shown above) and package bodies, where implementations are written. The definition of a type is required to be written in the interface part, as shown; it is just marked private. Cinnameg uses a similar approach, but does not require mentioning anything about the representation of a type in the interface part of a package. Saying

```
Type Point interface.
```

makes type Point available, but not any functions, or any kinds of operations, on it. Only those operations that are explicitly mentioned in the interface become available to other packages. In the implementation part of the same module, you define the type by giving its representation. The constructor and unconstructor are available only in the implementation part, where they are needed.

35.3. Types without models

Chapter 15 discusses boxes, or first-class variables. A box is an object that can hold one thing, called the content of the box, which can change over time. There are two operations, one that gets the content of the box and another that changes the content. In Cinnameg, a box that holds an integer has type [:Integer:].

In the model-theoretic view, type Boolean has two members, true and false, and those values are created when the type is created. You cannot create new values of type Boolean, or destroy them, while the program runs. Type Integer also has a fixed set of members.

Box types are different. You can create boxes on the fly and later destroy them, so the membership of a box type is constantly changing. Instead of saying that a given box is a member of type [:Integer:], we say that it an *instance* of that type. In general, we will call a type whose instances have time-varying characteristics an *imperative type*.

Semantics of imperative types

The model-theoretic view does not work for imperative types. Since instances can be created and destroyed, there is no fixed collection of things to model. But the formal approach can still work.

The signature of an imperative type lists the functions and procedures that are available for that type, just as it does for a type that is a set of values.

The semantics that is associated with the signature needs to be described by axioms, or properties of the procedures and functions that express how procedures change things and characteristics of the results produced by functions. You can give a precondition/postcondition pair for a sequence of operations, similar to what is done in Hoare logic

(Chapter 47). Axiom $\{p\}S\{q\}$ states that, if condition p is true, and you perform the sequence of steps S, then condition q will be true afterwards. ($\{\,\}$ stands for $\{\text{true}\}$.) For example, in Cinnameg, the signature for type [:Integer:] has function @: [:Integer:] → Integer, where @(b) is the current content of box b, and procedure Assign: ([:Integer:], Integer) → (), where Assign (b,v) %Assign stores v into box b. An obvious semantic axiom is

$$\begin{array}{l}\{\,\}\\ \quad \texttt{Assign (}b\texttt{, }v\texttt{).}\\ \{@(b) = v\}\end{array}$$

But, as we will see, a full semantic description of an abstract data type, even as simple as boxes, requires a little more than the obvious rules, and even the obvious rules can be tricky.

Example: stacks

Imagine a type of (last-in-first-out) stacks, with the following operations.

1. push(x,s) puts x onto the top of stack s (modifying s).
2. pop(x,s) removes the top item from stack s, and puts the result into variable x. (If s is empty, then it is an error to pop s.)
3. initialize(s) makes stack s empty. An empty stack contains no items.
4. empty(s) returns true if s is empty, and false if s is not empty.

An obvious property of the stack operations is

$$\begin{array}{l}\{\,\}\\ \quad \texttt{push(}x\texttt{, }s\texttt{);}\\ \quad \texttt{pop(}y\texttt{, }s\texttt{);}\\ \{y = x\}\end{array}$$

which says that, assuming no preconditions, if you push x onto stack s and then pop s, putting the result of the pop into y, then the values of y and x will be the same.

At any given time, the content of a stack of integers can be described as a list of integers, from the top of the stack to the bottom. For example, after pushing 2 and 9, in that order, onto an empty stack, the stack will contain list $[9, 2]$. That allows us to talk about the concept of the *current value* of a stack, and to compare two stack values. We will just write s in assertions to stand for the current value of stack s. An important property of stacks tells how push alters the stack.

$$\begin{array}{l}\{s = t\}\\ \quad \texttt{push(}x\texttt{, }s\texttt{);}\\ \{s = x :: t\}\end{array}$$

But it is not enough to say what the stack operations do; you also need to say what the stack operations do not do. For example, pushing x onto stack s does not change x, so you expect $\{x = 0\}$ `push(`x`,`s`)` $\{x = 0\}$ to be correct. That forces us to use fairly general rules, allowing any kind of assertion that is not affected by an operation to be in both the precondition and the postcondition. Say that $A[s \leftarrow e]$ is the result of replacing all occurrences of s in A by e. For example, if A is assertion $s = r$, then $A[s \leftarrow x :: z]$ is assertion $x :: z = r$. The action of push can be described by the following axiom.

$\text{empty}(s) \iff s = [\,].$

$\{A[s \leftarrow [\,]]\}$
  `initialize(`s`)`
$\{A\}$

$\{A[s \leftarrow x :: s]\}$
  `push(`x`,`s`)`
$\{A\}$

$\{\text{empty}(s)\ \&\ A\}$
  `pop(`x`,`s`)`
$\{\text{false}\}$

$\{\neg\text{empty}(s)\ \&\ A[s \leftarrow \text{tail}(s),\, y \leftarrow x]\}$
  `pop(`y`,`s`)`
$\{A\}$

Figure 35.1: **Axioms for a stack data type in a Hoare style.** Postcondition *false* indicates an error, since it can never be reached. Notation $A[s \leftarrow \text{tail}(s),\, y \leftarrow x]$ indicates the result of replacing s by tail(s) and y by x in A.

$\{A[s \leftarrow x :: s]\}$
  `push(`x`, `s`);`
$\{A\}$

With enough properties such as this, you can say all of the properties that stacks must satisfy, and give a completely formal description of stacks. Figure 35.1 contains a set of axioms for stacks.

Even those axioms need to come with a warning. The value pushed onto a stack could be computed by an arbitrary expression. For example, there is nothing wrong with

```
push(f(y), s);
```

According to our axiom for Push,

$\{f(y) = 1 \text{ and } f(y) :: s = r\}$
  `push(`$f(y)$`, `s`);`
$\{f(y) = 1 \text{ and } s = r\}$

You can see that the axiom is correct if f is a mathematical function. But what if function f has a side-effect, as functions are typically allowed to in imperative languages? For example, you might compute f(y) once and get value 1, and compute it again and get 2. (What if f is a pseudo-random number generator?) Clearly, our rules are not ready to handle that. Our semantic rules presume that functions and expressions have no side effects.

35.4. Summary of terminology and concepts

In the formal view, a type is characterized by its signature (the set of constants, functions and procedures that are available) and a collection of properties that constitute an axiomatic semantics of the type. The intent of the formal view is to avoid talking about the data items themselves, and instead to concentrate on the operations that can be performed on them.

Terminology

Abstract data type See fully abstract type.

Axiomatic semantics An axiomatic semantics provides partial information about semantics in the form of mathematical statements. The semantics of the formal view of a type is a form of axiomatic semantics, similar to the semantics discussed in Chapter 47.

Formal view of a type In the formal view, a type is described by its signature and an axiomatic semantics.

Fully abstract type Whether a type is abstract or not depends on who is looking at it. A type T is fully abstract in module M if, within module M, only the operations in the signature of the type can be used with it. There must be no back-door operations, such as selecting a component from the representation, or performing assignments, unless those operations are specifically mentioned as part of the signature.

Instance An item that belongs to a given imperative type is called an instance of that type. Instances can be created and destroyed while a program runs.

Semantics of a type The semantics of a type tells the effect of operations that are part of the signature. In the formal view, the semantics is given as a set of axioms telling properties that the operations must obey.

Signature The signature of a type is the set of constants, functions and procedures on that type, along with type information for those constants, functions and procedures (saying, for example, the types of the parameters of a function). This is a part of the formal view, but omits semantic information.

35.5. Exercises

35.1. What is the main difference in focus between the model-theoretic view of a type and the formal view of a type. Which view is more useful for imperative programming?

35.2. Write an axiomatic semantics for a type of object that behaves similarly to a box. Let's call it a store, and say that a store object holds one value, of type integer. Operation get(s, x) puts the current value of store s into variable x. Operation put(s, x) makes x the current value in store s. Design the semantics so that any implementation that satisfies the semantics must be a correct implementation of a store. You can use s in the semantics to indicate the current content of store s.

35.3. A first-in-first-out queue is a list that allows new things to be added at its back and values to removed from its front. Write axioms that define the semantics of a first-in-first-out queue, where the signature contains function empty(q) (true if q is empty) and procedures initialize(q) (make q empty), insert(x,q) (insert x in q, at its back) and remove(x,q) (remove the first member of q, and make x equal to it). Removal from an empty queue is an error.

35.4. Section 35.3 discusses an imperative implementation of a stack type. Suppose that you intend to use a persistent (in the sense of Section 19.2) implementation, where you cannot change anything. There is a constant emptyStack, a function push(x,s) that produces the stack that results from pushing x onto stack s, a function pop(s) that produces the result of popping stack s (or yields an error if s is empty), a function top(s) that returns the top of stack s (or an error if s is empty) and a predicate empty(s) that returns true just when s is empty.

Suppose that you already have a model-theoretic semantics of a stack as a list of values, read from top to bottom. Now you want a formal semantics of stacks that is consistent with the model-theoretic semantics. Write equational axioms that describe stacks, so that any implementation that satisfies the axioms will be faithful to the idea of a last-in-first-out stack.

35.5. Write a fully abstract implementation of binary search trees in a language, such as Java or C++, with which you are familiar. Include empty(t) (is t empty?) member(x, t) (is x in tree t?) and insert(x, t) (insert x into tree t). How can you ensure that your definition is fully abstract? Is anything defined behind the scenes by the compiler that prevents your definition from being fully abstract?

35.6. Enclose your definition of Queue from Problem 34.9 in a module, and make it fully abstract. Export emptyQueue, isEmpty, insert and delete, and nothing more.

Part XII

Polymorphism

Chapter 36

Polymorphism Without Variables

36.1. Motivation

Dynamically typed languages allow a programmer to write very general functions, such as a function that will take the length of any type of list. Statically typed languages appear to be more restrictive.

Imagine that you would like to provide a general list manipulation facility in an older programming language, such as Pascal. You should provide a type for lists and basic utilities such as a function that computes the length of a list, a function that takes the tail of a list, and so on. You immediately face an almost impenetrable barrier. What kind of values should the lists be allowed to hold? Here is the dilemma.

- If you insist that the members of lists must be integers, then you can provide the facility, but it is almost useless. Chances are that anyone who wants to use lists wants to put things other than integers in his or her lists. Even providing several different types of lists is not adequate, since lists of new types will not be supported. For example, a programmer who creates a new type Widget will not be able to use lists of Widgets.

- On the other hand, if you want to allow arbitrary types of lists, there is no way to express the types of the functions in Pascal, and you cannot write the facility.

Pascal is an example of a *monomorphic* language. Every variable, parameter, etc. must have *one* type. There is no way to express the concept of a general list. For writing simple programs, without significant use of libraries, monomorphic languages such as Pascal are often adequate. But for providing general purpose tools in a statically typed language, some mechanism for dealing with multiple types of items at once is essential. Functions and procedures that can deal with multiple types are said to be *polymorphic*.

This chapter explores how polymorphism can be supported and used in a statically typed programming language. The approach that we use here is simple, but relies on deferring some type checking to run time so that a programmer can take advantage of the polymorphism that is inherent in dynamic type checking. The next chapter continues the study of polymorphism, examining an important idea for strengthening static type checking of polymorphic programs so that the type checks do not need to be done while the program runs, and type errors are reported when a program is compiled.

36.2. Ad-hoc polymorphism

Overloading

Pascal has a built-in function called sqr that can be used to compute the square of an integer or the square of a real number. To a certain extent, sqr is polymorphic, since it can be used on more than one type of number. But its polymorphism is very limited. Any time it is used, the compiler knows whether it is squaring an integer or a real number. There would be no fundamental difference in having two functions, one called sqrInt, the other called sqrReal. The only thing offered by function sqr is the convenience that you do not need to remember two function names.

Identifier sqr is *overloaded* in Pascal; that is, it has more than one meaning. Pascal also provides a few other overloaded symbols, such as + and *, which can be used to add or multiply any type of number. Overloading is nothing but a naming convenience, so that it is not necessary to come up with different names for those operations. Overloading is also called *ad-hoc polymorphism*. It is polymorphism that is resolved entirely at compile time; the compiler determines the exact types involved. For example, to write another function called square that is the same as sqr, you need to write two separate definitions of square.

Some languages allow the programmer to overload names. (Pascal does not. It requires that you choose different names for different functions.) But even the ability to do your own overloading is still just a naming convenience. We are looking for something deeper.

Coercion

Suppose that a function called sqrt, of type Real → Real, computes square roots. Obviously, sqrt(2.0) is a correctly typed expression. If constant 2 has type Integer, then expression sqrt(2) would normally be poorly typed. But it is unlikely that sqrt(2) indicates an error in the programmer's reasoning. Rather, the programmer has simply used the mathematical fact that an integer is a special kind of real number. From a mathematical point of view, expression sqrt(2) makes sense, and should probably be allowed.

One way to allow sqrt(2) and similar expressions is to employ a mechanism called *coercion*. A coercion is an implicit type conversion that is inserted by the compiler when it is needed. For example, a coercion rule might state that, if an integer appears where a real number is expected, then the integer is automatically converted, or coerced, to type Real.

Coercion provides an apparent polymorphism that is similar to overloading. It appears that the sqrt function is polymorphic, allowing its parameter to be either an integer or a real number. But the type ascribed to sqrt is still Real → Real, and the polymorphism induced by coercion, while quite useful, is only apparent, and is not quite what we are looking for if a fully general facility for polymorphism is to be provided.

36.3. Polymorphism and abstract data types

Most languages offer the ability to print many different types of values. Imagine that you have a procedure called Print that takes any type of thing and prints it. What would the type of Print be? Print performs an action; it does not produce a value, so the codomain of its type is the void type ( ). As a procedure, Print can be said to have polymorphic type ⟨Any⟩ → ( ), where ⟨Any⟩ stands for any type at all. That is, Print is a polymorphic procedure that takes any type of parameter and produces no result.

Cinnameg provides a function called $ that converts any kind of thing to a string. Its polymorphic type is ⟨Any⟩ → String. The type of $ does not need to be chosen at compile time when $ is used, as would be required of an overloaded name. For example, if you define twodollar by

```
Define twodollar(x) = ($(x), $(x)).
```

then function twodollar is also polymorphic, and its type is ⟨Any⟩ → (String, String). A key feature of genuine polymorphism is that a polymorphic function can be used to define a new polymorphic function in a single definition.

Types, genera and signatures

The notion of a polymorphic function or procedure that can take any type of parameter is appealing, but it is important to gain a better understanding of just what ⟨Any⟩ really means, and how it fits into the scheme of a type system. As a starting point, think of a language that is not polymorphic. An abstract data type, such as Stack, has two parts: (1) a *signature* that tells which operations are available; and (2) an implementation that tells how those operations are done. The signature and implementation are tightly tied together, with exactly one implementation of each operation in the signature. You are free to change the implementation, but in a given version of the program, there is just one implementation of Stack.

One way to understand polymorphism is to admit the possibility of more than one implementation of a single signature in the same program. You might print an integer one way, a string another way, a widget yet another way. But to admit such a possibility, there has to be some separation between the signature and the (multiple) implementations of that signature. That is where the notion of ⟨Any⟩ comes in. The signature is attached to it instead of to the individual types (Integer, String, etc.) Thought of that way, ⟨Any⟩ is a way of separating signatures from implementations.

Terminology concerning types and polymorphism is not fully agreed on, with terms used differently in different settings. We will use the term *type* for a particular way of representing data and implementing operations, and *genus* for something that is used to create signatures that have multiple implementations. For example, Any is a genus, but Integer is a type.

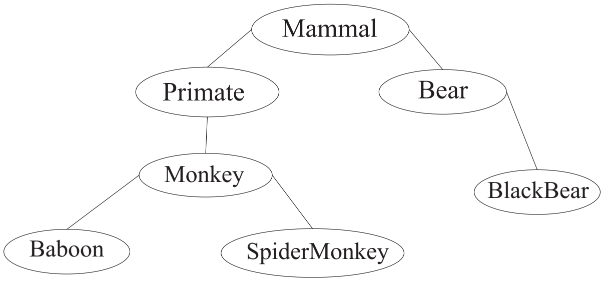
All types implement the signature of genus Any. But other genera whose signatures are only implemented by some types can also be defined. Genus Sequence might be implemented by various types, such as Stack and Queue, that represent particular kinds of sequences of things. Think of a genus as a collection of all of the types that implement it. If G is a genus, we write $\langle G\rangle$ in a type to indicate an arbitrary type that belongs to genus G. That is consistent with Cinnameg's notation.

Genera as sets of types

A simple viewpoint is that a genus is a set of types. For example, Any is the set of all types. A polymorphic type involving ⟨Any⟩ stands for the set of all types that can be obtained by replacing ⟨Any⟩ by one of the members of Any.

A polymorphic language almost certainly needs genera other than Any. For example, genus Number might stand for the set of all numeric types (Integer, Real, etc.), with a polymorphic square root function having polymorphic type ⟨Number⟩ → Real, indicating that it takes any kind of number and produces a real number. Genus Order can be defined to consist of all types that possess order tests ($<$, $>$, etc.). A procedure that sorts an array should typically sort any array whose members have a type that belongs to genus Order.

If a genus is a set,it makes sense to ask what it means for one genus to be a subset of another. Say that genus B is *beneath* genus A if B is a subset of A. A diagram showing subset relationships is called the genus *hierarchy*. Membership of a type in a genus is also included in the hierarchy; type T is beneath genus A if T is a member of A. For example, suppose that genus Mammal[1] contains all types that represent information about kinds of mammals. Specific examples of those types might include BlackBear, GrizzlyBear, SpiderMonkey, Gorilla, etc. It makes sense to create other genera that stand for restricted kinds of mammals. For example, genus Primate can contain all types of primates, genus Monkey can contain types of monkeys, etc. We will use names that are all upper case for genera, and capitalized names for types. Here is part of the hierarchy for Mammal and related genera and types.

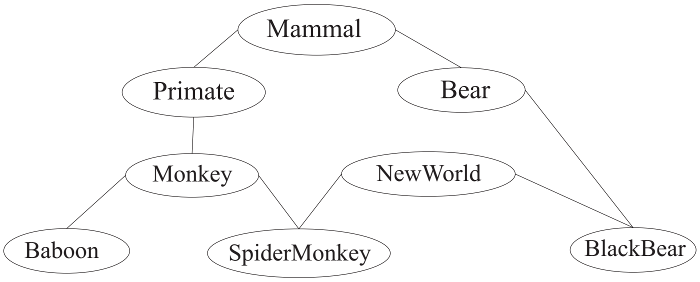


We will write $A \leq B$ if either A is the same as B or A lies beneath B in the hierarchy, where A and B are either types or genera. It is not necessary for A to lie *directly* beneath B. For example, BlackBear $\leq$ Mammal. The $\leq$ relation is *transitive*: if $A \leq B$ and $B \leq C$ then $A \leq C$.[2]

Not all hierarchy diagrams have the form of a tree. For example, suppose that another genus, NewWorld, is added, containing all new world types. Then a natural hierarchy has the following form, which is not a tree

[1] Biologically, classifications at different levels have different names, including phyla, classes, families, etc. But that kind of distinction is not useful for programming languages, and we call all classifications genera.

[2] The subset relation is a kind of *partial order*, and a diagram of the partial order is called a *Hasse diagram*. In the diagram, $B \leq A$ if there is a downward path through zero or more edges from A to B. See the end of the chapter for a definition of a partial order.



36.4. A formal view of genera

In the preceding section we viewed a genus as a set of types. But there is another viewpoint that has some advantages. In the formal view, a genus is described solely in terms of its signature. To describe the formal view of Number, you describe operations, such as + and *, whose types involve ⟨Number⟩.

The formal view also induces a hierarchy, but now the hierarchy is a result of the signature itself. Suppose, for example, that the only operation of genus Printable is one called Print, of polymorphic type ⟨Printable⟩ → ( ). Then any type T for which a procedure Print of type $T \to (\,)$ is defined should be beneath Printable. In general, a genus G is thought of as containing all types T such that all of the operations in the signature of G are defined if you replace $\langle G\rangle$ by T in their types.

Genera are typically *extensible*, meaning that new types can be added to them at various places in a program. Imagine that the signature for genus Mammal has a function *weight* of polymorphic type ⟨Mammal⟩ → Real, and that the weight function has already been implemented for all current types of Mammal. When a new type is added to Mammal, the compiler should check that the weight function is defined for it. To do that, the compiler must rely on information that is part of the signature. That is, the compiler relies on the formal view of a genus. The programmer might also need to be aware that adding type S to Mammal requires him or her to define the weight function. So the programmer, too, relies on the formal view.

Specifying the hierarchy

The formal view of genera induces an implicit hierarchy based on signatures. But most languages require a programmer to indicate explicitly how he or she wants the hierarchy to look, and the actual hierarchy is built up from that description. We will use the term *explicit hierarchy* for the relationships indicated by a program. The explicit hierarchy does not have to describe every possible relationship. For example, it is conceivable that type S has a Print procedure, but, for some reason, is not beneath Printable in the explicit hierarchy. It is usually up to the compiler to check that the explicit hierarchy is consistent with the implicit hierarchy induced by signatures, so that everything that must be defined is defined. That is, if $B \le A$ in the explicit hierarchy, then everything described in the signature of A is also available for B.

A Cinnameg program defines a hierarchy by creating genera and indicating relationships among them. The following creates two new genera Mammal and Bear, and declares that Bear is beneath Mammal. If you need to say several things that need to be taken together, then they need to be enclosed in Extension … %Extension, as is done here.

```
Extension
  Genus Mammal.                %% Creates genus Mammal
  Genus Bear.                  %% Creates genus Bear
  Relate Bear isKindOf Mammal. %% Bear is beneath Mammal
%Extension
```

You do not need to say which types belong to Mammal and Bear in order to establish that Bear is beneath Mammal; the hierarchy can be modified later by adding new things. But the ≤ relation must be transitive, so if you later add a type to Bear, that type is implicitly beneath Mammal as well. The following creates a type BlackBear that lies beneath Bear in the hierarchy.

```
Extension
  Type BlackBear interface.
  Relate BlackBear isKindOf Bear.
%Extension
```

Specifying the signature

Each language that allows the equivalent of genera has a way to specify the signature of each genus. In Cinnameg, you specify the signature by writing Expect or Anticipate declarations. For example, declaration

```
Anticipate Print: <Printable> -> ().
```

indicates that there should be a definition of Print for every type of the form $T \to (\,)$, where T is beneath genus Printable. Alternatively, write

```
Expect Print: <Printable> -> ().
```

The difference between Expect and Anticipate is one of responsibility. Anticipate makes it the responsibility of anyone who adds a type to Printable to define Print for that type. Expect says that such a definition is given here that will work for all types in genus Printable, even those that are added later. Using the terminology of the next section, Anticipate calls for ad-hoc definitions, while Expect calls for parametric definitions.

36.5. Defining polymorphic operations

The signatures of types and genera, along with the genus/type hierarchy, tell you which operations *exist*. For example, saying that Print has polymorphic type $\langle\text{Printable}\rangle \to (\,)$ tells you that there exists a procedure called Print for each type $T \to (\,)$, where T belongs to Printable. But it does not tell you what Print does, or how it is defined. That is a separate issue. There are two common ways to make definitions for polymorphic functions and procedures.

Ad-hoc polymorphic definitions

One way to make polymorphic definitions is to define each function or procedure separately for each type. Suppose, for example, that each type of Mammal represents information about itself in a different way. Then the weight function needs to be implemented in a different way for each type of mammal. Ad-hoc implementation allows you to write the definitions as you see fit.

Genus Number offers another example. Suppose that Number includes two types, Real and Integer. Certainly, there are some operations in common between Real and Integer, and those operations should be polymorphic. For instance, you can add or multiply two integers or two real numbers. But the + and * operations will need to be implemented in one way for integers and a different way for real numbers, so the definition will need to be ad-hoc.

Ad-hoc polymorphic definitions are common in the setting of object-oriented programming, where subprograms defined in an ad-hoc manner are called *virtual* subprograms. Chapter 42 explores this idea further in the context of object-oriented programming.

Parametric polymorphic definitions

Return to the Mammal example, and imagine that function weight has already been defined for each kind of mammal. Suppose you need to define a new function called big, where a mammal is defined to be big if its weight is more than 10. That new function is easily written in terms of the weight function. What is more, it only needs to be written once, and will continue to work, even if another type of mammal is created later. The code that implements big is shared by several types, and is called a *parametric* definition. Think of it as implicitly parameterized by the type on which it works, so that it can request a definition of weight that works on the same type.

As another example, suppose that genus Order supports order tests $<$, $>$, $\leq$, $\geq$, $=$ and $\neq$. To add a new type to Order, it appears to be necessary to write six operations for that type. But that can be reduced substantially by adding one more operation, compare, to the signature of Order, where compare is defined by

$$\text{compare}(x, y) = \begin{cases} -1 & \text{if } x < y \\ 0 & \text{if } x = y \\ 1 & \text{if } x > y. \end{cases}$$

Then operations such as $<$ and $>$ can be defined parametrically in terms of compare, and only need to be written once. For example,

$$x < y = (\text{compare}(x, y) = -1)$$

When a new type is added to Order, only the compare function needs to be defined for it, probably in an ad-hoc way.

36.6. Implementation of polymorphism

Continuing to look deeper at polymorphism, there is the issue of how polymorphic definitions are dealt with by the language implementation, either a compiler or an interpreter.

Parametric implementation

One form of implementation is used in cases where polymorphic definitions are made parametrically, so we will call it a *parametric implementation* of polymorphism. It uses a single implementation of each polymorphic thing, and that one implementation must work for all possible types. The empty list [ ] is polymorphic, since it can be any type of list. If lists are implemented in linked representation, then the empty list is implemented as a null pointer, and every empty list, regardless of its type, is represented by a null pointer. The null pointer is a parametric implementation of the empty list.

Polymorphic functions or procedures that are defined parametrically work in the same way regardless of the type of data on which they are working. It is the executable code of a function that is the single implementation of the function. For example, function $f(x) = (\text{head}(x), \text{tail}(x))$ works by computing the head and tail of x and building an ordered pair, and it does that regardless of the type of the members of list x.

In the case where all polymorphic definitions are parametric, polymorphism can be thought of as a way of attaching types to a typeless language. The function and procedure definitions, for example, could all be written in a typeless language just as well as in a statically typed language. Adding types to the language does nothing but allow a compiler to check that the definitions are sensible. The types do not change the definitions at all, and do not affect how they work. With parametric implementation, the type system is simply recognizing, checking and quantifying the polymorphism that was already present in a set of definitions.

Ad-hoc implementation

An alternative implementation of polymorphism takes the viewpoint that it is the names, not the items themselves, that are polymorphic. *Ad-hoc implementation of polymorphism* does not require shared representations. Instead, it uses a separate implementation for each type. For example, you can define a function called sqr that does one kind of operation on integers and another kind of operation on real numbers. You can design a Print procedure so that it prints a character as a character, prints an integer in decimal notation, prints a list of characters as a string, a list of integers with square brackets around it, and so on. Ad-hoc polymorphism is very flexible in how meanings are attached to polymorphic names.

In order to make this work, it is usually necessary to have a table where implementations can be found. For example, you can look up the implementation of function "insert", of type (Integer, [Integer]) → [Integer]. What you get is the code that performs insert for that type. A table of implementations is often called a *dispatch table.*

Ad-hoc implementation of polymorphism is more expensive to implement than parametric implementation because of the space used to store multiple implementations and the dispatch table. But entries in the dispatch table can point to parametrically defined code, so that several entries in the dispatch table point to the same place. That reduces the storage requirements, and makes it possible to combine ad-hoc and parametric definitions. That combination is used for object-oriented programming languages and languages such as Cinnameg, which support both ad-hoc and parametric definitions.

36.7. Type checking

In a typical statically typed monomorphic language, such as Pascal, the type checker makes sure that certain types match. For example, if you write expression $f(x)$, and f takes a parameter of type String, then the compiler ensures that x has type String; otherwise there is an error.

But polymorphic languages require more subtle type checking. Suppose that polymorphic function f takes a parameter of "type" $\langle A\rangle$, where A is a genus, and suppose that expression $f(x)$ occurs in a program, where x has "type" $\langle B\rangle$. Then the compiler only has to ensure that $B \le A$ rather than $B = A$. For example, suppose that function numberOfTeeth has polymorphic type $\langle$Mammal$\rangle \rightarrow$ Integer. If b has type BlackBear, then numberOfTeeth(b) should be well-typed, since b is a special kind of Mammal. Similarly, if v is a variable that is supposed to hold a Mammal, then a program should be allowed to store a value of type BlackBear into v, since BlackBear $\le$ Mammal.

Relationships among structured types

The concept of the hierarchy can be extended to structured types. For example, suppose that function g takes an ordered pair as a parameter. If its polymorphic type is ($\langle$Any$\rangle$, $\langle$Any$\rangle$) $\rightarrow$ Integer, then g should be able to take an actual parameter of type (Integer, String). In general, say that $(A, B) \le (C, D)$ when $A \le C$ and $B \le D$. For example, (Integer, String) $\le$ ($\langle$Any$\rangle$, $\langle$Any$\rangle$). With that definition of $\le$ for product types, the compiler still just checks that the formal parameter type A (such as ($\langle$Any$\rangle$, $\langle$Any$\rangle$)) and the actual parameter type B (such as (Integer,String)) satisfy $B \le A$.

The $\le$ relationship can also be extended to lists and functions. For lists, it should be clear that $[B] \le [A]$ just when $B \le A$. For example, a list of integers is a special case of a list of numbers, and you should be able to use a list of integers in a place where a list of numbers is required. Since list types $[A]$ and $[B]$ are ordered in the same way as their member types, A and B, list types are said to be *covariant*.

Generally, we use $\langle G\rangle$ to indicate an arbitrary type that belongs to G. When describing relationships, we will normally avoid the angle brackets, and just say, for example, that (Bear, Bear) $\le$ (Mammal, Mammal).

Relationships among functions

Some languages allow functions to be treated as values, and passed as parameters to other functions, so it is important to know how functions fit into polymorphism. Functions exhibit a different kind of relationship from lists or ordered pairs. Suppose that you have two functions gen and spec, where gen has polymorphic type $\langle$Number$\rangle \rightarrow$ Boolean, and spec has type Integer $\rightarrow$ Boolean. Since gen, the more general function, can take any kind of number as a parameter, it can be used any place where spec could (at least from the perspective of types). By the same reasoning that was used for pairs and lists, you should expect that (Number $\rightarrow$ Boolean) $\le$ (Integer $\rightarrow$ Boolean). Recall that, in type $A \rightarrow B$, A is called the domain type and B is called the codomain type. Functions are ordered in the opposite order as their domains, and, in general, $(A \rightarrow B) \le (C \rightarrow D)$ just when $C \le A$ and $B \le D$. Functions are covariant in their codomains, but *contravariant* in their domains, since they have the opposite order as their domains.

Relationships among box types

Boxes (variables, Chapter 15) need some thought. Using Cinnameg notation, we will write [: T :] for the type of a box that holds something of type T. Suppose that Bear $\le$ Mammal. How should [:Bear:] and [:Mammal:] be related? Boxes have two fundamental operations: fetching and storing. If you fetch a value from a box of type [:Mammal:], you do not necessarily get a Bear, so it should not be the case that [:Mammal:] is a special case of [:Bear:]. But you cannot store a general Mammal into a box of type [:Bear:], so, for the store operation, [:Bear:] is not a special case of [:Mammal:] either. The most reasonable thing is to say that boxes are *invariant*; [: A :] $\leq$ [: B :] only when $A = B$. That prevents the type checker from allowing a box to be used in a context (either a fetch or a store) where it would not make sense.

Polymorphism in the codomain

A function of polymorphic type ⟨Any⟩ → String has ⟨Any⟩ in the domain of its type, indicating that it can take any type of parameter. But what about a function that has ⟨Any⟩ in the codomain of its type? Suppose, for example, a particular function g has polymorphic type String → ⟨Any⟩. There are two obvious interpretations of what that means, depending on who chooses the type indicated by ⟨Any⟩.

In one view, the program (or, if you like, the programmer) makes the choice of type. The program says "Today I would like g("321") to be an integer," and function g must produce an integer. In this view, g is told the desired result type, and must produce a result with that type. Function g might be able to do that if g takes its own type as an implicit parameter. After inspecting its own type, g sees that it is being asked to produce an integer, and takes appropriate action. The Cinnameg library function *get* works in that way. Its type is String → (⟨Any⟩, String): given a string s, get tries to extract a prefix of s and to convert it into the desired type; get then returns pair (v, r) where v is the extracted value and r is the rest of the string. For example, (get("25 abc"): (Integer, String)) yields (25, " abc"), but (get("true abc"): (Boolean, String)) yields (true, " abc"). A function that allows the program to determine its result type is called a statically typed function in Cinnameg.

But there is a second viewpoint that is often more what you want. In the alternative view, it is the function, g, who chooses the result type, not the programmer. So the type of g("321") would be *some* type, at the discretion of g itself. The result will need to have a tag on it indicating its type. Notice the difference, in this view, between polymorphism in the domain of a function and polymorphism in the codomain of a function. In the domain of a function, ⟨Any⟩ indicates that the function can handle *every* type. But when ⟨Any⟩ occurs in the codomain, it indicates that *some* type of result is produced. Cinnameg supports this kind of function, calling it a dynamically typed function[3]. If the get function were dynamically typed, then expression get("24 abc") *might* produce (24, " abc"), of type (Integer, String), but it might equally well produce ("24", " abc"), of type (String, String), depending on how get is written.

The choice of statically typed or dynamically typed functions influences type checking. When statically typed functions are used, type checking can, at least conceivably, be carried out at compile time. For example, if function f requires an integer parameter, then expression $f(g(\text{"321"}))$ can be forced to be well-typed; g is simply asked to produce an integer. But when a dynamically typed function g is used, then it is only possible to see whether $f(g(\text{"321"}))$ is well-typed at run time, when the result produced by g is available. That forces the language to be weakly statically typed.

36.8. Difficulties with type checking

The polymorphism needed to support the Print procedure can be understood quite well by introducing ⟨Any⟩. Some other functions can be dealt with well in a similar manner. For

[3] To request a dynamically typed function in Cinnameg, write {--dynamic--} immediately after Define.

example, the function that takes the length of a list has polymorphic type $[\langle \text{Any}\rangle] \to \text{Integer}$, since it takes a list whose members can be of any type at all, and produces an integer. But what about the function that takes the head of a list? A first attempt at describing its polymorphic type would be $[\langle \text{Any}\rangle] \to \langle \text{Any}\rangle$. That is, the head function takes a list whose members can have any type at all, and produces a value whose type can also be anything at all. Unfortunately, something is lost. Suppose that you take the head of a list of integers. What is the type of the result? From a logical point of view, it must be Integer. But the type of head says only that it is a member of genus Any. Neither statically typed nor dynamically typed functions appear to correctly capture what head does. The proposed type $[\langle \text{Any}\rangle] \to \langle \text{Any}\rangle$ of head just does not convey the fact that if you take the head of a list whose type is $[T]$, you get a result of type T, for every T. But that information is critical to type checking. To drive the point home, consider two different functions f_1 and f_2, defined as follows.

$$f_1(x, y) = (\text{head}(x), \text{head}(y)) \tag{36.8.1}$$

$$f_2(x, y) = (\text{head}(y), \text{head}(x))) \tag{36.8.2}$$

Both f_1 and f_2 appear to have polymorphic type $([\langle \text{Any}\rangle], [\langle \text{Any}\rangle]) \to (\langle \text{Any}\rangle, \langle \text{Any}\rangle)$. Suppose that x has type [Integer], and y has type [String]. If left(p) is the left-hand member of ordered pair p, then expression left$(f_1(x, y))+1$ is well-typed, but expression left$(f_2(x, y))+1$ is poorly typed, and should be an error. How can a compiler know, from the (identical) types of functions f_1 and f_2, that one of those expressions is well-typed, while the other is not?

To get around this problem, some languages assume that head is dynamically typed, and programmers are forced to use *down-conversions*, which convert from a genus A to a *lower* genus or type B. For example, a down-conversion might convert from $\langle \text{Any}\rangle$ to Integer. To take the head of a list of integers, you might write something like

```
h = (to Integer) head(x)
```

explicitly indicating that the type of head(x) is Integer. The compiler inserts a run-time type check to ensure that the programmer is correct.

An alternative is to supply more information in polymorphic types, so that the compiler can realize that, if x is a list of integers, then head(x) must be an integer, making stronger static type checking possible. The next chapter explores a way of doing that.

36.9. Summary of terminology and concepts

Polymorphism is critical to a typed language that is intended to support general-purpose libraries.

One way to understand polymorphism is to separate the signature of an abstract data type from its implementation and to allow multiple implementations. Then any function that uses the abstract data type will work with all of its implementations, and so is polymorphic. The degree to which a signature or definition is polymorphic is determined by the genus/type hierarchy, relating genera by the subset relation. A signature that indicates it works for genus G must work for all types that belong to G.

When you promise, via a signature, to implement a given function for many different types, you can either fulfill your promise by defining it separately for each type (ad-hoc polymorphic definition) or by writing a single definition that works for all promised type (parametric polymorphic definition) or by a combination of the two, where several parametric definitions, taken together, cover all of the promised types. Ad-hoc implementations of polymorphic definitions are more flexible in how polymorphic definitions are made than are parametric implementations, but they are also more expensive to manage.

Type checking needs to take polymorphism into account, and the existence of dynamically typed polymorphic functions can lead to run-time type checks and a weakly statically typed language.

Terminology

Ad-hoc polymorphism and definition In the absence of signatures that describe polymorphic types, ad-hoc polymorphism is equivalent to overloading. Each use of a polymorphic procedure or function is assigned a single, monomorphic, type by the compiler, and ad-hoc polymorphic functions cannot be used to make polymorphic definitions of other functions. But when a signature is given, definitions for particular types can be made separately, even though the compiler treats the procedure or function as polymorphic, and allows it to be used to define other polymorphic procedures or functions.

Ad-hoc implementation of polymorphism An ad-hoc language implementation of polymorphism uses a dispatch table to find definitions, which can be ad-hoc (a separate definition for each type) or parameteric (one definition covering many types).

Coercion A coercion is an implicit conversion from one type to another. It can induce an apparent polymorphism, since any type that will be coerced to the required type for a given setting can be used there.

Contravariant A list type, $[T]$, is an example of a parameterized type; you can think of it as List(T). A parameterized type $P(T)$ is contravariant if, in the hierarchy, $P(A) \le P(B)$ whenever $B \le A$. Contravariance can occur on just some components of parameters as well. For example, function types are contravariant in their domains, since $A \to T \le B \to T$ when $B \le A$.

Covariant A parameterized type $P(T)$ is covariant if, in the hierarchy, $P(A) \le P(B)$ whenever $A \le B$. Lists are typically covariant, since a list of mammals is a special case of a list of animals. Function types are covariant in their codomains, since $T \to A \le T \to B$ when $A \le B$.

Dispatch table The dispatch table is used to support ad-hoc implementation of polymorphism. You look up the implementation of a given procedure or function, by name, of a given type.

Down-conversion A down-conversion converts a value to a type lower in the genus/type hierarchy. It requires run-time type checking, and can fail if types do not match at run time.

Genus A genus can be thought of as a set of types or as a signature shared by a collection of types.

Hierarchy The genus/type hierarchy is a partial order expressing the containment relationships of genera and memberships of types in genera.

Invariant A parameterized type $P(T)$ is invariant if $P(A) \leq P(B)$ only when $A = B$.

Overloading An identifier is overloaded when it has more than one meaning. The compiler must be able to determine (at compile time) which meaning is being used at any given occurrence of an identifier.

Polymorphism Polymorphism refers to the ability of a single name or definition to have several different types.

Parametric definition A parametric definition of a polymorphic function f defines f in terms of other polymorphic functions, and can be shared by many different types.

Parametric implementation of polymorphism Parametric implementation of polymorphism creates just one implementation of a parametric definition, typically by compiling it once and producing code that can work for any type of parameters.

Partial order A partial order is a relation $\leq$ that has the following properties for every x, y and z.

1. $x \leq x$.
2. If $x \leq y$ and $y \leq x$ then $x = y$.
3. If $x \leq y$ and $y \leq z$ then $x \leq z$.

If you define $A \leq B$ when genus A is a beneath B, then the $\leq$ relation satisfies the requirements of a partial order. The subset relation on a collection of sets is a partial order. The order is "partial" because there can be two things, X and Y, such that neither $X \leq Y$ nor $Y \leq X$ is true. For example, choose $X = \{1, 2\}$ and $Y = \{2, 3\}$. Neither is a subset of the other.

Type A type is a particular implementation of the concepts of a genus, or abstract data type.

Virtual function A virtual function is a polymorphic function whose definition is made in an ad-hoc manner.

36.10. Exercises

36.1. Explain why overloading is not usually regarded as genuine polymorphism in a programming language.

36.2. How does coercion lead to apparent polymorphism?

36.3. Section 36.2 indicates that Pascal's sqr function is overloaded, allowing it to square either an integer or a real number. Explain why overloading is a better explanation of sqr than coercion. (Pascal will coerce an integer to a real number, but it will not coerce a real number to an integer.)

36.4. Quite a few different coercions have been employed in programming languages. For example, some languages coerce real numbers to integers (by either truncating after the decimal point or rounding); some coerce strings of digits to numbers, and numbers to strings of digits; and some coerce a character to a string of length one.

Give an argument, with examples, for one of the following two positions. (1) Having a lot of sensible coercions frees programmers from having to think so much about types, allowing him or her to write whatever seems to make sense. (2) Having a large number of coercions can actually put more of a burden on a programmer, forcing him or her to think even more carefully about types, than would be necessary if there were no coercions at all.

36.5. Why is polymorphism important to a programming language? What feature of modern software development environments is most responsible for the need for polymorphism in programming languages?

36.6. Is the polymorphic definition of function cat given by equations (18.8.6) parametric or ad-hoc?

36.7. Give an example where ad-hoc polymorphic definitions appear to be necessary.

36.8. Assume that Bear $<$ Mammal in a particular hierarchy. (We say that $A < B$ if $A \leq B$ and $A \neq B$.) Which of the following is true?

(a) Bear $\leq$ Bear
(b) (Bear,Bear) $\leq$ (Bear,Mammal)
(c) (Bear,Mammal) $\leq$ (Mammal, Bear)
(d) (Bear $\rightarrow$ Bear) $\leq$ (Bear $\rightarrow$ Mammal)
(e) (Bear $\rightarrow$ Bear) $\leq$ (Mammal $\rightarrow$ Bear)

36.9. Extend the rule for relationships among ordered pairs to relationships among arbitrary tuples. When is $(A_1, \ldots, A_n) \leq (B_1, \ldots, B_n)$?

36.10. If $B \leq A$ in the formal hierarchy, would you more likely find that A has more operations in its signature than B, or that B has more operations in its signature than A?

36.11. After answering question 36.10, explain how the $\leq$ order of genera is similar to the subtype relationship on records described in Section 9.6.

36.12. Under what circumstances does a programmer typically need to insert a down-conversion into a program? Give an explanation and an example.

36.13. Suppose that there are only two basic types, Boolean and Integer, and that there are two type constructors, the pair constructor and the list constructor. So, for example, (Integer, [Boolean]) is an allowed type. Using only those, write a sensible definition of a function toString that converts an arbitrary thing to a string. For example, toString((2,true)) = "(2,true)", and toString([2,4,6]) = "[2,4,6]". Your definition of toString should be able to handle arbitrarily complex types built from those constructors. For example, toString((2,[1,5]), (true,false)) = "((2,[1,5]), (true,false))". Are you using ad-hoc definitions, parametric definitions, or a combination of the two?

Chapter 37

Polymorphism With Variables

37.1. Type variables

You are familiar from mathematics with variables that stand for arbitrary numbers, and with equations that express properties that such variables are required to have. For example, equation $x + 1 = 2y$ expresses a relationship between x and y. In this chapter, we introduce *type variables*, which stand for types, and we use equations to express properties that the type variables are required to have. We will use Greek letters, such as α and β, for type variables. For example, variable α might stand for Integer or String or any other type. Equation $\alpha = [\beta]$ expresses a relationship between the types that α and β stand for; if $\beta =$ Integer, then $\alpha =$ [Integer].

Type variables make it possible to supply more information about the type of a polymorphic function and to perform stronger static type checking. For example, the function that takes the head of a list has polymorphic type $[\alpha] \to \alpha$. So the head function can have type [Integer] → Integer, or [String] → String, or, in general, type $[T] \to T$ for any type T. Quite a few polymorphic functions, including the ones in the following list, are conveniently given polymorphic types using type variables.

1. Function tail has polymorphic type $[\alpha] \to [\alpha]$. To see that, look at expression tail(x). Clearly, x must be a list, but there is no constraint on what can be in that list. So x can be assigned type $[\alpha]$. The result of tail(x) must be the same type of list, so it also has type $[\alpha]$.

2. The length function takes an arbitrary kind of list and produces an integer. So its polymorphic type is $[\alpha] \to$ Integer. For this function, there is no need to express relationships between different parts of the type, but a type variable is still a natural way to express the polymorphic type.

3. Notice that 5 :: [3,6] = [5,3,6]. In this example, operator :: takes a pair consisting of an integer (5) and a list of integers ([3,6]), and it produces a list of integers. That is, the :: operator has type (Integer, [Integer]) → [Integer]. But operator :: is much more general than that. Its polymorphic type is $(\alpha, [\alpha]) \to [\alpha]$.

4. Suppose that transpose(x, y) = (y, x). Then transpose has polymorphic type $(\alpha, \beta) \to (\beta, \alpha)$. It would be too restrictive to say that the polymorphic type of transpose

$(\alpha, \alpha) \to (\alpha, \alpha)$, because that would say that transpose(3, true) is poorly typed. But transpose$(3, \text{true}) = (\text{true}, 3)$ makes perfectly good sense. Different variables are allowed to stand for different types, but are not required to be different. For example, transpose can work on a pair of integers.

5. Function zip takes a pair of lists and produces a list of pairs. For example, zip([2, 3, 4], ["abc", "def", "ghi"]) = [(2, "abc"), (3, "def"), (4, "ghi")]. The example shows zip being used with type ([Integer], [String]) → [(Integer, String)]. But zip can be used with any types of lists. The polymorphic type of zip is $([\alpha], [\beta]) \to [(\alpha, \beta)]$.

6. Functions f_1 and f_2 defined by equations (36.8.1) and (36.8.2) have different polymorphic types. They are f_1: $([\alpha], [\beta]) \to (\alpha, \beta)$ and f_2: $([\alpha], [\beta]) \to (\beta, \alpha)$.

To find the polymorphic type of a function, a good starting place is an example. After writing down the value that the function takes in and the value that it produces, try writing the domain and codomain types of the example. Then generalize, replacing types with variables. Choose an example that is as general as possible, using different types of things wherever possible.

Type variables in Cinnameg

Cinnameg uses notation $\langle x \rangle$ for a type variable called x, and, in general, allows a type variable to have any name, such as ⟨table⟩. The type of head in Cinnameg notation is $[\langle a \rangle] \to \langle a \rangle$. An anonymous variable, ⟨⟩, is a variable that is different from every other variable.

37.2. Type checking with type variables

Type variables offer a compiler extra information that allows it to perform a high degree of static type checking, even with polymorphism. This section describes an algorithm that a compiler can employ to determine the type of a function from its definition, and to ensure that a definition is well typed. The best place to start is an example. Function len computes the length of a list.

$$\texttt{len}([\,]) \quad = \quad 0 \qquad (37.2.1)$$

$$\texttt{len}(h :: t) \quad = \quad 1 + \texttt{len}(t) \qquad (37.2.2)$$

Presume, for simplicity, that + has type (Integer, Integer) → Integer and that 0 has type Integer. There are three phases to determining the polymorphic type of len.

1. Assign a different type variable to each identifier, constant and, in general, each subexpression of the definition. In cases where it is clear that two expressions must have the same type, you are free to choose the same variable, but the algorithm normally just uses a different variable for each type.

2. From the definition, determine equations that must be true in order for the definition to make any sense. Every subexpression of the definition has some type information associated with it. Rules for determining the equations are discussed below.

3. Determine values for the variables that make the equations found in phase (2) true, including which variables must be the same in order for the equations to be satisfied. That is, find a solution to the equations. Among all solutions, find the most general one (which restricts variables as little as possible), to allow as much polymorphism as possible. It is conceivable that no solution to the equations exists. In that case, the definition is poorly typed.

Phase 1 for the len function assigns type variables to the subexpressions that occur in the definition of len. Start with the identifiers and constants. In cases where the types are known (such as for ::, which has a known polymorphic type) attach the known type, but use different variables for each occurrence of the identifier or constant. Notation $A : T$ indicates that A has type T.

$$\begin{aligned}
0 &: \text{Integer}\\
1 &: \text{Integer}\\
\text{len} &: \alpha_1\\
h &: \alpha_2\\
t &: \alpha_3\\
[\,] &: [\alpha_4]\\
:: &: (\alpha_5, [\alpha_5]) \to \alpha_5\\
+ &: (\text{Integer}, \text{Integer}) \to \text{Integer}
\end{aligned}$$

Additional subexpressions that occur in the definition of len are as follows. Expression $h :: t$ is written $::\,(h, t)$, applying the :: function to ordered pair (h, t).

$$\text{len}([\,]) : \alpha_6 \tag{37.2.3}$$

$$(h, t) : \alpha_7 \tag{37.2.4}$$

$$::\,(h, t) : \alpha_8 \tag{37.2.5}$$

$$\text{len}(::\,(h, t)) : \alpha_9 \tag{37.2.6}$$

$$\text{len}(t) : \alpha_{10} \tag{37.2.7}$$

$$(1, \text{len}(t)) : \alpha_{11} \tag{37.2.8}$$

$$+(1, \text{len}(t)) : \alpha_{12} \tag{37.2.9}$$

Deriving equations

Phase 2 involves examining the subexpressions of the definition. Consider expression len([ ]), which occurs in Equation 37.2.1, and that has been assigned type α_6 by (37.2.3). The parameter [ ] of len in this expression has been assigned type $[\alpha_4]$, and len must produce a result whose type is α_6, since that is the type assigned to len([ ]). For this to make sense it should be the case that the type α_1 of len is the same as type $[\alpha_4] \to \alpha_6$. So write down the following equation.

$$\alpha_1 = [\alpha_4] \to \alpha_6$$

General rules for deriving type equations depend on the forms of expressions that a given programming language supports, but a few rules are easy to come up with.

1. Suppose that $f(a)$ is an expression that calls for applying function f to parameter a. Suppose that f has been assigned type α_f, a has been assigned type α_a and expression $f(a)$ has been assigned type α_e. Add equation $\alpha_f = \alpha_a \rightarrow \alpha_e$ to the collection of equations.

2. Suppose that expression (a, b) has been assigned type α_e, where a has type α_a and b has type α_b. Add equation $\alpha_e = (\alpha_a, \alpha_b)$.

3. Suppose that expression $(\texttt{if}\ a\ \texttt{then}\ b\ \texttt{else}\ c)$ has been assigned type α_e, where a has type α_a, b has type α_b and c has type α_c. Then add equations $\alpha_e = \alpha_b = \alpha_c$ and $\alpha_a = \text{Boolean}$.

