COMPILING WITH C# AND JAVA

P.D. Terry, Rhodes University, August 2017

CONTENTS

These notes consist of revised extracts from a preliminary draft of a book.

Chapter 13 - A Parva Compiler: the Back End

Chapter 14 - A Parva Compiler: Functions and Parameters

Appendix B - Library routines

Appendix C - Context-free grammars for Parva

Bibliography

1 TRANSLATORS AND LANGUAGES

1.1 Objectives

Computer languages provide an essential link in the chain between human and computer. In this text - extracts from a longer one (Terry 2005) - we hope to make the reader more aware of some aspects of:

- imperative programming languages - their syntactic and semantic features; the ways of specifying syntax and semantics; problem areas and ambiguities; the power and usefulness of various features of a language;
- translators for programming languages - the various classes of translator (assemblers, compilers, interpreters); implementation of translators;
- compiler generators - tools that are available to help automate the construction of translators for programming languages.

The approach taken in this book is pragmatic. It has been written so as not to be too theoretical but to relate easily to languages which the reader already knows or can readily understand - which these days probably means Java*TM*, C# or C++. The reader is expected to have a good background in one of those languages, access to an implementation of C# or Java and, preferably, some background in assembly language programming and simple machine architecture. We shall rely quite heavily on this background, especially on the understanding the reader should have of the meaning of various programming constructs.

Significant parts of the text concern themselves with case studies of actual translators for simple languages. Other important parts of the course are to be found in the many exercises and suggestions for further study and experimentation on the part of the reader. In short, the emphasis is on "doing" rather than just "reading" and the reader who does not attempt the exercises will miss many, if not most, of the finer points.

The primary language used to illustrate our discussion is C#. Complete source code for all the case studies is to be found in the *Resource Kit* that is included with the book and on the book's website. As well as C# versions of this code, we have provided equivalent source code in Java, and care has been taken to use coding techniques that are essentially identical in both languages. For contrast and clarity, some of the discussion is presented in a pseudo-code that often resembles Modula-2 rather more than it does a dialect of C. It is only fair to warn the reader that the code extracts in the book are often just that - extracts. There may be instances where identifiers are used whose significance may not be immediately apparent from their local context. The conscientious reader will have to expend some effort in browsing the code to appreciate the finer points that lie behind the broader principles emphasized in the text.

1.2 Systems programs and translators

Users of modern computing systems can be divided into two broad categories. There are those who never develop their own programs but simply use ones developed by others. Then there are those who are concerned as much with the development of programs as with their subsequent use. This latter group, of whom we as computer scientists form a part, is fortunate in that program development is usually aided by the use of so-called high-level languages for expressing algorithms, the use of interactive editors for program entry and modification, the use of version management systems for handling large projects and the use of sophisticated job control languages or graphical user interfaces for control of execution. Programmers armed with such tools have a very different picture of computer systems from those who are presented with the hardware alone, since the use of compilers, editors and operating systems - known generally as **systems programs** - removes from humans the burden of developing their systems at the so-called machine (or assembler) level. That is not to claim that the use of such tools removes all burdens or all possibilities for error, as the reader will be well aware.

Well within living memory, much program development was done in "machine language" - indeed, some of it, of necessity, still is - and perhaps some readers have even experienced this for themselves when experimenting with microprocessors. Just a brief exposure to programs written as almost meaningless collections of binary or hexadecimal digits is usually enough to make one grateful for the presence of "high-level" languages, clumsy and irritating though some of their features may be.

However, for high-level languages to be usable one must be able to convert programs written in them into the

binary or hexadecimal digits and bitstrings that a machine will understand. At an early stage in the history of computer science it was realized that if constraints were put on the syntax of a high-level language the translation process became one that could be automated. This led to the development of **translators** or **compilers** - programs which accept (as data) a textual representation of an algorithm expressed in a **source language** and which produce (as primary output) a representation of the same algorithm expressed in another language - the **object** or **target language**.

A translator, being a program in its own right, must itself be written in a computer language, known as its **host** or **implementation language**. Today it is rare to find translators that have been developed from scratch in machine language. Historically, the first translators had to be written in this way, and at the outset of translator development for any new system one has to come to terms with the machine language and machine architecture for that system. Even so, translators are now invariably developed in high-level languages, often using the techniques of **cross-compilation** and **bootstrapping** that will be discussed in more detail later.

