Final Honour School of Computer Science
Final Honour School of Mathmatics and Computer Science
Final Honour School of Engineering and Computing Science
Degree of Master of Science in Computation

Compilers

Lecture Notes

Mike Spivey

Another vital $\mathrm{P_tR.G.}$ project

Oxford University Computing Laboratory

Draft of 3.vii.2003

Chapter 1

Introduction

This book is about the techniques that are used in implementing conventional programming languages like PASCAL or C: specifically, about compilers that translate these languages into machine code. There are several reasons why this technology is worth studying:

- Compilers are part of the standard toolbox of computer programming: few programs that are written today are written without their help. Understanding how compilers work can help us to use them better, producing more robust programs that fully exploit what the compiler can do for us.
- Many software systems become more simple if a specialized language is part of the design: for example, a graphics package might become drastically simpler to implement and to use if it contained a little language for describing pictures, instead of just being a subroutine library for an existing language. The techniques covered in this book (particularly interpreters) can be used to build this kind of little language.
- Studying a compiler gives us an opportunity of seeing how a medium-size program fits together, in a domain where the software architecture has evolved into a well-understood form. Many algorithms and data structures that are used in building compilers find applications in other parts of software engineering.

In a short book like this, there isn't space to cover everything, so we make things easy for ourselves:

- We'll use a powerful implementation language (a dialect of ML). Even though many industrial-strength compilers are written in C, there's little reason for this, now that high-quality implementations of ML and related languages exist. The powerful features of ML are just what are needed to write compilers, so that makes it an appropriate choice. C has its place in tasks like implementing operating systems, where low-level interaction with the hadware of a computer is actually necessary.
- We'll keep each phase of the compiling process separate – even if this affects the speed of the compiler and the amount of memory it needs. This helps to split the compiler cleanly into modules; the chief disadvantage is that the compiler needs to keep the whole source program

in memory at once, and this limits the size of program that can be compiled.

- We'll choose a simple source language called PICOPASCAL that is more-or-less a subset of PASCAL.
- We'll do almost no optimization, concentrating on generating reasonably good code in the simplest way we can.
- In the first half of this book, we'll use an invented machine instead of any particular, real machine. The invented machine has been designed to make it easy to generate code for high-level language constructs, avoiding some of the problems we would face in using a real machine.
- We won't produce good error messages, and we'll abandon the whole compiling process after finding the first error, instead of trying to fix things up so that compiling can continue. This will make our compilers a little annoying to use, because it will only be possible to correct one error for each run of the compiler.

The author leaves it up to the reader to judge how much these compromises make our compilers unrealistic.

We'll want to discuss compiler techniques one at a time, but we'll also want to see how the techniques fit together to make a working compiler. To help with this, the examples in successive chapters build up into a complete compiler for a realistic programming language of about the same sophistication as Pascal or C.

1.1 Compilers and interpreters

As we study programming languages and the mechanisms needed to implement them, we will use both *compilers* and *interpreters*, and it's a good idea to begin by defining the difference between them. We'll view *compilers* as being the most important form of language implementation, but we'll also use *interpreters* extensively, as an easy way of studying language features before we look at how to implement them in machine terms.

A compiler implements a programming language in an indirect way, because it translates a program into another programming language – machine code – that has to be executed separately in order to produce the behaviour specified by the program. An interpreter does not use translation, but instead takes as its inputs *both* the high-level program *and* its input data, and produces as output the results of the computation described by the program.

In order to make the distinction precise, it is helpful to ignore for a moment the fact that we intend to build both compilers and interpreters by writing computer programs. Taking this step, we can define an interpreter to be a computing device that takes a program (written in some language) and produces the behaviour that the program describes. Under this definition, we are already familiar with a one class of interpreters: a computer is itself an interpreter for its own machine language. Thus whenever a computer program runs, at least one interpreter is involved, namely the computer itself.

We define a compiler as a computing device that takes a program written in one language and produces another program, written in a different language, that describes the same behaviour. Typically, the input language of the compiler will be a high-level programming language, and the output language will be the machine code of a computer.

With this distinction in mind, we can contemplate building compilers and interpreters by writing computer programs, which are no more and no less than descriptions of the behaviour of a computing device. If we were given a bare machine with no software at all, we could begin by writing programs in machine language and loading them into the machine, perhaps using binary switches and push-buttons. Our first implementation of a programming language other than machine code might be in the form of an interpreter, itself written in machine code. Running a program written in a high level language would then involve a tower of two interpreters; the lower one is the computer itself, obeying a machine-code program, and the upper one is the interpreter program, written in machine code and obeying a program written in a high-level language.

Later, we might build a compiler for the new machine, by writing it in machine code, by writing it in the language accepted by our interpreter, or by writing the compiler for a different machine entirely, translating high-level programs there into the machine language of our new machine, and copying them across before running them. In each case, the compiler describes the process of translating constructs of the high-level language into equivalent programs written in the machine code of our machine. Once this translation has been done, the compiler plays no further part in the execution of programs.

A common mistake is to describe an interpreter as a program or device that translates a program into machine code as each part of it is executed; this is a mistake because translation – the act of producing a program in a different language but with the same meaning – is not necessarily a part of executing a program in an interpreter; we will see that some interpreters set up a correspondence between constructs in a high-level langauage and the actions that they describe, but this correspondence is not brought into effect by means of a translation; rather, the correspondence determines the way in which the interpreter itself behaves when it is presented with different kinds of high-level program.

Every compiler written as a program involves three programming languages: the *source language* of programs that are inputs to the compiler, the *object language* of programs that are outputs, and the *implementation language* in which the compiler itself is written. These languages need not all be different. One bright idea is to choose the implementation language to be the same as the source language, so that the compiler can compile itself: this is called *bootstrapping*. Once the project is off the ground, extensions and improvements to the compiler can be made by modifying its source code, then using an old version of the compiler to compile the new one. There are several ways to get the whole thing started: one might be to quickly write a slow interpreter for the source language, and use it to run the compiler the first time.

In this book, we'll look at three kinds of language implementations, all of them practically useful:

- *Source-level interpreters* operate on a representation of the program that is not very far removed from the source text, and do only a minimal amount of analyisis of the program before they start to carry out its actions. In source-level interpreters in this book, the program will be converted into a convenient tree-like form, but no further analysis is done. We use source-level interpreters because they are the easiest form of language implementation to create. We shall use them to explore the meaning of programming languages and the consequences of language design decisions.

Source-level interpreters are used in practice for scripting languages like Tcl, AWK and the UNIX shell. Although it does not lead to very fast execution, if the primitive operations of a language are expensive in themselves, then the relative overhead of implementing the constructs of the language by source-level interpretation may well be small enough to give adequate performance. For example, in the UNIX shell, the primitive operations involve starting new UNIX processes to carry out commands; the time taken to create a new process is much greater than the time consumed by the source-level interpreter in preparing the command for execution, so the overhead for the interpreter is negligible.

- An implementation based on an *abstract machine* is a combination of a compiler and an interpreter. The compiler translates the source program into the machine language of a specially-invented machine, called *abstract* because it may not exist as a piece of hardware. The interpreter executes programs written in the abstract machine language.

  The task of translating the source program into abstract machine code can be significantly simpler than generating code for a real machine, because the abstract machine can be designed in a simple, uniform way to have exactly the operations needed for translating the source language. The performance of implementations based on abstract machines can be much better than source-level interpreters, because significant analysis can be done during the compiling phase that saves work during the execution phase.

  Abstract machines are used in practice where software portability is important. By writing one compiler that generates code for an abstract machine, and several very simple interpreters that implement the abstract machine language on a number of different real machines, it is possible to run exactly the same software on many machines with different architectures and obtain reasonable performance. This is the idea behind Java. The same idea is used to implement the Objective CAML language used in this book.

  If compiling for an abstract machine is so much easier than compiling for a real machine, it's reasonable to ask why real machines are not made more like our abstract machines. This has actually been done in machines like Sun's JavaStation (now defunct). The advantages for compiler simplicity are obvious, but the compensating disadvantage is lack of performance. By making the hardware resources of a machine less uniform, and requiring a compiler to allocate those resources as it compiles a program, it is possible to get better performance out of the resources as the program runs. The best example of this is that real machines have a fixed, finite set of fast registers, and they are replaced in our abstract machines by an arbitrarily large stack of temporary storage locations. This makes compiling easier, because we never need to worry about running out of working space, but faster machines can be built with a fixed set of registers than with a less predictable stack.

- The third kind of language implementation we'll study is a compiler that produces machine code (or more likely assembly language) for an actual machine. Like many compilers, the ones we shall build will be split into two parts: a *front end* that translates the source program into an intermediate form, and a *back end* that translates the intermediate form into machine code. This is a good idea, because it divides the compiling problem into two more manageable halves. Also, if the intermediate form is independent of the target machine, it eases the problem of making a compiler for a new machine: only the back end needs to change.

  There are many possible intermediate forms that provide the interface between the front end and the back end. We shall choose to make the intermediate form be very similar to the abstract machine code that we use in our abstract machine based interpreters, with the slight difference that the instructions are arranged in a tree structure instead of a linear stream. Thus the front end of our compilers will be almost exactly the same as the compiler part of our abstract machine based implementation, and the back end will be responsible for realizing the operations of the abstract machine in terms of the instructions available on a target machine.

In each of these styles of language implementation, our aim will be to understand the principles, rather than produce implementations with the best performance. This is particularly true of the two kinds of interpreter-based implementations. Nevertheless, I've included some hints at appropriate places in the book that explain how to make implementations that have reasonable performance.

1.2 Representing programs

Compilers are big programs, and the way to make a big program easy to understand is to build it from smaller modules, paying careful attention to the interfaces between each module and the others. These interfaces should be simple and have a clear meaning, so that we can begin to understand the function of each module by understanding the interfaces that link it to other modules in the program. A good way of building a compiler from modules is to design it as a chain of transformations, each taking as input a representation of the program being compiled, and producing as output another representation of the program that is closer to the final object code in some way. We call each transformation a *phase* of the compiling process.

The simplest way of structuring a compiler makes each phase into a separate program or *pass* that consumes the whole of its input and produces the whole of its output before the next pass begins. In old-fashioned compilers, designed to work on computers with small memories, each pass might read its input from a disk file and write its output to another disk file, ready for input by the next pass. If memory is more plentiful, we can make the input and output of each pass be data structures in memory, thereby saving the time needed for disk I/O at the cost of limiting the maximum size of program that can be compiled by the size of the computer's memory. In practice, this is rarely a serious problem, and useful compilers can be built that keep everything in memory throughout.

The compilers I shall describe in this book are all structured so that, as far as possible, each conceptual phase is implemented as a physically separate pass. This makes the compilers easy to understand, because each module does exactly one job. It is possible to combine several phases into a single pass, and there are advantages in doing so. For example, many compilers combine the process of checking the syntax of the source program with the process of checking that variables are declared and used properly into a single pass over the program. Combining phases into a single pass saves the time needed to create a data structure that represents the program in one pass and to decode it in the next pass, but it does make the compiler more difficult to understand. Once you understand how to make a multi-pass compiler like the ones in this book, however, merging the passes together is quite easy.

At the other end of the spectrum from our multi-pass compilers are compilers that combine all phases into a single pass over the program. With a suitably restricted source language, it is possible for the compiler to complete all the analysis of the first part of a source program and produce object code for it before beginning to process later parts of the program. Such single-pass compilers can be made very fast, and they need little storage for their intermediate results, because each piece of program can be processed into object code and discarded by the compiler before the next one is considered.

Despite their comparative slowness, however, multi-pass compilers provide a good way to begin studying compiling techniques, because the parts are kept as well separated as possible. In a sense, we are practising another separation of concerns in choosing to begin with them: we are separating the concern of understanding the compiling process from the concern of making the compiler go fast.

The rest of this chapter is an overview of the data structures that we shall use in our compilers and interpreters to represent programs and parts of programs. As an example, I shall use a tiny fragment of program in a Pascal-like language that implements Euclid's algorithm:

```
while x <> y do
  (* Inv: gcd(x, y) = gcd(x0, y0) *)
  if x < y then
    y := y - x
  else
    x := x - y
  end
end
```

This program computes the greatest common divisor (*gcd*) of two positive integers x and y. If $x < y$ then $y - x$ is also a positive integer, and the *gcd* of x and $y - x$ is the same as the *gcd* of x and y. Thus the assignment $y := y - x$ reduces the value of y without changing $gcd(x, y)$. Similarly, when the assignment $x := x - y$ is executed, this reduces x without changing $gcd(x, y)$. When eventually x and y are equal, their common value must be the *gcd* of the original numbers x_0 and y_0.

During this very quick tour of the stages in compiling a program, I shall use several terms without giving their definitions. These terms are shown in *italics*, with a reference to a chapter of the book where you will find a definition and discussion of the concepts they stand for.

1.3 Source text

The input to almost every compiler or interpreter is a text file, perhaps created with a text editor. It's important to realize that this file is simply a stream of characters, and *all* the structures that we use to understand and write programs must (if the compiler is to make use of them) be recovered by analysing the text. To emphasize this point, here is our example program, presented without any layout, so that the eye no longer recognizes the structure of the program from its appearance on the page:

```
while x <> y do (* Inv: gcd(x, y) = gcd (x0, y0 ) *) if x
< y then y := y - x else x := x - y end end
```

If layout is not significant in our source language, then this program is equivalent to the preceding one, and the compiler must be able to recover all relevant structural information from the plain stream of characters.

1.4 Token stream

The first phase in all our compilers and interpreters takes the stream of characters that is the source program and divides it into meaningful groups, classifying these groups or *tokens* as identifiers, punctuation marks, keywords of the language, and so on. The output of this phase of *lexical analysis* (Chapter 3) is a stream of tokens, like this:

> *While*, *Ident* "x", *Neq* , *Ident* "y", *Do*, *If*, *Ident* "x",
> *Less*, *Ident* "y", *Then*, *Ident* "y", *Assign*, *Ident* "y",
> *Minus*, *Ident* "x", *Else*, *Ident* "x", *Assign*,
> *Ident* "x", *Minus*, *Ident* "y", *End*, *End*.

In forming this sequence of tokens, the lexical analyser has recognized that the word “while” should be read as a single unit, and that it is a keyword of the source language. The next symbol “x” is a unit by itself, and it has been recognized as an identifier; and so on.

Lexical analysis naturally discards spaces from the program, once they have done their job of separating each token in the program from its neighbours; this is also a good time to discard comments, because they should have no effect on the rest of the compiling process. Discarding comments and layout information at this stage makes it easier to build the next part of the compiler, because doing so simplifies the job that it must do: once the layout and comments have been stripped away, the very next thing after the keyword do is the if that starts the loop body.

1.5 Abstract syntax tree

The next phase of compilation takes the stream of tokens and tries to match it with the grammatical rules of the programming language, building a tree that reflects the way the source program is structured; this phase is called *syntax analysis* or *parsing* (Chapter 4). For example, the rules of this programming language might include the fact that a while loop has the form

> while *expression* do *statements* end

so that (unlike some other languages) each while has a matching end. The compiler must reflect this fact, and use it to determine how many statements in the program fall under the control of the while.

Our example program might be represented by the tree structure shown in Figure 1.1. Each construct in the program is represented by a single node, with the components of the construct as its children.

This tree makes it immediately apparent, for example, that our fragment of program consists of a single while loop, that the body of the while loop is an if statement, and so on; this makes it a good representation for the next phases of analysis, which treat the program according to its syntactic structure. Abstract syntax trees are the main data structure used in our source-level interpreters. Instead of operating directly on the textual form of the source program, these will operate on the abstract syntax tree; this makes it easy to build an interpreter whose actions are determined by the structure of the program. It is still fair to say that these interpreters operate at the level of the source program, because it would be possible to recover the source program by printing out the tree. The interpreters use an additional module that represents the memory states of the running program.

There is no compelling reason for separating lexical analysis from syntax analysis, because the techniques that are used to build syntax analysers are powerful enough to solve the problem of lexical analysis also. It is more a matter of convenience and efficiency: convenience, because the syntax analysis problem becomes simpler if lexical analysis is treated separately, and efficiency because it is possible to do lexical analysis in a very efficient way. This efficiency matters because in many compilers (especially those that do not carry out extensive optimisation) it is the lexical analyser that consumes the largest fraction of the compiler's running time. The lexical analyser is the only part of the program that must deal with the source program one character at a time, and separating it makes it easier to tune for speed if necessary.

1.6 Symbol table and annotated tree

The next task carried out by our compilers will be to make a *symbol table* or *environment* (Chapter 6) for the program, containing the variables that it uses, with their types and possibly the storage locations that will be used for them in the object program. Our very first compilers will omit this step, but it is certainly needed once we consider languages where variables must be declared and can have different types.

In our example, there were no declarations for the variables x and y; but let's suppose that they were both declared with type integer, and that the

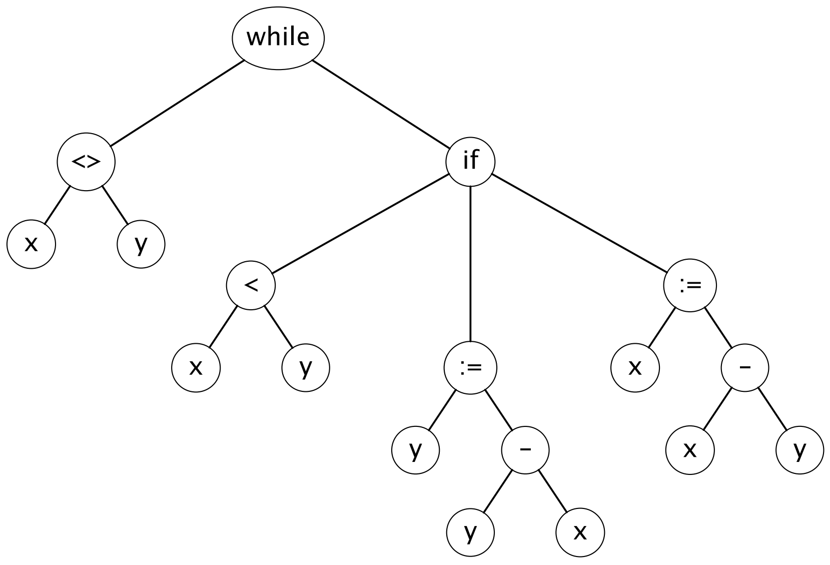

Figure 1.1: *Abstract syntax tree*

compiler has allocated them fixed run-time addresses:

Name	Type	Address
x	*integer*	100
y	*integer*	104

After building this table, the compiler can check that each statement and expression in the program obeys the type rules of the programming language: for example, the expression x < y asks for one integer to be compared with another, giving the Boolean result that is needed for the test of an if statement, and so on. When the time comes to generate object code for the program, the compiler will be able to use address 100 wherever x appears in the program, and address 104 wherever y appears. In our compilers, this information is communicated to the later phases of the compiler by adding *annotations* to the abstract syntax tree. An annotation is attached to each identifier in the tree, telling what are its type and its run-time address (Figure 1.2).

For realistic programming languages, the symbol table needs a much more elaborate structure than is shown here. The types of variables can be more complex, including arrays and records as well as simple types like integers and Booleans. Variables may be local to a procedure instead of global to the whole program, so they may have run-time addresses that are not absolute, but relative to the block of storage allocated to a particular activation of the procedure. The same identifier may be used several times in the program to name several local variables, so it is not sufficient to make a single table of meanings for identifiers. Also, the task of assigning absolute, numeric addresses to even the global variables is often delegated by the compiler to the assembler or linker that processes the compiler's output. All these refinements will be added at appropriate places in the book, as we steadily increase the sophistication of our compiler and the language it implements.

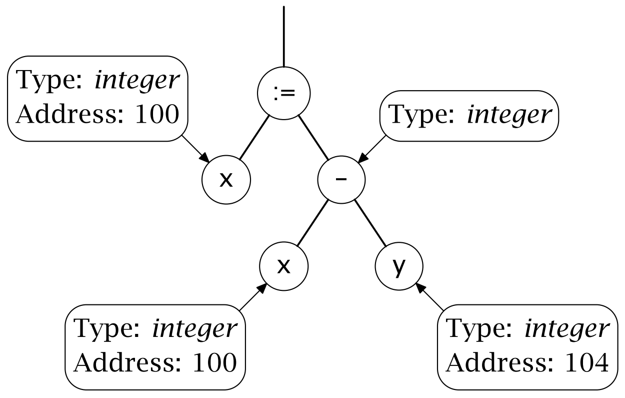

Figure 1.2: *Annotated syntax tree*

1.7 Abstract machine code

In the first half of this book, we shall study compilers that generate code for an invented abstract machine, rather than any real computer. In fact, there will be a series of abstract machines that slowly become more sophisticated as we add features to the machine that are needed to support various features of a programming language; but what all these machines share is an organisation based on a stack that holds temporary values during the evaluation of expressions. Although real machines have been built which have a stack for this purpose, virtually all modern computers have a fixed set of named registers instead. There are several ways in which generating code for register machines is more difficult than generating code for stack machines: for example, often the contents of registers must be saved when a procedure is called and restored when it returns, so that the actions of the procedure being called do not interfere in an undesirable way with the actions of the procedure that called it. These complications do not arise in our abstract machine, because the evaluation stack can grow as much as necessary to accomodate all the procedures that are active.

The function of the *intermediate code generator* (Chapter 5) in our compiler is to produce code for the abstract machine from the annotated syntax tree. The term *intermediate code* refers to the idea (which we take up later) that the final, target machine code output from a compiler is obtained by taking the intermediate code and translating it into the instructions of the target machine; thus the intermediate code comes half-way between the source program and the target program. In the first half of the book, we avoid the complexities of the translation process, and simply build an interpreter or *simulator* for the abstract machine code.

Such a simulator will only run programs at a fraction of the speed that could be achieved by translating them into real machine code, but it allows the programming language to be implemented in a way that is almost completely portable from one machine to another. We shall use a simulator to define the meaning of the abstract machine code used in this book, just as we use interpreters to define and explore the meaning of the source language. Using a simulator also means that we can build working implementations of small languages throughout the book, and can leave the treatment of ma-

```
L1 :
      LOAD 100
      LOAD 104
      JUMPC Eq L2
      LOAD 100
      LOAD 104
      JUMPC Geq L3
      LOAD 104
      LOAD 100
      BINOP Minus
      STORE 104
      JUMP L4
L3 :
      LOAD 100
      LOAD 104
      BINOP Minus
      STORE 100
L4 :
      JUMP L1
L2 :
      STOP
```

Figure 1.3: *Abstract machine code*

chine code for real machines until the last few chapters.

Returning to our implementation of Euclid's algorithm, Figure 1.3 shows the sort of abstract machine code that would be generated from the example program.[1] We shall be looking later at the full set of instructions needed to translate programs, so it should be enough for now to point out some of the instructions used in this particular translation. The first few instructions implement the test for the while loop

```
while x <> y do ...
```

First come two *Load* instructions that put the current values of x and y on the stack, using the addresses 100 and 104 that were assigned during semantic analysis. Then comes an instruction *Jumpc Eq* that compares these two values and jumps to the label L_2 (at the end of the program) if they are equal; this reflects the fact that a while loop terminates when its test evaluates to false. If the value of the test was true, execution continues with the next instruction, which is the beginning of the if test. The loop body ends with a *Jump* instruction that leads back to the very start of the loop, so after the body has been executed, the test is evaluated again for the next iteration. This style of intermediate code is called *postfix code* because the stack-based nature of the abstract machine means that operators appear in 'postfix position', *after* their operands rather than between them as in conventional infix notation.

[1] Actually, there will be several versions of our abstract machine that increase in power and realism as the language we implement increases in sophistication. The code shown here gives the flavour of all of them, but is slightly simpler than some.

The code in Figure 1.3 illustrates that even intermediate code provides opportunities for optimisation: just before label L_3 is a "Jump" instruction that leads to L_4, and L_4 labels another jump instruction, this time to L_1. It would be better to have the first jump go to L_1 directly, thereby saving time. Optimisations like this 'jump-to-jump' optimisation can be implemented by a compiler pass that takes an intermediate code program as its input and produces an equivalent, optimized program, also in intermediate code, as its output.

1.8 Object code

The final representation of the program that is used by a compiler is the object code for a particular target machine. Production compilers often output this code in binary form, ready for immediate execution, but we will output the code in assembly language, using the native assembler of the target machine to produce the final binary program. Thus our compilers, even when they output machine code for a real machine, are just programs that input one text (the source code of a program) and output another text (assembly language for the same program). There is no 'magic' in a compiler; even when the output is in binary form, it is just a data file in a particular format.

To complete our example, Figure 1.4 shows object code for the program in approximately the assembly language of the SPARC.[2] The stack of temporary values has been replaced by fixed registers with names like %l0 and %l1.

The statement x := x - y has been translated into the sequence

```
ld [%fp+100],%l0
ld [%fp+104],%l1
sub %l0,%l1,%l0
st %l0,[%fp+100]
```

The first two instructions load the contents of x and y into the two registers %l0 and %l1, using the addresses that were calculated earlier. Next, the sub instruction subtracts one from the other, putting the result back in the %l0 register. Finally, the difference is stored back into x.

Again, there is plenty of scope for optimisation of the object code: for example, at label L_3 we know that the %l0 register contains the value of x, so it is not necessary to use another ld instruction to load x into the register again. As we shall see, many such optimisations can be carried out as part of the transformation from intermediate code to machine code.

1.9 A road map

Figure 1.5 shows a 'road map' for the compilers we will study in the first half of this book. The translation and execution of each program will involve four distinct phases, one after the other:

(1) A lexical analyser and a syntax analyser (modules *Lexer* and *Parser*)

[2] I've taken some liberties here in the interests of exposition. In particular, the code shown doesn't contain the nop instructions that ought to fill the *branch delay slots* that follow each branch instruction.

```
L1:
      ld [%fp+100],%l0
      ld [%fp+104],%l1
      cmp %l0,%l1
      be L2
      ld [%fp+100],%l0
      ld [%fp+104],%l1
      cmp %l0,%l1
      bge L3
      ld [%fp+104],%l0
      ld [%fp+100],%l1
      sub %l0,%l1,%l0
      st %l0,[%fp+104]
      b L4
L3:
      ld [%fp+100],%l0
      ld [%fp+104],%l1
      sub %l0,%l1,%l0
      st %l0,[%fp+100]
L4:
      b L1
L2:
```

Figure 1.4: *Object code*

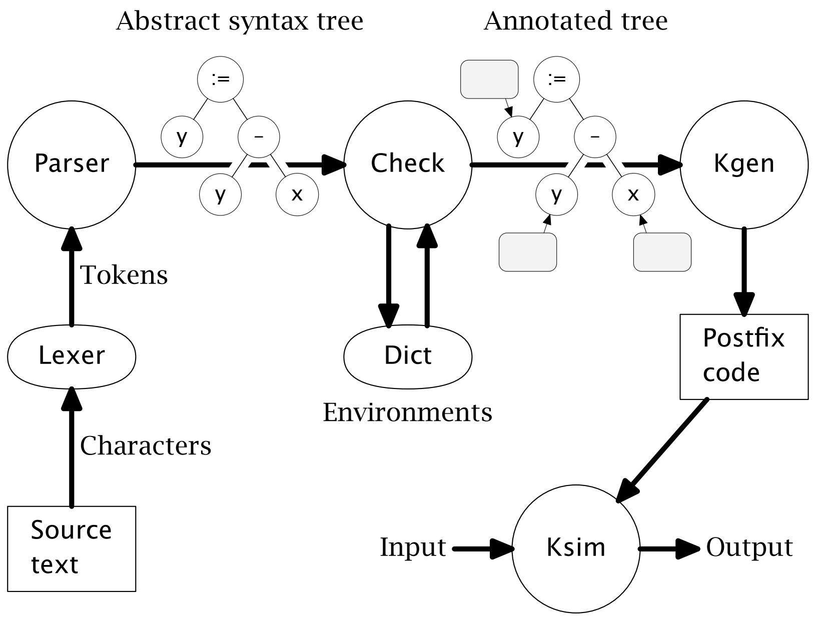

Figure 1.5: *A road map*

take the file of characters in which the program is stored, and produce first a stream of tokens, then an abstract syntax tree that represents the structure of the program. In our compilers, the lexical analyser will operate as a subroutine of the syntax analyser, reading the characters of one token each time it is called and returning the completed token to the syntax analyser. If the program contains any syntax errors, a message will be printed and the compiler does not continue with later phases; otherwise an abstract syntax tree is created and used as the input to later phases in the compiling process.

The abstract syntax tree is designed to contain all the information about the program that is needed for the rest of the compiling process, but to suppress detail that is irrelevant. Comments and information about the layout of the program is thrown away by the lexical analyser, and some furhter detail is thrown away as the tree is built: for example, parentheses may be used in the input program to show the order of evaluation of operators in expressions. Once each expression has been represented as a tree, this information is captured in the structure of the tree, and there is no further need to record the fact that parentheses were present in the source program.

(2) A semantic checker (module *Check*) takes the abstract syntax tree and checks that variables have been properly declared, that the arguments of each operator in the program have the right types, and so on. If there are errors of this kind, then error messages are printed and the compiler stops here. Otherwise, the semantic checker annotates the tree by attaching a label to each place where an identifier is used; the label contains information from the declaration of the identifier, such as its type and possibly the location it will occupy when the program runs.

This phase uses a helper module *Dict* (for *dictionary*) to build tables called *environments* that show the definition of each identifier that may be used in a certain region of the program. If the compiler implements a language like Pascal or C where subroutines can their own local variables, then a different environment will be created for the body of each subroutine.

The annotations that are attached to places in the tree where identifiers are used should contain all the information needed to produce object code. It is the responsibility of the semantic checker to move information around the tree, and the next phases should be able to operate in a purely local way, doing their work using only the information that is present in the small fragment of tree they are translating at a particular time.

(3) An intermediate code generator (module *Kgen*) produces postfix code for the program by traversing the annotated tree. The code that is generated contains both a sequence of instructions for the main program and labelled sequences of instructions for each subroutine.

(4) The postfix code is executed by an interpreter (module *Ksim*) that carries out the actions of each instruction of the abstract machine. The interpreter begins with the instructions for the main program, and uses the instructions that have been compiled for each subroutine when it

reaches a call to the subroutine. If the program contains input and output instructions, it is at this point that the input is read and the output is written.

In a practical compiler, the postfix code would be stored in a file that would be input by the interpreter at the start of execution, so that a program could be compiled once and executed many times, using the same file of postfix code each time. To avoid the complication of defining a file format for the code and writing routines to read and write it, we shall bundle the interpreter together with the compiler in a single program, and compile the source program afresh before each execution. To help with understanding and debugging our compilers, we will provide facilities to print out the code before it is executed and to print out each instruction each time the interpreter executes it.

Not all of the compilers we shall build include every phase: in particular, we shall begin our study of intermediate code generation for expressions and statements in Chapter 5 with a very simple language where all variables hold integer values, and variables do not have to be declared. We shall use a simplified abstract machine where storage locations are identified by names rather than numeric addresses, and that means we can omit the stage of semantic checking. That phase will enter the picture soon afterwards in Chapter 6, where we introduce different data types and change to a more realistic abstract machine with proper addressing.

Chapter 2

Just enough ML

The programs in this book are written in a dialect of the functional programming language ML. There are several dialects of ML in current use, and though they differ in details, all dialects share some common characterisitics. They all support functional programming with polymorphic typing and applicative-order evaluation. Thay all have a module system that permits programs to be built from a collection of largely independent components, with the implementation details of each module hidden from the other modules that use it. All dialects add to the purely functional core of the language certain concepts from imperative programming, including assignable variables, control structures and an exception mechanism. These features make it easier to use the ideas of functional programming whilst avoiding the cluttered style it sometimes leads to. For example, if a value is needed in a large part of a program but seldom changes, the imperative features of ML allow us to assign the value to a global variable, rather than passing it as an argument in every function call.

The dialect we shall use is called "Objective CAML"; there is a freely available implementation that runs on many different machines, including Sun workstations and the IBM PC under both Windows and Linux. In this chapter, I shall introduce just enough of the language to allow understanding of most of the programs in the book. The name "Objective CAML" refers to the fact that the language supports a form of object-oriented programming. Although objects can be used fruitfully in writing compilers, we shall not need to use the object-oriented features of the language. Objective CAML also has a sophisticated module system that allows nested modules with parameters, and we shall need to use only the simplest subset, where the text of each module is kept in a separate file and there is no nesting. Objective CAML has many other unusual features, including functions with optional and keyword parameters; I have avoided these features, in the hope that it will make the programs in this book more approachable, and easier to translate into other languages. What you will find in this chapter is a concise summary of just that part of the language that we shall use in the rest of this book. For conciseness, I shall in future refer to the language Objective CAML just as ML, except where I am drawing attention to specific features of the language or its implementation that are not shared by other dialects of ML.

This chapter is intended for readers who already have some familiarity with functional programming, perhaps in some other language such as

HASKELL, or another dialect of ML, and need a brief introduction to the syntax of Objective CAML and those of its features that are not part of the purely functional subset. The main way of programming in ML, as in any language that supports a functional style of programming, is to define functions recursively. This style works well in compilers, because the abstract syntax of a programming language is very naturally modelled by a family of recursive data types, and many tasks that the compiler must carry out, such as checking that a program obeys the rules of the programming language or translating it into machine code, can naturally be expressed as recursive functions over these data type; in this way, the syntax rules of the programming language guide the construction of the compiler. The chapter begins with some very brief examples of familiar functions defined in Objective CAML.

As well as supporting functional programming, Objective CAML has a number of other features that we shall use: in particular, there is a module system that allows programs to be split cleanly into independent pieces, with the source code of each module kept in a separate file and compiled independently of other modules.

The chapter ends with a brief explanation of how the printed form of ML programs that appears in this book is related to the plain ASCII form that is accepted by the ML compiler.

2.1 Defining functions

Functions can be defined by pattern matching, as in the following definition of the function *reverse* on lists:

```
let rec reverse =
  function
      [ ] → [ ]
    | x :: xs → reverse xs @ [x];;
```

The keywords **let rec** introduce a recursive function definition; *reverse* is defined by two patterns, one matching the empty list [ ], and the other matching a non-empty list *x* :: *xs* that has head *x* and tail *xs*. The value of *reverse* [ ] is [ ], and the value of *reverse* (*x* :: *xs*) is formed by recursively reversing the list *xs*, forming the singleton list *x*, and joining the two results with the operator @, denoting concatenation of lists. Like every top-level phrase in an ML program, the definition ends with a double semicolon ;;. The ML programs in this book have been formatted for ease of reading, using different styles of type and special symbols like →; when the programs are entered into a computer, an ASCII form of the language is used, and the arrow symbol is typed as ->, for example. Section 2.11 on page 34 explains the correspondence between the two forms.