Additionally, if a definition contains equation $A = B$, then the types of A and B must be the same. Inspection of each of the subexpressions and equations in the definition of len yields the following set of equations, each indicating the subexpression or equation that leads to it.

$$\begin{array}{rcll}
\alpha_1 & = & [\alpha_4] \rightarrow \alpha_6 & \text{(from (37.2.3))} \\
\alpha_7 & = & (\alpha_2, \alpha_3) & \text{(from (37.2.4))} \\
(\alpha_5, [\alpha_5]) \rightarrow \alpha_5 & = & \alpha_7 \rightarrow \alpha_8 & \text{(from (37.2.5))} \\
\alpha_1 & = & \alpha_8 \rightarrow \alpha_9 & \text{(from (37.2.6))} \\
\alpha_1 & = & \alpha_3 \rightarrow \alpha_{10} & \text{(from (37.2.7))} \\
\alpha_{11} & = & (\text{Integer}, \alpha_{10}) & \text{(from (37.2.8))} \\
\alpha_{11} \rightarrow \alpha_{12} & = & (\text{Integer}, \text{Integer}) \rightarrow \text{Integer} & \text{(from (37.2.9))} \\
\alpha_6 & = & \text{Integer} & \text{(from (37.2.1))} \\
\alpha_9 & = & \alpha_{12} & \text{(from (37.2.2))}
\end{array} \qquad (37.2.10)$$

Solving the equations

Finding a solution to the system of type equations generated in phase 2 can be done using a tool called *unification.*[1] Inspection of Equations (37.2.10) shows that the equations can, in general, have arbitrarily complex types on each side. Unification is an algorithm that takes a general equation and converts it into an equivalent collection of *substitutions*, where each substitution is an equation of form $\alpha_i = T$ for some type T, where α_i is a variable. Moreover, none of the variables that occurs on the left-hand side of one of the substitutions also occurs on the right-hand side of any of them, so replacing the variables (α_i by T, etc.) necessarily eliminates them. For example, unification will never produce set of equations $\{\alpha_1 = \alpha_2,\ \alpha_2 = \alpha_1\}$. Figure 37.1 gives an equational definition of the unification algorithm.

Start by converting the first equation to a set of substitutions. If unification succeeds, then perform those substitutions on all of the remaining equations, and also on the polymorphic type α_1 that has been assigned to len, and remove the first equation. That has the effect of eliminating one equation. Repeat until all of the equations have been eliminated, and all that is left is the result of all of the substitutions on α_1.

For example, the first two of Equations 37.2.10 are both already substitutions, and do not involve any common variables. Replacing each occurrence of α_1 by $[\alpha_4] \rightarrow \alpha_6$ and each occurrence of α_7 by (α_2, α_3) in all of the other equations (and in the type of len) yields

$$(\alpha_5, [\alpha_5]) \rightarrow \alpha_5 \quad = \quad (\alpha_2, \alpha_3) \rightarrow \alpha_8$$

[1] Unification is also studied in Part IX, on logic programming. Unification for polymorphic types is similar to unification used in logic programming, but the variant that includes the occur-check must be used.

$$\begin{aligned}
\text{unify}(v, v) &= \{\} \\
\text{unify}(v, A) &= \{v = A\} \text{ provided } v \text{ does not occur in } A \\
\text{unify}(A, v) &= \text{unify}(v, A) \text{ if } A\text{is not a variable} \\
\text{unify}(A \to B, C \to D) &= S \cup T \text{ where } S = \text{unify}(A, C) \text{ and } T = \text{unify}(S(B), S(D)) \\
\text{unify}((A, B), (C, D)) &= S \cup T \text{ where } S = \text{unify}(A, C) \text{ and } T = \text{unify}(S(B), S(D)) \\
\text{unify}([A], [B]) &= \text{unify}(A, B)
\end{aligned}$$

Figure 37.1: **Definition of unify**(A, B)**.** If equation $A = B$ has a solution then unify(A, B) yields a set of *substitutions* that is equivalent to equation $A = B$. Each substitution is an equation of the form $v = E$, where v is a variable. Also, none of the variables that occurs on the left-hand side of one of the substitutions also occurs on the right-hand side of any of them. For example, unify$(([\alpha], \beta),\ (\gamma, [\text{Integer}])) = \{\gamma = [\alpha],\ \beta = [\text{Integer}]\}$.

In the equations that define unify, v is a variable and A, B, C and D are arbitrary types. The equations should be tried in the order listed, and the first one that matches should be used. If none of these equations gives an answer for unify(A, B) then equation $A = B$ has no solution, and unification fails. That is true of the recursive calls as well. If unify(A,B) calls unify(C,D), and equation $C = D$ has no solution, then $A = B$ also has no solution.

If S is a set of substitutions, then $S(A)$ is the result of performing substitutions S on type A. For example, if $S = \{\alpha_1 = \alpha_5, \alpha_3 = (\alpha_5, \alpha_6)\}$ then $S(\alpha_1 \to (\alpha_3, \alpha_5)) = \alpha_5 \to ((\alpha_5, \alpha_6), \alpha_5)$.

$$\begin{array}{rcl}
[\alpha_4] \rightarrow \alpha_6 & = & \alpha_8 \rightarrow \alpha_9 \\
[\alpha_4] \rightarrow \alpha_6 & = & \alpha_3 \rightarrow \alpha_{10} \\
\alpha_{11} & = & (\text{Integer}, \alpha_{10}) \\
\alpha_{11} \rightarrow \alpha_{12} & = & (\text{Integer}, \text{Integer}) \rightarrow \text{Integer} \\
\alpha_6 & = & \text{Integer} \\
\alpha_9 & = & \alpha_{12} \\
\text{len} & : & [\alpha_4] \rightarrow \alpha_6
\end{array}$$

The first equation that is left is not a substitution, but the unification algorithm converts it into set of substitutions $\{\alpha_2 = \alpha_5,\ \alpha_3 = \alpha_5,\ \alpha_8 = \alpha_5\}$, and performing those substitutions on the remaining equations yields

$$\begin{array}{rcl}
[\alpha_4] \rightarrow \alpha_6 & = & \alpha_5 \rightarrow \alpha_9 \\
[\alpha_4] \rightarrow \alpha_6 & = & \alpha_5 \rightarrow \alpha_{10} \\
\alpha_{11} & = & (\text{Integer}, \alpha_{10}) \\
\alpha_{11} \rightarrow \alpha_{12} & = & (\text{Integer}, \text{Integer}) \rightarrow \text{Integer} \\
\alpha_6 & = & \text{Integer} \\
\alpha_9 & = & \alpha_{12} \\
\text{len} & : & [\alpha_4] \rightarrow \alpha_6
\end{array}$$

The first of the remaining equations can be converted into substitutions $\{\alpha_4 = \alpha_5,\ \alpha_6 = \alpha_9\}$, yielding

$$\begin{array}{rcl}
[\alpha_5] \rightarrow \alpha_9 & = & \alpha_5 \rightarrow \alpha_{10} \\
\alpha_{11} & = & (\text{Integer}, \alpha_{10}) \\
\alpha_{11} \rightarrow \alpha_{12} & = & (\text{Integer}, \text{Integer}) \rightarrow \text{Integer} \\
\alpha_9 & = & \text{Integer} \\
\alpha_9 & = & \alpha_{12} \\
\text{len} & : & [\alpha_5] \rightarrow \alpha_9
\end{array}$$

The process continues until all that is left is the last line showing the type of len. You might end up with len: $[\alpha_5] \rightarrow$ Integer. The only thing that can be different in your solution is the name of the variable α_5. (Which variable you end up with depends on the details of how the unification algorithm chooses substitutions. When unifying two variables α and β, you can either replace α by β or replace β by α.) But what you choose to call a variable does not matter.

Another sample type analysis

Define

$$f_1(x, y) = (\text{head}(x), \text{head}(y)). \tag{37.2.11}$$

A type analysis (or type *inference*) of the definition of f_1 starts by choosing type variables for the subexpressions. There is an important point, though, that comes up here. Identifier head occurs twice. But it is a polymorphic function, and there is no reason (at least to start) to think that the two occurrences of head must be used with the same type. So each occurrence is given a type that is a copy, with different type variables, of the polymorphic type of head. To distinguish the two, we write head_1 for the first occurrence of head and head_2 for the second occurrence. Since we do not know in advance the types of parameters or of the function f_1 being defined, we have no polymorphic types to copy, so we use the same type variable for every occurrence of them.

$$\begin{aligned}
f_1 &: \alpha_1 && (37.2.12)\\
x &: \alpha_2 && (37.2.13)\\
y &: \alpha_3 && (37.2.14)\\
(x, y) &: \alpha_4 && (37.2.15)\\
f_1(x, y) &: \alpha_5 && (37.2.16)\\
\text{head}_1 &: [\alpha_6] \to \alpha_6 && (37.2.17)\\
\text{head}_1(x) &: \alpha_7 && (37.2.18)\\
\text{head}_2 &: [\alpha_8] \to \alpha_8 && (37.2.19)\\
\text{head}_2(y) &: \alpha_9 && (37.2.20)\\
(\text{head}_1(x), \text{head}_2(y)) &: \alpha_{10} && (37.2.21)
\end{aligned}$$

The equations are as follows.

$$\begin{array}{rcll}
\alpha_4 & = & (\alpha_2, \alpha_3) & \text{(from (37.2.15))}\\
\alpha_1 & = & \alpha_4 \to \alpha_5 & \text{(from (37.2.16))}\\
[\alpha_6] \to \alpha_6 & = & \alpha_2 \to \alpha_7 & \text{(from (37.2.18))}\\
[\alpha_8] \to \alpha_8 & = & \alpha_3 \to \alpha_9 & \text{(from (37.2.20))}\\
\alpha_{10} & = & (\alpha_7, \alpha_9) & \text{(from (37.2.21))}\\
\alpha_5 & = & \alpha_{10} & \text{(from (37.2.11))}
\end{array}$$

The first two equations are already substitutions. Eliminating them yields the following.

$$\begin{array}{rcl}
[\alpha_6] \to \alpha_6 & = & \alpha_2 \to \alpha_7\\
[\alpha_8] \to \alpha_8 & = & \alpha_3 \to \alpha_9\\
\alpha_{10} & = & (\alpha_7, \alpha_9)\\
\alpha_5 & = & \alpha_{10}\\
f_1 & : & (\alpha_2, \alpha_3) \to \alpha_5
\end{array}$$

Unification can convert the first two of the remaining equations into substitutions $\{\alpha_2 = [\alpha_6], \alpha_7 = \alpha_6, \alpha_3 = [\alpha_8], \alpha_9 = \alpha_8\}$, and performing those substitutions yields the following.

$$\begin{array}{rcl}
\alpha_{10} & = & (\alpha_6, \alpha_8)\\
\alpha_5 & = & \alpha_{10}\\
f_1 & : & ([\alpha_6], [\alpha_8]) \to \alpha_5
\end{array}$$

Finally, replacing both α_5 and α_{10} by (α_6, α_8) as suggested by the equations yields

$$f_1 : ([\alpha_6], [\alpha_8]) \to (\alpha_6, \alpha_8).$$

Variable names are arbitrary. Choosing simpler names α for α_6 and β for α_8 gives

$$f_1 : ([\alpha], [\beta]) \to (\alpha, \beta).$$

37.3. Putting limits on polymorphism

Assume that a language has two numeric types, Integer and Real. A function that squares a number should probably be polymorphic, so that sqr of an integer produces an integer, but sqr of a real number produces a real number. Initially, a type of $\alpha \to \alpha$ seems appropriate for sqr. Unfortunately, that is too general, since variable α can stand for any type, including String or Boolean, but sqr only works on numbers.

Recall the notion of a genus from Chapter 36; we will adopt the viewpoint that a genus is a set of types. If there is a genus Number that contains types Integer and Real, then the polymorphic type of sqr can be expressed as a polymorphic type with a constraint on its variable: ($\alpha \to \alpha$ where $\alpha \leq$ Number). Cinnameg uses that approach. When you write a variable, you can indicate the genus telling the values that the variable can take on. Variable $\langle x$: Number$\rangle$ is a variable called x, but which can only stand for a type in genus Number. You can write the type of sqr as $\langle a$: Number$\rangle \to \langle a$: Number$\rangle$, or, if you prefer, you can express the variable constraint in a separate phrase, writing $\langle a\rangle \to \langle a\rangle$ `where` ($\langle a$: Number$\rangle$).

Type inference with restricted variables

Allowing variables to have associated genera requires some changes to the unification algorithm employed during type inference.

1. If two variables $\langle a : G\rangle$ and $\langle b : H\rangle$ must be equal, the compiler needs to find the intersection I of genera G and H, and to replace both $\langle a : G\rangle$ and and $\langle b : H\rangle$ by a new variable, $\langle c : I\rangle$. The intersection is chosen because any type T that can both be the value of $\langle a : G\rangle$ and of $\langle b : H\rangle$ must be in both G and H. If I is an empty set, then there is a type error. You can write the intersection of G and H as G&:H in a Cinnameg program.

2. If $\langle a : G\rangle$ needs to be replaced with a type T, then the compiler needs to check that T is a member of G; otherwise it is a type error.

3. Generalizing the prior rule, suppose that variable $\langle a : G\rangle$ needs to be replaced with a polymorphic type T, where T contains other type variables. The compiler should ensure that T will be a member of G no matter how variables are replaced later during type inference. That requires a few rules that are illustrated by the following example.

   Genus Order holds types that support order tests such as $<$ and $>$. Lists are ordered lexicographically, a generalization of alphabetical order of strings. In general, type $[T]$ is in Order just when T is in Order. Consequently, when it replaces variable $\langle a$: Order$\rangle$ by $[\langle b\rangle]$, the compiler also creates a new variable $\langle c$: Order$\rangle$, and adds equation $\langle b\rangle = \langle c$: Order$\rangle$.

37.4. Higher order functions

(This section uses concepts from Chapter 21. It can be skipped.) Higher order functions can also be given polymorphic types. Suppose that function pr is defined in Cinnameg by

```
Define (pr x) y = (y,x).
```

$$\begin{aligned} f &: \alpha_1 \\ x &: \alpha_2 \\ y &: \alpha_3 \\ f\ x &: \alpha_4 \\ (f\ x)\ y &: \alpha_5 \\ y\ x &: \alpha_6 \end{aligned}$$

Figure 37.2: Assignment of type variables to the subexpressions in definition (37.4.1).

There should not be any restriction on what types x and y can have. So use α for the type of x and β for the type of y. Notice that (pr x) is a function, since the definition says what (pr x) produces when it is applied to y. So pr takes parameter x, of type α, and produces a function. If F is the type of that function, then the type of pr is $\alpha \to F$.

Function (pr x) takes y and produces ordered pair (y, x). So its type is $\beta \to (\beta, \alpha)$. That is, type F is $\beta \to (\beta, \alpha)$. So the type of pr is $\alpha \to (\beta \to (\beta, \alpha))$. Since $\to$ by convention associates to the right, the type of pr can also be written, with fewer parentheses, as $\alpha \to \beta \to (\beta, \alpha)$.

Intuitive analysis of map

Recall that (map f $[a, b, c]$) = $[f(a), f(b), f(c)]$. To find the polymorphic type of map, try an example. For example (map length [”koala”, ”kangaroo”]) = [5,8].

Since map is a curried function, it makes perfectly good sense to write just map(length), giving map only the length function as a parameter. The length function in the example has type String → Integer. So the domain of map must be String → Integer, and map has type (String → Integer) → C for some codomain type C. The codomain of map can be found by looking at what function (map length) does. It takes parameter [”koala”, ”kangaroo”] and produces list [5,8]. So (map length) is a function of type [String] → [Integer]. Filling that in for C yields a type (String → Integer) → ([String] → [Integer]) for map.

But map is more general than that. There is nothing special about strings and integers here. Replacing them by separate variables yields a polymorphic type of $(\alpha \to \beta) \to ([\alpha] \to [\beta])$ for map. From its polymorphic type, you can see that map is a transformer of functions. You give it a function of type $\alpha \to \beta$, and it gives you another function of type $[\alpha] \to [\beta]$.

Algorithmic analysis of a higher order function

Intuitive type analysis, such as done above for map, often works (for a person, not a computer), but it is easy to make mistakes. The general type inference algorithm will work for higher order functions, without the need for ad-hoc thought. As an example, define function f as follows.

$$(f\ x)\ y = y\ x \tag{37.4.1}$$

Figure 37.2 shows phase 1, assignment of type variables to identifiers and subexpressions. Phase 2 yields a collection of equations, shown in Figure 37.3. In this case, all of the

Subexpression	**Equation**
$f\ x$	$\alpha_1 = \alpha_2 \to \alpha_4$
$(f\ x)\ y$	$\alpha_4 = \alpha_3 \to \alpha_5$
$y\ x$	$\alpha_3 = \alpha_2 \to \alpha_6$
$(f\ x)\ y = y\ x$	$\alpha_5 = \alpha_6$

Figure 37.3: Derivation of type equations from the subexpressions of definition (37.4.1).

equations are already substitutions, and performing them in sequence replaces α_1 by $\alpha_2 \to ((\alpha_2 \to \alpha_6) \to \alpha_6)$, so that is the polymorphic type of f. There are just two variables in the result, and what they are called is not relevant. Choosing shorter names $\alpha = \alpha_2$ and $\beta = \alpha_6$ yields type $\alpha \to ((\alpha \to \beta) \to \beta)$ for f. Since → associates to the right, the type of f can be written $\alpha \to (\alpha \to \beta) \to \beta$.

37.5. Issues concerning type inference

Polymorphic local definitions

We have not yet looked at an example that makes a local polymorphic definition within a function definition. Figure 37.4(a) illustrates one. Notice that g is defined (once) inside the definition of f but used twice, with different types at the two uses. For example, if x and y have type Integer, then the occurrence of g in expression $g(y)$ has type Integer $\to$ (Integer, Integer). But the other use of g (the first one in $g(g(y))$) takes the value of type (Integer, Integer) and produces a value of type (Integer, (Integer, Integer)). So, for example, $f(2,5) = (2,(2,5))$. Type inference needs to take into account the possibility of using a local definition with more than one type.

A conceptually simple way to handle local polymorphic definitions is to make them look like nonlocal definitions, and then handle them as we did other nonlocal definitions. We describe how to do that for a definition that has a single defining equation. The idea easily generalizes to multiple equations. Also, definitions are presumed not to be modifiable, so there are no "relets" that change the binding of an identifier; and, although local definitions can be recursive, we will not handle the case where there are mutually recursive local definitions.

Imagine a definition $f = d$ where d is an expression. Typically, it is function definitions that are polymorphic, and expression d might have the form (P`|->`b) for some list of parameters P and function body b. There can also be embedded functions. For example, d might be (P_1 `|->` P_2 `|->` b). Any variables defined on the left-hand side of `|->` is called a formal parameter in the following.

1. Suppose that f is defined by equation $f = d$, where d contains at least one local definition. Find a definition $g = e$ in d where e does not contain any other locally defined names except possibly formal parameters, and possibly g itself. There must be such a local definition: just pick the first one that is made. (Recall that we are not handling definitions that are mutually recursive.)

2. If expression e contains no formal parameters, then the definition of g is not really tied

```
Define f(x,y) =
  Define g(w) = (x,w).
  (g(g(y)))
%Define
```

(a)

```
Define f = ((x,y) |->
  Define g = (w |-> (x,w)).
  (g(g(y))))
%Define
```

(b)

```
Define g = (x |-> (w |-> (x,w))).
Define f = ((x,y) |->
  ((g x)((g x)(y))))
%Define
```

(c)

Figure 37.4: Part (a) shows a definition that uses a local polymorphic definition inside it. Identifier g is defined once but used twice, with different types. (The syntax is that of Cinnameg, but the semantics of Cinnameg actually does not allow this.) Part (b) shows the same definition in a more basic form, using function expressions. (x `|->` E is the function h defined by $h(x) = E$.) Part (c) shows the result of removing the definition of g from f, creating two separate definitions. Type inference can now be done on the definition of g first, and then on the definition of f, using the inferred type of g in the same way as predefined functions such as head are used. Each use of g in the definition of f is assigned a different copy of the polymorphic type found for g, making it possible for g to be used with more than one type.

to the context of f at all. Make a separate definition $g = e$ prior to the definition of f, and remove the definition of g from d. Perform type inference on the definition $g = e$ first, and then on the modified definition of f. If there are additional local definitions in the definition of f, then they will need to be removed in the same way.

3. If expression e contains one or more formal parameters, build a tuple L of all of the formal parameters that occur in e. For example, if e refers to formal parameters x and y, then L will be (x, y). Pull the definition of g out of d by making it into a function definition of the form $g = (L$ `|->` $e)$. Replace each occurrence of g in d by expression $g(L)$. Now g refers to a definition made prior to the definition of f, and so it can be polymorphic. Perform type inference on the definition of g first, then on the definition of f.

Note that the final compiled definition is the original one. Pulling definitions out has only been done as a way of performing type inference.

Most languages that support polymorphism with type variables allow local definitions to be polymorphic. Cinnameg does not; it assigns the same type to every occurrence of a locally defined identifier, rather than copying the types. That limits the amount of polymorphism that can be used locally, but significantly simplifies handling of local definitions. You can still create a polymorphic local definition in Cinnameg; just use **Polydef** instead of **Define**. The following is allowed in Cinnameg.

```
Define f(x) = (g(g(x))) |
  Polydef g(y) = (x,y).
%Define
```

Parameters and let-bound polymorphism

The reason that it makes sense for a definition to be polymorphic is that you can see, from the structure of the definition, exactly what it needs to do, and so you can determine the types of things that it will work on. For example, in definition

$$f(x) = \text{head}(x) :: y$$

you can clearly see that x must be a list from the structure of the definition. It is possible to infer the amount of polymorphism that the definition of f has.

Parameters are different. Their actual values are defined elsewhere, and the definition of a function cannot know how its actual parameters will be defined. As a result, a function cannot know to what extent its actual parameters might be polymorphic. To account for that, it must presume that parameters do not possess polymorphism. Each occurrence of a parameter in a function definition must have the same type variable attached to it, so that the parameter has the same type at each of its occurrences. To illustrate, suppose that you try to make the following definitions in Cinnameg.

```
Define g(x)   = (x,x).
Define f(r,y) = r(r(y)).
Define h(z)   = f(g,z).
```

Function f uses its parameter r (a function) in two places. Performing substitutions, it appears that $h(2) = f(g, 2) = g(g(2)) = g(2, 2) = ((2, 2), (2, 2))$. But type inference on the definition of f yields type $(\alpha \to \alpha, \alpha) \to \alpha$, indicating that the domain and codomain types of r must be the same. Function h passes f the polymorphic function g, of type $\alpha \to (\alpha, \alpha)$. Since the domain and codomain of g are not the same, the definition of h is poorly typed. The problem is that the polymorphism inherent in the definition of g cannot be passed through a parameter; only one specific type of g can be used. Notice that the following definition, where g is built into f, is well typed.

```
Define g(x) = (x,x).
Define f(y) = g(g(y)).
Define h(z) = f(z).
```

Polymorphism works well for definitions, but does not survive parameter passing. Since definitions are often made using a construct that begins with a word such as **let**, polymorphism using variables is said to be *let-bound*, or restricted to definitions made explicitly and excluding definitions of parameters that are made implicitly at function calls.

Recursion

Should a recursively defined polymorphic function be allowed to use itself with more than one type? There is no obvious reason why that should not be allowed. For example, the Cinnameg $ function converts a value to a string. For ordered pairs, it is defined so that $(1,true) = "(1,true)". A simple definition is as follows.

```
Let $(a,b) = "(" ++ $(a) ++ "," ++ $(b) ++ ")".
```

So $ for type (Integer, Boolean) → String needs to use $ of types Integer → String and Boolean → String. Our type inference algorithm does not allow that; it assigns the same type variable to each occurrence of a recursively defined function. It seems reasonable to ask for the type inference algorithm to be generalized so that it allows a function to call itself recursively with different types. The information about how the function works is present in its definition, so there seems no inherent limitation, like the one that affects parameters, that would prevent more general use of polymorphism.

Unfortunately, there is a subtle limitation that prevents us from handling polymorphism with recursion in the most general possible way. If a recursive function can use itself with more than one type, then the type inference problem that a compiler must solve becomes uncomputable,[2] making writing a correct compiler impossible. To make it possible to write a compiler, recursive function definitions must be restricted somehow. For example, type inference is solvable provided each recursive use of the function has the same type and no copying of the type is done.

Cinnameg deals with the issue of recursion by having two different kinds of definitions. Those that start with Define allow recursion, and assume that each recursive use has the same type. That is often what you want, but definition

```
Define $(a,b) = "(" ++ $(a) ++ "," ++ $(b) ++ ")".
```

is poorly typed. Definitions that start with Let also allow recursion, but if recursion is used, then the polymorphic type of every identifier that is used in the definition, including recursive calls, must have been given in advance, so that it does not need to be inferred. That given polymorphic type is copied for each occurrence of an identifier. For example,

[2] An uncomputable function cannot be solved on a computer. There does not exist any algorithm for it.

```
Type BinaryTree(<a>) =
    leaf (leaflabel ~: <a>)
  | tree (treelabel    ~: <a>,
          leftSubtree  ~: BinaryTree(<a>),
          rightSubtree ~: BinaryTree(<a>))
%Type
```

Figure 37.5: **A parameterized type of binary trees, written in Cinnameg.** BinaryTree(T) is a type of binary trees where each node is labeled by a value of type T.

```
Anticipate $: <a> -> String.
Let $(a,b) = "(" ++ $(a) ++ "," ++ $(b) ++ ")".
```

gives the polymorphic type of $ in advance. Notice that the given definition does not cover all of the types indicated by the Anticipate line. The definition is partially parametric, since it works for all product types, and partially ad-hoc, since other definitions need to be given for types that are not products.

37.6. Parameterized types

Figure 34.7 shows a definition of binary trees where each node is labeled by a string. But why should node labels be restricted to strings? What if you want, for example, a tree with an integer at each node, or with a pair holding an integer and a string? A more general tree would hold an arbitrary type of value at each node, whatever type you want to use.

Figure 37.5 shows a definition, in Cinnameg, of *parameterized* type BinaryTree, where BinaryTree(T) is a type of binary trees where each node has a label of type T. When you parameterize a type, the function definitions usually change only to the extent that any type information written in them needs to be consistent with the new, parameterized, type. Functions treeSize and treeFlatten of Section 34.5 do not contain any type information in their definitions, and they will work for the parameterized BinaryTree tree type without any change.

37.7. Generics

Ideally, if you have decided to support polymorphism with type variables, then polymorphism should be built into the foundation of a language and its implementation. But often, the objective is to add polymorphism to an existing language that does not currently support it, making as few changes to the language and its implementation as are necessary. Generic polymorphism is a way of supporting polymorphism that requires a minimum of additional machinery. It dispenses with the need for type inference and unification, and with the need for the run-time implementation to be aware of polymorphism.

A first attempt: named types

Take a language, such as C, that does not support polymorphism. Suppose that you want to create a parameterized type Pair(S, T) in C, supporting some basic operations on ordered pairs. You can start with the module shown in Figure 37.6, which creates type IntStringPair, a special case where S = int and T = string. Of course, that is not a polymorphic definition at all. When the need arises, you copy the entire definition and make some changes. For example, to create a pair type StringStringPair, holding two strings, just change the name of the type and change the definition of S. The required changes are simple and mechanical. It seems reasonable for a compiler to make them for you.

Generics in C++

C++ provides for polymorphism and parameterized types in its *template* facility. Figure 37.7 shows a definition of Pair in C++ in a more polymorphic way. To create a variable p that holds pair (2, “lucky”), you write the following.

```
Pair<int, string> p;
string str = "lucky";
p = buildPair(2, str);
```

A compiler can implement templates by recompiling the template definition for each new type that is used. For example, when it sees Pair<int, string> for the first time, it compiles the definition of Pair with S = int and T = string, choosing a new, internally generated, name for Pair. When it sees buildPair used, it compiles the definition of buildPair for the same types. It effectively converts a single parametric definition of buildPair into a collection of ad-hoc definitions, each of the same general form.

There are advantages and disadvantages of this approach to polymorhism. One advantage is that very little needs to be added to the language to provide for polymorphism. The C++ run-time system does not need to know about templates, although it normally knows the convention used to select names, so that it can show a type name as Pair<int, string> rather than the more cryptic name selected by the compiler. Type checking is performed when a definition is recompiled with specific types, so there is no real need for dealing with polymorphism in the type checker.

But there are disadvantages as well. One is that, in order to import a polymorphic definition, you need to import the source code for the definition, since it needs to be recompiled several times. If several modules use type Pair<int, string>, the definitions for that type can end up being recompiled for every one of those modules. If there are type errors, then they are reported at the places where the template is used, but occur within the template itself. So type errors can be difficult to understand. As an example, suppose that you make the following definition of the squaring function in C++.

```
template<typename T>
T sqr(T x)
{
  return x*x;
}
```

Then expression sqr(3) works, and produces 9, but expression sqr(”abc”) results in an error that is reported inside the template definition. To understand what it means, you need to read the template definition. Often, you use template definitions from libraries. Understanding those definitions and why they lead to errors can be difficult.

```
#define theName IntStringPair
typedef int S;    /* Left hand type */
typedef string T; /* Right hand type */
typedef struct theName
{
  S left;
  T right;
} theName;

S getLeft(theName p)
{
  return p.left;
}

T getRight(theName p)
{
  return p.right;
}

theName buildPair(S lft, T rgt)
{
  theName p;
  p.left = lft;
  p.right = rgt;
  return p;
}
```

Figure 37.6: **A definition of a pair type in C.** The type defined is IntStringPair, a pair containing an integer and a string. The #define line tells the compiler to replace every occurrence of 'theName' by 'IntStringPair'. The typedef lines create local types S, T and Pair, which are used in making the function definitions. The advantage of defining them is that, to create another type of ordered pairs StringStringPair, all that is necessary is to change the definition of S to string, and to change the definition of theName from IntStringPair to StringStringPair. You can create several modules, one for each ordered pair type that you need, with minor (and mechanical) editing needed for each.

```
template <typename S, typename T>
struct Pair
{
  S left;
  T right;
};

template <typename S, typename T>
S getLeft(Pair<S,T> p)
{
  return p.left;
}

template <typename S, typename T>
T getRight(Pair<S,T> p)
{
  return p.right;
}

template <typename S, typename T>
Pair<S,T> buildPair(S lft, T rgt)
{
  Pair<S,T> p;
  p.left = lft;
  p.right = rgt;
  return p;
}
```

Figure 37.7: **A definition of a pair type in C++.** This turns the definition of Figure 37.6 into a polymorphic definition that is handled automatically by the compiler. The template lines create the type variables S and T. Pair$<S,T>$ is an ordered pair type similar to (S,T).

Generics in Ada

The Ada programming language uses a similar kind of generic definition, but requires the programmer to request recompiling a definition explicitly. Ada definition

```
generic
  type T is private;
function Sqr(X: in T) return T;

function Sqr(X: in T) return T is
  return X*X;
end Sqr;
```

defines a generic function Sqr with a single parameter (marked as an **in** parameter, indicating an input to the function). But before you can use Sqr, you need to request recompilation with a particular type. The following creates two versions, one that squares an integer and the other that squares a real number.

```
function SqrInt is new Sqr(Integer);
function SqrReal is new Sqr(Float);
```

A well-designed Ada program can avoid repeated compilation of a generic definition for the same type by requesting the desired versions in only one place.

37.8. Summary of terminology and concepts

Type variables give a compiler more information about the type of a polymorphic function, and so allow for stronger static type checking than the approach of the previous chapter without type variables. They are also convenient for describing parameterized types, such as Tree(α). Type checking with type variables involves creating and solving a system of equations, using unification to convert an equation into an equivalent collections of substitutions.

Polymorphism can either be built into the foundation of a language implementation, or it can be added on by using generic polymorphism, converting parametric definitions to ad-hoc definitions, and creating several copies of each definition.

Polymorphism with variables is let-bound. That is, only explicit definitions can be polymorphic. Implicit definitions, such as those that are made as a result of parameter passing, cannot be polymorphic.

Terminology

Generic polymorphism A generic definition is made in a parametric style, but is compiled in an ad-hoc style, with a separate compiled version for each type.

Let-bound polymorphism Polymorphism with type variables works for definitions (lets and defines), but not for parameter passing. It is said to be let-bound.

Parameterized type A parameterized type takes a type or types and builds another type. For example, the list constructor [ ] takes type T and builds type $[T]$. A parameterized queue type Queue(T) allows you to select a type T and to create a queue whose members are of type T.

Substitution A substitution is an equation with just a variable on the left-hand side.

Type inference A compiler performs type inference when it determines type information about a program by examining the structure of the program.

Type variable A type variable is a variable whose value is a type. Type variables are used to express polymorphic types.

Unification Unification is a method of solving equations involving type variables. It converts an equation into an equivalent set of substitutions. Performing the indicated substitutions reduces the number of equations.

37.9. Exercises

37.1. Using type variables, give the most general polymorphic type of each of the following functions.

(a) $f(x) = (x, x)$

(b) $g(x) = (\text{tail}(x), \text{head}(x))$

(c) $h(x) = \text{head}(\text{tail}(x))$

(d) $i(x, y) = (y, x, x)$

(e) $j\ x\ y = \text{head}(x\ y)$

(f) $k\ x\ y = x\ y\ y$

37.2. Perform a type analysis, using type variables, of the definition of cat in Equations (18.8.6).

37.3. Perform a type analysis, using type variables, of the definition of take in Equations (18.8.5). Follow the Cinnameg assumption that, when $n + 1$ is used in a pattern, it has type Integer.

37.4. Perform a type analysis, using type variables, of the definition of member in Equations 18.8.4. When a variable occurs twice in a pattern, as m does in the second equation, an equality test needs to be done, implicitly using operator ==, and the polymorphic type of == is (⟨a: Eq⟩, ⟨a: Eq⟩) → Boolean. (Genus Eq contains types that support equality tests.) Be sure to take the need for equality tests into account.

37.5. Polymorphism can achieve in a typed language much of what is allowed in a typeless language, but it cannot achieve everything. Argue that the following function f makes sense if you ignore types, but that the definition of f is not well-typed. What is $f(3, 2)$?

$$\begin{aligned} f(0, x) &= x \\ f(n+1, x) &= (f(n, x), f(n, x)) \end{aligned}$$

37.6. Prove that the unification algorithm described in Figure 37.1 produces a set of substitutions where no variable that occurs on the left-hand side of any of the substitutions also occurs on the right-hand side of any of them.

37.7. Prove that the set of substitutions produced by unify(A,B) are, taken together, equivalent to equation $A = B$.

37.8. Modify the definition of queues from Exercise 34.9 so that Queue is parameterized, and you can create a queue of any desired type of thing.

37.9. How can the algorithm for doing type inference with polymorphic local definitions be extended to handle mutually recursive definitions?

37.10. Is Cinnameg definition

```
Define f(p) = (p(1), p("abc")).
```

well-typed? Why or why not? What if f is used as follows?

```
Define id(x) = x.
Define z = f(id).
```

37.10. Bibliographic notes

Type inference with type variables was discovered by Hindley [51] and Milner [77], and Cardelli [23] explains type inference. Cardelli and Wegner [22] survey polymorphic types.

Part XIII

Object-Oriented Programming

Chapter 38

Objects and Encapsulation

38.1. The object-based view of abstract data types

An abstract data type is characterized (partly) by the operations that form its signature, so those operations are thought of as part of the type. For example, if a Queue type supports operations Insert, Delete, front and isEmpty, then you would say that the type has those four operations.

But there is an equally valid *object-based* point of view, where the operations are associated not with the data type but with the data items, or *instances*, themselves. In the object-based view, each individual queue instance, called an *object*, supports the queue operations, and is thought of as carrying its operations with it. The signature is associated with the object, allowing each individual object to be its own implementation of a set of operations, choosing its own way of representing information and performing those operations. It is possible, for example, to have one object that implements Insert, Delete, front and isEmpty as a queue would, with insertion at the back, and another that implements them the way a stack would, where new insertions are at the front. From the outside, they look the same, because they support the same operations, and the two objects can be used in similar ways. That provides a programmer with a high degree of flexibility and makes it possible to make a subprogram have a variety of different behaviors by varying the characteristics of the objects that the subprogram uses.

Because object-based programming associates operations with objects, not with types, it is able to achieve some of the same goals as abstract data types, but to dispense with the notion of a type, if desired, allowing the programmer to concentrate on the objects themselves. That is not to say that object-based programming is incompatible with types; rather, types are optional.

Objects and object-based programming

In object-based terminology, the functions and procedures that an object holds are called the *methods* of the object.[1] But an object is more than a collection of methods; it also needs to remember information. For example, a queue object needs to know what is in

[1] The term *method* is probably not the best choice, since the intent of a function is to hide, not to expose, the method that is employed by and encapsulated within the function. But the terminology has become standard, so we use it.

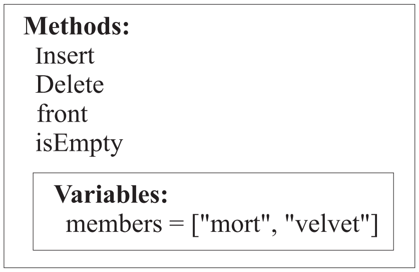


Figure 38.1: **A snapshot of a queue object.** The methods (Insert, Delete, front and isEmpty) are visible to other objects, but variable ’members’ is private, so is only visible to the methods inside this object.

the queue at the moment. Object-based programming is usually an imperative form of programming, based on changing variables. So, in addition to methods, an object typically contains variables whose values can be inspected and changed by the methods inside the object. The current set of values of the variables in an object, taken as a whole, is called the *state* of the object, telling exactly what information an object is storing at a given point in time. Figure 38.1 shows a view of a queue object holding values ”mort” and ”velvet” in its state, and also holding the queue methods.

Various characteristics of a type can be brought down to the level of objects. For example, an abstract data type has a carefully chosen set of operations, its *signature*, that it supports. By the same token, an object interacts with its surroundings in carefully controlled ways, through operations that are known as the *public* methods of the object. The state of an object is hidden from view, visible only to the object itself, in just the way that the representation of a fully abstract data type is hidden from view behind the name of the type. That is illustrated in Figure 38.1 by showing the state encapsulated inside the object. The state is *private* to the object, and can only be seen from outside the object indirectly through the public methods.

38.2. Agents, components and program organization

Focusing on objects suggests a different way of thinking about programs. In a procedural program, you think of performing a procedure on a variable or variables. But in object-based programming, you think of asking an object to perform an operation. The object is thought of as an *agent* capable of performing requested actions.

An object resembles a miniature computer. It has memory and some software that uses that memory. An object-based program can contain thousands of tiny computers communicating with one another, each one specialized at performing a particular task. An object-based programmer thinks, in some ways, like a computer network designer, placing different services in different machines, storing information on the computer that needs it, and arranging for machines to communicate with one another.

The difference in focus, from a procedure operating on an object to an object performing a procedure, is a little like walking around a statue and looking at it from another side. It might sound slight, but it can lead to a profound difference in the way programs are written. Procedural software design typically begins with a decision on which procedures need to be implemented, with the data flow among procedures following later. Since you start by concentrating on the procedures, you tend to group similar procedures together, and that leads to programs that are organized around procedures and algorithms. For example, all procedures that print something might be put together into a single module, while procedures for mathematical operations would be put into a different module.

In object-based design, the designer starts by thinking about the objects (the miniature computers) that the program will support, and what information they will remember and manipulate, with the methods following later. The focus is on the information that each object will be responsible for maintaining. When the methods are eventually written, it is natural to put all of the methods that belong to a particular object or group of related objects in one module. That leads to programs that are organized around data and types of data. For example, all operations that affect queues are put in one place, regardless of what those operations do. An operation for printing the content of a queue would naturally go with other operations that belong to a queue object, not with other operations that print things.

38.3. Building objects

The most obvious way to build an object is in a custom way, choosing each variable and method to put into the object, giving each a name, and determining which are public and which are private. But not all objects need to be built that way.

Copying

One way to create a new object is to make a copy of an object that you already have. If an object did not contain variables that could be changed, there would be little point in making a copy. But because an object can change its variables, two different copies of an object can go in different directions, modifying their states in different ways. To copy an object X, you make a copy of each variable contained in X. The copy keeps the same methods that were in X.

Extension and inheritance

If you have an automobile, sometimes you want to add options. For example, if you move from a cool climate to a warmer climate, you might discover that you need an air conditioner. You take the car to a dealer and have one installed. Objects in a program can similarly be extended by adding new variables and methods. The larger object is said to inherit features from the object that it extends.

Overriding methods

Sometimes, when you extend an object, you want to replace some methods rather than inheriting them. For example, suppose that a new queue object is to be created from an old one, but the new queue operation is intended not only to perform operations, but to keep track of the total number of Insert operations that have been done on it since its creation. The new object will need a new variable to hold the insertion count, and it will need to replace the Insert operation by one that not only does the insertion but also increments the counter. When a method is replaced, the replacement is said to *override* the original method.

Sometimes an overriding method wants to make use of the method that it is overriding. For example, a method that both inserts and adds one to a counter probably wants to make use of the Insert method that would have been inherited. The new Insert function can be built by getting (not running) the Insert method from one object, and then building a new function around that method.

38.4. Polymorphism in object-based programming

A function that uses an object will ask that object to perform certain operations, and it only requires that the object supports all of the operations that are actually used. Suppose, for example, that the object is asked to perform operations "Insert", "front" and "Delete", and nothing more. Then the function will work on any object that supports *at least* those operations. It will work on a queue object, or on an augmented queue object (with additional variables and methods), or even possibly an object that works very differently from a queue object, such as a stack or set object, as long as it has the required methods. Functions and procedures that use objects are, therefore, automatically polymorphic.

38.5. Summary of terminology and concepts

Object-based programming focuses the programmer's attention on the objects within a program. Rather than ascribing functions and procedures to types and thinking of them as operating on data, the object-based view ascribes the functions and procedures to the objects, and encourages the view that objects perform their own operations on themselves. Each object is like a miniature computer with a small amount of memory and software.

That is analogous to the way we work in everyday life. We interact with other people by asking them to perform tasks. We interact with machines in similar ways. Think of interacting with a vending machine. You think of the machinery as belonging to the machine itself. Putting coins into a vending machine and then pushing a button to make a selection is better viewed as making requests of the machine than as performing an operation on the machine.

With object-based programming it is possible to achieve some of the goals of abstract data types, but without the need for types. The fact that a vending machine is one of many similar machines is probably irrelevant to a single interaction, and you might prefer to think of the machine as an individual object rather than thinking in terms of its type. That does not preclude using concepts of types, and of static type checking, with object-based programming, but it makes types optional.

Terminology

Extension An object A that is built by adding components to a copy of another object B is called an extension of B.

Inheritance An object can be built by inheriting methods, variables and state from another object.

Method A function or procedure that an object can perform is called a method of that object.

Object An object holds a state (in a collection of variables) and a collection of methods that alter or look at that state. It is thought of as an agent that can be asked to perform certain methods.

Object-based programming Object-based programming is a paradigm of programming that focuses on the objects in a program, as opposed to focusing on the procedures. Operations are ascribed to the objects, and objects operate on themselves.

Override When an object is made by copying another object, you have the option of replacing, or overriding, some of the methods.

Private A private component of an object is one that is accessible only to that object's methods, and cannot be directly seen by any other object or function.

Public A public component of an object is one that is directly accessible to other objects and procedures. It is through the public methods that objects interact with one another.

Signature The signature of an object is the collection of public methods, variables and constants that the object supports, along with the names by which the object refers to them.

State The state of an object is the data that the object is currently remembering in its variables. The state can usually change over time.

38.6. Exercises

38.1. How is an object like an abstract data type?

38.2. What is inheritance in object-based programming?

38.3. What does it mean to override a method?

38.4. Explain why a function that takes an object as a parameter is polymorphic. What kinds of objects can it use?

38.5. For this problem, ignore the issue that some components are public and some are private. Suppose you decide to use the idea of records for types of objects. An object with components a: Integer and b: String → Integer has record type $\{a\text{: Integer}, b\text{: String} \rightarrow \text{Integer}\}$. You would like to keep the polymorphism that object-based programming enjoys. Define a relation $\leq$ on record types so that, if f takes a parameter of type B and $A \leq B$, then you should be allowed to apply f to a parameter of type A. (If you think of an object of type B as representing a mammal and an object of type A as representing an anteater, then you should be able to pass an anteater to a function that wants to receive a mammal.)

38.7. Bibliographic notes

Abadi and Cardelli [3] describe a theory of object-based programming, and Wegner [108] discusses object-based languages. Kay [63] discusses the idea of an object-based program as a collection of small, interacting computers in the context of the development of Smalltalk.

Chapter 39

Classes and Object-Oriented Programming

39.1. Introduction

Difficulties with object-based programming

Chapter 38 introduces the concept of object-based programming, where operations are brought down from the level of types to the level of the individual objects that belong to the types. Unfortunately, direct implementation of object-based programming has some drawbacks. One concerns efficiency. If objects carry their methods with them, then an object that holds many methods must occupy quite a bit of memory. When an object is copied, that memory is copied as well.

Another problem concerns just how methods will be written and attached to the objects. Although functional languages support functions as data items, most other kinds of languages typically do not do that in any but the most limited of ways. But if a function is not a value, what is being stored inside the object as a method?

Object-oriented programming

This chapter explores dealing with the difficulties of object-based programming by partially reversing what was done in the preceding chapter. Instead of associating functions with objects, we move them back to the level of the types (called classes), yielding an approach called *object-oriented* programming. That appears to backtrack and to lose the advantages that object-based programming affords. But that is not really the case. It is possible to *imagine* the operations as being kept with the objects when, in fact, they are kept with the types. Object-oriented programming offers the convenient fiction that it is object-based programming while keeping the advantages that the traditional view of abstract data types offers.

```
class Queue
  instance variables:
    members: [String]
  instance methods:
    Insert(s)
      members := members ++ [s]
    Delete()
      members := tail(members)
    front()   = head(members)
    isEmpty() = members == []
end class
```

Figure 39.1: **A queue class.** The syntax is generic, not that of any particular language.

39.2. Classes

In object-oriented programming, the notion of a type is replaced by the notion of a *class*. A class describes a type of object by describing characteristics that are shared by all objects that are instances of the class.

A class actually has several different aspects to it. It plays the role of a type by indicating how information is represented and how methods are implemented. It plays the role of a genus to manage polymorphism by describing signatures that are shared by several classes. Classes are responsible for building the objects that they describe, so they play the role of object factories. A class also takes on the responsibilities of a module, controlling visibility of components and scopes of names. This chapter concentrates on the role of classes as types and as object factories; later chapters look at other aspects of classes.

Classes as types: describing variables and methods

A class can be thought of as a template or plan for constructing an object along with some ways of building objects that follow the plan. Each object constructed by class C is called an *instance* of class C, and has a tag attached to it indicating that it belongs to class C.

A class definition consists of a collection of variable and method definitions describing characteristics of the instances. For example, class Bucket might indicate that each of its instances has a variable called *volume* of type *Liter*, and a method called *Grow*. The class definition gives the details of how Grow works. Since the variables and methods described by a class are actually characteristics of an instance of the class (using the object-based viewpoint), they are called *instance variables* and *instance methods*. All instances of a class share the same method implementations, but each object has its own copies of the instance variables.

Figure 39.1 illustrates, defining a class that describes the instance variables and instance methods of a queue. The syntax is not that of any particular language, although we use some syntax from Cinnameg for constants and expressions. For simplicity, the members of a queue are strings. A queue object holds a list of strings as its state and has methods for inserting a string (at the back), deleting a string (from the front), getting the front string, and checking whether the queue is empty.

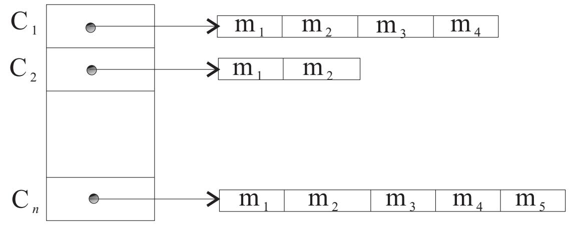


Figure 39.2: An implementation of a dispatch table, with a row for each class and a column for each method.

39.3. Getting components of an object: the dispatcher

The state of an object, represented by the values of its variables, is stored with the object. The instance methods, however, are stored elsewhere. When object A needs to run method m, the runtime support first gets the class of A from A's tag. Typically, each class is given a number, and each method within a class also has a number. A table, called the *dispatch table*, stores an array of pointers to methods for each class, as shown in Figure 39.2. Suppose class Queue is assigned number 4 and the Delete method is the second method in the class. Then a queue object finds its Delete method by looking in row 4 of the dispatch table, and getting in the second entry in the array that is found there. Two array fetches (one for the class and one for the method) suffice to get a method from the dispatch table, so looking up a method is cheap.

39.4. Constructing objects

In addition to describing the components that objects have, a class can take on the role of an *object factory*. When an object is created it is important to put values into its instance variables, since otherwise the object is in an undefined state. The initial state of an object is set up by a *constructor*. Ways of expressing constructors differ from one language to another. Figure 39.3 shows a generic approach to constructors.

When you order a car from a factory, you should be able to request options such as paint color and air conditioning. Objects can also be created with different options, and it is common for a class to have more than one constructor, allowing you to order objects with various initial configurations. To illustrate, Figure 39.3 shows the queue class with two constructors, so that

```
q := newQueue();
```

makes q a new instance of class Queue that is initially empty, and

```
q1 := newQueue(["cat", "dog"]);
```

makes $q1$ a new queue initially holding two strings.

```
class Queue
  instance variables:
    members: [String]
  constructors:
    newQueue()
      members := []
    newQueue(init: [String])
      members := init
  instance methods:
    as in Figure 39.1
end class
```

Figure 39.3: **The Queue class with two constructors.** The name newQueue is overloaded.

39.5. Object-oriented languages

Object-oriented programming languages provide some means of describing classes and creating objects that are instances of those classes. They also must provide a means of asking an object to perform a method. Java uses syntax *agent.action*. Cinnameg uses *agent*'s *action*; the action starts with a method, and you think of the method as belonging to the object. Smalltalk dispenses with extra syntax; you just write *agent action*. But those differences are minor compared to more substantive language differences. Object-oriented ideas have been incorporated into every kind of language, and some languages that were not even originally designed to be object-oriented have acquired object-oriented dialects.

Some object-oriented languages, such as Java, treat a class as a type, and their compilers perform some (static) type checking. But it is also possible to provide classes in a typeless language. Just because a program is based on classes does not mean that a compiler needs to remember what type of thing is in a given variable, or to perform any type checking. Remember that each object already carries with it a tag indicating its class, so it is a fairly simple matter to perform run-time type checking. Smalltalk is an example of a typeless object-oriented language. Classes in Smalltalk are used for describing characteristics of objects and for building objects, but they are not used for any kind of compile-time type checking. Although Smalltalk is object oriented, it has the feel of an object-based language, where you can use any object in any context, as long as the object has all of the methods that are used in that context.

39.6. Summary of terminology and concepts

Object-oriented programming is a practical approach to the ideas of object-based programming that relies on creating types of objects, called classes. A class can be viewed as a template for creating an object; it tells the variables and methods that each object will have, how the methods work and how an object is constructed. The constructors allow you to think of a class as an object factory..

Terminology

Class A class has several characteristics. A class is, among other things, a type of objects.

Constructor A constructor builds a new object, typically by initializing the object's variables. It makes the object ready for use.

Dispatch table The dispatch table is a data structure that associates a method implementation with a pair consisting of a class and a method name or number. In practice, the class and method are given numbers for efficient implementation of the table. For example, a lookup in the dispatch table might request method 5 of class 14.

Dispatcher The dispatcher is part of the run-time support that finds a method for an object. It examines the tag of the object and the name or number of the method and finds the method implementation in the dispatch table.

Instance An instance of a class is an object that belongs to that class.

Instance method An instance method in a class definition is a method (or subprogram) that belongs to an instance of the class. Each instance *logically* gets its own copy of each instance method. In reality, the instance methods are stored in the dispatch table, not with the objects, for efficiency.

Instance variable An instance variable is a variable that belongs to an instance of a class. Each instance gets a separate copy of each instance variable described in the class.