The first major translators written may have been the Fortran compilers developed by Backus and his colleagues at IBM in the 1950s, although machine code development aids were in existence by then. The first Fortran compiler is estimated to have taken about 18 person-years of effort. It is interesting to note that one of the primary concerns of the team was to develop a system that could produce object code whose efficiency of execution would compare favourably with that which expert human machine coders could achieve. An automatic translation process does not usually produce code that is as optimal as can be written by a really skilled user of machine language and, to this day, important components of systems are sometimes developed at (or very near to) machine level in the interests of saving time or space.

Translator programs themselves are never completely portable (although parts of them may be), and they usually depend to some extent on other systems programs that users have at their disposal. In particular, input/output (I/O) and file management on modern computer systems are usually controlled by the **operating system**. This is a program or suite of programs and routines, whose job it is to control the execution of other programs so as best to share resources such as printers, plotters, disk files and networks, often making use of sophisticated techniques such as parallel processing, multiprogramming and so on. In addition, operating systems provide for security in preventing one process invading or corrupting the memory space of another one. For many years the development of operating systems required the use of programming languages that remained closer to the machine code level than did languages suitable for scientific or commercial programming. During the 1970s a number of successful higher level languages were developed with the express purpose of catering for the implementation of operating systems and real-time control. The most obvious example of such a language is C, developed originally for the implementation of the UNIX operating system, and now still widely used in all areas of computing.

1.3 The benefits of using high-level languages

The reader will rapidly become aware that the design and implementation of translators is a subject that may be developed from many possible angles and approaches. The same is true for the design of programming languages. In passing we comment that there is a strong relationship between the abstract notion of a programming language and its corresponding concrete implementation, and it is easy to confuse features of one with features of the other. For example, garbage collection is the responsibility of an implementation of Java rather than being intrinsically a property of the Java language itself.

Unlike natural languages, which may be used to communicate a large variety of information, thoughts and emotions between speakers or authors and listeners or readers, computer languages are notations for describing how the solution to a problem posed at one level and in one domain may ultimately be effected by a processor or processors working at another level and in another, highly rigorous domain. What these notations have in common is that they attempt to bridge a "semantic gap". Where they differ is largely in how far across this gap, in what detail and from what standpoint they allow the user to express the solution. Thus we speak of **high-level languages**, where the notation may be quite close to being an abstract mathematical notation, and of **low-level languages** (like assembler languages) where the notation tends to permit little more than a list of atomic machine level instructions that are to be followed by a simple-minded drone of a processor.

High-level languages may further be classified as "procedural" (like Fortran, Ada, C, Pascal and Modula-2), "object-oriented" (like C++, Java and C#), "functional" (like Lisp, Scheme, ML or Haskell), or "logic" (like Prolog). This classification distinguishes between the various paradigms - approaches to formulating a solution - that have been found suitable for handling different categories of problems. Procedural and object-oriented languages are those that relate to an "imperative" paradigm - one where the state of a system is encapsulated in

data structures associated with named variables or objects, where these structures may be built, exploited and combined by the manipulation of expressions, and where the flow of the process for achieving all this is described in terms of algorithmic control structures involving decision, selection and repetition. This paradigm has been used for many years, and its implementation is the only one we consider in this text.

From the outset of the development of high-level imperative languages, claims have been made for their superiority over low-level languages.

- *Readability:* A good high-level language will allow programs to be written that in some ways resemble quasi-English descriptions of the underlying algorithms. If care is taken, the coding may be done in a way that is essentially self-documenting, a highly desirable property when one considers that many programs are written once but possibly studied by humans many times thereafter.

- *Familiarity:* Most computers are "binary" in nature. Blessed with ten toes on which to check out their number-crunching programs, humans may be somewhat relieved that high-level languages usually appear to make decimal arithmetic the rule, rather than the exception, and provide for mathematical operations in a notation consistent with standard mathematics.

- *Portability:* High-level languages, being essentially machine independent, hold out the promise of being used to develop portable software. This is software that can, in principle (and even occasionally in practice), run unchanged on a variety of different machines - provided only that the source code is recompiled as it moves from machine to machine.

  To achieve machine independence, high-level languages may deny access to low-level features, and are sometimes spurned by programmers who have to develop low-level machine dependent systems. However, some languages, like C and Modula-2, were specifically designed to allow access to these features from within the context of high-level constructs. More recently the access has been controlled through the provision of specially written libraries of routines, which can be called through generic mechanisms, although they may not necessarily themselves have been implemented in the language of the user.

- *Generality:* Most high-level languages allow the writing of a wide variety of programs, thus relieving the programmer of the need to become expert in many diverse languages.