ML has a strong, polymorphic type system which does not require types to be specified by the programmer: from the definition of *reverse* given here, the ML compiler can deduce that *reverse* has the type

$$\alpha\ list \rightarrow \alpha\ list$$

Here, α is a type variable that can stand for any type, and the type constructor *list* is written (as always in ML) after its argument. So this type expresses the fact that *reverse* can accept any list as argument, and delivers as its result another list of the same type.

ML has a number of basic types:

- *int*, integers of 31 bits.
- *char*, characters, written between single quotes like this: 'a', 'b', 'c'.
- *string*, strings of characters, written in double quotes like this:

  "This is a string".

  Note that in ML, strings are not the same as lists of characters: they have a more compact representation.

2.2 Type definitions

ML allows the programmer to define new data types, including recursive types such as trees. We shall use tree-like types a lot, because they provide a way of modelling the syntactic structure of the programs in a compiler: so we choose as an example a type that could be used to model arithmetic expressions.

```
type expr =
    Number of int
  | Variable of string
  | Binop of op * expr * expr
and op = Plus | Minus | Times | Divide;;
```

This definition introduces two types *expr* and *op*. The two definitions are joined by the keyword **and**, so that each definition may refer to the other one. The type *op* simply consists of the four values *Plus*, *Minus*, *Times*, and *Divide*, but the type *expr* has a more elaborate recursive structure. An *expr* may be simply a constant *Number n* for some integer *n*, or a variable *Variable x* named by the string *x*, but it can also be a compound expression *Binop* (*w*, *a*, *b*), where *w* is an operator and *a* and *b* are other expressions. For example, the expression

```
x * (y + 1)
```

would be represented as the following value of type *expr*:

```
Binop (Times, Variable "x",
        Binop (Plus, Variable "y", Number 1)).
```

Constructors like *Binop* and *Plus* must always begin with an upper-case letter, as must the names of modules (see later), but other names used in an Objective CAML program must begin with a lower-case letter.

We can use recursion to define functions that work on recursive types. Here is a function *eval* that gives the value of an arithmetic expression (at least, if it doesn't contain any variables).

```
let do_binop w a b =
  match w with
      Plus → a + b
    | Minus → a − b
```

```
      | Times → a * b
      | Divide → a/b;;

let rec eval =
  function
      Number n → n
    | Binop (w, e₁, e₂) →
        do_binop w (eval e₁) (eval e₂)
    | Variable x →
        failwith "sorry, I don't do variables";;
```

These definitions illustrate several new language features. The function *eval* is defined by pattern-matching on its argument, an expression tree. An expression that contains a binary operator is evaluated by recursively evaluating its two operands, then applying the operator to the results. The curried function *do_binop* takes three arguments: the operator, and the values of its two operands. It is defined by matching the operator against a sequence of patterns in a **match** expression, each pattern matching a particular arithmetic operation. These **match** expressions are rather like the **case** statements of other programming languages.

In more complex examples, we might need to define several recursive functions, each able to call the others. Normally, each function must be defined before it is used, so these *mutually recursive* functions present a problem. The solution provided by Objective Caml is to join the function definitions with the keyword **and**, so that they are treated as a unit by the compiler. To illustrate the syntax, here are two functions f and g that call each other:

```
let rec f n =
  if n = 0 then 1 else g (n−1)
and g x =
  if n = 0 then 0 else g (n−1) + f (n−1);;
```

The function f is defined a little like the Fibonacci function, except that it satisifes the recurrence

$$f\,(n) = f\,0 + f\,1 + \ldots + f\,(n-1) \qquad (n > 0),$$

and the sum on the right-hand side is computed by $g\,(n-1)$. In fact, $f\,n = 2^n$ for all n, as can be proved by induction. Because the programs in this book are often interspersed with the text, I haven't always used **and** when it is needed, or followed the rule that functions must be defined before they can be used.

We've seen two sorts of pattern matching: one introduced by the keyword **function**, and the other introduced by the keyword **match**. In fact, these are equivalent, in the sense that a definition

```
let rec f = function ...
```

can always be replaced by the equivalent form

```
let rec f x = match x with ...
```

We'll continue to use both forms, choosing whichever is the more convenient in each instance.

The treatment of expressions of the form *Variable x* illustrates another language feature: exceptions. Evaluating the expression *failwith s*, where *s* is a string, does not result in any value; instead, the outcome is an exception, which can be caught either by a surrounding program or by the ML system itself. In the latter case, execution of the program is abandoned, and ML prints an error report. We shall use *failwith* extensively to replace parts of our compilers that we have not yet implemented properly.

2.3 Tuples and records

ML provides a family of types for n-tuples: for example, (1, 2, "3") is an expression with type $int * int * string$. We have already seen such types used for the arguments of constructors in user-defined types. Standard functions *fst* and *snd* are provided for extracting the first component x and the second component y of an ordered pair (x, y): matching.

$$fst : \alpha * \beta \to \alpha$$
$$snd : \alpha * \beta \to \beta$$

These functions can be defined by pattern matching like this:

let *fst* $(x, y) = x$;;
let *snd* $(x, y) = y$;;

Components of bigger n-tuples can also be extracted by pattern matching; for example, here is a function the extracts the second component of a 3-tuple:

let $second_3\ (x, y, z) = y$;;

As an alternative to n-tuples, ML also provides record types. A type definition like

type *def* = { *d_key* : *string*; *d_value* : *int* };;

defines a new record type *def* with selectors *d_key* and *d_value*. Values of this record type can be constructed by expressions like

{ *d_key* = "foo"; *d_value* = 3 }

and if d is such a value, its components can be extracted as *d.d_key* and *d.d_value*. For our purposes, these record types provide nothing that cannot be achieved with tuples, but they sometimes serve to make our programs a little clearer.

In addition to type *definitions* that create new record types or recursive algebraic types, ML also allows type *abbreviations*. For example, the declaration

type *couple* = $string * int$;;

introduces *couple* as an alternative name for the type $string * int$.

2.4 Modules

Objective CAML has a sophisticated module system that allows nested modules and modules that take other modules as parameters. However, we shall not need this sophistication in the compilers and interpreters we build, and can get by instead with a simple subset of Objective CAML's features that supports separate compilation of modules, much as in Modula-2. Each module *M* consists of an interface file *M*.mli that advertises certain types and functions, and an implementation file *M*.ml that contains the implementations of those types and functions. The information in the interface file is shared between the implementation of the module and its users, but the implementation is effectively hidden from the users, thereby enforcing separation of concerns, and allowing the implementation to change without requiring re-compilation of all the user modules.

The interface file contains a declaration giving the type of each function that is provided by the module. For example, a module that contained the function *reverse* from the preceding section might contain this declaration in its interface:

```
val reverse : α list → α list;;
```

This declaration is a promise that the implementation of the module will define a function *reverse* with the type shown.

The interface file can also contain the definitions of types that are shared between the implementation and its users, and declarations for types that are implemented by the module but have hidden representations. As an example, a module that implements symbol tables might provide a type *dict*, together with operations for inserting symbols into a table and for looking up a given symbol. This module might have the following declarations in its interface file dict.mli:

```
(* dict.mli *)

type dict;;

val empty : dict;;

val insert : string → int → dict → dict;;

val lookup : string → dict → int;;
```

These declarations promise that a type *dict* will be implemented, and advertise a constant *empty* and two operations *insert* and *lookup* that will be provided, without revealing the representation that will be chosen in the implementation.

A simple implementation might represent *dict* values by lists of records with the type *def* we defined earlier. In that case, the implementation file dict.ml would contain the following definitions:

```
(* dict.ml *)

type def = { d_key : string; d_value : int };;

type dict = def list;;

let empty = [ ];;

let insert x v t = { d_key = x; d_value = v } :: t;;

let rec lookup x =
  function
      [ ] → raise Not_found
    | d :: t →
        if d.d_key = x then d.d_value else lookup x t;;
```

This gives the actual type that is chosen to represent symbol tables, and versions of *insert* and *lookup* that work for this choice of type. The choice is hidden from other modules that use the symbol table, making it possible to replace the data structure by a more efficient one without needing to change or even recompile the other modules. Like the *failwith* function we saw earlier, the expression *raise Not_found* raises an exception,

There are two ways in which expressions in one module can name the exported features of another module. One is to use a *qualified name* such as *Dict.insert* that contains the name of the module and the name of the exported identifier. The module name appears with an initial capital, and it is this that removes the ambiguity between qualified names $M.x$ and the notation $r.x$ for selecting field x from a record r. The other way is for the client module to contain an "**open**" directive such as

> **open** *Dict*;;

After such a directive, all the exported identifiers of *Dict* are available directly, without further need for qualification.

If types are defined in the implementation file, they may be made visible to client modules in several ways, with different effects:

- The type may not be mentioned in the interface file at all. In this case, the type is entirely private to the implementation. Functions that are exported in the interface file may not accept arguments or produce results of the type.
- The type may be declared in the interface without any details of its structure, like this:

  > **type** *dict*;;
  > **type** (α, β) *mapping*;;

  (As the second example shows, such types can have parameters too.) In this case, functions that are exported by the module may accept parameters and deliver results of that type, but clients may only create and manupulate values of the type by using the exported functions. This style of export is appropriate for modules that implement an abstract data type, because it allows the hidden definition of the type to be changed without the need to change or even recompile client modules.
- The definition from the implementation file may be repeated in full in the interface file. In this case, client modules have full freedom to create values of the type, and to manipulate them by pattern matching. This style is appropriate for modules that define a common data type used throughout a program, such as the abstract syntax trees in our compilers.

2.5 Exceptions

ML has two groups of features that take it outside the realm of pure functional programming: one is the exception mechanism, and the other is the facility for assignable variables. We have already met the expression

> *raise Not_found*

that raises the exception called *Not_found*, and the function *failwith*, which also raises an exception. We shall use *failwith* only to indicate that part of the compiler is missing or broken, but in general, exceptions provide a useful way of dealing with computations that fail in ways that can be predicted.

For example, the *lookup* function of the preceding section raises the exception *Not_found* if the identifier we are looking for is not present in the symbol table we are searching. Evaluating the expression *raise Not_found* does not result in a value in the ordinary way, but instead propagates the exceptional value *Not_found*. The beauty of exceptions (and also their weakness) is that these exception values implicitly propagate through the program until they reach a context where an exception handler has been created, or until they reach the outside world and are dealt with by terminating the whole program.

Complementary to *raise* is the ML facility for handling exceptions raised during the evaluation of an expression: for example, one might use *lookup* in the following expression, which returns the value associated with "Mike" in the table *age*, or 21 if there is no such value:

try *lookup* "Mike" *age* **with** *Not_found* → 21;;

Though we shall make little use of them, ML also provides exceptions with arguments; thus *failwith s* is equivalent to *raise* (*Failure s*), where *Failure* is a pre-defined exception constructor that takes a string argument. When this exception is raised and not caught inside a program, the run-time system of ML is able to catch the exception, extract its string argument, and print it in an error message. This makes *failwith* a convenient way to temporarily plug gaps in a program where it is unfinished; if the gap is ever reached, we can arrange that the ML system will give a recognisable message before ending execution. Although it is possible to catch the *Failure* exception with a **try** block, we shall never do so.

2.6 References

In addition to the purely functional data types we've seen so far, ML provides *reference types*, whose values can be changed. These values or *cells* can be used to simulate the variables of a language like PASCAL, allowing an imperative style of programming where that is more convenient than a purely functional style.

If x is any value, then evaluating the expression *ref* x creates a new cell r (different from all the others in use) and fills it with the initial value x. At any time, we can evaluate the expression $!r$ to retrieve the current value stored in cell r, and we can evaluate the expression $r := y$ to update the contents of cell r so that it contains the value y.

We shall use reference types for many purposes in our compilers:

- They allow us to write algorithms in an imperative style when that is convenient. For example, we shall often write programs that simulate the action of an abstract machine, and it is more natural to write these programs imperatively, so that they destructively update the state of the machine in the same way that hardware does.
- They allow us to build modules that have an internal state, with operations for changing and inspecting that state. This can make our

compilers simpler than they would be if all the data had to be passed around in a functional style. For example, we shall build a module whose internal state is the sequence of machine instructions that have been generated so far by the compiler, and provide an operation that adds another instruction to the sequence. This is both more efficient and more manageable than a scheme that passes around explicit lists of instructions.

- They allow us to build data structures with modifiable components. This is how we shall allow the semantic analysis phase to annotate the tree with information that is needed by the code generator.
- References are ML's only way of simulating the cyclic structures that we can build in a lazy language like Haskell. When we study languages with recursive procedures, these cyclic structures are a natural way of modelling the fact that any procedure can count itself among the set of procedures it may call. We'll study all that in detail later: for now, we just need to understand how to use ML to build data structures with cycles.

As an example of a simple use of references, here is a simple division algorithm written in an imperative style. The **if** ... **then** .. **else** expressions we have already met can be used as a conditional statement for imperative programming, but ML also provides sequencing (;), grouping with **begin** and **end**, and **while** loops:

```
let divide a b =
  let q, r = ref 0, ref a in
  begin
    while !r ≥ b do
      r := !r − b; q := !q + 1
    done;
    !q
  end;;
```

The keywords **begin** and **end** are simply a more familiar alternative to grouping with parentheses (...). Also illustrated here is the notation **let** *lhs* = *rhs* **in** *exp* for introducing a local definition; in this case, it used to declare two local reference cells, but it can also be used to define local variables or functions.

The result of a function written in imperative style is the last expression in its body, here the final value !*q* of the modifiable variable *q*. As you can see, the need to write the dereferencing operator ! explicitly at every variable reference is a powerful force in favour of a more functional style, when that is convenient:

```
let divide a b =
  let rec div₁ q r =
    if r < b then q else div₁ (q + 1) (r − b) in
  div₁ 0 a;;
```

Again, we have used **let**, this time to introduce a local function *divide*$_1$ that is *tail-recursive*, meaning that the only recursive call is the very last action in the function's body. A decent implementation of ML will execute such a tail-recursive function with the same efficiency as the imperative version.

Nevertheless, the imperative style will still be useful in some contexts, where a purely functional program would be more complicated.

As an example of a module with internal state, here is the definition of a module that maintains a single table of names and values:

```
(* onedict.mli *)

val add : string → int → unit;;

val find : string → int;;
```

The idea is that *add* inserts a new string into the table, returning the unit value ( ), that is to say, no value at all. The *find* function looks up a name and returns the corresponding value, or raises the exception *Not_found* if no such value exists.

We could implement this module in terms of the *dict* module we defined before, using a global reference cell to keep hold of the current table:

```
(* onedict.ml *)

let table = ref Dict.empty;;

let add x v = (table := Dict.insert x v !table);;

let find x = Dict.lookup x !table;;
```

To give an example of how references can be built in to a tree structure, here is a version of the *expr* type we defined earlier, modified so that each expression can be annotated with its value:

```
type expr =
  { e_guts : expr_guts; e_value : int ref }
and expr_guts =
    Binop of op * expr * expr
  | Number of int;;
```

If we have analysed an expression *e* by looking at its field *e.e_guts* and found it to have value *v*, then this value can be stored for future reference by the assignment *e.e_value* := *v*. Here, then, is a function that evaluates an expression, *both* returning the value as its result *and* storing the value of each sub-expression as an annotation:

```
let rec eval e =
  let v = match e.e_guts with
        Binop (w, e1, e2) → do_binop w (eval e1) (eval e2)
      | Number n → n in
  e.e_value := v; v;;
```

Finally, I should give an example of the use of references to build cyclic structures. We define the type *chain* as follows:

```
type chain = Empty | Cell of string * chain ref;;
```

This type looks rather like the type that would be used to build linked lists in PASCAL or C, with *ref* where we expect to see a pointer type used. Note however that there is an explicit alternative for the empty list: unlike PASCAL's pointers, ours don't have a dummy *nil* value. We can build a cyclic list structure with two elements "flip" and "flop" like this:

```
let link = ref Empty;;
```

let *cycle* = *Cell* ("flip", *ref* (*Cell* ("flop", *link*)));;

link := *cycle*;;

This works by first building an acyclic structure, then making it cyclic by 'tying the knot'. The object *link* is the reference cell that forms the tail of the "flop" node. It initially contains *Empty*, but the final assignment *link* := *cycle* changes it to contain ('point to') the "flip" node.

In addition to simple references, ML also provides *arrays*, created by the expression *Array.create n x*: this makes an array of n elements numbered from 0 up to $n-1$, all of them initially equal to x. If a is such an array, its i'th element can be accessed by the expression $a.(i)$, and it can be set to y by evaluating the expression $a.(i) \leftarrow y$.

2.7 Library functions

Objective CAML comes with a large library of standard modules that are quite well suited to writing compilers, perhaps because the library was partly developed to support the implementation of the CAML system itself. In this section, I'll give a brief overview of the functions from the standard library that we'll use in this book.

Two important abstract data types used in most compilers are *mappings* and *associative tables*, both representing functions from arbitrary keys to arbitrary values; often the keys will be identifiers in the program being compiled, and the values will be an attribute computed by the compiler, such as the types of the identifiers or their run-time addresses. The type of mappings provides a purely applicative interface, whilst the type of associative tables provides an imperative interface, where the mapping is built up by destructive modification of a single table. This makes greater efficiency possible, because the imperative interface can be implemented as a hash table with essentially constant-time access, whereas the applicative interface is implemented by a search tree that gives access in $O(\log N)$ time, where N is the number of keys in the mapping. Because these two data types are so important in building compilers, I have chosen not to use the perfectly adequate implementations provided in the standard library, but to build and describe implementations of my own. The interfaces of the two data types appear in the two sections immediately following this one: first mappings represented by B-trees, then associative tables represented by hash tables. One of the decisive strengths of ML is that it is possible to make a single implementation of these data types that can be used for many different types of keys and values; the implementations of the two types are given in Appendix A.

The last module in our library defines a family of functions – analogous to *printf* in C – that provide formatted output. Again, there is an implementation in the standard library of Objective CAML, but the version I shall present is extensible to handle the printing of new types of data.

2.7.1 Lists

The module *List* provides many of the standard operations on lists. Here are the basic ones:

val *length* : α *list* → *int*;;

```
val (@) : α list → α list → α list;;
val hd : α list → α;;
val tl : α list → α list;;
val nth : α list → int → α;;
val rev : α list → α list;;
```

If *xs* is the list $[x_0; x_1; \ldots; x_{n-1}]$ and *ys* is the list $[y_0; y_1; \ldots; y_{m-1}]$, then these functions deliver results as follows:

length xs = n

xs @ *ys* = [x_0; . . .; *xnmi*; y_0; . . .; *ymmi*]

hd xs = x_0

tl xs = [x_1; . . .; *xnmi*]

nth xs i = x_i

rev xs = $[x_{n-1}; \ldots; x_1; x_0]$

The function *concat* takes a list of lists and flattens it into a single list:

```
val concat : (α list) list → α list
```

If *xss* is the list of lists $[xs_0; xs_1; \ldots; xs_{n-1}]$, then

concat xss = xs_0 @ xs_1 @ · · · @ xs_{n-1}.

Much of the power of functional programming comes from the standard higher-order functions that can be defined, reducing the need for recursive definitions in programs that use them. We shall use the following:

```
val map : (α → β) → α list → β list;;
val iter : (α → unit) → α list → unit;;
val fold_left : (α → β → α) → α → β list → α;;
val fold_right : (α → β → β) → α list → β → β;;
```

Taking *xs* to be the list $[x_0; x_1; \ldots; x_{n-1}]$ as before, we find that

map f xs = $[f\ x_0; f\ x_1; \ldots; f x_{n-1}]$.

Because ML is not a purely functional language, it sometimes matters in what order the terms $f\ x_0, f\ x_1, \ldots, f x_{n-1}$ are evaluated: for example, the function f might have a side-effect that allows one evaluation of f to affect the action of future evaluations. The function *map* is defined so that these terms are evaluated in left-to-right order, so that any side-effects of evaluating $f\ x_0$ happen before those of $f\ x_1$, and so on. Here is a definition of *map* that has this property:

```
let rec map f =
  function
      [ ] → [ ]
    | x :: xs → let y = f x in y :: map f xs;;
```

It is the **let** expression that ensures that the value of $f\ x$ is determined before the recursive call *map f xs* is evaluated. The left-to-right evaluation is useful for tasks like defining the function *totals* that returns a list of running totals, like this:

totals [1; 2; 3; 4; 5] = [1; 3; 6; 10; 15]

We can use a reference cell to keep the running total, and map a fuction over the list that both increases the total and returns its current value:

```
let totals xs =
  let t = ref 0 in
  let f x = (t := !t + x; !t) in
  map f xs;;
```

In this simple example, it might be easier and clearer to define *totals* directly by recusion:

```
let totals xs =
  let rec tot₁ t =
    function
        [ ] → [ ]
      | y :: ys → let t′ = t + y in t′ :: tot₁ t′ys in
  tot₁ 0 xs;;
```

In more complicated situations, however, it is good to have the additional flexibility that a left-to-right *map* provides.

The function *iter* is similar to *map*, in that evaluating *iter f xs* entails evaluating $f\ x_0$, $f\ x_1$, ..., $f x_{n-1}$ in left-to-right order, but these evaluations are done purely for the sake of their side-effects, because the results are discarded, and the value returned by *iter* is the *unit* value ( ).

The two functions *fold_left* and *fold_right* combine all the elements of a list using a binary operation:

$$fold\_left\ f\ a\ xs = f\ (\ldots (f\ (f\ a\ x_0)\ x_1) \ldots)\ x_{n-1}$$

$$fold\_right\ f\ xs\ a = f\ x_0\ (f\ x_1\ (\ldots (f\ x_{n-1}\ a) \ldots))$$

Thus *fold_left* combines the elements of the list starting from the left, and *fold_right* starts from the right.[1] Here are the recursive definitions:

```
let rec fold_left f a ys =
  match ys with
      [ ] → a
    | x :: xs → fold_left f (f a x) xs;;

let rec fold_right f ys a =
  match ys with
      [ ] → a
    | x :: xs → f x (fold_right f xs a);;
```

If the function f is associative, so that $f\ (f\ x\ y)\ z = f\ x\ (f\ y\ z)$ for all x, y, z, and a is a left and right identity element for f, so that $f\ a\ x = x = f\ x\ a$, then there is no difference between the results returned by *fold_left* and *fold_right*. This is true, for example, if $f\ x\ y = x + y$ and $a = 0$. In this case, both *fold_left f a xs* and *fold_right f xs a* compute the sum of the list of integers *xs*. Using the ML notation (+) for the addition operator on integers, considered as a function of type *int* → *int* → *int*, we can therefore define a function *sum* : *int list* → *int* by

```
let sum xs = fold_left (+) 0 xs;;
```

[1] Readers familiar with other functional programming languages such as HASKELL should note the *fold_right* takes its arguments in a different order from the function *foldr* defined in other languages, so that *fold_right f xs a* = *foldr f a xs*.

Where the two give the same result, we prefer *fold_left* over *fold_right* because its tail-recursive definition leads to a program whose space usage is constant, rather than linear in the length of the argument list.

A number of deal with lists of ordered pairs of type $(\alpha * \beta)$ *list*:

```
val combine : α list → β list → (α * β) list;;
val assoc : α → (α * β) list → β;;
val remove_assoc : α → (α * β) list → (α * β) list;;
```

The function *combine* takes two lists (which must be of the same length), and pairs up their elements, returning a list of pairs: *combine xs ys* = $[(x_0, y_0); (x_1, y_1); \ldots; (x_{n-1}, y_{n-1})]$. Thus *combine xs ys* is a list of pairs such that *map fst* (*combine xs ys*) = *xs* and *map snd* (*combine xs ys*) = *ys*.

The function *assoc* is useful when a mapping is represented by a list of (argument, value) pairs. The arguments of *assoc* are an argument for the mapping, together with the list of pairs that represents the mapping itself, and the result is the corresponding value of the mapping; *assoc* raises the exception *Not_found* if no such value exists. If the list *ps* contains more than one pair (u, v) with $u = x$, then *assoc x ps* returns the value of v from the first such pair. Thus *assoc* may be defined as follows:

```
let rec assoc x =
  function
      [ ] → raise Not_found
    | (u, v) :: ps → if u = x then v else assoc x ps;;
```

There is another function *remove_assoc* that removes the first pair (u, v), if any, with a specified component $u = x$:

```
let rec remove_assoc x =
  function
      [ ] → [ ]
    | (u, v) :: ps →
        if u = x then ps else (u, v) :: (remove_assoc x ps);;
```

If the list is taken to represent a mapping, then this function removes x from its domain.

All the functions listed above are part of the module *List* in the standard library; they are used so often that we will usually open this module so that they can be used without qualification.

2.7.2 Optional values

Lists may contain any number of elements, from zero upwards. Sometimes it's convenient to use a more restrictive type that allows either zero or one element of another type; this is the purpose of the type constructor α *option*, defined as follows:

```
type α option = Some of α | None;;
```

For example, a language might have a `return` statement that can be followed either by an expression giving the value returned by a subroutine, or by nothing if the subroutine returns no result. In this case, we could represent `return` statements by in the abstract syntax tree like this:

```
type stmt = . . .
```

| *Return* **of** *expr option*
| ...

Then we could use *return* (*Some e*) for the `return` statement that contains expression *e*, and *return None* for the return statement containing no expression.

2.7.3 Arrays and strings

In addition to lists, which can be of unpredictable length, are purely functional in their behaviour, but require linear time to access an arbitrary element, ML also provides the alternative data structures of *arrays* and *strings*. An array has a fixed length and allows constant-time access to its elements, which are identified by numeric indices. There is an operation to retrieve an element given its index and an operation to destructively update the array so that a given index is associated with a new value. Arrays thus share with reference cells a non-functional character that depends on side-effects. Strings in ML are similar to arrays, but their elements are always characters; strings are stored in a particularly efficient way so that the memory space occupied by a string is one byte per character, plus a small constant overhead.

An array with elements of type α has the built-in type α *array*. The module *Array* provides the following operations on arrays:

val *create* : *int* $\to$ α $\to$ α *array*;;
val *length* : α $\to$ *int*;;

An array of length n is created by *Array.create n x*; all the elements of this array are initialized to x. The i'th element of array a is written $a.(i)$ for $i = 0, 1, \ldots, n-1$, and the i'th element is set to y by the operation $a.(i) \leftarrow y$.[2] The length of a is given by *Array.length a*.

Strings belong to the built-in type *string*. There is a library module *String* that provides the following interface:

val *create* : *int* $\to$ *string*;;
val *length* : *string* $\to$ *int*;;
val *sub* : *string* $\to$ *int* $\to$ *int* $\to$ *string*;;

A string of length n is created by *String.create n*; all characters of this string are initially undefined.[3] Strings are also created as the value of string constants appearing in ML programs. The i'th character of a string s is written $s.[i]$ for $i = 0, 1, \ldots, n-1$, and it can be set to a character c by evaluating $s.[i] \leftarrow c$.[4] The length of s is given by *String.length s*, and two strings s_1 and s_2 of lengths n_1 and n_2 can be concatenated to form a single new string of length $n_1 + n_2$ by writing $s_1 \frown s_2$. The sub-string of a string s starting at character $s.[i]$ and continuing to character $s.[i + j-1]$ (and therefore of length j) is written *String.sub s i j*.

[2] The notations $a.(i)$ and $a.(i) \leftarrow y$ are abbreviations for *Array.get a i* and *Array.set a i y*, and use operations provided by the *Array* module.
[3] This is one of very few places in Objective CAML where an undefined value is created
[4] The notations $s.[i]$ and $s.[i] \leftarrow c$ are abbreviations for *String.get s i* and *String.set s i c*.

2.8 *B*-trees

In compilers, it is frequently necessary to represent a mapping from one data type to another. For example, we shall want to make *symbol tables* that show, for each identifier that has been declared in a program, what its type is and where it is located in run-time storage; these symbol tables are mappings from identifiers to records that contain this information.

As we have seen, one representation of a mapping as an *association list*, such as is used with the standard function *assoc*. An association list represents a mapping from a type α to a type β is simply a list of pairs with the type $(\alpha * \beta)$ *list*. Although this way of representing mappings works perfectly well, it has two defects that it would be nice to avoid.

- The first defect is in the level of abstraction: it would be nice to distinguish those contexts in which a list of pairs is used for its own sake from those contexts in which it is used as a way of representing a mapping.
- The second defect is one of efficiency. In order to find the value of the mapping for a particular argument, it will be necessary to examine half the pairs in the list on average, and all of them in the worst case. The time to find a value of the mapping is thus linear in the size of the mapping. Finding find the value of a mapping of size N at N values of its argument would therefore take time $O(N^2)$, and it would be nice to do better than this.

We can answer the first criticism by introducing a module that encapsulates the data type of mappings, hiding its representation. Here is the interface of that module:

```
(* mapping – type of mappings from α to β *)
type (α, β) mapping;;

(* empty – empty mapping *)
val empty : (α, β) mapping;;

(* add – insert new pair into mapping *)
val add : α → β → (α, β) mapping → (α, β) mapping;;

(* find – find value of mapping, or raise Not_found *)
val find : α → (α, β) mapping → β;;

(* remove – delete a pair *)
val remove : α → (α, β) mapping → (α, β) mapping;;

(* iter – visit all pairs *)
val iter : (α → β → unit) → (α, β) mapping → unit;;
```

The type (α, β) *mapping* is the type of mappings from α to β. The simplest such mapping is the empty mapping *empty*, which is defined for no values of its argument. Larger mappings can be created using the function *add*, defined so that $m' = add\ m\ x\ y$ is a mapping defined for all arguments on which m is defined, and also at argument x. The value of m' at argument x is y, and its values at other arguments conicide with the values of m at the same arguments. The function *remove* is defined so that *remove* $m\ x$ is a mapping that agrees with the mapping m, except that it is not defined for argument x.

The value of a mapping at a given argument can be found using the function *find*. If the value of m at argument x is defined, then evaluating *find* m x returns this value; otherwise it raises the exception *Not_found*.

The higher-order function *iter* gives a way of finding all the arguments for which a mapping is defined; *iter* f m evaluates for its side-effect the expression $f\ x\ y$ for each argument x such that m is defined at x; the value of y is the corresponding value of the mapping. There is no guarantee about the order in which different arguments are presented to f.

This interface does not reveal any information about how the *mapping* data type is implemented. The simplest implementation would represent mappings by lists of ordered pairs, with *find* being the standard function *assoc* and *iter* being a variation on the theme of the *iter* function on lists. One of the advantages of hiding the implementation is that the rest of our compiler cannot possibly depend for its correct operation on the way the *mapping* type is implemented. We are free to implement mappings as lists of ordered pairs at first, then to improve the efficiency of the compiler by substituting a more sophisticated implementation, such as the one we discuss below.

Many data structures for representing mappings have been invented, and several are described in standard algorithms texts such as [ref]. The data structure I have chosen uses *B-trees* or 2/3-*trees* to give $O(\log N)$ time for *find*. B–trees have the advantage that they cannot become unbalanced, so the worst case time for *find*, as well as the average case, is $O(\log N)$. Details of the implementation of mappings using B–trees may be found in Appendix A.

2.9 Hash tables

In the preceding section, we examined a module that provides an abstract data type of mappings. The interface to this data type is still purely functional, in the sense that creating a new mapping by adding elements does not destroy the old one. If we write

let m' = *add* $m\ x\ y$ **in** E

then the expression E can use both m and m' without any danger of interference between them. This is possible because the B–tree module creates new nodes to represent the new mapping m', possibly sharing some of the nodes used to represent m, but not destroying any of them.

In this section, I discuss a still more efficient representation of mappings that is no longer purely functional, but under the right circumstances gives constant-time insertion and look-up. The representation – *hash tables* – is necessarily not functional, because it has hidden state, and destroys the representation of the old mapping when adding a new pair to create a new mapping. Accordingly, the interface presented by the module is different:

(∗ *table* – type of tables with keys α and values β ∗)
type (α, β) *table*;;

(∗ *create* – create new table with specified number of buckets ∗)
val *create* : *int* → (α, β) *table*;;

(∗ *clear* – empty a hash table ∗)
val *clear* : (α, β) *table* → *unit*;;

(∗ *add* – add or overwrite one element ∗)
val *add* : $\alpha \to \beta \to (\alpha, \beta)$ *table* → *unit*;;

(∗ *find* – look up an element ∗)
val *find* : $\alpha \to (\alpha, \beta)$ *table* → β;;

(∗ *remove* – delete an element ∗)
val *remove* : $\alpha \to (\alpha, \beta)$ *table* → *unit*;;

(∗ *iter* – apply a function to each pair in the table ∗)
val *iter* : ($\alpha \to \beta \to$ *unit*) → (α, β) *table* → *unit*;;

One of the differences between this interface and the previous one is that *add* no longer returns a new mapping: instead, it destructively modifies an existing mapping and returns nothing. The function *create* that creates a new, empty table takes an integer argument that determines the number of 'buckets' in the table. The effect this number has on the efficiency of a hash table is explained below, but I should explain now that it is *not* an upper limit on the number of values stored in the table. If a table is created by calling *create n*, more than *n* values can be inserted in the table, but if the number of values is very much larger than *n*, then performance will suffer.

Plainly we could implement this interface by using the *B*-tree module directly, defining

type *table* == *Btree.mapping ref*;;

and making each operation use the corresponding *B*-tree operation on the current contents of the table. But we want to do better than that! Appendix A gives the details of implementing associative tables efficiently using hashing.

2.10 Formatted output

The module *print* provides a general facility for formatted output via the following three functions:

(∗ *printf* – print on standard output ∗)
val *printf* : *string* → *arg list* → *unit*;;

(∗ *fprintf* – print to a file ∗)
val *fprintf* : *out_channel* → *string* → *arg list* → *unit*;;

(∗ *sprintf* – print to a string ∗)
val *sprintf* : *string* → *arg list* → *string*;;

Calling *printf format args* formats the list of items *args* and inserts the resulting text in the places indicated by dollar signs in the string *format*. For example,

printf "$ is $ years old\n" [*fStr* "Mike"; *fNum* 34];;

prints the text "Mike is 34 years old" (followed by a newline) on standard output. Similarly, the function *fprintf* provides formatted output on an arbitrary output channel, and the function *sprintf* formats data in the same way and returns the result as a string. In our compilers, we shall actually use *fprintf* only with the standard error channel *stderr* for printing error messages.