Object-oriented programming Object-oriented programming is similar to object-based programming, but it uses classes to describe types of objects, rather than building each object on a custom basis.

39.7. Exercises

39.1. What is an advantage of storing instance methods in the dispatch table rather than with the objects?

39.2. Why is a notion such as a class necessary for using the dispatch table?

39.3. Since instance methods are not stored inside objects in object-oriented programming, how do you think an instance method gets access to the object (self) to which it supposedly belongs? Describe a way in which an implementation might do that.

39.4. What is the purpose of a constructor?

39.5. Would it work to store the dispatch table as an array of linked lists? What would be the advantage or disadvantage of doing that?

39.6. Why is it important for an implementation to use integer tags instead of using the class name (a string) as a tag? What would be the consequences of tagging items by a string? Keep in mind that, every time any instance method wants to use another instance method, it needs to do a lookup in the dispatch table.

39.7. Using any reasonable syntax, write an implementation of a class whose instances represent DVDs at a rental store. Each DVD has to remember its state (rented or on the shelf) and, if rented, it has to remember who rented it, and when it is due back. How can you represent the information? What instance methods should each object have?

39.8. Parameters to methods can be simulated, to some extent, by instance variables. For example, if you have several methods that take a string as a parameter, then you can remove that parameter, add an instance variable (theString), and add a method to set theString to a given string. To use a method, you first set theString to your desired string, and then call the method, which just looks in theString to find out the value of the desired parameter. Would you recommend that approach? Why or why not?

39.8. Bibliographic notes

Castagna [26] explains object-oriented programming. Cook [33] compares object-oriented programming with abstract data types.

Chapter 40

Inheritance and Polymorphism

40.1. Polymorphism

A class describes a *signature* consisting of the variables and methods that all of its instances possess. For example, suppose the Queue class has the signature shown in Figure 39.1. When a new queue object is created, it has *exactly* those characteristics, no more and no less.

But imagine *using* a queue object, by employing the public features that it possesses. From the point of view of a part of a program that interacts with a queue object, the object must have *no less* than the features mentioned in the signature, since those are the features that are used. It does not really matter if it has additional features. That suggests that methods that use queues can be polymorphic, working on any type of queue that possesses at least the signature of queues.

Our viewpoint so far has been that a class is an abstract data type, providing, through its method definitions, a particular implementation of that abstract data type. But, as discussed in Chapter 36, introducing polymorphism requires the addition of another concept, a genus that stands for an abstract data type with multiple implementations. The genus and the type are two different aspects of a class.

Creating a class does more than just create a genus and a type. It makes the operations from the type polymorphic over the genus.

40.2. Inheritance

When you create a new type of queue, you do so by extension, inheriting methods from Queue by taking advantage of their polymorphism. In effect, you reuse those methods, avoiding the need to write them again for the new class. Figure 40.1 shows a definition of class LQueue using inheritance, where an object of class LQueue has not only all of the Queue operations but also has one that yields the current length of the queue. To implement the inheritance, the row of the dispatch table for Queue is copied into the row for LQueue, so that instances of class LQueue end up using the Queue methods to perform Insert, Delete, isEmpty and front.

The *class hierarchy* indicates how classes relate to one another in the same way that the genus hierarchy indicates how genera are related to other genera. Class *A* is a *base class*

```
class LQueue extends Queue
  instance methods:
    length() = length(members)
  constructors:
    newLQueue()
      super()
    newLQueue(init: [String])
      super(init)
end class
```

Figure 40.1: **Class LQueue, a subclass of Queue.** The syntax is generic. In this chapter we are not concerned with issues of privacy, and allow class LQueue to use what would normally be a private component, members, of class Queue. How to deal with that is discussed in Chapter 41.

Each constructor needs to run a constructor in the base class (Queue), indicated in our generic syntax by a pseudo-constructor called super.

of B if A is immediately above B in the hierarchy. For example, the base class of LQueue is Queue. Class B is a *subclass* of class A if B is beneath A in the hierarchy, possibly through a chain of classes. A subclass is sometimes called a *derived class*. Class A is a *superclass* of B if B is a subclass of A.

40.3. Constructors in subclasses

Class LQueue needs a constructor that initializes the inherited variable, members. But class Queue already has such a constructor, and it is sensible to use Queue's constructor to do the initialization. (When we introduce issues of privacy in Chapter 41, we will see that using an inherited constructor is sometimes unavoidable.) The problem is that we want to run the Queue constructor on an object of class LQueue, even though the Queue constructor normally creates an object of class Queue. Object-oriented languages have various ways of dealing with that. In our generic syntax, we write super(p) to call the constructor in the base class with parameter p. Figure 40.1 shows class LQueue with two constructors.

40.4. Inheritance by position

Inheritance is enabled by the polymorphism of a class's method definitions. For example, class LQueue gets its front method implicitly because that method is polymorphic over all subclasses of Queue. But there are some issues in inheritance and polymorphism that still need to be handled.

Getting instance variables

An object is typically represented as a block of memory holding the object's tag and its instance variables. In typical implementations, each variable is found in a particular place in the representation. For example, the tag might come first, followed by the first variable,

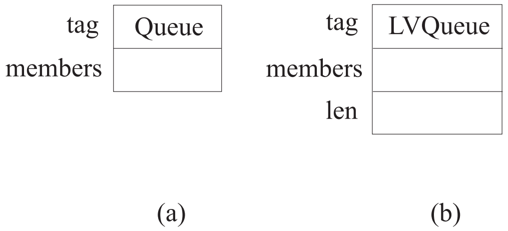


Figure 40.2: **The representations of two objects, (a) an object of class Queue and (b) an object of class LVQueue, with an additional variable.** The members variable is the second variable, counting the tag, in both cases.

then the second variable, etc. To get a variable from an object you need some way of finding that variable. Imagine running a function, called a *selector function*, that finds the variable. For example, the members variable is obtained from a Queue object by running a selector that returns the memory address where that variable is stored. You might not see the selector explicitly written in your program, but the language implementation must use one implicitly.

A class inherits variables from its base class. For example, an object of class LQueue has a variable called members, inherited from class Queue. The variables themselves are not inherited, since each object has its own variables, but both classes Queue and LQueue must possess a selector function that is used to extract that variable from the object, and it is really the selector function that is inherited.

As we have seen, polymorphism enables inheritance, so the selector function must, somehow, be polymorphic. A simple way to do that is to store each variable in the same place in every object, regardless of the object's class. For example, if members is always the first variable, then it is easy to create a polymorphic selector for it. We call this idea *inheritance by position.*

Adding new variables

Our definition of class LQueue does not introduce any new variables, but, for illustration, let's add one, creating class LVQueue. A reasonable variable is one called len that remembers the length of the queue, so that the length does not need to be computed each time it is needed. If an instance of class LVQueue has more variables than an instance of class Queue, then you cannot use exactly the same representation for both. But if variable len is put after members, as shown in Figure 40.2, then members is in the same position for both classes, and so the members selector is polymorphic, and can be inherited.

Positional inheritance of methods

There is another kind of selector at work in the implementation of an object-oriented language. When a method is looked up in the dispatch table, it is obtained by performing a selection from that table. So methods are really selected, just as variables are, and their selectors should be polymorphic.

The dispatch table assigns a number to each method in a class. The idea that we have used for variables, to put each variable in a fixed position within the memory that represents an object, can also be used for methods, by putting each method in a fixed position in a row of the dispatch table. The Insert method, for example, should be given the same number in class LQueue as it is given in class Queue. To achieve inheritance, it suffices to copy the entire row of class Queue into the row for class LQueue, and to put new methods of class LQueue at the end of the row.

40.5. Single and multiple inheritance

So far we have only looked at inheritance where a class has just one base class. A language that forces every class to have at most one base class is called a *single-inheritance* language. But there are cases where *multiple inheritance* is called for, in which a class can have more than one base class. Chapter 36 gives an example where a SpiderMonkey is both a Monkey and a NewWorld animal. Since a Spider Monkey logically belongs to both groups, it should be able to inherit features from both, and so has two base classes.

Multiple inheritance is common in graphical systems. An object often needs to know information about what it represents but also needs to be able to draw an image of itself and to interact with the user. For example, a class whose instances represent bears might need to be a subclass of Mammal and also a subclass of Graphics, so that a bear object can be used in contexts where it is asked to draw itself. If *b* is a bear object, you would like both *b*.eat() and *b*.draw() to be meaningful.

Multiple inheritance presents some serious problems to object-oriented languages. One problem is how variable and method selectors are managed. In a single-inheritance language, variables and methods in a subclass are added to the end of the variables and methods of its base class and selectors are automatically polymorphic. With multiple inheritance, that is not possible. The variables for the base classes will need to be combined, and it will not be possible to keep them in the same positions.

Another problem concerns determining what gets inherited. Suppose that classes Mammal and Graphics each define a method called grow. If an object of class Bear is asked to perform its grow method, which one should it use? Some sort of priority indications will need to be given in the program. So multiple inheritance has implications for the language as well as for the implementation.

Some object-oriented languages, such as Eiffel and C++, support multiple inheritance in order to allow the programmer freedom to relate classes in whatever way corresponds most closely with reality. But those languages are necessarily more complicated than languages that restrict the programmer to single inheritance, such as Smalltalk and Java,

40.6. Overriding methods

After adding the len variable to class LVQueue, you need to ensure that len is incremented each time a value is added to the queue and it is decremented each time a value is removed from the queue. Unfortunately, the Insert and Delete methods inherited from class Queue do not do that, and must be replaced in the subclass. That is, their definitions must be *overridden* rather than inherited. Figure 40.3 shows class LVQueue with Insert and Delete overridden. But there is an unpleasant aspect of that definition. Notice that the bodies of Insert and Delete from class Queue have been copied and modified. But why should they be copied? It would be better to use the definitions from class Queue, but to perform some

```
class LVQueue extends Queue
  instance variables:
    len: Integer
  constructors:
    newLVQueue()
      super()
      len := 0
    newLVQueue(init: [String])
      super(init)
      len := length(init)
  instance methods:
    length() = len
    Insert(s)
      members := members ++ [s]
      len := len + 1
    Delete()
      members := tail(members)
      len := len - 1.
end class
```

Figure 40.3: **Definition of class LVQueue with a new variable to hold the length of the queue.** The Insert and Delete methods are overridden, but the front and isEmpty methods are inherited.

```
class LVQueue extends Queue
  ...
    Insert(s)
      prior Insert(s)
      len := len + 1
    Delete()
      prior Delete()
      len := len - 1
end class
```

Figure 40.4: **An implementation of class LVQueue using prior methods.** Only the definitions of Insert and Delete are shown.

extra steps as well. That is more important than just a matter of additional work in writing the methods. If later you modify the method definitions in the Queue class, you do not want to be required to modify the definition of the LVQueue class as well.

So an object of class LVQueue needs to be able to get the Insert method out of the dispatch table in the row for class Queue. Object-oriented languages generally provide a way to do that. In our generic syntax, we will write *prior Insert* to indicate the version of Insert that would have been inherited. A better definition of class LVQueue is shown in Figure 40.4.

Overriding methods is a versatile tool, but it must be used with caution. Generally, a method should keep its intent even when it is overridden. Otherwise it will not work as expected where it is used. Remember that a polymorphic method might think that it is working with a Queue object when what it really has is an LVQueue object. Fortunately, the overriding definition of Insert for class LVQueue still inserts.

40.7. Inheritance and libraries

Imagine that you are writing a graphically based program, and, as part of this program, you create a window that holds some text, editable by the user, that allows the user to type, backspace, select text with the mouse, copy text to the clipboard, change the font of text, etc. Typically, this kind of object needs to respond to a large number of events such as a key being pressed or released, the mouse moving over the window, and so on.

Obviously, that is just the sort of thing that you would like to have provided for you in a library, where all you have to do is create a standard kind of object and let the library deal with all of the possible events. But whoever writes the library cannot think of everything that you might want to have. For example, suppose that you want a control-I keystroke to change to an italic font, but the library class does not work that way. All you need to do is to create a subclass of the library class where your subclass handles the event of a key being pressed. If it sees control-I, it changes the font, and otherwise it runs the key-press event method that it would have inherited.

Libraries made up of classes tend to be versatile and, due to inheritance, modifiable, since programmers can create subclasses that inherit most of their behaviors from the library, and only explicitly say what they want to do differently.

40.8. Pragmatics of inheritance

Imagine that, in order to keep track of how much money a particular individual has on hand, you have created a class called Wallet. A wallet can receive money and pay out money, and remembers a variable, Cash, telling how much cash it contains.

Now you would like to create another class, Person, representing characteristics of a person who is modeled within your program. One characteristic is how much money this person has on hand. If you think in terms of the variables, you realize that this person needs to have a Cash variable. But class Wallet already has that variable, plus methods for managing it. If you make Person a subclass of Wallet, then every instance of class Person will automatically inherit the Cash variable. A Person instance also inherits methods to accept money and pay out money, which is reasonable, since those methods will be handled by the Wallet class. Handing some money to a person becomes equivalent to handing that same money to his or her wallet. This certainly sounds like an appealing way to do things.

Unfortunately, making Person a subclass of Wallet suggests that a person is a special kind of wallet, which is absurd. It allows you to put an instance of class Person into a variable of class Wallet. But doing so is, very likely, a mistake in the program, not something that a programmer would deliberately do.

The problem is that, instead of saying that a person *is* a wallet, you want to say that a person *has* a wallet. Well-principled use of inheritance suggests that you should only use inheritance for the case where something of class B *is* a special kind of thing of class A. Instead of making Person a subclass of Wallet, you add an instance variable of class Wallet to class Person. Of course, that means that methods that might have been inherited from Wallet need to be written for Person, but at least you have a more principled and sensible design.

40.9. Summary of terminology and concepts

Inheritance makes it possible to avoid duplication of method definitions. A method can be placed in its most natural class, and used by all of that class's subclasses. Inheritance facilitates *reuse* of definitions.

A class can also inherit instance variables. But that is a very different thing from inheritance of methods, both in terms of what you get and how it is implemented. When a variable is inherited, each instance of a class gets its own variable; the variable is not shared. What is inherited is the characteristic of possessing a variable by a given name and the selector that finds that variable.

Methods are inherited by copying (pointers to) them from one row of the dispatch table to another row. The copying makes sense because the method definitions are polymorphic, and so can be shared by more than one class.

The mechanism for inheriting variables and methods in single-inheritance languages is simple; a given variable has the same physical location in the representation of every object, regardless of the class of that object, and a given method is put in the same position in the dispatch table. Multiple-inheritance languages need to implement inheritance in a more complex way. Methods need to be inherited by some kind of priority scheme in cases where there are conflicting definitions that could be inherited.

Class libraries implement classes that are usually intended to have subclasses. Customized objects can be created as instances of a subclass of a library class. Overriding methods allows the behavior of objects in subclasses to be tailored to the programmer's desires.

Terminology

Base class A base class of class C is a class from which C was derived. It is the class immediately above class C in the class hierarchy.

Class hierarchy The class hierarchy is similar to a genus hierarchy. It is a partial order on classes. In fact, you can view a genus as being one aspect of a class, and the class hierarchy is just the hierarchy of those genera.

Inheritance Inheritance is a mechanism that allows one class to use operations that are defined in another class. It is closely related to polymorphism, but is not the same. Polymorphism is an export issue; it controls which classes are allowed to run a particular method, and is connected to the signature of a class. Inheritance is an import issue; it is concerned with how a class chooses to define (or inherit) its methods, possibly taking methods that are available to it as a consequence of polymorphism.

Inheritance by position In a single-inheritance language, variable selectors are inherited by the fact that each variable is always in the same position in the representation of an object. Methods are found by looking for them at a fixed position in a row of the dispatch table.

Multiple inheritance A multiple inheritance language allows a class to have more than one base classes.

Single inheritance A single-inheritance language only allows each class to have one base class.

Subclass Class B is a subclass of class A if B is beneath class A in the class hierarchy. That is, B is derived from A or B is derived from a subclass of A.

Superclass Class A is a superclass of class B if B is a subclass of A.

40.10. Exercises

40.1. What is the difference between polymorphism and inheritance in object-oriented programming?

40.2. Suppose that you are writing in a statically typed object-oriented programming language. Suppose that B is a subclass of A.

(a) Should you be allowed to store an object of class B into a variable of nominal type A? Why or why not?

(b) Should you be allowed to store an object of class A into a variable of nominal type B? Why or why not?

40.3. What is one motivation for restricting a language to single inheritance?

40.4. Why is inheritance important to someone who only wants to use a class library, and does not intend to create his or her own class library?

40.5. Constructors are not inherited by subclasses. Why not?

40.6. One style of implementing a dispatch table for a single-inheritance language is to store, with each class, only a table of the methods defined in that class, along with the number of the first method defined in the subclass, and the number of the base class. For example, suppose that Bear is a subclass of Mammal, class Mammal contains four methods, and class Bear contains two additional methods. Then the dispatch table entry for Bear contains (1) the number [4] of methods that it inherits, (2) entries for the two methods defined in class Bear, and (3) the number of class Mammal, where additional methods can be found.

(a) Write a pseudo-code algorithm to look up a method definition in the dispatch table. Assume that the i-th row of the table T contains information $T[i]$.baseclass (the number of the base class), $T[i]$.numInherited (the number of methods in the base class) and $T[i]$.localMethods (an array of methods for this class that are added). Lookup(c,n) should return method number n in class number c. Be careful to note that $T[i]$.localMethods[1] is the first method defined locally in class i, not the i-th method of class i.

(b) What is an advantage of this implementation of the dispatch table?

(c) What is a disadvantage?

40.7. A subclass typically adds methods to those that it inherits. Should it be allowed to remove methods, by choosing neither to inherit them nor to implement them? What problems might that cause?

40.8. Using the syntax of this chapter, write a definition of a class that is a subclass of Queue, but that also allows adding a string to the beginning of the queue instead of to the end. Include constructors.

40.9. C++ is an unusual object-oriented programming language in that, by default, it does not use the tag on an object to find a method in the dispatch table. Instead, it employs *static method selection*, where the class is the one associated with the type of the variable that holds the object. For example, if a variable m of class Mammal contains an object b of class Bear, then m.roar() uses the the roar method in class Mammal, not the one from class Bear. But if a method, such as roar, is declared *virtual*, then C++ uses the usual *dynamic method selection*, based on the tag of the object itself.

(a) We saw that an overriding method often wants to perform the method that it overrides. Argue that selection of that method is a kind of static selection, depending on the class where the overriding definition occurs, and should not be based on the tag of the object performing the overriding method.

(b) Give an example where static method selection will not work, and dynamic method selection is required.

(c) Explain how static method selection might lead to a more efficient program.

40.11. Bibliographic notes

LaLonde and Pugh [68] discuss pragmatics of inheritance. Meyer describes Eiffel [75], an object-oriented language based on multiple inheritance.

Chapter 41

Classes as Modules

41.1. Visibility

Modules are used not only to group things but also as a way of hiding some of them so that, while making modifications to a module, you know that none of its private things are visible in any other module. In object-based programming objects also possess notions of visiblity, so they share some characteristics of modules.

Object-oriented programming is heavily influenced by ideas of object-based programming. But, having moved method definitions from objects to the larger level of the class, it makes sense to move issues of privacy to that level too. That has an important consequence; each object can see private things inside other objects of the same class. For example, suppose that you want to augment the Queue class by an operation that concatenates two queues together. The concatenate operation is performed by one queue, the agent, with another queue q as a parameter. In object-oriented programming, the implementation of the concatenate method can use not only the private variables of the agent queue, but also those of the parameter queue q. That would not be possible in object-based programming.

Is the ability of one object to see private aspects of another reasonable? A good approach to answering that question is to ask what is gained and what is lost. Certainly, something is gained. Efficient and sensible concatenation of queues will almost surely require direct use of the representations of both queues, and the object-oriented view of privacy at the class level makes that possible. But losses are small. Remember that the main point of making things private is to ensure that you do not need to look outside of a module when making modifications to private things. The privacy constraints decouple different modules, making a program more flexible for modification. But since the class is the module, and all objects of a given class share the same implementation, allowing one object to make use of private things in another object of the same class does not make modification much more difficult. You still need to examine the entire class, and nothing more, when you make modifications to its private variables and methods.

Indicating visibility

A simple way to indicate visibility of components, employed by Java, is to mark each thing in a class as private or public. Another way to indicate visiblity is to separate a class into two parts, one describing the public aspects and the other telling additional (private) information. Only the part of the class that describes public features of the class is put in a place where it can be read by other modules. Cinnameg uses that approach.

Programming languages strive for ideals, but are often limited by practical considerations. In order to allocate memory for an object of a given class, it is often necessary to know how much space is needed, and that requires knowledge of information about the variables in the class. When positional inheritance is used, each subclass needs to know how much space is used by its base class in order to know where its own variables fit into the representation, and how to implement the selectors. Assigning numbers to methods also requires knowledge of the number of methods in the base class, including private methods. A consequence of those requirements is that, while compiling one class, a compiler typically needs to know private information about other classes.

Languages like Java that only employ visibility annotations make all information available to a compiler. Some languages, such as C++, allow you to separate a class into interface and implementation parts, but require you to list some information about private variables and methods in the class interface (marking them private) so that a compiler can see that information. Cinnameg avoids the necessity of adding private information to the class interface by calling functions, implicitly defined with the implementation part of a class, that provide the necessary information at run time. That is a more expensive way of doing things than what is done by C++ or Java. There are often tradeoffs between ideals and efficiency, and language designers need to decide where their priorities lie.

41.2. Protected visibility

Using classes instead of objects as modules increases visibility from one object to another in some cases, but it also has another effect: it can decrease visibility within a single object. We have classified each method or variable of an object as either public (accessible to everybody) or private (accessible only within the module of the object's class). Now bring the class hierarchy into the picture. Class LQueue of Figure 40.1 is derived from class Queue. If the members variable of Queue is private, then it is inaccessible in class LQueue, making it impossible to write the length method. Private variables are inherited, and every instance of class LQueue possesses them, but they can only be used in the class where they are created. So an object can contain variables (members) to which only some of its methods (those that are written in the module of class Queue) can refer.

From an object-based standpoint, where the objects themselves are thought of as the modules, it does not seem to make much sense to give some of the methods within an object special privileges not shared by other methods in the same object. But from the perspective of traditional modular software design, it makes very good sense to say that a particular concept (the existence of a variable called members, for example) can only be used in a single module, since then any change to that concept can only affect that one module.

So there are conflicting ideas about visibility. It is not clear whether privacy should be defined in an object-based or an object-oriented way. That is, should private variables and methods be accessible only in the module of a class, or in the module of that class and all of its subclasses? The solution that is typically adopted is to let the programmer decide, on a case-by-case basis, which notion of privacy to use. A variable or method that is marked *private* is accessible only in the module of the class in which it is defined. A variable or method that is marked *protected* is accessible not only in the class where it is defined, but also in the modules of all subclasses of that class.

41.3. Class components and classes as objects

Occasionally, a programmer needs a variable that is shared by all instances of a given class. For example, you might want to keep track of the total number of members of all queues of a given kind, using a variable that is shared by all queues, and that the Insert and Delete operations can increment and decrement. Some classes also need to have constants. For example, a class that is responsible for conversions between different distance measures probably needs to know that an inch is 2.54 centimeters. It does not make sense to put a copy of a constant in every object; the value is the same, and storing it in every object is wasteful of memory. Instead, a single copy of each constant should be shared by all of the objects.

Variables that are shared by all instances of a given class are called *class variables*, to distinguish them from variables that belong to the instances of the class. Similarly, a class constant is stored with the class, and is accessible to all instances of that class. Methods that belong to a class, rather than to instances of the class, are called *class methods.*

An obvious way to handle shared components is to associate class variables, constants and methods with the module of a class. Java takes that approach. (In Java, you label class variables and methods **static**.)

Classes as objects

There is another idea for handling class variables, constants and methods that is both versatile and intuitively appealing, illustrated by Smalltalk. Think of a class as an object factory. The factory itself is also a kind of object, and can have variables and methods. When you create class variables and methods, you are simply populating the class object with components. When you use a class method, you ask the class object to perform the method, just as you would ask any other object to perform a method.

One of a factory's methods produces a new object. So the constructors of a class are really part of the class object, not part of the instances of a class. (An automobile does possess the capability to build a new automobile.) To create a new instance of a class, you ask the class object to run one of its constructor methods.

An advantage of making classes objects is that they become first-class citizens; you can pass a class as a parameter to a method, return a class from a method, or store a class object in a variable. Of course, if a class is an object, you expect it to be an instance of some other class, such as a class of all object factories. Clearly, treating a class as an object has implications for the language design; Smalltalk's solution to these issues is discussed in Chapter 43.

41.4. Summary of terminology and concepts

A class plays the role of a module in object-oriented programming, in contrast to object-based programming, where individual objects are modules. In most cases, the effects are the same, and you imagine that objects are modules. Two notable differences are that, in object-oriented programming, (1) an object can see the private features of another object of the same class and (2) an object's private features are only visible in methods that are defined in the class where those private features are created. A protected component is visible in a class and all of its subclasses, and has visiblity that is more like visibility in an object-based setting.

Classes can sometimes play the role of objects. They can possess variables and methods, which are shared by all instances of the class.

Terminology

Class method/ Class variable A class method or variable is shared by all instances of the class. It does not belong to any particular instance.

Private A private variable or method is only visible in the module of the class where it is created.

Protected A protected variable or method is only visible in the module of the class C where it is created and in the modules of all subclasses of C, allowing all methods of an object, not just those methods that are defined in class C, to use the protected variable or method.

Public A public variable or method of class C is visible in all classes that know about class C.

41.5. Exercises

41.1. What is the difference between a private variable and a protected variable?

41.2. Does it make sense to have a protected method? Give an example.

41.3. Object-based and object-oriented programming have different notions of visibility. What is an important difference between the two, and what causes the difference?

41.4. Do you have to create an instance of class C in order to use one of C's class methods? If so, why? If not, which object would you ask to perform the class method?

41.5. Which objects should be able to use a private class variable of class C? Try to think of all kinds.

41.6. Can a class method refer to instance variables or methods of the same class? Why or why not?

41.7. Does it make sense to have a protected class variable? Explain. Where would it be visible?

41.8. Suppose that class B is a subclass of A, and that class A defines a protected instance variable V. An instance method m defined in class B takes a parameter, y, of (nominal) class A. Should m be allowed to use y's variable V? Consider the pros and cons, assuming that you are making the choice for a new programming language. Keep in mind that A might have other subclasses.

41.9. This continues the ideas of the previous question. Suppose that B and C are two subclasses of class A, and that class A defines a protected instance variable V. By inheritance, objects of class B and C all have variable V. Should a method that is part of an object of class B be able to use the variable V that belongs to another object of class C? Support your conclusion.

41.10. Show that, if checking of what is allowed is to be done at compile time, and if your answer to question 41.8 is that an object of subclass B should be able to see protected variables of objects of class A, then for all practical purposes, your answer to question 41.9 is yes, because an object of class B can already see the variable V that belongs to objects of class C. How can a class B object do that? Remember that you are allowed to store an object of class B or C into a variable of type A.

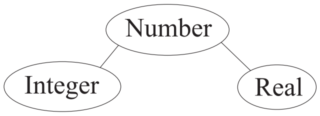
Chapter 42

Virtual Methods and Run-Time Typing

42.1. Virtual methods

Suppose that you need to implement arithmetic on numbers with large numbers of digits. You create two classes, class Integer to handle large integers and class Real for real numbers with very high precision. For each class, you implement the operations sum, difference and product. Now you decide to implement some operations that can be defined in terms of sum, difference and product. For example, you define sqr(x) $= x^2 = x \cdot x$ using the product method.

If classes Real and Integer are implemented separately, then you must write the sqr method twice, once in class Real and once in class Integer. But the point of the class hierarchy is to give you places to write methods that are shared by several classes. A class Number seems to be called for, serving as a place to write methods that are shared by classes Integer and Real. So you create a class hierarchy as follows.



Looking at the methods in more detail, suppose the product method runs inside one number (the agent) and takes another number as a parameter, returning the product of the parameter with the agent number. The sqr method does not need any explicit parameters; it just takes the product of the agent number with itself. An initial stab at writing class Number looks like as follows, where self indicates the object that is running the method.

```
class Number
  instance methods:
    sqr() = product(self)
end class
```

But there is a serious difficulty with that; the sqr method refers to a product method that is not defined in class Number. Rather, product is defined in classes Integer and Real. Since an object of class Number has no product method, it cannot perform such a method.

```
abstract class Number
  virtual instance methods:
    sum:        Number -> Number
    difference: Number -> Number
    product:    Number -> Number
  instance methods:
    sqr() = product(self)
end class
```

Figure 42.1: **Class Number with virtual methods.** The sqr method can use product, even though product is not defined here. The class is abstract to indicate that it has at least one virtual method.

An obvious thing to do is to define the product method in class Number. But that also meets with difficulties. The product method must be defined one way for real numbers and another way for integers. In fact, the product method has already been written in class Real, and in class Integer. It would not make sense to attempt to implement it again in class Number. The point of class Number is to capture similarities of real numbers and integers, not to implement them again.

In a nutshell, the problem is that every kind of number needs to have methods sum, difference and product, so those methods *logically* belong to class Number. But those methods cannot be defined in class Number; they must be defined in subclasses of Number. Methods that have this characteristic are called *virtual* methods. Using the terminology of Chapter 36, normal (nonvirtual) polymorphic methods are defined parametrically, intended to work the same way in every subclass. Virtual methods are defined in an ad-hoc manner, with a separate definition for each (immediate) subclass. Classes Real and Integer do not inherit definitions of sum, difference and product from Number, but instead inherit a *responsibility* to define those methods themselves.

42.2. Abstract classes

A class that has at least one virtual method should have no instances, since those instances would not possess at least one of the required methods. Only its subclasses should have instances. Classes that do not possess instances are called *abstract classes.* Figure 42.1 shows a definition of abstract class Number. Think of it as a place to attach shared parametric method definitions (and occasionally shared variables).

42.3. Virtual methods and union types

Chapter 34 shows how to define a type with multiple constructors. For example, Figure 34.9 defines a type BinarySearchTree with two forms, an empty tree and a nonempty tree. Subclasses and virtual methods offer another way to achieve the same goal.

Imagine that you intend to create a module for symbolic computation on expressions, as software such as Mathematica does. For example, you would like a method that takes the derivative of an expression with respect to x. There are several kinds of expression: the independent variable, x; a constant, such as 1.0; a sum of two given expressions; a product

```
abstract class Timer
  virtual instance methods:
    run: () -> ().
  instance methods:
    time() = t2 - t1 |
      Let t1 = currentTime();
      (run())
      Let t2 = currentTime();
end class
```

Figure 42.2: **Class Timer.** The function to be timed is virtual. For each function that you want to time, you create a subclass containing an appropriate definition of run.

of two expressions; etc. To handle that, you define class Expression and then add a subclass for each kind of expression. Since each kind of expression has its own rule for computing the derivative, the derivative method is virtual, and the methods for each kind of data are placed with that kind of data, in its class.

When a single union type is used instead, as done in Chapter 34, adding a new kind of expression requires modifying functions by adding a new case to handle that kind of expression. But the class approach avoids that. Adding a new kind of expression just requires adding a new subclass of Expression and defining the virtual methods for that class. There is no need to modify any existing classes or methods at all.

Now imagine providing expressions and symbolic computation on them in a library. Keep in mind that most programmers are not allowed to modify the library, but can only use it. The single-type approach allows the writer of the library to include a fixed collection of expression forms, but does not allow anyone else to add new expressions. Classes and virtual methods allow anyone to add new kinds of expressions in his or her program without modifying the library at all.

Sometimes you do not want to add a new variation but to remove one because it is no longer needed. That is also easy to achieve using the object-oriented approach; you just remove the subclass that implements that variation (and any place where constructors of that subclass are used). There is no need to modify the implementations of other classes.

42.4. Virtual methods as function parameters

Some programming languages allow you to pass functions as parameters to other functions, but most object-oriented languages do not allow you to do that. But sometimes you can simulate function parameters by using virtual methods.

Suppose that you want a function or method, time(f), that runs another function f and tells you how long f takes to run. In an object-oriented programming language, you create a class called Timer, as in Figure 42.2, containing a virtual method called run that represents the function that is to be timed. It is a *hook* where the actual function can be attached. To time a function, you define a subclass of Timer where the run method runs the desired function. Instead of asking one object to time the execution of a method in another object, you let the subclass inherit the time method, so an object of the subclass knows how to time itself.

Chapter 21 showed that the ability to pass functions as parameters to other functions allows the creation of versatile libraries containing functions that implement entire classes of algorithms. Virtual methods in class libraries can be used to achieve similar generality.

42.5. Summary of terminology and concepts

Virtual methods are a powerful mechanism that allows methods defined in a class to use methods that can only be defined in subclasses. That makes it possible to share code that otherwise would need to be written more than once.

Virtual methods provide an alternative way to implement union types that is more flexible than the approach of defining all of the variations in a single type definition. They can also be used to implement versatile library classes that can be customized by creating subclasses of them, and can be used to simulate passing methods as parameters to other methods.

Terminology

Abstract class An abstract class is a class that has at least one virtual method. No object will ever be created whose tag is an abstract class, since such an object would be without some of its methods.

Virtual method A virtual method is a method that logically belongs to a class A, but that can only be implemented in subclasses of A.

42.6. Exercises

42.1. Why aren't objects that are tagged by abstract classes typically created?

42.2. Our abstract classes have not had constructors. Can you think of a circumstance where an abstract class would need a constructor even though it is not possible to create an object whose tag is abstract class?

42.3. Does the dispatch table need a row for an abstract class? Explain your reasoning.

42.4. Using generic syntax, write class Expression, then add classes VariableExpr and NumberExpr, where there is just one object x, of class VariableExpr, and an object of class NumberExpr stands for a particular constant. Include definitions of the derivative method. What information does a VariableExpr need to hold? What about a NumberExpr? Be sure that each class only contains variables that it needs and that it contains an appropriate constructor.

42.5. Add a subclass SumExpr to Expression that represents the sum of two given expressions. See the preceding exercise.

42.6. Implement binary search trees, with insertion and membership testing, by creating three classes representing (1) a general binary search tree, (2) an empty tree, and (3) a nonempty tree. Make the member and insert methods virtual, and implement them in the subclasses. Make the insert function modify the tree. For simplicity, assume that each value in the tree is a string. Now add a method that adds all of the members of a given list to the tree. Where should that method be written?

42.7. Write a definition of linked lists using three classes, (1) a class LinkedList, (2) a subclass EmptyList of LinkedList, and (3) a subclass NonemptyList of LinkedList that holds a head and a tail of a list. Create and implement virtual methods null(), which tells whether the list that runs it is empty; head(), which returns the head of a list; and tail(), which returns the tail of a list. Add constructors nil(), which produces an empty list; and cons(h, t), which produces a list whose head is h and whose tail is t. Finally, write a function length() in two different ways. First, write it as a function entirely in class LinkedList. Then rewrite it as a virtual method, with a separate implementation in each subclass.

42.8. After implementing linked lists as in the preceding problem, write a class Mapper that has a virtual method f and a nonvirtual method, map, that takes a linked list L and produces the result of mapping f onto L. For example, if z is an instance of a subclass of Mapper that defines f by $f(x) = x^2$, then z.map([2,4,6]) = [2^2, 3^2, 4^2] = [4,9,16].

42.9. (This problem requires some familiarity with C++ and access to a C++ compiler.) In C++, write a class A with a virtual method. In C++ you are allowed to give an implementation of a virtual method. Implement the virtual method so that it prints something, so that you can see that it is running.

Now use the virtual method from inside a constructor for class A.

Now write a subclass B of class A, and override the virtual method so that it prints something different. Make a constructor for class B that uses the constructor for class A. Now use the constructor for class B to build an object of class B. Which implementation of the virtual method do you think will be used from inside the constructor, the one written in class A or the one in class B? Try the program. Which implementation of the virtual method was actually used? Can you explain what is happening?

42.7. Bibliographic notes

Virtual methods have been an important component of object-oriented languages since Simula [17]. The Timer class is similar to an example in Arnold, Gosling and Holmes [7].

Part XIV

Object-Oriented Programming Languages

Chapter 43

Object-Oriented Programming in Smalltalk

43.1. Introduction to Smalltalk

Object-oriented programming got its start with the programming language Simula, which introduced classes and inheritance by adding them to Algol 60. Objects and classes were primarily intended to aid in performing simulations. The Simula library contains a class called Simulation that provides support for simulations, handling the mechanics of creating, scheduling and selecting events, with the idea that a programmer should create a subclass of Simulation for doing a particular simulation, and for deciding just which events to schedule.

The convenience and flexibility of class libraries soon became apparent, and they were especially attractive for the emerging area of graphical user interfaces, where large, versatile libraries are a necessity. But, rather than grafting classes onto an existing language, as Simula had done, it seemed sensible to make classes the central concept of a new language, suitable for programming in an environment that employed graphical interactions and new (at the time) tools, such as the mouse. Smalltalk was developed in that spirit to take full advantage of classes and objects. As one of the first truly object-oriented languages, Smalltalk serves as a prototype for object-oriented languages today.

Smalltalk went through several revisions over time; Smalltalk-80 is the first dialect to be widely used. This chapter describes the general ideas behind Smalltalk, including some of its interesting characteristics, rather than the details of one version. We do not attempt to describe the entire language, and only touch on the library.

43.2. Classes

Smalltalk supports single inheritance. That is, each class has just one base class. There is one class, Object, that has no base class; it is at the top of the class hierarchy.

Describing classes

One unusual feature of Smalltalk is that it has no standard syntax that programmers are supposed to use for describing classes.[1] There is a syntax for writing individual methods, but none for listing instance variables of a class, or for indicating relationships among classes, or whether methods are public or private. Instead, a Smalltalk implementation should come with a graphical editor that gives you menu options, such as creating a new class, adding a variable, adding a method, or indicating whether a method is public or private. We will write classes in a pseudo-syntax similar to that used in preceding chapters.

Smalltalk is typeless, so variables are just listed, by their names, and there is no indication of the types of parameters, or of the result from a function. By convention, parameter names are chosen to have an indication of the type that you expect them to have. For example, a parameter called aNumber is one that you expect to be a number. But there is nothing to prevent you from passing any object as the parameter called aNumber.

43.3. Lexical issues

Smalltalk is case-sensitive, so be careful not to confuse *r* with *R*. Smalltalk if free-form, so a line break is treated like a space in most places. Comments are written in double quotes.

Constants and identifiers

Smalltalk identifiers begin with a letter and contain letters and digits. Identifiers are used as the names of variables, methods and classes. Private variables in a class have names that begin with lower case letters, and names of public variables (much less common) begin with upper case letters.

Numeric constants are written in the usual way, such as 58 or 91.4. Character constants are written preceded by a dollar sign. For example $X is the upper case *X* character, and $+ is the character +. Strings are written in single quotes, such as 'Smalltalk'. Smalltalk also supports symbols, similar to the symbols of other languages, such as Scheme. You write a symbol by preceding it by a # sign. Constant #lion is the symbol whose name is "lion".

Every object has a class, and classes have names that start with upper case letters. Constant 41.2, for example, has class Float, $A has class Character, 'a string' has class String and #yak has class Symbol. Integers are somewhat complicated because there are three classes, SmallInteger, LargePositiveInteger and LargeNegativeInteger, for integers. Each is a subclass of abstract class Integer. Constant 35 has class SmallInteger.

There is an object called **nil**, of class UndefinedObject, that is the default initial value of variables.

43.4. Expressions: using methods

All computation is performed by methods. There are three different syntactic forms for methods.

[1] Smalltalk programs have exchange syntax that is used for moving programs from one machine to another, but programmers normally do not see it.

1. A *unary* method has an identifier as its name, and is written immediately *after* the object that performs it. For example, expression (x isNil) returns true if x is nil, and (x reciprocal) returns $1/x$ when x is a number.

2. A *binary* method has a name that is one or two special characters, or just a comma, and is written between the object that runs it and its parameter. For example, $(x + 5)$ runs method + in object x with parameter 5, and (x `<=` 28) runs method `<=` in object x with parameter 28. Special characters are the members of string “`+/\*~<>=@%|&?!`”. Even fundamental operations such as addition and multiplication of numbers are accomplished by methods running in objects. Expression 3 * 4 asks object 3 (a number object) to run method * with parameter 4; it yields result 12, an object of class SmallInteger.

   There is only one level of precedence among binary operators, and all operators are left-associative. So expression $3 + 2 * 5$ yields 25; it is computed by first adding $3 + 2$, and multiplying the result by 5. Use parentheses to control the parsing. Unary methods have higher precedence than binary methods.

3. A *keyword* method takes one or more parameters, each preceded by a keyword. Each keyword ends with a colon, and the name of the method is formed by concatenating the names of the keywords. For example, expression

   ```
   frog jump: 2 over: toad
   ```

   selects a method called jump:over: from object frog, and runs it with parameters 2 and toad. Keyword messages have lowest precedence.

The Smalltalk library provides a large collection of methods. Numbers support the binary methods $+$, $-$, $*$ (multiplication), `/` (division), `//` (integer division, throwing away the remainder) and \\(integer remainder, or modulus). Division of integers can yield objects of class Fraction, which are stored as fractions. So expression 1/3 returns the number 1/3, without rounding.

If s is a string, then (s at: n) is the n-th character of s, and (s at: n put: c) modifies string object s, replacing its n-th character by c. Expression (s, t) yields the concatenation of strings s and t, a new string object. For example, ('choco', 'late') yields string 'chocolate'. Parentheses are not required; they are only shown to set off the expressions.

43.5. Defining methods

A method definition begins with a heading that looks like a method call, with the object that performs the method missing. After that is a sequence of statements, separated by periods. (So a period is a separator, not a statement ender.) Each statement in a method definition has one of the following forms.

1. An expression is also a statement. It is evaluated, and its result is ignored.

2. An *assignment* has the form *variable* := *expression*, though some Smalltalk dialects use ← in place of :=.

3. A return statement has the form ^*expression.* It evaluates the expression and returns its value as the result of the method. Some version of Smalltalk use ↑ in place of ^.

For example, the following two method definitions define a keyword method called add:, with one parameter (called aNumber), and a unary method called jump. They assume that there is an instance variable called amount in the class.

```
add: aNumber
  amount := amount + aNumber

jump
  amount := amount - 1.
  ^amount
```

Self

A method refers to the object that contains it as **self**. You refer to variables that belong to self by their names, but to run a method that is part of self, you must mention self explicitly. For example,

```
leap
  self jump.
  self jump
```

defines unary method leap, which jumps twice. If a method does not return any value, as leap does not, then by default it returns self.

Cascading

If you want to ask one object to perform several methods in a row, you can *cascade* methods using a semicolon. Think of a semicolon as standing for the object that performed the preceding step. So the leap method can be written as follows.

```
leap
  self jump; jump
```

Local variables

A method can have local variables, for use only while running that method. At the beginning of the method, write the names of the local variables inside |. . . |, separated by spaces. For example, the following method returns aNumber to the eighth power.

```
eighthPower: aNumber
  |aNumberSq aNumberFour|
  aNumberSq := aNumber * aNumber.
  aNumberFour := aNumberSq * aNumberSq.
  ^aNumberFour * aNumberFour
```

Method selection

Smalltalk relies on a dispatch table, looking up a method based on the tag of the object that is asked to perform the method. There is an exception, however. To allow an overriding method to get the method that it is overriding, pseudo-object **super** is the same as self, but the method is selected from the row for the superclass of the current class, regardless of the tag on the object. A typical method that is overridden is one that initializes an object.

initialize: anObject
```
super initialize: anObject.
x = 0
```

initializes variables from the base class and then initializes x, which might be a new variable.

43.6. Blocks

Early versions of Smalltalk were quite functional in appearance, and methods could be defined by a set of (semantic, if not syntactic) equations. For example, a definition of factorial in Smalltalk-71 looks like this.

to 'factorial' 0 **is** 1
to 'factorial' :n **do** 'n*factorial n-1'

Later versions of Smalltalk opted for a more imperative feel, but the functional heritage is still present in Smalltalk's first-class functions, called *blocks*. A block is a function that is written in-line inside a method. To create a block with no parameters, enclose a group of statements inside square brackets. For example, statement

```
blk := [count := count + 1.
        size := size - 1]
```

puts a block into variable blk that, when run, adds 1 to count and subtracts 1 from size. A block is an object, and, to run it, you ask it to perform unary method **value**. So

```
blk value
```

has the effect of adding 1 to count and subtracting 1 from size. A block produces a value, the value of its last expression. So

```
x := 5.
aBlock := [x + 1].
x := 9.
val := aBlock value
```

sets variable val to 10, the value of expression $x + 1$ at the time when the block is evaluated. There is an empty block, [ ], whose value is nil.

A block can have parameters, just as other functions can. A block with one parameter called x has the form [:x | . . .], and a block with two parameters x and y has the form [:x :y | . . .]. In general, each parameter is preceded by a colon, and the list of parameters is ended with a vertical bar. To evaluate a block with one parameter, use keyword method **value:**, with the parameter. A block with two parameters uses method **value:value:**, and a block with n parameters uses method **value:value:. . . value:**, with n parameters passed to it. For example,

```
z := [:x :y | x + (y * y)] value: 3 value: 5
```

sets z to 28.

Blocks are not truly autonomous functions. If, during evaluation, a block performs a return statement ($\hat{}E$) then it causes the method that evaluates the block to return. So

```
x := [^1] value.
y := 0
```

will not set x or y to a value. It will return value 1 from the method when it evaluates the block.

43.7. Boolean objects and comparisons

Smalltalk provides a standard class, Boolean, with two subclasses, True and False. Class True has just one instance, **true**, and class False has just one instance, **false**. Methods that produce Boolean results include comparisons of numbers (=, ~=, <, <=, >, >=).

You can compare any two objects to see whether they are the same object. Expression $x == y$ yields true if x and y are the same object, and x~~y yields true when x and y are different objects. Alternatively, you can use = and ~=, but the meanings are different. The idea of = is that $x = y$ should be true if objects x and y represent the same information, and each class typically decides how to implement that equality test. For example, two strings are equal under = if they contain the same characters, in the same order, but they are equal under == if they are exactly the same object (stored at the same place in memory).

43.8. Conditionals and loops

Smalltalk does not have any built-in control structures, such as conditionals and loops. The philosophy is to make the language small and simple, and to do as much with objects and classes as possible, pushing the language complexity and versatility into the library. Conditional expressions and loops are handled as methods of the Boolean class and of blocks.

If a is an object of class Boolean and b and c are two blocks, then (a ifTrue: b ifFalse: c) produces (b value) if a is true, and produces (c value) if a is false. For example, the Integer class provides unary method **factorial**, so that

```
x := y factorial
```

sets x to the factorial of y. The following is an implementation of the factorial method. Since it belongs to the Integer class, self is an integer.

```
factorial
  self = 0
    ifTrue: [^1]
    ifFalse: (self > 0 ifTrue: [^self * (self - 1) factorial]
                       ifFalse: [self error:
                                  'factorial of a negative number']
```

You can also use method ifTrue: (which does nothing if the object is false) and ifFalse: (which does nothing if the object is true). An alternative implementation of factorial is as follows.

```
factorial
  self = 0 ifTrue: [^1].
  self > 0 ifTrue: [^self * (self - 1) factorial].
  self error:  'factorial of negative number'
```

It is important that the arguments of ifTrue: and ifFalse: are blocks, for two reasons. First, you do not want to run them before you even perform the test; they should be deferred, as blocks are. Second, the ifTrue:ifFalse: method will ask one of those parameters to perform method **value**, as blocks do.

You can create a loop using method whileTrue: on a block object. If b and c are blocks, then (b whileTrue: c) evaluates block b, which should yield a Boolean result. If b

yields value true, then block c is evaluated, and b is evaluated again. Evaluation continues until b yields false. When the loop is done, it returns nil. For example, here is another implementation of factorial, this time using a loop. Remember that self is the integer whose factorial you are computing.

```
factorial
  |n r|
  n := self.
  r := 1.
  [n > 1] whileTrue: [
    r := r * n.
    n := n - 1].
  ^r
```

You can ask an integer n to evaluate a block n times. For example,

```
5 timesRepeat: [n := n * n]
```

squares n five times. Integers can also create counting loops. Statement

```
1 to: 10 do: [:i | count := count + i]
```

performs the indicated block for $i = 1, \ldots, 10$.

43.9. Constructors

Among other things, a class is an object factory, and it performs methods that create new instances of itself. Constructors are class methods, and each new class is automatically given a constructor called **new** (or **new:** — see indexed variables below) that creates a new, uninitialized member of the class. For example, expression `Widget new` asks the Widget class, as an object, to run its new method, which returns a new instance of class Widget. You can create other constructors by any names that you like. For example, if instances of class Point are points in the plane, then you might create a constructor called x:y: that creates a new point with a given x- and y-coordinate, as follows. It is a class method of class Point, so self is class Point, and expression (self new) creates a new, uninitialized Point.

```
x: x y: y
  p := self new.
  p initializex: x y: y.
  ^p
```

Instance method initializex:y: of class point is as follows, assuming that class Point has two instance variables x and y.

```
initializex: xNumber y: yNumber
  x := xNumber.
  y := yNumber
```

Now, the following creates a new initialized point called p.

```
p := Point x: 0 y: 2
```

Notice that this uses Point as an object, asked to run method x:y:.

43.10. Arrays

Constant arrays

Smalltalk supports lists, which it calls *arrays.* The list that would be written [1, 2, 3] in Cinnameg is written #(1 2 3) in Smalltalk. Since Smalltalk is typeless, you can mix different kinds of things in an array, and there is no distinction between a list and a tuple. Smalltalk value #(1 $X) corresponds to the ordered pair (1, 'X') in Cinnameg. You can only write an array of constants in this syntax. If x is a variable, #(x) does not give you an array that contains the value of variable x, but instead gives you a singleton array holding the symbol #x.

To create a list of lists, you only write the # sign once. For example, Smalltalk's #((1 'one') (2 'two')) is similar to the constant [(1, "one"), (2, "two")] in Cinnameg. To write a symbol in an array, you do not precede it by #. So #(one two three) is an array of three symbols, #one, #two and #three. The empty array is #( ).

Creating and working with arrays

You can create an array of a given size n, filled with nil values, using expression (Array new: n). For example,

```
a := Array new: 10
```

creates a new array of size 10. Operations on arrays include the following. In all cases, x is an array. Parentheses are shown to set off expressions, but are not required.

1. (x at: n) is the n-th value in x, numbering from 1. Arrays are objects, and can be changed. Statement (x at: n put: v) modifies array x so that (x at: n) yields v.

2. (x first) yields the first member of x, and (x removeFirst) returns a new object, the array produced by removing the first member of x. So first and removeFirst are similar to the Cinnameg head and tail operations.

3. (x addFirst: v) produces a new array object obtained by adding v to the front of array x. AddFirst is similar to the Cinnameg :: operator. For example, if x is array #(1 2 3) then (x addFirst: 0) yields a new array holding #(0 1 2 3). Similarly, (x addLast: v) produces a new array object that looks like x, but with v added to the end.

4. (x size) is the current size of array x.

5. The comma operator concatenates two arrays. So (x, y) is a new array object that holds all of the members of array x followed by all of the members of array y.

6. (x do: b) takes a block b with one parameter. It evaluates (b value: v) for each value v in array x. Use this to loop over the members of an array, where block b is the body of the loop. It is similar to map in Cinnameg.

For example,

```
a := Array new: 20.
1 to: 20 do: [:i | a at: i put: 0]
```

creates an array of 20 values, all 0.

Indexed variables

A major design goal of Smalltalk is to provide a relatively small number of primitive concepts and to let all other ideas be written in Smalltalk. The array class, for example, is written in Smalltalk, as part of the library. But then, where does the basic representation of arrays and indexing come from?