- *Brevity:* Programs expressed in high-level languages are often considerably shorter (in terms of their number of source lines) than their low-level equivalents.

- *Error checking:* Being human, a programmer is likely to make many mistakes in the development of a computer program. Many high-level languages - or at least their implementations - can, and often do, enforce a great deal of error checking both at compile-time and at run-time. For this they are, of course, often criticized by programmers who have to develop time-critical code or who want their programs to abort as quickly as possible.

These advantages sometimes appear to be over-rated, or at any rate, hard to reconcile with reality. For example, readability is usually within the confines of a rather stilted style and some beginners are disillusioned when they discover just how unnatural a high-level language is. Similarly, the generality of many languages is confined to relatively narrow areas and programmers are often dismayed to come across areas (like string handling in standard Pascal) which seem to be very poorly handled. The explanation is often to be found in the close coupling between the development of high-level languages and of their translators. When one examines successful languages, one finds numerous examples of compromise dictated largely by the need to accommodate language ideas on rather uncompromising, if not unsuitable, machine architectures. To a lesser extent, compromise is also dictated by the quirks of the interface to established operating systems on machines. Finally, some appealing language features turn out to be either impossibly difficult to implement or too expensive to justify in terms of the machine resources needed. It may not immediately be apparent that the design of Pascal (and of several of its successors such as Modula-2 and Oberon) was governed partly by a desire to make it easy to compile. It is a tribute to its designer that, in spite of the limitations which this desire naturally introduced, Pascal became so popular, the model for so many other languages and extensions, and encouraged the development of superfast compilers such as were found in Borland's Turbo Pascal and Delphi systems.

The fact that the use of high-level languages distances the programmer from the low-level nuances of a machine may lead to other difficulties and misconceptions. For example, beginners often fail to distinguish between the compilation (compile-time) and execution (run-time) phases in developing and using programs written in high-

level languages. This is an easy trap to fall into, since the translation (compilation) is often hidden from sight, or invoked with a special function key or mouse click from within an integrated development environment that may possess many other magic function keys and menus. Furthermore, beginners are often taught programming with this distinction deliberately blurred, their teachers offering explanations such as "when a computer executes a *read* statement it reads a number from the input data into a variable". This hides several low-level operations from the beginner. The underlying implications of file handling, character conversion, and storage allocation are glibly ignored - as indeed is the necessity for the computer to be programmed to recognize the word *read* in the first place. Anyone who has attempted to program input/output (I/O) operations directly in assembler languages will know that many of them are non-trivial to implement. Hopefully some of these misconceptions will be cleared up after studying the implementation of high-level languages in a book such as this.

In this regard we might also observe that the aspect of programming in a high-level language that most annoys beginners is one which few of these notations attempt to handle. Humans, in conversation with one another, soon learn to be tolerant of lapses of grammar, spelling or vocabulary. Computer languages show no such tolerance and the normal refusal of a compiler to allow a program to deviate from syntactic perfection is, understandably, very irritating - to say nothing of the refusal of most compiled programs to react to any circumstances or errors in use that have not been foreseen by the developer of the system.

1.4 The curse of complexity

As the field of computing has advanced, so has come about the realization that the programming process is far from being simple. Indeed, programming may be the most complex activity humans have invented for themselves - and it appears to become ever more complex as the demands of users grow and as new uses are found for computers. One of the responsibilities of computer science is to develop and evaluate ways to control this complexity, and the programming language is the primary tool through which this can be accomplished. This is something that has to be done in the face of conflicting requirements - users look for a tool that is multi-purpose and will describe solutions succinctly, yet remains expressive, and, importantly, capable of describing algorithms that can be executed efficiently.