The arguments to *printf* and friends are taken from the type *Print.arg*, which has the following interface:

```
type arg;;

(* Basic formats *)
val fNum : int → arg;;              (* Decimal number *)
val fFix : int * int → arg;;        (* Fixed-width number (val, width) *)
val fFlo : float → arg;;            (* Floating-point number *)
val fStr : string → arg;;           (* String *)
val fChr : char → arg;;             (* Character *)
val fBool : bool → arg;;            (* Boolean *)

(* fMeta – insert output of recursive call to printf *)
val fMeta : string * arg list → arg;;

(* fList – format a comma-separated list *)
val fList : (α → arg) → α list → arg;;

(* fExt – higher-order extension *)
val fExt : ((string → arg list → unit) → unit) → arg;;
```

The functions listed provide ways of converting various common types into values of type *Print.arg*. Most of the possibilities are self-explanatory, except for the functions *fMeta*, *fList* and *fExt*, which allow for extensions to the range of types that can be printed: details are given in Appendix A.

2.11 Computer representation of ML programs

The ML programs that appear in this book have been formatted in a nice way for printing: keywords appear in bold face type, and fancy symbols like → have been used in place of ASCII combinations like ->. I find that these conventions make printed ML programs much easier to read. The ML compiler, however, expects programs to be represented in ASCII form, so that the definition that appears in the book as

```
let rec eval =
  function
      Number n → n
    | Binop (w, e₁, e₂) →
        do_binop w (eval e₁) (eval e₂)
    | Variable x →
        failwith "sorry, I don't do variables";;
```

would actually look like this when submitted to the ML compiler:

```
let rec eval =
  function
      Number n -> n
    | Binop (w, e1, e2) ->
        do_binop w (eval e1) (eval e2)
    | Variable x ->
        failwith "sorry, I don't do variables";;
```

Symbol	ASCII *equivalent*
$\neq$	`<>`
$\leq$	`<=`
$\geq$	`>=`
$\rightarrow$	`->`
$\leftarrow$	`<-`
$\frown$	`^`

Table 2.1: ASCII *equivalents for special symbols*

The rules for transcribing programs as they appear in the book into the form expected by the ML compiler are very simple.[5]

- Replace all **bold-face** keywords and *italic* identifiers by the same keywords and identifiers written in ordinary type. Replace identifiers that appear in *SMALL CAPS* by the same identifiers written in all capitals.
- Replace symbols like $\leftarrow$ by equivalents like `<-` made from several ASCII characters. Table 2.1 shows the special symbols that are used in the printed ML programs in this book, together with their ASCII equivalents.
- Replace subscripted identifiers like env_0 and y'_1 by ordinary identifiers like `env0` and `y1'`.
- Replace the Greek letters α, β, etc., used for type variables, by the ASCII forms `'a`, `'b`, etc..

[5] Of course, the transcription really goes the other way, and the author has written a program *ocamlgrind* that converts the ASCII form into input for TeX that produces the printed form.

Chapter 3

Lexical analysis

Lexical analysis is the process of dividing the text of the program being compiled into 'meaningful' units, and classifying those units as keywords, identifiers, integer constants, operators, and so on. We call each lexical unit in the program text a *token*, and we call each class of lexical units recognized by the lexical analyser a *token class*.

A good way to specify the lexical rules of a programming language is to use the notation of *regular expressions*, by giving a regular expression that describes each token class. Regular expressions are less powerful than the notations we shall use later to describe the syntax rules for a whole language, but they are concise and easy to understand. Regular expressions also have the big advantage that software exists that can transform a lexical specification written using regular expressions into an efficient lexical analyser. Whilst writing lexical analysers by hand is not difficult, it is boring work, and it is much more pleasant and reliable to have the job done automatically.

3.1 Regular expressions

A regular expression describes a set of strings, such as the set of valid identifiers in a programming language. I'll describe the structure and meaning of regular expressions first, and leave until later the details of to write them in the input file for the *ocamllex* lexical analyser generator, which allows some convenient abbreviations.

The simplest regular expressions are ϵ, which denotes the empty string, and a single character a, which denotes a string that contains just that one character. Larger regular expressions can be built up using the operations of concatenation AB, alternation $A|B$ and closure $A*$. Parentheses can be used for grouping. These operations are summarized in Table 3.1.

The value of each regular expression is a *set* of strings. The simple expressions ϵ and a each denote a set containing just one string, but the $|$ operation builds expressions that contain more than one string. The closure operator $*$ allows us to write expressions that denote infinite sets of strings, because a string in the expression $A*$ can contain any number of instances of strings from A. Here are some examples of regular expressions:

- $ab{*}a$ denotes the set of strings $\{aa, aba, abba, abbba, \ldots\}$.

Empty string	ϵ	{ “ ” }
Literal Character	a	{ “a” }
Concatenation	AB	$\{\, s \frown t \mid s \in A$ and $t \in B \,\}$
Alternation	$A\|B$	$A \cup B$
Closure	$A*$	$\epsilon\|A\|AA\|AAA\| \ldots$
Grouping	(A)	A

Table 3.1: *Regular expressions*

- $a(b|c)*a$ denotes the set of strings $\{aa, aba, aca, abba, abca, \ldots\}$, that is, the set of all strings of a's, b's and c's that begin and end with an a and have any mixture of b's and c's in the middle.
- $a(b|cc)*a$ denotes a subset of this set, where the letter c must always appear in groups of two.

In regular expressions, parentheses can be used for grouping. Since the concatenation and alternative operators are associative, there is no harm in leaving out the parentheses in expressions like $A(BC)$ or $A|(B|C)$.

Regular expressions satisfy a number of algebraic laws, including the following:

- $(AB)C = A(BC)$
- $\epsilon A = A\epsilon = A$
- $A|B = B|A$
- $(A|B)|C = A|(B|C)$
- $A* = \epsilon|AA*$
- $(A*)* = A*$
- $A*A* = A*$

Most of these laws may be proved by simple manipulation of the sets of strings that are denoted by the expressions. An additional law is useful for proofs about expressions involving the closure operator. If A is a regular expression that does not contain the empty string, and $B = \epsilon|AB$, then $B = A*$.

3.2 Recognising regular expressions

If we use regular expressions to specify the lexical structure of a programming language, then we face the problem of building a program that can recognize instances of a given regular expression. Luckily, there is a systematic way of doing this, based on the observation that for each regular expression, it is possible to build a finite-state machine that recognizes instances of the expression. We won't prove that theorem here, but will content ourselves with a couple of examples of finite machines that correspond to regular expressions.

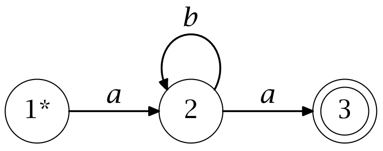

Figure 3.1: *Finite state machine for ab∗a*

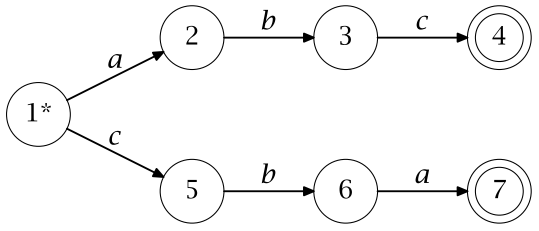

Figure 3.2: *Finite state machine for abc|cba*

Strings from the regular expression *ab∗a* are recognized by the finite machine shown in Figure 3.1. The machine starts in state 1, and begins to read characters from the string that is to be recognized. When it reads each character, the machine moves to its next state by following an arrow labelled with that character. If there is no such arrow, the machine switches itself off. Thus, after reading an initial *a*, the machine moves to state 2, and it remains in state 2 as long as it continues to read *b*'s. If it reads another *a*, the machine moves to state 3, which is marked by a double ring as an *accepting state*. Thus after reading a string that is described by the regular expression *ab∗a*, the machine will be in state 3. Any other string will either leave the machine in some other state, or cause the machine to switch off.

As another example, the machine in Figure 3.2 recognizes the expression *abc|cba*. This time, there are two accepting states, one corresponding to the string *abc* and the other corresponding to the string *cba*. We consider the machine to have recognized its input as an instance of *abc|cba* if it finishes in either of these two states.

3.3 Using ocamllex

The *lex* generator takes a specification for a lexical analyser and constructs a program that implements the specification. The original *lex* produced programs in C (or an obsolete language called RATFOR), but we shall use a version called *ocamllex* that produces an ML program as its output. The program generated by *lex* (whether in C or ML) can be compiled in the ordinary way, and incorporated with other code to build a compiler.

The notation we have been using for regular expressions is well-suited to studying their properties, but would be cumbersome for specifying lexical analysers in practice. *Ocamllex* uses the following conventions:

• A regular expression that is a literal character is written in '. . . ' quotes, like a character constants in ML.

- Strings of literal characters may be written in double quotes "...", like the string constants of ML. This is an abbreviation that combines literal characters and concatenation.
- The expression 'a'|'b'|'c' may be abbreviated to ['a''b''c'] or to ['a'–'c']. Thus the expression ['A'–'Z''a'–'z'] describes any upper-case or lower-case letter. An initial ^ character takes the complement of the set of characters, so that [^'A'–'Z''a'–'z'] denotes any single character that is *not* a letter.
- The notation _ denotes any single character at all.
- The expression A? denotes an optional occurrence of A; it is equivalent to $A|\epsilon$.
- The expression $AA*$ (denoting one or more instances of A) may be abbreviated to A+.
- The notation eof denotes the end of an input file.

Other versions of *lex* have similar conventions, although they are not exactly the same.

The main part of a *ocamllex* specification consists of a set of *rules* of the form

```
regexp      { value }
```

where *regexp* is a regular expression, and *value* is an ordinary ML expression. The program generated by *ocamllex* defines a function that reads and recognizes one token from an input file each time it is called. The function finds the longest string from the input that matches the regular expression in one of the rules, and returns the value of the corresponding expression: usually these values are taken from an enumerated type with one element for each token class recognized by the lexical analyser. Within the *value* expression, the sub-expression *lexeme lexbuf* can be used: this returns the string that was matched by the regular expression.

Figures 3.3 and 3.4 show a lexical analyser for the PASCAL-like language that we shall implement in Chapter 5. Many of the rules are simply literal strings that match an operator or punctuation sign; the corresponding value is a constant token type. More interesting is the rule for identifiers and keywords. The compiler separates keywords like if from ordinary identifiers by using a small hash table. Thus all identifiers and keywords initially match the regular expression

```
['A'–'Z''a'–'z']['A'–'Z''a'–'z''0'–'9''_']*
```

which describes the set of strings of letters, digits and underscores that start with a letter. The value expression uses a function *lookup* to look up the actual token in the hash table and find whether it is a keyword or not. An alternative approach would be to write an explicit rule in the *lex* script for each keyword, but this leads to a much bigger lexical analyser.

The rules for white-space [' ''\t']+ and newlines "\n" call the lexical analyser recursively in the action part; this trick causes the corresponding tokens to be discarded, and the token returned by the lexical analyser is the next non-blank token it finds.

```
{
open Parser;; open Lexing;; open Tree;; open List;;

(* lineno – line number for use in error messages *)
let lineno = ref 1;;

(* A little table to recognize keywords *)
let kwtable =
  let insert t (k, v) = Btree.add k v t in
  fold_left insert Btree.empty
    [ ("begin", BEGIN); ("do", DO); ("if", IF ); ("else", ELSE);
      ("end", END); ("then", THEN); ("while", WHILE); ("print", PRINT);
      ("and", MULOP And); ("div", MULOP Div); ("or", ADDOP Or);
      ("not", MONOP Not); ("mod", MULOP Mod) ];;

let lookup s =
  try Btree.find s kwtable with Not_found → IDENT s;;
}

rule token =
  parse
      ['A' – 'Z''a' – 'z']['A' – 'Z''a' – 'z''0' – '9''_']*
                           { lookup (lexeme lexbuf) }
    | ['0' – '9']+         { NUMBER (int_of_string (lexeme lexbuf)) }
    | ";"                  { SEMI }
    | "."                  { DOT }
    | ":"                  { COLON }
    | "("                  { LPAR }
    | ")"                  { RPAR }
    | ","                  { COMMA }
    | "="                  { RELOP Eq }
    | "+"                  { ADDOP Plus }
    | "-"                  { MINUS }
    | "*"                  { MULOP Times }
    | "<"                  { RELOP Lt }
    | ">"                  { RELOP Gt }
    | "<>"                 { RELOP Neq }
    | "<="                 { RELOP Leq }
    | ">="                 { RELOP Geq }
    | ":="                 { ASSIGN }
    | [' ''\t']+           { token lexbuf }
    | "(*"                 { comment lexbuf; token lexbuf }
    | "\n"                 { incr lineno; token lexbuf }
    | _                    { BADTOK }
    | eof                  { EOF }
```

Figure 3.3: *Lexical analyser*

```
and comment =
  parse
      "*)"                 { () }
    | "\n"                 { incr lineno; comment lexbuf }
    | _                    { comment lexbuf }
    | eof                  { () }
```

Figure 3.4: *Lexical analyser for comments*

The code in Figure 3.4 skips over comments in the PASCAL program, and is invoked when the main lexical analyser sees the characters "(∗". It would be possible to write a regular expression that describes the syntax of comments, but very long comments could cause the buffer used by the generated lexical analyser to overflow.

The specification of the lexical analyser is theoretically ambiguous, because (for example), the text <> could be analysed as a single token, or as a < token followed by a > token. *Lex* resolves this ambiguity by always taking the longest string from the input that matches one of the rules.

3.4 Writing lexical analysers by hand

Although *lex* provides a convenient way to automate the process of building lexical analysers, there are times when it is necessary to write one by hand: for example, a calculator program for a hand-held computer might have to fit in a very small memory space, and a hand-written lexical analyser may be smaller than one built using *lex*. In such circumstances, it is more likely that a compact, low-level language like C would be used, but we will continue to use Objective CAML for consistency with the rest of this book.

Most languages make it possible to recognize where a token ends by looking at just one character from the input: for example, an identifier typically consists of a sequence of letters and digits, and ends just before the first character in the input that is not a letter or digit. It is convenient to introduce a reference cell *ch* that always contains the next input character, and to package the lexical analyser as a function *next_token* : *unit* → *token* that expects *ch* to contain the first character of a token when *next_token* is called, and leaves *ch* containing the first character after the end of the token when it returns.

The function *next_token* (Figure 3.5) decides on the kind of token by looking at the initial value of *ch*. In some cases, this character forms a token by itself, so all what is needed is to advance *ch* to the next character using a function *nextch* and to return the appropriate value of type *token*. In other cases such as identifiers and decimal numbers, the token may consist of several characters, and in that case there is a loop that consumes these characters one at a time until the value of *ch* no longer belongs to this token.

This straight-forward technique does not have quite the same power as general regular expressions. For example, the regular expression $(ab)*$ describes alternating sequences of a's and b's; if the input is $ababac$, then the expression matches $abab$, and it is not possible to tell just by looking at the next character (a) that the token ends here. Nevertheless, almost every input

```
let rec next_token () =
  match !ch with
      'A'..'Z' | 'a'..'z' →
        let ident = String.create 128 in
        let k = ref 0 in
        while is_letter !ch || is_digit !ch || !ch = '_' do
          ident.[!k] ← !ch; incr k; nextch ()
        done;
        IDENT (String.sub ident 0 !k)
    | '0'..'9' →
        let number = String.create 128 in
        let k = ref 0 in
        while is_digit !ch do
          number.[!k] ← !ch; incr k; nextch ()
        done;
        if !ch = '.' then begin
          number.[!k] ← !ch; incr k; nextch ();
          while is_digit !ch do
            number.[!k] ← !ch; incr k; nextch ()
          done
        end;
        NUMBER (float_of_string (String.sub number 0 !k))
    | '(' → nextch (); OPEN
    | ')' → nextch (); CLOSE
    | '=' → nextch (); EQUAL
    | '+' → nextch (); PLUS
    | '-' → nextch (); MINUS
    | '*' → nextch (); TIMES
    | '/' → nextch (); DIVIDE
    | ' ' | '\t' →
        while !ch = ' ' || !ch = '\t' do nextch () done;
        next_token ();
    | '#' → EOF
    | _ → nextch(); BADTOK;;
```

Figure 3.5: *Function next_token*

language encountered in practice can be handled with the simple technique descried here.

Exercises

3.1 Give regular expressions that describe the following sets of strings over the alphabet $\{a, b, c\}$:

(a) The set of strings containing a certain number of a's followed by a certain number of b's, such that the total length of the string is odd.

(b) The set of all strings that do not contain ab.[1]

(c) The set of strings that contain at least two c's.

(d) The set of strings that do not contain more than two c's.

(e) The set of all strings that do not contain more than two c's in a row.

3.2 A regular set is a set of strings that can be described by a regular expression. Which of the following are regular sets, and why?

(a) The set of currently valid car registration numbers.

(b) The set of car registration numbers that would fetch more than 1,000 pounds at an auction of 'collectible' registrations.

(c) The set of strings over the alphabet $\{u, d\}$ that describe a possible sequence of motions of the Computing Laboratory's lift – d means go down one floor, u means go up.

3.3 Prove the following equations between regular expressions:

(a) $(a|aa)* = a*$.

(b) $(a|b)* = a*(ba*)*$.

3.4 The scanner for comments shown in Figure 3.4 deals only with comments that are not nested: a comment begins with (∗ and ends with the next occurrence of ∗), even if another occurrence of (∗ appears between them. Nested comments are useful because they make it possible to 'comment out' a section of program by enclosing it in comment brackets, even if the code itself contains comments. Show how, by adding a counter, the lexical analyser can be extended to handle nested comments properly.

[1] This exercise has practical utility: a comment in C is can consist of any string of characters that doesn't contain ∗/.

Chapter 4

Syntax analysis

Regular expressions provide a useful notation for describing the small-scale syntax of a programming language, but there are many syntactic features of programming languages that they cannot describe. For example, there is no regular expression that describes the set of strings $S = \{ a^n b^n \mid n \geq 0 \}$, that is, the set of strings that consist of some number of a's followed by the same number of b's. We can tell that this is so, because we know that every regular expression can be recognized by a finite-state machine, and there can be no finite-state machine that recognizes the set S. Why not? Because after the machine has read the $a \frown n$ part of the input, it must remember the number n of b's that it is expecting. Since this may be an arbitrarily large number, no finite-state machine can have enough states to cover all the possibilities.

The set S does not look much like part of a programming language, but there are other sets that pose the same sort of problem: the set of expressions with properly-matched parentheses, for example, is also not described by any regular expression. Because of the weakness of regular expressions in describing the syntax of languages, it is useful to introduce a more powerful description mechanism, *context-free grammars*, that is able to describe the set S and others like it.

4.1 Context-free grammars

A context-free grammar involves two alphabets of symbols:

- *terminal symbols*, which may appear in the strings described by the grammar, and
- *non-terminal symbols*, which are used internally in the grammar, but do not appear in the strings it describes.

In compiler applications, the terminal symbols are commonly tokens returned by the lexical analyser, and the non-terminal symbols are given names like *expression* and *statement*: they correspond to the major grammatical categories of the language. To make our examples easier to follow, we'll use characters directly as the terminal symbols, ignoring the lexical analysis phase. Everything we say, however, works just the same if terminal symbols are actually the tokens output by a lexical analyser. The grammar itself is a set of *production rules*, each with the form

$$A \rightarrow \alpha$$

where A is a non-terminal symbol, and α is a string of terminal and non-terminal symbols. The production rule is an instruction that allows the non-terminal A to be replaced by the string α as part of a process that generates syntactically correct programs. Grammars of this kind are called *context-free* because a production rule allows the symbol A on the left to be replaced by the string α without any restriction on the context in which the replacement happens.

Each grammar defines a process or 'game' that generates each string that is well-formed according to the grammar. The game begins with a single non-terminal symbol S that has been chosen as the *start symbol* of the grammar. Each move in the game consists of choosing a non-terminal symbol in the string that has been generated so far, and replacing it by the right-hand side of some production that has the non-terminal in question as its left-hand side. The game finishes when the string contains no more non-terminal symbols.

As an example, here is a grammar G_0 that describes expressions made up of the variables x and y and the operators + and ∗:

$$\begin{aligned}
\mathit{expr} &\rightarrow \mathsf{x} \\
\mathit{expr} &\rightarrow \mathsf{y} \\
\mathit{expr} &\rightarrow \mathit{expr} + \mathit{expr} \\
\mathit{expr} &\rightarrow \mathit{expr} * \mathit{expr}
\end{aligned}$$

There is just one non-terminal *expr*, and it is also the start symbol. From this grammar, we can generate the string x + x ∗ y by the following sequence of moves:

$$\begin{aligned}
\mathit{expr} &\Rightarrow \underline{\mathit{expr}} * \mathit{expr} \\
&\Rightarrow \underline{\mathit{expr}} + \mathit{expr} * \mathit{expr} \\
&\Rightarrow \mathsf{x} + \underline{\mathit{expr}} * \mathit{expr} \\
&\Rightarrow \mathsf{x} + \mathsf{x} * \underline{\mathit{expr}} \\
&\Rightarrow \mathsf{x} + \mathsf{x} * \mathsf{y}
\end{aligned}$$

At each stage, the non-terminal that is underlined is the one being replaced in the move. The double-tailed arrow ⇒ is used to indicate a move of replacing a non-terminal by a string, rather than the production (written with a single-tailed arrow) that justifies the move.

In general, each string that can be generated in the game can be generated in many ways. Some of these variations are trivial: for example, the final three moves in our example could have been made in any order, because they replace the three separate occurrences of *expr* with the symbols x and x and y, and these three replacements are independent of each other. Other derivations, however, reveal that there may be more than one essentially different way to generate the same string. For example, the string x+x∗y can also be generated by the derivation

$$\begin{aligned}
\underline{\mathit{expr}} &\Rightarrow \mathit{expr} + \underline{\mathit{expr}} \\
&\Rightarrow \underline{\mathit{expr}} + \mathit{expr} * \mathit{expr}
\end{aligned}$$

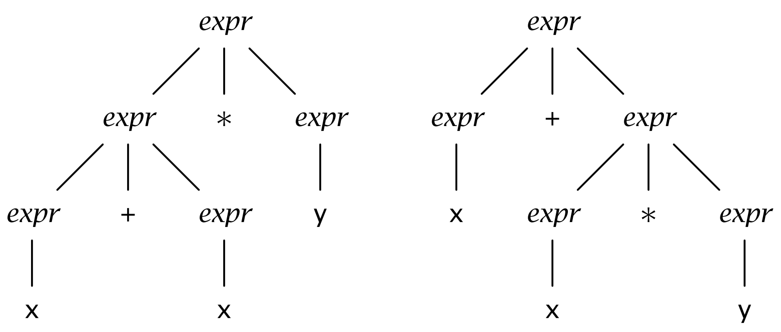

Figure 4.1: *Two derivation trees for* x+x∗y *in* G_0

$$
\begin{aligned}
&\Rightarrow \quad \mathsf{x} + \underline{\mathit{expr}} * \mathit{expr} \\
&\Rightarrow \quad \mathsf{x} + \mathsf{x} * \underline{\mathit{expr}} \\
&\Rightarrow \quad \mathsf{x} + \mathsf{x} * \mathsf{y}
\end{aligned}
$$

This derivation is truly different, because the second step, which introduces the operator ∗, operates on an occurrence of *expr* that was introduced by the first step, which introduced +. The derivation seems to give the final expression the structure x + (x ∗ y) instead of the structure (x + x) ∗ y that it had before. We can reveal the essential difference between the two derivations by viewing them as *derivation trees* rather than linear sequences of moves. Figure 4.1 shows the two derivations as trees, and it is clear that one of them makes ∗ the principal operator of the expression, and the other one makes + the principal operator.

In building a compiler, we are very definitely interested in the structure that a grammar assigns to a program, because it is on this structure that we plan to base all the work of compiling. This means that the grammar above, though it correctly generates all the well-formed expressions and no others, is not of much use to us. The grammar generates the expression x + x ∗ y in two ways, and only one of these reflects the convention that ∗ binds more tightly than +. For building a compiler, we would rather have a grammar that generates each well-formed expression in only one way, and in a way that corresponds to the meaning we intend the expression to have. For the small class of expressions generated by our example grammar, an alternative grammar G_1 contains the following productions:

$$
\begin{aligned}
&\mathit{expr} &&\rightarrow\quad \mathit{term} \mid \mathit{expr} + \mathit{term} \\
&\mathit{term} &&\rightarrow\quad \mathit{factor} \mid \mathit{term} * \mathit{factor} \\
&\mathit{factor} &&\rightarrow\quad \mathsf{x} \mid \mathsf{y}
\end{aligned}
$$

The notation $A \rightarrow \alpha \mid \beta$ is a shorthand for the two productions $A \rightarrow \alpha$ and $A \rightarrow \beta$. This grammar is *unambiguous*, and generates each expression in only one way. Figure 4.2 shows the derivation for x + x ∗ y as a tree, making it clear that + is the principal operator.

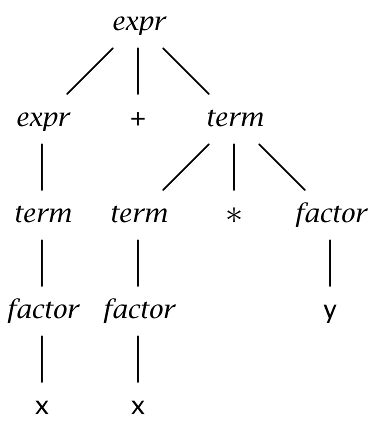

Figure 4.2: *The unique derivation tree for* x+x∗y *in* G_1

4.2 Parsing

For each regular expression, we found that there is a simple algorithm for testing whether a given string is in the language described by the regular expression: in fact these algorithms are so simple that they can be expressed as finite-state machines.

It's natural to ask whether the same recognition problem can be solved for context-free grammars. The answer is yes, but the algorithms involved are more complex than finite-state machines. In general, the parsing problem can be solved by a kind of dynamic programming method that has time complexity $O(N^3)$, where N is the length of the input string. This is far too slow to be used in practical compilers, so generally compiler writers use other methods that can only handle a restricted class of grammars, but give a much faster parser. One such method, *LR* parsing, is the basis for the parser generator *yacc*, which takes a context-free grammar as input, and produces as its output a program that solves the recognition problem for that grammar. The original *yacc* produced C programs as its output, but we shall use the version called *ocamlyacc*, which produces programs in Objective CAML. Before looking at the form of the input script for *ocamlyacc*, it's worth looking at the ideas behind the parsing method it uses, because that will help us to understand what class of grammars can be handled.

A parser built by any version of *yacc* reads its input once from left to right, and it maintains a stack of symbols – terminals and non-terminals – that represents the portion of the input that it has read so far. When parsing the input x + x ∗ y using our unambiguous grammar for expressions, an *LR* parser would go through the following sequence of configurations:

Stack	Input	Action
ϵ	x + x ∗ y #	shift
x	+ x ∗ y #	reduce *factor* → x
factor	+ x ∗ y #	reduce *term* → *factor*
term	+ x ∗ y #	reduce *expr* → *term*
expr	+ x ∗ y #	shift

$expr\ +$	$\mathsf{x} * \mathsf{y}\ \#$	shift
$expr + \mathsf{x}$	$* \mathsf{y}\ \#$	reduce $factor \rightarrow \mathsf{x}$
$expr + factor$	$* \mathsf{y}\ \#$	reduce $term \rightarrow factor$
$expr + term$	$* \mathsf{y}\ \#$	shift
$expr + term\ *$	$\mathsf{y}\ \#$	shift
$expr + term * \mathsf{y}$	$\#$	reduce $factor \rightarrow \mathsf{y}$
$expr + term * factor$	$\#$	reduce $term \rightarrow term * factor$
$expr + term$	$\#$	reduce $expr \rightarrow expr + term$
$expr$	$\#$	accept

In this table, the column headed 'Stack' represents the stack of the parser, with the top of the stack at the right, and the column headed 'Input' represents the part of the input that has not been consumed, followed by a dummy symbol #, meaning 'end of file'. At each stage, the parser may perform one of two actions:

- it may consume the next input symbol and add it to the stack, an action called 'shift'.
- if the right-hand side of a production $A \rightarrow \alpha$ is present at the top of the stack, the parser may replace it by the left-hand side, an action called 'reduce $A \rightarrow \alpha$'

If the parser successfully recognizes the input, then it will eventually reach a configuration where the input is empty (apart from the marker #), and the stack contains just the start symbol of the grammar. If so, then the sequence of reduce actions *reading backwards* gives a derivation of the input sentence. For this reason, we say it is a *bottom-up* parser, because it constructs the derivation by beginning at the leaves of the derivation tree and working towards the root. This derivation will be one in which the *rightmost* non-terminal is always the one replaced. This explains the *R* in the term '*LR* parsing'; the *L* means that the input is read from left to right.

In the example, we can write the following derivation by reading the reduce actions backwards:

$$\begin{aligned}
\underline{expr} &\Rightarrow expr + \underline{term} \\
&\Rightarrow expr + term * \underline{factor} \\
&\Rightarrow expr + \underline{term} * \mathsf{y} \\
&\Rightarrow expr + \underline{factor} * \mathsf{y} \\
&\Rightarrow \underline{expr} + \mathsf{x} * \mathsf{y} \\
&\Rightarrow \underline{term} + \mathsf{x} * \mathsf{y} \\
&\Rightarrow \underline{factor} + \mathsf{x} * \mathsf{y} \\
&\Rightarrow \mathsf{x} + \mathsf{x} * \mathsf{y}
\end{aligned}$$

Yacc generates *LALR*(1) parsers, which look at only one symbol from the unconsumed input before deciding whether to shift or reduce. This means that the only information that the parser can use in deciding what on the next action is the contents of the stack and one input symbol. For some grammars, and certainly for any ambiguous grammar, the parser will not always have enough information to make this choice. If so, then *yacc* reports a ‘conflict’ when it is trying to generate a parser for the grammar: either a ‘shift/reduce conflict’, meaning that the parser sometimes cannot decide whether to shift the next symbol immediately or carry out a reduction first, or a ‘reduce/reduce conflict’, meaning that the parser sometimes cannot decide whether to reduce using one production or another one. In these cases, a parser would either have to choose one action or the other in an arbitrary way – in which case making the wrong choice may cause the parser to fail to recognize an input that was actually correct – or it would have to try both alternatives, perhaps by backtracking, and this would quickly become inefficient if many conflists were encountered. Luckily, most programming language constructs can be described in a way that avoids these conflicts.

As an example of a case where parsing conflicts arise, we can consider parsing the expression x + x ∗ y using the ambiguous grammar for expressions. The first few configurations are these:

Stack	Input	Action
ϵ	x + x ∗ y #	shift
x	+ x ∗ y #	reduce *expr* → x
expr	+ x ∗ y #	shift
expr +	x ∗ y #	shift
expr + x	∗ y #	reduce *expr* → x
expr + *expr*	∗ y #	. . .

Now there is a choice that cannot be made with the information available to the parser. It can either shift the ∗, in which case the run finishes like this . . .

expr + *expr*	∗ y #	shift
expr + *expr* ∗	y #	shift
expr + *expr* ∗ y	#	reduce *expr* → y
expr + *expr* ∗ *expr*	#	reduce *expr* → *expr* ∗ *expr*
expr + *expr*	#	reduce *expr* → *expr* + *expr*
expr	#	accept

. . . or it can reduce with *expr* → *expr* + *expr* and finish like this . . .

expr + *expr*	∗ y #	reduce *expr* → *expr* + *expr*
expr	∗ y #	shift
expr ∗	y #	shift
expr ∗ y	#	reduce *expr* → y
expr ∗ *expr*	#	reduce *expr* → *expr* ∗ *expr*
expr	#	accept

One of these corresponds to parsing the expression as x + (x ∗ y), and the other to parsing it as (x + x) ∗ y.

There are two solutions to this problem that an ambiguous grammar will always cause. One solution is to discard the grammar, and use an equivalent unambiguous grammar instead. Another solution is to resolve the conflict by other means. *Yacc* allows the compiler writer to specify the precedence and associativity of operators in the source language, and uses this information to resolve shift/reduce conflicts.

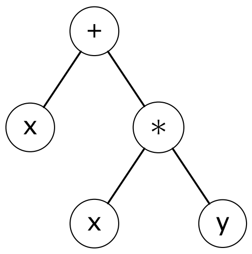

Figure 4.3: *Abstract syntax tree for* x+x∗y

We can see how this works by looking at the example. When the stack contains *expr* + *expr* and the next symbol is ∗, what the parser should do depends on whether + or ∗ is the operator with higher binding power. If it is +, the right thing to do is *reduce*, so that *expr* + *expr* becomes an *expr* that is later incorporated as the left-hand operand of ∗. If ∗ binds more tightly, then the right thing to do is *shift*, so that the whole sub-expression x ∗ y can be collected and reduced to an *expr* that is the right-hand operand of +.

Trying to parse an expression like x – x – y reveals that single operators can also cause shift/reduce conflicts that need to be resolved. Here the relevant fact is that – associates to the left, so that the expression should be parsed as (x – x) – y; when two minus signs meet, the right thing to do is *reduce*, so that the left-hand minus becomes part of the operand of the right-hand minus.

Yacc's rules are as follows: when trying to decide whether to shift or to reduce by a certain production, compare the precedence of the look-ahead symbol (i.e., the first symbol from the unconsumed input) with the precedence of the last terminal symbol in the right-hand side of the production. If the look-ahead symbol has higher precedence, then shift. If it has lower precedence, then reduce. If the two symbols have the same precedence, then look at their associativity. If they are left associative, then reduce; if they are right associative, then shift. Operator precedence and associativity can only be used to resolve shift/reduce conflicts, never reduce/reduce conflicts.

4.3 Semantic actions

A parser is not much use if all it can do is tell well-formed programs from those that contain syntax errors. To be useful in a compiler, the parser must also produce some information about the syntactic structure of the source program that can be used by the rest of the compiler. It might produce, for example, a derivation of the source program in the grammar of the programming language, or perhaps a derivation tree of the kind shown in Figure 4.2. More convenient than either of these, however, is an *abstract syntax tree* for the source program. Figure 4.3 shows an abstract syntax tree for the expression x + x ∗ y, and Figure 1.1 in Chapter 1 shows the abstract syntax tree of a larger piece of program.

What is the difference between a derivation tree and an abstract syntax tree? In a derivation tree, the nodes are labelled with symbols: terminal symbols at the leaves, and non-terminal symbols at internal nodes. An abstract syntax tree has nodes that are really labelled with *names of productions*. Thus the node that is marked $*$ in Figure 4.3 is effectively labelled with the production

$$expr \rightarrow expr * expr.$$

Abstract syntax trees have two advantages: they are more compact, and the labels on the nodes make it easy to discover the structure of the source program.