Smalltalk allows a class to have a single implicit array, called an indexed variable, accessed by an index instead of by name, using at: and at:put: methods. A class that has an implicit array uses constructor new: rather than new, where the parameter tells the size of the array. The array class is simply defined to have an indexed variable. Remaining methods, such as do:, are written in Smalltalk.

43.11. An example class

Figure 43.1 shows a class of queues, similar to the queues in Chapter 39. Recall that Smalltalk has no standard syntax for describing classes, only for methods. So we use the previous syntax.

43.12. Overriding and virtual methods

When you create a subclass of another class, you can override methods, as discussed in Chapter 40. Smalltalk has no special support for virtual methods or abstract classes. Instead, if a method cannot be implemented in a given class, then it is defined to yield an error, and should be overridden in subclasses. For example, if method find: cannot be implemented in class Collect, then it is written as follows.

```
find: aValue
  self error: 'find not implemented for class Collect'
```

An abstract class is any class that contains a method that must be overridden in subclasses.

43.13. Classes as objects

When you run a constructor or other class method, you think of the class an an object that performs the method. So classes are objects. But in Smalltalk, every object has a class, so every class must itself belong to a class. Think of a class as an object factory. The Queue class cannot belong to class Queue, since a queue factory is not a queue any more than a bicycle factory is a bicycle.

Metaclasses

The solution adopted by Smalltalk is to create new classes, called *metaclasses*, whose instances are other classes. In fact, for every class C, there is a metaclass called *C class*, whose only instance is class C. For example, class *Integer class* has just one instance, class *Integer*. The metaclasses are related to one another in the same way as their instances. For example, *SmallInteger* is a subclass of *Integer*, so *SmallInteger class* is a subclass of *Integer class*. At the top of the hierarchy of metaclasses is a class called *Class*. All metaclasses are subclasses of *Class*.

```
Class Queue
  instance variables:
    "members is an array of the members of the
    queue, from front to back"
    members
  class methods:
    new
      q := self basicNew.
      q initialize
  instance methods:
    insert: anObject
      members addLast: anObject
    delete
      |result|
      result := members first.
      members := members removeFirst.
      ^result
    makeEmpty
      members := Array new: 0
    isEmpty
      ^(members size = 0)
  private instance methods:
    initialize
      self makeEmpty
End class
```

Figure 43.1: **A Queue class, with methods written in Smalltalk.** This class overrides new. Standard class method **basicNew** is the initially the same as **new**, and it can be used when **new** is overridden.

Metaclasses are used for organizing the Smalltalk class editor. The variables and methods listed under the metaclass are class variables and class methods, as described in Chapter 41. They can be used in instance methods or in other class methods.

Getting the class of an object

Every object responds to unary method **class** with its class. So

```
c := 3 class.
```

sets c = `SmallInteger`. Suppose that, inside class Widget, you try to make another object of the same class as self. You might try the following.

```
cpy := Widget new
```

But that creates a new object of class Widget. Remember that methods should be polymorphic, and you while writing the methods of class Widget, you should think about subclasses that might be added later. A more sensible polymorphic method would work as follows.

```
cpy := self class new
```

Now, regardless of the actual class of self, cpy has the same class, and this method is suitable not just for Widget but for any subclass of Widget.

The class of a metaclass

If all classes are objects, and every objects has a class, then even the metaclass `Integer class` must have a class. Every metaclass is an instance of class MetaClass. All of this yields a fairly complex hierarchy, and, in truth, there is more to metaclasses than we have space to go into here. But the key point is that Smalltalk, by its organization of classes and metaclasses, sticks to the principle of object-oriented programming that every object has a class, and simultaneously allows you to think of a class as an object that performs class methods, such as constructors.

Introspection

Smalltalk allows a program to do much more than just get the class of an object dynamically. A program can inspect the class hierarchy by moving up and down (to superclasses and subclasses), getting metaclasses, and finding methods and variables. It can even add new methods and variables to a class or remove methods and variables while the program is running. A program can write code, as a string, compile it on the fly, and add it to a class. That allows the Smalltalk development environment itself to be written in Smalltalk. Just one example of the methods that are available is sourceCodeAt:, where, for example, expression

```
Array sourceCodeAt: #do:
```

produces a string that is the source code of the do: method in class Array.

43.14. Summary of terminology and concepts

Smalltalk is a small, typeless object-oriented programming language. All data items are objects, and every object has a class. Classes are also objects.

The syntax of Smalltalk is unique, quite different from other object-oriented languages. The semantics of Smalltalk, on the other hand, is not unusual, but is prototypical for object-oriented languages.

Blocks are a powerful concept, allowing Smalltalk to forgo syntax for conditionals and loops. Instead, control structures are written as methods of classes, and you can add new control structures. Most of the methods that you use are written in Smalltalk, as part of the library.

Smalltalk allows a program to manipulate the definitions classes, adding and removing methods on the fly.

Terminology

Binary method A binary method takes one parameter in addition to the object that runs it. It is written between the object and the parameter.

Block A block is a piece of code surrounded by square brackets. It can be run at any point.

Cascading A cascade asks one object to perform more than one method in succession. It is indicated by a semicolon.

Keyword method A keyword method can take any number of parameters, one for each keyword in the method name.

Metaclass A metaclass is a class whose instance is another class.

Self Expression `self` inside a method refers to the object that is performing the method. Inside a class method, self is the class.

Super Expression `super` inside a method refers to the object that is performing the method, but requests that the method be chosen from the superclass of the class where the method is defined.

Unary method A unary method is a parameterless method written after the object that performs it.

43.15. Exercises

43.1. Is Smalltalk intended to be a large or a small language? How does it provide a great deal of functionality to the programmer?

43.2. What do you think is the purpose of having two subclasses, True and False, of Boolean, rather than just having two instances, true and false, of class Boolean. Think about how operations such as ifTrue:ifFalse: are implemented. Does having two subclasses simplify their implementation?

43.3. Are there any dangers in allowing a program to make modifications to classes on-the-fly? Explain what might happen.

43.4. Smalltalk allows most methods to be overridden in subclasses, but encourages you not to override some of them. Explain difficulties that can happen if you override some library methods.

43.5. Blocks have class BlockContext. Write a definition of the whileTrue: method of the BlockContext class. You should not need to use a loop to do this. Smalltalk allows method definitions to be recursive.

43.6. Section 13.4 discusses tail recursion. Based on your answer to question 43.5, do you think that the Smalltalk system performs efficient tail recursion? Why or why not?

43.7. Write an implementation of method quadraticWithCoefficients: a and: b and: c. It should be part of the Float class, and should return array (y,s) where s is an array of the real solutions of equation $ax^2 + bx + c = 0$ and y is the value of expression $ax^2 + bx + c$ evaluated at $x =$ self.

43.8. Most object-oriented systems do not allow you to modify the classes in the library, but instead encourage you to create subclasses of them. Based on what you have read in this chapter, do you thing that a Smalltalk system allows you to add a new method to library class Float? Why or why not?

43.9. Write an implementation of stacks in Smalltalk as a class Stack.

43.10. For question 43.9, why is it not necessary to write a parameterized class Stack(T)?

43.11. A block normally does not outlive the method that created it. Can you see how a block might outlive its method? Can you see a problem with that? Explain.

43.16. Bibliographic notes

Texts on Smalltalk include Goldberg and Robson [41], Budd [19], LaLonde and Pugh [66, 67], Shafer, Herndon and Rozier [97], and Smith [98]. The journal, *The Smalltalk Report*, contains numerous articles on Smalltalk and how to use it. Alan Kay [63] gives a history of the development of Smalltalk. The Smalltalk-71 factorial method definition is from Kay's article.

Chapter 44

Object-Oriented Programming in Java

44.1. Introduction to Java

Java is an object-oriented language that was designed to borrow much its syntax from C++, but much of its semantics from Smalltalk. The goal was to make it easy for programmers who were already familiar with C++ to learn Java. Java is not designed to be compatible with C++, however; a Java program cannot be compiled by a C++ compiler, and a C++ program cannot be compiled by a Java compiler. The designers of Java decided that it was better to be compatible with C++ programmers than with C++ programs. Roughly speaking, the core of Java can be described as statically typed Smalltalk with C syntax.

One important difference between Java and C++ concerns memory management. Where C++ requires the programmer to manage memory in the heap, Java follows Smalltalk's lead and relies on a garbage collector. That eliminates some difficult issues that C++ programmers must contend with.

Java is still a work in progress, and has gone through versions 1.0 to 1.6 as of this writing.[1]Java 1.0 introduced the language, and some modifications to the language were made with the introduction of Java 1.1. Quite a few of the modifications were concerned with the library, not the language, but Java 1.5 involves significant extensions to the language, as well as changes in the library. For example, generics, discussed in Chapters 36 and 37, were added as of version 1.5.

This chapter describes the basics of Java along with a few interesting features, but does not attempt to describe the entire language. This chapter is not concerned with the extensive class library that, to many people, is what makes Java what it is. The reader is encouraged to consult any of the numerous books about Java for more information.

44.2. Types

Java is a weakly statically typed language; some type checking is done at compile time, some at run time, and some (depending on what you expect from the language) is not done

[1] Sometimes the version is only given as the number after the decimal point. For example, Java 1.6 is also called Java 6

at all.

A class is a type, and, if B is a subclass of A, then an object of class B can be used in a context where an object of class A is needed. For example, an assignment statement can make $x = y$ in cases where y has type B and x has type A. You can also convert from a class to its subclass, as long as the conversion is explicit. For example, suppose that Bear is a subclass of Animal. In

```
Animal a;
Bear b;
...
b = (Bear) a;
```

the last statement calls for a run-time type check. If variable a happens to hold a bear, then it succeeds, and stores that bear into variable b. But if a does not hold a bear, then it fails, throwing an exception. Section 44.10 discusses exception handling.

You can avoid the possible exception by checking whether an object belongs to a given class before trying to perform the conversion. Expression (x instanceof C) evaluates to true if object x is an instance of class C or a subclass of C. So a program that needs to do a down-conversion usually asks first whether the down-conversion would be valid. For example,

```
if(a instanceof Bear) {
  Bear b = (Bear) a;
  ...
}
```

only performs the down-conversion when it is guaranteed to work.

Primitive types

Java makes a distinction between objects (which carry tags and have a basic set of shared capabilities) and primitive data values, such as integers and real numbers. The motivation is to keep handling of the most basic kinds of data efficient, not unnecessarily burdened by the need to support capabilities that are rarely needed.

Java's primitive types include **boolean** (true or false), **char** (characters), **int** (32 bit integers), **long** (64 bit integers) and **double** (64 bit real numbers). The primitive types have names that begin with lower case letters, while classes names are usually capitalized. Java is designed to use the *unicode* alphabet, and values of type char are unicode characters. You can get any unicode character by giving its number in hexadecimal form after \u. For example, '\u0034' is unicode character 52, which is '4'.

Boxing

Since primitive values do not have the capabilities of objects, there are restrictions on how they can be used. To allow you to treat them as objects, each primitive type has an associated class. For example, associated with type **int** is class **Integer**. In versions of Java before 1.5, you were required to convert between primitive types and their classes explicitly. But Java 1.5 introduces coercions that perform such conversions implicitly. For example, if you use a value of type int in a setting where class Integer is required, a conversion (called a *boxing* conversion) is inserted for you. Similarly, conversion from Integer to int (*unboxing*) is automatic.

44.3. Classes and methods

Classes

As in Smalltalk, all computation is performed by methods written in classes. A class definition

```
class classname extends baseclassname {
  class features
}
```

creates class *classname* with base class *baseclassname*, where *class features* list the variables, constants and methods of the class. If the **extends** phrase is omitted, the base class is assumed to be Object, which is a superclass of every class but itself.

Instance variables

You create an instance variable by writing its type and name, followed by a semicolon. For example,

```
int size;
```

creates variable size, of type int. You can give an initial value of an instance variable, as in `int size = 1;`, but you typically initialize variables in constructors, discussed in Section 44.5

Instance methods

Instance methods are written using a syntax that is similar to that of C; examples are given in Chapters 14 and 16. You ask an object x to perform method m with parameters a, b, and c by writing $x.m(a,b,c)$. For example, `frog.bumpsize()` asks an object called frog to run its bumpsize method with no parameters.

Each instance method can refer to the instance variables and instance methods of the same object by omitting an agent. For example, a method that adds 1 to the size variable of the current object is written as follows.

```
void bumpsize() {
  size = size + 1;
}
```

Another method can use bumpsize without an agent.

```
void morph() {
  bumpsize();
  bumpsize();
}
```

The bumpsize methods are performed by the same object that performs the morph operation. (Compare to Smalltalk, where you always need to include an agent, even if that agent is self.) Occasionally, an instance method needs to refer its object (self in Smalltalk). It uses reserved word **this** for that purpose. The morph method can be written in a more Smalltalk-like fashion as follows.

```
void morph() {
  this.bumpsize();
  this.bumpsize();
}
```

44.4. The class as a module

Each variable or method definition can be preceded by one of the words **private**, **protected** or **public**, where **private** indicates that the variable or method can only be referred to explicitly by methods that are part of this class, **public** indicates that the variable or method can be referred to anywhere, and **protected** is used with inheritance, making the variable or method visible in subclasses.[2] For example, here is a public method that adds n to the current object's size variable.

```
public void upsize(int n) {
  size = size + n;
}
```

If you do not indicate a visibility, the default is that the variable or method is visible everywhere in the package that contains it. Packages are discussed in Section 44.9.

Class variables and methods

Instance variables and instance methods are associated with instances of a class, not with the class itself. You can create *class variables* and *class methods*, associated with the class itself, by using modifier **static**.[3]

Some fundamental functions are provided as static methods in standard classes. For example, the sqrt function, which takes the square root of a number, is found in the class java.lang.Math. Its definition in that class begins

```
public static double sqrt(double x)
```

To use a class method or variable, you write the name of the class, a dot, and the name of the variable or method. For example, to set y to the square root of x, you write `y = java.lang.Math.sqrt(x)`, where the class is java.lang.Math, and the method is sqrt. If you are writing a method inside the java.lang.Math class, you do not need to indicate the class, and can write `y = sqrt(`x`)`.

Named constants are typically static, since there is no sense in forcing each object to carry its own copy of a constant. You indicate that a variable is a constant by using modifier **final**.[4] Class java.lang.Math contains a definition of constant PI.

```
public static final double PI = 3.14159265358979323846;
```

[2] A protected method is actually visible not only in a given class and its subclasses, but also in all classes in the same package.

[3] The word **static** is not a very good name for a class variable or method, but it is borrowed from C++, which borrowed it from C, where it meant something else and actually made some sense.

[4] Curiously, the modifier **final** is not borrowed from C++. In C++ the word **const** is used instead. You have to wonder why some other features of C++ had to be kept.

Initialization blocks

When you create a class variable, you typically give it an initial value. You can perform more complex initialization by writing a *static initialization block* in a class. Block

```
static {
  cv = 0;
}
```

is run once when the class is created. If you omit the word **static**, and just include a block that is not associated with a method, then that block is implicitly performed at the beginning of every constructor in the class.

The main program

When you run a Java class MyClass, the interpreter runs method MyClass.main, which must have the following heading.

```
public static void main(String[] args)
```

It takes an array of strings as a parameter (see Section 44.8), and produces no result. The array consists of the parameters of the command that invokes the interpreter.

44.5. Building objects

Instance variables are typically initialized by *constructors*, which appear to be functions with the same name as the class and without a return type. For example, a constructor for class Animal that sets instance variable size to a given value might look as follows.

```
public Animal(int s) {
  size = s;
}
```

To use a constructor for class C with parameter a, b and c, write `new` C(*a*,*b*,*c*). For example,

```
Animal dog = new Animal(3);
```

creates a new instance of class Animal and runs the constructor Animal(3) to initialize the new object.

Uninitialized variables

If you do not give an initial value to an instance or class variable, then it is given a standard default value: 0 for numbers, false for boolean variables, '\u0000' for characters, and **null** for objects. Pseudo-object **null** indicates a non-object, and you use it in places where you do not know an object. Do not ask null to perform an action; it is not an object.

A local variable in a method is never initialized for you, and you cannot use it until you have given it a value. In fact, you are required to write every method in such a way that a compiler can check that each local variable is given a value before it is used.

Using one constructor inside another

Normally you use a constructor in conjunction with the **new** operator. But Java also allows one constructor to use another one. For example, suppose that a constructor for class Widget has a parameter x. It wants to call the parameterless constructor, Widget( ), but wants to perform additional initialization as well. That constructor can be written as follows.

```
Widget(int x) {
  this();
  ...  (additional initialization)
}
```

with this( ) indicating a call to the parameterless constructor. Any constructor can call any other one to perform initialization, by calling it **this**.

If a class has a superclass other than Object, it needs to initialize its inherited variables, and that is done by running a constructor in its base class. Simply replace **this** by **super**.

44.6. Copying and comparing objects

When a program stores an object in a variable, it always stores a *reference* (or pointer) to the object. When you put an object in another variable, the reference is copied; the object itself is not copied. For example,

```
Animal sheep = new Animal(4);
Animal ram = sheep;
```

leaves variables sheep and ram referring to the same object. Any changes that are made to one of them are seen in the other. For example, using the bumpsize method that adds 1 to the size, `sheep.bumpsize()` causes both sheep's and ram's size to become 5, since sheep and ram refer to the same object.

The same kind of thing happens with parameter passing. When an object is passed as a parameter to a method, the object is not copied, and the method gets access to the same object as the caller. Any changes that the method makes to the object will be seen by the caller when the method returns. Objects are always passed by reference.

Sometimes you want to copy (or *clone*) an object. There is a method called clone that most (but not all) objects support. Expression x.clone( ) yields a new object that is a copy of object x. But be careful with clone. As discussed in Chapter 15, the meaning of copying is not always obvious. The difficult issue is, when object x refers (through one of its instance variables) to another object y, does a copy of x include y itself, or a copy of y? In Java, the meaning is defined by the programmer in the definition of the clone method for a given class.

Standard comparison operator == asks whether two objects are the same object. (It compares the pointers.)

44.7. Overriding methods

A derived class normally inherits operations from its base class. But it can also override methods, replacing inherited methods with new ones. To override a method, just write a definition of it in the subclass. The new definition implicitly overrides the inherited one for that subclass. To use the method *m* that would have been inherited, use notation super.*m*.

Sometimes, you want to prevent a method from being overridden in subclasses of a class that you are writing. (One reason is to ensure that you know just what every object of this class will do when asked to perform that method.) Using modifier **final** on the method prevents it from being overridden. For example, if you want to ensure that method getsize returns the value of the size variable, write

```
public final int getsize() {
  return size;
}
```

44.8. Strings and arrays

An array is an object. An array of integers has type int[ ] and, in general, *T*[ ] is the type of an array each of whose members has type *T*. The constructor for an array takes the size of the array as a parameter, written in square brackets, and creates an array all of whose members are initialized to their standard default values. So

```
int[] nums = new int[20];
```

binds variable nums to a newly created array object holding 20 integers, each initially 0. Expression nums.length yields the size of array nums.

Arrays are indexed from 0, and nums[*k*] is the member of array nums at index *k*. For example, here is a class method that sums all members of an array of integers and returns the sum.

```
public static int sum(int[] nums) {
  int s = 0;
  int k = 0;
  while(k < nums.length) {
    s = s + nums[k];
    k = k + 1;
  }
  return s;
}
```

Arrays and inheritance

Arrays types mimic the class structure of non-arrays. If *B* is a subclass of *A*, then *B*[ ] is also a subclass of *A*[ ]. That is, in the terminology of Section 36.7, arrays are covariant. For example, suppose that Canine is a subclass of Mammal. Then it is acceptable to write

```
Mammal[] A = new Canine[];
```

since an array of canines is considered to be a special case of an array of mammals. You have to be careful with this. Suppose that Feline is another subclass of Mammal. What happens if you write the following?

```
Canine[] M = new Canine[];
Mammal[] A = M;
A[0] = new Feline();
```

From a type standpoint, it should be acceptable to put a new Feline into $A[0]$, since the type of $A[0]$ is Mammal, and a variable of type Mammal should be able to hold a Feline. But the object stored in A is M, an array of Canines, and it does not make sense to set $M[0]$ to hold a Feline. The assignment $A[0]$ = new Feline( ) will cause an exception to be thrown at run time.

Strings

Java provides a standard class String whose instances are similar to (but not identical to) arrays of characters. A string constant, written in double quotes such as "string constant", has type String. Operations include binary operator + (string concatenation), method length (string length) and method equals (string equality test). For example, s.length( ) is the length of string s. Notice the empty set of parentheses. For strings, length is a method. For arrays, it is treated like a variable. You can convert between strings and arrays of characters. Constructor **new** String(a) is a string containing the characters in array a, and String method toCharArray( ) produces an array of characters.

You can get the i-th character of string s, indexing from 0, as s.charAt(i). Be careful about string equality. A string is an object, and expression ($s == t$) is true if s and t are the same object. To ask whether objects s and t hold the same sequence of characters, write expression s.equals(t).[5]

Many classes have a toString( ) method that converts an object to a string that describes the object. If one of the parameters of the string concatenation operator + is a string and the other is not, the nonstring parameter is converted to a string implicitly by using toString. In the case of a primitive type, the value is converted using a standard conversion. For example, expression ("round " + 21) yields string "round 21".

44.9. Java packages and the Java library

The basic unit of a Java program is a class, and a source file such as Widget.java must contain a definition of class Widget. But classes can be collected together into a package by writing `package` *pname;* at the beginning of each class file that is part of package pname. In addition to the notions of private, protected and public visibility, there is a notion of package visibility; a package-visible definition is visible throughout the package, but not outside the package. That allows a few classes to share some definitions, even when none are subclasses of the others.

A Java compiler makes two passes over a source file. The first pass is used to find out what is being defined, and the second pass compiles the definitions. Because of this, a definition in a Java class can refer to things that are defined later in the class.

Naming

To import other modules, most languages require a program to indicate the location of the module in the file system. Java avoids that by giving the things that a program uses names that tell where they can be found. Package names are chosen to match the directory or folder structure that contains the packages. For example, all files that belong to a package

[5] String *constants* are *interned*; they are stored in a table, so that two occurrences of a given string constant are the same object. So "dog" == "dog" is true. You can intern any string; expression str.intern( ) yields an interned version of string str.

called pqr.stu must be found in subdirectory stu of directory pqr. The definition of package java.lang.Math is found in file Math.java in subdirectory lang of directory java.

In a program, you can refer to each thing by its full name. For example, to set y to the square root of x, you would write

```
y = java.lang.Math.sqrt(x);
```

But you often prefer to shorten that. Directive

```
import java.lang.Math;
```

makes it so that you can refer to the class java.lang.Math by its short name, Math, and say y = Math.sqrt(x). You can import all classes names in package java.lang as follows.

```
import java.lang.*;
```

For static methods such as sqrt, you can even go further, using *static import* directive

```
import static java.lang.Math.sqrt;
```

to allow you to say simply that y = sqrt(x). To import all static methods in the Math class, use

```
import static java.lang.Math.*;
```

You can import a class from a package or a static method from a class, but you cannot import instance methods. For example, class java.lang.System contains a static variable called out. You can do a static import of java.lang.System.out, allowing you to write out.println( ), but you cannot shorten that any further.

An example: Applets

An applet is a small program that is typically run within a browser window, showing some graphics and responding to events such as mouse clicks. One of the standard Java classes is javax.swing.JApplet, which contains methods for setting up a graphics context on which a program can draw, and defines several methods that are typically overridden in subclasses. For example, JApplet has a method called paint that determines just what an applet paints on its canvas, and that is normally overridden. Figure 44.1 shows a simple applet that shows a string on the canvas. Parameter g of the paint method is the canvas, an object of class java.awt.Graphics. Paint needs to get a font for the string, which is available via class java.awt.Font. The drawString method draws a string at a particular x- and y- coordinate of the canvas.

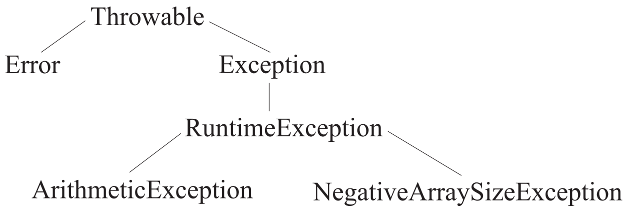
44.10. Exception handling

Exception classes

Java supports exception handling, as described in Chapter 25. Exceptions are objects, each having class, and part of the class hierarchy for exceptions is as follows.

```
package applets.examples;
import java.awt.Font;
import javax.swing.JApplet;
class AppletExample extends JApplet {
  public void paint(java.awt.Graphics g) {
    Font f = new Font("System", Font.BOLD, 18);
    g.setFont(f);
    g.drawString("Good day, sir", 25, 25);
  }
}
```

Figure 44.1: **A Java Applet.** This file would need to be in subdirectory examples of directory applets.



Class Error and its subclasses hold exceptions that are considered serious errors in programs, which should rarely be caught and handled. Class Exception contains exceptions that are typically caught.

Throwing exceptions

An exception is thrown via a **throw** statement, of the form **throw** *e*, where *e* is an exception. Typically, the exception is created using **new**. For example,

```
throw new ArithmeticException();
```

throws ArithmeticException.

Catching exceptions

An exception can be caught by a try statement of the general form

```
try
  {body: statements where exception might be thrown}
catch(Exception-class1 e)
  {exception handler for all exceptions in Exception-class1}
... more catch parts
finally
  {statements that are done after the try}
```

If an exception occurs within the body, the catch parts are tried in sequence. The class hierarchy provides a convenient mechanism to catch entire groups of related exceptions. For example, to catch any kind of RuntimeException, you would use

```
catch(RuntimeException e)
```

The **finally** clause, which can be omitted, allows the programmer to give statements that should be done when the try is finished, regardless of why it finishes. The finally part is done regardless of whether an exception occurred, or whether an exception was caught. Even a return statement, which normally returns from a method immediately, will be followed by the finally part. Typically, the finally part releases or destroys any resources that were allocated in the try statement. By releasing them in the finally part, you are sure that they will be released, regardless of what happens.

Writing methods that might throw exceptions

When a method throws an exception, it performs an action (throwing the exception) that is not indicated in its type. Hidden behaviors are generally undesirable, and the designers of Java wanted to make the possibility of throwing an exception more visible. To that end, a method definition can have one or more **throws** clauses just after its heading, indicating exceptions that the method can throw to its caller. If you say that the method throws RuntimeException, you mean that it might throw an object of class RuntimeException or of any subclass of RuntimeException. For example, a method that pops a value from a stack of strings, but that might throw a StackUnderflow exception, would look like this.

```
public String pop() throws StackUnderflow {
  if(empty()) throw new StackUnderflow();
  else
    ...
}
```

This method throws a StackUnderflow exception directly. A throws clause is needed in the heading regardless of whether the exception is thrown here or in some method that is called by this one. Any potentially uncaught exception must be mentioned.

All methods potentially throw error exceptions, since some kinds of things can always go wrong. (The system might, for example, run out of memory.) It is not necessary to mention that possibility in a throws clause. Only exceptions beneath Exception need to be indicated.

44.11. Abstraction and polymorphism

Chapter 42 discusses virtual methods, which Java calls *abstract methods*. To declare a method abstract, use modifier **abstract**. Any class that contains an abstract method is an abstract class, and should be declared abstract. For example, a class of expressions might begin

```
abstract class Expression {
  public abstract Expression sum(Expression x, Expression y) {
    ...
  }
  ...
}
```

Each subclass of Expression must define method sum, or redeclare it to be abstract.

Interfaces

Like Smalltalk, Java is a single-inheritance language, so a class can only extend one other class. But Java provides a feature, the *interface*, that allows a programmer to do some of what a multiple-inheritance language offers. An interface is a genus, in the sense of Chapter 36. Unlike an abstract class, an interface cannot contain any variables or nonabstract methods; an interface has only abstract methods. For example, the standard interface Runnable is defined as follows.

```
interface Runnable {
  public abstract void run();
}
```

Since all methods in an interface are required to be abstract, the modifier **abstract** can be omitted for them. It is inserted by the compiler.

A class can be put beneath an interface by indicating that it **implements** the interface. A class can implement any number of interfaces. For example, class MyApplet might extend the standard class JApplet, and also implement the interface Runnable.

```
class MyApplet extends JApplet implements Runnable {
  public void run() {
    ...
  }
  ...
}
```

The problems that occur with multiple inheritance, discussed in Chapter 40, come from inheritance of variables method definitions. Since there are no variables or method definitions in an interface, no such problems occur. All that a class inherits from an interface is the *responsibility* to define certain methods. For example, class MyApplet inherits, from interface Runnable, the responsibility to define method run, taking no parameters and producing a void result. Interfaces only control polymorphism, not inheritance. For example, you can put an object of class MyApplet into a variable of type Runnable as follows.

```
Runnable r = new MyApplet();
```

A method that takes a parameter of type Runnable is polymorphic; it works for any class that implements Runnable.

44.12. Event handling and inner classes

Java is designed for use in programs that manage and use graphical user interfaces, among other things, and such a program must handle *events* such as keystrokes and mouse clicks.

Since an operating system recognizes those events, the program must somehow interact with the operating system to handle events appropriately. For example, a particular applet might want to start an animation when the user clicks the mouse on a button labeled “start”.

It is not our purpose to go into details about event handling in Java. That is a beyond the scope of this book. Rather, we will take a brief look at the language mechanism that Java provides in support of event handling, and the motivation for it, since it is an interesting aspect of Java.

Event loops

There are two common ways to handle events. The first is the use of an *event loop*, which repeatedly calls a system method (say, CheckForEvent) to ask whether an event has happened. That method waits for an event and returns when an event has happened, providing information about the event, such as whether it was a keystroke or mouse click. The event loop decides what to do to handle the event, and then goes back and calls CheckForEvent again to get the next event.

The event loop approach has the advantage that the system only needs to provide one method, CheckForEvent. But it has an unpleasant aspect. All event handling tends to be put in one place. After getting an event, the event loop needs to contain code to decide what to do for every possible event that the program will handle, and so it needs to know about all of the events. (The event loop might call a method in some other class, but the call needs to be there for every possible event.) Many different aspects of handling events, related to different classes, need to be combined into a single event loop. When you modify your program, you will often need to modify the event loop as well. That is awkward for modular design, and is precisely the kind of thing that object-oriented programming tries to avoid.

Moving the event-handling code closer to its natural setting

A second approach is to interact with the system or at a more intimate level. Different parts of the program indicate to the system that they want to handle certain kinds of events, and say how those events should be handled. Ideally, you would give the system a method (sometimes called a “callback”) that it should call when the particular event occurs. A request might say “When this object has the keyboard focus, handle a keystroke by calling method handleKeystroke,” where handleKeystroke is some method that belongs to a particular object or class.

Passing a method to the system requires that methods be considered values. But in Java, methods are not values. They are components of objects that cannot be separated from the object. An alternative to passing a method is passing an entire object that contains the method. For example, suppose that object A wants to handle keystroke events. Then A could define an event handler for keystrokes, and pass *itself* to the system to register itself as an object that handles certain kinds of events.

But an event handling object will almost certainly not want to define all aspects of handling events itself. For example, if it wants to handle keystrokes, it should not have to say how to handle mouse clicks as well. That, too, has an obvious solution. The language library can define a general event handling class, such as EventHandler, that contains default event-handling methods. Particular event handling classes can be subclasses of EventHandler, and can inherit all aspects of event handling that they do not want to handle themselves, and override methods as they choose. Overriding only the keystroke handler will leave other event handlers to be inherited.

We are getting closer, but there is still another difficulty. Our event handling class needs to be a subclass of EventHandler. But in a single inheritance language, such as Java, that precludes our event handling class from being a subclass of other classes as well. The effect is to segregate event handling classes from the rest of the program. But that takes us right back to the event loop structure, where event handling has to be grouped together, separate from the rest of the program. What we want is an object that is simultaneously an event handler and something else (indicating the logical nature of the object). An event handler that is responsible for drawing animations might want to be an Animation object as well as an EventHandler object. We seem to require multiple inheritance, which Java does not support.

Inner classes

Java 1.0 had no way to avoid this problem. Event handling was managed using event loops. Java 1.1 and subsequent versions include a concept, called an *inner class*, that improves modular design of event handlers by offering an approximation to multiple inheritance.

In a functional language a function can be defined inside another function. It automatically gets access to the things that were defined locally within the embedding function. Functions are implemented as *function closures*, where a function is given access to the environment in which it was created.

An inner class is similar to a function closure. But instead of defining a function inside another function, you create an object inside another object. An object B of class Bear can contain another object E of class BearEventHandler. Object E gets full access to the private things in object B, via an environment pointer similar to the environment in a function closure. However, E's class does not need to have any relationship to class Bear. For example, BearEventHandler will not, in general, be a subclass of Bear. Instead, it will be a subclass of EventHandler. It is as if a class can be broken into parts, where one part is a subclass of Bear and another part is a subclass of EventHandler.

To illustrate inner classes, let's take a simpler situation than an event handler. We will create an interface, IntegerProperty, with abstract methods that get and set a particular integer variable of an object. We will then create a class that does not implement the IntegerProperty interface, but that contains an inner class that does implement IntegerProperty. Objects of the inner class will modify a variable of the object that embeds them. So the object that does not implement IntegerProperty looks almost as if it does. The IntegerProperty interface is as follows.

```
interface IntegerProperty {
  public int get();        // return the value of the property
  public void set(int x);  // set the property to x
}
```

Figure 44.2 show class MyClass containing inner class MyClassProperty. Notice that class MyClassProperty is private. That makes all of its features, including its constructor, only accessible within class MyClass. Also notice method makePropertyObject in class MyClass, which creates an IntegerProperty object (actually of class MyClassProperty) that manipulates the characteristic variable of the MyClass object. The following creates a MyClass

```
class MyClass {
  private class MyClassProperty implements IntegerProperty {
    public MyClassProperty() {}
    public int get() {
      return characteristic;
    }
    public void set(int x) {
      characteristic = x;
    }
  }

  // Constructor for the IntegerProperty object

  public IntegerProperty makePropertyObject() {
    return new MyClassProperty();
  }

  // Rest of class MyClass, including the characteristic variable

  private int characteristic;

  public MyClass (int n) {
    characteristic = n;
  }

  ...
}
```

Figure 44.2: **Class MyClass with inner class MyClassProperty.**

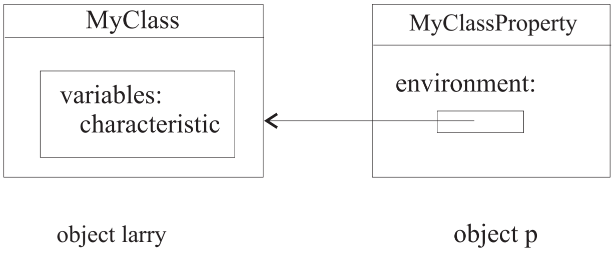


Figure 44.3: **Object larry, of class MyClass, and object p, of class MyClassProperty.** Since MyClassProperty is embedded inside MyClass, object p contains an implicit environment pointer to the object that created it. That allows p to have access to larry's private variables.

object called larry and an associated IntegerProperty object that is tied, through closure, to larry.

```
MyClass larry = new MyClass(0);
IntegerProperty p = larry.makePropertyObject();
p.set(5);
```

Figure 44.3 shows the relationships between objects larry and p. Statement p.set(5) sets larry's characteristic variable to 5.

More general embedding of classes

Once you have decided to allow classes to be defined inside other classes, it seems natural to allow classes to be defined anywhere. Java does just that. For example, a class C can be defined inside a method M. Then only method M can create members of class C, since C is only visible inside method M. Since class C is defined in the context of method M, you would expect an object of class C to be able to use the local variables of method M. In fact, Java only allows objects of class C to use the local *constants* of M, that is, variables that have been marked final. The values of those constants are copied into the object when it is created so that it makes sense for the object to exist after the method that created it has returned.

44.13. Reflection

Like Smalltalk, Java allows a running program to examine its own classes and methods on the fly. An object of Java class **Class** represents a class, as an object. One way to build a Class object is to use static method forName, which produces a class from its full name. For example,

```
Class objclass = Class.forName("java.lang.Object");
```

gets an object that represents class Object. You can ask a class object its name (method getName( )), its base class (method getSuperclass( ), which returns null for class Object) or a variety of other things. The getMethods( ) method returns an array of all of the public methods of a class. For example,

```
Class objclass = Class.forName("java.lang.Object");
java.lang.reflect.Method[] objmethods = objclass.getMethods();
```

gives all of the methods of class Object, in an array. The Method class has methods including getName( ); getReturnType( ), which returns a Class object; getParameterTypes( ), which returns an array of Class objects telling the types of the parameters of this method; and invoke(obj, params), which runs the method. For example, if meth is an object of class Method, then meth.invoke(x, p) behaves like x.meth(p), where x is the object that performs the method and array p contains the parameters.

Reflection allows a program to modify the class hierarchy on-the-fly by loading classes, examining them and using them.

44.14. Generics

Section 37.7 describes the idea of generics, allowing parameterized type, module and function definitions. With version 1.5, Java includes support for generics. The support is extensive, and we will not attempt to cover all aspects. Instead, we will give an idea of what it can do, and some of the pitfalls inherent in grafting a rich generic structure onto a language and language implementation that were not originally designed for it.

An example

Before generics were added to Java, you would create a general linked list class by making each thing in the linked list have type Object. That works fine for adding a value to the linked list, since you can pass a parameter of any class in a place where type Object is required. But suppose that you have created a linked list of called L that you know only contains integers, because that is all you have added to it. To get the head of L, you employ the head method.

```
Integer h = L.head();
```

But that does not work because L.head() has type Object, and cannot be stored in a variable of type Integer. To get the head, you would have to do an explicit down-conversion,

```
Integer h = (Integer) L.head();
```

requesting a run-time type check. If you have made a mistake, that mistake will only show up when it occurs while the program runs.

Generics come to the rescue, allowing the compiler to perform stronger compile-time type checking. We will implement a simple generic linked list type, LinkedList<T>, where an object of type LinkedList<Integer>, for example, is a list of integers. As a preliminary step, Figure 44.4 defines a type of linked list cells, each holding a value and a (pointer to) another linked list cell. The next cell will be **null** to indicate the end of the list. Figure 44.5 shows a definition of generic class LinkedList<T>. It provides methods head, tail and isEmpty to examine a list, method setHead to modify the head, and constructors to build an empty list, and to build a list with a given head and tail. The machinery required to

```
class ListCell<T> {
  private ListCell<T> next;
  private T item;
  public ListCell(T anItem, ListCell<T> theNext) {
    item = anItem;
    next = theNext;
  }
  public T getItem() {
    return item;
  }
  public void setItem(T anItem) {
    item = anItem;
  }
  public ListCell<T> getNext() {
    return next;
  }
}
```

Figure 44.4: Class ListCell<*T*>**.**

```
class LinkedList<T> {
  private ListCell<T> firstCell;
  public LinkedList() {
    firstCell = null;
  }
  public LinkedList(T anItem, LinkedList<T> theNext) {
    firstCell = new ListCell<T>(anItem, theNext.firstCell);
  }
  private LinkedList(ListCell<T> cell) {
    firstCell = cell;
  }
  public boolean isEmpty() {
    return firstCell == null;
  }
  public T head() {
    return firstCell.getItem();
  }
  public LinkedList<T> tail() {
    return new LinkedList<T>(firstCell.getNext());
  }
  public void setHead(T x) {
    firstCell.setItem(x);
  }
}
```

Figure 44.5: Class LinkedList<T>.

make the class generic appears to be minimal; you just attach a parameter <T> to the class and use T inside the class definition as the list element type.

The Java compiler is aware of generics, and performs type checking with parameters taken into account. For example, you cannot put an object of type LinkedList<Integer> into a variable of type LinkedList<String>. It also knows that the head of an object of type LinkedList<Integer> has type Integer; no down-conversion is required.

Generic classes are invariant, in the sense of Chapter 36, so you cannot store an object of class LinkedList<Mammal> into a variable of type LinkedList<Animal>, even when Mammal is a subclass of Animal.

Difficulties with generics

But generics are an add-on, and the interpreter is unaware of them. As far as the interpreter knows, there is no such thing as LinkedList<String>; there is only a class LinkedList, essentially the same as it would have been before generics were introduced. Consequently, you cannot make use of the parameter T in ways that would require the interpreter to know what it is. Inside class LinkedList<T>, for example, you are not allowed to create a new object of class T or $T[\,]$ using **new**, because to do that, the interpreter would need to know T.

Suppose, for example, that you want to add another method to class LinkedList<T> that returns an array holding the members of the linked list. That method needs to build the array, which it cannot do because it does not know T. So a method L.toArray() simply cannnot be written. There are options for conversion to an array. For example, you can define method L.toArray(T), where you explicitly pass class T (as an object) to the method so that it can create the necessary array, employing Java's reflection features. But if you make a mistake and pass the wrong class, then your program contains a serious type error that the compiler cannot recognize. Remember that the whole purpose of generics is to get the compiler to catch precisely this kind of error at compile time.

Since the interpreter is unaware of the parameters, run-time type checks have to be relaxed. The definition of class LinkedList<T> actually creates class LinkedList; the compiler allows you to use LinkedList directly (with warnings), and an object of class LinkedList<T> can be stored in a variable of type LinkedList. Consider the following lines.

```
LinkedList<String> A = new LinkedList<String>();
A = new LinkedList<String>("frog", A);
LinkedList B = A;
LinkedList<Integer> C = B;
C.setHead(3);
```

Variable B appears to have forgotten some information; it only remembers that it holds a linked list, not that it holds a linked list of strings. The conversion implicit in C = B really should fail at run time. But the interpreter has no way of checking it, and allows it to succeed. The effect of the last line, C.setHead(3), is that list A, nominally a list of strings, ends up holding an integer. That line does not throw an exception because the interpreter does not have enough information to allow it to know that something is wrong. Java's generic system is consequently type-unsafe, if you think it unreasonable that an integer can be put into a list of strings.

44.15. Summary of terminology and concepts

Java is a single-inheritance object-oriented programming language. It is designed to use C++ syntax for most things, to make it easy for programmers familiar with C++ to learn Java. But Java behind the syntax Java is much closer in spirit to Smalltalk than to C++.

Java provides most of the standard ideas of object-oriented programming, including inheritance and virtual (abstract) methods. Additionally, it provides the ideas of interfaces and inner classes, which provide some of the benefits of multiple inheritance.

Java has a very large and growing library. Part of the library is called the Java Application Program Interface. (It changes each time a new version of Java is defined.) The rest of the library is distributed around the world as classes provided by individuals and organizations.

Web browsers can interact with a Java interpreter so that the browser can run some Java programs. The browser runs *applets*, which are java classes that are subclasses of the standard Java class JApplet. They typically draw on the screen, and can be used for animations or other computations. The JApplet class provides everything that the browser needs to use, but will only do something useful of some of its methods are overridden.

Terminology

Abstract Virtual methods and abstract classes are labeled *abstract* in Java.

Applet An applet is a Java class that can be run by a web browser.

Clone Java uses the word *clone* to indicate the operation of making a copy of an object.

Embedded class An inner class is a class that is embedded inside another class. That helps to simulate, in a single-inheritance language, some of what a multiple-inheritance language would allow. Classes can also be embedded in methods and initializers, where they are called embedded classes.

Final Something labeled final cannot be changed. A final variable is a constant. A final method cannot be overridden in a subclass. A final class cannot be extended, and so cannot have any subclasses.

Interface An interface is similar to an abstract class, but it can only contain abstract methods. A class can implement many interfaces. This is another feature that allows Java, a single-inheritance language, to achieve some of the benefits of multiple inheritance without the difficulties that come with it.

New The new operator is used to create a new object. All objects are created with this operator. Using **new** calls for running a constructor.

Null Every class allows a null pseudo-object.

Package A package is a collection of related classes. The library is a collection of packages.

Private A private method or variable is only accessible within the class where it is defined.

Protected A protected method or variable is only accessible within the class where it is defined or any of that class's subclasses, or by other classes in the same package.

Public A public method or variable is accessible to all classes.

Static A static variable or method is associated with the class itself, not with instances of the class.

Super The word **super** stands for the current (agent) object, but it uses the methods that would be used as if this object were in the base class of the current class. This allows you to use methods from the base class that have been overridden, as well as constructors in the base class.

This The word **this** indicates the current (agent) object, just as the word **self** represents the current object in Smalltalk.

Unicode Unicode is an international character encoding standard that has codes for characters of many different alphabets. Java characters are unicode, and Java programs are written in unicode.

44.16. Exercises

44.1. Why did the designers of Java use C++ syntax if they had no intention of making Java compatible with C++?

44.2. Write a Java class with a method that uses a local variable without storing any value into it. What happens?

44.3. Java requires that a compiler be able to check that each local variable in a method is initialized before it is used. Give an example of a method definition where a variable is initialized before it is used, but the compiler cannot detect that. (**Hint.** Use two if-statements. The compiler cannot know which branch of an if-statement will be taken, so it presumes that all paths through successive if-statements are possible.)

44.4. What was the motivation for adding inner classes?

44.5. What word indicates a class method in Java?

44.6. Java requires every method that might throw an exception to say so in its heading. Would it be reasonable for Java to make it optional to indicate exceptions, for those programmers who prefer to do that? Why or why not?

44.7. (See the footnote on page 464). Explain how an interned string in Java is like a symbol in Scheme.

44.8. The null pseudo-object cannot be asked to perform a method. (If you try, an exception is thrown.) Suppose that you decide to modify Java so that a null object can perform a method.

(a) In current implementations of Java, objects are represented by pointers and the null object is represented by a null pointer. Would allowing a null object to perform a method require a change to the way the null object is represented?

(b) Can you suggest a modification to the language that would make it possible to say how a null object behaves?

(c) Do you think these modifications would improve Java or make it worse?

44.9. Write an implementation of a Queue class similar to the one shown in Figure 43.1 in Java. For this question, do not make the Queue class generic, but make a queue hold objects of class Object.

44.10. Redo the previous question, but now make the Queue class generic. What needs to change?

44.11. Suppose that class Bird has two subclasses, Duck and Chicken. You create an array D of type Duck[ ]. According to the rules of Java, you can store this array into a variable B of type Bird[ ]. Now what happens if you try to store a Chicken into $B[0]$?

(a) Should that be allowed? Why or why not?

(b) Does Java allow it? Explain.

44.12. Java arrays are covariant.

(a) Give an example where it is convenient for arrays to be covariant.

(b) Give an argument that, from a type checking standpoint, arrays should probably be invariant. What is a consequence for type checking of making them covariant?

44.13. Can you think of any reason for making LinkedList$< T >$ invariant but $T[]$ covariant, as the definition of Java does?

44.17. Bibliographic Notes

Arnold and Gosling [7] describe the Java language, and Gosling, Joy and Steele [44] give a detailed definition of Java. Lindholm and Yellin [70] describe the Java abstract machine. Horstmann and Cornell [57, 58] describe the Java libraries. A great deal of material on Java can be found at www.java.sun.com.

Chapter 45

Object-Oriented Programming in Cinnameg

45.1. Introduction

This chapter shows how some of the ideas of object-oriented programming are provided in Cinnameg. No attempt is made to cover all issues or all of what Cinnameg offers.

Cinnameg does not separate its object-oriented component from the rest of the language, but offers a close integration among different parts, and some ideas concerning objects can be viewed in either a typically object-oriented manner or in a way that has little to do with object-oriented programming. That allows you to adopt a viewpoint in each part of a program that you find simplest and most appealing for that part.

45.2. Objects

You ask an object x to perform method $m(u, v)$ by writing x's $m(u, v)$. In general, any time you want to select a part of x called p, write x's p. You are allowed to select a method from an object without running it if you want to treat it as a value.

45.3. Creating a class

A class lists components that its instances possess. A class is broken into two parts: the interface part, telling what the public components are, without implementing them; and the implementation part, where private data components and implementations of the methods and constructors are written. Figure 45.1 shows the form of a class interface.

A class is not considered a module in Cinnameg. Instead, you embed the class in a package and let the package play the role of the module of the class. Normally, you write the class interface in the export part of the module and the class representation and implementation in the implementation part of the module. Figure 45.2 shows a package holding a class.

```
Class classname
    extends superclass-name
    interface
  instanceCapabilities:
    expectations of instance methods
  constructors:
    expectations of constructors
  instanceInformation:
    data declarations
%Class
```

Figure 45.1: **The form of a class interface.** Each expectation has the form `Expect` *name*: *type* `%Expect`. The instanceInformation part is in a class interface is only intended for public data components. The parts (instanceCapabilities, constructors and instanceInformation) can in any order. If the **extends** phrase is omitted, the base class is AnyObject.

Data components

Following **instanceInformation:**, write a list of item and variable declarations. Declaration

`Item` *name*: *type*.

says that the object has a component with the given name and type. Use it for adding components that are installed when an object is created and remain fixed during the lifetime of the object. Declaration

`Variable` *name*: *type*.

says that objects of this class contain a variable with the given name and type. Expression obj's v is the (current) value of data component v on object obj. To change the value of variable v in object obj, use statement `Make obj's` v `=!` e `%Make`, where expression e computes the new value.

Defining methods and constructors

The class implementation part, identified by the word *implementation*, contains method and constructor definitions. Each method is defined as a component an object called self, so method m is called self's m. Figure 45.2 shows examples. A constructor definition has the form

```
Define constructor f(params)
  beforeCreation:
    definition of items
  afterCreation:
    initialization of variables and other actions
%Define
```

where the part after **beforeCreation:** defines the items (constants) that are part of the object and the part after **afterCreation:** gives values to the object's variables. Call the object self in both parts. (Technically, the object has not yet been created inside the **beforeCreation** part because it cannot be created until its constants are known. Nevertheless, you refer to the constant called *c* as self's *c* as a way of noting that it will belong to object self when self is created.) Figure 45.2 shows an example.

You typically add additional data components in the class implementation part, introducing them with the word **instanceInformation**, as in the class interface. There is an important restriction, though. In the class implementation, there can only be one **instanceInformation** part, and it must come first.

45.4. Examples

Figure 45.2 shows class PRNG, whose instances are pseudo-random number generators.[1] You create a PRNG object *r* using constructor newPRNG(*s*,*n*), where *s* is a seed and *n* tells *r* to behave like an *n*-sided die with faces labeled 1 to *n*. Expression (*r*'s nextReal) yields a pseudo-random real number between 0 and 1, and (*r*'s nextInt) yields a new pseudo-random integer between 1 and (*r*'s numSides). As a simple example, the following just simulates tossing a die 100 times and shows the sum of all of the results.

```
Let r = newPRNG(6).
Let sum = 0.
For ? from 1 to 100 do
  Relet sum = sum + r's nextInt.
%For
Displayln "The sum is ".
Displayln sum.
```

Appendix C contains an object-oriented implementation of a program that plays the game of Nim. Each strategy for a player is implemented as a different class. That way, you can play a game between any two players, with any strategies that have been implemented. The method where a given strategy chooses its next move is virtual, since it needs to be written in a different way in each subclass.

45.5. Summary

Although Cinnameg is not a paradigmatic object-oriented language, it allows you to write programs in an object-oriented style, and to meld that style with others, allowing you to take advantage of what each approach has to offer.

Cinnameg separates concepts that are normally all combined into the notion of a class. For example, there is only one notion of a module; a class cannot be a module, but is normally embedded in a package, which handles issues related to modules.