Historians of computer science can point to a steady stream of ideas that have been put forward to achieve these goals. Within the sphere of imperative programming, earlier languages like Fortran were not that far removed from assembler languages in what they allowed the programmer to describe. In particular they did little to prevent the writing of "spaghetti code" - a style of coding characterized by undisciplined branching between the sections responsible for performing subtasks. This was all corrected in the 1960s when "structured programming" - the restriction of control structures to a few highly regular forms - was proposed as the model to follow and was supported, if not subtly enforced, by the languages of the day. At about this time the importance of type-checking and the usefulness of being able easily to define data structures appropriate to the task at hand came to be appreciated. Languages (like Pascal) which supported strict typing along with a systematic approach to data structuring and control structuring attracted an enormous and justifiable following. Shortly afterwards came the emphasis on the "abstract data type" - the packaging of data structures and the algorithms that manipulate them in such a way that, once done, a user could safely ignore what could be perceived as irrelevant implementation details and just plug such packages into new code. This led to the development and acceptance of the module concept in languages like Ada and Modula-2, which were still regarded as inherently "procedural", in that the programmers using them decomposed the solutions to problems by concentrating more on *what* to do, rather than on what to do it *with*. Growing out of this has come the most recent movement, that towards "object-oriented programming", where a system is conceived as being comprised of a collection of objects which, like the abstract data types of the recent past, package data and operations on that data into coherent sub-systems, but from which other related objects can be derived through a process called inheritance, and which are able to decide how best to attend to their own affairs. The realization of these ideas in the language C++, which swept almost all other languages before it for several years, and more recently in the simpler languages Java and C#, probably marks the current "state of the art", although some cynics would contend that, while the use of these languages may simplify the solution of complicated problems, they also complicate the solution of simple problems.

1.5 Successful language design

The design of a programming language requires a high degree of skill and judgement. There is evidence to show that language is not only useful for *expressing* ideas. Because language is also used to *formulate* and *develop* ideas, familiarity with language largely determines *how* and, indeed, *what* one can think. In the case of programming languages, there has been much controversy over this. For example, in languages like Fortran - for

long the *lingua franca* of the scientific computing community - recursive algorithms were "difficult" to use (not impossible, just difficult!), with the result that many programmers brought up on Fortran found recursion strange and unnatural, even something to be avoided at all costs. It is true that recursive algorithms are sometimes "inefficient" and that compilers for languages which allow recursion may exacerbate this. On the other hand, it is also true that some algorithms are more simply explained in a recursive way than in one which depends on explicit repetition (the best examples probably being those associated with tree manipulation).

There are two divergent schools of thought as to how programming languages should be designed. The one, typified by the Wirth school, stresses that languages should be small and understandable, and that much time should be spent in consideration of what tempting features might be omitted without crippling the language as a vehicle for system development (one is reminded of the adage, usually attributed to Einstein, that one should "make everything as simple as possible, but no simpler"). The other, beloved of languages designed by committees with the desire to please everyone, packs a language full of every conceivable potentially useful feature. Both schools claim success. The Wirth school has given us Pascal, Modula-2 and Oberon, all of which have had an enormous effect on the thinking of computer scientists. The other approach has given us Ada, C and C++, which are far more difficult to master well and extremely complicated to implement correctly, but which claim spectacular successes in the marketplace.

It is of interest to note the existence of a cyclic phenomenon in the approach taken to language design. Once a language has gained some credence it seems inevitable that it then attracts the attention of well-meaning people who build on it and extend it, often resulting in products that are almost hideously complicated. Hoare is reputed to have said, not without cause, that "Algol 60 was not only a great improvement on its predecessors, but also on nearly all of its successors", and Eisenbach remarked wryly that "Modula-2 would be the perfect language if we added just one more feature. Unfortunately we all have a different idea as to what that feature should be". Uncontrolled accretion is not the hallmark of good science, regardless of the field. Thankfully, from time to time there are revolts against this process, and designers come up with radically simpler languages that nevertheless seem to incorporate all the advantages of their cluttered forebears, and which rapidly gain acceptance among discerning scientists. Examples of this are to be seen in the spectacular success of Pascal, designed partly in reaction to what was perceived as an overly complex predecessor, Algol-68 and, more recently, Java and C#, two fairly similar object-oriented languages which are much simpler than C++, the most popular of such languages until they appeared. Critics of the headlong rush to embrace Java and C# will, however, point out that while the languages themselves might be simpler than C++, users of either language have to become familiar with an enormous set of supporting library facilities if they are to use them to best advantage.

The elusive search for the perfect language shows no sign of abating, but in this regard, aspects of language design that contribute to success include the following.

- *Orthogonality:* Good languages tend to have a small number of well thought out features that can be combined in a logical way to supply more powerful building blocks. Ideally these features should not interfere with one another, and should not be hedged about by a host of inconsistencies, exceptional cases and arbitrary restrictions. Most languages have blemishes - for example, in Wirth's original Pascal a function could only return a scalar value, not one of any structured type. Many potentially attractive extensions to well-established languages prove to be extremely vulnerable to unfortunate oversights in this regard.

- *Easily learned:* When new languages are proposed, these often take the form of derivatives or dialects of well-established ones, so that programmers can be tempted to migrate to the new language and still feel largely at home. This was the route taken in developing C++ from C, Java and C# from C++, and Oberon from Modula-2.