Our way of describing the abstract syntax of a programming language (that is, the set of abstract syntax trees) will be to give a recursive type definition in ML, as we did for the abstract syntax of expressions in Lab 1. There is no absolute need for the abstract syntax to mirror the concrete syntax exactly. For example, parentheses may be used in expressions purely for grouping, and we can represent the grouping directly in the structure of the abstract syntax tree for an expression; so there is no need to represent the parentheses explicitly in the abstract syntax tree. Similarly, the elsif keyword of picoPascal (like Modula-2) is just a shorthand: the program

```
if expr₁ then
   stmts₁
elsif expr₂ then
   stmts₂
else
   stmts₃
end
```

is an abbreviation for

```
if expr₁ then
   stmts₁
else
   if expr₂ then
      stmts₂
   else
      stmts₃
   end
end
```

and these two programs can be represented by the same abstract syntax tree, thereby simplifying the rest of the compiler.

We can extend the shift-reduce parsing method so that it also builds an abstract syntax tree for the source program, in addition to checking that it obeys the syntax rules of the language. To do this, we associate with each production an expression that builds a node in the abstract syntax tree. For example, suppose the grammar for expressions contains a production like

$$expr \rightarrow expr + expr.$$

If e_1 and e_2 are the abstract syntax trees for the two sub-expressions, then the abstract syntax tree for the whole expression is *Binop* (*Plus*, e_1, e_2). The parser keeps a stack of *values* alongside the stack of symbols that it uses to represent the part of the input it has read. This value stack contains the abstract syntax tree for each phrase that has been recognized, and every reduce action pops some values off the value stack and pushes a larger piece of tree, built by evaluating the expression associated with the production being reduced.

Following the conventions of *ocamlyacc*, we can write the expression associated with a production in curly brackets, like this:

$$expr \rightarrow expr + expr \quad \{\, Binop\,(Plus, \$1, \$3)\, \}$$

Here the markers $1 and $3 refer to the values associated with the first and third symbols on the right-hand side, and the expression gives the value associated with the left-hand side.

If every production has an action that builds a node in the abstract syntax tree, then when the parser eventually reaches its accepting state, with the start symbol on the symbol stack and no input left, the value stack will contain just the abstract syntax tree for the whole input.

4.4 Using ocamlyacc

You now know almost all you need to build parsers with *ocamlyacc*. We illustrate the details with an example, taken from the materials for Lab 2. Figure 4.4 shows a compact summary of the syntax of the dialect of PICO-PASCAL that is accepted by the compiler provided in the lab kit. In this summary, keywords in bold and symbols in "..." quotes are literal symbols of the language. The notation [*stuff*] stands for an optional occurrence of *stuff*, and { *stuff* } stands for zero or more repetitions of *stuff*. Thus, for example, [*stmt* { ";" *stmt* }] stands for an optional list of identifiers (*stmt*) separated by semicolons. The binary operators (*binop*) are shown in decreasing order of binding power; they bind less tightly than the unary operators (*monop*). This form of syntax summary is not strictly a context-free grammar, but it is fairly easy to expand it into one so that it can be processed by *yacc*.[1] Figure 4.5 shows the definition of a type of abstract syntax trees for this language, and Figures 4.6 and 4.7 show the script for *ocamlyacc*, including semantic actions that construct an abstract syntax tree of this type.

An *ocamlyacc* script is in two sections, separated by a line that contains just "%%". The first section contains declarations of the terminal symbols and start symbol of the grammar, operator precedences, and so on, and the second part contains the productions.

Ocamlyacc is designed to build parsers that will be used with a lexical analyser, especially the kind of lexical analyser that can be built with *ocamllex*; because of this, the terminal symbols in a grammar are not characters but tokens. The compiler writer can specify a list of token classes at the start of the parser description, and *ocamlyacc* generates an ML type definition for the type *token* that can be used in the lexical analyser to create token values. For example, the parser description may contain the declaration

%token *SEMI DOT COLON LPAR RPAR COMMA*

This declares six token classes that may be returned by the lexer and may be used in writing the grammar. Typically, some token classes correspond

[1] It might be nice to have a version of *yacc* that could handle abbreviations like the ones used in Figure 4.4; on the other hand, it is hard to see how the semantic actions would be written

```
program  →  begin stmts end "."

stmts    →  stmt { ";" stmt }

stmt     →  empty
         |  ident ":=" expr
         |  print expr
         |  if expr then stmts [ else stmts ] end
         |  while expr do stmts end

expr     →  ident
         |  number
         |  monop expr
         |  expr binop expr
         |  "(" expr ")"

monop    →  "-" | not

binop    →  "*" | "/"
         |  "+" | "-"
         |  "<" | "<=" | "=" | "<>" | ">" | ">="
```

Figure 4.4: *Syntax summary of* PICOPASCAL

```
type program = Program of stmt

and stmt =
      Skip
    | Seq of stmt list
    | Assign of name * expr
    | Print of expr
    | IfStmt of expr * stmt * stmt
    | WhileStmt of expr * stmt

and expr =
      Number of int
    | Variable of name
    | Monop of op * expr
    | Binop of op * expr * expr;;
```

Figure 4.5: *Abstract syntax of* PICOPASCAL

```
%{
open Tree;;
%}

%token ⟨Tree.ident⟩ IDENT
%token ⟨Tree.op⟩ MONOP MULOP ADDOP RELOP
%token ⟨int⟩ NUMBER

/∗ punctuation and keywords ∗/
%token SEMI DOT COLON LPAR RPAR COMMA MINUS
%token ASSIGN EOF BADTOK
%token BEGIN DO ELSE END IF THEN WHILE PRINT

/∗ operator priorities ∗/
%left RELOP
%left ADDOP MINUS
%left MULOP
%nonassoc MONOP

%type ⟨Tree.program⟩ program

%start program

%%

program :
    BEGIN stmts END DOT                 { Program (Seq $2) } ;

stmts :
    stmt                                { [$1] }
  | stmt SEMI stmts                     { $1 :: $3 } ;

stmt :
    /∗ empty ∗/                         { Skip }
  | IDENT ASSIGN expr                   { Assign (makeName $1, $3) }
  | PRINT expr                          { Print $2 }
  | IF expr THEN stmts else_part END    { IfStmt ($2, Seq $4, Seq $5) }
  | WHILE expr DO stmts END             { WhileStmt ($2, Seq $4) } ;

else_part :
    /∗ empty ∗/                         { [] }
  | ELSE stmts                          { $2 } ;
```

Figure 4.6: *Ocamlyacc script for* picoPascal *(part 1)*

```
expr :
      IDENT                          { Variable (makeName $1) }
    | NUMBER                         { Number $1 }
    | MONOP expr                     { Monop ($1, $2) }
    | MINUS expr %prec MONOP         { Monop (Uminus, $2) }
    | expr MULOP expr                { Binop ($2, $1, $3) }
    | expr ADDOP expr                { Binop ($2, $1, $3) }
    | expr MINUS expr                { Binop (Minus, $1, $3) }
    | expr RELOP expr                { Binop ($2, $1, $3) }
    | LPAR expr RPAR                 { $2 } ;
```

Figure 4.7: *Ocamlyacc script for* PICOPASCAL *(part 2)*

to individual keywords, some to operator symbols and punctuation marks, and some to larger classes like identifiers and integer constants.

In addition to belonging to a token class, a token can have a value associated with it: for example, every integer constant might belong to the same class *NUMBER*, but each token in this class would then be associated with an integer that would be the value of the constant. This would be specified by writing the following declaration:

```
%token⟨int⟩ NUMBER
```

The lexical analyser may then return a value such as *NUMBER* 3, and the parser description may contain productions like this:

```
expr : NUMBER                        { Number $1 }
```

The action associated with this production takes the integer value that was returned by the lexical analyser, and makes it part of the abstract syntax tree that is built for the expression. Other kinds of tokens that are commonly associated with values include identifiers and string constants. In the scripts of Figures 4.6 and 4.7, each group of binary operators that have the same precedence are represented by a single token class, and the actual operator is identified by an associated value drawn from an enumerated type. This means that the lexical analyser returns the tokens < and > as *RELOP Lt* and *RELOP Gt* repectively, and returns the token + and *ADDOP Plus*. This is possible because the parser need not distinguish between operators of the same precedence as far as parsing is concerned, and just needs to build the appropriate operator into the abstract syntax tree.

The next part of the script consists of lines beginning with %left, %right or %nonassoc that declare the precedence and associativity of operators. All the tokens on the same line have the same precedence, and they are left- or right- or non-associative according to the keyword that begins the line. Operators declared on later lines bind more tightly than those declared on earlier lines.

The *ocamlyacc* script also needs to specify one or more start symbols for the grammar, and to give the type of the abstract syntax tree that is associated with them. In the example, the start symbol is *program*, and it is associated with an abstract syntax tree of type *prog*. This is specified by the two declarations

```
%start program
%type⟨Tree.prog⟩ program
```

(The qualified name *Tree.prog* shows that the type *prog* is defined in the *Tree* module.)

In the abstract syntax tree, identifiers are represented by a type *name* to allow for annotations, and there is a function *makeName* : *ident* → *name* that constructs an element of this type from an identifier, such as might be returned from the lexical analyser. For present purposes, we can pretend that *name* is the same as *ident*, and that *makeName* is the identity function.

The script in Figures 4.6 and 4.7 illustrates a useful technique for dealing with the minus sign, which may be used both as a binary operator and as a unary operator, with different precedence in each case. The difference in precedence is needed in an expression such as -3+4, which must be interpreted as (-3)+4. A minus sign is represented by the token *MINUS* with the same precedence as other additive operators like +, and its use as a binary operator is covered by the rule

```
expr : expr MINUS expr                { Binop (Minus, $1, $3) }
```

An additional rule

```
expr : MINUS expr %prec MONOP         { Monop (Minus, $2) }
```

covers its use as a unary operator, and the annotation %prec MONOP causes the rule to be treated, as far as resolving parsing conflicts is concerned, as if the operator were *MONOP* rather than *MINUS*. This means, for example, that when there is a choice between reducing with this rule and shifting a plus sign – represented by the token *ADDOP Plus* – the parser will reduce, because *MONOP* has a higher precedence than *ADDOP*.

The parser script for the complete picoPascal language, including the declarations, data types and procedures we shall add later in the book, is about twice the size of this example, but it contains just more of the same sort of thing.

4.5 Writing parsers by hand

Although *yacc* is a convenient tool for building parsers, it sometimes happens that it is more convenient to build a parser by hand, perhaps because there no useable version of *yacc* is on hand. The simplest method for doing this is called *recursive descent*, because it involves writing a function to recognize each syntactic class of the language; these functions call each other recursively in the same way that recursion is used to structure the grammar itself.

To illustrate the technique, I shall use the language of arithmetic expressions that appeared in Lab lab1. This is small enough to treat in detail, but also provides the opportunity to explain the main difficulties that you are likely to encounter in building parsers using recursive descent.

The first step is to write a grammar for the language that does not rely on operator precedence to resolve ambiguities, as do the grammars we have used with *yacc*. For the moment, we will also avoid the abbreviations that

equation	→	*expr*
	|	IDENT EQUAL *expr*
expr	→	*term*
	|	*expr* PLUS *term*
	|	*expr* MINUS *term*
term	→	*factor*
	|	*term* TIMES *factor*
	|	*term* DIVIDE *factor*
factor	→	NUMBER
	|	IDENT
	|	OPEN *expr* CLOSE

Figure 4.8: *Grammar from Lab 1*

were used in Figure 4.4, and write out all the productions explicitly. Our grammar is shown in Figure 4.8, using the tokens we defined in Section 3.4.

We begin our parser by declaring a global variable *tok* that will always contain the next token from the input, and defining a function *scan* that uses the lexer to advance *tok* to the next token:

```
let tok = ref BADTOK;;

let scan () =
  tok := Mylex.next_token ();;
```

The variable *tok* is initially set to the illegal token BADTOK, but we will initialize it later by calling *scan* to set it to the first token from the input.

For each nonterminal *N* of our grammar (*equation*, *expr*, *term*, *factor*), we now write a function called *p_N* that will advance *tok* past all the tokens in an instance of *N*. For example, when *p_factor* is called, it expects *tok* to contain the first token of a *factor*. It will call *scan* repeatedly, until the value of *tok* is the first token after the *factor*. If the first token is a number NUMBER *v* or an identifier IDENT *x*, this means calling *scan* just once; on the other hand, if the first token is an opening parenthesis (OPEN), this indicates that what follows is a nested expression and a closing parenthesis (CLOSE). The function *p_factor* does its job in this case by calling (recursively) the function *p_expr*, which leaves *tok* at the token following the nested *expr*, which we expect will be a closing parenthesis.

In addition to recognizing the input, we also want to build an abstract syntax tree. We can do this systematically by adding one further convention: each parsing function will return the tree for the construct it has recognized. Following this convention, we can write the following code for *p_factor*:

```
let rec p_factor () =
  match !tok with
      NUMBER v →
        scan (); Number v
    | IDENT x →
```

```
            scan (); Variable x
        | Open →
            scan ();
            let e₁ = p_expr () in
            if !tok = Close then
              scan ()
            else
              failwith "Closing parenthesis expected";
            e₁
        | _ →
            failwith "Syntax error"
```

Writing the function *p_term* is more difficult, for two reasons. First, there are three possibilities, *factor*, *term* TIMES *factor* and *term* DIVIDE *factor*, and we cannot tell which of these applies just by looking at the first token, because if it is IDENT, say, this might belong to any one of them. The second difficulty is that the actions *term* TIMES *factor* ought to begin with a recursive call of *p_term*, so that the first action of *p_term* would be to call itself recursively. This *left recursion* can only lead to an infinite regress, so we must find a way to avoid it.

Since a *term* is essentially a list of *factors* separated by TIMES and DIVIDE operators, we can give an alternative description of its syntax as follows:

$$
\begin{array}{lcl}
\textit{term} & \rightarrow & \textit{factor}\ \textit{termtail} \\[1ex]
\textit{termtail} & \rightarrow & \epsilon \\
 & | & \textsc{Times}\ \textit{factor}\ \textit{termtail} \\
 & | & \textsc{Divide}\ \textit{factor}\ \textit{termtail}
\end{array}
$$

(As usual, ϵ stands for the empty string.) This grammar for terms avoids left recursion, and we can use it to write functions *p_term* and *p_termtail* that do the right thing. For the moment, let's ignore the problem of producing the correct abstract syntax tree, returning the type *unit* instead:

```
let p_term () =
  p_factor (); p_termtail ()

and p_termtail () =
  match !tok with
      Times →
        scan (); p_factor (); p_termtail ()
    | Divide →
        scan (); p_factor (); p_termtail ()
    | _ →
        ();;
```

As you can see, these functions are a direct translation of our alternative grammar for terms. The remaining problem is to construct the correct abstract syntax tree, so that an expression like x/y/z produces the tree

Binop (*Divide*, *Binop* (*Divide*, *x*, *y*), *z*),

equivalent to the reading (x/y)/z and not x/(y/z). A simple way of doing this is to pass as an argument to *p_termtail* the tree for the term that appears to the left, so that *p_termtail* now has the type *expr* → *expr*:

```
let p_term () =
  let e0 = p_factor () in p_termtail e0

and p_termtail e0 =
  match !tok with
      TIMES →
        scan ();
        let e1 = p_factor () in
        p_termtail (Binop (Times, e0, e1))
    | DIVIDE →
        scan ();
        let e1 = p_factor () in
        p_termtail (Binop (Divide, e0, e1))
    | _ →
        e0
```

Expressions are built up from terms using + and – in the same way that terms are built from factors, so the code for *p_expr* is very similar to *p_term*:

```
let p_expr () =
  let e0 = p_term () in p_exprtail e0

and p_exprtail e0 =
  match !tok with
      PLUS →
        scan ();
        let e1 = p_term () in
        p_exprtail (Binop (Plus, e0, e1))
    | MINUS →
        scan ();
        let e1 = p_term () in
        p_exprtail (Binop (Minus, e0, e1))
    | _ →
        e0;;
```

In point of fact, since *p_expr* is called by *p_factor* to deal with the case where a factor is an expression in parentheses, all these functions from *p_factor* to *p_expr* must be defined in one mutually-recursive family. The term 'recursive descent' is well chosen.

Not all grammars make it possible to write a parser using the method of recursive descent: we have seen that left recursion must be avoided, and that where there is more than one alternative production for a nonterminal, the first symbols of the different alternatives must be different so that we can decide between them. Precise mathematical conditions have been worked out for grammars to be parseable by recursive descent, and you can find these conditions described in some of the books that are recommended as Further Reading; the conditions are made slightly complicated by the possibility that non-terminals may generate the empty string.

From a programming point of view, the situation is straightforward: if we succeed in writing a parser, then the grammar is shown to be one that can be parsed by recursive descent. If not, then we must try to re-phrase the grammar in the same way that we re-phrased the grammar for expressions, until we do succeed in writing the parser. Happily, most programming languages have grammars that can be massaged into the right form quite easily; it helps greatly if each kind of statement begins with a distinct keyword, so that the kind of statement can be deduced from its first token.

4.6 Dealing with errors

The parsers we have considered so far do not deal at all well with input programs that contain syntax errors. The best we can hope for is that the parsing process will stop and print a message as soon as it finds a token that cannot appear in a valid input; *yacc* generates parsers that provide at least this minimal degree of error-handling. However, for a practically useful parser, we would want to improve this behaviour in two ways:

(1) When an error is found, a more informative error message would be helpful. For example, if an expression contains more left parentheses than right parentheses, a message that says so might be better than just 'syntax error'.

(2) After an error, it would be useful if the parser could go on to find other syntax errors in the input, instead of just stopping immediately.

Although these two features are nice to have, it is not always clear exactly what should be done. On one hand, an extensive theory of error recovery and repair has been developed, including algorithms that, for certain classes of grammars, find the smallest number of tokens that can be added to the input or deleted from it to give a valid program. Some of these methods are presented in the book by Backhouse recommended as Further Reading. On the other hand, a parser that produces clear and helpful error messages usually cannot be based on these automatic techniques alone, because helpful error messages will depend on the kinds of errors that are common in the language being parsed.

Exercises

4.1 Write a context-free grammar that describes the set $\{ a^n b^n \mid n \geq 0 \}$.

4.2 The original ALGOL-60 report included the following description of the syntax of if statements:

$$\begin{array}{lll} stmt & \rightarrow & basic\text{-}stmt \\ & \mid & \mathsf{if}\ expr\ \mathsf{then}\ stmt \\ & \mid & \mathsf{if}\ expr\ \mathsf{then}\ stmt\ \mathsf{else}\ stmt. \end{array}$$

Show that this grammar is ambiguous. Propose another description that generates the same set of programs but unambiguously associates each else with the closest possible if–then. For the ambiguous grammar, show a configuration of a shift-reduce parser where both *shift* and *reduce* are possible actions. Should *shift* or *reduce* be chosen if a parser based on this grammar is to be faithful to the unambiguous grammar's interpretation?

4.3 In PICOPASCAL, if statements have an explicit terminator end that removes the ambiguity discussed in the preceding exercise. However, this makes it cumbersome to write a chain of if tests, since the end keyword must be repeated once for each if. Show how to change the parser shown in Figures 4.6 and 4.7 to allow the abbreviation suggested on page 51, where an arbitrarily long chain of tests written with the keyword elsif can have a single end. Arrange for the parser to build the same abstract syntax tree for the abbreviated program as it would for its equivalent written without elsif.

4.4 Show that the following grammar for expressions is unambiguous:

expr	→	*term* | *expr* "+" *term*
term	→	*factor* | *term* "$*$" *factor*
factor	→	"x" | "y" | "(" *expr* ")"

[Note the extra rule that allows parentheses.]

4.5 Suppose S is the set of strings described by a regular expression R. Show that there is a context-free grammar that generates exactly the strings in S. [Hint: use structural induction on the form of R.]

4.6 In the recursive-descent parser for expressions in Section 4.5, why is it not possible simply to recast the grammar so that left recursion is replaced by right recursion as follows?

term	→	*factor*
	|	*factor* TIMES *term*
	|	*factor* DIVIDE *term*

4.7 In the recursive-descent parser for expressions, the tree for a term was constructed by making *p_termtail* take as an argument the term to its left, so that it had the type *expr* → *expr*. Show how to redefine *p_termtail* so that it returns a list of operators and factors and has type *unit* → (*op* $*$ *expr*) *list*, and redefine *p_term* to produce the correct tree from this.

Chapter 5

Expressions and statements

We now know how to take a program text and turn it into an abstract syntax tree. From now on, therefore, all we have to say about programs can be said in the language of abstract syntax. In Section 2.2, we already defined a function that evaluates expressions, and in Lab 1 we put it together with a small parser to give a “pocket calculator” program that accepts an expression and prints its value. We now extend that function to give an interpreter for a language of expressions and statements that works directly with the abstract syntax tree. We will call this a *source-level interpreter*, like most of the interpreters we will build, because it uses an internal representation of the program being executed – an abstract syntax tree – that is very close to the source text. We shall use the abstract syntax shown in Figure 4.5. Later in this chapter, I shall show how to translate this language into code for a stack-based abstract machine.

5.1 Source-level interpreters

Our first interpreter will be a pure functional program; although the language it implements will have modifiable variables and assignment, we will not use these features of ML in writing the interpreter. This will make our interpreter very inefficient as a way of actually running programs; we shall take the opportunity to examine a more efficient interpreter in Lab 4. On the other hand, the purely functional interpreter captures the meaning of programs in a particularly simple way, so that we'll be able to *prove* that a compiler for at least the sub-language of expressions works correctly.

The first step is to modify the function *eval* so that it takes an explicit state as an argument. Initially, we can think of the state as a simple function from identifiers to values, and we'll suppose that integers are the only values that can be expressed in our programming language. In addition to the initial state *init*, where all variables are undefined, we'll need two operations on states:

```
val init : state;;
val fetch : ident → state → int;;
val update : ident → int → state → state;;
```

The function *fetch* finds the value of a variable in a given state, and the function *update* is defined so that *update* $x\ v\ s$ is a state that is identical to s, except that variable x has value v. We do not care how *fetch* and *update* and implemented, except that we expect them to have certain sensible properties, such as the fact that *update* changes the value of exactly one variable:

$$\mathit{fetch}\ x\ (\mathit{update}\ y\ v\ s) = \textbf{if}\ x = y\ \textbf{then}\ v\ \textbf{else}\ \mathit{fetch}\ x\ s.$$

We can use *fetch* to write a new definition of *eval* that deals with variables, and incidentally with unary operators too:

```
let rec eval s =
  function
      Number n → n
    | Variable x → fetch x s
    | Monop (w, e_1) → do_monop w (eval s e_1)
    | Binop (w, e_1, e_2) → do_binop w (eval s e_1) (eval s e_2);;
```

The next step is to define a function

$$\mathit{exec} : \mathit{state} \to \mathit{stmt} \to \mathit{state}$$

that maps the state before executing a statement to the state afterwards:

```
let rec exec s =
  function
      Assign (x, e) →
        update x (eval s e) s
    | Seq stmts →
        fold_left exec s stmts
    | IfStmt (test, thenpt, elsept) →
        if eval s test ≠ 0 then
          exec s thenpt
        else
          exec s elsept
    | WhileStmt (test, body) →
        if eval s test ≠ 0 then
          let s' = exec s body in
          exec s' (WhileStmt (test, body))
        else
          s
```

The state after an assignment $x := e$ is the same as the state before it, except that the value of x after the assignment has changed to be the value of e before the assignment. The effect of an if statement is the same as either the effect of the then part or the effect of the else part, according to whether the test is true or false. The effect of a while statement is to leave the state unchanged of the test is initially false, or if the test is true, the effect is to execute the body and then execute the whole while statement again. Finally, executing a sequence of statements *Seq ss* is the same as executing each statement in turn. We use *fold_left*, so that the final state returned by one statement in the sequence becomes the initial state for the next one. Thus

$$\mathit{exec}\ s_0\ (\mathit{Seq}\ [p; q; r]) = \mathit{exec}\ (\mathit{exec}\ (\mathit{exec}\ s_0\ p)\ q)\ r.$$

It's important to notice that the definition of *exec* does not use *structural* recursion on the syntax of the source program, that is, it does not follow the common pattern where the value of a function on each kind of tree is defined in terms of its value on the proper parts of that tree. This pattern is violated in the rule for executing a while statement, which involves not only the proper parts of the statement – the test and the body – but also the whole statement itself. On reflection, we can see that this must be the case, because any functional program that uses only structural recursion necessarily terminates for all values of its argument, and on the other hand, we know that there are programs written in the language of if's and while's that do not terminate. When our interpreter runs such a program, the interpreter itself does not terminate; so the interpreter could not be written using structural recursion alone.

This method of building interpreters that represent the source program as a tree becomes practical is we adopt a more efficient way of representing states, for example by using the reference cells of ML. Assignments can then be implemented by destructively updating these cells; also, control structures like while in the source language can be interpreted directly in terms of the same control structures in the meta-language. Although source-level interpreters like this can never approach the efficiency of compiled code, they provide a good way to implement all sorts of little languages: evidence for this is provided by the success of languages like AWK, PERL, the UNIX shells, and so on, many of which are implemented using this kind of technique.

5.2 Compiling expressions

The function *exec* we defined in the preceding section is an *interpreter*, because it realizes directly the meaning of the source program. We'll now begin to move towards a *compiler*, which generates instructions that another interpreter can follow, rather than executing the program directly. In the first part of this book, the compilers we'll build will generate code for an invented machine, one that makes compiling especially easy.

This isn't just to make the course easier to follow, but is a reasonable way of building a real compiler. The idea is to generate code for an invented, 'abstract' machine in the first part of the compiler, and to have a translator as the last part that generates the instructions for a real machine that correspond to the abstract machine code. This makes the compiler easier to understand by separating concerns about the meaning of constructs in the source language from concerns about the architecture of the target machine.

Another advantage is that the compiler becomes easier to move from one target machine to another, because the only part that needs to be changed is the translation from the abstract machine code to the real object code, and the part of the compiler that produces the abstract machine code is common to all target machines.

A third advantage of using abstract machine code is that we can get away without generating any real machine code at all, by building a *simulator* for the abstract machine code, that is, a low-level interpreter that simulates the actions of the abstract machine when it executes a particular program. Such a simulator will only run programs at a fraction of the speed that could be achieved by translating them into real machine code, but it allows the programming language to be implemented in a way that is almost completely portable from one machine to another. The compiler that you'll work with in Labs 2, 4 and 6 is built in this way.

Our first step is to design an abstract machine and a compiler for evaluating expressions. The abstract machine has a stack for holding the values of sub-expressions, and its instructions are defined by the following ML data type:

```
type inst =
    Const of int                (* Load constant (value) *)
  | Load of ident               (* Load value (name) *)
  | Monop of op                 (* Perform unary operation (op) *)
  | Binop of op                 (* Perform binary operation (op) *)
  | ...
```

The '...' here shows where we will add more instructions later, on page 67 and elsewhere throughout the book. The meaning of these instructions is specified by functions

```
val exec_inst : state → stack → inst → stack;;

val run : state → stack → inst list → stack;;
```

that are defined as follows:

```
let exec_inst s r i =
  match (i, r) with
      (Const n, _) → n :: r
    | (Load x, _) → fetch x s :: r
    | (Monop w, a :: r′) → do_monop w a :: r′
    | (Binop w, b :: a :: r′) → do_binop w a b :: r′
    | (_, _) → failwith "stack underflow";;

let run s r code = fold_left (exec_inst s) r code;;
```

These functions take a *state* argument that gives the values that are used in *Load* instructions. Because (so far) we are only dealing with expressions that are free of side-effects, this state cannot change during execution of the code.

The other part of our task is to design a compiling function

```
val gen_expr : expr → inst list;;
```

that generates the sequence of instructions needed to evaluate an expression and leave its value on the stack:

```
let rec gen_expr =
  function
      Number n → [Const n]
    | Variable x → [Load x]
    | Monop (w, e1) → gen_expr e1 @ [Monop w]
    | Binop (w, e1, e2) → gen_expr e1 @ gen_expr e2 @ [Binop w];;
```

For example, let us consider in detail how the expression "x+5∗y" would be compiled and executed. From this input expression the lexical analyser produces the token stream

```
Ident "x", Addop Plus, Number 5, Mulop Times, Ident "y"
```

The parser then produces the tree

Binop (*Plus*, *Variable* "x", *Binop* (*Times*, *Number* 5, *Variable* "y"))

Next, *gen_expr* compiles this into the following sequence of instructions:

LOAD "x"
CONST 5
LOAD "y"
BINOP Times
BINOP Plus,

Because this program is obtained by compiling an expression of the form *Binop* (*Plus*, e_1, e_2), it consists of programs for e_1 = *Variable* "x" and e_2 = *Binop* (*Times*, *Number* 5, *Variable* "y"), followed by the instruction *BINOP Plus*. Those smaller programs are themselves formed by translating e_1 and e_2 in a recursive process; e_1 generates just the single instruction *LOAD* "x", and e_2 requires instance of the pattern involving *BINOP*.

Finally, executing this sequence of instructions will leave the value of the expression on top of the stack. If we use a state where x has value 3 and y has value 4, then the execution goes like this:

Instruction	Stack (top on left)
LOAD "x"	3
CONST 5	5 3
LOAD "y"	4 5 3
BINOP Times	20 3
BINOP Plus	23

For a compiler as simple as this, it is possible to prove mathematically that the code generated is correct. We have precise specifications of the source language and the target machine in the functions *eval* and *run* respectively, and we need to prove for any expression *e* and state *s* that

$$run\ s\ [\,]\ (gen\_expr\ e) = [eval\ s\ e].$$

The proof is a structural induction on the expression *e*. What we actually prove is a slightly more general property of the compiler called the *net effect property*:

$$run\ s\ r\ (gen\_expr\ e) = eval\ s\ e :: r.$$

This says that the net effect of running the code for an expression *e* is to push the value of *e* on top of whatever was on the stack beforehand.

We consider first the case where *e* is a constant *Number n*:

$$\begin{aligned} &run\ s\ r\ (gen\_expr\ (Number\ n)) \\ &\quad = run\ s\ r\ [\textit{CONST}\ n] \\ &\quad = exec\_inst\ s\ r\ (\textit{CONST}\ n) \\ &\quad = n :: r \\ &\quad = eval\ s\ (Number\ n) :: r \end{aligned}$$

Similarly, if *e* is a variable *Variable v*, then the value that gets pushed is *fetch v s*.

If e has the form $Binop\,(w, e_1, e_2)$, then we may assume as induction hypotheses that the net effect property holds of e_1 and e_2. We then calculate

$$\begin{aligned}
&run\ s\ r\ (gen\_expr\ (Binop\,(w, e_1, e_2))\\
&\quad = run\ s\ r\ (gen\_expr\ e_1\ @\ gen\_expr\ e_2\ @\ [\textsc{Binop}\ w])\\
&\quad = run\ s\ (run\ s\ (run\ s\ r\ (gen\_expr\ e_1))\ (gen\_expr\ e_2))\ [\textsc{Binop}\ w]\\
&\quad = run\ s\ (run\ s\ (eval\ s\ e_1 :: r)\ (gen\_expr\ e_2))\ [\textsc{Binop}\ w]\\
&\quad = run\ s\ (eval\ s\ e_2 :: eval\ s\ e_1 :: r)\ [\textsc{Binop}\ w]\\
&\quad = exec\_inst\ (eval\ s\ e_2 :: eval\ s\ e_1 :: r)\ (\textsc{Binop}\ w)\\
&\quad = do\_binop\ w\ (eval\ s\ e_1)\ (eval\ s\ e_2) :: r\\
&\quad = eval\ s\ (Binop\,(w, e_1, e_2)) :: r.
\end{aligned}$$

This proof depends on the following property of *run*, which it enjoys because of its definition in terms on *fold_left*:

$$run\ s\ r\ (c_1\ @\ c_2) = run\ s\ (run\ s\ r\ c_1)\ c_2.$$

The case where e is $Monop\,(w, e_1)$ is similar, and that completes the proof.

Although correctness of the code generated by compilers is very important, that's the last time we shall actually *prove* that our code is right.

5.3 Compiling statements

To implement assignment statements and control structures like if and while, we'll need to add instructions to the abstract machine that can change the state, and instructions that can cause other sequences of instructions to be executed repeatedly or not at all. We extend the type *inst* as follows:

```
type inst = ...
  | Store of ident              (* Store (name) *)
  | Jump of codelab             (* Unconditional branch (dest) *)
  | Jumpb of bool * codelab     (* Branch on boolean (val, dest) *)
  | Label of codelab            (* Set code label *)
  | Stop                        (* End of program *)
  | ...
```

A *Label* instruction has no effect itself, but provides a target for *Jump* and *Jumpb* instructions. *Jump* instructions always transfer control to the label, but an instruction *Jumpb* (b, lab) pops a Boolean value off the stack transfer control only if the value matches b. We'll represent the Boolean value *true* as the integer 1, and *false* by the integer 0.

The new compiling function for statements will have the type

```
val gen_stmt : stmt → inst list;;
```

To compile an assignment statement $v := e$, we should generate code as follows:

```
⟨Code for e⟩
Store v
```

This works because the code for e will leave the value of e on the stack; the *STORE v* instruction pops this value and stores it as a new value of v. We can therefore begin our definition of *gen_stmt* like this:

```
let rec gen_stmt =
  function
      Assign (v, e) → gen_expr e @ [STORE v]
```

We can compile the if statement *IfStmt* (*test*, *thenpt*, *elsept*) into the following code:

```
⟨Code for test⟩
JUMPB (false, lab₁)
⟨Code for thenpt⟩
JUMP lab₂
LABEL lab₁
⟨Code for elsept⟩
LABEL lab₂
```

This code, like all the code sequences we shall generate for statements, has a single entry at the top and a single exit at the bottom. The code first evaluates the test in the if statement, leaving a Boolean value on top of the stack. The *JUMPB* instruction pops this value, and if it is *false* jumps to label lab_1, where the code for the else part of the statement is found. Otherwise, the code for the then part is executed, followed by a jump to label lab_2, where the then and else parts join.