[1] A pseudo-random number generator provides a sequence of values that give the appearance of being chosen at random. It starts with a *seed* value and has a function to update that value to the next one. For a particular seed *s*, the sequence *s*, next(*s*), next(next(*s*)), . . . is always the same; it is just intended to *appear* jumbled up.

```
Package PRNG
export
  Class PRNG interface
    instanceCapabilities:
      Expect
        nextInt   : Integer;
        nextReal  : Real
      %Expect
    instanceInformation:
      Item numSides  : Integer.
    constructors:
      Expect newPRNG: (Integer, Integer) -> PRNG.
  %Class
===============================================
implementation %% of the package
  Let a = 16807.
  Let m = 2147483647.
  Class PRNG implementation
    instanceInformation:
      Variable currentVal: Integer.
    constructors:
      Define constructor newPRNG(seed, n)
        beforeCreation:
          Let self's numSides = n.
        afterCreation:
          Make self's currentVal =! seed.
      %Define
    instanceCapabilities:
      Define
        self's next = self's currentVal |
          Make self's currentVal =! (a * self's currentVal) `mod` m.;
        self's nextReal  = real(self's next) / real(m);
        self's nextInt   =
          1 + floor(self's nextReal * real(self's numSides))
      %Define
    %Class
  %Package
```

Figure 45.2: **A pseudo-random number generator class, embedded in a package.** Expression real(n) is the real number equivalent to integer n, and floor(x) is the largest integer that is not larger than real number x. Notice that a and m are constants external to the class.

45.6. Bibliographic notes

The random number generation algorithm used in class PRNG is suggested by Park and Miller [85].

Part XV

Topics in Imperative Programming

Chapter 46

Semantics of Imperative Programming

46.1. Introduction

Semantic rules for an imperative programming language tell what each statement does, and how the values of expressions are determined. The semantic rules can be important because some aspects of imperative computation are subtle, and need to be understood carefully.

Truly careful semantic descriptions of entire programming languages tend to be large and to take time to understand, and languages whose descriptions are too mathematical tend to fair poorly because few programmers are willing to wade through that level of detail. The report on the programming language Algol 68, for example, employs a lengthy and mathematical semantic description. Algol 68 is a large and complex language, and the fact that its report was so difficult to read was one factor contributing to Algol 68 not enjoying as widespread use as its predecessor, Algol 60. But careful semantic rules, particularly for portions of languages, can help programmers to understand those features, and can be especially useful to programming language implementors, who need to know exactly how implementations are required to behave.

This chapter examines a way of describing the semantics of imperative languages. A careful description of semantics is necessarily detailed and mathematical. You derive the semantics of a program by proving a fact, and parts of this chapter employ concepts from mathematical logic and proofs.

46.2. Operational semantics

States

The *state* of a computer at a moment in time tells the entire content of the computer's memory and everything else associated with the computer, such as the contents of files. (A full description of the state for an interactive program also needs to describe the state of mind of the program's user, so that you know what the user will do; otherwise you cannot say what the program will do. But to avoid that complication, we will restrict attention to programs that do not interact with a human.)

Atomic steps

A program runs by performing a sequence of fundamental, or *atomic*, steps, each step making a small change to the state. For example, a step might consist of adding two numbers and storing the result into a variable. More complex statements can be performed by doing a sequence of atomic steps. For example, assignment statement $w := x + y + z$ needs to add x and y, then add that total to z before changing w.

Part of a semantics of an imperative language is identification of the atomic steps that a program can make. Any mathematical semantics is only an approximation of what really happens in a given computer. One semantic description of a language might say that assignment statement $k := k + 1$ is an atomic action, taking just one step to perform. Another semantics of the same language might offer more detail, and say that statement $k := k + 1$ requires fetching k into a register, adding 1 to that register, and then storing the register into k, for a total of three steps. Both take you to the same state at the end, but one takes a longer route there.

There is no obvious "right" choice of atomic steps. It would not be a good idea, for example, to model the semantics on the exact actions of one brand of processor, since then the language is inherently tied to that one processor. Our approach will be to choose actions that make the semantics easy to describe. Any set of atomic actions that, at a minimum, gives the correct final state of a program can at least be used to determine the answer that a program produces, and offers at least some idea of the steps that got it there.

Sequences of states

A program performs a sequence of atomic steps. But you can describe a sequence of actions implicitly by giving the states that result from the actions. For example, if the program starts in state s_1 and then moves to state s_2, you know that it performed some action that would change s_1 into s_2, and it is not necessary to say just what the action is. A semantics that is based on sequences of states is called an *operational semantics.*

46.3. A notation for states

For simplicity, we will only model variables, not files and other things connected to a computer. So a state tells the current values of all of a program's variables, or at least of those that have values. We assume that each variable has a name. We write variable values in braces; $\{x = 4, y = 7\}$ indicates that x has value 4 and y has value 7. A collection of variable bindings in braces is called a *simple state.*

An assignment statement changes the state. For example, performing assignment $x := x + y$, starting in simple state $\{x = 4, y = 7\}$, yields simple state $\{x = 11, y = 7\}$ (possibly after a few intermediate states).

If S is a simple state, write $\theta_S(x)$ for the value of variable x in simple state S. For example suppose that state W is $\{x = 4, y = 9\}$. Then $\theta_W(x) = 4$ and $\theta_W(y) = 9$.

Write $S[x := v]$ for the simple state that is obtained from S by either adding a new binding $x = v$ (if S has no binding for x) or by replacing the binding of x by v (if S has a binding for x). For example, if simple state W is $\{x = 4, y = 9\}$, then $W[x := 33]$ is simple state $\{x = 33, y = 9\}$, and $W[z := 5]$ is $\{x = 4, y = 9, z = 5\}$.

Full states

Simple states are a start, but they are not enough for describing semantics. For each state, an operational semantics needs to say which state comes next in a computation. But that depends on the program itself, and on where the program's thread of control is currently pointing, so that you know which statement the program will perform next. So the program needs to be part of the state.

A *full state* has the form $\langle P \parallel S\rangle$, where P is a program and S is a simple state. It is assumed that P starts with its first statement. For example, full state $\langle y := 3;\ z := 1;\ \parallel \{x = 0, y = 1\}\rangle$ indicates that the program has already reached a state where $x = 0$ and $y = 1$, and it is about to continue with statement $y := 3$, which will in turn be followed by statement $z := 1$.

Our semantics needs to indicate how expressions are evaluated, so a full state can contain an expression instead of a statement. For example, full state $\langle x + 5 \parallel \{x = 1\}\rangle$ indicates that expression $x + 5$ is being computed, and that x has value 1. The next full state could be $\langle 1 + 5 \parallel \{x = 1\}\rangle$, where the value of x has been fetched and substituted into the expression. Assuming that addition of integers is an atomic step, the next full state would be $\langle 6 \parallel \{x = 1\}\rangle$, where the expression is just a number.

46.4. The single-step relation

If F and G are full states, then we write $F \to G$ to indicate that G immediately follows F in a computation. That is, a single atomic step has been performed, converting full state F to full state G. For example, $\langle y := 3;\ z := 1;\ \parallel \{x = 0, y = 1\}\rangle \to \langle z := 1;\ \parallel \{x = 0, y = 3\}\rangle$. Notice that statement $y := 3$, having been performed, is removed, making $z := 1$ the next statement to be done.

The single-step relation also applies to evaluation of expressions. For example, $\langle x + 5 \parallel \{x = 1\}\rangle \to \langle 1 + 5 \parallel \{x = 1\}\rangle$ shows one step in evaluation of expression $x+5$, moving it closer to its final value. Some changes in a program do not remove a statement, but modify an expression within the statement. For example, assuming additions are performed from left to right,

$$\begin{array}{lllll} & \langle x := 3 + 5 + y; z := 41; & \parallel & \{y = 2\}\rangle \\ \to & \langle x := 8 + y; z := 41; & \parallel & \{y = 2\}\rangle \\ \to & \langle x := 8 + 2; z := 41; & \parallel & \{y = 2\}\rangle \\ \to & \langle x := 10; z := 41; & \parallel & \{y = 2\}\rangle \end{array}$$

and evaluation continues from there. A sequence of full states $s_1 \to s_2 \to s_3 \to \cdots \to s_n$ is called an *evaluation sequence*. Figure 46.1 shows two short evaluation sequences, and Figure 46.2 shows a longer example involving a while loop.

46.5. A notation for describing an operational semantics

An operational semantics needs to define the single-step relation $\to$, where $S \to T$ indicates that full state S is immediately followed by full state T during computation. One way to define the $\to$ relation is via *rules of inference*, allowing you to draw conclusions, possibly based on other conclusions that you have already made. If Γ is a list of relationships and A

$$\begin{array}{lll}
 & \langle x := 2; y := x+6; & \| \quad \{\}\rangle \\
\to & \langle y := x+6; & \| \quad \{x=2\}\rangle \\
\to & \langle y := 2+6; & \| \quad \{x=2\}\rangle \\
\to & \langle y := 8; & \| \quad \{x=2\}\rangle \\
\to & \langle & \| \quad \{x=2, y=8\}\rangle
\end{array}$$

(a)

$$\begin{array}{lll}
 & \langle x := 2;\ \textbf{if}\ x > 4\ \textbf{then}\ y := x;\ \textbf{else}\ y := 0;\ \textbf{endif} & \| \ \{\}\rangle \\
\to & \langle \textbf{if}\ x > 4\ \textbf{then}\ y := x;\ \textbf{else}\ y := 0;\ \textbf{endif} & \| \ \{x=2\}\rangle \\
\to & \langle \textbf{if}\ 2 > 4\ \textbf{then}\ y := x;\ \textbf{else}\ y := 0;\ \textbf{endif} & \| \ \{x=2\}\rangle \\
\to & \langle \textbf{if}\ \text{false}\ \textbf{then}\ y := x;\ \textbf{else}\ y := 0;\ \textbf{endif} & \| \ \{x=2\}\rangle \\
\to & \langle y := 0; & \| \ \{x=2\}\rangle \\
\to & \langle & \| \ \{x=2, y=0\}\rangle
\end{array}$$

(b)

Figure 46.1: **Two evaluation sequences.** Each line represents one small step in evaluating a program. State $\langle \text{P} \parallel S\rangle$ indicates that we are about to run program P, where the current variable values are given by simple state S. Computation stops when the program is empty. These evaluation sequences follow the general rules in Figures 46.3 and 46.4.

is a relationship, then we write

$$\frac{\Gamma}{A}$$

to indicate that relationship A is true whenever all of the relationships in list Γ are true. For example, inference rule

$$\frac{\langle \alpha \parallel S\rangle \to \langle \beta \parallel T\rangle}{\langle v\ :=\ \alpha;\ L \parallel S\rangle \to \langle v\ :=\ \beta;\ L \parallel T\rangle} \tag{46.5.1}$$

tells a property of assignment statements, of the form $v := \alpha$. In this rule, v stands for an arbitrary variable, α and β stand for expressions, and L is any sequence of statements. Rule (46.5.1) says that, if you have already concluded that performing one step in the evaluation of expression α (in simple state S) yields expression β (and, possibly due to side-effects of the evaluation, new simple state T), then you can perform one step in the evaluation of assignment statement $v := \alpha$ (also in simple state S) by changing this assignment statement to $v := \beta$ (and changing the simple state to T). For example, suppose that you already know that $\langle 1+4 \parallel S\rangle \to \langle 5 \parallel S\rangle$ for any simple state S. Then you can use Rule (46.5.1) to infer $\langle x\ :=\ 1+4;\ y\ :=\ 9; \parallel \{z=2\}\rangle \to \langle x\ :=\ 5;\ y\ :=\ 9; \parallel \{z=2\}\rangle$.

Some rules do not require that you already know anything. We write

$$\frac{}{A}$$

to indicate that A is always true, and does not depend on any previously shown facts. For

$$
\begin{array}{lll}
 & \langle x := 0;\ \mathbf{while}\ x < 2\ \mathbf{do}\ x := x + 1;\ \mathbf{endwhile} & \parallel \{\}\rangle \\
\rightarrow & \langle \mathbf{while}\ x < 2\ \mathbf{do}\ x := x + 1;\ \mathbf{endwhile} & \parallel \{x = 0\}\rangle \\
\rightarrow & \langle \mathbf{if}\ x < 2\ \mathbf{then}\ x := x + 1;\ \mathbf{while}\ x < 2\ \mathbf{do}\ x := x + 1;\ \mathbf{endwhile}\ \mathbf{endif} & \parallel \{x = 0\}\rangle \\
\rightarrow & \langle \mathbf{if}\ 0 < 2\ \mathbf{then}\ x := x + 1;\ \mathbf{while}\ x < 2\ \mathbf{do}\ x := x + 1;\ \mathbf{endwhile}\ \mathbf{endif} & \parallel \{x = 0\}\rangle \\
\rightarrow & \langle \mathbf{if}\ \mathrm{true}\ \mathbf{then}\ x := x + 1;\ \mathbf{while}\ x < 2\ \mathbf{do}\ x := x + 1;\ \mathbf{endwhile}\ \mathbf{endif} & \parallel \{x = 0\}\rangle \\
\rightarrow & \langle x := x + 1;\ \mathbf{while}\ x < 2\ \mathbf{do}\ x := x + 1;\ \mathbf{endwhile} & \parallel \{x = 0\}\rangle \\
\rightarrow & \langle x := 0 + 1;\ \mathbf{while}\ x < 2\ \mathbf{do}\ x := x + 1;\ \mathbf{endwhile} & \parallel \{x = 0\}\rangle \\
\rightarrow & \langle x := 1;\ \mathbf{while}\ x < 2\ \mathbf{do}\ x := x + 1;\ \mathbf{endwhile} & \parallel \{x = 0\}\rangle \\
\rightarrow & \langle \mathbf{while}\ x < 2\ \mathbf{do}\ x := x + 1;\ \mathbf{endwhile} & \parallel \{x = 1\}\rangle \\
\rightarrow & \langle \mathbf{if}\ x < 2\ \mathbf{then}\ x := x + 1;\ \mathbf{while}\ x < 2\ \mathbf{do}\ x := x + 1;\ \mathbf{endwhile}\ \mathbf{endif} & \parallel \{x = 1\}\rangle \\
\rightarrow & \langle \mathbf{if}\ 1 < 2\ \mathbf{then}\ x := x + 1;\ \mathbf{while}\ x < 2\ \mathbf{do}\ x := x + 1;\ \mathbf{endwhile}\ \mathbf{endif} & \parallel \{x = 1\}\rangle \\
\rightarrow & \langle \mathbf{if}\ \mathrm{true}\ \mathbf{then}\ x := x + 1;\ \mathbf{while}\ x < 2\ \mathbf{do}\ x := x + 1;\ \mathbf{endwhile}\ \mathbf{endif} & \parallel \{x = 1\}\rangle \\
\rightarrow & \langle x := x + 1;\ \mathbf{while}\ x < 2\ \mathbf{do}\ x := x + 1;\ \mathbf{endwhile} & \parallel \{x = 1\}\rangle \\
\rightarrow & \langle x := 1 + 1;\ \mathbf{while}\ x < 2\ \mathbf{do}\ x := x + 1;\ \mathbf{endwhile} & \parallel \{x = 1\}\rangle \\
\rightarrow & \langle x := 2;\ \mathbf{while}\ x < 2\ \mathbf{do}\ x := x + 1;\ \mathbf{endwhile} & \parallel \{x = 1\}\rangle \\
\rightarrow & \langle \mathbf{while}\ x < 2\ \mathbf{do}\ x := x + 1;\ \mathbf{endwhile} & \parallel \{x = 2\}\rangle \\
\rightarrow & \langle \mathbf{if}\ x < 2\ \mathbf{then}\ x := x + 1;\ \mathbf{while}\ x < 2\ \mathbf{do}\ x := x + 1;\ \mathbf{endwhile}\ \mathbf{endif} & \parallel \{x = 2\}\rangle \\
\rightarrow & \langle \mathbf{if}\ 2 < 2\ \mathbf{then}\ x := x + 1;\ \mathbf{while}\ x < 2\ \mathbf{do}\ x := x + 1;\ \mathbf{endwhile}\ \mathbf{endif} & \parallel \{x = 2\}\rangle \\
\rightarrow & \langle \mathbf{if}\ \mathrm{false}\ \mathbf{then}\ x := x + 1;\ \mathbf{while}\ x < 2\ \mathbf{do}\ x := x + 1;\ \mathbf{endwhile}\ \mathbf{endif} & \parallel \{x = 2\}\rangle \\
\rightarrow & \langle & \parallel \{x = 2\}\rangle
\end{array}
$$

Figure 46.2: **Evaluation sequence involving a loop.** This figure shows a longer example of an evaluation sequence for program fragment

```
x := 0;
while x < 2 do
   x := x + 1;
endwhile
```

starting in an empty simple state. As you can see, the semantics provides a great deal of detail about exactly what happens during computation. The iteration expressed by the while-loop is modeled by copying the body of the loop in front of the loop, and wrapping it all inside an if-statement, using the fact that statement **while** B **do** S **endwhile** is equivalent to

```
if B then
   S
   while B do S endwhile
endif
```

This evaluation sequence follows the general rules in Figures 46.3 and 46.4.

example, a basic fact for variables (as a simple form of expression) is

$$\frac{}{\langle v \parallel S\rangle \to \langle \theta_S(v) \parallel S\rangle} \tag{46.5.2}$$

where v is a variable. This says that evaluation of a variable is performed by getting the value of the variable out of the simple state. For example, relationship $\langle x \parallel \{x = 20\}\rangle \to \langle 20 \parallel \{x = 20\}\rangle$ follows from rule (46.5.2). Inference rule

$$\frac{}{\langle v\ :=\ c;\ L \parallel S\rangle \to \langle L \parallel S[v := c]\rangle} \tag{46.5.3}$$

tells how to perform an assignment statement once the right-hand side has been simplified to a constant c. You simply add binding $v = c$ to the state and remove the assignment statement from the program. Rule (46.5.3) tells you, for example, that relationship ⟨x := 2; y := 4; $\parallel \{z = 0\}\rangle \to \langle$y := 4; $\parallel \{x = 2, z = 0\}\rangle$ is true.

46.6. An example operational semantics

Figures 46.3 and 46.4 show an operational semantics for a very small programming language. Expressions include constants and variables, and allow the use of binary operators such as + and <. For example, $x < 5$ is an expression, indicating a test that produces either **true** or **false** as its result. Any collection of constants and binary operators can be handled by this semantics, as long as the concept of the value of an expression found in rule (op 1) is already defined. In general, value(a op b) is assumed to tell the value of expression a op b, where a and b are constants. For example value(4 + 11) = 15. (A careful definition of value(a op b) for each operator op is part of the semantic definition, but we will not go into that here.)

Statements are (1) assignments of the form v **:=** e**;**, (2) conditionals of the form **if** e **then** S_1 **else** S_2 **endif** and **if** e **then** S_1 **endif**, and (3) loops of the form **while** e **do** S_1 **endwhile**. In the conditional and loop statements, each of S_1 and S_2 can be a sequence of statements.

You should be able to see the rules in Figures 46.3 and 46.4 at work in Figure 46.2. Figure 46.5 shows a proof of relationship $\langle y := x + 1;\ \parallel \{x = 8\}\rangle \to \langle y := 8 + 1;\ \parallel \{x = 8\}\rangle$ using those inference rules.

Notice that it can takes several *inference* steps to derive one fact of the form $s \to t$. The inference steps are not performed by the program, and are not part of the computation. It only takes one atomic step, in our semantics, to get from $\langle y := x + 1;\ \parallel \{x = 8\}\rangle$ to $\langle y := 8 + 1;\ \parallel \{x = 8\}\rangle$, even though it takes us several steps to demonstrate that such a step will be made.

46.7. Summary of terminology and concepts

An operational semantics gives a precise definition of what it means to run a program. You can use rules of inference to define an operational semantics for a programming language.

Terminology

Atomic step An atomic step is one that is not broken down into (visible) substeps. It makes a discrete change from one full state to another.

Rule Name	Rule
(var 1)	$\overline{\langle v \parallel S\rangle \to \langle \theta_S(v) \parallel S\rangle}$
(op 1)	$\overline{\langle c_1 \text{ op } c_2 \parallel S\rangle \to \langle \text{value}(c_1 \text{ op } c_2) \parallel S\rangle}$
(op 2)	$\dfrac{\langle \alpha \parallel S\rangle \to \langle \gamma \parallel T\rangle}{\langle \alpha \text{ op } \beta \parallel S\rangle \to \langle \gamma \text{ op } \beta \parallel T\rangle}$
(op 3)	$\dfrac{\langle \alpha \parallel S\rangle \to \langle \gamma \parallel T\rangle}{\langle c \text{ op } \alpha \parallel S\rangle \to \langle c \text{ op } \gamma \parallel T\rangle}$

Figure 46.3: **An operational semantics of expressions in a very small programming language.** Symbols c, c_1 and c_2 stand for arbitrary constants; v stands for a variable; α, β and γ stand for arbitrary expressions. Notation value(c_1 op c_2) indicates the value of an expression. For example, value$(3 + 5) = 8$, and value$(3 < 5) =$ true.

Rule (var 1) says to fetch the value of a variable from the state when you encounter a variable.

Taken together, rules (op 1), (op 2) and (op 3) say that, to evaluate expression (A op B), first evaluate A, then evaluate B, and finally perform the operation that is called for.

Evaluation sequence An evaluation sequence is a sequence of full states $s_1 \to s_2 \to \cdots \to s_n$, where consecutive states are related by the → relationship.

Full state A full state tells the value of each variable, and also tells the state of the program, so that you know what the program will do next, and how it will continue running.

Operational semantics An operational semantics associates, with each program p and simple state s, a sequence of full states through which program p, started in simple state s, takes a computer when it runs.

Rule of inference A rule of inference allows you to conclude a fact from zero or more previously proved facts.

Simple state A simple state tells the value of each variable that has a value.

Single-step relation Two full states are related by the single-step relation, →, if one follows the other in a computation. We write $s \to t$ to indicate that full state t comes immediately after full state s, as the result of a single atomic step.

46.8. Exercises

46.1. Using the semantic rules of Figures 46.3 and 46.4, show the entire sequence of full states in the evaluation of statement

Rule Name	**Rule**
(assign 1)	$\dfrac{\langle \alpha \parallel S\rangle \to \langle \beta \parallel T\rangle}{\langle v\ :=\ \alpha;\ L \parallel S\rangle \to \langle v\ :=\ \beta;\ L \parallel T\rangle}$
(assign 2)	$\dfrac{}{\langle v\ :=\ c;\ L \parallel S\rangle \to \langle L \parallel S[v := c]\rangle}$
(if 1)	$\dfrac{}{\langle \textbf{if}\ \text{true}\ \textbf{then}\ A\ \textbf{else}\ B\ \textbf{endif}\ L \parallel S\rangle \to \langle A\,L \parallel S\rangle}$
(if 2)	$\dfrac{}{\langle \textbf{if}\ \text{false}\ \textbf{then}\ A\ \textbf{else}\ B\ \textbf{endif}\ L \parallel S\rangle \to \langle B\,L \parallel S\rangle}$
(if 3)	$\dfrac{\langle \alpha \parallel S\rangle \to \langle \beta \parallel T\rangle}{\begin{array}{l}\langle \textbf{if}\ \alpha\ \textbf{then}\ A\ \textbf{else}\ B\ \textbf{endif}\ L \parallel S\rangle \to \\ \langle \textbf{if}\ \beta\ \textbf{then}\ A\ \textbf{else}\ B\ \textbf{endif}\ L \parallel T\rangle\end{array}}$
(if 4)	$\dfrac{}{\langle \textbf{if}\ \text{true}\ \textbf{then}\ A\ \textbf{endif}\ L \parallel S\rangle \to \langle A\ L \parallel S\rangle}$
(if 5)	$\dfrac{}{\langle \textbf{if}\ \text{false}\ \textbf{then}\ A\ \textbf{endif}\ L \parallel S\rangle \to \langle L \parallel S\rangle}$
(if 6)	$\dfrac{\langle \alpha \parallel S\rangle \to \langle \beta \parallel T\rangle}{\begin{array}{l}\langle \textbf{if}\ \alpha\ \textbf{then}\ A\ \textbf{endif}\ L \parallel S\rangle \to \\ \langle \textbf{if}\ \beta\ \textbf{then}\ A\ \textbf{endif}\ L \parallel T\rangle\end{array}}$
(while 1)	$\dfrac{}{\begin{array}{l}\langle \textbf{while}\ \alpha\ \textbf{do}\ A\ \textbf{endwhile}\ L \parallel S\rangle \to \\ \langle \textbf{if}\ \alpha\ \textbf{then}\ A\ \textbf{while}\ \alpha\ \textbf{do}\ A\ \textbf{endwhile}\ \textbf{endif}\ L \parallel S\rangle\end{array}}$

Figure 46.4: **An operational semantics of statements in a very small programming language.** Symbol c stands for an arbitrary constant; v stands for a variable; α and β stand for arbitrary expressions; and A, B and L stand for sequences of (zero or more) statements.

The rules for assignment say that, to perform $v := E$, first evaluate E (assign 1), yielding some number c, then add $v = c$ to the state, and remove the assignment from the program, since it has been done (assign 2). You should be able to understand what the remaining rules say to do.

1. $\langle x \parallel \{x = 8\}\rangle \to \langle 8 \parallel \{x = 8\}\rangle$ [by (var 1)].
2. $\langle x + 1 \parallel \{x = 8\}\rangle \to \langle 8 + 1 \parallel \{x = 8\}\rangle$ [by (op 2), using fact 1].
3. $\langle y := x + 1; \parallel \{x = 8\}\rangle \to \langle y := 8{+}1; \parallel \{x = 8\}\rangle$ [by (assign 1), using fact 2].

Figure 46.5: **Proof that $\langle y$:= $x + 1$; $\parallel \{x = 8\}\rangle \to \langle y$:= $8 + 1$; $\parallel \{x = 8\}\rangle$.** Notice that it can take several steps of reasoning to derive what a single step of the program does. You start with simple relationships and build up to larger ones.

```
if x > 0 then
  y := y + 1;
else
  z := z + 1;
endif
```

starting in simple state {x = 1, y = 5, z = 10}, until the program fragment is done.

46.2. Using the semantic rules of Figures 46.3 and 46.4, show the entire sequence of full states in the evaluation of statement

```
while x < 3 do
  y := y + y;
  x := x + 1;
endwhile
```

starting in simple state {x = 1, y = 5}, until the program fragment is done.

46.3. Which of the following relations are true according to the rules in Figures 46.3 and 46.4?

(a) $\langle y+1 \parallel \{y=8\}\rangle \to \langle 9 \parallel \{y=8\}\rangle$

(b) $\langle y+1 \parallel \{y=8\}\rangle \to \langle 8+1 \parallel \{y=8\}\rangle$

(c) $\langle x := 4;\ y := 10; \parallel \{\}\rangle \to \langle y := 10; \parallel \{x=4\}\rangle$

(d) $\langle$**if** $x<y$ **then** $w := 4;$ **else** $w := 5;$ **endif** $\parallel \{x=1, y=2\}\rangle \to \langle w := 4; \parallel \{x=1, y=2\}\rangle$

(e) $\langle$**if** $x<y$ **then** $z := 5;$ **else** $z := 10;$ **endif** $\parallel \{x=1, y=2\}\rangle \to \langle$**if** $1<y$ **then** $z := 5;$ **else** $z := 10;$ **endif** $\parallel \{x=1, y=2\}\rangle$

(f) $\langle x := 2+3; \parallel \{x=7\}\rangle \to \langle x := 5 \parallel \{x=7\}\rangle$

(g) $\langle x := x+1; \parallel \{x=7\}\rangle \to \langle\ \parallel \{x=8\}\rangle$

(h) $\langle x := 2+3; \parallel \{x=7\}\rangle \to \langle x := 2{+}3; \parallel \{x=7\}\rangle$

46.4. Using the rules of Figures 46.3 and 46.4, prove each of the relationships in the previous exercise that is true.

46.5. Using the notation of Section 46.5, provide semantic rules for a conditional expression (if a then b else c), which produces the value of expression b when expression a produces true, and of c when a produces false.

46.6. Using the notation of Section 46.5, provide semantic rules for the C++ or Java expression x++, which produces the current value of variable x, but has the side effect of adding 1 to x. So if x currently has value 3, then expression x++ has value 3, but evaluation of x++ causes x to take on value 4.

46.7. Using the notation of Section 46.5, provide semantic rules for the C++ or Java expression ++x, which adds one to x and produces the new value x. So if x currently has value 3, then expression ++x has value 4, and evaluation of ++x causes x to get value 4.

46.9. Bibliographic notes

The Algol 68 report [105] gives a formal semantics of Algol 68, and its effects are discussed by Lindsey [71]. Milner, Tofte and Harper [78] give a formal semantics of ML. Winskel [110] describes different ways of describing programming language semantics. Pierce [87] explains operational semantics. Arnold, Joy, Steele and Bracha [8] give a much less mathematical definition of Java.

Chapter 47

Reasoning About Imperative Programs

47.1. Reasoning about correctness

How can you convince yourself that an imperative program is correct? How do you know that it does what you expect it to do? An obvious approach is to test it. Clearly, testing is an important part of creating a program, but it cannot be the entire basis for justifying the correctness of a program, for the following reasons.

1. Testing can show the presence of mistakes, but it cannot guarantee their absence. Testing inevitably misses some cases where errors might occur. So testing does not really convince you that your program is free of errors.

2. If testing is the only way of understanding whether or why a program works, then how was the program written? Was it written entirely by trial and error? That is very unlikely. You must have had in mind some reason for writing the program the way you did, and you must have used some form of correctness reasoning that had nothing to do with testing. It might help if that reasoning could be understood in more than strictly intuitive terms.

Another approach to convincing yourself that a program is correct is to perform a hand simulation. That goes directly to the underlying operational semantics. To see that a program is correct, you follow the sequence of states through which the program takes a computer. As a start, you can perform a hand simulation on specific test inputs. To test a factorial function, for example, you might hand simulate it to compute the factorial of 3. Unfortunately, just trying a few cases by hand cannot, in itself, convince you that the program works for all inputs, any more than doing those tests on a computer can convince you.

So, after trying some test cases, you might try doing a hand simulation on *generic* data, where you do not select specific values for the inputs, but leave them unknown. (If the input is an integer, n, just call it n. Keep everything symbolic.) That approach concentrates on the dynamics of the program, following the course of computations to see that they lead to the correct solution. Although that can often be made to work, we will take an alternative approach that, instead of looking at how a program changes things, concentrates on static aspects of the program: things that do not change while the program runs.

This chapter explores a method of *program verification* that allows you to give a mathematical proof that a program works according to its specification. In addition to providing solid evidence that a program is correct, program verification has an advantage over hand simulation. Even small programs can take a computer through millions of states, as the program goes around a loop, and performing a hand simulation can be prohibitively time consuming. But the amount of work required to perform a verification by the method described here is more closely tied to the size of the program itself than to the length of the computation, so verification can be tractable when hand simulation is difficult.

47.2. Partial and total correctness

Program verification generally breaks down into two parts. First, you must convince yourself that any results that the program produces are correct. A program that cannot produce incorrect results is said to be *partially correct.* Partial correctness is obviously important, but it is not the whole story. A program can be partially correct, yet produce no results at all. For example, it might loop forever, never giving an answer. The second part of program verification is demonstrating that the program must eventually produce some results. A program that is both partially correct and is sure to terminate is said to be *totally correct.*

We only look at demonstrations of partial correctness here. Proofs of termination typically need to be done by other means.

47.3. Verification of flowgraphs

Flowgraphs offer a simple mechanism for expressing control, and they are a good starting point for studying program verification. The main idea is to concentrate not on the dynamics of computation, but on things that never change. There are three major steps to doing this.

1. For each edge in the flowgraph, choose a condition that is true every time the flowgraph reaches that edge. The condition associated with an edge is called an *assertion*, and it must assert properties of the *current* values of the program's variables. Remember that it is static, and does not discuss how the variables will change later, or what their values have been in the past.

2. Of course, just making a claim is not enough; the claim must be demonstrated. From the assertions that you have attached to flowgraph edges, derive verification conditions in a mechanical fashion so that the truth of the verification conditions implies the truth of the claim that each assertion is true whenever its edge is reached. The method of getting the verification conditions is described below.

3. The final step is to show that each of the verification conditions is true, using standard mathematical reasoning.

An example is the best place to start. Figure 47.1 shows a flowgraph with assertions attached to the edges, called an *augmented* flowgraph. The verification conditions (derived using rules

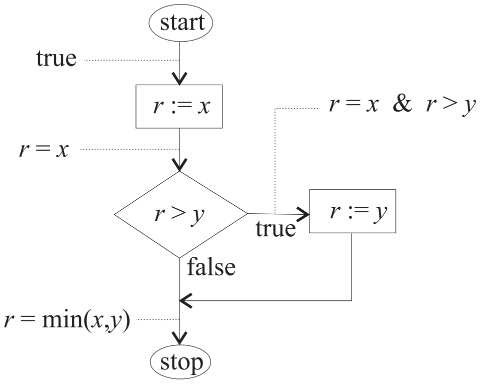


Figure 47.1: **An augmented flowgraph for computing** $r = \mathbf{min}(x, y)$**.** Each edge has an assertion attached to it that is a property of the variable values when the flowgraph reaches that edge. Assertion **true** indicates that nothing is known initially.

from the next subsections) are as follows.

$$\begin{aligned} \text{true} \quad &\Rightarrow \quad x = x && (47.3.1)\\ r = x \;\;\&\;\; r \le y \quad &\Rightarrow \quad r = \min(x, y) && (47.3.2)\\ r = x \;\;\&\;\; r > y \quad &\Rightarrow \quad r = x \;\;\&\;\; r > y && (47.3.3)\\ r = x \;\;\&\;\; r > y \quad &\Rightarrow \quad y = \min(x, y) && (47.3.4) \end{aligned}$$

There is one verification condition for each assignment node in the flowgraph, and two for each decision node (one for the true branch and one for the false branch). Verification condition (47.3.1) expresses that the initial assignment statement, $r := x$, changes things in accordance with what the assertions say is true. Conditions (47.3.2) and (47.3.3) cover the false and true branches, respectively, of the decision node, and condition (47.3.4) is for the assignment node $r := y$.

Once you have the verification conditions, you check that each is true. You do not need to know where the conditions come from at that stage. For example, you know that $x = x$, so you know that verification condition (47.3.1) is true.

Figure 47.2 shows another example, the factorial flowgraph from Chapter 14, with an assertion attached to each edge. The assertion $n > 0$ on the start edge is a *precondition* that the input value must satisfy in order for the verification (and the program) to be correct. The remaining assertions are claimed to be true when their respective edges are reached, as long as the precondition is true at the beginning.

You should convince yourself informally that each assertion attached to an edge is true when that edge is reached during computation. Try some examples. For example, run through a computation of the factorial flowgraph with $n = 3$. Check the assertions as you go. Notice that the assertions are not assignment statements, but mathematical statements. The assertions are not part of the program. They are part of the justification that the program works.

An augmented flowgraph constitutes a verification of the flowgraph if it meets two general requirements. The first is simple: all edges pointing to a given node must have the

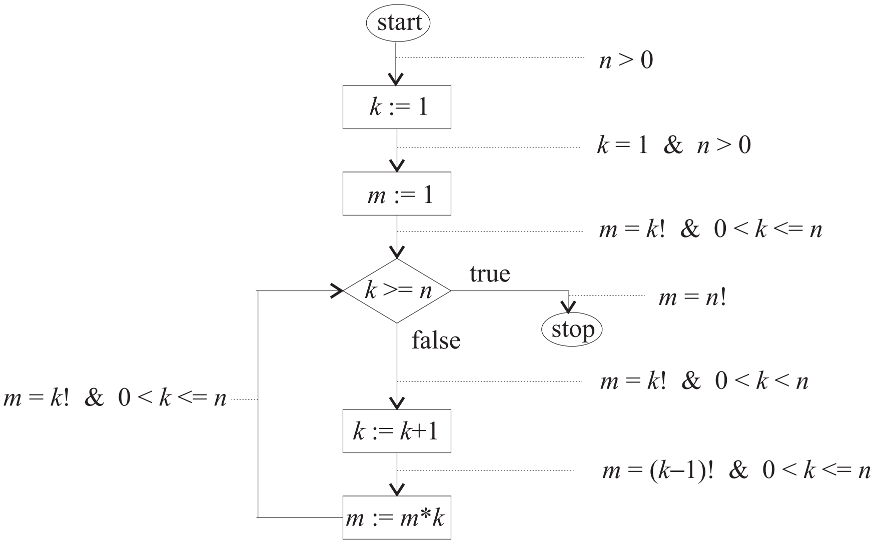


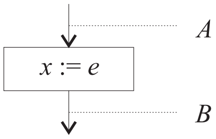
Figure 47.2: **An augmented flowgraph for computing $m = n!$.** This flowgraph contains a loop, but follows the same rules for attaching assertions to its edges as a flowgraph without a loop. Assertion $n > 0$ is a precondition that is required by this flowgraph.

same assertion attached to them. For example, in the augmented flowgraph for factorial, there are two edges pointing to the test $k \geq n$. Both of those edges have the same assertion attached to them. You are really attaching an assertion to the entry point for each node.

The second requirement is that verification conditions that are derived from the augmented flowgraph must be true. Rules for deriving verification conditions are described in the next two subsections.

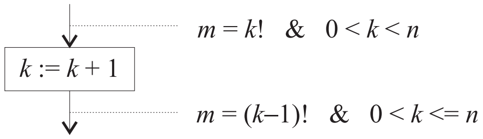
Verification conditions for assignment nodes

Suppose there is an assignment node



holding assignment $x := e$, where A is the assertion on the incoming edge and B is the assertion on the outgoing edge. (If there is more than one incoming edge, recall that all must have the same assertion on them.) Assertion A is called the *precondition* for this node, since it is presumed true before the assignment is performed, and assertion B is called the *postcondition* for the node, since it is claimed to be true after the assignment.

Let B' be the mathematical expression derived from B by replacing each occurrence of x by expression e. Then the verification condition for this assignment node is $A \Rightarrow B'$. That is, if A is true, then B' must be true. For example, consider the assignment



from Figure 47.2. Here, assertion A is $(m = k!\ \ \&\ \ 0 < k < n)$ and assertion B is $(m = (k-1)!\ \ \&\ \ 0 < k \leq n)$. Replacing each occurrence of k in assertion B by $k+1$ (the right-hand side of the assignment statement) yields assertion B', which is $(m = ((k+1)-1)!\ \ \&\ \ 0 < k+1 \leq n)$. So this node leads to verification condition

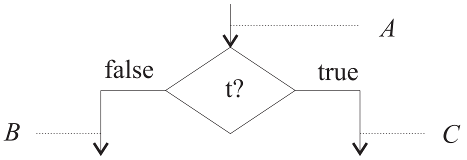
$$m = k!\ \ \&\ \ 0 < k < n \quad \Rightarrow \quad m = ((k+1)-1)!\ \ \&\ \ 0 < k+1 \leq n,$$

which is easily seen to be true for all integers k, m and n. (The term "easily" is relative. It means take a moment and work it out. You will see it. Remember that k and n are integers, so $k < n$ is equivalent to $k+1 \leq n$.)

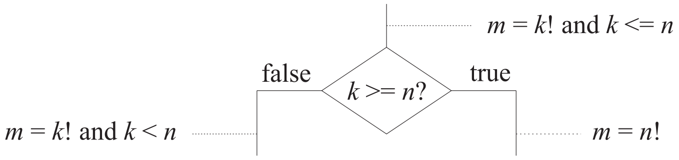
In order to understand why verification conditions for assignment statements are chosen in this way, think backwards from the postcondition to the precondition. Ask yourself, if you want assertion B to be true after performing assignment $x := v$, what assertion would have to be true before doing that assignment? Since the value of x will become v after the assignment, what you need before the assignment is assertion B', where x has been replaced by its new value v in B. For example, if you want to claim that x is prime after the assignment, you need to know that v is prime before the assignment, since v is the value that x will take on.

Verification conditions for test nodes

Suppose there is a test node



where t is the condition being tested, A is the assertion on the incoming edge, B is the assertion of the outgoing edge that is labeled false and C is the assertion on the outgoing edge that is labeled true. Two verification conditions are derived from this, and both must be proved. They are $((A\ \ \&\ \ t) \Rightarrow C)$ and $((A\ \ \&\ \ \neg t) \Rightarrow B)$, where $\neg t$ is read "not t", and is true just when t is false. The augmented factorial flowgraph, for example, contains a test node



The verification conditions are

$$m = k!\ \ \&\ \ 0 < k \leq n\ \ \&\ \ k \geq n \quad \Rightarrow \quad m = n!$$
$$m = k!\ \ \&\ \ 0 < k \leq n\ \ \&\ \ k < n \quad \Rightarrow \quad m = k!\ \ \&\ \ 0 < k < n$$

Each of these conditions is easy to check. For the first one, since $k \leq n$ and $k \geq n$, it must be the case that $k = n$. Try extracting the remaining verification conditions from the augmented flowgraph in Figure 47.2, and check that each is true.

Deriving assertions

Up to now, we have assumed that assertions have already been attached to the edges of a flowgraph. But it is reasonable to ask where they come from. Like the flowgraph itself, they are the product of the programmer's reasoning. To augment a flowgraph with assertions, begin by adding obvious facts. For example, at a test with condition t, add t to the true branch, and $\neg t$ to the false branch. For simple flowgraphs, such as the minimum-calculator in Figure 47.1, that might be all you need. In fact, you might find that some parts of the assertions are unnecessary, and you can remove them.

The presence of loops often requires more thought. Ask yourself what the loop is accomplishing, and what is true each time you reach a particular node in the loop. If you do a hand simulation of the flowgraph of Figure 47.2 with $n = 4$, and look at the values of n, k and m each time you reach the choice node, you get the following collections of values.

$$\begin{array}{l}\{k = 1, m = 1, n = 4\}\\ \{k = 2, m = 2, n = 4\}\\ \{k = 3, m = 6, n = 4\}\\ \{k = 4, m = 24, n = 4\}\end{array}$$

Since your intent is to compute factorials, it should jump out at you that $m = k!$. The remaining assertions are added to make the verification condions provable, by trial and error and some thought.

47.4. Verification of structured programs

Every structured program can be converted to an equivalent flowgraph. So one approach to verification of structured programs is to convert to a flowgraph and perform the verification on the flowgraph. It is possible, however, to perform a verification of a structured program without conversion to a flowgraph. Instead of writing the entire program and then attaching assertions, you derive the assertions and the program together, building up larger pieces from smaller ones. So you build *verified* programs rather than just plain programs.

A *Hoare triple* has the form $\{A\}\, S\, \{B\}$ where A and B are assertions and S is a statement or sequence of statements. The meaning of that triple is that, if A is true, and then statement(s) S are performed, then B will be true afterwards. For example, Hoare triple $\{k = 1\}$ $k := k + 1$ $\{k = 2\}$ says that, if assertion $k = 1$ is true now, and statement $k := k + 1$ is performed, then assertion $k = 2$ will be true afterwards. In Hoare triple $\{A\}\, S\, \{B\}$, assertion A is called the *precondition* and assertion B is called the *postcondition.*

Verification is done by using rules for inferring true Hoare triples from other true Hoare triples using a system of reasoning is called *Hoare logic*, named after its inventor. Rules of inference are written $\dfrac{a}{b}$ where a is a collection of known facts and b is a fact that is derived from them. If there is nothing above the line, then what is below the line can be concluded without any prior facts.

Figure 47.3 shows rules of inference for the Hoare logic of assignments, sequencing, if-then-else and while-do, the same tiny language whose operational semantics is defined in

Rule Name	Rule
(Hoare-assign)	$\dfrac{}{\{A[x \leftarrow e]\}\ x\ :=\ e;\ \{A\}}$
(Hoare-sequence)	$\dfrac{\{A\}\,S\,\{B\},\ \{B\}\,T\,\{C\}}{\{A\}\ S\,T\ \{C\}}$
(Hoare-if)	$\dfrac{\{A\ \&\ C\}\,S\,\{B\},\ \{A\ \&\ \neg C\}\,T\,\{B\}}{\{A\}\ \textbf{if}\ C\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{endif}\ \{B\}}$
(Hoare-while)	$\dfrac{\{A\ \&\ C\}\,S\,\{A\}}{\{A\}\ \textbf{while}\ C\ \textbf{do}\ S\ \textbf{endwhile}\ \{A\ \&\ \neg C\}}$
(Hoare-weaken)	$\dfrac{A \Rightarrow B,\ C \Rightarrow D,\ \{B\}\,S\,\{C\}}{\{A\}\,S\,\{D\}}$

Figure 47.3: Inference rules for Hoare logic.

Chapter 46, omitting the if-statement without an else part. We assume that expressions only compute values, and do not change anything. Notation $A[x \leftarrow e]$ indicates the expression obtained by replacing every occurrence of variable x in expression A by expression e. For example, if A is $x + x$, then $A[x \leftarrow 9]$ is 9+9. Parentheses need to be used if necessary to prevent operator precedence from changing how the expression is broken up into parts. For example, if A is $x * y$, then $A[x \leftarrow u + v] = (u + v) * y$, not $u + v * y$. Notation $\neg A$ indicates that A is false.

Most of the Hoare rules are easy to justify. The sequence rule (Hoare-sequence), for example, presumes that you already know $\{A\}\,S\,\{B\}$ and $\{B\}\,T\,\{C\}$. Since the postcondition for statement S is identical to the precondition for statement T, those two triples can be combined to conclude that $\{A\}\,S\,T\,\{C\}$. Rule (Hoare-weaken) states that you can always strengthen the precondition and weaken the postcondition. For example, if $\{k \geq 0\}\,S\,\{n > 0\}$ is known to be true, then certainly $\{k > 0\}\,S\,\{n \geq 0\}$ must also be true.

Rule (Hoare-while) for the while loop needs some discussion. Assertion A is called an *invariant* of the loop, and is an assertion that is true each time the loop reaches its top. Suppose that, in a given program, A is an invariant for a given loop **while** C **do** S **endwhile**. Then A will certainly need to be true when the loop starts, since otherwise it would not be true every time the loop is at its top. That is, A is a precondition of the loop.

Now think about the last time the loop reaches its top, just before the loop ends. Since A is an invariant, A will be true. But if the loop is to exit, then C must be false. The program goes to the end of the loop, with A true and C false. Putting this together, we get Hoare triple $\{A\}$ **while** C **do** S **endwhile** $\{A\ \&\ \neg C\}$. But this presumes that A is indeed an invariant of the loop. In order for that to be true, statement(s) S must not ruin the invariant. Notice that S is only performed when C is true; to show that S preserves the truth of A, we need to show that, if A and C are both true when S starts, then A is true again when S is finished. So what is required is $\{A\ \&\ C\}\,S\,\{A\}$.

A short sample verification

To prove characteristics of programs using Hoare-style rules, you need to keep two things in mind. First, build up the verification as you build up the program. So start with small pieces. Second, keep your objective in mind. That is, prove what you want to prove, not just what is obviously true. Suppose that you want to prove that program fragment

```
if $x > y$ then
   $r := x$;
else
   $r := y$;
end if
```

sets r to max(x, y), the larger of x and y. Start with the two assignment statements. But remember that you are only interested in what statement $r := x$ does in the case when $x > y$, because that is the only time it is used. Obviously, it sets $r = x$. But, more to the point, it sets $r = \max(x, y)$ in the case where it is used. Using (Hoare-assign) twice, derive the following two Hoare-triples.

$$\begin{array}{rcl} \{x = \max(x,y) \;\;\&\;\; x > y\} & r := x; & \{r = \max(x,y) \;\;\&\;\; x > y\} \\ \{y = \max(x,y) \;\;\&\;\; \neg(x > y)\} & r := y; & \{r = \max(x,y) \;\;\&\;\; \neg(x > y)\} \end{array}$$

It was necessary to put $x > y$ or $\neg(x > y)$ in both the precondition and the postcondition of each Hoare-triple in order for rule (Hoare-assign) to apply. But you do not really care whether $x > y$ or not after the assignment is finished, so remove that irrelevant information. Also, notice that $x > y \Rightarrow x = \max(x,y)$ and $\neg(x > y) \Rightarrow y = \max(x,y)$, so the preconditions can be simplified. Use rule (Hoare-weaken) to derive the following.

$$\begin{array}{rcl} \{\text{true} \;\;\&\;\; x > y\} & r := x; & \{r = \max(x,y)\} \\ \{\text{true} \;\;\&\;\; \neg(x > y)\} & r := y; & \{r = \max(x,y)\} \end{array}$$

Now use rule (Hoare-if), where C is $x > y$.

$$\{\text{true}\} \quad \textbf{if } x > y \textbf{ then } r := x \textbf{ else } r := y \textbf{ end if} \quad \{r = \max(x,y)\}$$

A longer sample verification

It is possible to demonstrate from the precondition $n > 0$ that the following program ends satisfying postcondition $m = n!$.

```
$k := 1$;
$m := 1$;
while $k < n$ do
   $k := k + 1$;
   $m := m * k$;
endwhile
```

The proof goes as follows, carried out in detail. Hoare triples are broken into lines instead of being written in a single line. You should see strong similarities between this proof and the flowgraph verification. Check the steps.

Step	Conclusion	Justification
1	$\{1 = 1 \ \ \& \ \ n > 0\}$ $\quad k := 1;$ $\{k = 1 \ \ \& \ \ n > 0\}$	by (Hoare-assign)
2	$\{n > 0\}$ $\quad k := 1;$ $\{k = 1 \ \ \& \ \ n > 0\}$	by fact (1) and rule (Hoare-weaken). This is because $(n > 0 \Rightarrow 1 = 1 \ \ \& \ \ n > 0)$. We will not continue to state such obvious facts.
3	$\{k = 1 \ \ \& \ \ 1 = 1! \ \ \& \ \ n > 0\}$ $\quad m := 1;$ $\{k = 1 \ \ \& \ \ m = 1! \ \ \& \ \ n > 0\}$	by (Hoare-assign)
4	$\{k = 1 \ \ \& \ \ n > 0\}$ $\quad m := 1;$ $\{m = k! \ \ \& \ \ 0 < k \leq n\}$	by fact (3) and rule (Hoare-weaken), and obvious facts about the integers. For example, $(k = 1$ and $n > 0) \Rightarrow k \leq n$ when k and n are integers.
5	$\{n > 0\}$ $\quad k := 1;$ $\quad m := 1;$ $\{m = k! \ \ \& \ \ 0 < k \leq n\}$	by facts (2) and (4) and rule (Hoare-sequence)
6	$\{m = ((k+1) - 1)! \ \ \& \ \ 0 < k + 1 \leq n\}$ $\quad k := k + 1;$ $\{m = (k - 1)! \ \ \& \ \ 0 < k \leq n\}$	by (Hoare-assign)
7	$\{m = k! \ \ \& \ \ 0 < k < n\}$ $\quad k := k + 1;$ $\{m = (k - 1)! \ \ \& \ \ 0 < k \leq n\}$	by fact (6) and rule (Hoare-weaken), and obvious facts about the integers
8	$\{m * k = k! \ \ \& \ \ 0 < k \leq n\}$ $\quad m := m * k;$ $\{m = k! \ \ \& \ \ 0 < k \leq n\}$	by (Hoare-assign)
9	$\{m = (k - 1)! \ \ \& \ \ 0 < k \leq n\}$ $\quad m := m * k;$ $\{m = k! \ \ \& \ \ 0 < k \leq n\}$	by fact (8) and rule (Hoare-weaken), and the fact that $m = (k - 1)! \Rightarrow m * k = k!$ when $k > 0$
10	$\{m = k! \ \ \& \ \ 0 < k < n\}$ $\quad k := k + 1;$ $\quad m := m * k;$ $\{m = k! \ \ \& \ \ 0 < k \leq n\}$	by facts (7) and (9) and rule (Hoare-sequence)
11	$\{m = k! \ \ \& \ \ 0 < k \leq n \ \ \& \ \ k < n\}$ $\quad k := k + 1;$ $\quad m := m * k;$ $\{m = k! \ \ \& \ \ 0 < k \leq n\}$	by fact (10) and rule (Hoare-weaken)
12	$\{m = k! \ \ \& \ \ 0 < k \leq n\}$ **while** $k < n$ **do** $\quad\quad k := k + 1;$ $\quad\quad m := m * k;$ **endwhile** $\{m = k! \ \ \& \ \ 0 < k \leq n \ \ \& \ \ \neg(k < n)\}$	by fact (11) and rule (Hoare-while)

13	$\{m = k!\ \ \&\ \ 0 < k \leq n\}$ **while** $k < n$ **do** $k := k + 1;$ $m := m * k;$ **endwhile** $\{m = n!\}$	by fact (12) and rule (Hoare-weaken), and obvious facts about the integers. Notice that $(k \leq n)\ \ \&\ \ \neg(k < n) \Rightarrow k = n$. But then, if $m = k!$, surely $m = n!$.
14	$\{n > 0\}$ $k := 1;$ $m := 1;$ **while** $k < n$ **do** $k := k + 1;$ $m := m * k;$ **endwhile** $\{m = n!\}$	by facts (5) and (13) and rule (Hoare-sequence)

47.5. Axiomatic semantics

A program is only useful in practice to the extent that the programmer understands what it does, and can convince himself or herself of the program's correctness. If a program were written, for example, by randomly typing on the keyboard, and nobody understood the program, then the program would almost surely need to be thrown away, even if, by some fantastic fortune, it happened to do exactly what was desired. Indeed, many a working computer program or procedure definition, written by another programmer, has been thrown away because a current programmer did not understand it, and preferred to substitute his or her own version that was more easily understood (by him or her).