- *Clearly defined:* A language must be clearly and unambiguously described for the benefit of both the user and the compiler writer.

- *Quickly translated:* Programs written in the language should admit quick translation so that program development time when using the language is not excessive.

- *Modularity:* It is desirable that programs can be developed in the language as a collection of separately compiled modules, with appropriate mechanisms for ensuring self-consistency between these modules.

- *Efficient:* The language features should permit the generation of efficient object code.

- *Widely available:* It should be possible to provide translators for all the major machines and for all the major operating systems.

1.6 The benefits of formal description

The importance of a clear language description or specification cannot be over-emphasized. This must apply firstly to the so-called **syntax** of the language - that is, it must specify accurately what form a source program may assume. It must apply secondly to the so-called **static semantics** of the language - for example, it must be clear what constraints are placed on the use of entities of differing types, and the scope that various identifiers have across the program text. Finally, the specification must also apply to the **dynamic semantics** of programs that satisfy the syntactic and static semantic rules - that is, it must be capable of predicting the effect any program expressed in that language will have when it is executed.

Programming language description is extremely difficult to do accurately, especially if it is attempted through the medium of potentially confusing languages like English. There is an increasing trend towards the use of formalism for this purpose, some of which will be illustrated in later chapters. Formal methods have the advantage of precision since they make use of the clearly defined notations of mathematics. To offset this they may be somewhat daunting to programmers weak in mathematics, and do not necessarily have the advantage of being very concise - for example, the informal description of Modula-2 (albeit slightly ambiguous in places) took only some 35 pages (Wirth 1985) while a formal description prepared by an ISO committee runs to over 700 pages.

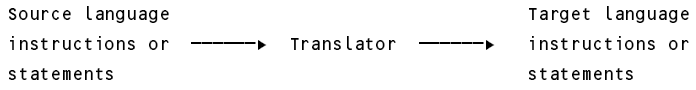
Formal specifications have the added advantage that, in principle, and to a growing degree in practice, they may be used to help automate the implementation of translators for the language. Indeed, it is increasingly rare to find modern compilers that have been implemented without the help of so-called **compiler generators**. These are programs that take a formal description of the syntax and semantics of a programming language as input and produce major parts of a compiler for that language as output. Our development in future chapters will illustrate the use of such a tool, although we shall also show how compilers may be crafted by hand.

Further reading

As we proceed, we hope to make the reader more aware of some of the points raised in this chapter. An excellent survey of programming paradigms and the principles of programming languages is to be found in the book by Bal and Grune (1994). Language design is a difficult area, and much has been, and continues to be, written on the topic. The reader with an interest in early computing history might like to refer to the books by Tremblay and Sorenson (1985), Watson (1989), and Watt (1990, 2004) for readable summaries of the subject, and to the papers by Wirth (1974, 1976a, 1988), Kernighan (1981), Welsh, Sneeringer and Hoare (1977) and Cailliau (1982). Interesting background on several well-known older languages can be found in *ACM SIGPLAN Notices* for August 1978 and March 1993 (Lee and Sammet 1978, 1993) - two special issues of that journal devoted to the history of programming language development. Stroustrup (1993) gives a fascinating exposition of the development of C++. The terms "static semantics" and "dynamic semantics" are not used by all authors; for a discussion on this point see the paper by Meek (1990).

2 TRANSLATOR CLASSIFICATION AND STRUCTURE

A translator may formally be defined as a function, whose domain is a source language, and whose range is contained in an object or target language.



A little experience with translators will reveal that it is not usually considered part of the translator's function to execute the algorithm expressed by the source, merely to change its representation from one form to another. In fact, at least three languages are involved in the development of translators: the source language to be translated; the target language to be generated; the host language to be used for implementing the translator. If the translation takes place in several stages, there may even be other, intermediate, languages. Most of these - and, indeed, the host language and object languages themselves - usually remain hidden from a user of the source language.

2.1 Classes of translator

It is common to distinguish between several well-established kinds of programming language translator:

- The term **assembler** is usually associated with those translators that map low-level language statements into machine code which can then, in principle, be executed directly. Individual source language statements usually map one-for-one to machine-level instructions. Confusingly, source code for such translators is usually said to be written "in assembler". When a clear distinction is needed, this text will use the word "ASSEMBLER" to denote the source language, and "assembler" to denote the translator program.