To generate code like this, we need to invent fresh names for the labels lab_1 and lab_2, for we can't use the same names in every if statement without causing confusion. Let's suppose that there is an (impure) function *label* : *unit* → *codelabel* that returns a different label each time it is called. Then the code we have sketched can be generated as follows:

```
let rec gen_stmt =
  function . . .
    | IfStmt (test, thenpt, elsept) →
        let lab₁ = label ( ) and lab₂ = label ( ) in
        gen_expr test @ [JUMPB (false, lab₁)] @ gen_stmt thenpt
          @ [JUMP lab₂; LABEL lab₁] @ gen_stmt elsept @ [LABEL lab₂]
    | . . .
```

Similar techniques allow us to generate code for the while loop; the code for the statement *WhileStmt* (*test*, *body*) follows the scheme

```
LABEL lab₁
⟨Code for test⟩
JUMPB (false, lab₂)
⟨Code for body⟩
JUMP lab₁
LABEL lab₂
```

Here is the rule in *gen_stmt* that generates this code:

```
let rec gen_stmt =
  function . . .
    | WhileStmt (test, body) →
```

```
        let lab_1 = label ( ) and lab_2 = label ( ) in
        [LABEL lab_1] @ gen_expr test @ [JUMPB (false, lab_2)]
          @ gen_stmt body @ [JUMP lab_1; LABEL lab_2]
    | ...
```

This way of translating while statements is appealing, because the code for the test and body appears in the same order as the test and body appear in the source program. However, as Exercise 5.7 shows, it is better to reverse the order of the two parts.

Although it's neat to put together the object code by concatenating lists of instructions, it gets a bit cumbersome if the compiler becomes much more complicated, and it can have a bad effect on the efficiency of the compiler, because of all the long lists of instructions that have to be concatenated.[1] So we'll adopt instead an imperative approach, where the module that's responsible for outputting the code provides a function

```
val gen : inst → unit;;
```

that adds one instruction to the growing object program. In this style, the code for while is generated by the following statements:

```
let lab_1 = label ( ) and lab_2 = label ( ) in
gen (LABEL lab_1);
gen_expr test;
gen (JUMPB (false, lab_2));
gen_stmts body;
gen (JUMP lab_1);
gen (LABEL lab_2)
```

The other part of our task in implementing statements is to describe what the new instructions mean. Because we've added labels and jumps to our machine, it makes sense to make explicit at this point the idea that the object program is stored in a memory whose cells have addresses. We can then explain jump instructions as setting a program counter that holds the address of the next instruction to be executed.

So we declare a global array *prog* that will contain the instructions that are generated by the compiler, and a hash table *labdict* that maps labels to the addresses they denote. A variable *loc* contains the address of the next instruction to be generated.[2]

```
let memsize = 1000;;

let prog = Array.create memsize STOP;;     (* Program memory *)
let labdict = Hash.create 100;;     (* Dictionary of labels *)
let loc = ref 0;;                   (* Current assembly location *)
```

[1] An alternative approach for efficiency would be to build a nested list structure that is only flattened at the end; the flattening can be done in $O(N)$ time rather than $O(N^2)$, where N is the number of instructions. We'll do something similar in Chapter [c:regvars], where our translation functions will produce trees rather than lists of instructions.

[2] For simplicity, I've given the *prog* array a fixed size of 1000 instructions. It would be better to allow the array to grow if necessary, so as not to limit the size of program that can be compiled.

The function *gen* usually adds an instruction to the program, unless the instruction is a label, in which case it just defines the label as the current value of *loc*.

```
let gen =
  function
      LABEL lab → Hash.add lab !loc labdict
    | i → prog.(!loc) ← i; incr loc;;
```

Figure 5.1 shows a simulator that works with this representation of the program. This simulator also makes it explicit that there is never any need to store more than one copy of the state and stack contents, by using global variables for these two components of the machine.

Jump instructions (*JUMP lab* and *JUMPB* (*b*, *lab*) are handled by looking up the label *lab* in the hash table *labdict* each time the instruction is executed. This costs a little time, which could be saved by changing the way these instructions are represented, so that the label was replaced by its value before execution started. This is (of course) what happens in real machines, where the assembler and loader cooperate to replace labels in an assembly language program by numeric addresses before the program begins execution.

5.4 Jumping code for conditions

Our compiler so far implements scalar variables with assignment and the control structures if–then–else and while–do. Let's think about the way the tests in the control structures are evaluated, and particularly about what would happen if they included and and or operators.

As things stand, we could easily add these operators to the language, and generate postfix code that evaluates *p* and *q*, for example, by evaluating each of *p* and *q* to Boolean values, then combining the two values. This works well in simple cases, but it is inefficient to evaluate *q* in this expression if we have already evaluated *p* to *false*. What's more, it is more convenient to allow code like this:

```
i := 0;
while i < 10 and a[i] <> x do
  i := i+1
end
```

This code works only if we may assume that the second operand won't be evaluated if the first one is false. In this example, if the array a has 10 elements numbered from 0 to 9, the test will cause an array bound error for i = 10 unless the and operator is evaluated by the 'short-circuit' rule. (We haven't intruduced arrays into our language yet, but they are coming soon.)

We have to look at the definition of the programming language we are implementing to find out whether it is acceptable to generate code for and and or that ignores the second operand in some cases. It matters, because the second operand might be a function call with side-effects, and a programmer might be entitled (according to the language designer) to expect those side-effects to happen. PASCAL, for example, demands that both arguments are evaluated, but MODULA-2 and C (with the operators && and ||) demand short-circuit evaluation.

```
let pc = ref 0;;                    (∗ Program counter ∗)
let stack = ref ([ ] : int list);;  (∗ Stack ∗)
let state = ref init;;              (∗ Data memory ∗)

(∗ push, pop – push or pop the expression stack ∗)
let push v = stack := v :: !stack;;
let pop ( ) = let v = hd (!stack) in stack := tl !stack; v;;

exception Finish;;                  (∗ Raised by STOP instuction ∗)

(∗ find_lab – find location of label ∗)
let find_lab lab = Hash.find lab labdict;;

(∗ exec_inst – execute an instruction ∗)
let exec_inst =
  function
      CONST n → push n
    | LOAD x → push (fetch x !state)
    | STORE x → state := update x (pop ( )) !state
    | MONOP w → push (do_monop w (pop ( )))
    | BINOP w → let b = pop ( ) in let a = pop ( ) in
                  push (do_binop w a b)
    | JUMP l → pc := find_lab l
    | JUMPB (b, l) → if b = (pop ( ) ≠ 0) then pc := find_lab l
    | STOP → raise Finish
    | _ → failwith "bad instruction";;

(∗ run – execute the compiled program ∗)
let run ( ) =
  pc := 0; stack := [ ];
  try
    while true do
      let i = prog.(!pc) in
      incr pc; exec_inst i
    done
  with Finish → ( );;
```

Figure 5.1: *Instruction interpreter*

Let's suppose short-circuit evaluation is allowed (or demanded) by the language design. How are we going to implement it? One solution is to design a function *gen_cond* such that the call *gen_cond sense lab e*, where *sense* is *true* or *false*, generates code that jumps to label *lab* if the Boolean expression *e* has value *sense*. Thus the test of a `while` loop (which should jump beyond the end of the loop if the test is *false*) can be generated by the call *gen_cond false lab test*. The whole loop is compiled by the following rule in the definition of *gen_stmt*:

```
let rec gen_stmt =
  function ...
    | WhileStmt (test, body) →
        let $lab_1$ = label ( ) and $lab_2$ = label ( ) in
        gen (LABEL $lab_1$);
        gen_cond false $lab_2$ test;
        gen_stmt body;
        gen (JUMP $lab_1$);
        gen (LABEL $lab_2$)
    | ...
```

Similarly, `if` statements can be compiled by using *gen_cond* to generate code that conditionally jumps to the `else` part if the test is false.

Now we need to implement the function *gen_cond*. For expressions that do not involve short-circuit evaluation, we simply evaluate the expression and then use a *JUMPB* instruction:

```
and gen_cond sense lab =
  function
      Monop (Not, e) →
        gen_cond (not sense) lab e
    | Binop (And, $e_1$, $e_2$) →
        if sense then begin
          let $lab_1$ = label ( ) in
          gen_cond false $lab_1$ $e_1$;
          gen_cond true lab $e_2$;
          gen (LABEL $lab_1$)
        end
        else begin
          gen_cond false lab $e_1$;
          gen_cond false lab $e_2$
        end
    | Binop (Or, $e_1$, $e_2$) →
        if sense then begin
          gen_cond true lab $e_1$;
          gen_cond true lab $e_2$
        end
        else begin
          let $lab_1$ = label ( ) in
          gen_cond true $lab_1$ $e_1$;
          gen_cond false lab $e_2$;
          gen (LABEL $lab_1$)
        end
```

```
| e →
    gen_expr e;
    gen (JUMPB (sense, lab));;
  ...
| e →
    gen_expr e;
    gen (JUMPB (sense, lab));;
```

By making this 'catch-all' rule the last one, we ensure that expressions that do need special treatment are handled by one of the rules we are about to examine. Next, we can consider what code should be generated by

gen_cond true lab e

where *e* represents the expression *p* and *q*. This code should have the effect of jumping to *lab* if the expression *e* is true, so the following scheme does the job:

```
⟨Jump to lab₁ if p is false⟩
⟨Jump to lab if q is true⟩
LABEL lab₁
```

The two sub-sequences shown in angle brackets are both generated by recursive calls of *gen_cond*. If *p* is false, this code branches directly to lab_1 without evaluating *q*; if *p* is true, then execution will reach the code for *q*, and if *q* is true also, then a jump to label *lab* will be taken.

With the same expression *p* and *q* but *sense* = *false*, we want code that jumps to *lab* if either *p* or *q* is false. This effect is acheived by the following scheme:

```
⟨Jump to lab if p is false⟩
⟨Jump to lab if q is false⟩
```

Again, if *p* is false, the code jumps to *lab* immediately, without executing the code code for *q*. If *p* is true, then the code for *q* is executed, and results in a jump the label *lab* unless *q* is also true.

These two code schemes for and are used in the version of *gen_cond* shown in Figure 5.2. The figure also shows the generation of jumping code from expressions of the form *p* or *q*; the same code patterns are very similar, but with *true* swapped with *false* throughout, according to de Morgan's laws. The not operator can be treated simply by reversing the value of the *sense* parameter.

By using *gen_cond* in all the places where control structures use a Boolean condition, we can implement short-circuit evaluation of nearly every boolean expression. The only other place where a Boolean expression could occur is on the right hand-side of an assignment, as in

```
done := (k < n) and (a[k] = x)
```

We can arrange for such expressions also to receive short-circuit evaluation by adding a clause to *gen_expr* that recognizes them and uses *gen_cond*, pushing either the constant 1 or the constant 0 as the result:

```
let rec gen_expr e =
  match e with
      ...
```

```
and gen_cond sense lab =
  function
      Monop (Not, e) →
        gen_cond (not sense) lab e
    | Binop (And, $e_1$, $e_2$) →
        if sense then begin
          let $lab_1$ = label ( ) in
          gen_cond false $lab_1$ $e_1$;
          gen_cond true lab $e_2$;
          gen (LABEL $lab_1$)
        end
        else begin
          gen_cond false lab $e_1$;
          gen_cond false lab $e_2$
        end
    | Binop (Or, $e_1$, $e_2$) →
        if sense then begin
          gen_cond true lab $e_1$;
          gen_cond true lab $e_2$
        end
        else begin
          let $lab_1$ = label ( ) in
          gen_cond true $lab_1$ $e_1$;
          gen_cond false lab $e_2$;
          gen (LABEL $lab_1$)
        end
    | e →
        gen_expr e;
        gen (JUMPB (sense, lab));;
```

Figure 5.2: *Implementation of gen_cond*

```
    | Binop ((And | Or), _, _) →
        let $lab_1$ = label ( ) and $lab_2$ = label ( ) in
        gen_cond false $lab_1$ e;
        gen (CONST 1);
        gen (JUMP $lab_2$);
        gen (LABEL $lab_1$);
        gen (CONST 0);
        gen (LABEL $lab_2$)
      . . .
```

The pattern in this rule matches all *Binop* expressions where the operator is either *And* or *Or*.

5.5 Compiling programs

Now that we can compile expressions and statements, compiling programs written in the language of Figure 4.5 is easy enough: we just have to compile the sequence of statements that makes up the program body:

```
let compile (Program body) =
  gen_stmt body;
  gen STOP;;
```

This generates code for the statements in the program, beginning at address zero. The code is followed by a *STOP* instruction, so that it is ready to run by setting the program counter to zero and beginning to execute instructions. This completes our very first compiler!

Exercises

In these exercises, assume that the abstract syntax of expressions is as follows:

```
type expr =
    Number of int
  | Variable of ident
  | Binop of op * expr * expr;;
```

5.1 Define a function

$$subst : ident \to expr \to expr \to expr$$

so that $subst\; x\; e'\; e$ is the result of substituting e' for the variable x throughout the expression e.

Now consider the state-based interpreter of Section 5.1. Prove by structural induction that for all states s,

$$eval\; s\; (subst\; x\; e'\; e) = eval\; (update\; x\; (eval\; s\; e')\; s)\; e$$

Deduce that the two programs $x := e';\, x := e$ and $x := e''$ are equivalent, where $e'' = subst\; x\; e'\; e$.

5.2 Some machines have an expression stack implemented in hardware, but with a finite limit on its depth. For these machines, it is important to generate postfix code that makes the maximum stack depth reached during execution as small as possible.

(a) Define a function $max\_depth : inst\; list \to int$ (where *inst* is the type of instructions defined in Section 5.2) that gives the maximum stack depth needed to execute a list of instructions.

(b) Let the *SWAP* instruction be defined so that it swaps the two top elements of the stack. Show how to use this instruction to evaluate the expression 1/(1+x) without ever having more than two items on the stack.

(c) Prove that if expression e_1 can be evaluated in depth d_1, and e_2 can be evaluated in depth d_2, then $Binop\,(w, e_1, e_2)$ can be evaluated in depth

$$min\,(max\; d_1\; (d_2 + 1))\,(max\; (d_1 + 1)\; d_2).$$

Write a function *cost* : *expr* → *int* that calculates the stack depth that is needed to evaluate an expression by this method. Show that if *e* has fewer than 2^N operands, then *cost e* ≤ *N*.

(d) Write an expression compiler *gen_expr* : *expr* → *inst list* that generates the code that evaluates an expression *e* within stack depth *cost e*. [Hint: use *cost* in your definition.]

5.3 Now consider a machine that has a finite stack of depth *N*. In order to make it possible to evaluate expressions of arbitrary size, the machine is also supplied with a large collection of temporary storage locations numbered 0, 1, 2, ... There are two additional machine instructions:

```
type inst = ...
  | PUT of int              (∗ Save temp (address) ∗)
  | GET of int              (∗ Fetch temp (address) ∗)
```

The instruction *PUT n* pops a value from the stack and stores it in temporary location *n*, and the instruction *GET n* fetches the value previously stored in temporary location *n* and pushes it on the stack.

Assuming *N* ≥ 2, define a new version of *gen_expr* that exploits these new instructions, and places no limit on the size of expressions. The code generated should use as few *GET* and *PUT* instructions as possible, but you may ignore the possibility that the source expression contains repeated subexpressions.

[*Hint*: optimal code for an expression consists of a phase during which sub-expressions are evaluated and saved in temporaries, followed by a phase where the expression itself is evaluated on the stack. Define by recursion a function

code : *expr* → (*inst list* ∗ *inst list* ∗ *int*)

such that if *code e* = (*prep*, *eval*, *depth*) then *prep* is code for the preparation phase, *eval* is code for the final evaluation, and *depth* is the stack depth needed to execute *eval*.]

5.4 A simple language of expressions contains just variables and binary operators. Its abstract syntax is as follows:

```
type expr =
    Variable of ident         (∗ Variable (name) ∗)
  | Binop of op ∗ expr ∗ expr  (∗ Binary operation ∗)
```

These expressions must be translated into code for a machine that has a single accumulator register *A*, but a large collection of addressible temporary storage cells. The machine has the following instructions:

```
type inst =
    LOAD of addr            (∗ Load from address into A ∗)
  | STORE of addr           (∗ Store from A into address ∗)
  | BINOP of op ∗ addr      (∗ Perform binary operation ∗)
```

Each address (type *addr*) is either the name of a variable or the number of a temporary cell:

```
type addr =
    Var of ident            (∗ Address of variable (name) ∗)
  | Temp of temp            (∗ Temporary cell ∗)
```

Thus, the instruction *LOAD* (*Var x*) loads the value of variable x into A, the instruction *STORE* (*Temp t*) stores the value in A into temporary cell t, and the instruction *BINOP* (*Minus*, *Var y*) subtracts the value of variable y from the contents of A and puts the result back in A. Each instruction has the same cost in space and execution time.

The object program is generated by calling a function $gen : inst \to unit$ that adds one instruction to the program. Fresh temporary locations $t_1, t_2, \ldots$ may be allocated by calling the function $alloc\_temp : unit \to temp$; there is no need to economize by reusing these locations.

(a) Write down code that evaluates the expression `x - (y / z)`, leaving the result in the accumulator.

(b) Show, by giving the code for an example expression, that better code can be produced by taking into account the commutativity of operators like + and $*$.

(c) Define a pair of functions $trans\_acc : expr \to unit$ and $trans\_mem : expr \to addr$ that together generate code for an arbitrary expression. The function *trans_acc* should leave the value of the expression in the accumulator A, and *trans_mem* should leave the value in memory, returning its address. At this stage, you need not exploit commutativity of operators.

(d) Show how to modify your definitions so as to generate code using the best evaluation order. You may assume a test $commutatative : op \to bool$ that identifies commutative operators. You need not consider optimizations that might result from repeated sub-expressions or associative properties of operators.

Hint: define mutually recursive functions *cost_acc e* and *cost_mem e* that calculate respectively the cost of computing the value of expression *e* and leaving the result in the accumulator or in memory.

5.5 A *primitive recovery block* has the following syntax:

try *stmt*; . . . ; *stmt* ensuring *expr*

Executing it has the following effect: first the statements in the body are executed; then the final expression is evaluated in the state that results. If the expression evaluates to *true* then this state is also the final state that results from executing the recovery block; otherwise, the state is reset to what it was before execution of the recovery block started.

Show how to implement primitive recovery blocks in a source-level interpreter with explicit state. What feature of the implementation corresponds to the fact that they cannot be implemented without making multiple copies of the state?

5.6 The function *gen_cond* defined in Figure 5.2 does not treat constant expressions specially, but generates code for them using the default rule. Devise an improvement to *gen_cond* that generates the best code for them.

What code does your improved version of *gen_cond* generate for the condition `true or (x = y)`, where `true` is represented by the constant *Number* 1? Can it be improved still further?

5.7 The code we have been generating for while statements puts code for the test first, and code for the body afterwards. An alternative translation reverses the order: the code for the test is put after the code for the body, and jumps back to the body if another iteration is needed. The whole loop begins with an unconditional branch to the test. Sketch the layout of code that follows this translation scheme, and show how to generate this code in *gen_stmt*, using *gen_cond* as appropriate. By counting the number of jump instructions executed in the program

```
i := 0; while i < n do i := i+1 end
```

explain why this code is better than that suggested in the text.

Chapter 6

Types and data structures

So far, we have seen how to implement lexical and syntactic analysis, so that a source program can be transformed into an abstract syntax tree, convenient for processing by recursive tree-walking algorithms. We have looked at the implementation of statements and expressions in a compiler for a simple language, with global integer variables and no subroutines. Our compiler translates programs into code for an abstract stack machine, and we have seen how to build a simulator for this machine – that is, an interpreter for the abstract machine code.

There are many programming language features that are not covered by the simple compilers and interpreters we have designed so far. Here are some of the features that we would like to support in our compiler:

- Support for data types such as arrays and records, in addition to simple integers.
- A subroutine mechanism, so that a fragment of program that performs a certain task can be named and used from many places in a larger program.
- Parameters for subroutines, so that subroutines can be developed for a general task, and used to perform different instances of the task each time they are called.
- Local variables, so that unwanted interference between subroutines can be avoided.

Also, our model of the target machine is unrealistic, in that its state is a direct mapping from source-language identifiers to values, and real machines have memory that is addressed by numeric addresses instead of symbolic names. Translating a program so that references to variables are replaced by numeric addresses is part of the job of the compiler.

The first step in implementing these programming language features and generating code for a more realistic machine is to develop a semantic analysis phase for the compiler. This phase builds up a *symbol table* or *dictionary* from the declarations that appear in the source program, and uses it to annotate each use of a variable with information about its type and its address at run time; it also checks the program for errors, such as type mismatches and undeclared variables, that are not detected during the syntax analysis phase.

This chapter begins by showing how to implement a semantic analyser for a new language that requires variables to be declared, and allows them to have different types, in addition to having the features we studied in Chapter 5. We begin with the scalar types integer and boolean, and later add arrays and records formed from these types and other array and record types. The semantic analyser calculates the type of each variable and its run-time address, and adds this information as annotations to the abstract syntax tree.

Later in the chapter, we shall see how to use the annotations to generate code for a machine that has a data store accessed by numeric addresses. Each variable will be accessed at the address that was allocated by the semantic analyser, and code to access an element of an array or record will involve calculating the address of the element.

The language treated in this chapter still has no procedures and no local variables, but the semantic analysis techniques we develop here will continue to be useful when we add those features later. The compiler architecture described in this chapter is based on the compiler that you will be working with in Lab 4.

6.1 Annotating the tree

One of the jobs of semantic analysis is to associate each place where an identifier is used with information about its declaration. This information allows the code generator to generate the right code to access the value of the identifier. In our compiler, we will represent the information about an identifier as a record with the following type *def*:

```
type def =
  { d_tag : ident;              (* Name *)
    d_type : ptype;             (* Type *)
    d_addr : int };;            (* Address *)
```

Each definition contains a variable name *d_tag* that says which identifier it is defining. There is also its type *d_type* and its run-time address *d_addr*. Initially, we will support a small collection of scalar types like integer and boolean: each of these corresponds to a value such as *IntType* or *BoolType* of type *ptype*.[1] Later, we will add structured types like arrays and records.

At present, storage for each variable will be allocated statically with absolute addresses, so that the variable has a fixed address that can be chosen at compile time. Thus the address field *d_addr* can be a simple integer. We shall change this later when we introduce procedures with local variables and recursion: then variables associated with different invocations of a procedure will have different absolute addresses, and a more complex scheme will be needed for describing their locations. In many compilers, even global variables are not assigned a fixed address, but are left in symbolic form, so that the linker can collect together all the global variables for the whole program and assign them addresses consistently.

The symbol tables of our compiler are implemented by the module *Dict*.

[1] We spell *ptype* with a '*p*' because **type** is a reserved word of ML; the '*p*' is silent, as in 'ptarmigan'.

This provides a type *environment* that represents mappings from identifiers to definitions. The most important operations on environments are *define*, which adds a new definition, and *lookup*, which retrieves the definition of an identifier:

```
type environment;;

exception Already_defined;;

val define : def → environment → environment;;

val lookup : ident → environment → def;;

val init_env : environment;;
```

The *define* function raises the exception *Already_defined* if the identifier being declared already exists in the environment, and *lookup* raises the standard exception *Not_found* if the identifier can't be found. One of these situations arises when an identifier is declared more than once, and the other arises when an identifier is used without being declared. To start things off, there is a constant environment *init_env* that contains no definitions at all, and provides the basis on which all later environments are built. This module is straight-forward to implement efficiently using the type *Btree.map* defined in Section 2.8.

The semantic analyser communicates the results of its analysis to the code generator by *annotating* the abstract syntax tree of the program. We implement this by arranging that, at certain points, the tree built by the parser contains reference cells; the semantic analyser can update these to record information that can be retrieved later and used in generating code. For present purposes, we need to annotate each use of an identifier with its corresponding definition, and we need to annotate each expression with its type. To allow for this, we represent each applied occurrence in the tree by a record of type *name*:

```
type name =
  { x_name : ident;            (∗ Name of the reference ∗)
    x_line : int;              (∗ Line number ∗)
    x_def : def option ref };; (∗ Definition in scope ∗)
```

The standard type *α option* used here is defined like this:

```
type α option = Some α | None;;
```

This type provides a general way to represent values that may be either present or absent, since a value of type *α option* is either *Some x*, where *x* is a value of type *α*, or it is *None*. The *x_def* field of a *name* record has type (*def option*) *ref*: it is thus a reference cell that can contain either the value *Some d* for some definition *d*, or the value *None*.

Initially, the parser builds a tree in which each variable *x* that appears in an expression is represented by a record,

```
{ x_name = x; x_line = n; x_def = ref None },
```

where *n* is the line number on which the identifier appears. Storing the line number like this is a good way to pass the information needed by the semantic analyser to put fairly accurate line numbers in error messages, because most semantic errors are associated with the use of identifiers.

So that expressions in the tree can be annotated with their types, they are represented by values of type *expr*, defined like this:

```
type expr =
  { e_guts : expr_guts;           (* The expression itself *)
    e_type : ptype ref }          (* Space for a type annotation *)

and expr_guts =
    Number of int
  | Variable of name
  | Monop of op * expr
  | Binop of op * expr * expr
```

The changes from the previous version of *expr* are that variables now have a *name* record instead of an identifier, and each expression is represented with a record with space to add a type annotation. Initially, the parser builds a tree in which all these annotations have the dummy value *VoidType*.

To complete the story about abstract syntax trees, we look at the other end of the scale. A complete program in our little language consists of some declarations that introduce variables, followed by a list of statements that use the variables. The abstract syntax is of a program is therefore as follows:

```
type program = Program of decl list * stmt

and decl = Decl of ident list * typexpr

and typexpr =
    Integer | Boolean
  | ...

and stmt = ...
```

The type *typexpr* represents the 'type expressions' that appear in a variable declaration. For now, the possible expressions correspond exactly the integer and Boolean types in our initial language, but we will later add further kinds of expression that denote array and record types.[2]

6.2 Checking declarations and statements

In our compiler, the semantic analyser will be implemented in a module called *Check*. It analyses the declarations in the source program to construct an environment, then uses this environment to annotate the statements in the program body, checking as it does so that all variables have been declared, and that each expression has an appropriate type for the context in which it is used.

A single declaration in the tree has the form *Decl* (*vs*, *te*), where *vs* is a list of identifiers and *te* is a type expression. We can check this declaration using the function *check_decl*, defined as follows:

```
let rec check_decl size env (Decl (vs, te)) =
  let t = check_typexpr te in
  let n = type_size t in
```

[2] In our language, the names `integer` and `boolean` are keywords, rather than pre-defined identifiers as is usual in PASCAL-like languages.

```
    let declare env x = add_def (make_def x t (alloc size n)) env in
    fold_left declare env vs
```

In analysing declarations, we must keep track of the amount of storage that has been allocated and the environment that has been built up so far. The argument *size* is a reference cell that holds the total size of the storage allocated to far, and *env* is an environment containing all the variables that have been declared previously in the program. We use a function *check_typexpr* to compute the *ptype* value denoted by the type expression *te*; for now, this is trivial:

```
let check_typexpr =
  function
      Integer → IntType
    | Boolean → BoolType
    | ...
```

Continuing with the definition of *check_decl*, the function *type_size* computes the amount of storage occupied by this type; again, this can be trivial for now, because we will assume that both integers and Booleans occupy one unit of storage. This gives us the information we need to process each of the variables *vs*: we use *fold_left* to add them to the environment one at a time, and for each one, we allocate *n* units of storage (*alloc size n*), construct a definition of the variable (*make_def x t* ...), and add it to the growing environment (*add_def env* ...).

A program contains a list of declarations, each declaring one or more variables, so to annotate a whole program, we need another application of *fold_left* to process these declarations one at a time:

```
let check_decls size env ds = fold_left (check_decl size) env ds
```

The details are shown in Figure 6.1.

We can use these functions in analysing programs by first constructing an environment that contains the variables declared at the beginning of the program, then using this environment to check the statement that is the program's body.

```
let annotate (Program (decls, body)) =
  let size = ref 0 in
  let env = check_decls size init_env decls in
  check_stmt env body;;
```

Checking a statement involves checking and annotating the expressions that appear in the statement, and checking that the expressions have appropriate types. For example, here is the code for checking assignments, sequncing and if statements:[3]

```
let rec check_stmt env =
  function
      Assign (lhs, rhs) →
        let t1 = check_expr env lhs
        and t2 = check_expr env rhs in
```

[3] For reasons that will appear later, we treat the left-hand side of an assignment as an expression, just like the right-hand side.

```
(* add_def – add definition to env, give error if already declared *)
let add_def d env =
  try define d env with
    Already_defined →
      sem_error "$ is already declared" [fStr d.d_tag];;

(* alloc – allocate storage *)
let alloc size n =
  let a = !size in size := !size + n; a;;

(* make_def – construct definition of variable *)
let make_def x t a = { d_tag = x; d_type = t; d_addr = a };;

(* check_decl – check declaration and add to environment *)
let rec check_decl size env (Decl (vs, te)) =
  let t = check_typexpr te in
  let n = type_size t in
  let declare env x = add_def (make_def x t (alloc size n)) env in
  fold_left declare env vs

(* check_decls – check a list of declarations *)
and check_decls size env ds = fold_left (check_decl size) env ds
```

Figure 6.1: *Analysis of declarations*

```
        if t1 ≠ t2 then sem_error "type mismatch in assignment" [ ]
    | Seq ss →
        List.iter (check_stmt env) ss
    | IfStmt (cond, thenpt, elsept) →
        let t = check_expr env cond in
        if t ≠ BoolType then
          sem_error "boolean needed in if statement" [ ];
        check_stmt env thenpt;
        check_stmt env elsept
    | . . .
```

Note that the checking is done exactly once for each part of the program: in an if statement, we check *both* the then and else parts, and similarly, we check the body of a while loop exactly once. This is a crucial difference between analysis of a program by a compiler and execution of the program by a source-level interpreter.

To check expressions, we use a function *check_expr* : *environment* → *expr* → *ptype*. This function checks for semantic errors (like undeclared variables) in the expression, annotates each variable with the proper definition, annotates the expression and each sub-expression with its type, and returns the type of the whole expression. It's simplest to define *check_expr* by using mutual recursion with a function *expr_type* that does everything but annotate the root of the tree, like this:

```
let rec check_expr env e =
  let t = expr_type env e in e.e_type := t; t

and expr_type env e =
    match e.e_guts with
        Number n → IntType
      | Variable x → lookup_def x env
      | ...
```

The code for *expr_type* uses pattern matching to find the type of an expression by examining its *e_guts* field; *check_expr* then stores that type as an annotation of the expression and returns it. I've filled in the case that deals with integer constants, because it's easy – the type is *Integer* – and the case that deals with identifiers, which uses an auxiliary function *lookup_def*. If an error is found, it is reported by raising an exception, so *check_expr* does not return any type for the expression.

The function *lookup_def* that deals with identifiers is defined as follows:

```
let lookup_def x env =
  err_line := x.x_line;
  try
    let d = lookup x.x_name env in
    x.x_def := Some d; d.d_type
  with
    Not_found →
      sem_error "$ is not declared" [fStr x.x_name];;
```

This function looks up an applied occurrence *x* in the environment *env*. If the name is not declared, then *lookup* raises the exception *Not_found*. That exception is caught here and the result is a semantic error. We'll assume that *sem_error* raises another exception that isn't caught except in the compiler's main program, so we don't have to worry here about what happens next. If, however, the name *has* been declared, we annotate *x* with the definition and return its type. The definition *d* will later be retrieved by the code generator and used to determine the address of the variable.

Another case in *expr_type* covers binary operations *Binop* (w, e_1, e_2). To deal with these, we first check the sub-expressions e_1 and e_2 and find their types, then check with an auxiliary function *check_binop* that these types are allowed for operands of the operator *w*:

```
        ...
      | Binop (w, e1, e2) →
          let t1 = check_expr env e1
          and t2 = check_expr env e2 in
          check_binop w t1 t2
```

The function *check_binop* itself has one case for each kind of operator. It examines the types t_1 and t_2 of the two operands to make sure they are suitable; if so, it returns the type of the result:

```
let check_binop w t1 t2 =
  match w with
      (Plus | Minus | Times | Div | Mod) →
        if t1 ≠ IntType || t2 ≠ IntType then type_error ();
        IntType
    | (Eq | Lt | Gt | Leq | Geq | Neq) →
        if t1 ≠ t2 then type_error ();
        BoolType
        | ...
```

Unary operations of the form *Monop* (w, e_1) can be treated in a similar way, completing the semantic analyser for our language.

6.3 Extending the abstract machine

The abstract machine code we have generated so far has been unrealistic in that the ‘state’ of the machine’s memory has been a mapping from identifiers to values, and real machines use numbers instead of names to identify memory locations, for the very good reason that hardware is easier to build for decoding addresses that are numbers.

Changing to numeric addressing means modifying our abstract machine, replacing the *state* indexed by names by a *store* indexed by integer addresses. We could do this by just replacing the instruction *LOAD x* (where *x* is an identifier) with a new instruction *LOAD a* (where *a* is a numeric address), and the same for the *STORE* instruction. However, we can better pave the way for future developments if we add a new instruction *GLOBAL a*, so that the two instructions *LOAD x* and *STORE x* are replaced by the three instructions *GLOBAL a*, *LOAD*, and *STORE*. Here are the effects of these three instructions:

- *GLOBAL a* pushes the address *a* onto the stack. We choose the name *GLOBAL* because *a* is always the address of a global variable.
- *LOAD* pops an address *a* off the stack, and pushes the contents of memory location *a*.
- *STORE* pops *two* items off the stack: a value *v* and an address *a*. It stores the value *v* in memory at the address *a*.

Thus the function of the old *LOAD* instruction has been taken over by the two instructions *GLOBAL* and *LOAD*, and the function of the old *STORE* instruction has been taken over by *GLOBAL* and *STORE*. The advantage of this way of doing things starts to show when we implement data types like arrays and records: then we shall want to be able to *compute* the address of a data structure element before loading or storing its value, and these *LOAD* and *STORE* instructions neatly allow for that.

We can build a simulator for the modified abstract machine by introducing an vector *mem* that holds the contents of data memory. This vector is completely separate from the vector *prog* that holds the program. We also keep the list variable *stack* that holds the contents of the expression stack, and define functions *push* and *pop* that access it:

```
let mem = Array.create memsize 0;;     (* Data memory *)
let stack = ref [ ];;                  (* Stack *)

let push v = stack := v :: !stack;;
let pop ( ) = let v = hd (!stack) in stack := tl !stack; v;;
```

Figure 7.2 shows a version of *exec_inst* that implements the new instructions; this can be used with same fetch-execute cycle *run* as before to make a simulator for the new machine. The implementations of the *CONST* and *GLOBAL* instructions are identical, so the two instructions could be combined into one; however, one deals with numeric constants, and the other deals with

```
let exec_inst =
  function
      Const x → push x
    | Global a → push a
    | Load → push mem.(pop ())
    | Store → let a = pop () in mem.(a) ← pop ()
    | Monop w → push (do_monop w (pop ()))
    | Binop w → let b = pop () in let a = pop () in
                        push (do_binop w a b)
    | Jump l → pc := find_lab l
    | Jumpb (b, l) → if b = (pop () ≠ 0) then pc := find_lab l
    | Stop → raise Finish
    | _ → failwith "bad instruction";;
```

Figure 6.2: *Simulator with numeric addresses*

addresses, so it aids clarity to keep the separate. Also, we shall eventually want to replace *Global* with a version that uses symbolic addresses instead of plain numbers, so it is good to keep the two instructions separate from the start.