It might be argued, therefore, that the semantics of a programming language is only useful to the extent that it helps a programmer reason about the behavior of programs. Suppose that a programmer has decided to use Hoare logic to reason about programs. Then the inference rules in Figure 47.3 are the only facts about programs that the programmer will use. They might just as well *be* the semantics of programs.

A semantics based on providing facts and rules of inference for reasoning about programs is called an *axiomatic semantics.* The Hoare-logic rules of inference form an axiomatic semantics of a simple programming language. Rather than discussing what a program does when it is run, the axiomatic semantics only tells you which facts can be proved about programs. Axiomatic semantics have the advantage that they can tell a programmer important facts about a programming language without becoming mired in the details of how computation is performed. They work at a level of abstraction above all of those details, and so are a very high level semantics.

By the same token, however, by suppressing the details, an axiomatic semantics can leave a programmer uneasy about what a program does, and how it works. A practical approach to providing the semantics of a programming language is to provide multiple consistent semantics. An operational semantics might be useful to a programmer for some parts of the programming process. At other stages, the programmer might prefer to think in terms of an axiomatic semantics that is consistent with the operational semantics, but that suppresses unwanted details. Providing more than one form of semantics is probably the most reasonable approach to defining a programming language.

47.6. Practical reasoning about programs

The goal of program verification is to prove, beyond any doubt, that a program performs according to some clearly stated requirements. It should be fairly evident from the examples that full verification of programs is a very lengthy and difficult process. Indeed, there can be so many steps that, in any verification performed by a human, there will almost surely be some mistakes. The potential presence of those mistakes casts doubt on the verification itself, and hence on the software that has purportedly been verified. It can be argued that a further verification of the verification is required!

Does this apparent circularity spell the death of program verification? Fortunately, there are ways out of the difficulties.

Informal verification

One way out is to relax the goal. In real software design, it might not be necessary to have absolute faith that there are no mistakes in a program. Nonetheless, any method that will increase confidence in the program's correctness and will decrease the presence of mistakes will be useful. Program verification offers just such a method. A programmer should think through at least a sketch of a proof that his or her program works according to specifications. Hoare logic provides a framework for such proof sketches.

Difficulties in imperative programs are often associated with loops. Hoare logic suggests that you find a loop invariant for your loop, not just after you have written it, but during the process of writing the loop. For example, suppose that you need a function to compute x^n, where x is a number and n is a positive integer. You can compute successive powers x^1, x^2, x^3, ..., x^n. Those powers will need to be stored somewhere, so suppose that variable r successively holds those powers. Somehow, the program must keep track of which of those powers it currently has. Let variable k play that role. Then a loop invariant will be $r = x^k$. The program must begin by performing some assignment statements that make this invariant true. (It suffices to set $r := x$ and $k := 1$.) Then it must perform a loop, each time making k larger so that the desired power x^n will be approached, and each time keeping the invariant true. Statements $k := k + 1; r := r * x;$ will perform the updates. A little thought shows that those steps keep assertion $r = x^k$ true. Now when should the loop stop? Since $r = x^k$ throughout the loop, and we want $r = x^n$ when the loop stops, the loop should obviously stop when $k = n$. This reasoning leads to the following program fragment that sets $r = x^n$.

```
$r := x$;
$k := 1$;
while $k \neq n$ do
   $k := k + 1$;
   $r := r * x$;
endwhile
```

Although it has not been fully verified, a rough verification of the program fragment, based on the idea of a loop invariant, has been done during the process of writing it.

Formal verification

Complete verification of a large program by a human is out of reach. No person will be able to do the entire job with such care that even he or she will be confident that there are no errors in his or her own work; certainly, nobody else will have such confidence. Formal verification can be done, though, provided the verification is checked by a computer. Automatic proof checkers can be used to check that verifications have been done correctly. The best proof checkers not only check the steps of a proof, but can also fill in some missing details, freeing the human from having to write down each and every tiny step of the proof.

47.7. Summary of terminology and concepts

Verification is a way of demonstrating that a program satisfies certain clearly stated requirements. The requirements are written in a logical notation, and verification employs mathematical logic to yield a proof that a program meets the requirements. Verification of a flowgraph has three steps.

1. Augment the program by adding assertions.

2. From the assertions, derive verification conditions.

3. Prove the verification conditions.

Assertions in flowgraphs, or Hoare logic for structured programs, can be used to demonstrate the partial correctness of programs. Proving termination must be done by means that have not been discussed here.

Full verification is very difficult, but partial or informal verification can be an important tool for the programmer.

Axiomatic semantics is a form of programming language semantics that is based on telling a programmer not what a program does, but how the programmer can reason about programs. Axiomatic semantics is a high level semantics, suppressing details of computation.

Terminology

Assertion An assertion is a condition that is asserted to be true whenever a program reaches a certain point during its execution. An assertion gives properties of and relationships among the *current* values of a program's variables. It does not discuss how variables are changed by the program.

Augmented flowgraph An augmented flowgraph is a flowgraph with an assertion attached to each edge.

Axiomatic semantics Axiomatic semantics is a form of semantics where the only thing that is said about the behavior of a program is what you can prove about that behavior. The semantics of a language is given as a collection of rules of inference.

Hoare logic Hoare logic is a system of reasoning in which statements about the partial correctness of programs can be proved. It is similar in flavor to verification of a flowgraph by verification conditions, but Hoare logic applies to structured programs.

Hoare triple A Hoare triple has the form $\{A\}\ S\ \{B\}$ where A and B are assertions and S is a statement or sequence of statements. It says that, if A is true initially and statement(s) S are executed, then B will be true afterward. A is called the precondition of the triple, and B is called the postcondition of the triple.

Loop invariant A loop invariant for a given loop is an assertion that is true whenever control reaches the top of that loop. Loop invariants are critical to the verification of loops both using Hoare logic and using augmented flowgraphs. Loop invariants are also useful tools for designing loops.

Partial correctness A program is partially correct if it cannot produce incorrect results.

Postcondition A postcondition is an assertion that is true when execution of a statement (or sequence of statements) is finished.

Precondition A precondition is an assertion that must be true before a statement (or sequence of statements) is performed in order for the statement to have some desired property, such as ensuring that a given postcondition is true when the statement is finished.

Total correctness A program that is partially correct and that is guaranteed to produce results is totally correct.

Verification condition A verification condition is a condition written in mathematical logic whose truth must be shown in order to show that an augmented flowgraph is correctly augmented, and so constitutes a verification of the program.

47.8. Exercises

47.1. Which of the following are reasonable assertions that might be attached to an edge of a flowgraph? An assertion that is always false is not a reasonable one.

(a) $x > y$

(b) $x = x + 1$

(c) Variable x increases by one each time around the loop.

(d) Variable x has been converted into meters.

(e) The new x is twice as large as the old x.

(f) Variable x holds a nonempty string that begins with letter 'a'.

47.2. Suppose that you would like assertion $x > y + 10$ to be true after performing assignment statement $x := 2 * y - 17$. What precondition for the assignment will guarantee that postcondition? Give the most general possible precondition. For example, precondition $y = 100$ guarantees the postcondition, but it is too restrictive, and excludes other values that will also work, so it is not the most general precondition. (Hint: Use the assignment rule (Hoare-assign). It naturally works backwards.)

47.3. Suppose that you would like assertion $x > y + 10$ to be true after performing sequence of statements $y := y + z; x := 2 * y - 17; x := x + 1$. What is the most general precondition for this sequence of statements that will guarantee that postcondition? (Hint: work backwards across the sequence of statements, using rule (Hoare-assign).)

47.4. An empty assertion in a Hoare triple is implicitly true. Prove that Hoare triple {} x := 0 {x = 0} is true, using the rules in Figure 47.3.

47.5. Write all of the verification conditions for the augmented flowgraph in Figure 47.2, and check that each verification condition is true.

47.6. Use the power program fragment of Section 47.6 as a guide for this problem

(a) Draw a flowgraph for a program that computes $p = x^n$, where x and n are positive integers that are the inputs to the program.

(b) Determine assertions to attach to the edges, and show them in the flowgraph. Try a hand simulation as a check that your assertions are reasonable. Is each one true each time you reach its edge?

(c) Derive all verification conditions for your augmented flowgraph.

(d) Argue that each verification condition true. Be sure not to refer back to the flowgraph in this step. You are only concerned with the verification conditions.

47.7. Some languages include a **null** statement that does nothing. Give a Hoare-style rule of inference for a null statement.

47.8. Figure 47.3 shows a rule for an if-then-else statement, but does not include a rule for an if-then statement, without an else part. Give a Hoare-style rule of inference for an if-then statement.

47.9. Give a Hoare-style inference rule for the C or Java do-loop, which has the test at the end instead of at the beginning. It performs the body once before performing the test to see whether to continue the loop.

47.9. Bibliographic notes

Floyd [40] and Hoare [54] introduce ideas of program verification. Other approaches are described by Burstall [20, 21] and Manna et. al. [73]. Hoare and Wirth [55] use axiomatic semantics to describe Pascal. Nipkow, Paulson and Wenzel [82] describe a versatile automatic proof checker.

DeMillo, Lipton and Perlis [76] notice the problem that program verifications themselves must be verified, and point out that few people will want to read other people's verifications.

Part XVI

Topics in Functional Programming

Chapter 48

Reasoning About Equational Programs

48.1. Demonstrating correctness of equational programs

Program correctness can be broken down into two aspects: *partial correctness*, which indicates that a program never produces incorrect results, and *termination*, showing that a program will eventually produce results. Chapter 47 discusses how to demonstrate partial correctness of imperative programs by attaching assertions to the edges of a flowgraph and, from those assertions, deriving *verification conditions* whose truth guarantees the partial correctness of the program.

For equational programming, the equations that you write down *are* the verification conditions, and their correctness guarantees partial correctness of your program. You need to settle on a clear and precise semantic definition of each function, and then show that your equations are true, according to that definition. A program that only contains correct equations will never produce an incorrect answer.

Proving termination for elementary recursive definitions

That leaves the issue of termination. It is possible to write equations that are true, but that cannot be used to produce any answers. An obvious one is to define function f by equation $f(x) = f(x)$, an equation that is certainly is true, but that leads to an infinite recursion when it is used.

Termination can be a sticky issue in a setting where lazy evaluation is employed and data structures can be infinitely large, as described in Chapter 22. For simplicity, we assume that all lists are finitely long here, and we will concentrate on computations that will terminate regardless of whether evaluation is lazy or eager.

We will start by restricting attention to function definitions that can employ recursion, but that do not use mutual recursion among several functions. Then, to demonstrate termination, it suffices show that each recursive call uses a parameter that is, in some (suitable) sense, smaller than the one you started with. If $f(x) = e$ is one of your equations, where e is an expression, then you show that each call to f inside e uses a parameter whose size is smaller than the size of x. You also need to ensure that, whatever measure of size you are using, it is not possible to do infinitely many steps of decreasing the size. Typically, the size is a nonnegative integer, and, since you cannot decrease it below 0, it cannot continue to be decreased forever. A common notion of size is the length of a list. In definition

$$\begin{array}{rcl} \text{length}\,[\,] & = & 0 \\ \text{length}(x :: xs) & = & 1 + \text{length}\ xs \end{array}$$

the recursive call uses a list xs that is shorter than x::xs. When the parameter of a function is an ordered tuple, you often choose the size of one of the tuple's members to be the size of the tuple. For example, suppose that function drop is defined as follows.

$$\begin{array}{rcl} \text{drop}(0, x) & = & x \\ \text{drop}(n, [\,]) & = & [\,] \\ \text{drop}(n+1, x :: xs) & = & \text{drop}(n, xs) \end{array}$$

Of those equations, only the last one involves recursion. The parameter of drop is an ordered pair; suppose the size of pair (n, x) is chosen to be the length of list x. Notice that pair (n, xs) is smaller, in that sense, than pair $(n+1, x :: xs)$, so, since the size is a nonnegative integer, the recursion must eventually terminate. For the definition of drop, more than one notion of size will work. If you define the size of (n, x) to be n, then (n, xs) is smaller than $(n+1, x :: xs)$, and again you have argued that the recursion must terminate.

Not every notion of size will work. For example, if you choose a size that is an integer (positive or negative) then you can continue decreasing the size forever, from -1 to -2, etc. But there is a concept that captures just what is required of a size. A set S of values, along with an ordering relation on S, is called *well-founded* if it is not possible to produce an infinite decreasing sequence $x_1 > x_2 > x_3 > \ldots$ of members of set S. The most commonly used well-founded set is the set of nonnegative integers, with the usual ordering, but there are others. The set of all strings, ordered alphabetically, is well-founded as long as you define alphabetical ordering so that $s < t$ when s is a prefix of t. Another example of a well-founded set is the set of ordered pairs of nonnegative integers, ordered *lexicographically*, where you say that $(x, y) < (u, v)$ if either $x < u$ or $x = u$ and $y < v$. For example, the following equations define a function called Ackermann's function.[1] Values m and n are nonnegative integers, and A produces a nonnegative integer.

$$\begin{array}{rcl} A(0, m) & = & m + 1 \\ A(n+1, 0) & = & A(n, 1) \\ A(n+1, m+1) & = & A(n, A(n+1, m)) \end{array}$$

Let pairs (x, y) be ordered lexicographically. The second equation starts with pair $(n+1, 0)$ and does a recursive call with pair $(n, 1)$, which is smaller than pair $(n+1, 0)$ in the lexicographic order. The third equation makes two recursive calls. But $(n+1, m) < (n+1, m+1)$ in lexicographic ordering, and $(n, z) < (n+1, m+1)$ regardless of what z is, so both recursive calls to A use smaller parameters.

Mutual recursion

A definition is mutually recursive if there are two or more functions that use one another; you might find, for example, that f_1 uses f_2 and f_2 uses f_1. To prove that a mutually

[1] Be careful with this innocent looking definition. Ackermann's function grows very rapidly, and takes a very long time to compute. Although A(10,10) must terminate, it will probably not do so in your lifetime, even on the fastest computer.

recursive definition must terminate, assign each function an integer value in any way that you like. Choose a single well-founded ordering on pairs (n,x) where n is a function number and x is a value of the type that the n-th function takes. (So the type of x can depend on the value of n.) Suppose that equation $f(x) = e$ uses $g(y)$ inside expression e. If f is numbered n_f and g is numbered n_g, then you show that pair $(n_g, y) < (n_f, x)$ in your selected well-founded ordering.

For example, suppose that trees are defined so that each tree is either a leaf containing an integer or is a node with a list of zero or more subtrees. Let's write leaf(n) to indicate a leaf holding integer n, and node(ts) to indicate a node with list of subtrees ts. Function numleaves(t) returns the number of leaves in tree t, and totalleaves(ts) takes a list ts of trees, and returns the sum, over all trees t in ts, of the number of leaves in t.

$$\begin{aligned}\text{numleaves}(\text{leaf}(n)) &= 1\\ \text{numleaves}(\text{node}(ts)) &= \text{totalleaves}(ts)\\ \text{totalleaves}([\,]) &= 0\\ \text{totalleaves}(t :: ts) &= \text{numleaves}(t) + \text{totalleaves}(ts)\end{aligned}$$

Assume that lists and trees are required to be finitely large. Let nodes(t) be the number of nodes and leaves in tree t, and nodes(L) be the sum of nodes(t) over all trees t in list-of-trees L. Let numleaves be function 1 and totalleaves be function 2. Say that $(n_1, a) < (n_2, b)$ just when nodes(a) < nodes(b), where a and b are either lists or trees. You can see that all of the function calls use smaller value in this sense since nodes(node(ts)) < nodes(ts) and nodes(t) < nodes($t :: ts$) and nodes(ts) < nodes($t :: ts$).

48.2. Proving characteristics of functions

Each function in a program should really have two definitions, the semantic definition that tells what you want the function to compute and the program that tells how the function is computed. Proving that a function definition is correct amounts to proving that the two definitions are consistent with one another.

Equational programs are often so close to a semantic definition that it is difficult to come up with any better definition than the program itself. The length of a list, for example, is difficult to define more concisely than to say that length([ ]) = 0 and length($x :: xs$) = $1 +$ length(xs). In those cases, you do not have two definitions to check against one another, and you just have to accept the equational definition as correct.

But there are still some assertions that you might want to prove, involving properties that you believe your functions possess. Suppose that you have defined concatenation of lists, as follows.

$$[\,] \mathbin{++} y = y \qquad (48.2.1)$$

$$(h :: t) \mathbin{++} y = h :: (t \mathbin{++} y) \qquad (48.2.2)$$

One property that ++ possesses is associativity:

$$(x \mathbin{++} y) \mathbin{++} z = x \mathbin{++} (y \mathbin{++} z). \qquad (48.2.3)$$

You can prove Equation (48.2.3) directly from the definition of ++ using the notion of *mathematical induction*. Do a thought experiment where you prove the equation for various lists x (and arbitrary y and z) starting with small lists x.

Step 0. Prove Equation (48.2.3) for $x = [\,]$ (and all y and z).
Step 1. Prove Equation (48.2.3) for all lists x that have length 1 (and all y and z).
Step 2. Prove Equation (48.2.3) for all lists x that have length 2 (and all y and z).

This process has to keep going forever, proving Equation (48.2.3) for larger and larger lists x. But it does not really matter that you never finish, because this is only a thought experiment. All that you really need to do is to show that, for an arbitrary n, you can carry out Step n. But notice that, when you are doing that step, you have already completed all of the prior steps. So you can assume that Equation (48.2.3) is true for lists x that have fewer than n members; you proved that earlier. Here are the arguments, consisting of step 0 and step n for an arbitrary $n > 0$.

Step 0.

$$\begin{array}{rcll} ([\,] \mathbin{++} y) \mathbin{++} z & = & y \mathbin{++} z & \text{by (48.2.1)} \\ [\,] \mathbin{++} (y \mathbin{++} z) & = & y \mathbin{++} z & \text{by (48.2.1)} \end{array}$$

Since the two left-hand sides are equal to the same thing, they are equal to one another, so Equation 48.2.3 is true in the case where $x = [\,]$.

Step n. If list x has length $n > 0$, then x must have the form $h :: t$ for some h and t. But t is shorter than x, so we have already proved, at step $n - 1$, that

$$(t \mathbin{++} y) \mathbin{++} z = t \mathbin{++} (y \mathbin{++} z). \tag{48.2.4}$$

Now use Equations (48.2.2) and (48.2.4) to find expressions equivalent each of $(x \mathbin{++} y) \mathbin{++} z$ and $x \mathbin{++} (y \mathbin{++} z)$.

$$\begin{array}{rcll} (x \mathbin{++} y) \mathbin{++} z & = & ((h :: t) \mathbin{++} y) \mathbin{++} z & \text{since } x = h :: t \\ & = & (h :: (t \mathbin{++} y)) \mathbin{++} z & \text{by (48.2.2)} \\ & = & h :: ((t \mathbin{++} y) \mathbin{++} z) & \text{by (48.2.2)} \\ & = & h :: (t \mathbin{++} (y \mathbin{++} z)) & \text{by (48.2.4)} \\ x \mathbin{++} (y \mathbin{++} z) & = & (h :: t) \mathbin{++} (y \mathbin{++} z) & \text{since } x = h :: t \\ & = & h :: (t \mathbin{++} (y \mathbin{++} z) & \text{by (48.2.2)} \end{array}$$

Since both are equal to the same thing, we have proved that Equation 48.2.3 is true for lists x of length n, and for arbitrary lists y and z. ∎

Example: length($x \mathbin{++} y$) = length(x) + length(y)

The length function is defined by

$$\begin{array}{rcll} \text{length}([\,]) & = & 0 & (48.2.5) \\ \text{length}(h :: t) & = & 1 + \text{length}(t) & (48.2.6) \end{array}$$

Here is a proof, from the defining equations of length and ++, that length(x) + length(y) = length($x \mathbin{++} y$), using mathematical induction on list x.

Step 0. Suppose that $x = [\,]$.

$$\begin{array}{rcll} \text{length}([\,]) + \text{length}(y) & = & 0 + \text{length}(y) & \text{by (48.2.5)} \\ & = & \text{length}(y) & \\ \text{length}([\,] \mathbin{++} y) & = & \text{length}(y) & \text{by (48.2.1)} \end{array}$$

so length(x) + length(y) = length($x \mathbin{++} y$) in the case where $x = [\,]$.

Step ***n*.** If list x has length $n > 0$, then x must have the form $h :: t$ for some h and t. But t is shorter then x, so we have already proved, at step $n - 1$, that

$$\text{length}(t) + \text{length}(y) = \text{length}(t \mathbin{++} y). \tag{48.2.7}$$

Now

$$\begin{array}{rcll}
\text{length}(x) + \text{length}(y) & = & \text{length}(h :: t) + \text{length}(y) & \text{since } x = h :: t \\
& = & (1 + \text{length}(t)) + \text{length}(y) & \text{by (48.2.6)} \\
& = & 1 + (\text{length}(t) + \text{length}(y)) & \\
& = & 1 + \text{length}(t \mathbin{++} y) & \text{by (48.2.7)} \\
\text{length}(x \mathbin{++} y) & = & \text{length}((h :: t) \mathbin{++} y) & \text{since } x = h :: t \\
& = & \text{length}(h :: (t \mathbin{++} y)) & \text{by (48.2.2)} \\
& = & 1 + \text{length}(t \mathbin{++} y) & \text{by (48.2.6)}
\end{array}$$

so length(x) + length(y) = length($x \mathbin{++} y$) in this case too. ∎

48.3. Summary of terminology and concepts

Terminology

Lexicographic order Ordered pairs are ordered lexicographically if you say that $(x, y) < (u, v)$ just when either $x < u$ or $x = u$ and $y < v$.

Mathematical induction Mathematical induction is a method of proving a claim that, conceptually, proves a claim by proving it for things of size 0, then for things of size 1, then size 2, etc. The proof is made finitely long by providing a general proof for things of size n. The key is that, when proving the claim for things of size n, you can imagine that you have already proved it for all smaller things in earlier steps.

You can use mathematical induction with any well-founded set of values, not just for nonnegative integers. You do a step for every member of a well-founded set. When you are proving a particular case, you can assume that all smaller cases have already been proved.

Partial correctness A partially correct program can never produce an incorrect answer. It might, however, fail to produce an answer at all, by going into an infinite loop or infinite recursion.

Well-founded set An ordered set is well-founded if there do not exist any infinite decreasing sequences of members of that set. For example, the set of natural numbers (with the usual ordering <) is well-founded.

48.4. Exercises

48.1. What function f do Equations (48.4.1) define? Express f in “closed form”, as a single non-recursive equation. Assume that f only applies to nonnegative integers.

Prove that your assertion about f is true. (**Hint.** Try computing $f(x)$ for a few small integer values of x, and look for a pattern.)

$$\begin{array}{rcl} f(0) & = & 0 \\ f(n+1) & = & f(n) + 2n + 1 \end{array} \tag{48.4.1}$$

48.2. What function s do Equations (18.6.1) define? Assume that s only applies to integers. Prove your assertion first for nonnegative x, and then prove it for negative x.

48.3. Let f be an arbitrary function, and define function mapf as follows.

$$\begin{array}{rcl} \text{mapf}([\,]) & = & [\,] \\ \text{mapf}(h :: t) & = & f(h) :: \text{mapf}(t) \end{array}$$

(a) Expressed in terms of f, a, b and c, what list does mapf($[a,b,c]$) produce?

(b) Assume that $f(z)$ produces an answer for every z. From the definitions of mapf and ++, prove that mapf$(x \mathbin{++} y)$ = mapf(x) ++ mapf(y) for all lists x and y. This equation is true regardless of what function f computes.

48.4. Using Equations (48.2.5) and (48.2.6) and the definition of mapf from the preceding question, prove that length(mapf(x)) = length(x).

48.5. Prove that the set of ordered pairs of integers, ordered lexicographically, is well-founded.

48.6. Suppose that a binary tree has two variants, nil (an empty tree) and node(x, L, R), where x is the node's label, L is the left subtree and R is the right subtree. Function mirror is defined as follows.

```
mirror(nil)           = nil
mirror(node(x, L, R)) = node(x, mirror(R), mirror(L))
```

Prove that mirror(mirror(t)) = t for all binary trees t.

Chapter 49

Semantics of Expressions and Lambda Calculus

49.1. Introduction

This chapter explores methods of carefully describing the semantics of expressions in a functional programming language, including semantics for languages that support higher-order functions. The semantics is based on a mathematical foundation called λ-calculus. The λ-calculus is a small programming language in its own right, and in addition to providing a semantic basis, it serves as an example of a simple but powerful programming language.

49.2. Denotational semantics

Chapter 46 explains how to give an operational semantics of expressions, based on showing the sequence of steps that are involved in doing an evaluation. An alternative approach is a *denotational* semantics, which avoids explaining how an expression is evaluated, and instead jumps directly to the result. The result of evaluating an expression is said the be the value that the expression *denotes*.

Denotational semantics is a kind of *model theory*. A model theory starts with a *model* that is assumed to be understood. For example, the model might be the set of integers, and you must already understand what integers are in order for a model theory based on them to be effective. Each expression is given a meaning that is a member of the model. For example, expression $3 + 5$ might denote the number 8 in the model.

Elementary denotational semantics

If e is an expression, let $D(e)$ indicate the denotation of e. For example, $D(8 - 2) = 6$. We need general rules for defining the D function. An obvious rule concerns numbers and other constants. The symbol, or numeral, 5 denotes the number 5. That is,

$$D(n) = n \quad \text{for any numeral } n.$$

It is important to distinguish between the *numeral* n (just some symbol that occurs in an expression) and the *number* n (a concept from mathematics, or from the model). For example, the numeral 341 is a sequence of three characters. Its denotation, the number 341, is a mathematical concept that you might explain to someone using a pile of 341 toothpicks. Numbers might be written in many different ways in programming languages. For example, imagine that language Caesar uses Roman numerals for numeric constants. Then, in Caesar, $D(XVII) = 17$. That is, the Roman numeral XVII denotes the number 17.

The denotation of $a + b$ can be defined in terms of the addition operation in the model. Assume that you already know how to add numbers in the model; then just explain that $a + b$ in the language calls for performing an addition in the model. Notice that, like numbers, we find ourselves using the same symbol (+ in this case) for two different things, both as a symbol of the programming language and as an operation in the model. It is important to distinguish between the two. If you change your mind and decide to use @ for addition in the programming language, you would still refer to addition in the model by the standard addition symbol, +.

It is tempting to say that $D(a + b) = a + b$. But that is not right, since a and b are not numbers, but expressions. The + on the right-hand side of the equation is the addition operator of mathematics (part of the model), and it does not add expressions; it adds numbers. Before computing the denotation of $a + b$, we need to know the denotations of expressions a and b.

$$D(a + b) = D(a) + D(b).$$

That is, the denotation of expression $a + b$ is defined in terms of the denotations of its subexpressions a and b. For example, you know that $D(4) = 4$ and $D(7) = 7$, so $D(4+7) = 11$. Other definitions immediately suggest themselves.

$$\begin{aligned} D(a * b) &= D(a) \cdot D(b) \\ D(a - b) &= D(a) - D(b) \\ D(a/b) &= \frac{D(a)}{D(b)} \end{aligned}$$

A general principle of denotational semantics is that the denotation of an expression is defined in terms of the denotations of its parts.

Handling variables

There is a close relationship between a denotational semantics and an interpreter written in a functional style. Examination of the simple interpreter of Figure 23.5, written in Scheme, shows that, ignoring differences in notation, the interpreter is almost identical to the denotational semantics of the preceding subsection. To evaluate $a + b$, the interpreter calls itself recursively on a and b, then it adds the results.

The semantics defined in the preceding subsection, like the Scheme interpreter eval-expr, handles expressions that only contain constants and operators. But what about expressions that contain variables (or names), such as x and y? Expression $x + y$ can only be given a denotation if the values of x and y are given. That is, it can only be given meaning in a particular state, or *environment*, that defines bindings of a collection of symbols. For example, environment $\{x = 5, y = 8\}$ indicates bindings for x and y, but not for any other symbols.

If η (the Greek letter eta) is an environment, write $\eta(x)$ for the value that η gives to name x. For example, if η is $\{x = 5, y = 8\}$, then $\eta(y) = 8$. If environment η does not provide a binding of x, then $\eta(x)$ is undefined.

An interpreter that handles variables is generally written by adding another parameter, indicating the environment. Similarly, the concept of the denotation of an expression needs to be changed to take environments into account. If η is an environment and e is an expression, then let $D(e, \eta)$ be the denotation of expression e subject to the bindings in environment η. For example, $D(x+y, \{x = 5,\ y = 8\}) = 13$. Definitions follow fairly easily. Here are just a few.

$$\begin{array}{rcll} D(x,\eta) & = & \eta(x) & \text{for a symbol } x \\ D(n,\eta) & = & n & \text{for a numeral } n \\ D(a+b,\eta) & = & D(a,\eta) + D(b,\eta) & \end{array}$$

Any expression that refers to an unbound symbol has no denotation. For example, $D(x, \{\})$ is undefined.

Suppose that, in a certain language, expression `let` x `=` v `in` e is supposed to evaluate expression v, bind identifier x to the value of v, and then evaluate expression e, yielding the value obtained for e. For example, expression `let` y `= 30 + 2 in y + y` has value 64. The semantics of let-expressions is easy to describe using denotational semantics. If η is an environment, let $\eta/\{\mathrm{x} = \mathrm{v}\}$ be the environment that has all bindings of η, but additionally binds identifier x to v. So if η is $\{x = 3, y = 7\}$ then $\eta/\{z = 9\}$ is environment $\{x = 3, y = 7, z = 9\}$. If η already has a binding for x, then $\eta/\{x = v\}$ uses binding v for x, replacing the prior binding. Using that notation, the semantics of let-expressions is defined as follows.

$$D(\text{let } x = a \text{ in } b,\ \eta) = D(b,\ \eta/\{x = D(a, \eta)).$$

Big steps

Denotational semantics hides details about how expressions are evaluated. The only thing that you care about is the answer that is computed. For example, expressions 3+3+3+3 and 12 both denote the number 12, even if the shorter one, 12, is quicker to compute. In general, denotational semantics suppresses issues of performance of programs, concentrating on the answers that are ultimately produced. Part of the simplicity and elegance of a denotational semantics is that it takes you directly to the answer, ignoring the details of the intermediate steps.

49.3. Introduction to Lambda-calculus

Denotational semantics is a kind of model theory. But in order to have a viable model theory, you must start with a model, so an obvious first step in defining a denotational semantics is selection of an appropriate model. A key criterion for this selection is that the model must contain a value for each thing in the programming language that you want to describe. For simple expressions involving integers, the set of integers provides an excellent model.

But programming languages are much richer than that. Not only are there many different kinds of data values, such as lists and other data structures, but a particular concern is that some of the things that must be dealt with are not typical data values at all, but are functions. How can those be modeled? For example, if you write a factorial function in a program, you would like to provide a denotation for the factorial function itself, so that you can say exactly what the function that you have written *is* in the model. Obviously, the model has to contain the factorial function (or something that represents the factorial function) as one of its members.

There is another serious problem in choosing a model. The point of model theory is that the model must be understood ahead of time, and the language is then explained in terms of that model. What if each programming language uses a different model for its semantics? Then part of the semantics of the language must be a careful description of the model. That is, you need yet another semantics of the model. If every concept in the programming language has simply been added directly to the model, then the model offers no real clarification; its semantics will be just as difficult as the semantics of the programming language.

For denotational semantics truly to work well, some model must be chosen that (a) provides functions as values and (b) is rich enough that it can be used as the model for all of the concepts in almost any programming language. That is a tall order. Fortunately, there is such a model, called λ-calculus (lambda calculus). It was, in fact, the λ-calculus that lead to the programming language Lisp, of which Scheme is a dialect.

An intuitive introduction to λ-calculus

Lambda-calculus uses notation $\lambda x.e$ for the function that, when run on parameter x, produces value e. For example, the function that squares its argument is written $\lambda x.x \cdot x$. Function $\lambda x.x$ is the identity function. In $\lambda x.e$, x is called the formal parameter and e is called the body.

A function can be applied to an argument. To indicate function application (or simply *application*), write the function to the left of the argument. That is, juxtaposition indicates function application. For example $(\lambda x.x)\,5$ indicates application of function $\lambda x.x$ to argument 5. (The result is 5.)

Functions applications are evaluated by substitution. For example, $(\lambda x.x \cdot x)\,5 = 5 \cdot 5$ by substituting 5 for x in the body of the function. Substitution is called a *reduction* in λ-calculus terminology, and computation is done by performing a sequence of reductions until no more reductions are possible. When A reduces to B, we write $A \rightarrow B$. For example, $(\lambda x.x \cdot x)\,5 \rightarrow 5 \cdot 5$.

You can use λ notation and function application in arbitrary ways to build up expressions of λ-calculus. For example, $\lambda x.(x\,x)$ is the rather peculiar function that takes a parameter x (a function) and returns the result of applying function x to itself. By convention, application has higher precedence than dot, so $\lambda x.x\,x$ is read as if parenthesized $\lambda x.(x\,x)$, not as $(\lambda x.x)x$. Also, by convention, application associates to the left, so $A\,B\,C$ is understood as $(A\,B)C$.

Functions are values in λ-calculus, and a function can produce another function as its result. Function $\lambda f.(\lambda x.f(x))$, for example, takes parameter f (a function) and produces result $\lambda x.f(x)$ (also a function). Evaluating $(\lambda y.(\lambda r.r\,y))(5)$ by performing a substitution yields $\lambda r.r\,5$. Each of functions $\lambda f.(\lambda x.f(x))$ and $\lambda y.(\lambda r.r\,y)$ is *curried*, taking its parameters one at a time. Figure 49.1 shows two sample evaluations by substitution, also employing basic rules of arithmetic.

Be careful about parenthesizing terms. Function application is *not* associative. That is $(a\,b)\,c$ is not, in general, the same as $a\,(b\,c)$. It would not be correct to begin an evaluation as follows.

$$((\lambda f.(\lambda x.f(x))))\,\underline{(\lambda y.y \cdot y)\,6} \rightarrow ((\lambda f.(\lambda x.f(x))))\,(6 \cdot 6)$$

$$\begin{array}{rcl}
\underline{((\lambda f.(\lambda x.f(x))))\,(\lambda y.y\cdot y)}\,(6) & \to & \underline{(\lambda x.(\lambda y.y\cdot y)x)\,(6)} \\
 & \to & \underline{(\lambda y.y\cdot y)\,(6)} \\
 & \to & \underline{6\cdot 6} \\
 & \to & 36 \\
(\lambda x.(\lambda y.y+(x+1)))\,(9)\,\underline{((\lambda z.z)(12))} & \to & \underline{(\lambda x.(\lambda y.y+(x+1)))\,(9)}\,(12) \\
 & \to & (\lambda y.y+(\underline{9+1}))\,(12) \\
 & \to & \underline{(\lambda y.y+(10))\,(12)} \\
 & \to & \underline{(12)+(10)} \\
 & \to & 22
\end{array}$$

Figure 49.1: **Two evaluations by substitution.** At each step, the subexpression that is being replaced is underlined. The fundamental rule of computation in λ-calculus is that a term of the form $(\lambda x.E)(A)$ can be replaced by E', the result of replacing x in E by A. So $(\lambda x.x+x)(z) \to z+z$. We also use elementary rules of arithmetic to simplify some expressions.

Substituting $6\cdot 6 = 36$ and continuing this faulty evaluation yields term $\lambda x.36\,x$, which makes no sense, since it is using 36 as a function. In λ-calculus, juxtaposition indicates function application, not multiplication. Remember that juxtaposition is left-associative.

Lambda notation encourages curried functions by defining λ to associate to the right. For example, the function $((\lambda f.(\lambda x.f(x))))$ can be written with fewer parentheses as $\lambda f.\lambda x.f\,x$.

49.4. More detail on Lambda-calculus

The preceding section gave an informal introduction to λ-calculus. This section gives a more careful development of the fundamental ideas.

Terms: the syntax of λ-calculus

Expressions in λ-calculus are called *terms*, and are built using the following rules.

1. A variable is a term. Any identifier can be used as a variable; we will use x, y and z as variable names.

2. If x is a variable and e is a term, then $\lambda x.e$ is a term. This kind of term is called an *abstraction.*

3. If a and b are terms then $a\,b$ is a term. This kind of term is called an *application.*

Parentheses can be used freely to show structure. Application (juxtaposition) has higher precedence than abstraction; application associates to the left, and abstraction associates to the right.

Only the forms (1)-(3) are supported by pure λ-calculus. Sometimes, though, it is useful to imagine that some other kinds of terms are also available. For example, terms that are numeric constants (3, 42, etc.) and elementary operations of arithmetic such as addition and subtraction are useful, especially for illustrating the ideas of λ-calculus. We will use those kinds of terms for discussions, but will show how to do without them later.

Free and bound variables

An occurrence of a variable in a term is either *bound* or *free* or *formal.* A formal occurrence is one that occurs immediately after λ. For example, the x in $\lambda x.y$ is a formal variable. A bound occurrence of variable x is one that is not formal, but occurs within term e of larger term $\lambda x.e$. A bound variable refers to a function parameter, and will be substituted for when the function is applied. All other occurrences of variables are free. For example, in $\lambda x.y\,x$, variable y is free (since it does not occur inside any larger term of the form $\lambda y.e$) and the last x is bound (since it occurs with the scope of λx).

Lambda calculus allows *shadowing.* A bound variable x is said to be bound by the innermost abstraction that has that same variable as a formal parameter. So, in $\lambda x.(\lambda x.x)$, the bound occurrence of x is bound by the inner abstraction $\lambda x.x$, not by the outer, or first, one. (The inner formal x shadows the outer one.)

You can think of formal, bound and free variables in terms related to programming languages. A formal variable is like a formal parameter in a function definition. A bound variable is a formal parameter used inside a function definition. A free variable is like a global (or nonlocal) variable, not defined inside the function. There is no notion in λ-calculus that corresponds to a local variable that is neither a function parameter nor a global variable.

Reduction and computation

Computation involves substitutions. The substitution rule, called the rule of *β-reduction*, calls for replacing an an actual parameter for a formal parameter. Say that term $(\lambda x.e)a \to e'$ if e' is the result of replacing each free occurrence of x in e by (a). For example, $(\lambda y.y\,x)\,z \to (z)x$. Notice that the λy has been removed, and y has been replaced by (z) in term $y\,x$, yielding $(z)x$.

You must put parentheses around the substituted value unless leaving them out does not change the structure of the term. For example, $(\lambda x.x\,x)(y\,y) \to (y\,y)(y\,y)$. It would not be correct to say that $(\lambda x.x\,x)(y\,y) \to y\,y\,y\,y$, since $y\,y\,y\,y$ implicitly means $((y\,y)y)y$, not $(y\,y)(y\,y)$. Remember that juxtaposition is not an associative operator in λ-calculus, and that, by convention, it is left-associative.

There is an important restriction, discussed below, on when β-reductions can be done.

Conversions and equivalence of terms

Computation only goes one way; if $A \to B$, you convert A into B, not B into A. There is a notion of *equivalence* of terms, though, that is more general, and that is symmetric ($x = y$ just when $y = x$) so we need a symmetric way to convert between equivalent terms.

There are three kinds of conversions among terms of λ-calculus. If there is a conversion between two terms, the two terms are thought of as standing for the same thing, or as being equivalent, just as the terms 2+4 and 3+3 stand for the same value, even though they are different terms.

1. The names that you choose for formal parameters of functions are arbitrary, and do not affect the meaning of a term. Accordingly, the first kind of conversion, called an

α-conversion, consists of nothing but renaming a formal parameter. Write $s \equiv_\alpha t$ to indicate that s can be transformed into t by renaming a formal parameter of a function. For example, $\lambda x.x \equiv_\alpha \lambda y.y$.

More precisely, say that $\lambda x.e \equiv_\alpha \lambda y.e'$ if e' is the result of replacing each free occurrence of x in e by y. (The occurrences of x that are free in term e by itself are just those that are bound by λx in $\lambda x.e$.) There is an important restriction, discussed below, on when an α-conversion can be done.

2. The second kind of conversion is called a *β-conversion*, which is a symmetric version of β-reduction. Say that $s \equiv_\beta t$ provided either $s \rightarrow t$ or $t \rightarrow s$.

3. The third kind of conversion is less obvious than the first two, and is called an *η-conversion*. Consider term $\lambda x.f(x)$. On input x, this function produces result $f(x)$. That, of course, is exactly what function f does; on input x, function f produces result $f(x)$. So $\lambda x.f(x)$ is just the same function as f. Say that $\lambda x.f(x) \equiv_\eta f$ and, by symmetry, $f \equiv_\eta \lambda x.f(x)$.

In general, conversions are not required to be done on entire terms, but can also be done on subterms. Suppose, for example, that $a \equiv_\beta b$, and that a is a subterm of a larger term A. If B is the term that results from term A by replacing a by b, then we say that $A \equiv_\beta B$. Similar rules apply to the other conversions. For example $\lambda x.\lambda y.x + y \equiv_\alpha \lambda x.\lambda v.x + v$ because the α conversion $\lambda y.x + y \equiv_\alpha \lambda v.x + v$ can be done on subterm $\lambda y.x + y$.

The conversions discussed so far perform only one step. A more general notion of conversion is also needed that allows several steps to be performed. Say that $A \equiv B$ (A is can be converted to B, or A is *equivalent* to B) if there is a sequence of terms A_0, A_1, $\ldots$, A_n, for $n \geq 0$, where $A_0 = A$, $A_n = B$ and for $i = 1, \ldots, n$, either $A_{i-1} \equiv_\alpha A_i$ or $A_{i-1} \equiv_\beta A_i$ or $A_{i-1} \equiv_\eta A_i$. That is $A \equiv B$ if A can be converted to B by a sequence of zero or more basic conversions. Also, it is convenient to extend the notion of α-conversion to allow a sequence of conversions, so we write $A \equiv_\alpha B$ if A can be converted to B by doing one or more α conversions.

Say that $A \rightarrow^* B$ (A reduces to B) if there is a sequence of terms $A_0, A_1, \ldots, A_n$ where $A_0 = A$, $A_n = B$ and for $i = 1, \ldots, n$, either $A_{i-1} \equiv_\alpha A_i$ or $A_{i-1} \rightarrow A_i$.

Restrictions on reductions and conversions

There is an important restriction on when α-conversions and β-reductions can be done. Consider term $\lambda x.\lambda y.x(y)$. If parameter x is renamed to y, you get $\lambda y.\lambda y.y(y)$. That is not the same thing, since now both bound occurrences of y are bound by the inner λ. This situation is called a *capture*. The inner function $\lambda y.x(y)$ captures x for itself when x is replaced by y.

Capture can also occur as the result of substitutions for formal parameters. Consider function $f = \lambda x.\lambda y.x + y$, a curried function that produces the sum of its two parameters. Imagine that you have, in your environment, a variable y whose value is 40. You would like to compute $y + 2$, so you write $(f\ y)\,2$, which should yield 42. But f is the function $\lambda x.\lambda y.x + y$, and substituting y for x in the body $\lambda y.x + y$ of f yields $f\ y = \lambda y.y + y$. Applying that function to 2 yields $2 + 2$, or 4. The dependence on the variable y in the environment has disappeared! The problem is that the substitution put the variable y from the environment into a context where it was really a different thing; it was captured, and became the parameter of the function.

(Capture, or the lack of capture, really, shows up in programming languages, and it often results in confusion among beginning programmers. If two functions each have a local variable called x, a beginning programmer might presume that storing a value in one of those variables (say, inside function f) will cause another one of those variables (say, inside function g) to change. The beginning programmer is hoping that one function captures the other's variable, and is surprised when it is does not.)

You are only allowed to perform α-conversions and β-reductions when they do not result in any capture. A free variable must not become bound, and a bound variable must not become bound by a different λ. If a β-reduction would result in a capture, you must rename variables, by doing α-conversions, before performing the β-reduction. For example,

$$\begin{aligned} (\lambda x.\lambda y.x + y)\ y\ 2 &\equiv_\alpha (\lambda x.\lambda z.x + z)\ y\ 2 \\ &\to (\lambda z.y + z)\ 2 \\ &\to y + 2 \end{aligned}$$

49.5. Fundamental data types

Integers and elementary operations

Recall that the motivation for studying λ-calculus is to present a model in which a large variety of concepts are already present. It is not acceptable just to add each new concept to the model whenever a new concept is needed. Instead, new concepts should be modeled in terms of old concepts.

Until now, we have presumed that integer constants and arithmetic operations were available as primitive things. But, really, we should look for integers already present, in terms of what has already been defined. Although pure λ-calculus has no integers per se, it does have terms that can act as stand-ins for integers, and so can model integers. For simplicity, we will only explore nonnegative integers by defining, for each nonnegative integer n, a term $\bar{n}$ of λ-calculus that represents n. Term $\bar{0}$ models the number 0, $\bar{1}$ models the number 1, etc. Here are the first few of what are called the *Church numerals.*

$$\begin{aligned} \bar{0} &= \lambda f.\lambda x.x \\ \bar{1} &= \lambda f.\lambda x.f\,x \\ \bar{2} &= \lambda f.\lambda x.f(f\,x) \\ \bar{3} &= \lambda f.\lambda x.f(f(f\,x)) \\ \bar{4} &= \lambda f.\lambda x.f(f(f(f\,x))) \end{aligned}$$

The pattern continues. Using $f^n(x)$ to stand for $f(f(...(f(x))))$ (with n occurrences of f), the term $\bar{n}$ is $\lambda f.\lambda x.f^n(x)$. Notice that $\bar{n}(f)(x) = f^n(x)$.

To say that the Church numerals are reasonable stand-ins for integers, we need to be able to define basic operations on integers, making them work in the model. One of the most basic is the successor function, which adds one to an integer. It can be defined by

$$\text{succ} = \lambda m.\lambda f.\lambda x.f(m\,f\,x). \tag{49.5.1}$$

To demonstrate that the definition of succ is correct, it suffices to show that $\text{succ}(\bar{n}) = \overline{n+1}$. Subterms that are about to be reduced are underlined.

$$\text{succ}(\bar{n}) \quad = \quad \underline{(\lambda m.\lambda f.\lambda x.f(m\,f\,x))(\bar{n})}$$

$$\begin{aligned}
&\to && \lambda f.\lambda x.f(\bar{n} f\, x)\\
&= && \lambda f.\lambda x.f(\underline{(\lambda u.\lambda v.u^n(v))\, f}\, x)\\
&\to && \lambda f.\lambda x.f(\underline{(\lambda v.f^n(v))x})\\
&\to && \lambda f.\lambda x.f(f^n(x))\\
&= && \lambda f.\lambda x.f^{n+1}(x))\\
&\equiv_\alpha && \overline{n+1}
\end{aligned}$$

To define the addition function, *plus*, remember that $\bar{n} f$ is the same as f^n; it calls for applying function f a total of n times. (We are using exponentiation of *functions*, not of numbers.) Since $f^{n+m}(x) = f^n(f^m(x)) = (\bar{n} f)(\bar{m} f x)$, a reasonable definition of plus is

$$\text{plus} = \lambda n.\lambda m.\lambda f.\lambda v.(n\, f)(m\, f\, v).$$

To show that this works, try it with a general $\bar{n}$ and $\bar{m}$.

$$\begin{aligned}
\text{plus}\ \bar{n}\ \bar{m} &= && \underline{(\lambda n.\lambda m.\lambda f.\lambda v.(n\, f)(m\, f\, v))\ \bar{n}}\ \bar{m}\\
&\to && \underline{(\lambda m.\lambda f.\lambda v.(\bar{n}\, f)(m\, f\, v))\ \bar{m}}\\
&\to && \lambda f.\lambda v.(\bar{n}\, f)(\bar{m}\, f\, v)\\
&= && \lambda f.\lambda v.(\bar{n}\, f)(\underline{(\lambda x.\lambda y.x^m(y))\, f}\, v)\\
&\to && \lambda f.\lambda v.(\bar{n}\, f)(\underline{(\lambda y.f^m(y))\, v})\\
&\to && \lambda f.\lambda v.(\bar{n}\, f)(f^m(v))\\
&= && \lambda f.\lambda v.(\underline{(\lambda x.\lambda y.x^n(y))\, f})(f^m(v))\\
&\to && \lambda f.\lambda v.\underline{(\lambda y.f^n(y))(f^m(v))}\\
&\to && \lambda f.\lambda v.f^n(f^m(v))\\
&= && \lambda f.\lambda v.f^{n+m}(v)\\
&\equiv_\alpha && \overline{n+m}
\end{aligned}$$

Integer multiplication is implemented by function *product*, defined as follows.

$$\text{product} = \lambda m.\lambda n.\lambda f.m(nf) \tag{49.5.2}$$

That is, perform a nested loop; repetition n times is done m times.

Predecessor and subtraction

Subtraction is tricky to do. The starting point is a function *pred* (for predecessor) that subtracts one from a number. Since we are only handling nonnegative numbers, define $\text{pred}(\bar{0}) = \bar{0}$. The intuition is to write a loop whose body is performed n times, but that produces $f^{n-1}(x)$. Such a loop might go something like this.

```
A = x;
B = x;
for i = 1,..., n do
  B = A;
  A = f(A);
end for
```

At the end of the loop, B contains $f^{n-1}(x)$ (or just x if $n = 0$). The two variables A and B can be encoded as an ordered pair, and the body of the loop can be carried out by a function that takes one ordered pair and produces the next one. That requires a function *pair* for creating ordered pairs (where (pair x y) represents the ordered pair (x, y)), and functions *left* and *right* for getting the parts of a pair (where left(pair x y) $\equiv x$ and right(pair x y) $\equiv y$). Those functions are defined later in this section.

Suppose that $g(x, y) = (f(x), x)$, where g deliberately ignores y. The parameter of g is an ordered pair, as encoded by function *pair*. Then

$$\begin{aligned} g(x,y) &= (f(x), x) \\ g(g(x,y)) &= (f(f(x)), f(x)) \end{aligned}$$

and, in general, $g^n(x, y) = (f^n(x),\ f^{n-1}(x))$, so $f^{n-1}(x) = \text{right}(g^n(x, y)) = \text{right}(\bar{n}\, g\, (x, y))$. The following definition of pred uses that idea. Since the value of $g^n(x, y)$ is independent of y, it is just as well to compute $g^n(x, x)$, and that is what is done. Function g is not written directly, but is produced as the result of another function; g is predhelp(f).

$$\begin{aligned} \text{predhelp} &= \lambda f.\lambda p.\text{pair}\ (f(\text{left}\ p))\ (\text{left}\ p) \\ \text{pred} &= \lambda n.\lambda f.\lambda x.\text{right}(n\ (\text{predhelp}\ f)\ (\text{pair}\ x\ x)) \end{aligned}$$

Subtraction can be defined by repeatedly taking the predecessor. It is left as an exercise.