- The term **macro-assembler** is also associated with those translators that map low-level language statements into machine code, and is a variation on the above. Most source language statements map one-for-one into their target language equivalents, but some *macro* statements map into a sequence of machine-level instructions - effectively providing a text replacement facility and thereby extending the assembly language to suit the user. (This is not to be confused with the use of procedures or other subprograms to "extend" high-level languages, because the method of implementation is usually very different.)

- The term **compiler** is usually associated with those translators that map high-level language statements into machine code which can then, in principle, be executed directly. Individual source language statements usually map into many machine-level instructions.

- The term **pre-processor** is usually associated with those translators that map programs written in a superset of a high-level language into programs expressed in the original high-level language, or that perform simple textual substitutions before translation takes place. The best-known pre-processor is probably that which forms an integral part of implementations of the language C, and which provides many of the features that contribute to the once widely-held perception that C was the only really portable language.

- The term **high-level translator** is often associated with those translators that map programs written in one high-level language into equivalent programs expressed in another high-level language - usually one for which sophisticated compilers already exist on a range of machines. Such translators are particularly useful as components of a two-stage compiling system or in assisting with the bootstrapping techniques to be discussed shortly.

- The terms **decompiler** and **disassembler** refer to translators which attempt to take object code at a low level and regenerate source code for an equivalent program expressed in a higher level language. While this can be done quite successfully for the production of assembler-level source code, it is much more difficult when one tries to recreate source code originally written in, say, Ada.

2.2 T-diagrams

A useful notation for describing a computer program is in terms of a so-called **T-diagram**, examples of which are shown in Figure 2.1. (a) shows the general form of the diagram, while (b) shows a particular example, for a simple sorting program as might be supplied as one of the utilities that accompany an operating system:

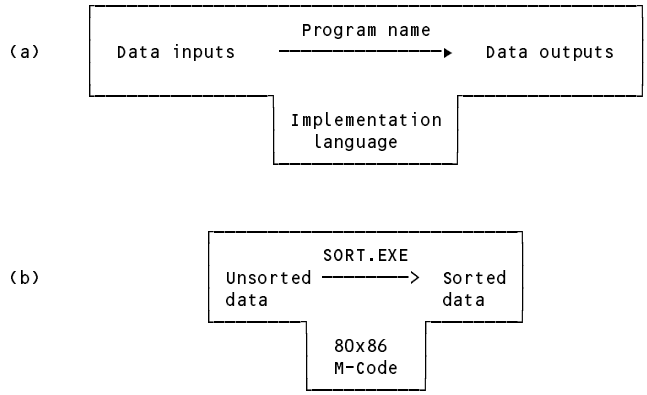


Figure 2.1 Two simple T-diagrams (a) the general form (b) A sorting program for an 80x86 machine

A program becomes interesting when it is executed. The above example implies that the algorithm embodied in the sorting program *SORT.EXE* has somehow been implemented in 80x86 machine code and is able to be plugged into an 80x86 machine and fed with some unsorted data which it then processes, to deliver sorted data as output. The way in which the various code, files and machine are plugged together (a bit like blocks of Lego) is shown in Figure 2.2. "Plugging" is a simplification - the process involves interactions with file and operating systems that are not shown in detail here.

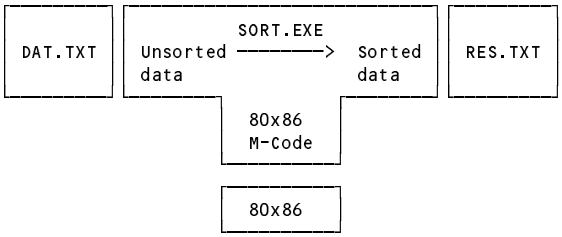


Figure 2.2 Executing a sorting program on an 80x86 machine

As has already been noted, while it is common to find programs like *SORT.EXE* supplied "ready to run", they are rarely if ever developed directly in machine code. More likely they are developed in a flavour-of-the-epoch high-level language. A few years ago this might have happened using one or other of the languages Pascal or C. These programs might be represented as below. The underlying algorithms might be exactly equivalent - and equivalent to the one expressed in 80x86 machine code - but all three programs would look rather different if they were printed out in source form.