6.4 Revising the code generator

Only two changes are needed to our intermediate code generator to cope with the new, numeric addressing scheme. The function *gen_expr* that generates code for expression evaluation needs to be changed to generate the new form of the *Load* instruction:

```
  match e.e_guts with
      Variable _ | Sub (_, _) | Select (_, _) →
        gen_addr e;
        gen Load
    | . . .
```

Here *get_def* is a function that extracts the definition that has been added as an annotation to a *name* record. The auxiliary function *gen_addr* expects to be passed a variable; it generates an instruction that pushes the address of the variable onto the stack:

```
let gen_addr e =
  match e.e_guts with
      Variable x →
        let d = get_def x in
        gen (Global d.d_addr)
```

The *Global* instruction that is generated uses the address that was allocated for the variable and stored in its definition by the semantic analyser.

The only other change is to the treatment of assignment statements, which must now use the new *Store* instructions. We are treating the left-hand side of an assignment as an expression, so we can use *gen_addr* again here:

```
        Assign (e₁, e₂) →
          gen_expr e₂;
          gen_addr e₁;
          gen STORE
      | . . .
```

This enhancement to the intermediate code generator completes our new compiler.

In order to generate code that uses numeric addresses instead of symbolic names, what we have done is to split the mapping from names to values into two parts. What was previously a single mapping – the *state* – now becomes the composition of two mappings: an *environment* that maps identifiers to 'locations', and a *store* that maps locations to values. Locations are just the same as addresses in the target machine, and the store is just the same as the contents of run-time memory.

For a simple language, where there are no recursive procedures, the environment can be just the same as the compiler's symbol table: looking up an identifier in the symbol table gives its address in run-time memory. FORTRAN (in its early versions) was a language in this class, and its design allowed all storage to be allocated at compile time, giving maximum efficiency even on the crude machines of the late 50's.

With recursive procedures, the picture is more complicated, because the absolute addresses of variables are not known at compile time. A recursive procedure may have several calls active at once, and each activation will have its own parameters and local variables. This implemented by using a stack to allocate to each activation of the procedure a block of space called a *stack frame* in which to store its local variables. Each local variable of a procedure has a fixed location within this frame, but the address of the frame itself differs from one activation to another.

We can explain this by saying that the environment exists partly at compile time and partly at run time. The compiler's symbol table contains part of the environment, but another part is in the run-time data structure that describes the location of the frame for each procedure activation. For now, however, we will stick with a simple mapping from identifiers to fixed addresses.

6.5 Adding arrays

So far, our code generator and abstract machine support only simple scalar variables. We now turn our attention to remedying this defect by showing how to extend each part of the compiler to support arrays. This requires changes to several parts of our compiler. We'll just sketch the changes here, and leave you to work through some of the details as you do Lab 4. Briefly, we need to change:

- The abstract syntax – to allow array types, and to add variable references that involve subscripting.

- The semantic analyser – to deal with declarations of objects that are bigger than one storage unit, and to handle subscripting by checking, e.g., that we don't try to use a subscript with a variable that isn't an array.

- The intermediate code generator – to generate code for subscripting in expressions and assignments.

We won't need to change the abstract machine, because the *GLOBAL*, *LOAD* and *STORE* instructions we recently introduced provide just what we need.

6.5.1 Extending the abstract syntax

To add arrays, we need to extend the syntax of types. For simplicity, we'll make all arrays start at zero, so there's just an upper bound. We need to redefine the type *ptype* like this:

```
type ptype =
    IntType | BoolType | VoidType
  | ArrayType of int ∗ ptype;;
```

The syntax is recursive, so we can have types like

```
array 3 of array 5 of integer
```

for an array of 15 integers numbered from a[0][0] up to a[2][4]. This type is represented in the compiler by the *ptype* value *Array* (3, *Array* (5, *Integer*)).

We also need to extend the syntax of expressions to allow subscripting:

```
type expr_guts = ...
  | Sub of expr ∗ expr
  | ...
```

The tree *Sub* (e_1, e_2) represents the expression $e_1[e_2]$. We allow e_1 to be an *expression* (rather than a plain identifier) to accommodate cases like a[i][j] where the second subscripting operation acts on the result of the first one.

Subscripting can be used on the *left* of assignments as well as on the right, and we allow for this by making both sides be expressions, like this:

```
type stmt = ...
  | Assign of expr ∗ expr
  | ...
```

We can rely on the syntax analyser to reject things like

```
x + 4 := 5
```

that don't make any sense.

6.5.2 Semantic analysis

An array a of size *n* is represented in run-time memory by a block of storage that contains *n* copies of the representation of the array's element type. The address of the element a[i] can be calculated as the address of a plus *i* times the size of one element.

The changes to needed in the semantic analyser are fairly simple: we need to add some code to check expressions that involve subscripting, and we need to remove the assumption that every data item occupies one unit of storage. The details are all dealt with in Lab 4.

6.5.3 Code Generation

We also need to extend the intermediate code generator to deal with array subscripting. A key observation is that what we really need to be able to do is calculate the *address* of an array element; then we will be able to deal with subscripts in both expressions and assignments.

At present, the code we generate to fetch the value of a simple variable x is

```
GLOBAL ⟨address of x⟩
LOAD
```

and the code we generate to assign to x is

```
⟨Compute new value ⟩
GLOBAL ⟨address of x⟩
STORE
```

To deal with subscripting in expressions a[i] and in assignments a[i] := ..., we just need to replace the *Global* instruction in each of these code fragments with some code that computes the address of the array element onto the stack.

The address of an array element a[i], where a has type array N of T, is computed by adding the address of the array and the offset of the element required, and the offset is computed by multiplying the value of i by the size of an object of type T:

```
⟨Compute address of a⟩
⟨Compute subscript i⟩
CONST ⟨size of T⟩
BINOP Times
BINOP Plus
```

Multi-dimensional arrays can be represented in picoPascal as arrays of arrays. If M is declared by

```
var M: array 10 of array 10 of integer;
```

then each element M[i] of M is an array of 10 integers; and each element M[i][j] of that array is an integer. To deal with subscripting of such arrays, we can just apply the scheme above recursively. Thus code to load the value of M[i][j] is as follows:

```
GLOBAL ⟨address of M⟩
GLOBAL ⟨address of i⟩
LOAD
CONST 10
BINOP Times
BINOP Plus
GLOBAL ⟨address of j⟩
LOAD
CONST 1
BINOP Times
BINOP Plus
LOAD
```

As I've tried to show in the schematic view in Figure 7.3, this code can be generated by following the recipe for loading the value of M[i][j], and as part of the recipe recursively following the recipe for loading the *address* of M[i].

```
⟨Compute address of M[i][j]⟩
    ⟨Compute address of M[i]⟩
        ⟨Compute address of M⟩
            GLOBAL ⟨address of M⟩
        ⟨Compute subscript i⟩
            GLOBAL ⟨address of i⟩
            LOAD
        CONST 10
        BINOP Times
        BINOP Plus
    ⟨Compute subscript j⟩
        GLOBAL ⟨address of j⟩
        LOAD
    CONST 1
    BINOP Times
    BINOP Plus
LOAD
```

Figure 6.3: *Code for* M[i][j]

A convenient way to organize the code generator uses two mutually recursive functions *gen_addr* and *gen_expr*. As before, *gen_expr* generates code that has the net effect of pushing the value of a given expression onto the stack; and *gen_addr* generates code that has the net effect of pushing the address of a given variable. The two procedures are mutually recursive because *gen_expr* calls *gen_addr* to calculate the address of a variable before loading its value with a *LOAD* instruction, and *gen_addr* calls *gen_expr* when the variable involves subscripting, in order to calculate the value of the subscript.

6.6 Adding records

Records, like arrays, can be implemented with some simple arithmetic on addresses. Each record of a certain type is laid out in the same way, so we can compute the address of a field r.x by taking the address of the field r and adding a constant that depends on the field x that is being selected. Records, however, present a new problem, because we must deal with the fact that fields are identified by names, and we must keep track of what field names are defined, and what offset is associated with each name. We already have a mechanism – environments — for dealing with this when the names are the variables that have been declared in a program and the offsets are their addresses.

Adding records to the language we have built up so far gives the following concrete syntax for declarations and type expressions:

decls	→	*decl* { *decl* }
decl	→	*ident* { "," *ident* } ":" *typexp* ";"
typexp	→	integer

```
                | bool
                | array number of typexp
                | record decls end
```

With similar additions, the abstract syntax of declarations and type expressions is as follows:

```
type decl = Decl of ident list ∗ typexpr

and typexpr =
      Integer | Boolean
    | Array of int ∗ typexpr
    | Record of decl list
```

This syntax is recursive in a new way, because declarations may contain record types which themselves contain declarations for fields; we immediately raise the possiblity of nested record types.

It is now that our distinction between types and type expressions begins to pay off, because we can conveniently introduce a representation of record types where the fields are represented by a list of definitions, together with the total size:

```
type ptype =
      IntType | BoolType | VoidType
    | ArrayType of int ∗ ptype
    | RecordType of def list ∗ int
```

Among other things, this representation discards the irrelevant detail of exactly how the fields of a record were declared, so that the two type expressions

```
record x: integer; y: integer; end
```

and

```
record x, y: integer; end
```

will denote exactly similar types.

In order to convert a type expression to the type it denotes, we will need a two new operations on environments, which will also be useful when we come to consider nested blocks and procedures. The first operation lets us start a new block:

```
val new_block : environment → environment;;
```

The effective difference between the environment *env′* = *new_block env* and the environment *env* is that if *env* already contains a variable *x*, then we receive a *Already_defined* exception if we try to declare *x* afresh in *env*, but we are permitted to declare *x* in *env′*, just once. Although it is not a very good idea to do it deliberately, this will allow us to deal with records that have a field with the same name as a variable in the program; later we shall use the same function to deal with inner blocks that contain a varaible with the same name as another variable in an outer block.

The other new operation on environments is related: we can obtain a list of names defined since the last *new_block* operation by calling the function *top_block*:

```
val top_block : environment → def list;;
```

This operation is perfect for implementing record types: we make a new block, add to it all the field declarations, automatically checking as we go that no field is declared more than once, then obtain the list of fields by calling *top_block*:

```
let rec check_typexpr =
  function ...
    | Record fs →
        let size = ref 0 in
        let env = check_decls size init_env fs in
        RecordType (top_block env, !size);;
```

The function *check_decls* used here is exactly the same function that is used to check the variable declarations in the program.

So that programs can refer to the fields of records, we add a new kind of expression with the concrete syntax *variable* "." *ident* and the abstract syntax

```
type expr = ...
  | Select of expr * name
  | ...
```

To check that such an expression $e_1.x$ is valid, we must check that e_1 is a valid expression with a record type, and that x is one of its fields:

```
let rec expr_type env e =
  match e.e_guts with ...
      | Select (e1, x) →
          let t1 = check_expr env e1 in
          ( match t1 with
                RecordType (fs, s) →
                  let d = try find_def x.x_name fs
                    with Not_found →
                      sem_error "selecting non-existent field" [ ] in
                  x.x_def := Some d; d.d_type
              | _ → sem_error "selecting from a non-record" [ ] );;
```

As a side-effect, this checking code annotates the name x with the definition of the field. The function *find_def* : *ident* → *def list* → *def* used here searches a list of definitions for a given identifier, returning the first matching definition, or raising *Not_found* if no definition matches.

The final detail needed to implement records is to explain how to calculate the address of a field, given the address of the record containing it. Here is the relevant code from the function *gen_addr*:

```
let rec gen_addr e =
  match e.e_guts with ...
    | Select (e1, x) →
        let d = get_def x in
        gen_addr e1;
        gen (CONST d.d_addr);
        gen (BINOP Plus)
```

The code that is generated adds together the address of the record and the offset of the field.

6.7 Pointers and recursive types

The final step of increasing sophistication that we will consider is to introduce recursive types with pointers: this brings our language almost up to the level of C or Pascal.

The usual way of introducing recursive types into a Pascal-like language is to allow a type declaration

```
type t1 = pointer to t2;
```

where t2 is a type that has not yet been defined, but will be defined later in the same declaration. Using this style, a simple type of linked lists might be declared as follows:

```
type list = pointer to cell;
  cell = record data: integer; next: list end;
```

This style is a little inconvenient at times, because it forces us to invent a name for the type cell, even if only the type list is used to declare variables.

We will use the Pascal-like notation p^ where p is a pointer, to denote the object that p points to, and represent this with the abstract syntax *Deref p*. We will also provide a statement new p, represented by the abstract syntax *NewStmt p*, that allocates a fresh block of storage and assigns its address to p. In Pascal, this function is performed by a built-in procedure.

Recursive types introduce a new problem for type-checking, because the question whether two types are the same is no longer a question about the equality of two finite expressions. Two types with the same structure may be defined by different recursive definitions, and if the rules of the source language require such types to be treated as being the same, then the compiler must contain an algorithm for deciding whether two recursive types have the same structure. This notion of type equivalence is called *structural equivalence*, and although algorithms for deciding structural equivalence exist, they are quite complicated.

In practical terms, structural equivalence adds very little expressive power to the programming language, compared to the alternative notion of *name equivalence*. According to this, each use of a keyword such as record or array or pointer creates a new type that is different from every other type in the program, even if the other type is defined in a similar way.

We can implement name equivalence very simply: each type should be labelled with a unique time-stamp that is assigned when the type is created, and two types are equal exactly if they bear the same time-stamp. For clarity, we will make time-stamps be integers, although in a compiler written in a lower-level language than ML, the time-stamp of a type might be the address of the record that represents it, so that type equivalence becomes pointer equality. Some languages relax name equivalence a little, perhaps so that record types are treated by name equivalence, but two types written as different occurrences of pointer to t are nevertheless regarded as equivalent.

Generating code for the dereferencing operation p^ is suprisingly simple: the *address* of p^ is the *value* of p. Thus we can add the following clause to the definition of *gen_addr*:

```
let gen_addr e =
  match e.e_guts with ...
```

```
    | Deref e1 →
        gen_expr e1
    | ...
```

The value of p^ is obtained by loading from this address, and code for this is generated by the general rule that the value of a variable is obtained by pushing its address and performing a *LOAD* instruction:

```
let gen_expr e =
  match e.e_guts with
      Variable _ | Sub (_, _) | Select (_, _) | Deref _ →
        gen_addr e;
        gen LOAD
    | ...
```

To give an example, the assignment statement p^ := q^ produces the code

```
GLOBAL q
LOAD
LOAD
GLOBAL p
LOAD
STORE
```

In this code, the sequence *GLOBAL q*/*LOAD* pushes the value of the pointer variable q; this value is the address of the variable that is written q^, and a further *LOAD* instruction gives the value of that variable, which is what is copied by the assignment. In order to store the value into the variable p^, we must first push the address of that variable, which is the same as the value of the pointer variable p, obtained by the sequence *GLOBAL p*/*LOAD*. Finally, the *STORE* instruction stores the computed value of q^ into the variable p^ whose address has just been computed.

Pointers are more commonly used in combination with records to build reference-linked data structures, so let us look at an example of that. Suppose we have made the declarations,

```
type list = pointer to cell;
  cell = record key, data: integer; next: list end;

var p: list;
```

(I've given each cell two integer fields to help the example.) The assignment

```
p^.data := p^.next^.data
```

copies into the data field of the record pointed to by p whatever value is held in the data field of the next record. Code for this assignment may be generated by following the rules given above. First, we must compute the value of p^.next^.data, To do so, we begin with the value of pointer variable p:

```
GLOBAL p
LOAD
```

This is the address of the record from which we need the next field, and to find the address of that field, we must add its offset (2) to the address of the record:

```
CONST 2
BINOP Plus
LOAD
```

This again gives the address of a record, and we must retrieve the value of the data field (offset 1):

```
CONST 1
BINOP Plus
LOAD
```

We have completed the code that pushes the value of the RHS of the assignment statement, and must now produce code to compute the address of the variable on the LHS. Since this is a field of the record to which p points, we take the value of p and add the offset 1:

```
GLOBAL p
LOAD
CONST 1
BINOP Plus
```

The final step is to store the computed value of the RHS into the computed address of the LHS:

```
STORE
```

I hope these examples have convinced you that our small collection of instructions is sufficient to produce code for any combination of dereferencing, selection and indexing. Since our list of instructions is so small, the code sequences that are generated are sometimes rather long. They can be shortened by adding further instructions that are equivalent to various combinations of more basic instructions: for example, any code that manipulates records is likely to contain many sequences of the form *CONST n*/*BINOP Plus*/*LOAD* or *CONST n*/*BINOP Plus*/*STORE*, and we could shorten the code by introducing new instructions that abbreviate these sequences. Doing that would also increase the speed of an implementation based on interpreting the Keiko code, because the interpreter would be able to carry out the operation as a single action instead of three separate ones.

6.8 Record and array assignments

There is one glaring error in the compilers we have developed in this chapter: if a and b are array or record variables, then an assignment a := b does entirely the wrong thing, because it copies only a single value, not the whole array or record.

Exercises

6.1 A small extension to our language would be to allow blocks with local variables. We can extend the syntax by adding a new kind of statement:

stmt → declare *decls* in *stmts* end

For example, here is a program that prints 53:

```
var x, y: integer;
begin
  y := 4;
  declare y: integer; in
    y := 3 + 4; x := y * y
  end;
  print x + y
end.
```

Space for the local variables can be allocated statically beyond the end of the space allocated for global variables (and beyond the space for any enclosing blocks). Sketch the changes needed in our compiler to add this extension. Try to arrange things so that local blocks can share the same storage if they are not nested.

6.2 A certain imperative programming language contains a looping construct that consists of named loops with exit and next statements. Here is an example program:

```
loop outer:
  loop inner:
    if x = 1 then exit outer end;
    if even(x) then x := x/2; next inner end;
    exit inner
  end;
  x := 3*x+1
end
```

Each loop loop L: ... end has a label L; its body may contain statements of the form next L or exit L, which may be nested inside inner loops. A loop is executed by executing its body repeatedly, until a statement exit L is encountered. The statement next L has the effect of beginning the next iteration of the loop labelled L immediately.

(a) Suggest an abstract syntax for this construct.

(b) Suggest what information should be held about each loop name in a compiler's symbol table.

(c) Briefly discuss the checks that the semantic analysis phase of a compiler should make for the loop construct, and the annotations it should add to the abstract syntax tree to support code generation. Give ML code for parts of a suitable analysis function.

(d) Show how the construct can be translated into a suitable intermediate code, and give ML code for the relevant parts of a translation function.

6.3 In the compiler design discussed in the text, the fields of a record are stored as a simple list of definitions. Outline the changes that would be needed to keep the fields as an environment, represented like other environments by using the *Btree* module. What would be the advantages and disadvantages of this scheme?

Chapter 7

Interpreting procedures

Perhaps the most important feature that separates one family of programming languages from another is the support they give for subroutines or procedures. The support for subroutines is also more important than almost any other language feature in determining the usefulness of a language for programming. For example, poor data structuring facilities can be tolerated if the language provides a good subroutine mechanism that can be used to encapsulate the details of how a data structure is implemented into a group of subroutines.

We can classify languages into groups according to the kinds of subroutines that they support:

- FORTRAN supports simple subroutines without recursion or nesting, but with access to global variables through common blocks.
- C supports recursive subroutines with global variables, but they cannot be nested.
- ALGOL-like languages (including PASCAL and MODULA-2) have nested, recursive subroutines with full access to variables in enclosing routines.
- Both C and the ALGOL family allow subroutines to take arguments that are other subroutines.
- Functional languages like ML have subroutines that return other subroutines as their results. Combining this with nested procedures makes the procedure mechanism in these languages very powerful.

Figure 7.1 shows an example of a PICOPASCAL program that exploits nested procedures and the use of one procedure as an argument to another. The function *sum* takes two arguments: an integer n and a function f, and computes $f(0) + f(1) + \cdots + f(n-1)$. The function *sum_powers* uses this function to compute $0^k + 1^k + \cdots + (n-1)^k$ for given n and k. It does this by passing to *sum* a nested function *power* that computes x^k for given x.

It's worth noticing that this program could not be translated directly into C, because the *power* function could not get access to the parameter k unless it was nested inside *sum_powers*. An equivalent C program would have to rely on a surreptitious global variable for communication between *sum_powers* and *power*.

```
proc sum(n: integer; proc f(x: integer): integer): integer;
  var i, s: integer;
begin
  i := 0; s := 0;
  while i < n do
    s := s + f(i);
    i := i + 1
  end;
  return s
end;

proc sum_powers(n: integer; k: integer): integer;
  proc power(x: integer): integer;
    var j, p: integer;
  begin
    j := 0; p := 1;
    while j < k do
      p := p * x;
      j := j + 1
    end;
    return p
  end;
begin
  return sum(n, power)
end;

begin
  print_num(sum_powers(10, 3));
  newline()
end.
```

Figure 7.1: *Nested procedures and procedural parameters*

An example of the power of nested procedures coupled with recursion is shown in Figure 7.2. This program solves the following puzzle:

> Find a nine-digit number whose digits are a permutation of the decimal digits 1 to 9, such that the number itself is divisible by 9, the number consisting of all but the last digit is divisible by 8, the number consisting of all but the last two digits is divisible by 7, and so on.

In the program, procedure *search* looks for a way to complete a solution by adding more digits to the $(k-1)$-digit number n. The procedural parameter *avail* is a Boolean function that says whether each digit is still available for use. It would be idle to pretend that this program represents a reasonable way of solving the puzzle, when a simple array of Booleans would be a much more practical way of representing the set of available digits; but it does give a small example of the power of programming with functions. Again, the program could not be translated directly into C, because the nested function $avail_1$ needs access to the local variable d of *search*.

The next section introduces the idea of subroutines more formally by showing how to build a source-level interpreter for a language with subrou-

```
proc search(k, n: integer; proc avail(x: integer): boolean);
  var d, n1: integer;
  proc avail1(x: integer): boolean;
  begin
    return (x <> d) and avail(x)
  end;
begin
  if k = 10 then
    print_num(n); newline()
  else
    d := 1;
    while d < 10 do
      n1 := 10 * n + d;
      if (n1 mod k = 0) and avail(d) then
        search(k+1, n1, avail1)
      end;
      d := d+1
    end
  end
end;

proc all_avail(x: integer): boolean;
begin
  return true
end;

begin
  search(1, 0, all_avail)
end.
```

Figure 7.2: *Procedural parameters combined with recursion*

tines: you will be constructing a similar interpreter in Lab 5, and extending it to handle reference and functional parameters in addition to value parameters. The rest of this chapter uses source level interpreters to explore the choices that have to be made in defining the meaning of procedures.

In the next chapter, we look in stages at the implementation of procedures in a compiler, beginning with the simplest subroutines without parameters or nesting, and ending with the something close to the full procedure mechanism of PASCAL.

7.1 Procedures and parameters

To explore what might be needed to implement procedures, we begin by taking the easy option and building a source-level interpreter. As a first try, we'll implement the simplest kind of procedure without parameters. To do this, the *easiest* thing (although, as we'll find, one that is inadequate to support parameters properly) is to change our notion of *state* so that a state maps names either to integers or to procedures, and to represent a procedure by the statement list that is its body:

```
type value =
    IntVal of int
  | ProcVal of stmt list;;
```

A radical step is being taken here, because we are storing a statement list (which is part of the abstract syntax) as part of the state (which is part of our semantics). Thus we are taking a definite step along the road to operational semantics, and away from denotational semantics.

All that aside, how can we build our interpreter? Well, we'll have to modify the function that evaluates expressions so that it expects values from the state to have the form *IntVal n* instead of being just the integer *n*: that's easy. We should also add a new kind of statement:

```
type stmt = ...
  | Call of ident
  | ...
```

and extend *exec_stmt* to handle it:

```
let rec exec_stmt env =
  function ...
    | Call p →
        let (ProcVal b) = value s p in
        exec_stmts s b
    | ...
```

So a call statement is executed by just executing the statements that form the procedure body. We also need some mechanism for defining procedures, but we'll leave that until we can do it properly.

An important thing to notice is that our rule for executing procedure calls is not structural recursion, unlike many of the other constructs of the language, but like while loops. This is an intimation that the procedure call mechanism in our language is already rather powerful: it supports recursion, and we can write programs like this one, which uses recursion to calculate factorials:

```
proc fac;
begin
  if n > 0 then
    f := f * n;
    n := n - 1;
    fac
  end
end;
```

```
begin
  n := 10; f := 1;
  fac
end.
```

The next step is to add parameters. Our first thought is to make them like ordinary variables, and make parameter passing like assignment. When a procedure is called, all the actual parameters would be evaluated, and their values would be assigned to the formal parameters by updating the state, just like assignment. This might work for some programs, but it is a disaster if a formal parameter ever has the same name as a global variable, or if a procedure is recursive, as in the following implementation of fac:

```
var f;

proc fac(n);
begin
  if n = 0 then
    f := 1
  else
    fac(n-1);
    f := f * n
  end
end;
```

Because the recursive call of the fac procedure destroys the value of n, this procedure sets f to 0 if the argument is greater than zero, and 1 otherwise: it computes 0^n instead in $n!$.

The problem here is that while fac is active, there should be several variables in the program that have the name n: one for each copy of the fac procedure that is active. It's hopeless to try and store all these values in the state under a single name n. Thus we are forced into re-discovering the idea of separating *environment* and *store*, and introducing a concept of *location*. If this idea had not been suggested by our machines, it would have been necessary to invent it in order to explain our programming languages.

It's time to revise our story about states in the source-level interpreter, as we revised it for our abstract machine, but for a different reason this time. The *store* part of the story is this: there are two types *store* and *location*, and we can save and recover integer values at a given location in a store. There's also an initial store *init_store*:

type *store*;;

type *location*;;

val *get* : *location* → *store* → *int*;;

val *put* : *location* → *int* → *store* → *store*;;

val *init_store* : *store*;;

We'll add a new operation that invents a fresh location and initializes it with a given value, returning the new location and the modified store:

val *fresh* : *int* → *store* → *location* ∗ *store*;;

This operation is used in declaring variables to make sure that they are given a location that doesn't clash with any other.

So much for the *store* part of our framework: what about the *environment* part? We'll make an environment be in effect a mapping from identifiers to *definitions*, and definitions will contain either a location or a procedure definition. For now, a procedure definition contains a list of identifiers (the formal parameters) and a list of statements (the procedure body).

```
type procdef = ident list * stmt;;

type def =
    VarDef of Store.location      (* Variable (location) *)
  | ProcDef of procdef;;          (* Procedure (closure) *)
```

As before, we'll have operations *define* and *lookup* on environments, and an initial, empty environment *init_env*:

```
type environment;;

val define : ident → def → environment → environment;;

val lookup : ident → environment → def;;

val init_env : environment;;
```

In our previous source-level interpreters, the *eval* function has taken two parameters: a state and the expression to be evaluated. Now we'll have to pass three parameters, because the state has split into two parts: the environment and the store:

```
value eval : environment → store → expr → int;;
```

The vital change is in the way variables are handled: we use the environment first to find the relevant location, then use the store to find the value stored in that location.

```
let rec eval env s =
  function
      Variable x →
        let (VarDef a) = lookup x env in get a s
    | . . .
```

The type of *exec* also changes: in previous versions, it took a state and a statement and returned the new state after executing the statement. Now it takes an environment, a store and a statement, and returns a new store. Since executing a statement does not change the environment, we don't need to make *exec* return a new environment.

```
value exec : environment → store → stmt → store
```

The nature of the change can be seen in the way assignment statements are handled:

```
let rec exec env s =
  function
      Assign (x, e) →
        let (VarDef a) = lookup x env in
        put a (eval env s e) s
    | . . .
```

What happens is this: we look up the variable on the left hand side in the environment: this gives a location, and we update the contents of that location with the value of the right hand side.

I should explain how to process declarations. Although the discussion in Chapter 6 showed how to handle declarations with types, we'll go back here to a programming language where all variables are integers: procedures are hard enough to understand without *that* added complication.

Declaring a variable modifies both the store (by initialising a fresh location) and the environment (by adding a new definition). Here's a function that does both these things for a single variable, returning the new environment and store:

```
let decl_var (env, s) x =
  let (a, s') = fresh 0 s in
  (define x (VarDef a) env, s');;
```

For simplicity, I've initialized each new variable with zero. We can use that function to define a function

```
value declare : environment ∗ store → decl → environment ∗ store;;
```

that processes both variable and procedure declarations:

```
let declare (env, s) =
  function
      VarDecl vs → fold_left decl_var (env, s) vs
    | ProcDecl (p, formals, body) →
        (define p (ProcDef (formals, body)) env, s);;
```

This is the last ingredient we need to build our interpreter:

```
let run (Program (globals, main)) =
  let (env, s) = fold_left declare (init_env, init_store) globals in
  exec env s main;;
```

Did I say the last ingredient had been added? Not quite! I still haven't explained how to execute procedure calls, so we need to add another rule to *exec* for that:

```
let rec exec env s =
  function . . .
    | Call (p, es) →
        let (ProcDef (formals, body)) = lookup p env in
        let actuals = map (eval env s) es in
        let (env', s') =
          fold_left decl_arg (env, s) (combine formals actuals) in
        exec env' s' body
    | . . .
```

Let's take it step by step . . .

- To execute a procedure call *Call* (*p*, *es*), we first look up the procedure name *p* in the environment, hoping to find that it is defined as a procedure *ProcDef* (*formals*, *body*), where *formals* is the list of formal parameters, and *body* is the list of statements in the procedure definition.

- Then we evaluate all the actual parameters, getting a list *actuals* of parameter values. A new environment *env′* is formed by adding the parameters to the environment one after another: we first combine the list *formal* of formal parameter names with the list *actual* of actual parameter values to form a list of pairs, then process them using *fold_left* and a function *decl_arg*, defined like this:

  ```
  let decl_arg (env, s) (fp, av) =
    let (a, s′) = fresh av s in
    (define fp (VarDef a) env, s′);;
  ```

  This function finds a fresh location *a* and initializes it with the value *av* of the actual parameter; then the formal parameter name *fp* is defined in the environment as a variable with the location *a*.

- Finally, the statements in the procedure body are executed in this environment: they can use the parameters, as well as variables that were declared globally and are still present in the environment *env′* that we have constructed. When execution of the procedure body is over, the final state is kept, but the environment *env′* is thrown away: this undoes the binding between the formal parameter names and the locations we invented to hold the actual parameters.

This really does complete our source-level interpreter for a language with procedures and parameters. In the next section, we examine some of the properties of this implementation.

Lab 5 asks you to repeat some of the work of this section in a setting that is slightly easier to manage: there, you will use an abstract data type with destructive operations to represent the state, and will use the destructive operations to update it in assignment statements and parameter passing. This makes it possible to avoid passing the state around the interpreter explicitly, but it means that we can't implement constructs like the *recovery blocks* of Exercise 5.5 that require several states to be accessible at once: maybe that's a good thing.

Isn't storage space for procedure parameters usually allocated from a stack? Yes it is – and we shall be looking at ways of organising such a stack in the next chapter. But that's an *implementation*, and we want to use our source-level interpreters as *specifications* (or if you prefer *prototypes*) for the programming language. Some languages (like PASCAL) can be implemented by allocating storage from a stack, and others (like ML and Haskell) can't. The use of a stack is just an implementation of the idea of allocating fresh locations that exploits some restrictions on the lifetimes of locations that are imposed by a certain class of programming languages. Of course, the designers of those languages had in mind the restrictions needed to allow stack-based storage: but we don't have to use a stack to explain what programs in the language mean.

7.2 Static and dynamic binding

What will our interpreter do with this program?

```
var x;
```

```
proc p();
begin
  x := 3
end;

proc q(x);
begin
  p()
end;

begin
  x := 2;
  q(4);
  print x
end
```

In this program, there are two entities that have the name x: a global variable, and the formal parameter of procedure q. The applied occurrence of x in procedure p *looks as if* it should be associated with the global x, and if p were called from the main program, it certainly would be. Unfortunately, however, it's the x that is the formal parameter of q that gets modified when p is called from q. The program prints 2 instead of 3.

To see why this should be so, let's consider how our interpreter builds up the environment that is used to execute the body of p. It begins with the initial, empty environment. Elaboration of the global declarations builds an environment that contains the variable x and the procedures p and q. Next, a new environment is built for the call of q from the main program. In this environment, the procedures p and q are still visible, but the global variable x is hidden by the formal parameter x, which is bound to a fresh location that is initialized with the actual parameter value 4. The call to p from the body of q causes another new environment to be built; but because p has no parameters, this environment is the same as the previous one. This table summarizes the environment seen from the body of procedure p:

Environment for p:	(No new objects)
Environment for q:	Parameter x
Global environment:	Variable x (hidden) Procedure p Procedure q

So the body of p is executed in an environment where p and q are bound to the two procedures, and (crucially) x is bound to the formal parameter of q. Thus the assignment x := 3 in the body of p affects the value of q's formal parameter, not the global x.

The binding rule that is used by our interpreter can be summarized by saying that each applied occurrence of an identifier is bound to the closest declaration in the *dynamic* sequence of nested procedure calls. In the example, the main program calls q and q calls p; in the body of p, the dynamically closest declaration of x is its declaration as a formal parameter of q. This rule is called *dynamic binding*.

Although some programming languages implement dynamic binding (examples are old-fashioned dialects of LISP, and the text-formatting language TEX that is so conspicuously being used to print these notes), most modern languages have another binding rule, *static binding.* Under this binding rule, applied occurrences of identifiers are bound to the declaration in the nearest *textually* enclosing scope. In our example, the environment seen by the body of p does not include the definitions added in calling q, but only the global environment:

Environment for p:	(No new objects)
Global environment:	Variable x Procedure p Procedure q

This means that the applied occurrence of x in the body of p is always bound to the global definition of x.

The static binding rule has many advantages: the most important ones are that it is possible to compile programs into more efficient object code, that it is possible to check programs for type errors once-and-for-all at compile time, and that it is easier to specify the behaviour of procedures and to prove that they meet their specifications. To see why static binding gives more efficient object code, consider our example program again. With dynamic binding, the applied occurrence of x in the body of p might be bound to one definition for one call of p, and to a different one for another call. With static binding, the compiler can work out that this occurrence of x always refers to the global variable x, so it can compile efficient code to access that global variable every time. Because the name x in the body of p always refers to the global x, the compiler can also check that the global x has the right type for the use it is put to by q. Dynamic binding does not share these advantages, because the compiler cannot link each applied occurrence with a unique declaration.