Structured data

An ordered pair can be constructed by a function that takes three parameters. The first two parameters are the two members of the pair. The third parameter is a selector, which tells which member of the pair is desired. The pair constructor, *pair*, is defined as follows.

$$\text{pair} = \lambda x.\lambda y.\lambda s.s\,x\,y \tag{49.5.3}$$

Since pair is curried, we do not need to give it all of its parameters right away. Term (pair 5 7) models the ordered pair (5,7). Evaluation yields

$$\begin{aligned} \text{pair}\ 5\ 7 &= (\lambda x.\lambda y.\lambda s.s\ x\ y)\ 5\ 7 \\ &\to (\lambda y.\lambda s.s\ 5\ y)\ 7 \\ &\to (\lambda s.s\ 5\ 7) \end{aligned}$$

The pair is represented by a function that is waiting for a selector s. Notice that the selector is applied to 5 and 7, in that order, in the body s 5 7. Here are two functions, *left* and *right*, that take the left and right-hand member of an ordered pair, respectively, by supplying the pair with a suitable selector s.

$$\begin{aligned} \text{left} &= \lambda p.p(\lambda x.\lambda y.x) \\ \text{right} &= \lambda p.p(\lambda x.\lambda y.y) \end{aligned}$$

For example,

$$\begin{aligned} \text{left}(\text{pair}\ 5\ 7) &\to^* (\lambda p.p(\lambda x.\lambda y.x))(\lambda s.s\ 5\ 7) \\ &\to (\lambda s.s\ 5\ 7)(\lambda x.\lambda y.x) \\ &\to (\lambda x.\lambda y.x)\ 5\ 7 \\ &\to (\lambda y.5)\ 7 \\ &\to 5 \end{aligned}$$

Modeling lists is left as an exercise.

49.6. Control structures

We have seen that λ-calculus can model basic data values. But can it provide control structures, such as if-then-else and recursion? It turns out that both of those can be modeled without adding anything new. Conditionals are fairly simple to model. Recursion is more involved, and is covered in the next section.

Conditionals and boolean values

The conditional function *cond* is defined so that (cond $x\,y\,z$) is y when x is true, and is z when x is false. The conditional function and associated boolean values *true* and *false* can be defined as follows.

$$\begin{array}{rcl} \text{cond} & = & \lambda x.\lambda y.\lambda z.x\,y\,z \\ \text{true} & = & \lambda x.\lambda y.x \\ \text{false} & = & \lambda x.\lambda y.y \end{array}$$

(Compare with pair, left and right.) For example,

$$\begin{array}{rcl} \text{cond true } M\ N & = & (\lambda x.\lambda y.\lambda z.x\ y\ z) \text{ true } M\ N \\ & \rightarrow & (\lambda y.\lambda z.\text{true } y\ z)\ M\ N \\ & \rightarrow & (\lambda z.\text{true } M\ z)\ N \\ & \rightarrow & \text{true } M\ N \\ & = & (\lambda x.\lambda y.x)\ M\ N \\ & \rightarrow & (\lambda y.M)\ N \\ & \rightarrow & M \end{array}$$

49.7. Recursion

Lambda-calculus provides no direct way to make recursive definitions. You are always free to give names to terms, as we have done for values such as false, true and cond, but you cannot use a name before it is defined.

Recursion amounts to solving equations

For ease of understanding, suppose that integers and a few basic operations on integers are already implemented. Write $m * n$ for the product of m and n and $m - n$ for their difference. Also assume that a predicate that tests whether a given number is zero has been implemented. For clarity, that test is written $(n = 0)$ (Expressing those functions is left as an exercise.) Now suppose that you want to define the factorial function as

$$\text{factorial}(n) = \text{cond } (n = 0)\ 1\ (n * \text{factorial}(n - 1))$$

That would be fine, except that it is not allowed by λ-calculus, since the definition uses factorial before it is defined. Another way to express the definition of factorial is

$$\text{factorial} = \lambda n.\text{cond } (n = 0)\ 1\ (n * \text{factorial}(n-1)) \tag{49.7.1}$$

but that is still not allowed. But generalize Equation (49.7.1) to the following equation involving a variable X that stands for an unknown function.

$$X = \lambda n.\text{cond } (n = 0)\ 1\ (n * X(n-1)) \tag{49.7.2}$$

Clearly, $X = \text{factorial}$ is a solution of Equation (49.7.2). (Substitute $X = \text{factorial}$ and you get equation (49.7.1).) So the problem of defining the factorial function can be expressed as finding a function X that satisfies Equation (49.7.2), and, in general, recursive definitions can be viewed as finding solutions to certain equations, where the solutions are functions. All we need is a way to solve those equations.

Fixed-point iterations

Computers are frequently used to solve equations. One way that is commonly employed is to express the equation in the form $x = f(x)$. That is, find a value x, called a *fixed point* of f, that f maps to itself. An example is computation of square roots. If $f(x) = (x + n/x)/2$, then it is easy to check that $\sqrt{n}$ is a fixed point of f. That is, $f(\sqrt{n}) = \sqrt{n}$. You can get better and better approximations to $\sqrt{n}$ by defining $x_0 = n$ and $x_i = f(x_{i-1})$ for $i > 0$. So $x_1 = f(x_0)$, $x_2 = f(f(x_0))$, etc. Suppose, for example, that $n = 9$. The first few values x_0, x_1, ... are

$$\begin{array}{lcl} x_0 & = & 9 \\ x_1 & = & 5 \\ x_2 & = & 3.4 \\ x_3 & \approx & 3.0235 \end{array}$$

The correct answer, of course, is the square root of 9, or 3. As more and more values in the sequence are computed, the answer gets closer and closer to 3. The starting value x_0 is not terribly important, except that starting values closer to the answer cause the solution to converge to a correct answer faster.[1]

The same general approach can be used to find fixed points where the fixed point is not a number but a function. To compute the factorial function, define the function

$$H = \lambda f.\lambda n.\text{cond } (n = 0)\ 1\ (n * f(n-1)) \tag{49.7.3}$$

Equation $X = H(X)$ is the same as equation 49.7.2. So the factorial function is a fixed point of H. Suppose that a function f_0 is chosen as a first approximation of factorial. Exactly what f_0 does is irrelevant, and will have no bearing on the final answer. Compute $f_1 = H(f_0)$, $f_2 = H(f_1) = H(H(f_0))$, $f_3 = H(f_2) = H(H(H(f_0)))$, etc. For example,

$$f_1 = \lambda n.\text{cond } (n = 0)\ 1\ (n * f_0(n-1))$$

[1] Once you get close, this particular fixed-point iteration converges very fast, giving you about twice as many digits of precision at each step as you had at the previous step.

and $f_1(0) = 1$, regardless of what f_0 does. Since

$$f_2 = \lambda n.\text{cond } (n = 0)\ 1\ (n * f_1(n-1))$$

you can see that $f_2(1) = 1 * f_1(0) = 1$. Continuing shows that $f_3(2) = 2$ and $f_4(3) = 6$. A table of $f_4(n)$ for natural numbers n shows that $f_4(0) = 1$, $f_4(1) = 1$, $f_4(2) = 2$, $f_4(3) = 6$, and $f_4(n)$ depends on what f_0 does, for $n > 3$. As more and more copies of H are applied, the resulting function computes more and more of the factorial function. Function $f_6 = H(H(H(H(H(H(f_0))))))$ gets the correct values on all parameters up to 5, regardless of what f_0 is. To get the full factorial function would require performing H infinitely many times. But that is not really necessary. If you want to compute factorial(5), it suffices to compute function f_6 and to apply that function to 5. The answer 120 comes out in a finite amount of time. In fact, to compute factorial(5), you can use function f_k for any $k > 5$.

Recursion is handled in λ-calculus by creating a function, fix, that finds a fixed point of a function H by repeatedly applying H. For any function F, fix(F) is a fixed point of F. So $F(\text{fix}(F)) = \text{fix}(F)$. To get the factorial function, you define H by equation 49.7.3 and define factorial to be fix(H). The fix function can be defined as follows.

$$\text{fix} = \lambda f.(\lambda x.f(x\,x))(\lambda x.f(x\,x)) \tag{49.7.4}$$

Notice that, for any function F,

$$\text{fix}\,F \to (\lambda x.F(x\,x))(\lambda x.F(x\,x)) \tag{49.7.5}$$

Performing another β-reduction yields

$$\text{fix}\,F \to^* F((\lambda x.F(x\,x))(\lambda x.F(x\,x)))$$

Now, using (49.7.5) backwards yields

$$\text{fix}\,F \equiv F(\text{fix}\,F).$$

So (fix F) is a fixed point of F. The fix function just does repeated evaluation. Using the fixed point property, $\text{fix}\,F \equiv F(\text{fix}\,F) \equiv F(F(\text{fix}\,F)) \equiv F(F(F(\text{fix}\,F)))\ldots$ If you think of the inner (fix F) as the function f_0, you can see how the fixed point is found.

Recursion is a powerful tool, and, using fix, you can get the effect of recursive definitions in λ-calculus, even though recursion is not supported directly. It is generally believed that λ-calculus is powerful enough to express any function that can be expressed in any (realistically implementable) programming language. That observation is attributed to Alonzo Church, the creator of λ-calculus.

> **Church's Thesis.** All algorithmically computable functions are expressible in λ-calculus.

So λ-calculus models the kinds of things that are needed for programming languages, and is capable of expressing any function that can be defined in any language.

49.8. Evaluation order

How do you know when a computation in λ-calculus is finished? Say that a term of λ-calculus is in *normal form* if there are no β-reductions that can be applied to it. (If there is a β-reduction that is blocked because it would cause a capture, then the term is not in normal form. Appropriate renaming must be done, and the β-reduction carried out.) The goal of computation is to convert a term to normal form, called *normalizing* the term.

A term can be evaluated in various orders. Inside-out, or eager, evaluation is commonly employed. But outside-in evaluation, and lazy evaluation, can also be used. Evaluation order is partly a matter of efficiency; some orders can lead to a normal form in fewer steps than other orders. But your choice of evaluation order can also determine whether you will *ever* reach a normal form. The problem is that some terms have no normal form. Consider term $L = (\lambda x.x\,x)(\lambda x.x\,x)$. Performing a β-reduction shows that $L \to L$. Any attempt to evaluate L by performing β-reductions will never terminate. Now suppose that you want to evaluate term (cond true $\bar{0}$ L). If you use the rule for cond first, you get answer $\bar{0}$, which is in normal form. But if instead you decide to evaluate L first, you never reach that answer, and get stuck in an infinite computation.

The question to be answered is, does there exist an evaluation strategy that finds a normal form whenever one exists? There is, and a simple one is left-to-right, outside-in evaluation. Define a *redex* (short for reducible expression) to be any term of the form $(\lambda x.e)(a)$, which can be replaced by performing a β-reduction. Suppose that a term contains more than one redex. An *outermost* redex is one that is not contained in any other redexes as a subterm. *Normal order* evaluation always selects the leftmost outermost redex, and performs a β-reduction on that subterm. Normal order evaluation reaches a normal form whenever it is possible to reach a normal form. (Proof of this fact is left to more in-depth studies of λ-calculus.)

The remainder of this chapter gives an equational definition of an evaluator that converts a term to normal form using normal order evaluation. The evaluator forms the basis of an operational semantics for λ-calculus, since it indicates how to perform computations, step by step. But the equational nature of the definition means that the steps are implicit, not explicitly stated as they would be using rules of inference. For simplicity, some of the details of the evaluator, such as dealing with capture, are only stated, not written out using equations.

Head-normal form

Head-normal form is a form that is not quite normal form, but is a good start toward it. Say that a term is in head-normal form if it has the form $\lambda x_1.\lambda x_2.\ldots\lambda x_m.y t_1 \ldots t_n$ for $m \geq 0$ and $n \geq 0$, where y is a variable and $t_1, \ldots, t_n$ are arbitrary terms. That is, it consists of zero or more λ's, then a variable name (which can be one of $x_1, \ldots, x_m$ or can be something else) then a sequence of zero or more terms $t_1, \ldots, t_n$. For example, $\lambda x.x((\lambda y.y)z)$ is in head-normal form, with $m = 1$ and $n = 1$. Term t_1 is $((\lambda y.y)z)$. You are not required to have any λ's. For example, term xy is in head-normal form. (It is also in normal form.)

The normal form of $\lambda x_1.\lambda x_2.\ldots\lambda x_m.y\,t_1 \ldots t_n$, if it exists, is clearly $\lambda x_1.\lambda x_2.\ldots\lambda x_m.y\,v_1 \ldots v_n$, where v_i is the normal form of t_i, for $i = 1, \ldots, n$. No reductions can disturb the outer structure; they must all happen within the terms $t_1, \ldots, t_n$. (Application associates to the left. After evaluating terms $t_1, \ldots, t_n$, we have $\lambda x_1.\lambda x_2.\ldots\lambda x_m.(((y\,v_1)\,v_2)\ldots v_n)$. There are no β-reductions to do.) So to normalize a term u it suffices to convert u to head-normal form $\lambda x_1.\lambda x_2.\ldots\lambda x_m.y t_1 \ldots t_n$, and then to normalize each of $t_1, \ldots, t_n$.

Figure 49.2 gives an equational definition of function hnf, which converts a term to head-normal form. To understand how it works, imagine a term that is not in head-normal

$$
\begin{array}{rcll}
\mathrm{hnf}(\lambda x.t) & = & \lambda x.\mathrm{hnf}(t) & \text{(abstraction)} \\
\mathrm{hnf}(t_1\ t_2) & = & \mathrm{hnfhelp}(\mathrm{hnf}(t_1)\ t_2) & \text{(application)} \\
\mathrm{hnf}(x) & = & x & \text{(variable)} \\
\mathrm{hnfhelp}((\lambda x.t)z) & = & \mathrm{hnf}(t/\{x \leftarrow z\}) & \\
\mathrm{hnfhelp}(r\ s) & = & r\ s & (r \text{ not an abstraction})
\end{array}
$$

Figure 49.2: **Definition of function** hnf**, which converts a term to head-normal form.** Notation $t/\{x \leftarrow z\}$ indicates the result of substituting z for all free occurrences of x in t. The first equation for hnfhelp presumes that no capture results from this substitution. If a capture would result, you must rename variables before performing the substitution.

$$
\begin{array}{rcll}
\mathrm{nf}(t) & = & \mathrm{nfhelp}(\mathrm{hnf}(t)) & \\
\mathrm{nfhelp}(\lambda x.t) & = & \lambda x.\mathrm{nfhelp}(t) & \text{(abstraction)} \\
\mathrm{nfhelp}(t_1\ t_2) & = & (\mathrm{nfhelp}(t_1))\ (\mathrm{nf}(t_2)) & \text{(application)} \\
\mathrm{nfhelp}(x) & = & x & \text{(variable)}
\end{array}
$$

Figure 49.3: **Definition of function** nf**, which converts a term to normal form.**

form. If you eliminate all redundant parentheses, then you must have a term of the form $\lambda x_1.\lambda x_2.\ldots\lambda x_m.r\,t_1\ldots t_n$, for $m \geq 0$ and $n \geq 0$, where term r is an abstraction $(\lambda y.a)$. (If r is an application, then it is not the first thing in the list after removing redundant parentheses. If r is a variable, then this term is in head-normal form.) So a term that is not in head-normal form must have the form $\lambda x_1.\lambda x_2.\ldots\lambda x_m.(\lambda y.a)t_1\ldots t_n$, where $n > 0$. Conversion to head-normal form is just a matter of performing a β-reduction of $(\lambda y.a)t_1$ and recursively converting the result to head-normal form. That is what the definition of hnf does.

Normal order evaluation

Conversion to normal form is completed by converting each of the terms $t_1, \ldots, t_n$ to normal form. Figure 49.3 shows a definition of function nf, which converts a term to normal form (or runs forever trying).

Confluence

Reduction using normal order is guaranteed to find a normal form if one exists, but it does not necessarily give the most efficient route to the normal form. It uses outside-in evaluation, which often requires evaluating a term more than once. Suppose that you decide to use a different strategy, such as inside-out evaluation (which evaluates an application $(\lambda x.A)\,B$ by giving preference to redexes in B before substituting it into A), and then decide that doing so was not such a good idea. To find a normal form, will you have to undo what you have done, and start back at the beginning with normal order? It turns out that you never need to back up. Lambda-calculus has a property called *confluence*.

> **Confluence property.** Suppose that term t has a normal form n. All evaluation strategies that yield a normal form yield the same normal form n. Moreover, if t reduces to t' by a sequence of α and β reductions, then there is a way to reduce t' to n by a sequence of α and β reductions.

The confluence property shows that you can never get into a blind alley and need to back up. You can always switch to head-normal evaluation at a later time if you need to.

49.9. Summary of terminology and concepts

An operational semantics of expressions describes the details of how an expression is evaluated. Denotational semantics associates a denotation with each expression. The denotation of an expression is defined in terms of the denotations of its subexpressions and the environment, or state (giving values of variables) in which the denotation is evaluated.

Denotational semantics is a kind of model theory. In a model theory, each thing in one domain (such as a programming language) is explained as being modeled by something in another domain (the model). In order to use denotational semantics effectively, a general purpose model of programming languages needs to be available. Lambda-calculus is such a model. Lambda-calculus has a simple description and only two constructs, one that builds a function (abstraction) and one that uses a function (application), but it has a surprisingly rich structure. It is so rich that every computable function can be expressed in λ-calculus.

Terminology

$A \rightarrow B$ In an operational semantics, relationship $A \rightarrow B$ indicates that expression A can be converted to expression B by one step of evaluation. In the specific setting of λ-calculus, $A \rightarrow B$ means that A can be converted to B by doing one β-reduction.

$A \rightarrow^* B$ In λ-calculus, $A \rightarrow^* B$ if there is a sequence of zero or more steps of α-conversion and β-reduction that will convert A into B.

α-conversion An α-conversion is a renaming of a formal parameter in a term.

β-reduction A β-reduction is a substitution of an actual parameter for a formal parameter in a function application, where the function being applied is an abstraction.

β-conversion Term s β-converts to term t if either s β-reduces to t or t β-reduces to s.

η-conversion Term f and $\lambda x.f(x)$ are equivalent by η-conversion.

λ-calculus Lambda-calculus is a formalism that models higher order functions. It is extremely versatile, capable of expressing any computable higher order function.

Abstraction In λ-calculus, a term of the form $\lambda x.e$ is called an abstraction.

Application In λ-calculus, a term of the form AB, where A and B are terms, is called an application.

Bound variable A bound variable is a variable inside a term that refers to one of the parameters of a function. For example, an occurrence of x in term e is bound in term $\lambda x.e$.

Capture A variable becomes captured as the result of a substitution or renaming if either it was unbound before the substitution but became bound, or it became bound by a different λ as a result of the substitution.

Confluence Because of confluence, you never need to undo a β-reduction to reach a normal form. There are no dead-ends that you can get caught in where you cannot reach the normal form.

Conversion Two terms of λ-calculus are considered equivalent (standing for the same thing) if it is possible to convert one to the other. A conversion is a sequence of α, β and η-conversions.

Denotation The denotation of an expression is the value that the expression has, when you evaluate it.

Denotational semantics Denotational semantics is a semantics in which each thing being described is given a meaning in a model.

Environment An environment is a binding of names to values. It is used to evaluate terms that involve variables.

Fix The fix function finds fixed points of functions. For all f, $f(\text{fix}\, f) = \text{fix}\, f$.

Fixed point A fixed point of a function f is a value x such that $f(x) = x$.

Formal variable A formal variable is a variable occurrence that occurs immediately after λ.

Free variable A free variable in a term is a variable that is not bound and not formal.

Head-normal form Head-normal form is a form that is not quite normal form, but is a step on the way toward normal form. A term in head-normal form looks like $\lambda x_1.\lambda x_2.\ldots\lambda x_m.yt_1\ldots t_n$ for $m \geq 0$ and $n \geq 0$, where y is a variable.

Model theory A model theory uses a model to explain concepts of another setting, such as a programming language.

Normal form A term of λ-calculus is in normal form if there are no β-reductions that can be applied to it (and none that are blocked by the capture rule but will be enabled by renaming variables).

Normal order evaluation Normal order evaluation chooses a leftmost, outermost term for evaluation.

Normalization Normalization is the process of converting a term to normal form.

Operational semantics An operational semantics of expressions describes the steps involved in converting an expression to normal form.

Redex A redex is a subterm of the form $(\lambda x.a)(b)$.

Reduction A reduction is a special kind of conversion. It substitutes a value for a formal parameter.

Term Expressions of λ-calculus are called terms.

49.10. Exercises

49.1. Write a definition of the denotation of expression `if` A `then` B `else` C, which produces B when A is true and produces C when A is false. Use $D(E, \eta)$ notation.

49.2. An alternative notation for the denotation of an expression uses a relation, $\Rightarrow$, where $e \Rightarrow v$ indicates that the denotation of expression e is v. An extension includes an environment: write $\eta \vdash e \Rightarrow v$ to indicate that, when variables have the values indicated in environment η, the denotation of expression e is v. For example, $\{x = 5\} \vdash x + 1 \Rightarrow 6$.

Facts about $\Rightarrow$ can be expressed using rules of inference. For example, a rule for addition is

$$\frac{\eta \vdash a \Rightarrow u,\ \ \eta \vdash b \Rightarrow v}{\eta \vdash a + b \Rightarrow u + v}$$

Using this notation, write rules of inference for the denotation of n, $a * b$, x and (let $x = a$ in b), where x is a variable (bound by the environment), a and b are expressions, and n is a numeral.

49.3. Add parentheses to each of the following terms to show their structure. Add enough parentheses so that no rules of precedence or associativity are required to understand them.

(a) $\lambda x.x\,x\,y$

(b) $\lambda x.\lambda y.y\,x$

(c) $\lambda x.x\,\lambda y.y$

49.4. By carefully performing β-reductions, show that succ($\bar{2}$) $\to^* \bar{3}$, using function succ defined in Equation (49.5.1).

49.5. Show that (product $\bar{2}\ \bar{3}$) $\to^* \bar{6}$, using Equation (49.5.2) as the definition of product.

49.6. Reduce each of the following to normal form. Equation (49.5.3) defines pair.

(a) $\bar{0}\,\bar{0}$.

(b) $\bar{1}\,\bar{0}$.

(c) pair $\bar{0}\ \bar{2}$.

49.7. Let fix be the fixed-point operator defined in equation 49.7.4.

(a) Evaluate fix g using normal order evaluation. (Here, g is a free variable, and has no value, so the result must still have g in it.)

(b) Show that if fix g is evaluated using inside-out evaluation instead of normal order evaluation, the evaluation will not terminate.

(c) An alternative fixed-point operator fix' is defined by $\text{fix}' = \lambda f.(\lambda x.f(\lambda y.x\,x\,y))(\lambda x.f(\lambda y.x\,x\,y))$. Show that evaluation of $\text{fix}'\,g$ by inside-out evaluation terminates. (Operator fix' has added machinery to force evaluation to be delayed until it is needed.)

49.8. Show that function fix' from the preceding exercise is a fixed point operator. That is, $f(\text{fix}'f) = \text{fix}'f$.

49.9. Give a definition of a function z in λ-calculus so that $z(\bar{0}) = \text{true}$ and $z(\bar{n}) = \text{false}$ for all integers $n > 0$. (This is the function that tests whether a number is 0.) (Hint: You might find it useful to write a function that always produces false. Notice that if F is defined so that $F(x) = \text{false}$ for every x, then $F^n(x) = \text{false}$ for every $n > 0$. Try computing $\bar{n}Fv$. What is the answer when $n = 0$? What is it when $n > 0$?)

49.10. Define the subtraction function diff in λ-calculus so that $\text{diff}\ \bar{n}\ \bar{m} = \overline{n-m}$ when $n \geq m$, and $\text{diff}\ \bar{n}\ \bar{m} = \bar{0}$ when $n < m$. (Hint: It suffices to take n predecessors of m. Keep in mind that $\bar{n}$ is designed to repeat an operation n times, so there is a built-in loop mechanism.)

49.11. Give a definition of the exponentiation function $\text{pow}\ \bar{n}\ \bar{m} = \overline{n^m}$ for nonnegative integers in λ-calculus. (There is a very short definition. See if you can find it.)

49.12. Define an equality predicate that works on numbers. So equal $\bar{n}\ \bar{n}$ should produce true, and equal $\bar{n}\ \bar{m}$ should produce false when $n \neq m$.

49.13. Show how to model lists in λ-calculus. Model the empty list, and functions null, ::, head and tail. (null(x) = true if x is the empty list, and false otherwise.) Use the implementation of pairs as a guide. Since there are two kinds of lists (empty and nonempty), you might find a tag useful. A boolean value makes a convenient tag. How can you attach a tag to a value?

49.14. You can define the fix function in Cinnameg as follows.

```
Define fix f x = f (fix f) x.
```

Define function H whose fixed point is the factorial function, and use fix to get its fixed point. (But call it h, since Cinnameg prefers lower case names.) Try it to see whether it computes factorials.

49.15. Suppose the preceding problem uses the following definition.

```
Define fix f = f(fix f).
```

That does not work. Why not? What effect does the extra parameter x have? Is there an easy way to modify the new definition to make it work, without introducing the extra parameter x?

49.16. The capture problem makes implementation of an interpreter for λ-calculus awkward. One approach to reducing the difficulty is to use a representation that does not give names to formal parameters at all. Instead, each bound variable is replaced by a nonnegative integer, indicating how many λ's need to be skipped over, moving outwards from the location of the number, to find the λ that binds this variable. For example, term $\lambda x.\lambda y.xy(\lambda z.xz)$ is represented as $\lambda.\lambda.1\,0(\lambda.2\,0)$. Use this representation of λ-terms to write a normal-order evaluator for λ-calculus.

Be careful, and try some examples by hand first. When you perform a substitution, some of the variables move into different contexts, and need their numbers changed. (Remember that a β-reduction can be performed on a subterm of a larger term, and the rest of the term provides the context.) You will find the following function useful. It modifies variable numbers in a term by adding n to each free variable number that is larger than m. k is a term that is variable number k.

$$\begin{aligned}
\text{renum}(m,n,k) &= k \text{ when } k < m\\
\text{renum}(m,n,k) &= k+n \text{ when } k >= m\\
\text{renum}(m,n,\lambda.t) &= \lambda.\text{renum}(m+1,n,t)\\
\text{renum}(m,n,s\,t) &= \text{renum}(m,n,s)\ \text{renum}(m,n,t)
\end{aligned}$$

49.17. Give rules of inference that describe an operational semantics showing how to evaluate a term of λ-calculus. The rules should define a single-step relation $A \to B$ that involves performing one β-reduction. Write the semantics in two different ways.

(a) Use call-by-value, where a parameter is converted to normal form before passing it to a function. For example, to evaluate $(\lambda x.A)(B)$, first evaluate B. To evaluate $A\,B$, evaluate A first, up to the point where it is an abstraction.

(b) Use normal order evaluation, where outermost reductions are done first. That is, to evaluate $(\lambda x.A)(B)$, perform the reduction (replacing free occurrences of x in A by B) before evaluating B. To evaluate $A\,B$, evaluate A first until it is an abstraction.

49.11. Bibliographic notes

Gordon [43] Mitchell [79], Pagan [84], Scott and Strachy [95], Schmidt [93], Stoy [101] and Winskel [110] all describe denotational semantics. Lambda calculus was introduced by Church [29], and is covered by Barendregt [12] and Hindley and Seldin [52]. There is also a typed version of λ-calculus, as described by Barendregt [13]. Rosser [91] gives a history of λ-calculus.

The idea for representing terms described in Exercise 49.16 is due to DeBruijn [37].

Chapter 50

Actions, Sequencing and Monads

50.1. Actions

Imperative programming is based on actions. An action can change a variable, show something on the console, modify a file, etc.

A functional program only computes values, and pure functional programming languages cannot allow a program to request that an action be done. For example, you cannot ask to print something, or to create a new file, since those things do not just involve computing a value, or giving a name to a value. But that is a serious problem, since a general purpose programming language should allow a program to interact with its environment in ways that involve making changes to the environment, such as creating or modifying files.

One solution is to back away from pure functional programming and to add actions to the language, yielding a language that is only partially functional. That approach is taken by Scheme and SML, for example. But there are some unpleasant consequences of that. A functional language is intended to offer mathematical simplicity. For example, if you know that two expressions A and B are equal then you should be able to replace A by B or B by A without changing the results that a program produces. But suppose that actions are allowed, and random() produces a different random number each time it is used. Then

```
Let x = random().
Let y = x + x.
```

is certainly not equivalent to

```
Let x = random().
Let y = random() + random().
```

When actions are introduced, proofs that programs are correct, as discussed in Chapter 48, have to take imperative aspects into account; the reasoning is usually sound only if no actions are involved. Mathematical simplicity is lost.

Delayed actions

But it turns out that actions can be added to a programming language in such a way that the language remains purely functional and the substitution rule always works. The key idea is to provide actions, but not to provide a way to perform an action. You can build an action, but you cannot run it! There is nothing imperative about a function that, for example, takes an integer n and produces an action that would print n, *if that action were run.* All you are doing is producing values of an action type. Because the actions do not do anything until they are triggered later, we will call them *delayed actions.*

Of course, *somebody* needs to be able to run a delayed action, or there is no point to it. A typical approach is to provide a user interface to a functional language where you can evaluate expressions. If the result of evaluating an expression is a delayed action, then the user interface (not the program) runs the action. That probably sounds strange, but it accomplishes the goal of embedding actions in a purely functional language. Also, whatever you want the program to do can be entirely encapsulated in a single delayed action; you can think of your functional program as building a procedural program as its result, which you then run via the user interface. We illustrate these ideas using Haskell (Chapter 24.)

50.2. Valueless actions

In general, a delayed action can not only have an effect, but it can also produce an answer. For example, if you read a value from a file, you need to know what was read. But for now, let's concentrate on *valueless delayed actions*, which only have an effect, and do not produce an answer.

Haskell function putStrLn takes a string and produces a valueless delayed action that, when run, will write that string, followed by a newline. So if printCompliment is defined by

```
printCompliment adj = putStrLn "You are " ++ adj
```

then (printCompliment ”wise”) is a delayed action that, when run, will write “You are wise”.

One of the fundamental concepts of imperative programming is performing actions in sequence; first do action A, then action B. Other constructs, such as conditionals and loops, are just additional ways of controlling the sequence in which actions are performed. You can sequence delayed actions in Haskell using the **do** construct. If A, B and C are valueless delayed actions, then expression

```
do A
   B
   C
```

produces, as its value, a delayed action that performs A, B and C, in that order. You can sequence any number of actions. For example,

```
do printCompliment "wise"
   printCompliment "generous"
```

creates a valueless delayed action that writes two compliments. It is important to keep in mind that the do construct does not actually do anything. It is an expression whose value is a delayed action that will do things, when the action is performed.

Tests and loops

You can simulate simple tests and loops by deciding how to build a delayed action. For example, suppose that you want to print “You are wise” n times. You will want an *do-nothing* action, to make it possible for n to be 0. It can be defined as follows in Haskell.

```
skip = return ()
```

Then to write the compliment n times, it suffices to chain together n delayed actions.

```
printWiseNTimes n =
  if n == 0
    then skip
    else do printCompliment "wise"
            printWiseNTimes (n-1)
```

Notice the need for another do in the else part to sequence delayed actions. Think of it as similar to a compound statement. Once again, printWiseNTimes(3) does not actually write anything. It is an expression whose value is a delayed action that, when run (if ever) will write n lines.

50.3. Actions that produce answers

Sometimes an action yields an answer. Delayed action

```
return v
```

does nothing but yield answer v. (It does not return from a function! It just yields a delayed action.) Haskell provides deferred action getLine that, when run, will read a line of text and return it (sans end-of-line character). To get the value that an action yields within a **do** expression, use form x `<-` action. For example, readTwoLines, defined by

```
readTwoLines =
  do line1 <- getLine
     line2 <- getLine
     return line1 ++ line2
```

is a deferred action that, when run, reads two lines and returns the two of them concatenated together. If you want to run an action and get its answer, you will need to use the `<-` notation and give a name to the answer. It would not work to do

```
readTwoLines =
  return getLine ++ getLine
```

Remember that getLine is a deferred action. Expression getLine ++ getLine asks to concatenate two deferred actions, not two strings.

50.4. Cautions

Delayed actions and the do construct make it easy to imagine that you are doing imperative programming. For example, you can define your own while-loop construct as follows, where both the condition to test and the body are given by delayed actions.

```
while cond body =
  do t <- cond
     if t
        then do body
                while cond body
        else skip
```

Now suppose that you want to read lines and only write those that begin with '#'. Using while, you can do that as follows, using Haskell function isEOF to test for end-of-file.

```
printSharpLines =
  while (do t <- isEOF
            return(not t))
        (do line <- getLine
            if line /= [] && head line == '#'
              then putStrLn line
              else skip)
```

But working with delayed actions in a functional language is really quite different from working in an imperative programming language. Haskell does not allow you to change the value of a variable, once it is given a value. (Allowing that would take Haskell out of the realm of functional programming.) Suppose that you decide to add up the values in a list of numbers as follows.

```
sum lst =
  do x <- lst
     s <- return 0
     while (return (x /= []))
           (do s <- return (head x + s)
               x <- return (tail x))
     return s
```

The loop tries to change *s* and *x*. But what happens is that new variables *s* and *x* are created in the loop body that shadow those outside the loop. The test keeps looking at the original list. So, if the original list is not empty, the loop goes forever. You are better off using recursion, and remembering never to try to change an existing variable value.

50.5. Foundations of sequencing actions

The **do** construct and notation x <- action are not primitive; they are defined in terms of more fundamental ideas.

A delayed action that produces an answer of type *t* has type IO *t*. An action that does not produce an answer has type IO ( ). A few primitive things provided by Haskell include

```
return   :: t -> IO t
putStrLn :: String -> IO ()
getLine  ::  IO ()
```

Haskell operator >>= performs sequencing. Its type is

```
(>>=) :: IO s -> (s -> IO t) -> IO t
```

The idea is that a >>= f produces an action that, when run, will (1) perform action a, yielding result r; (2) compute $b = f(r)$, where b is another action; (3) perform action b; and (4) produce the result of action b as its own result. Construct

```
do x <- a
   b
```

is converted to (a >>= (\x -> b)). Construct

```
do a
   b
```

just ignores the result produced by a. It is converted to (a >>= (_ -> b))

50.6. Monads and notions of sequencing

There is more than one notion of sequencing, and more than one kind of thing that can be sequenced. You can string together pearls on necklace, floats in a parade, or steps in an assembly process. A *monad* represents a particular definition of sequencing, and it is defined in terms of how operator >>= works. We have already seen how Haskell deals with actions. We look at two other examples, handling errors and backtracking, to show what monads offer.

Handling errors

Haskell does not support exception handling. But it does provide type Maybe t, whose values can be of the form Just(v) or Nothing, where Nothing indicates a lack of an answer. In addition to the return function, a monad offers a fail function, where fail(s) indicates a failure and s is a string that indicates the reason. There is no single definition of fail; it depends on the monad.

Imagine that you intend to compute expressions A and B, where computation of B depends on the value produced by A. If A produces Nothing, you probably do not want to continue with B; just produce Nothing for the entire computation, without computing B at all. Define

```
(Just x) >>= f = f x
Nothing  >>= f = Nothing
return x       = Just x
fail s         = Nothing
```

Then, using the definition of **do** expressions,

```
do x <- Just 1
   return x + 1
```

yields Just(2), but

```
do x <- fail "no result"
   return x + 1
```

yields Nothing.

Backtracking

Chapter 26 introduces backtracking and the Cinnameg Backtrack construct that employs it. Backtracking allows an expression to produce more than one value by forking into several branches and producing a different value in each branch. But it is not necessary to have direct support for backtracking; you can simulate backtracking by letting each expression produce a list of values. For example, an expression that produces list $[1, 2, 3]$ can be thought of as producing three different values, one in each of three branches. A key issue is how to sequence expressions that simulate backtracking. If expression A produces $[1, 2]$ and B produces $[5, 9]$, then $x = A$ followed by $y = B$ should be able to produce (x, y) values [(1,5), (1,9), (2,5), (2,9)]. The backtrack monad implements sequencing that way. For example, expression

```
do x <- [1,2]
   y <- [5,9]
   return x+y
```

yields list $[6, 10, 7, 11]$. Definitions for the backtracking monad are as follows.

```
xs >>= f = concat (map f xs)
return x = [x]
fail s  = []
```

Standard Haskell function concat takes a list of lists and produces their concatenation. For example, concat [[1, 2], [3, 4], [5]] = $[1, 2, 3, 4, 5]$. For example,

```
do x <- [1,2,3]
   return (x + 1)
```

is translated to ($[1, 2, 3]$ >>= (\x -> return(x+1))), which is computed as follows.

$$\begin{aligned}&[1, 2, 3] \texttt{ >>= } (\backslash x \texttt{ -> } \text{return}(x + 1))\\ &\quad = \text{concat}(\text{map } (\backslash x \texttt{ -> } [x + 1])\ [1, 2, 3])\\ &\quad = \text{concat}([[2], [3], [4]])\\ &\quad = [2, 3, 4]\end{aligned}$$

The Monad class

Class Monad t contains types such as IO t, Maybe t and $[t]$ that are defined to be monads. For example, the definition of the backtracking monad is as follows, where [ ] indicates the list-type constructor.

```
instance Monad [] where
  xs >>= f = concat (map f xs)
  return x = [x]
  fails s  = []
```

(Notice that Monad and [ ] are both parameterized. Saying that constructor [ ] belongs to Monad implicitly says that [*t*] belongs to Monad *t* for every type *t*.) To create a new monad, it is just a matter of adding a new type to Monad.

50.7. Summary of terminology and concepts

Actions can be incorporated into a functional programming language without compromising the principles that lie at the foundation of functional programming. It suffices to let a program build up actions, but not to let it to perform the actions. A functional program builds a procedural program, which is run by some agent that is external to the language.

The introduction of actions as values requires a way to perform actions in a particular sequence, and that leads to the ideas of monads. There is more than one way to define sequencing, depending on what is being sequenced, and a monad defines a particular meaning for sequencing. Examples of monads are a monad that sequences actions by performing one after the other, a monad that sequences steps that might yield errors by stopping at the first error, and a monad that simulates backtracking.

Terminology

Delayed action An action is a statement that will perform some side-effect when it is run. For example, an action might print a string to a file. In a purely functional language, actions are delayed because, although you can create an action, you cannot run it. Instead, an action that is built by a functional program is given to some other system, and run there.

Monad A monad defines rules for sequencing things together in a particular order.

Side-effect A side-effect is an action that is performed during the process of a computation.

50.8. Exercises

50.1. What is the advantage of using monads over simply adding imperative constucts to a functional language?

50.2. Convert each of the following definitions to its underlying form using `>>=`.

(a) printCompliment

(b) readTwoLines

50.3. Haskell allows an arbitrary pattern on the left-hand side of `<-`. As a first approximation,

```
do p <- a
   b
```

is translated to (a >>= (\p -> b)). But there is a problem with that. Although Haskell does allow a pattern after \, such patterns are restricted to those that cannot fail to match. It is not allowed to have a function (\x:xs -> . . .) since pattern x:xs can only match a nonempty list. To deal with that, a failed pattern match is defined to yield fail(s) where s is a string that says the match failed. Show how to translate the above do expression, for an arbitrary pattern p, so that it has the desired effect.

50.4. Write a Haskell definition of function conditional(A, B, C) that simulates a conditional (if-then-else) statement where the condition A to be tested is a delayed action that produces a boolean result, and the two options B and C are valueless delayed actions.

50.5. Would the while function make any sense for the maybe monad or the backtracking monad, or is there something about the concept of a loop that says it should only be used to sequence actions?

50.6. Section 50.1 suggests that you can think of a functional program as building a procedural program, which you then run. But the functional program does not produce the entire procedural program before you begin performing the actions. Rather, the functional program builds the procedural program on the fly, producing what is needed as it is needed.

Examine function printSharpLines. Argue that it is not possible for the functional program to build the desired sequence of actions ahead of time. Instead, it is critical that the actions are produced while they are being run, so that the procedural program and the functional program that builds it must run side by side, not one after the other.

50.7. Solve Exercise 26.3 in Haskell, using the backtracking monad for backtracking.

50.8. The maybe monad acts as a stand-in for exception handling. Can you see any reason for using it instead of providing direct support for exception handling in a purely functional language?

Part XVII

Appendices

Appendix A

Introduction to Cinnameg

A.1. Introduction and general principles

Cinnameg is a programming language that is used in this book to illustrate general programming language principles. The ideas presented by Cinnameg are representative of those found in a variety of other languages. This appendix introduces you to Cinnameg, but is not intended to be a full description of the language. Some aspects of Cinnameg that are not covered here are discussed in other chapters.

Cinnameg is intended to be a conceptual programming language, and is further removed from the characteristics of computer hardware than some languages. For example, integers are of arbitrary size, not limited by the size of a machine word, so that the sum of two positive integers is always a positive integer, as it is in mathematics.

Some programming languages are *paradigmatic*, meaning that they enforce or highly encourage a particular programming paradigm. Java, for example, requires the programmer to adopt an object-oriented approach to programming; Prolog requires a logic-programming approach; and Haskell enforces a functional approach. Cinnameg is deliberately designed not to be paradigmatic, but to support a variety of paradigms. You can use Cinnameg either for declarative programming or for imperative programming. Much of the library is designed in a functional style because that tends to be conceptually the simplest and leads to the fewest errors. But different styles of programming can be blended seamlessly.

Cinnameg is designed to provide different paradigms in a way that is easy to pick up by someone who is familiar with the principles of that paradigm. You do need to learn the syntax of Cinnameg, however. No attempt is made to be compatible with the syntax of other languages.

Some experts in programming languages believe that, wherever possible, a language should provide just one way to do a particular thing. That tends to reduce language complexity, makes the language easier to learn and can make programs easier to read. Because of Cinnameg's philosophy, that is not possible. The whole point is to provide a variety of ways of doing things. But Cinnameg is intended to be learned in subsets. You can begin using it without a large investment of time. Also, things that you have not yet seen are intended to be relatively easy to read and understand, as long as you understand the underlying ideas.

A.2. Syntactic issues

Lexical issues

Cinnameg is free-form. In most contexts, a line break has the same meaning as a space. Normally, spaces are only allowed between tokens, and are only required where the boundary between two tokens would otherwise be unclear.

A comment begins with %% and continues to the end of a line. You can also use any sequence of four or more = signs or - signs as a comment, but ====... can only occur between two declarations.

Identifiers have two forms. *Ordinary identifiers* begin with a letter and can contain letters, digits and question marks. For example, `horse1` and `nil?` are ordinary identifiers. *Symbolic identifiers* are symbols such as +, * and /. Symbolic identifiers can also be several characters long, such as ++ or +//, and can contain characters in string “`~#+|-*/^@!=<>&$_\?`”, as long as they do not start with “?”. Some reserved words and symbols are not allowed to be identifiers.

Bracketing

The syntax is *bracketed.* You write `Define ...%Define`, beginning with an introductory word `Define` and ending with `%Define`, meaning end-of-define. In general, any construct whose first word begins with an upper case letter must end on a corresponding end marker. If it starts with Foo, it ends with %Foo.

Bracketing has the advantage that it is clear where each unit of the program ends. But it tends to be awkward for short things. To reduce the inconvenience, you can use a period as a replacement for the end marker. So you can write

```
Define x = y + 1.
```

rather than

```
Define x = y + 1 %Define
```

to give name x to $y + 1$.

A semicolon ends one construct and begins another of the same kind. For example, when it occurs where %Define is expected, a semicolon stands for %Define Define.

A.3. Naming and scope

You can give a name to an item using either Define or Let. Although there are slight differences in the meaning of the between the two, discussed later, for now assume they are the same. For example,

```
Let kangaroo = 50.
```

makes kangaroo refer to 50. Having bound a *local* identifier (within a function definition or execute block) you can rebind it to a new value using a Relet statement. For example,

```
Relet kangaroo = kangaroo + 1.
```

binds kangaroo to one larger than its former value. Use a Let for the first binding and Relet only to change an existing binding.

Scope

As discussed in Chapter 12, scope at the local level is defined in terms of the concept of a *block*; bindings done inside a block are only visible inside that block. With a few exceptions, a block in Cinnameg is any section that starts with an upper case word and ends at the corresponding end word. For example, Let ...%Let is a block. Exceptions include constructs, such as (x |-> E), where bindings in E are actually done in a different context.

You can request that some constructs not be blocks by preceding them with the word **open**. For example, open Let ...%Let is not considered to be a block and bindings done inside it (that are not scope-limited by an embedded block, and that are sure to be done) are visible outside of itself.

A.4. Expressions

Expressions can be written in typical mathematical notation, using numerical constants and operators such as + for addition and * for multiplication. The usual precedence rules are used. For example, expression 1+(2+3)*5 has value 26. Here are a few standard binary operators.

+, -, *, /	Addition, subtraction, multiplication and division operators
`div`, `mod `	Integer quotient (ignoring remainder) and integer remainder
==, =/=	Equality and inequality tests
>, <, >=, <=	Order tests
and, or, not	(short-circuited) boolean operators

When a binary operator is an ordinary identifier, enclose it in left single quotes, like `div`. The quotes are optional on **and** and **or**.

Statements

The term *statement* is usually used in programming languages for a command, requesting that some action be performed. Cinnameg allows an expression to have a side-effect, and treats a statement as an expression whose value is a special value written ( ). For example, Let x = 1 %Let is an expression that produces result ( ).

You can form an expression by writing any sequence of expressions, one after the other, provided all but the last one (and possibly the last one too) produce result ( ). The expressions are evaluated, in the order written; the value of the entire sequence is the same as the value of the last expression in the sequence. For example, expression

```
Let r = 5.
Let s = 6.
(r + s)
```

has value 11. Since statements are expressions, you can use parentheses with them. For example, (Let r = 5. (Let s = 6.) (r + s)) is an expression. There is also a notation, $E \mid S$, that is equivalent to $S\,E$, when S is a sequence of statements. Read | as "where". For example,

```
r + s |
  Let r = 5.
  Let s = 6.
```

is the value of expression $r + s$ after performing the definitions that come after `|`.

A.5. Data

Numbers are written in the usual forms, such as 32 and 71.4. Truth values are **true** and **false**. Character constants are written in single quotes, such as ’a’. Special characters include ’\n’ (newline) and ’\t’ (tab).

Ordered pairs

An ordered pair is written in parentheses, such as (2,3). If p is an ordered pair, then left(p) is the left-hand member of p, and right(p) is the right-hand member. For example, right(2,3) = 3. The members of an ordered pair can be anything, including other ordered pairs, as in $(1, (2, 3))$. That is exploited in provided for ordered triples, 4-tuples, etc. Expression (A, B, C) abbreviates (A, (B, C)), and (1, 2, 3, 4) abbreviates (1, (2, (3,4))). So right(1,2,3,4) = (2,3,4).

Lists

A list is a sequence of items that all have the same type. You can write a list in square brackets, such as $[a, b, c]$, with members separated by commas. The empty list is [ ]. Notation [1, . . . , 5] stands for the list $[1, 2, 3, 4, 5]$. In general, [x, . . . , y] stands for a list that starts at x and counts *up* to y. If you want to count down, or to count by any increment other than 1, use a form such as $[10, 8, \ldots, 2]$, which is list $[10, 8, 6, 4, 2]$. Some standard functions on lists are shown in Figure A.1.

Operator :: associates to the right, so $a :: b :: c :: d$ is understood to be the same as $a :: (b :: (c :: d))$. Notation $[a, b, c]$ abbreviates $a :: b :: c ::$ [ ]. Pay heed to the empty list at the end; $[a, b, c]$ is *not* the same as $a :: b :: c$.

A string is a list of characters. For example, ”abc” and [’a’,’b’,’c’] mean the same thing. Since a string is a list, you can use the general list operations on strings. For example ”abcdefg” _ 2 = ’b’ and ”abcdefg” _* [3, . . . , 6] = ”cdef”.

Boxes

A box is an object that has a content that can be read and changed. You can pass boxes among functions, store boxes in other variables, etc. Use function `@` to fetch the content of a box, and statement `Make @`*b* `=!` E `%Make` to store the value of expression E as the new content of box b. A box is allowed to be empty. Expression [: :] evaluates to a new empty box, and expression [: E :] evaluates to a new box with initial content E. An alternative way to create a box or a list of boxes is using a var statement. Statements

```
Var frog : Real.
Var frogs(20) : Real.
Var goat: Integer =! 100.
Var goats(80) : Integer all =! 0
```

Function	**Meaning**
head	head(x) is the first member of list x.
last	last(x) is the last member of list x.
tail	tail(x) is the list obtained by removing the first member from list x. For example, tail([3,4,5]) = [4.5].
allButLast	allButLast(x) is the list obtained by removing the last member of list x. For example, allButLast($[3, 4, 5]$) = $[3, 4]$.
nil?	nil?(x) is true if x is an empty list.
_	x _ n is the n-th member of list x, numbering from 1. For example, $[3, 4, 5, 6]$ _ $2 = 4$.
_*	x _* y is the sublist of list x indicated by the indices in list y. For example, $[6, 2, 5, 10, 8]$ _* $[3, 4] = [5, 10]$.
::	x::y is the list whose head is x and whose tail is y. For example, 4::$[6, 1] = [4, 6, 1]$.
++	x++y is the concatenation of lists x and y. For example, $[2, 4, 6]$++$[3, 5, 7] = [2, 4, 6, 3, 5, 7]$.
`in`	n `in` x is true if n is one of the members of list x. For example, 2 `in` $[3, 2, 5, 7]$ is true.
length	length(x) is the length of list x — that is, it tells how many members list x contains. For example, length($[8, 5]$) = 2.
reverse	reverse(x) is the reversal of list x. For example, reverse($[5, 2, 7, 8]$) = $[8, 7, 2, 5]$.

Figure A.1: Some standard functions on lists.

create: an initially empty box called frog that can hold a real number; a list called frogs of 20 boxes, each initially empty, where each variable in list frogs also can hold a real number; a box called goat that can hold an integer, and that initially holds 100; and a list of 80 boxes called goats, all initially holding 0.

A.6. Functions and procedures

All functions have one parameter and produce one result. Expressions $f\,x$ and x `~>` f each indicate the result of applying function f to argument x. Both associate to the left. So $f\,x\,y$ is understood to the the same as $(f\,x)\,y$ and x `~>` f g is understood as $(x$ `~>` $f)$ g.

You can let a function *appear* to have two parameters by letting its parameter be an ordered pair, as in $f(x, y)$. Binary operators such as + and * are considered nothing but syntactic conveniences. Expression $x + y$ is just an abbreviation for $(+)(x, y)$, which applies the addition function (+) to the ordered pair (x, y). You can write $(+)(x, y)$ in your program if you prefer, but you will need to put operator (+) in parentheses to cause the compiler to suppress its syntactic treatment as a binary operator.

There is another way to allow a function to have more than one parameter. Function application associates to the left, so if you write $f\,x\,y$, you implicitly mean $(f\,x)\,y$. Expression $(f\,x)$ yields a function that takes parameter y and computes another value; such a function is said to be *curried*. Some of the library functions are curried. For example, function **take** is defined so that (take n x) returns the length n prefix of list x. So take 3 $[5, 9, 1, 4, 7, 2] = [5, 9, 1]$. Notice that there are no parentheses around the parameters, since they are not put together into an ordered pair. You can write take(n)(x), but not take(n,x).