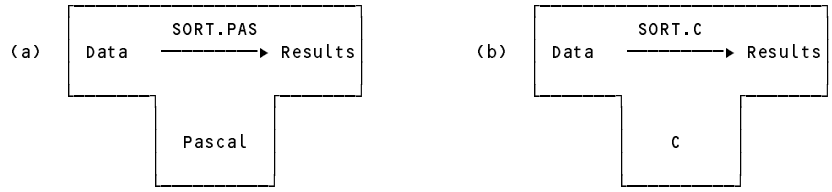


Figure 2.3 Two sorting programs, one expressed (a) in Pascal and another (b) in C

Neither of the programs in Figure 2.3 can be executed directly, except by way of a thought experiment. That is, neither can be plugged into an electronic machine directly.

Enter the concept of the compiler. A compiler has, as input data, not an unsorted list of numbers, but a text file of statements defining an algorithm in a high-level language. The output result is, not a sorted list of numbers, but a text (or binary) file of machine-level statements - which can be plugged into a machine directly. Figure 2.4 shows this - (a) shows the general form of a compiler, while (b) shows a diagram for a famous compiler of the 1980s, *TPC.EXE* whose purpose was to translate a program expressed in Pascal into a semantically equivalent one expressed in 80x86 machine language, which could then be plugged into such a machine and executed. The *TPC.EXE* compiler, of course, was marketed in the form which could be plugged directly into the 80x86 host machine. We shall return to the interesting topic of how the compiler itself came to be written in a later section.

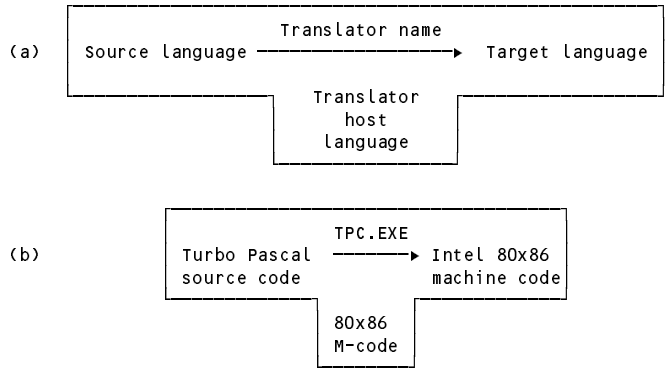


Figure 2.4 (a) a general translator; (b) binary version of a Pascal compiler for an MS-DOS system.

Compilation of the source code of the *SORT.PAS* program using this compiler is depicted by plugging the translator *TPC.EXE* into a machine and plugging the source program code into the left arm. After compilation, the executable version can be unplugged from the right arm, as shown in Figure 2.5, executed by plugging it into the same or another 80x86 machine, or filed away for later use or distribution.

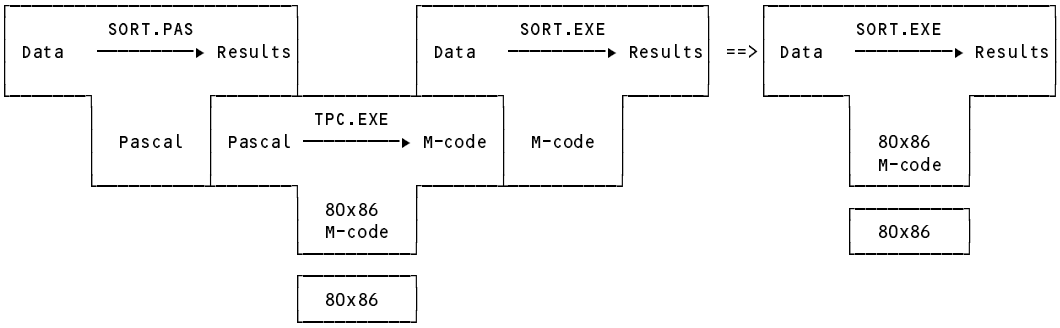


Figure 2.5 A Turbo Pascal compilation on an 80x86 machine

Similarly, Figure 2.6 shows how the C version of the sorting program could be compiled with another famous compiler from Borland International, *BCC.EXE*.