To see why static binding makes reasoning easier, suppose we changed the example program so that q's formal parameter was called y instead of x. If the formal parameter of q is genuinely a local variable, this should not affect the meaning of the program. But with dynamic binding, changing this name does affect the meaning of the program, because it makes the assignment in p refer to the global variable, rather than q's formal parameter.

To implement static binding, we have to arrange that the environment in which the body of each procedure is executed contains exactly the objects that are declared in the *textual* context of the procedure. The mistake in our interpreter was to build the environment for a procedure body starting with the environment of the call; what we need to do instead is begin with the *environment of the procedure declaration.* This means saving that environment as part of the procedure's definition in the environment. The following changes are needed (I've underlined the code that has changed):

- We modify the type *procdef* by adding an environment component:

  type *procdef* = *ident list* ∗ *stmt* ∗ <u>*environment*</u>;;

  This makes the types *procdef* and *environment* mutually recursive.

- We modify the function *declare* so that it saves the current environment as part of the procedure being defined:

  let *declare* (*env*, *s*) =
      function . . .

```
    | ProcDecl (p, formals, body) →
        (define p (ProcDef (formals, body, env)) env, s);;
```

- We modify the function *exec* so that it builds the environment for the procedure body starting with the saved environment of the declaration, not the environment of the call:

  ```
  let rec exec_stmt env s =
    function ...
      | Call (p, es) →
          let (ProcDef (formals, body, defenv)) = lookup p env in
          let actuals = map (eval env s) es in
          let (env′, s′) = fold_left decl_arg (defenv, s)
                                    (combine formals actuals) in
          exec env′ s′ body
      | ...
  ```

  Note that the actual parameters are still evaluated in the environment of the call: otherwise (for example) local variables of the calling procedure could not be used as actual parameters.

These modifications give us static binding, but we also lose something – recursion. The body of a procedure p is executed in the environment of p's definition, extended with definitions of the formal parameters, and this environment does not contain the definition of p itself. To fix this, we need to arrange that the environment stored with p includes p itself. This means creating an infinite or cyclic structure, because the environment has to contain procedure definitions that in turn contain the environment itself. Such cyclic structures can be created in ML only through the use of mutable cells. One way to fix the problem is as follows:

- Include a mutable cell in each *procdef*, to which we can later assign the appropriate environment:

  ```
  type procdef = ident list ∗ stmt ∗ environment ref;;
  ```

- Make *declare* take an extra parameter *er*: the mutable cell in which the environment for procedures will later be stored.

  ```
  let declare er (env, s) =
    function ...
      | ProcDecl (p, formals, body) →
          (define p (ProcDef (formals, body, er)) env, s);;
      | ...
  ```

- In *exec*, add the dereferencing operator ! in the appropriate place to convert the environment reference into an environment.

- In *run*, add code to create a mutable cell *er* (initialized with the empty environment, though that doesn't matter), then build *er* into every procedure definition, and finally assign to *er* the whole environment, procedures and all:

  ```
  let run (Program (globals, main)) =
    let er = ref init_env in
  ```

```
let (env, s) =
  fold_left (declare er) (init_env, init_store) globals in
er := env;
exec env s main;;
```

This creation of a cyclic environment is traditionally called 'tying the knot'. The way the knot is tied here allows mutual recursion, because each procedure is given an environment that includes all of them.

7.3 Nested procedures and local variables

The mechanisms we've built up are actually adequate to handle nested procedures and procedures with local variables without much more effort. All we need to do is introduce into the abstract syntax the concept of a block (containing some declarations and a list of statements), and arrange that both the body of a procedure and the main program be blocks. Then we can define a function *exec_block*, and use it both to execute procedure bodies and to write a new definition of *run*.

```
let rec exec env s =
  function ...
    | Call (p, es) →
        let (ProcDef (formals, body, er)) = lookup p env in
        let actuals = map (eval env s) es in
        let (env′, s′) =
          fold_left decl_arg (!er, s) (combine formals actuals) in
        exec_block env′ s′ body
    | ...

and exec_block env s (Block (locals, body)) =
  let er = ref init_env in
  let (env′, s′) = fold_left (declare er) (env, s) locals in
  er := env′;
  exec env′ s′ body;;

let run (Program b) =
  exec_block init_env init_store b;;
```

Here is an example of a program with nested procedures, written in a language where procedures return results:

```
proc sumpow(x, y, k : integer): integer;
  proc pow(u: integer): integer;
    var i, p: integer;
  begin
    i := 0; p := 1;
    while i < k do
      p := p ∗ u;
      i := i + 1
    end;
    return u
  end;
```

```
    begin
      return pow(x) + pow(y)
    end;
```

In this program, the nesting is important because the inner procedure pow refers to the quantity k, a parameter of the outer procedure. Here is the environment structure seen by the body of pow:

Environment for pow:	Parameter u Variable i Variable p
Environment for sumpow:	Parameter x Parameter y Parameter k Procedure pow
Global environment:	Procedure sumpow

Thus k is accessible from the nested procedure pow because the environment for its body is built on top of the environment for sumpow.

7.4 Summary

We've used source-level interpreters to explore some of the features of subroutine mechanisms. Among the lessons we've learnt are

- that a separation of the notion of *environment* from the notion of *store* is needed if we are to allow procedures with parameters or local variables.
- that the proper base on which to build the environment for a procedure body is the environment of its definition, not the environment of its call.

There are still some other features that we'd like to add: specifically, reference parameters (which allow values to be passed back to the calling procedure) and functional parameters (in which one subroutine can be passed to another one and called from its body). Another very worthwhile addition would be functions that return a value.

We'll leave those big and small refinements for later, and first look at how to implement procedures in a compiler. We do this by extending our abstract machine with a stack that can hold data about active procedures.

Exercises

7.1 Our source-level interpreters model a language with local variables by initializing each fresh variable with zero. Many compilers, however, do not guarantee any initial value for local variables: in C, for example, global variables are guaranteed to be initialized with zero, but local ones are 'randomly' initialized. Adam dislikes this inaccuracy, and builds an interpreter that initializes local variables using a library function $rand : unit \to int$ that returns a random integer each time it is called. Beth wants to build a program that simulates dice throws, and writes the following function:

```
proc throw(): integer;
  var local: integer;
begin
  return (local mod 6) + 1
end
```

She tests the function using Adam's interpreter, and finds that it gives just the results she wants. But when she compiles the program using Carl's compiler, she finds that the dice show all ones. What has gone wrong, and who is to blame?

7.2 [Difficult]. Consider a version of Lab 5's miniBasic language that has static binding and reference parameters, but no recursion: the environment for a procedure is that of its definition, and does not contain the procedure itself. Suppose we also delete while loops from the language. Show that it is still possible to write a procedure fac(n) that sets the global variable f to the factorial of n.

The next few exercises concern implementations of a very simple language infraCalc that has integers as the only data type, with conditionals and recursive functions, but no assignable variables or loops. For example, here us an infraCalc program for computing the twentieth Fibonacci number:

```
let fib(n) =
  if n <= 1 then 1 else fib(n-1) + fib(n-2);

fib(20);
```

Here is an abstract syntax for the infraCalc language:

type *prog* = *Program* **of** *decl list* ∗ *expr*

and *decl* = *FuncDecl* **of** *ident* ∗ *ident list* ∗ *expr*

and *expr* =
  Number **of** *int*
  | *Variable* **of** *ident*
  | *Binop* **of** *op* ∗ *expr* ∗ *expr*
  | *Monop* **of** *op* ∗ *expr*
  | *Call* **of** *ident* ∗ *expr list*
  | *IfExpr* **of** *expr* ∗ *expr* ∗ *expr*;;

7.3 Using a version of environments in which variable identifiers are bound to integers (not mutable cells), and function identifiers are bound to closures of type *ident list* ∗ *expr* ∗ *environment ref*, construct a source-level interpreter for infraCalc: that is, a function $run : prog \to int$ that maps each program to the integer value of its final expression.

7.4 It was said on page 102 that a single mapping from identifiers to values was not enough to explain programming languages with parameters and recursion, yet your answer to question 7.3 does precisely that. How can this be?

7.5 What would happen if you used your interpreter to run the following program?

```
let forever() = forever();

let fst(x, y) = x;

fst(3, forever());
```

7.6 Modify your interpreter so that parameters are passed by name instead of by value. In this version of the interpreter, variable identifiers are bound to objects of type *expr* ∗ *environment*, and the value of a variable is obtained by evaluating the expression in the environment. In a function call, the formal parameters become bound to the *unevaluated* actual parameters, and evaluation of the parameters is delayed until they are actually used inside the function body. What happens if you run the program of question (5) using this new interpreter?

7.7 Consider programs that terminate both when parameters are passed by name and when they are passed by value. Explain briefly why a program can take much longer to run when parameters are passed by name than when they are passed by value.

Let the *abstract running time* of a program in our language be the number of calls to *eval* made during execution of the program. Give an example of a program containing a constant N that has an abstract running times when parameters are passed by name that is greater than when they are passed by value by a factor that is exponential in N.

7.8 An alternative implementation of name parameters uses textual substitution. Define a function *subst* : (*ident* ∗ *expr*) *list* → *expr* → *expr* such that

$$subst\ [(x_1, e_1); \ldots; (x_n, e_n)]\ e$$

is the result of simultaneously substituting e_1 for x_1 and ... and e_n for x_n in the expression e. (Make your function substitute for variable names but not procedure names.) Use this function to write a new interpreter for INFRACALC in which only procedures appear in the environment, and each is bound to an object of type *ident list* ∗ *expr*, with parameter passing implemented by textual substitution. Does your implementation have static or dynamic binding?

7.9 The following type definition describes the abstract syntax of a simple programming language:

```
type expr =
      Number of int
    | Variable of ident
    | Binop of op * expr * expr
    | IfExpr of expr * expr * expr
    | Call of ident * expr list
    | Let of ident * expr * expr
    | Func of ident * ident list * expr * expr;;
```

An expression in the language may have an integer or a function as its value. The test of a conditional expression is assumed to have an integer as its value, and it is considered as true if the value is non-zero. The abstract syntax *Let* (x, e_1, e_2) corresponds to the definition of a local variable, as in let x = e_1 in e_2. The abstract syntax *Func* (f, xs, e_1, e_2) corresponds to the definition of a local function, as in func $f(x_1,\ x_2)$ = e_1 in e_2, where $xs = [x_1;\ x_2]$ is the list of formal parameters. The language has static binding and parameters are passed by value; functions may be recursive.

(a) Define appropriate types *def* to represent the definitions of names in the language, and *value* to represent values of expressions.

(b) Assuming an appropriate data type of environments (which you need not implement), give a source-level interpreter for the language.

(c) Consider the following program, which computes 4^4:

```
func twice(f) = (func ff(x) = f(f(x)) in ff) in
func square(x) = x * x in
let fourth = twice(square) in
fourth(4)
```

List the environments that are created during the execution of this program, showing the names that are defined in each, together with their definitions.

(d) Briefly explain why this programming language does not have an implementation in which activation records are allocated from a stack.

Chapter 8

Compiling procedures

In this chapter, we describe how to implement procedures using an abstract machine with stack-based storage. The first step is to extend our abstract machine with instructions that implement subroutine linkage, and modify other instructions so that they allow access to both local and global variables. Then we explain how to compile simple procedures (with value parameters and no nesting) for this machine. Later sections of the chapter are devoted to extending this simple implementation with the more exotic features described in the preceding chapter: reference parameters, nested procedures, and procedures that take other procedures as arguments.

8.1 Extending the abstract machine

In order to implement procedures, we must extend our abstract machine to support the stack operations that are needed. We will add six new instructions:

- *CALL n* is used to translate procedure calls: it represents a call to the procedure with *n* parameters. The address of the procedure itself and the *n* parameters are taken from the expression stack.
- *PUSHL L* is used as part of the translation of a procedure call. It loads onto the stack the address associated with label *L*.
- *RETURN* is used to translate return statements.
- *FRAME s* is used as the first instruction of a procedure. It allocates a stack frame of size *s* for local variables of the procedure.
- *ENDPROC* marks the end of a procedure. It will never be executed, unless the program contains an error that makes it possible to reach the end of a procedure without executing a return statement.
- *LOCAL* (*d*, *o*) is used like *GLOBAL a* to push the address of a variable. The variable is identified by a *pair* of numbers (*d*, *o*). The number *d* is the *depth* of the variable being accessed: for now, it will always be zero, indicating a local variable of the current procedure; other possible values will be introduced later. The offset *o* may be positive or negative:

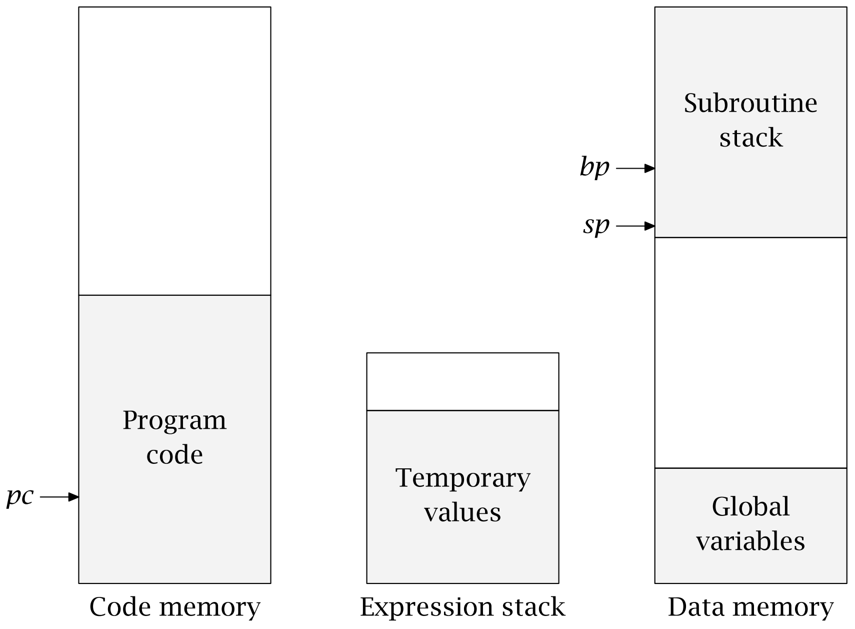

Figure 8.1: *Memory model*

after finding the storage area indicated by *d*, we add *o* to get the actual address that is pushed on the stack.

Many of these instructions have rather complex actions, and it is likely that on a real machine each would be represented by a short sequences of machine instructions.

To explain more precisely the effect of the *Call*, *Frame* and *Return* instructions, we need to refine slightly our picture of the abstract machine, as shown in Figure 8.1. The change here is that the data memory now has a definite layout, with global variables kept at low addresses in the data memory, and data about active procedures kept in the high part of memory as a stack that grows downwards. To the *pc* register, we add two new ones: *sp*, which always points to the lowest location of the subroutine stack that is in use, and *bp*, which always points to some place inside the stack. Local variables of a procedure are accessed by adding offsets to the contents of the *bp* register. All three of these registers – *pc*, *sp* and *bp* – are what we might call 'special-purpose' registers, always used to hold the same specific quantity. In place of the 'general-purpose' registers that most real computers provide for evaluating expressions, we will continue to use an evaluation stack.

An expanded view of the block of memory – the *stack frame* – associated with a single activation of a procedure is shown in Figure 8.2. The *bp* register points to a fixed location in the frame, the first location of a three-word *frame head* that includes the return address, i.e., the address of the next instruction to be executed after the procedure returns. The *dynamic link* is the value of the *bp* register before this stack frame was created: it is used to restore the old value of *bp* when the procedure returns. The other value in the frame head is the *static link*, which we shall use later to implement nested procedures; at present, the static link will always be zero.

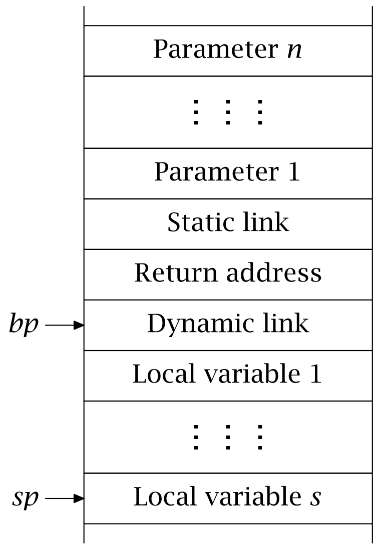

Figure 8.2: *Layout of a stack frame*

The parameters of the procedure call are allocated space above the frame head, and the local variables below it, so that the parameters are accessible by adding positive offsets to *bp*, and the local variables are accessible at negative offsets from *bp*. The parameters appear in reverse order: this makes things marginally more convenient in the compiler, because the first parameter is always at the same offset from *bp*. In the figure, the procedure has n parameters and s local variables.

The effects of the new instructions *Call*, *Frame* and *Return* are as follows:

- The instruction *Call* n expects to find $n + 2$ items on the expression stack. On top of the stack, it expects to find the address of the procedure that is to be called: this value might have been put there by an immediately preceding *Pushl* instruction. Just beneath the procedure address is the value that will be stored as the static link: at present it is always zero. Beneath the static link are the n parameters of the procedure.

  The *Call* instruction moves the static link and the n parameters onto the subroutine stack, and after them (at the next lower address, since the stack grows downwards) it puts the current contents of the *pc* register, which is the address in code memory of the instruction following the *Call* instruction. Then it loads the *pc* register with the address of the procedure. The overall effect of the *Call* instruction can be seen by comparing Figure 8.3 with Figure 8.4.

- The instruction *Frame* s acts as follows: first, the value of the *bp* register is pushed onto the subroutine stack as the dynamic link of the procedure that has just been called. Next, the *bp* register is set to the current value of the stack pointer, and finally the *sp* register is decreased by s, thereby allocating space for the local variables of the procedure. Together, the *Call* and *Frame* instructions have the effect of setting

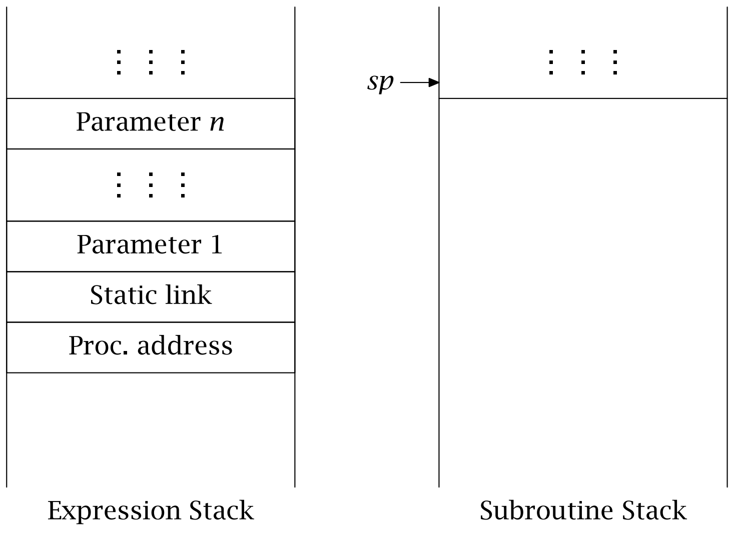

Figure 8.3: *Before the* CALL *instruction*

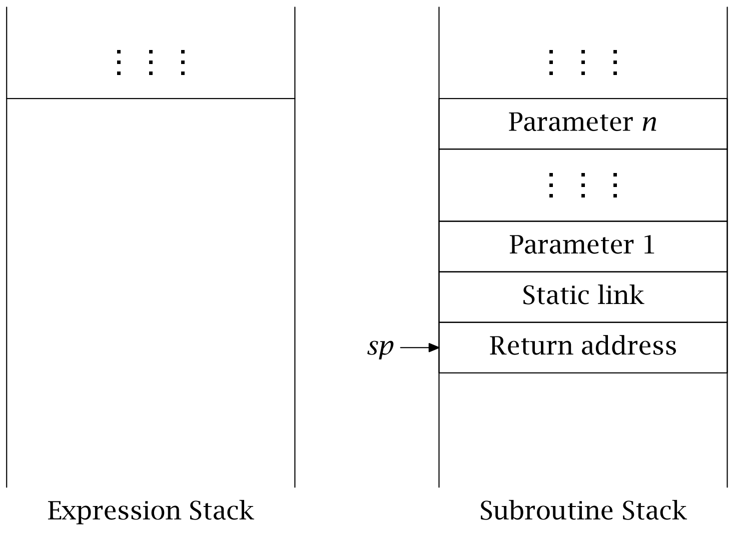

Figure 8.4: *After the* CALL *instruction*

```
let pc = ref 0;;                        (* Program counter *)
let sp = ref 0;;                        (* Proc stack pointer *)
let bp = ref 0;;                        (* Frame base *)
let stack = ref [ ];;                   (* Expression stack *)
let mem = Array.create memsize 0;;      (* Data memory *)

let push v = stack := v :: !stack;;
let pop ( ) = let v = hd (!stack) in stack := tl !stack; v;;

(* arg_count – find arg count of returning procedure *)
let arg_count ( ) =
  match prog.(!pc−1) with
      CALL n → n
    | _ → failwith "arg_count";;

(* addr – compute address of local variable *)
let addr d o = !bp + o;;

(* exec_inst – execute an instruction *)
let exec_inst =
  function
      CONST x → push x
    | LOCAL (d, o) → push (addr d o)
    | GLOBAL n → push n
    | PUSHL l → push (find_lab l)
    | LOAD → push mem.(pop ( ))
    | STORE → let a = pop ( ) in mem.(a) ← pop ( )
    | CALL n → sp := !sp − n−2; mem.(!sp) ← !pc; pc := pop ( );
                    for i = 0 to n do mem.(!sp + i + 1) ← pop ( ) done
    | FRAME n → sp := !sp−1; mem.(!sp) ← !bp; bp := !sp; sp := !sp − n
    | RETURN → sp := !bp; bp := mem.(!sp); pc := mem.(!sp + 1);
                    sp := !sp + arg_count ( ) + 2
    | ENDPROC → failwith "procedure failed to return"
    | STOP → raise Finish
    | MONOP w → push (do_monop w (pop ( )))
    | BINOP w → let b = pop ( ) in let a = pop ( ) in
                    push (do_binop w a b)
    | JUMP l → pc := find_lab l
    | JUMPB (b, l) → if b = (pop ( ) ≠ 0) then pc := find_lab l
    | _ → failwith "bad instruction";;
```

Figure 8.5: *Simulator with new instructions*

up the stack frame structure shown in Figure 8.2.

- The instruction *Return* destroys the frame that was created by the *Call* and *Frame* instructions, and returns to the calling program. The steps are as follows: first, *sp* is reset to *bp*. Then the old value of *bp* and the saved value of *pc* are restored by popping them off the subroutine stack one after the other. Finally, the quantity $n + 1$ is then added to *sp*, where *n* is the number of parameters that was specified in the *Call* instruction when the procedure was activated. This removes the static link and the *n* parameters from the subroutine stack. Execution continues from with the instruction following the *Call* instruction.

A simulator that implements these new instructions is shown in Figure 8.5.

8.2 Semantic analysis for procedures

Using the new instructions, we will shortly design code sequences for procedure calls, and for procedures themselves. First, however, we need to extend the semantic analyser so that it annotates the program with context information that the code generator will need, allocating space in different ways for:

- global variables (which grow upwards from the bottom of memory),
- formal parameters (which are located at increasing positive offsets from the frame base of a procedure), and
- local variables (which are located at increasing negative offsets from the frame base).

The semantic analyser should record for each name what kind of object (variable or formal parameter) it is, and two integers: the *nesting level* and the *offset* of the object. Global variables are at nesting level 0, and their offsets are their absolute addresses; local variables and parameters are at nesting level 1, and their offsets are the (positive or negative) distances from the frame base.

For simplicity, we will temporarily return to a language that has only integer variables, and has procedures that return an integer result, so that a procedure call will be an integer expression rather than a statement. (We will relax some of these restrictions later.) Our semantic analyser will attach a record of type *def* to each applied occurrence, where *def* is defined as follows:

```
(* def – definitions in environment *)
type def =
  { d_tag : ident;             (* Name *)
    d_level : int;             (* Nesting level *)
    d_guts : def_guts }        (* Definition *)

and def_guts =
    VarDef of int              (* Variable (offset) *)
  | ProcDef of codelab * int;;  (* Procedure (label, nparams) *)
```

Each definition has a *d_tag* field that contains its name, and a *d_level* field that contains the nesting level of the object. The *d_guts* field indicates whether

```
var a, b;

proc scale(x);
  var y;
begin
  y := a * x;
  return y + b
end;

begin (* Main program *)
  print scale(4)
end.
```

Figure 8.6: *Program with co-ordinates*

the object is: an (integer) variable or a procedure. For a variable, the *d_guts* field will be *VarDef o*, where *o* is the offset of the storage for the variable, either in the stack frame of its procedure, or in the global storage area. For a procedure, the *d_guts* field will be *ProcDef* (*lab*, *np*), where *lab* is a label we will attach to the code for the procedure, and *np* is the number of parameters. Because all variables are integers, and each procedure returns an integer result, we need not store a type for as part of the definition of an object.

Using the same approach as in Chapter 6, we can build a semantic analyser that annotates a program with these definitions. Unlike the analyser discussed there, this one must build a different environment for the body of each procedure, so that each identifier is associated with the appropriate declaration. The semantic analyser should issue an error message if an identifier that has been declared as a variable is used as the procedure in a procedure call, or if a procedure is used as if it were a variable. It can also check that each procedure call contains the correct number of arguments for the procedure being called.

Figure 8.6 illustrates a syntax we might use for programs in this language. The semantic analyser will label each variable with its definition, so the first statement in the body of scale will look something like

$$\mathsf{y}^{(1,-1)} := \mathsf{a}^{(0,0)} * \mathsf{x}^{(1,3)};$$

(I have labelled each variable with its 'co-ordinates' (l, o), where l is the level and o is the offset.) The global variables a and b are at level 0, and are allocated successive locations in the global storage area. There are two variables local to procedure scale: the formal parameter x, and the local variable y, both of them with a level of 1. According to the layout for a stack frame shown in Figure 8.2, parameters exist above the frame head, starting at offset 3, and local variables are below it, starting at offset −1. This explains the offsets chosen for x and y.

8.3 Compiling code for procedures

We now turn to the code that should be generated for procedure calls and procedure definitions. For a simple procedure with value parameters, a typical call looks like

$P(e_1, e_2, \ldots, e_n)$,

where P is the name of a procedure, and $e_1, e_2, \ldots, e_n$ are expressions. The semantic analyser will associate the name P with a definition that includes the label L_P that should be used for calling it. The procedure call can be translated as follows:

```
⟨Code for $e_n$⟩
  ...
⟨Code for $e_2$⟩
⟨Code for $e_1$⟩
CONST 0
PUSHL $L_P$
CALL $n$
```

This code first evaluates the parameters in reverse order. The net effect of the code for each parameter is to push its value onto the expression stack, after the other parameters that have already been evaluated. The instruction *CONST* 0 adds the static link to the expression stack; we use a zero for the present. Finally, the value of the label L_P is pushed with a *PUSHL* instruction. The contents of the expression stack now match Figure 8.2. The *CALL* instruction copies this information into the new stack frame into the subroutine stack and branches to the beginning of the procedure.

The code for procedure P will look like this:

```
LABEL $L_P$
FRAME $s$
⟨Code for the procedure body ⟩
RETURN
ENDPROC
```

After the label that is used to call the procedure, this code begins with a *FRAME* instruction that finishes the creation of the stack frame. Everything is now set up for execution of the procedure body, which can access parameters and local variables at fixed offsets from *bp*, and global variables at their absolute addresses. The body code is followed by a *RETURN* instruction that discards the frame and returns to the caller, and finally an *ENDPROC* instruction (which should never be reached). Any `return` statements in the procedure body can also be translated as the instruction *RETURN*.

All variable references in the procedure body are annotated by the semantic analyser with the nesting level (0 or 1) of the variable and its offset. It is easy to translate this into a *LOCAL* or *GLOBAL* instruction that specifies the same offset.

A function can return a result to its caller by leaving the result on the expression stack. This is achieved by translating the statement `return` E into the code sequence:

```
⟨Code to evaluate $E$⟩
RETURN
```

The net effect of executing the code for E will be to leave its value on top of the expression stack, and the *RETURN* instruction does not affect the contents of the stack, so the value is there to be used by the caller.

Let's apply this scheme to the program shown in Figure 8.6. The procedure scale has a single local variable, so its code begins with the instruction *FRAME* 1 that allocates one word of space:

```
LABEL $L_{scale}$
FRAME 1.
```

Next comes code for the statement y := a ∗ x, which uses the global variable a, the local y, and the parameter x. The code for this is

```
GLOBAL 0/LOAD
LOCAL 3/LOAD
BINOP Times
LOCAL −1/STORE.
```

The next statement is return y + b; for this, we generate code to push the value of y + b onto the expression stack, then add a *RETURN* instruction:

```
LOCAL −1/LOAD
GLOBAL 1/LOAD
BINOP Plus
RETURN
```

The code for scale finishes with an *ENDPROC* instruction:

```
ENDPROC
```

Now let's look at the code for the main program, which begins with the code for scale(4):

```
CONST 4
CONST 0
PUSHL $L_{scale}$
CALL 1
```

This code pushes the argument 4, the static link (zero for now), and the label for scale; then there is a *CALL* instruction specifying one parameter. The main program finishes with a *PRINT* instruction and a *STOP* instruction:

```
PRINT
STOP
```

This completes our description of how to compile code for simple procedures. Our implementation implements recursion naturally, because a frame for each invocation of a procedure is allocated from a stack, and there is no difference between allocating another stack frame for the same procedure and allocating a frame for a different procedure. Static binding is implemented just as naturally by having the semantic analyser use an environment containing global and local variables to annotate each applied occurrence with the nesting level and offset from its definition. We now turn to refinements of the simple procedure mechanism outlined here.

8.4 Value and reference parameters

The first refinement is to add reference parameters, like the var parameters of PASCAL. These require actual parameters that are variables, and assignments to the formal parameters in the procedure body affect the values of the actual parameters. Thus reference parameters can be used to return results from a procedure.

Reference parameters can be implemented by passing the *address* of the actual parameter to the procedure, instead of its value. We rely on semantic analysis to annotate the abstract syntax tree in such a way that value and reference parameters can be distinguished both when we are compiling code for a procedure body, and when we are compiling code for a call to the procedure.

To compile code for a procedure body, we need to be able to compile each variable or parameter in two ways: either to load its value onto the stack, or to load its address. For example, an assignment statement x := y is compiled as

```
⟨Load address of x⟩
⟨Load value of y⟩
STORE
```

For value parameters and variables, we already know how to load their values and their addresses onto the stack: we use an *LOCAL* instruction to get the address, and follow it with a *LOAD* instruction if we want the value instead.

For reference parameters, the value that is passed as a parameter is in fact the address of the actual parameter: so to load the address of a reference parameter, we use an *LOCAL* instruction followed by a *LOAD* instruction. This will copy onto the stack the value that was passed by the calling procedure, and this value is the address of the actual parameter. To load the *value* of a reference parameter, we need to find the contents of this address, so we generate a *LOCAL* instruction followed by *two LOAD* instructions.

This completes the additions that are needed to the translation of procedure bodies; what about calls to procedures with reference parameters? It's quite simple: for value parameters, we continue to generate code that loads the value of the actual parameter expression onto the stack, exactly as if the parameter expression had appeared on the right hand side of an assignment statement. For reference parameters, we compile code to load the *address* of the parameter onto the stack, as if it had appeared on the *left-hand side* in an assignment. Thus, if an ordinary variable is passed as a reference parameter, a *LOCAL* or *GLOBAL* instruction is generated. If a reference parameter of one procedure is passed to as a reference parameter to another, an *LOCAL* instruction and a *LOAD* instruction are generated. These instructions have the effect of taking a copy of the value that was passed to the first procedure (i.e., the address of the actual parameter) and passing it to the second procedure.

Here is a (completely artificial) example program that illustrates all the possible combinations:

```
proc p(x: integer; var y: integer);
begin ... end;

proc q(a: integer; var b: integer);
begin
  p(a, a);
  p(b, b)
end;
```

The following code is generated for q:

```
LABEL L₂                  (* Prelude for q *)
FRAME 0

LOCAL (0, 3)              (* Call p(a, a): Second parameter *)
LOCAL (0, 3)              (* First parameter *)
LOAD
CONST 0                   (* Static link *)
PUSHL L₁                  (* Address of p *)
CALL 2

LOCAL (0, 4)              (* Call p(b, b): Second parameter *)
LOAD
LOCAL (0, 4)              (* First parameter *)
LOAD
LOAD
CONST 0                   (* Static link *)
PUSHL L₁                  (* Address of p *)
CALL 2

RETURN                    (* Postlude for q *)
ENDPROC
```

8.5 Nested procedures

The next stage of sophistication is to allow one procedure to be nested inside another, so that the inner procedure has access to the parameters and local variables of the outer procedure in addition to its own. With static binding, the semantic analysis of such procedures is not difficult, because the environment that should be used in checking each procedure body can be determined statically at compile time. We can arrange that the definition of each object – procedure, parameter, or variable – records the nesting level to which it belongs, so that the code generator has the co-ordinates of each occurrence of a variable as a (level, offset) pair. These co-ordinates uniquely identify the run-time object to which the occurrence refers: in the body of a level 3 procedure, for example, a reference to a level 2 object must refer to an object declared in the textually enclosing procedure.

At run-time, the object is located in the frame of the latest activation of this procedure. For example, in the following program (which we used earlier as an example of nested procedures), the occurrence of k in the while test will be marked by the semantic analyser as belonging to level 1, and it occurs in the procedure pow that operates at level 2. I've labelled each identifier with a superscript that denotes its level number:

```
proc sumpow$^1$(x, y, k : integer): integer;
  proc pow$^2$(u: integer): integer;
    var i, p: integer;
  begin
    i$^2$ := 0; p$^2$ := 1;
    while i$^2$ < k$^1$ do
      p$^2$ := p$^2$ * u$^2$;
      $i^2$ := $i^2$ + 1
    end;
    return $u^2$
  end;
begin
  return pow($x^1$) + pow($y^1$)
end;
```

The pow procedure can only be called from sumpow, so whenever a call of pow is active, there must be an enclosing activation of sumpow. It is in the stack frame of this activation of sumpow that we find the variable k.