An advantage of curried functions is that you can pass just some of the arguments. Expression (take 2 "human") is equivalent to ((take 2) "human"), and yields "hu". So you would expect that, if you define function grab to be (take 2), then (grab "human") should yield "hu", which it does. For example,

```
Let grab = take 2.
Let str = grab "human".
```

binds str to "hu".

Creating functions

Expression (x `|->` E) stands for a function which, on input x, produces result E. For example, evaluating expression ((x `|->` $x + 1$) 4) yields result 5. In general, if p is any pattern (see below) then (p `|->` E) is a function that matches p against its parameter and then produces the value of expression E.

Functions can be given names using Define or Let. Statement

```
Define incr = (x |-> x + 1).
```

binds identifier **incr** to a function, so that incr(5) = 6. Alternatively, you can use an abbreviation for defining functions. Definition

```
Define incr(x) = x + 1.
```

abbreviates the previous definition of **incr**. In general, `Define` $f(p)$ `=` E `%Define` is equivalent to `Define` f `= (p |->` E`) %Define`. Using the same abbreviation twice,

```
Let crunch x y = x + 2*y.
```

is equivalent to

```
Let crunch = x |-> y |-> x + 2*y.
```

You can define binary operators. For example, `Define x + y =` *E* `%Define` abbreviates `Define (+)(x,y) =` *E* `%Define`, which in turn abbreviates `Define (+) = ((x,y) |->` *E*`) %Define`.

Procedures

Sometimes you want a function (called a *procedure*) that does not produce a result, but instead just performs an action. In keeping with the idea that a statement is a kind of expression, a procedure is a kind of function whose result is the value ( ). By convention, procedures are given names that start with upper case letters. For example, one of the standard procedures is called Writeln, which writes a list of strings to the standard output, followed by an end-of-line marker. When you use a procedure (with a name that starts with an upper case letter) you must include a matching end marker. For example, you might write

```
Writeln ["hello"] %Writeln
```

or, using a period to stand for %Writeln,

```
Writeln ["hello"].
```

Procedures can be curried. To call a curried procedure, separate the parameters by commas. For example, library procedure FWriteln is used to print a list of strings on a file. It is curried, taking two parameters, a file to write onto and a list of strings to write. You write

```
FWriteln f, ["hello"].
```

to write “hello” onto file f. Notice that there are no parentheses around the parameters. They are not being put into an ordered pair.

A.7. Pattern matching

We have already used some pattern matching in defining functions. You can also make an explicit request for a pattern match. Statement

`Match` p `=` v `%Match`

matches pattern p against target v. A pattern match typically asks for solving an equation. For example, if you match pattern (x, y) against value (2,4), you are solving equation $(x, y) = (2, 4)$ for x and y. The result is to bind x to 2 and y to 4. So

```
Match (x,y) = (2,4).
```

has the same effect as

```
Let x = 2.
Let y = 4.
```

Type	Meaning
Char	Characters
Boolean	true and false
Integer	Integers
Real	Real numbers (represented approximately)
(Integer,Real)	The type of ordered pair (2:Integer, 6.5:Real)
[Integer]	The type of [1:Integer, 2, 3]
[Char]	The type of "abc". You can also write String.
(Integer,Integer) -> Integer	The type of the addition function that takes a pair of integers and produces an integer.

Figure A.2: Some standard types of Cinnameg.

The solution to equation $h :: t = [2, 4, 6]$ is $h = 2$ and t = $[4, 6]$. You can use a question mark as a place holder for an arbitrary value. For example, `Match h::? = x %Match` is equivalent to `Let h = head x %Let`.

Pattern matches can fail. You can include already known values v in pattern matches by writing $(=v)$. For example, `Match (=x) = y %Match` succeeds if $x = y$, and fails otherwise. Later sections indicate what happens when a pattern match fails.

A.8. Types and type inference

Cinnameg is a typed language and if you write a definition that does not make sense from the standpoint of types, the compiler will report an error. For example, expression (6+true) is not allowed.

When you encounter a type error, first try to understand, at least partially, what the compiler is saying. You might, for example, be able to see that the compiler thinks that x must both be a character (type Char) and a list of characters (type [Char]). Then examine your program to see where you have made an error. Do not only look at the line where the error occurred. Often, the error occurred at a different line in the same definition. If you still do not see the problem, try adding some type information, as shown in the next subsection, to tell the compiler what you believe the types are. The error will not go away, but it might be reported in a way that you will understand better.

Supplying type information

You are not required to give type information in programs. The compiler will infer type information from context. For example, if you write `Define x = 7 %Define` the compiler can see that x is a number. Occasionally, you will find it convenient to add type information to your program, to be sure that you understand what you are getting. One way to do that is by making conventions about names. Writing

```
Assume n: Integer.
```

tells the compiler that you intend for all occurrences of identifier n to name a value of type Integer. Table A.2 shows a few of the standard types that are available.

Genus	**Meaning**
Any	Genus Any contains all types
AnyEq	Genus AnyEq contains types that support equality tests.
AnyOrder	Genus AnyOrder contains types that support order tests, such as <.
AnyNum	Genus AnyNum contains Real, Rational, and Integer. These types support addition and multiplication, but not necessarily division.
AnyRfield	Genus AnyRfield contains Real and Rational. The key feature of these types of numbers is that they support division (/), as well as the other usual operations of arithmetic.

Figure A.3: **Some standard genera of Cinnameg.**

You can build types using type constructors. Type (S, T) is the type of ordered pairs (x, y) where x has type S and y has type T. Notation $(T_1, \ldots, T_n)$ generalizes that to arbitrary tuples; (A, B, C) abbreviates $(A, (B, C))$ and (A, B, C, D) abbreviates $(A, (B, (C, D))$. Type $[T]$ is the type of lists whose members have type T. Type S `->` T is the type of a function whose parameter has type S and whose result has type T.

You can tag any expression with a type. For example,

```
Define x: Integer = 7.
```

indicates that x has type Integer. But be careful; the colon operator has very low precedence, so it applies to a large expression. Writing (x+y: Integer) indicates that the entire expression $x + y$ has type Integer.

A.9. Polymorphism

Polymorphism is expressed in terms of type variables. A variable of the form $\langle x\rangle$ stands for any type at all. A genus is a set of types. Variable ⟨x: AnyEq⟩stands for a type that belongs to genus AnyEq. For example, the equality testing operator == has polymorphic type (⟨x: AnyEq⟩, ⟨x: AnyEq⟩) `->` Boolean, indicating that it can have type (T, T) `->` Boolean for any type T that belongs to AnyEq. You can compare numbers, character, strings, lists of numbers, etc. for equality, since all of those types belong to AnyEq. Figure A.3 shows some of the standard genera.

Indicating a genus for a variable once is all that is needed. For example, the addition function, (+), has polymorphic type (⟨a: AnyNum⟩, ⟨a⟩) `->` ⟨a⟩. All occurrences of ⟨a⟩must be the same, so all are limited by a genus if any are. You can move the requirement to a **where** clause after the type, containing a comma-separated parenthesized list of variables. The addition function has polymorphic type ((⟨a⟩, ⟨a⟩) `->` ⟨a⟩ where (⟨a: AnyNum⟩)). If a type variable only occurs once, it is convenient to use omit the variable's name. Each occurrence of variable ⟨AnyEq⟩ is a different variable.

Numeric constants are polymorphic. For example, constant 2 has polymorphic type ⟨a: AnyNum⟩, and constant 2.5 has polymorphic type ⟨a: AnyRfield⟩.

A.10. Input and output

Output

You will need to be able to write things. Procedures Display and Displayln each takes a single parameter, of any type, and writes it to the standard output. Procedures Write and Writeln each takes a list of strings and prints those strings on the standard output. (Writeln and Displayln add an end-of-line marker, but Write and Display do not.) To write just an end-of-line marker, use

```
Writeln.
```

In order to print something that is not a string using Write or Writeln, you must convert it to a string. There is a function with the symbolic name $ that converts things to strings. For example, $(24) = ”24”. Statement

```
Writeln["n = ", $(n)].
```

shows the value of *n*. Be careful about using $ on a string. It adds extra quote marks around the string.

Input

Input is based on string processing. To read a file, you use function **fileContents** or **lazyFileContents** to convert a file name into the contents of the file. For example, expression fileContents(”data.txt”) produces the contents of file data.txt, as a string. Now use any string operations on the file contents.

A convenient way to work with strings when reading data is to put a string into a box and then let functions get a prefix of the string, setting the box to hold the suffix that was not read. Use an Extract statement for that. Statement `Extract [`p_1`, ..., `p_n`] from` *b* `%Extract` matches patterns p_1, ..., p_n in sequence against the contents of box *b*. Each matches a prefix of what is left, and removes it. The patterns need to say what is being matched. For example,

```
Let b = [:"20 rings of gold":].
Extract [$(n:Integer), theString(s)] from b.
```

starts by finding a prefix that might be $(*n*) for some *n*. (Initial white space is skipped first.) It gets the prefix “20” and binds $n = 20$. Pattern theString(s) looks for a white-space terminated or quoted string, and gets that. In this case, *s* is bound to “rings”. Box *b* is left holding “ of gold”.

There are also a few functions for reading from boxes. Each reads a prefix of the content of the box, replacing the content by the part not read. `readString(bx)` reads a white-space terminated string (or a quoted string), `readInt(bx)` and `readReal(bx)` read an integer or real number, respectively, and `readChar(bx)` gets just one character. The read functions have versions that take parameter ( ) instead of a box, and that read from the standard input. For example, `readInt()` reads an integer from the standard input. Expression `stdin()` yields the box that contains the standard input.

A.11. Conditionals

If-then-else

Expression `If` A `then` B `else` C `%If` produces the value of expression B if expression A has value true, and produces the value of C if A has value false. Use if-then-else for simple (usually binary) choices. The following is a definition of a function that computes the maximum of two numbers.

```
Define max(x,y) =
  If x > y
    then x
    else y
  %If
%Define
```

You can omit either the then part or the else part; they default to **then** ( ) or **else** ( ).

Multiway choices

An alternative way to make choices is via the Choose construct. Expression

```
Choose
  case G₁ => V₁
  case G₂ => V₂
  ...
  case Gₙ => Vₙ
  else    => Vₙ₊₁
%Choose
```

evaluates expressions G_1, ..., G_n (called the *guards*) up to the first one that is found to be true. If G_i is the first true guard, then V_i is evaluated, and its value is the value of the Choose expression. If none of the guards are true, then V_{n+1} is the value of the Choose expression. (You are not required to have an else case. But if you don't, and none of the cases apply, the Choose expression will fail.) For example, the following implements the `in` operator for lists.

```
Define x `in` y =
  Choose
    case nil? y      => false
    case head y == x => true
    else             => x `in` tail y
  %Choose
%Define
```

Alternatively, you can indicate that the guards are patterns that are to be matched against a particular value. The first pattern match that succeeds is chosen. Here is an example, an alternative definition of the `in` operator. Notice that pattern (=x)::? requests an equality test.

```
Define x `in` y =
  Choose matching y
    case []        => false
    case (=x):: ?  => true
    case ? :: more => x `in` more
  %Choose
%Define
```

A.12. Loops

While loops

Statement `While` E `do` S `%While` starts by evaluating expression E. If E is false, then the loop finishes and the expression produces value ( ). If E is true, then statement S is evaluated, and the loop starts again. The loop keeps going until E is evaluated and found to be false. `in` operator using a loop.

For loops

Statement `For` p `from` x `do` S `%For` matches p against each member of list x. If the pattern match succeeds, then S is performed, using the bindings done during the pattern match. If the match fails, then this member of list x is skipped. The following function takes a list x of lists and prints the heads of all of the members of x that are not empty. For example, PrintHeads([[3,4], [ ], [8,2]]) prints 3 and 8.

```
Define PrintHeads x. =
  For h::? from x do
    Writeln[$h].
  %For
%Define
```

General loops

A more general loop has the form

```
Loop $p$ = $init$
  exitcase $G_1$ => $V_1$
  loopcase $G_2$ => $S_2$ Continue $V_2$.
  exitcase $G_3$ => $V_3$
  loopcase $G_4$ => Continue $V_4$.
  ...
  loopcase $G_n$ => $S_n$ Continue $V_n$.
%Loop
```

The loop starts by matching pattern p against the value of expression *init*. Each case is tried, in the order written, up to the first one whose guard G_i is true. (The cases do not need to alternate between **exitcase** and **loopcase**. The case kinds can be mixed in any way.) If the case with the first true guard starts with the word **exitcase**, then expression V_i is evaluated, and the loop expression finishes with value V_i. But if the first case with a true guard starts with **loopcase**, then S_i is evaluated for its side effects and the loop starts again, this time matching pattern p against the value of expression V_i. Statement S_i can be omitted to do nothing before starting the loop again.

The guard in the last case can be the word **else** to indicate that this case is always taken when it is reached. For example, the following is yet another implementation of the `in` operator, this time using a Loop construct.

```
Define x `in` y =
  Loop rest = y
    exitcase nil? rest   => false
    exitcase head y == x => true
    loopcase else        => Continue(tail rest).
  %Loop
%Define
```

Loops are similar to choose expressions, and have similar forms available. For example, the following is another way to express the definition of `in` shown above.

```
Define x `in` y =
  Loop rest = y matching rest
    exitcase []       => false
    exitcase (=x) :: ?  => true
    loopcase ? :: t => Continue(t).
  %Loop
%Define
```

A.13. Definition by cases

You can define a function by giving several equations. For example, the factorial function can be written using two cases, as follows.

```
Define
  case factorial(0)    = 1
  case factorial(n+1) = n*factorial(n)
%Define
```

The first case whose pattern match succeeds is chosen. You can add provisos of the form **when** E that must also be true in order for a case to be chosen. The following is a definition of the filter function, where (filter f x) is a list of all members m of list x where $f(m)$ is true. For example, `(filter (x |-> x < 100) [20,150,200,15])` yields [20,15].

```
Define
  case filter f []     = []
  case filter f (h::t) = h :: filter f t   when f(x)
  case filter f (h::t) = filter f t
%Define
```

If two or more cases in a row have the same left-hand side, you can omit the left-hand side in all but the first of those cases. For example, the following is an equivalent way to write filter.

```
Define
  case filter f []     = []
  case filter f (h::t) = h :: filter f t   when f(x)
                       = filter f t
%Define
```

A.14. Examples and unit testing

You can add examples to your program that will be run to check your definitions. Typically, you group them with your definition into a **team**. For example,

```
Team
  Example factorial(3) = 6.
  Define
    case factorial(0)   = 1
    case factorial(n+1) = (n+1)*factorial(n)
  %Define
%Team
```

causes factorial(3) to be evaluated when this module is loaded. If the result is not 6, then computation will abort, telling you which example did not work. An example can be an equation or any expression of type Boolean.

The compiler looks at examples when it does type inference to tell how you intend to use a function. If you use it on a number, for example, then the compiler will only allow it to work on numbers. If you want a polymorphic function, use the word Sample instead of Example. Here is an example, a function that computes the length of a list.

```
Team
  Sample len("abcd") = 4.
  Sample len [1,2,3] = 3.
  Define
    case len []     = 0
    case len (?::t) = 1 + len t
  %Define
%Team
```

You can suppress example/sample tests by specifying, in the cmgr command, that example is 'no'. For example,

```
cmgr -dexample=no myprogram
```

runs myprogram.cmg with testing of examples suppressed.

A.15. Evaluation time

Normally, evaluation is strict: expressions are evaluated as soon as they are encountered. So, for example, if your program performs statement `Let frog = find(amphibian) %Let`, then expression find(amphibian) is evaluated, and identifier frog is bound to its result. But you can request that evaluation should be deferred until a value is needed. Statement `Define frog = find(amphibian) %Define` binds frog to an unevaluated expression that will only

Exception	**Meaning**
conversionX	A conversion has failed. This happens when you try to convert string ”abc” to a number, for example.
domainX(s)	A parameter is not in the domain of a function. String s describes the function. If you divide by 0, you get exception domainX(”divide by zero”).
emptyBoxX	Attempt to get the content of an empty box.
emptyX	Attempt to get the head or tail of an empty list.
exampleX(s)	An example has failed. String s tells where the example is located.
noCaseX	A choose or loop construct does not have a suitable case.
noFileX(s)	Attempt to open a nonexistent file.
subscriptX(n)	Subscript n is too large or too small for a list.
testX	A test failed. This usually means that one branch of a computation failed because it has made some incorrect choices, and it wants to let other branches run instead.

Figure A.4: **Some standard exceptions.**

be evaluated when (and if) the value of frog is examined. An alternative way to do that is to write `Let frog = (:find(amphibian):) %Let`. In general, smiles `(:...:)` request deferred evaluation.

Several library functions are lazy; they produce only as much of their results as a program looks at. For example, lazyFileContents(f) produces the contents of file f, but only reads information from the file when it is needed.

A.16. Catching failure

Evaluation of an expression can fail, and each failure has an associated *exception*, indicating the reason for the failure. Figure A.4 shows some standard exceptions. To recover from a failure, you can use a Try expression, such as the following.

```
Try s
  then b
  else c
%Try
```

It starts by evaluating statement s. If evaluation of s succeeds, then the try-expression has value b. If it fails, then the try-expression has value c. You can omit either or both of the then-part and the else part; they default to **then** ( ) or **else** ( ).

Within expression c in the above try expression you can examine the exception associated with the failure by referring to **exception**, whose value might, for example, be conversionX. You might use pattern matching.

```
Try a
  else
    Choose matching exception
      case conversionX =>
        ...
    %Choose
%Try
```

Causing failure

Expression fail(e) fails with exception e. For example, evaluating fail(conversionX) stops the current line of computation, and looks for a try expression to catch exception conversionX.

Some convenient abbreviations

You can use the following abbreviations.

Abbreviation	**Abbreviates**
`Succeeds` E `%Succeeds`	`Try` E `then true else false %Try`
A `butIfFail` B	`Try Let val =` A `%Let then val else` B `%Try`
`MakeSure` c `else` e `%MakeSure`	`If` c `then () else fail(`e`) %If`
`MakeSure` c `%MakeSure`	`MakeSure` c `else testX %MakeSure`

For example, statement `MakeSure x > 0 %MakeSure` succeeds if $x > 0$, and fails with exception testX otherwise.

A.17. Packages and declarations

A Cinnameg program is broken into packages. Each package has a heading giving a name to the package and a sequence of *declarations* that define things or request that something be done. Declarations can use Define or Let to make definitions and can use Execute to ask for a computation to be done when the package is loaded. Figure A.5 shows a complete package that prints the contents of a given file on the standard output.

The library and imports

The library contains some general purpose types, functions, etc. To use a given package in the library, you must import it. Declaration

```
Import "collect/string".
```

imports everything defined in the library package called *string* from the collection library. Figure A.6 shows some of the standard packages that are available.

Exports

If you want a package to have exports, break it into two parts, one part stating what is exported and the part other providing definitions. In the export part, you can use an **Expect** declaration to indicate a binding that is being exported. You must give the name and its type in the Expect declaration. Figure A.7 shows a package that defines a function

```
Package showfile
  Define getFileNameFromUser() = readString() |
    Writeln["What file do you want to see?"].
  %Define

  Execute
    Let infileName = getFileNameFromUser().
    Let infileContents = lazyFileContents(inFileName).
    Write[infileContents].
  %Execute
%Package
```

Figure A.5: A Cinnameg program to read a file name from the user and show the contents of that file.

Import	Purpose
collect/list	List functions
logic/logic	Logic programming support
math/math	Some mathematical functions
misc/random	A random number generator
system/system	Some system functions
system/url	Reading web pages

Figure A.6: Some standard packages.

```
Package prime
  export
    Expect isPrime : Integer -> Boolean.

  implementation
    Define divisibleBy(n,k) = n `mod` k == 0.

    Define isPrime(n) =
      If n < 2 then false else
        Let k = 2.
        While k < n and not divisibleBy(n,k) do
          Relet k = k + 1.
        %While
        (k < n and divisibleBy(n,k))
      %If
    %Define
%Package
```

Figure A.7: **A Cinnameg package that exports a function.** The expect declaration in the export part tells what is exported. The word **implementation** introduces the implementation part, where the definitions are made.

that determines whether a number is prime, in a style similar to the Ada package of Figure 12.5.

A.18. More material

Section 33.1 describes support for logic programming and Chapter 26 covers direct use of backtracking. Chapter 21 discusses some higher order functions from the library. Chapter 34 explains how to create new types, and Chapter 36 tells how to create new genera and Chapter 37 shows how to make polymorphic definitions. More material is available from the language documentation at www.cs.ecu.edu/~karl/cinnameg/5-1/.

A.19. Exercises

You will need to examine documentation on Cinnameg and using an implementation to do some of these exercises.

1. Write a Cinnameg program that reads an integer and prints its square.

2. Write a Cinnameg definition of function second that takes a list of at least two members and produces the second member of the list. For example, second([2,4,6,8]) = 4 and second("abcde") = 'b'.

3. Read the documentation on the math/prime package. Write a package that reads a positive integer n and prints the prime factors of n, using the factor function. What paradigm does the factor function best belong to? What paradigm did you use to write your program?

4. Write and test a Cinnameg package that reads a number n and prints all of the prime numbers that are less than n.

5. Write an implementation of the factorial function in Cinnameg. Write a package that reads a number n and writes a table of the factorials of all numbers less than or equal to n.

Appendix B

A Logic Programming Example

B.1. Solving cryptograms

Suppose that you would like a program that solves cryptograms of the kind that you might find on the newspaper puzzle page, where each letter stands for one different letter. We will derive a program that solves those cryptograms using a mixture of logic programming and other styles.

One issue is how to represent information about character correspondences. A simple idea is to use a list of pairs, where [('A', 'P'), ('R', 'E')] says that plain character 'A' is represented by cipher character 'P' and plain character 'R' is represented by cipher character 'E'. We will also need a dictionary, which is represented as a list of lists of strings, where the first list contains words of length 1, the next contains words of length 2, etc. For example, the dictionary might start [["A", "I"], ["AD", "AM", "AN", ...], ...].

Now it is just a matter of trying possible plain/cipher word correspondences and augmenting a cipher list to take each correspondence into account, failing if that is not possible, and backtracking among choices. For example, if you decide to try cipher word "XVE" corresponding to plain word "FOR", then you need to ensure that the cipher contains pairs [('F', 'X'), ('O', 'V'), ('R, 'E')], and that all of those character correspondences are consistent with other pairs in the cipher.

Figure B.1 shows four predicate definitions, written in Cinnameg, for that purpose. All four have a similar general intent: they establish a correspondence and update the cipher. But they work on different size pieces (characters, words or lists of words). For example, extendCipherChar(inciph, outciph, *p*, *c*) is true if it is consistent with inciph that plain character *p* has associated cipher character *c*, and outciph is a cipher that contains all correspondences in inciph plus (p, c) (at its end) if needed. For example, extendCipherChar([('A', 'R')], [('A', 'R'), ('D', 'M')], 'D', 'M') is true. Predicate extendChipherChar(inciph, outciph, *p*, *c*) is intended to be used in a mode where inciph, *p* and *c* are in-parameters and outciph is an out-parameter.

The first two predicates (extendCipherChar and extendCipherWord) just establish a given correspondence, without regard for any dictionary. The next two (extendCipherWordDict and extendCipherWordsDict) choose the plain-text word from a dictionary.

There are some support functions needed for handling the dictionary. It is most convenient to write those in other styles, not a predicates. Figure B.2 shows them. Finally, Figure B.3 shows a function that gets the dictionary and shows the solution.

```
Import "logic/unify".  %% Mode: extendCipherChar(in inciph, out outciph,
%%                        in plain: Char, in ciph: Char)
Define first
  case extendCipherChar([], [(p,c)], p, c) <-
  case extendCipherChar((a,b)::r, (a,b)::r, a, b) <-
  case extendCipherChar((a,b)::r, (a,b)::s, p, c) <-
    a =/= p and b =/= c and extendCipherChar(r, s, p, c)
%Define
=======================================================
%% Mode: extendCipherWord(in inciph, out outciph,
%%                        in plain: String, in ciph: String)
Define first
  case extendCipherWord(inciph, inciph, [], []) <-
  case extendCipherWord(inciph, outciph, p::pw, c::cw) <-
    Exists cipher1 |
      extendCipherChar(inciph, cipher1, p, c) and
      extendCipherWord(cipher1, outciph, pw, cw)
    %Exists
%Define
=======================================================
%% Mode: extendCipherWordDict(in inciph, out outciph,
%%                            out plain: String, in ciph: String,
%%                            in dict)
Define extendCipherWordDict(inciph, outciph, pw, cw, dict) <-
  activeMember(pw, getDict(dict, length(cw))) and
  extendCipherWord(inciph, outciph, pw, cw)
%Define
=======================================================
%% Mode: extendCipherWordsDict(in inciph, out outciph,
%%                             out plain: [String], in ciph: [String],
%%                             in dict)
Define first
  case extendCipherWordsDict(inciph, inciph, [], [], ?)  <-
  case extendCipherWordsDict(inciph, outciph, pw::pws, cw::cws, dict) <-
    Exists cipher1 |
      extendCipherWordDict(inciph, cipher1, pw, cw, dict) and
      extendCipherWordsDict(cipher1, outciph, pws, cws, dict)
    %Exists
%Define
```

Figure B.1: **Predicate definitions for the cryptogram solver.** Each of these predicates takes a current cipher (inciph) and produces a modified cipher (outciph), possibly with more pairs in it, or fails if no new cipher exists. Predicates extendCipherChar and extendCipherWord handle matching a particular pair of characters or words. Predicates extendCipherWordDict and extendCipherWordsDict turn the plain word (or word list) into an out parameter, drawing the possible words from the given dictionary. Choice mode **first** is used to prevent trying the second case when empty lists are found.

We assume that getDict(dict, n) returns a list of the length n words in dict. activeMember is a Cinnameg membership testing predicate that uses unification, so that it can be used in mode activeMember(out, in).

```
Import "collect/string".  Team
  Example
    sortByLength ["THE", "DOG", "A", "RABBIT"] =
      [["A"], [], ["DOG", "THE"], [], [], ["RABBIT"]];
    buildDictionary("the\ndog\na\nrabbit") =
      [["A"], [], ["DOG", "THE"], [], [], ["RABBIT"]]
  %Example
  ================================================
  Define sortByLength(words) =
    Var lists(*) all =! [].  %% An infinite array
    Loop (n, restwords) = (0, words) matching restwords
      loopcase w::ws => %% Add w to its bucket
        Let len = length(w).
        Make @(lists _ len) =! w::@(lists _ len).
        Continue (max[n,len], ws).
      exitcase [] => map (@) (take n lists)
    %Loop
  %Define
  ================================================
  Define buildDictionary(words) =
    words ~> breakIntoLines ~> map toUpper ~> sortByLength
  %Define
%Team
================================================
Define
  case getDict(dict, n) = dict _ n when n <= length(dict)
  else getDict(?)  = []
%Define
```

Figure B.2: **Handling the dictionary.** BuildDictionary takes a list of words and converts it into a dictionary in the form we are using by sorting it by length (using a *bucket sort*). These functions are not written in logic programming style, since that is not the best style for this kind of job.

Expression x ~> f is the same as $f(x)$. x_n is the n-th member of list x. The members of array it lists are boxes, and @(b) is the value stored in box b. Expression (take n lists) yields a list of the first n boxes in the array. The map function is described in Section 21.7.

```
Define solveFirst(ciphertext: String) =
  Unique
    Let cwords: [String] = ciphertext ~> breakIntoWords.
    Let dict: [[String]] = buildDictionary(lazyFileContents("dict.txt")).
    Try
      Exists outciph, pwords |
        MakeSure extendCipherWordsDict([], outciph, pwords, cwords, dict).
        Writeln["Cipher:  ", $(cwords)].
        Writeln["Plain:  ", $(pwords)].
        Writeln["Cipher:  ", $(outciph)].
      %Exists
    else
      Writeln["No solutions for cipher ", $(cwords)].
    %Try
  %Unique
%Define
```

Figure B.3: **Finishing up.** We assume that file dict.txt holds a dictionary with one word per line. Function lazyFileContents takes the name of the file and produces its contents (a string).

B.2. Exercises

B.1. Write the definition of getDict (Figure B.2) as a logic-programming predicate instead of as a function.

B.2. Write the definition of sortByLength (Figure B.2) as a predicate instead of as a function. Use a style similar to Prolog's.

B.3. Prolog programmers often stray from the ideals of logic programming. This exercises illustrates one reason.

Some kinds of predicates that have out-parameters (from the perspective of how they are used) can be difficult to describe as pure tests. The problem is that they might have a choice of representations for the out-parameters, and be designed to select just one representation. Predicate extendCipherChar is an example. As a pure test, you would like to say that extendCipherChar(inciph, outciph, p, c) is true if outciph is an allowed cipher (meaning that it does not contain two pairs with the same left-hand side or with the same right-hand side) and that it contains pair (p, c) and all pairs in inciph and nothing more.

But that is not quite what it does. In fact, extendCipherChar([(’A’,’Z’)], [(’B’,’Y’), (’A’,’Z’)], ’B’, ’Y’) is false, according to the definition in Figure B.1, because extendCipherChar adds the new pair to the end of the code, not to the beginning.

(a) Would it be significantly more difficult to design extendCipherChar so that it will accept any outcipher that has exactly the required character correspondences? Would it help to make the ciphers sets instead of lists? What would be involved in making the modification?

(b) What practical advantage would there be to modifying extendCipherChar to accept any reasonable outcipher?

(c) Describe what extendCipherWordDict does from a procedural standpoint.

B.4. It would be considerably more efficient for extendCipherWordsDict to determine how many words in the dictionary each cipher word can correspond to, and to handle the word with the fewest possible matches first. After a correspondence is chosen for the first word, the number of possible matches for remaining words that are consistent with the updated cipher shrinks. Once again, select the word that has the fewest possible matches, using the updated cipher, and keep going. Modify the definition of extendCipherWordsDict and related predicates to work that way.

Appendix C

An Object-Oriented Example

C.1. Nim

To play the game of Nim, two players start with a group of stacks of coins. For example, there might be three stacks, initially holding 3, 4 and 5 coins, respectively. Players alternate moves, each removing any number of coins from a single nonempty stack. If all of the stacks are empty when it is your turn, you lose.

C.2. Playing Nim

This chapter contains a program written in Cinnameg in an object-oriented style that plays a game of Nim between the computer and a person. But the program is not written so that it can only do that. Instead, it allows any number of strategies for playing the game to be developed, and, once the strategies have been implemented, it is a simple matter to play any two given players against one another. The human player is represented by a strategy that acts as a proxy by asking the user what to do for each move. The components are as follows.

1. Class AnyNimPlayer is an abstact class whose subclasses represent particular strategies. See Figure C.1.

2. Class UserNimPlayer is the strategy that acts as a proxy for the user. See Figure C.2.

3. Class OptimalNimPlayer implements an optimal strategy. See Figure C.4.

4. Class OptimalNimPlayerSupport defines some functions needed by class OptimalNimPlyer. See Figure C.3.

5. Class NimSupervisor controls the game, alternating moves between the players. See Figure C.5.

6. Module PlayNim builds the objects and starts the supervisor. See Figure C.6.

C.3. How the optimal player works

A *configuration* is a list telling how many coins are in each stack. For a given configuration, write each of the numbers in binary notation, and line them up, aligned at their right ends. For example, for configuration $[4, 3, 7]$, write

```
100
011
111
```

Count the number of 1's in each column. In the example, each of the columns has two 1's in it. Say that a column has *even parity* if it has an even number of 1's, and that it has odd parity otherwise. Say that a configuration is *balanced* if all of the columns have even parity. That is true for configuration $[4, 3, 7]$ shown above. Another balanced configuration is $[1, 3, 5, 7]$. But configuration $[4, 3, 2]$ is not balanced. Its binary representations are as follows.

```
100
011
010
```

Two columns have odd parity, and it only takes one odd parity column to make it unbalanced. We leave it as an exercise to the reader to justify the following two facts.

1. If a configuration is balanced, then any move (removing a positive number of coins from one of the stacks) will result in an unbalanced configuration.

2. If a configuration is unbalanced, then there exists a move that will result in a balanced configuration.

Say that a configuration is *empty* if all of the stacks are empty. Of course, if the configuration is empty, and it is your move, then you lose the game. Notice that an empty configuration is balanced.

If it is your move, and the configuration is unbalanced, you choose a move that will make the configuration balanced. Your opponent has no choice but to make the configuration unbalanced. Once again, you choose a move that balances the configuration. This will continue until the configuration is empty, which has to occur on your opponent's move, since you always see an unbalanced configuration.

C.4. Exercises

C.1. Why is the move method in class AnyNimPlayer virtual?

C.2. Prove that the two claims about balanced configurations are true.

C.3. What would be involved in giving the user a break and offering a different (less smart) strategy? What would you have to write, and what would you have to change?

C.4. Would would you need to do to play two optimal players against one another?

```
Package AnyNimPlayer
===============================================================
                          export
Abbrev Configuration = [Integer].  %% Another name for [Integer]
Class AnyNimPlayer interface
  instanceCapabilities:
    Expect{--virtual--} move: Configuration -> (Integer, Integer).
  constructors:
    Expect inheritAnyNimPlayer: () -> AnyNimPlayer.
%Class
===============================================================
                          implementation
Class AnyNimPlayer implementation
  constructors:
    Define constructor inheritAnyNimPlayer().
%Class
%Package
```

Figure C.1: **Class AnyNimPlayer.** Virtual method move(config) yields a pair (s,n) indicating that the player wants to remove n coins from stack number s. The constructor is only for subclasses to use.

```
Package UserNimPlayer
===============================================================
                          export
Import "AnyNimPlayer".
Class UserNimPlayer extends AnyNimPlayer interface
  instanceCapabilities:
    Expect WriteInstructions: ().
  constructors:
    Expect userNimPlayer: () -> UserNimPlayer.
%Class
====================================================================
                          implementation
Import "collect/string".
Class UserNimPlayer implementation
  constructors:
    Define constructor userNimPlayer()
      beforeCreation:
        Let self's super = inheritAnyNimPlayer().
    %Define
  instanceCapabilities:
    Define
      self's WriteInstructions.  =
        Write["Move: Give stack number then number of coins "].;
      self's move(config) = (col, k) |
        Write["config: ", $(config), "\nMove: "].
        Extract[theInteger(col), theInteger(k)].
    %Define
%Class
%Package
```

Figure C.2: **Class UserNimPlayer.** A constructor in a subclass initializes its inherited variables and items by defining `self's super`.

```
Package OptimalNimPlayerSupport
===========================================================
                          export
Abbrev Configuration = [Integer].
Expect
  oddCount    : Configuration -> Integer;
    %% oddCount(c) counts the odd numbers in c.
  shift       : Configuration -> Configuration;
    %% shift(c) divides each number in c by 2.
  emptyConfig?: Configuration -> Boolean;
    %% emptyConfig?(c) is true if c is empty.
  coinCount   : (Configuration, Integer) -> Integer;
    %% coinCount(cfg, k) returns the number of coins to remove
    %% from the k-th stack to make configuration cfg balanced
  goodStack   : Configuration -> [Integer];
    %% goodStack(c) yields [] if c is a balanced configuration
    %% and yields [k] if c is unbalanced and c_k is one of the
    %% numbers in list c that has a 1 in the leftmost odd-parity column.
%Expect
===========================================================
                        implementation
Import "collect/list".
Define
  case oddCount []        = 0
  case oddCount (x :: xs) = (x `mod` 2) + oddCount xs;
  shift         = map (x |-> x `div` 2);
  emptyConfig? = allSatisfy zero?;
  case coinCount(cfg,k)
    = 0                                    when emptyConfig?(cfg)
    = 2*coinCount(shift cfg, k)            when not(odd?(oddCount cfg))
    = 2*coinCount(shift cfg, k) + 1        when odd?(cfg_k)
    = 2*coinCount(shift(cfg2), k) + 1 |
        Let cfg2 = cfg\{_k = cfg_k - 1}.;
  case goodStack cfg
    = []            when emptyConfig?(cfg)
    = g             when g =/= []
    = []            when not(odd?(oddCount(cfg)))
    = [oddindex(cfg)]
    with Define
      g = goodStack(shift cfg);
      oddindex(lst) = select (k |-> odd?(lst_k)) [1,...].
  %Package
```

Figure C.3: **Package OptimalNimPlayerSupport.** Cinnameg notation $L\{\_k = v\}$ is the list that you get by replacing the k-th member of L by v. The **with** clause gives definitions that are shared among the preceding cases for goodStack.

```
Package OptimalNimPlayer
===========================================================
                          export
Import "AnyNimPlayer"; "misc/random".
Class OptimalNimPlayer extends AnyNimPlayer interface
  constructors:
    Expect optimalNimPlayer: () -> OptimalNimPlayer.
%Class
===========================================================
                        implementation
Class OptimalNimPlayer implementation
  constructors:
    Define constructor optimalNimPlayer()
      beforeCreation:
        Let self's super = inheritAnyNimPlayer().
    %Define
  instanceCapabilities:
    Define self's pickAMove(cfg) =
      Let k = randomRange(1, length cfg).
      If cfg_k > 0
        then (k, randomRange(1, cfg_k))
        else pickAMove cfg
      %If
    %Define
    Define self's move(config) =
      Choose matching goodStack(config)
        case [] => pickAMove(config)
        case [k] => (k, coinCount(config, k))
      %Choose
    %Define
%Class
%Package
```

Figure C.4: **Class OptimalNimPlayer.** randomRange(a, b) returns a pseudo-random integer in the range from a to b.

```
Package NimSupervisor
==============================================================
                         export
Import "AnyNimPlayer".
==============================================================
Class NimSupervisor interface
  instanceCapabilities:
    Expect Play: [Integer] -> ().
  constructors:
    Expect nimSupervisor: (AnyNimPlayer, AnyNimPlayer) -> NimSupervisor.
%Class
==============================================================
                          implementation
Import "collect/list".
==============================================================
Class NimSupervisor implementation
  instanceInformation:
    Item player1, player2: AnyNimPlayer.
constructors:
  Define constructor nimSupervisor(p1,p2)
    beforeCreation:
      Let self's player1 = p1.
      Let self's player2 = p2.
  %Define
instanceCapabilities:
  Define self's Play(startConfig).  =
    Let players = [self's player1, self's player2].
    Let numStacks = length(startConfig).
    Loop (config, mover) = (startConfig, 1)
      exitcase allSatisfy zero? config =>
        Writeln["Player ", $(mover), " loses."].
      do Match (which, howmany) = (players _ mover)'s move(config).
         Write ["Player ", $(mover), " removes ",
                $(howmany), " from stack ", $(which)].
      exitcase which < 1 or which > numStacks or
               config_which < howmany =>
        Writeln["\nPlayer ", $(mover), " loses for cheating."].
      loopcase else =>
        Let newconfig = config{_which = config_which - howmany}.
        Writeln[" resulting in configuration ", $(newconfig)].
        Continue (newconfig, 3 - mover).
    %Loop
  %Define
%Class
%Package
```

Figure C.5: **Class NimSupervisor**

```
Package PlayNim
Import "OptimalNimPlayer"; "UserNimPlayer"; "NimSupervisor".
Import "collect/string".
Flexible
  Execute
    Writeln["What is the initial configuration?"].
    Writeln["Write as a list, such as [4,2,5]."].
    Extract[$(start: [Integer])].
    Let player1 = userNimPlayer();
        player2 = optimalNimPlayer();
        sup     = nimSupervisor(player1, player2).
    player1's WriteInstructions.
    Writeln.
    Writeln["You are player 1.  You start."].
    sup's Play(start).
  %Execute
%Flexible
%Package
```

Figure C.6: **Program PlayNim,** which sets up the Nim game and starts playing. Flexible requests relaxed type checking that inserts implicit up-conversions (for example, from UserNimPlayer to AnyNimPlayer).

Bibliography

[1] *1990 International Conference on Computer Languages, March 12-15 1990, New Orleans, Louisiana, USA*. IEEE Computer Society, 1990.

[2] Annika Aasa, Sören Holmström, and Christina Nelson. An efficiency comparison of some representations of purely functional arrays. *BIT*, 28:490–503, 1988.

[3] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, Berlin, 1996.

[4] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 1985.

[5] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, second edition, 2007.

[6] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, England, 1998.

[7] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, Reading, MA, fourth edition, 2005.

[8] Ken Arnold, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, Reading, MA, third edition, 2005.

[9] John Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, June 2003.

[10] John Backus. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. In *Proceedings of the International Conference on Information Processing*, pages 125–132, 1959. UNESCO.

[11] John Backus. Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs. *Communications of ACM*, 21(8):613–641, 1978.

[12] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam, second edition, 1984.

[13] Henk P. Barendregt. Lambda calculi with types. In Abramsky, Gabbay, and Maibaum, editors, *Handbook of Logic in Computer Science*, volume II. Oxford University Press, 1992.

[14] Jon Bentley. *Programming Pearls*. Addison-Wesley, Reading, MA, second edition, 2000.

[15] Bernard Billoud, Milutin Kontic, and Alain Viari. Palingol: A declarative programming language to describe nucleic acids' secondary structures and to scan sequence databases. *Nucleic Acids Research*, 24(8):1395–1403, 1996.

[16] Richard Bird. *Introduction to Functional Programming Using Haskell*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1998.

[17] Graham M. Birtwistle, Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. *Simula Begin*. Auerbach, Phildelphia, PA, 1973.

[18] Hans-Juergen. Boehm. Space efficient conservative garbage collection. In *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 39(4), pages 490–499, 2004.

[19] Timothy Budd. *A Little Smalltalk*. Addison-Wesley, Reading, MA, 1987.

[20] Rod M. Burstall. Proving properties of programs by structural induction. *Computer Journal*, 12(1):41–48, 1969.

[21] Rod M. Burstall. Program proving as hand simulation with a little induction. In *IFIP Congress*, pages 3–10, 1974.

[22] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(3):431–447, December 1985.

[23] Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, 1987.

[24] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Technical report, DEC/Compaq Systems Research Center, 1989. Research Report 52.

[25] Frank M. Carrano and Janet J. Prichard. *Data Abstraction and Problem Solving with Java*. Addison-Wesley, Reading, MA, updated edition, 2004.

[26] Giuseppe Castagna. *Object-Oriented Programming: A Unified Foundation*. Springer-Verlag, Berlin, 1997.

[27] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.

[28] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(3):137–167, 1959.

[29] A. Church. The calculus of lambda conversion. *Annals of Mathematical Studies*, 6, 1941.

[30] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.

[31] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog: Using the ISO Standard*. Springer-Verlag, Berlin, fifth edition, 2003.

[32] A. Colmerauer and P. Roussel. The birth of Prolog. *ACM SIGPLAN Notices*, 28(3):37–52, 1993.

[33] William Cook. Object-oriented programming versus abstract data types. In *Foundations of Object-Oriented Languages*, number 489 in Lecture Notes in Computer Science, pages 151–178. Springer-Verlag, Berlin, 1991.

[34] N. J. Cutland. *Computability: An Introduction to Recursive Function Theory.* Cambridge University Press, Cambridge, England, 1980.

[35] O. J. Dahl, E. W. Dijstra, and C. A. R. Hoare. *Structured Programming.* Academic Press, New York, 1972.

[36] J. Darlington, P. Henderson, and D. A. Turner. *Functional Programming and Its Applications: An Advanced Course.* Cambridge University Press, Cambridge, England, 1982.

[37] N. G. DeBruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[38] Edsgar W. Dijstra. Goto statement considered harmful. *Communications of ACM*, 11(3):147–149, 1968.

[39] R. Finkel, V. W. Marek, and M. Truszczynski. Constraint lingo: Towards high-level constraint programming. *Software: Practice and Experience*, 34(15):1481–1504, December 2004.

[40] R. W. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967.

[41] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, MA, 1983.

[42] J. B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of ACM*, 16(12):683–696, December 1975.

[43] M. J. C. Gordon. *The Denotational Description of Programming Languages.* Springer-Verlag, Berlin, 1979.

[44] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* Addison-Wesley, Reading, MA, third edition, 2005.

[45] Paul Graham. *ANSI Common Lisp.* Prentice-Hall, Englewood Cliffs, NJ, 1996.

[46] R. E. Griswold, J. F. Poage, and I. P. Polansky. *The SNOBOL4 Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, 1971.

[47] Ralph E. Griswold and Madge T. Griswold. History of the Icon programming language. In Thomas J. Bergin and Richard G. Gibson, editors, *History of Programming Languages II*, pages 599–624. Addison-Wesley, 1996.

[48] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, third edition, 2000.

[49] Dick Grune, Henri E. Bal, Ceriel J. H. Jacobs, and Koen G. Langendoen. *Modern Compiler Design.* Wiley, New York, 2000.

[50] Samuel P. Harbison. *Modula-3.* Prentice-Hall, Englewood Cliffs, NJ, 1992.

[51] J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of AMS*, 146:29–60, 1969.

[52] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-Calculus.* Cambridge University Press, Cambridge, England, 1986.

[53] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[54] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of ACM*, 12(10):576–580, 1969.

[55] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica*, 2:335–355, 1973.

[56] C. J. Hogger. *Introduction to Logic Programming.* Academic Press, Orlando, FL, 1984.

[57] Cay Horstmann and Gary Cornell. *Core Java2, Volume I: Fundamentals.* Prentice-Hall PTR, Englewood Cliffs, NJ, seventh edition, 2004.

[58] Cay Horstmann and Gary Cornell. *Core Java2, Volume II: Advanced Features.* Prentice-Hall PTR, Englewood Cliffs, NJ, seventh edition, 2004.

[59] P. Hudak. Conception, evolution and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[60] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III).*

[61] IEEE Computer Society. IEEE standard 754-1985 for binary floating-point numbers. Technical report, IEEE Computer Society, 1985.

[62] K. Jensen and N. Wirth. *Pascal User Manual and Report.* Springer-Verlag, Berlin, third edition, 1985.

[63] Alan Kay. The early history of Smalltalk. In Thomas J. Bergin and Richard G. Gibson, editors, *History of Programming Languages II*, pages 511–598. Addison-Wesley, 1996.

[64] Donald E. Knuth. Structured programming with goto statements. *ACM Computing Surveys*, 6(4):261–301, December 1974.

[65] R. A. Kowalski. *Logic for Problem Solving.* Elsevier North Holland, New York, 1979.

[66] W. LaLonde and J. Pugh. *Inside Smalltalk, Volume I.* Prentice Hall, Englewood Cliffs, NJ, 1990.

[67] W. LaLonde and J. Pugh. *Inside Smalltalk, Volume II.* Prentice Hall, Englewood Cliffs, NJ, 1990.

[68] W. LaLonde and J. Pugh. Subclassing ~= subtyping ~= is-a. *Journal of Object-Oriented Programming*, 3(5):57–62, January 1991.

[69] Peter van der Linden. *Expert C Programming: Deep C Secrets.* SunSoft Press, 1994.

[70] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, Reading, MA, 1996.

[71] C. H. Lindsey. A history of ALGOL 68. In Thomas J. Bergin and Richard G. Gibson, editors, *History of Programming Languages II*, pages 27–96. Addison-Wesley, 1996.

[72] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanism in CLU. *Communications of ACM*, 20:564–576, 1977.

[73] Z. Manna, S. Ness, and J. Vuillemin. Induction methods for proving properties of programs. *Communications of ACM*, 16:491–502, 1973.

[74] Kim Marriott and Peter J. Stuckey. *Programming with Constraints.* MIT Press, Cambridge, MA, 1998.

[75] Bertrand Meyer. *The Eiffel Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, 1992.

[76] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Communications of ACM*, 22(5):271–280, 1979.

[77] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 16(3):348–375, 1978.

[78] R. Milner, M. Tofte, and R. M. Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1990.

[79] John C. Mitchell. *Foundations for Programming Languages.* MIT Press, Cambridge, MA, 1996.

[80] Lee Naish. *Negation and Control in Prolog.* Number 238 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1986.

[81] Peter Naur (ed.). Revised report on the algorithmic language ALGOL 60. *Communications of ACM*, 6(1):1–17, 1963.

[82] Tobias Nipkow, Lawrence C. Paulson, and Marlus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher Order Logic.* Springer-Verlag, Berlin, 2002.

[83] Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, Cambridge, England, 1998.

[84] F. G. Pagan. *Formal Specification of Programming Languages.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[85] Stephen K. Park and Keith W. Miller. Random number generators: Good ones are hard to find. *Communications of ACM*, 31(10):1192–1201, October 1988.

[86] Lawrence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, Cambridge, England, second edition, 1996.

[87] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, Cambridge, MA, 2002.

[88] Jonathan Rees and William Clinger. The revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 21(12):37–79, 1986.

[89] Jonathan Rees and William Clinger. The revised^5 report on the algorithmic language Scheme. Technical report, Massachussetts Institute of Technology, 2001.

[90] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of ACM*, 12(1):23–41, 1965.

[91] J. B. Rosser. Highlights of the history of the lambda-calculus. *Annals of the History of Computing*, 6(4):337–349, 1984.

[92] H. Rutishauser. *Description of ALGOL 60.* Springer-Verlag, Berlin, 1967.

[93] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Boston, MA, 1986.

[94] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Shonberg. *Programming with Sets: An Introduction to SETL.* Springer-Verlag, Berlin, 1986.

[95] D. S. Scott and D. Strachey. Towards a mathematical semantics for computer languages. In *Symposium on Computers and Automata*, pages 19–46, Brooklyn, NY, 1971. Polytechnic Press.

[96] R. Sedgewick. Implementing Quicksort programs. *Communications of ACM*, 21:847–857, 1978.

[97] D. Shafer, S. Herndon, and L. Rozier. *Smalltalk Programming for Windows.* Prima Publishing, Rocklin, CA, 1994.

[98] D. N. Smith. *IBM Smalltalk: The Language.* Benjamin/Cummings, Redwood City, CA, 1994.

[99] Guy L. Steele Jr. *Common LISP.* Digital Press, Burlington, MA, 1984.

[100] Guy L. Steele Jr. and Richard P. Gabriel. The evolution of Lisp. In Thomas J. Bergin and Richard G. Gibson, editors, *History of Programming Languages II*, pages 233–330. Addison-Wesley, 1996.

[101] J. E. Stoy. *Denotational Semantics.* MIT Press, Cambridge, MA, 1977.

[102] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, third edition, 2000.

[103] Simon Thompson. *Haskell: The Craft of Functional Programming.* Addison-Wesley, Reading, MA, second edition, 1999.

[104] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 1–16, Berlin, 1985. Springer-Verlag.

[105] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Singzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker. *Revised Report on the Algorithmic Language ALGOL 68*. Springer-Verlag, Berlin, 1976.

[106] L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl*. O'Reilly, Sebastopol, CA, second edition, 1996.

[107] D. H. D. Warren. Logic programming and compiler writing. *Sofware Practice and Experience*, 10(2):97–125, 1980.

[108] P. Wegner. Dimensions of object-based language design. *ACM SIGPLAN Notices*, 22(12):168–182, 1987. Proceedings of OOPSLA 87.

[109] Ulf Wiger. 20 years of industrial functional programming. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, ACM SIGPLAN Notices 39(8), page 162, 2004.

[110] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, MA, 1993.

[111] N. Wirth. The module: A system structuring facility in high-level programming languages. In *Language Design and Programming Methodology*, number 79 in Lecture Notes in Computer Science, pages 1–24, 1979.

[112] Nicklaus Wirth. *Programming in Modula-2*. Springer-Verlag, Berlin, second edition, 1982.

[113] Niklaus Wirth. From Modula to Oberon. *Software Practice and Experience*, 18(7):661–670, 1988.

[114] Stephen Wolfram. *The Mathematica Book*. Wolfram Media, fifth edition, 2003.

[115] W. A. Wulf and M. Shaw. Global variables considered harmful. *ACM SIGPLAN Notices*, 8(2):80–86, 1973.

Index