The problem now is to arrange for the adress of each variable to be calcualted from its (level, offset) co-ordinates. This will have to be done (at least partly) at run time, because the compiler cannot know the address of the stack frames in which the variables live. A natural first thought is to use the chain of pointers that links each stack frame to the one beneath it on the stack: in the example above, the stack frame below the one for pow will always be the frame for the enclosing activation of sumpow. This does not always work, however: for example, in the following program, one procedure at level 2 calls another one, also at level 2:

```
proc f(x, y: integer): integer;

  proc g(u: integer): integer;
  begin return u+y end;

  proc h(z: integer): integer;
  begin return g(z)-x end;

begin
  return g(x div y)+h(x mod y)
end;
```

When g is called from h, there will be both an activation of h and an activation of f below it on the stack, and it is the stack frame for f that contains the parameter y that is referred to by g: the dynamic link in the frame for g does not lead directly to the frame of the statically enclosing procedure f. Also, the number of frames between the frame for g and the frame for f can vary from call to call: g is also called directly from the body of f, and in that case, no frames come between them on the stack.

The problem here is exactly the one that was revealed in Chapter 7 when we considered nested procedures: it is the distinction between static and dynamic binding. Plainly, something other than the chain of dynamic links is needed, and that's where the *static link* enters the story. When a procedure at nesting level k is called, we will pass as its static link the frame address for the most recent activation of its enclosing procedure (at level $k-1$). This link is stored, as we have seen, at a fixed place in the stack frame for the new activation; it is therefore possible to follow a chain of static links to find the stack frame for the most recent activation of any of the $k-1$ enclosing procedures. In our example, where f calls h and h calls g, the dynamic link of g points to the activation of h, and that of h points to the activation of f; this is shown by the black arrows leaving the upper left-hand corner of each activation in Figure 8.7. However, the static links of both h and g point to the

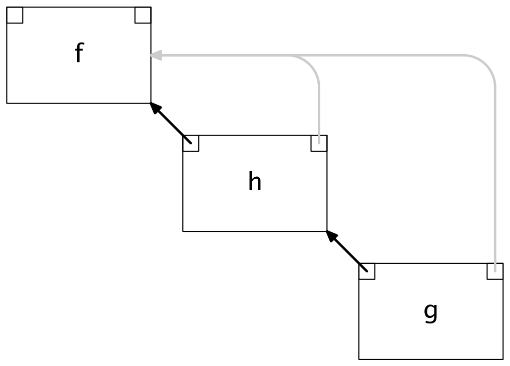

Figure 8.7: *Static and dynamic links*

enclosing activation of f, as is shown by the grey arrows leaving the upper left corner of each activation.

If we can arrange for the static chain to be set up like this, then we can find the address for any variable or parameter reference as follows. Suppose the reference is to an object at nesting level j, and it occurs in the body of a procedure P at nesting depth $k \geq j$.

- If $j = k$, then the object is local to P and can be addressed at an offset from *bp*.
- If $j = 0$, then the object is global, and has a known absolute address.
- If $0 < j < k$, then the object is in an *intermediate frame*, and can be found by following static links $k - j$ times, then taking an offset from base of the frame that results.

Experience shows that most references are to variables that are either local to the current procedure or global to the whole program; both are immediately accessible in this scheme. We expect relatively few references to variables in intermediate frames. Also, the nesting depth of typical programs is rather small, so the number of static links that must be followed in the worst case is not too large. The number of links that is followed for a particular variable reference is fixed, so we can generate a fixed instruction in our intermediate code, or compile a fixed sequence of actual machine instructions, to access each variable.

The picture becomes slightly more complicated when recursion is used, but the scheme based on static links is still adequate. For example, here is a function power that uses a nested, recursive function to compute x to the power k. The inner function pow makes a non-local reference to the parameter x of power.

```
proc power(x, k : integer): integer;
  proc pow(j: integer): integer;
  begin
    if j = 0 then
      return 1
    else
      return x * pow(j-1)
    end
  end;
begin
  return pow(k)
end;
```

Because of the recursion, there may be many stack frames for activations of pow between the one on top of the stack and the frame for the enclosing activation of power.

To add addressing for intermediate frames to our simulator, we generalize the *Local* instruction by allowing a *depth* field other than 0. Any non-zero value means "follow static links *depth* times, then add the offset". This meaning is implemented by extending the function *addr* of Figure 8.5:

```
let addr d o =
  let rec base n b =
    if n = 0 then b else base (n−1) mem.(b + 2) in
  base d !bp + o;;
```

The magic constant 2 used here in the expression *mem.*($b + 2$) reflects the fact that the static link is stored at that offset from the base of each stack frame (see Figure 8.2).

Generating *Local* and *Global* instructions with the right depth fields in them is easy: if the nesting level j of the object is 0, then we use a *Global* instruction; otherwise we use a *Local* instruction with depth $k - j$, where k is the nesting level of the procedure body being compiled. Using relative depths like this means that our machine can operate without keeping track of the nesting level of the procedure body it is executing.

The remaining problem is to establish the static link when a procedure is called. The essential insight here is that the static link that should be passed to a procedure can be found somewhere in the static chain of the calling procedure. How it is found depends on the nesting level k of the calling procedure P and the nesting level j of the procedure Q that is called:

- If $j = 1$, then the Q is a global procedure. By convention, we set its static link to zero. We can load the static link with the instruction *Const* 0.
- If $j = k + 1$, then Q is one of the local procedures nested inside P, and its static link is the base address of the calling frame: this can be loaded with the instruction *Local* (0, 0).
- If $1 < j = k$, then the Q is one of the siblings of the P, and both are nested inside the same outer procedure. They should also share the same static link, which is loaded onto the stack with the instruction *Local* (1, 0).
- If $1 < j < k$, then Q is declared in another procedure R that is an ancestor of P: otherwise, the Q would not be in scope in the body of P. The base address for the stack frame of the enclosing procedure can be loaded with the instruction *Local* ($k - j + 1$, 0).

Actually, the only real distinction here is between between the case $j = 1$ and the others, since the other all follow the formula *Local* ($k - j + 1$, 0).

A concrete example: here's the code we produce for the f–g–h example:

```
LABEL L2               (* Procedure g *)
FRAME 0
LOCAL (0, 3)           (* Local u *)
LOAD
LOCAL (1, 4)           (* Nonlocal y *)
LOAD
BINOP Plus
RETURN
ENDPROC

LABEL L3               (* Procedure h *)
FRAME 0
LOCAL (0, 3)           (* Local z *)
LOAD
LOCAL (1, 0)           (* Static link *)
PUSHL L2               (* Address of g *)
CALL 1
LOCAL (1, 3)           (* Nonlocal x *)
LOAD
BINOP Minus
RETURN 1 1
ENDPROC

LABEL L1               (* Procedure f *)
FRAME 0
LOCAL (0, 3)           (* Local x *)
LOAD
LOCAL (0, 4)           (* Local y *)
LOAD
BINOP Div
LOCAL (0, 0)           (* Static link *)
PUSHL L2               (* Address of g *)
CALL 1
LOCAL (0, 3)           (* Local x *)
LOAD
LOCAL (0, 4)           (* Local y *)
LOAD
BINOP Mod
LOCAL (0, 0)           (* Static link *)
PUSHL L3               (* Address of h *)
CALL 1
BINOP Plus
RETURN 2 1
ENDPROC
```

8.6 Closures and functional parameters

The final feature we will discuss is the idea of letting a procedure P take another procedure Q as a parameter, so that Q can be called from within P.

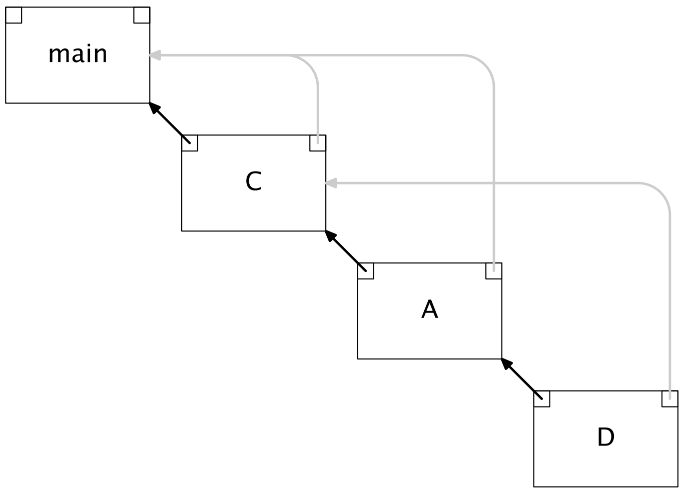

Figure 8.8: *Static and dynamic links with functional parameters*

Figure 7.1 on page 99 shows an example program where this feature is used. For simplicity, let us consider this smaller example:

```
proc A(proc f(x: integer));
begin return f(3) end;

proc B(x: integer);
begin return x+1 end;

proc C(u: integer);
  proc D(v: integer);
  begin return u + v end;
begin
  return A(D)
end;

begin print A(B); print C(2) end.
```

Here, procedure A takes as its parameter another procedure f, which it applies to the parameter 3, returning whatever result is returned by f. In the main program, A is first applied to a procedure B which returns one more than its parameter: A will pass B the parameter 3, and return the result 4, which is then printed.

Functional parameters interact with nested procedures and static binding, as the example in Figure 7.1 shows. A simpler example of this is contained in the program above, for procedure C calls A, passing it as parameter a local procedure D that refers both to its own parameter v and to the parameter u of C. C is called from the main program with parameter 2, and A will call D with parameter 3, so D returns 2 + 3 = 5, and that is the second number printed. The storage layout when D is running is shown in Figure 8.8. In order for this to happen, D must have access to the frame for C, even when it is called from within A.

This means that a functional parameter must consist of more than the just the address of the code to jump to when the procedure is called: we also need the static link to pass in the call. This pair of values – a code address and a static link representing an environment – are called a *closure*. In order to pass a procedure R as a functional parameter to another procedure P, we push these two values on the evaluation stack. The code to do this is exactly the same as the code needed to push the static link and code address on the stack in a call to P:

```
LOCAL $(k - j + 1, 0)$ or CONST 0
PUSHL $L_R$
```

where k is the level of the current procedure, j is the level of procedure P, and p is the label for P. We then count the two words as two parameters in the instruction that calls Q.

In the call A(B) in the example, procedure B is global and expects a static link of zero; A expects a static link of zero also. So the code is

```
CONST 0
PUSHL $L_B$
CONST 0
PUSHL $L_A$
CALL 2
```

The first two instructions push the static link and code address for the functional parameter B. The next two push the static link and code address of A, as usual in a procedure call, and the final instruction calls A, passing two words of parameter information.

Procedure C of the example contains a call A(D) in which A is applied to a local procedure D. The static link for D points to the frame for C, and can be pushed on the stack with the instruction *LOCAL* (0, 0); in our notation, k is 1 and j is 2, so $k - j + 1$ is 0. Thus the call A(D) corresponds to the code

```
LOCAL (0, 0)
PUSHL $L_D$
CONST 0
PUSHL $L_A$
CALL 2
```

Inside a procedure P with a functional parameter Q, we will wish to use the name Q to call the procedure that has been supplied as the parameter. The call follows the same pattern as any other procedure call, except that instead of explicit instructions to push the static link and code address, the values that were passed as the functional parameter are used. Thus the call $Q(e_1, e_2, \ldots, e_n)$ is compiled into

```
⟨Code for $e_n$⟩
  ...
⟨Code for $e_2$⟩
⟨Code for $e_1$⟩
LOCAL $(0, o + 1)$/LOAD
LOCAL $(0, o)$/LOAD
CALL $n$
```

where o is the offset of the formal parameter Q.

In the example, the call f(3) becomes

```
CONST 3
LOCAL 4/LOAD
LOCAL 3/LOAD
CALL 1
```

Exercises

8.1 Show the Keiko code for the following program, explaining the purpose of each instruction.

```
proc double(x: integer): integer;
begin
  return x + x
end;

proc apply3(proc f(x:integer): integer): integer;
begin
  return f(3)
end;

begin
  print_num(apply3(double));
  newline()
end.
```

8.2 Here is a procedure that combines nesting and recursion:

```
proc flip(x: integer): integer;
  proc flop(y: integer): integer;
  begin
    if y = 0 then return 1 else return flip(y-1) + x end
  end;
begin
  if x = 0 then return 1 else return 2 * flop(x-1) end
end;
```

(a) Copy out the program text, annotating each applied occurrence with its level number.

(b) If the main program contains the call flip(4), use the style of Figures 8.7 and 8.8 to show the layout of the stack with static and dynamic links at the point where procedure calls are most deeply nested.

8.3 The following PICOPASCAL program is written in what is called 'continuation-passing style':

```
proc fac(n: integer;
          proc k(r: integer): integer): integer;
  proc k1(r: integer): integer;
  begin
    return k(n * r)
  end;
```

```
begin
  if n = 0 then
    return k(1)
  else
    return fac(n-1, k1)
  end
end;

proc id(r: integer): integer;
begin
  return r
end;

begin
  print_num(fac(3, id));
  newline()
end.
```

When this program runs, it eventually makes a call to id. Draw a diagram of the stack layout at that point, showing the static and dynamic links.

Appendix A

Library Implementation

A.1 Module *Btree*

A *B*-tree may be empty or a leaf (containing an argument of the mapping it represents, together with the corresponding value of the mapping), or it may consist of a node with either 2 or 3 subtrees

```
type (α, β) btree =
    Empty
  | Leaf of α ∗ β
  | Deleted of α
  | Node2 of α ∗ (α, β) btree ∗ (α, β) btree
  | Node3 of α ∗ (α, β) btree ∗ (α, β) btree ∗ (α, β) btree
  | Split of (α, β) btree ∗ (α, β) btree;;
```

The tree *Empty* represents the empty mapping, and a leaf *Leaf* (x, y) represents a mapping m that is defined only for the argument x, with $m(x) = y$. A node $Node_3$ (u, t_1, t_2, t_3) contains the three subtrees t_1, t_2 and t_3. In a well-formed *B*-tree, the trees t_1, t_2 and t_3 are all non-empty, the keys in tree t_1 are less than the keys in tree t_2, and these keys are in turn less than the keys in tree t_3; the value u is the smallest key present in the tree. Similar remarks apply to a node $Node_2$ (x, t_1, t_2), except that it has two subtrees rather than three. Figure A.1 shows a *B*-tree that represents the mapping

$$\{1 \mapsto 31, 2 \mapsto 41, 4 \mapsto 26, 5 \mapsto 53, 6 \mapsto 58\}.$$

Two other kinds of tree are used in our implementation. Trees of the form *Deleted* x are used after some leaves in a tree have been deleted, and a tree *Split* (t_1, t_2) exists only during the process of adding a new value to a tree: it represents a tree that has split into two halves when the new value has been added. The two halves will be added as children of the next node up the tree – unless that node needs to split as well.

We use *B*-trees as our representation of mappings:

```
type (α, β) mapping = (α, β) btree;;
```

The information in each node is sufficient to write an implementation of *find* that searches a single path from the root of a *B*-tree to the desired leaf, if it

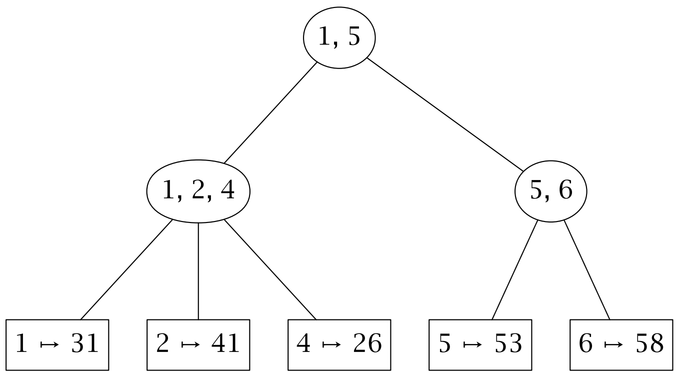

Figure A.1: *B-tree representation of a mapping*

exists:

```
let rec find x =
  function
      (Empty | Deleted _) → raise Not_found
    | Leaf (p, q) → if x = p then q else raise Not_found
    | Node2 (m, t1, t2) →
        if x < minkey t2 then find x t1
        else find x t2
    | Node3 (m, t1, t2, t3) →
        if x < minkey t2 then find x t1
        else if x < minkey t3 then find x t2
        else find x t3
    | _ → failwith "Btree.find";;
```

The helper function *minkey* returns the smallest key in any non-empty tree; thanks to the extra information held in nodes, this can be done in constant time.

```
let minkey =
  function
      Leaf (x, y) → x
    | Deleted x → x
    | Node2 (m, _, _) → m
    | Node3 (m, _, _, _) → m
    | _ → failwith "Btree.minkey";;
```

We have been talking about a test whether one key is smaller than another and the smallest key in a certain set, without being specific about what this means. Since we want to define a type (α, β) *mapping* for each pair of types α and β, we need to be sure that these notions make sense for any type of elements. Luckily, Objective CAML provides a built-in operators like $<$ and $\leq$ that will work for any type. For the types like integers and strings that we care about, these operators give the well-known ordering we would expect.

We shall arrange things so that every *B*-tree has all its leaves on the same level, as the one shown in Figure A.1 does. We need to implement insertion of new leaves in such a way that this property is maintained. Inserting a

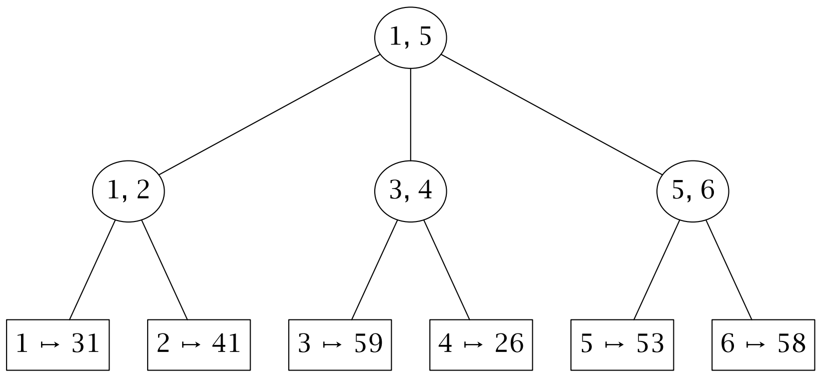

Figure A.2: *B-tree after insertion of* 3 ↦ 59

new leaf beneath a 2-node is easy: we just change the 2-node into a 3-node, putting the new leaf on the same level as the existing leaves Inserting a new leaf below a 3-node is slightly more difficult, because we need to split the 3-node and make two 2-nodes, then insert these nodes below the one above. This may cause a chain reaction, with more nodes splitting in their turn; if the root splits, then it is necessary to add a new root and increase the depth of the tree by one. Figure A.2 shows the result of inserting the new pair 3 ↦ 59 to the tree in Figure A.1; one of the nodes has been split to make room for the new leaf.

To implement insertion as a program, I shall use auxiliary functions $make_2$ and $make_3$ that take two or three trees and form a new node above them. Each of these functions can handle the case where at most one of the argument trees has split: if so, $make_2$ increases the number of children from two to three, and $make_3$ itself makes a split tree with two nodes of two children each.

```
let make₂ t₁ t₂ =
  match (t₁, t₂) with
      (Split (u₁, u₂), u₃) →
        Node₃ (minkey u₁, u₁, u₂, u₃)
    | (u₁, Split (u₂, u₃)) →
        Node₃ (minkey u₁, u₁, u₂, u₃)
    | (u₁, u₂) →
        Node₂ (minkey u₁, u₁, u₂);;

let make₃ t₁ t₂ t₃ =
  match (t₁, t₂, t₃) with
      (Split (u₁, u₂), u₃, u₄) →
        Split (Node₂ (minkey u₁, u₁, u₂), Node₂ (minkey u₃, u₃, u₄))
    | (u₁, Split (u₂, u₃), u₄) →
        Split (Node₂ (minkey u₁, u₁, u₂), Node₂ (minkey u₃, u₃, u₄))
    | (u₁, u₂, Split (u₃, u₄)) →
        Split (Node₂ (minkey u₁, u₁, u₂), Node₂ (minkey u₃, u₃, u₄))
    | (u₁, u₂, u₃) →
        Node₃ (minkey u₁, u₁, u₂, u₃);;
```

Another helper function deals with the case where the root of a tree must split:

```
let fixup =
  function
      Split (t1, t2) → Node2 (minkey t1, t1, t2)
    | t → t;;
```

These helper functions allow us to define *add* as follows:

```
let rec add0 x y =
  function
      (Empty | Deleted _) → Leaf (x, y)
    | Leaf (p, q) →
        if x = p then Leaf (x, y)
        else if x < p then Split (Leaf (x, y), Leaf (p, q))
        else Split (Leaf (p, q), Leaf (x, y))
    | Node2 (m, t1, t2) →
        if x < minkey t2 then make2 (add0 x y t1) t2
        else make2 t1 (add0 x y t2)
    | Node3 (m, t1, t2, t3) →
        if x < minkey t2 then make3 (add0 x y t1) t2 t3
        else if x < minkey t3 then make3 t1 (add0 x y t2) t3
        else make3 t1 t2 (add0 x y t3)
    | _ → failwith "Btree.add0";;

let add x y t = fixup (add0 x y t);;
```

The recursive function *add_xy* traverses a path in the tree in the same way as *find*, and uses $make_2$ and $make_3$ to rebuild the nodes along the path. A final use of *fixup* allows the tree to grow a new level when necessary.

Because every leaf is at the same depth, and every node in a B-tree has degree 2 or 3, we know that a B-tree of depth k has between 2^k and 3^k leaves. Put another way, a B-tree with N leaves has depth at most $\lfloor \log_2 N \rfloor$, so *find* runs in a worst-case time that is $O(\log N)$. In the worst case, *add* splits the nodes along a path all the way up to the root of the tree, so it also runs in $O(\log N)$ time.

It is possible, but very complicated, to implement deletion of nodes in a B-tree in such a way that the balanced nature of the tree is preserved. Happily, we shall rarely need to use deletion in our compilers, so we can be content with a simpler implementation that simply replaces a leaf *Leaf* (x, y) with *Deleted* x; the key x is kept in order to allow *minkey* to work properly. The resulting trees will no longer be proper B-trees, and they may become unbalanced, but at least they work correctly. We implement *remove* as follows:

```
let rec remove x t =
  match t with
      Empty → Empty
    | Deleted u → Deleted u
    | Leaf (p, q) → if x = p then Deleted p else Leaf (p, q)
    | Node2 (m, t1, t2) →
        if x < minkey t2 then Node2 (m, remove x t1, t2)
        else Node2 (m, t1, remove x t2)
    | Node3 (m, t1, t2, t3) →
```

```
        if $x < minkey\ t_2$ then $Node_3\ (m, remove\ x\ t_1, t_2, t_3)$
        else if $x < minkey\ t_3$ then $Node_3\ (m, t_1, remove\ x\ t_2, t_3)$
        else $Node_3\ (m, t_1, t_2, remove\ x\ t_3)$
    | _ → failwith "Btree.remove";;
```

Finally, the higher-order function *iter* is easy to define by recursion on the structure of the tree:

```
let rec iter f =
  function
      (Empty | Deleted _) → ()
    | Leaf (x, y) → f x y
    | Node2 (m, t1, t2) →
        iter f t1; iter f t2
    | Node3 (m, t1, t2, t3) →
        iter f t1; iter f t2; iter f t3
    | _ → failwith "Btree.iterate";;
```

By a happy accident, this version of *iter* visit the leaves of the tree in left-to-right order; that is, in increasing order of the value of the argument x. In both these functions, we do not handle the case *Split* (t_1, t_2), because such nodes do not occur in a well-formed tree.

A.2 Module *Hash*

We use a function *hash* that maps each key to an integer. Objective CAML provides this function as a built-in primitive that can accept an argument of any type. Using *hash*, we divide the keys among the buckets, so that each key x is assigned to the bucket $hash\, x \bmod n$, where n is the number of buckets. Because the number of different possible keys is usually larger than n, it may happen that several keys are mapped to the same bucket, and it is not sufficient to make each bucket capable of holding only a single (key, value) pair. Instead, we arrange that each bucket is itself a mapping from keys to values. Since each bucket will, all being well, contain only a few values, it is sufficient to use an association list to represent this mapping. With these design choices, a hash table can be implemented as an array of lists. We use an array because there is a fixed number of buckets and we would like constant-time access to each bucket, given its index. Creating a new hash table with n buckets amounts to creating an array of n mappings, each of them initially empty.

```
type (α, β) table = (α * β) list array;;

let create k = Array.create k [ ];;
```

Section 2.7.3 describes the type of arrays and the operations on arrays that we use here.

To look up a key x, we first find which bucket contains x if it is present, then use the *assoc* function to look in that bucket:

```
let find x m =
  let i = hash x mod Array.length m in
  List.assoc x m.(i);;
```

The expression *Array.length m* gives the size *n* with which the table *m* was initially created.

To add a new key *x* and value *y*, it is necessary to update the appropriate bucket, first using *remove_assoc* to remove any old value associated with *x*, then adding the pair (*x*, *y*) at the front:

```
let add x y m =
  let i = hash x mod Array.length m in
  m.(i) ← (x, y) :: List.remove_assoc x m.(i);;
```

The form *m*.(*i*) ← *E* modifies array *m* so that the value in bucket *i* is changed to the value of *E*.

The other functions *clear*, which sets each bucket back to *empty*, and *remove*, which removes a given key, are easy to define using similar methods. Finally, the function *iter* involves a loop over all buckets, applying the specified function to each element of each of them:

```
let iter f m =
  let g (x, y) = f x y in
  for i = 0 to Array.length m − 1 do List.iter g m.(i) done;;
```

Here, the function *g* is like *f*, but expects its arguments as an ordered pair. Unlike the corresponding function on mappings represented by *B*–trees, this definition does not visit the keys in any order that is easy to predict. This is a consequence of the unpredictable behaviour of the *hash* function.

Good performance of hash tables depends on making good choices for the hash function and the number of buckets. We hope for a hash function that will spread the key values that occur in an application fairly evenly over all the buckets, since if all values are clustered in one bucket, the performance will be no better than that given by the underlying implementation of mappings. In order to make an implementation of hash tables that works for any type of key, we are using the built-in *hash* function provided by the Objective Caml system, and so have no control over this. For a more specific implementation where, for example, the keys are strings, an appropriate choice of hash function might combine all the characters of the string like this:

```
let string_hash s =
  let h = ref 0 in
  for i = 0 to string_length s − 1 do
    h := 5 ∗ !h + int_of_char s.[i]
  done;
  !h;;
```

The multiplication by 5 ensures that a small change to the string, such as transposing two adjacent characters, gives a different value for the hash function. If a hash table is going to store identifiers in a compiler, then it is worth checking at least that all the built-in identifiers in the language are mapped to different buckets in the hash table, and that there is a resonable spread of identifiers between buckets for some sample programs written in the source language of the compiler.

Choosing the number of buckets in a hash table involves estimating the number of keys that will be entered in a typical execution. Using a large number of buckets means that there will be fewer collisions where several keys hash to the same bucket, but large tables obviously take more space, and they also take more time to initialize. The ideal is to have, say, twice as many buckets as the typical number of entries in the table. Then few buckets will contain more than one or two entries, and the time to look up a key in a bucket will be almost bounded by a constant, giving effectively constant time look-up.

A.3 Module *Print*

The module *Print* provides the function *printf* that prints formatted text on standard output, as well as *fprintf* for printing on any output stream and *sprintf*, which returns the formatted text as a string. All are defined in terms of a function

```
do_print : (char → unit) → string → [arg list] → unit
```

that is defined so that *do_print f fmt args* formats the arguments *args* according to the format *fmt*, and outputs the result by passing each character to the function f. Thus *fprintf* may be defined by

```
let fprintf fp fmt args = do_print (output_char fp) fmt args;;
```

and *printf* is almost equivalent to *fprintf stdout*:

```
let printf fmt args = fprintf stdout fmt args; flush stdout;;
```

(except that we flush the output at the end). The function *sprintf* stores successive characters in a string and returns a suitable substring at the end:

```
let sprintf fmt args =
  let buf = String.create 256 and n = ref 0 in
  let store c = buf.[!n] ← c; incr n in
  do_print store fmt args;
  String.sub buf 0 !n;;
```

For simplicity, we limit the length of the string to 256; a better implementation would allow the string to grow as needed.

To implement *do_print* itself, we need the definition of the type *arg*. There are just three basic kinds of argument, and all others are implemented in terms of these:

```
type arg =
    Str of string                         (∗ String ∗)
  | Chr of char                           (∗ Character ∗)
  | Ext of ((string → arg list → unit) → unit);;     (∗ Extension ∗)
```

When printed, the argument *Str s* produces the literal string s, and the argument *Chr c* produces the single character c. Arguments formed with *Ext* make the printing mechanism extensible in a way I will describe below.

The *do_print* function works its way through the format string, inserting one argument whenever it finds a '$' character in the format:

```
let rec do_print out_fun fmt args =
  let j = ref 0 in
  for i = 0 to String.length fmt − 1 do
    let c = fmt.[i] in
    if c ≠ '$' then
      out_fun c
    else begin
      begin match nth args !j with
          Str s →
            for k = 0 to String.length s − 1 do out_fun s.[k] done
        | Chr c →
            out_fun c
        | Ext f →
            f (do_print out_fun)
      end;
      incr j
    end
  done;;
```

The argument forms *Str s* and *Chr c* provide the basis for implementing the public functions for formatting integers, floating-point numbers, strings, characters and booleans:

```
let fNum n = Str (string_of_int n);;
let fFlo x = Str (string_of_float x);;
let fStr s = Str s;;
let fChr c = Chr c;;
let fBool b = Str (if b then "true" else "false");;
```

For uniformity of notation, we define public versions *fStr* and *fChr* of the private constructors *Str* and *Chr*.

We now turn to the extension mechanism. A simple way of extending *printf* is via the function *fMeta* : *string* ∗ *arg list* → *arg*. The idea is that an argument *fMeta* (*fmt*, *args*) produces the same formatted text as would be output by *printf fmt args*. This allows formats to be defined for new types. For example, if we define the type *complex* as pairs of *float*'s:

```
type complex == float * float;;
```

then we can define a format for printing complex numbers as follows:

```
let fCompl (re, im) = fMeta ("$+$i", [fFlo re; fFlo Im]);;
```

With this definition, the call

```
printf "z = $\n" [fCompl (2.0, 3.0)]
```

prints "z = 2.0+3.0i".

Another useful function is *fList*, which provides a general-purpose way of printing lists. For example, the call

```
printf "List = $" [fList(fNum) [1; 2; 3]]
```

prints the string "List = 1, 2, 3". The function *fList* is a higher-order formatting function that takes another formatting function as its argument, and produces a formatting function for lists of items. The type of *fList* is

$$fList : (\alpha \to arg) \to \alpha\ list \to arg,$$

because in an application like *fList*(*fNum*) [1; 2; 3], the function *fList* is applied to *fNum* : *int* → *arg* to produce a function of type *int list* → *arg*, and this works when any other type replaces *int*.

Both these functions can be defined in terms of the higher-order function *Ext*, a generalization of *fMeta*. If f is a function with the appropriate type, then the argument *Ext f* produces the same formatted text as would be printed by applying f to the function *printf*. Thus the call

> *printf* "\$" [*Ext f*]

produces the same output as *f printf*. In practice, an argument of the form *Ext f* may be supplied to other formatting functions: for example,

> *fprintf stderr* "\$" [*Ext f*]

produces the same output on *stderr* as *f* (*fprintf stderr*), and

> *sprintf* "\$" [*Ext f*]

results in the call *f prf*, where *prf* is a function like *printf* that stores the characters in a string buffer.

For *Ext f* to be correctly typed, f must be a function that can be applied to *printf*; thus f must have type (*string* → *arg list* → *unit*) → *unit*. The constructor *Ext* is declared to take an argument of this type in the definition of type *arg*, and the public version *fExt* has type

> ((*string* → *arg list* → *unit*) → *unit*) → *arg*.

The fact that the function f in *Ext f* receives *printf* or an equivalent as its argument means that it can use that function in recursive calls – in fact, it should produce all its output using the function it receives as an argument, so that the output goes to the right place when *fprintf* or *sprintf* are used in place of *printf*.

To solve the problem of printing complex numbers using *Ext* in place of *fMeta*, we would define a function *print_complex* as follows:

> **let** *print_complex prf* (*x*, *y*) = *prf* "\$+\$i" [*fFlo x*; *fFlo y*];;

This function prints a complex number (*x*, *y*) using a *printf*-like function *prf* that it receives as an argument. Then we can make a new definition of *fCompl* as follows:

> **let** *fCompl z* =
>   **let** *f prf* = *print_complex prf z* **in** *fExt f*;;

Like many applications of higher-order functions, this seems complicated at first, but the effect can be worked out simply:

> *printf* "\$" [*fComplex* (2.0, 3.0)]
>   ≡ *printf* "\$" [*fExt f*] where *f prf* = *print_complex prf* (2.0, 3.0)
>   ≡ *f printf*
>   ≡ *print_complex printf* (2.0, 3.0)
>   ≡ *printf* "\$+\$i" [*fFlo* 2.0; *fFlo* 3.0]

and this prints "2.0+3.0i". In this example, the recursive call of *printf* made by *print_complex* uses only basic arguments like *fFlo x* that are defined in terms of the *Str* constructor for type *arg*, but there is no reason why these recursive calls may not also use the *Ext* constructor in their turn: this is a convenient way to print values of a recursive data type.

We have just used *Ext* to do something that we did with *fMeta* earlier, and in fact *fMeta* can be implemented in terms of *Ext*. We want

```
printf "$" [fMeta (fmt, args)]
```

to be equivalent to *printf fmt args*, so we may reason as follows:

```
printf "$" [fMeta fmt args]
  ≡ printf fmt args
  ≡ f printf where f prf = prf fmt args
  ≡ printf "$" [Ext f]
```

This justifies the following definition of *fMeta* in terms of *Ext*:

```
let fMeta (fmt, args) = Ext (function prf → prf fmt args);;
```

The other function *fList* can also be implemented in terms of *Ext*:

```
let fList cvt xs =
  let f prf =
    if xs ≠ [ ] then begin
      prf "$" [cvt (hd xs)];
      List.iter (function y → prf ", $" [cvt y]) (tl xs)
    end in
  Ext f;;
```

The three functions *fMeta*, *fList* and *fExt* are used in the lab exercises to define printing formats for various types that are introduced in our compilers, such as the type of abstract machine instructions, but they are not otherwise used in this book.