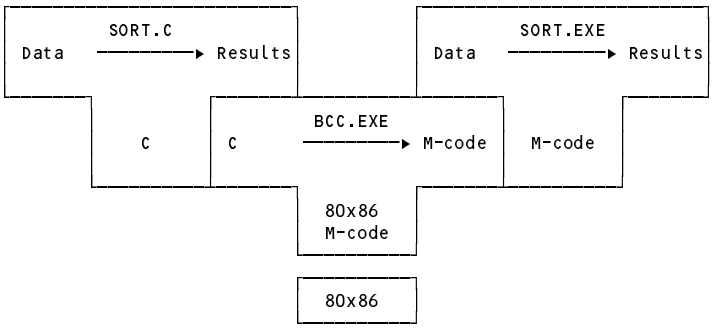


Figure 2.6 A C compilation on an 80x86 machine

We can also regard the combination of the *TPC.EXE* compiler plugged into a machine as depicting a dedicated machine whose aim in life is to convert Turbo Pascal source programs into their 80x86 machine code equivalents. Note in Figure 2.6 that the semantic meaning of *SORT.PAS* is assumed to be preserved in the translation process; that is, *SORT.PAS* and *SORT.EXE* are merely two different representations of exactly the same algorithm (and similarly for the version developed in C). Hopefully it will be obvious that attempting to plug *SORT.PAS* into *BCC.EXE* or *SORT.C* into *TPC.EXE* will not do anything useful, and similarly plugging either of *TPC.EXE* or *BCC.EXE* onto a machine other that one that employs 80x86 machine code will do nothing useful either.

T-diagrams were introduced by Bratman (1961). They were refined by Earley and Sturgis (1970) and are also used in the books by Bennett (1990), Louden (1997), Watt (1993), and Aho, Sethi and Ullman (1986).

2.3 Self-resident compilers, cross compilers and self-compiling compilers.

Many translators generate code for their host machines. These are called **self-resident translators**. Others, known as **cross-translators**, generate code for machines other than the host machine. Cross-translators are often used in connection with microcomputers, especially in embedded systems which may themselves be too small to host self-resident translators satisfactorily. Of course, cross-translation introduces additional problems in connection with transferring the object code from the donor machine to the machine that is to execute the translated program, and can lead to delays and frustration in program development.

Figure 2.7 illustrates how one might develop an app for a smartphone by using a cross compiler. We assume that we have such a compiler that some cell phone company might have supplied to developers as part of an Intel based SDK (software development kit) that allows them to translate C source code into code for the Krait 400 processor used in the Galaxy S5. Once again, we defer discussion of how the 80x86 version of the extremely useful and valuable SDK is developed.

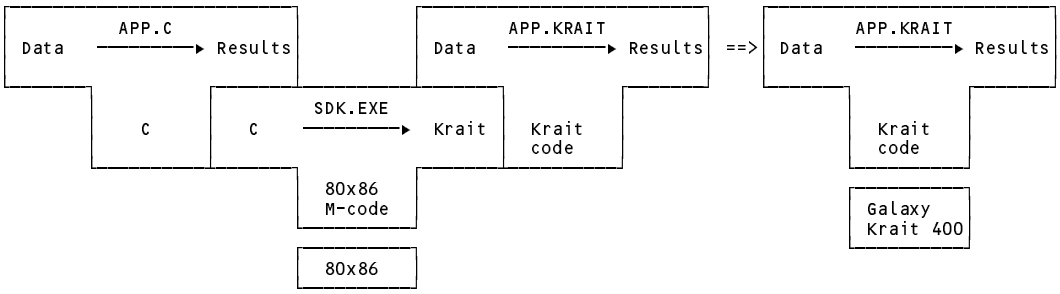


Figure 2.7 Development of an app for a Galaxy S5 incorporating a Krait 400 processor

2.4 Linkers and loaders

The output of some translators is absolute machine code, left loaded at fixed locations in a machine ready for immediate execution. Other translators, known as **load-and-go** translators, may even initiate execution of this code. However, a great many translators do not produce fixed-address machine code. Rather, they produce something closely akin to it known as **semicompiled**, **binary symbolic** or **relocatable** form. Code compiled in this way is then linked together by programs called **linkage editors**, **linkers** or **loaders**, which may be regarded almost as providing the final stage for a multi-stage translator. Languages that encourage the separate compilation of parts of a program - like Modula-2, Java, C# and C++ - depend critically on the existence of such linkers, as the reader may be aware.

A variation on this theme allows for the development of composite libraries of special purpose routines, possibly originating from a mixture of source languages. For developing really large software projects this approach has become indispensable - although for the sort of "throwaway" programs on which most students cut their teeth, it can initially appear to be a nuisance because of the overheads of managing several files and the time taken to link their contents together. Fortunately modern management systems can be used to hide most of the mechanism from the casual user.

T-diagrams can be combined to show the interdependence of translators, loaders and so on. For example, the Fitted Software Tools (FST) Modula-2 system made use of two stages - using a compiler followed by a linker, as shown in Figure 2.8. A user might not have been aware of this process, as simple operating system commands are usually developed to hide these aspects from users.

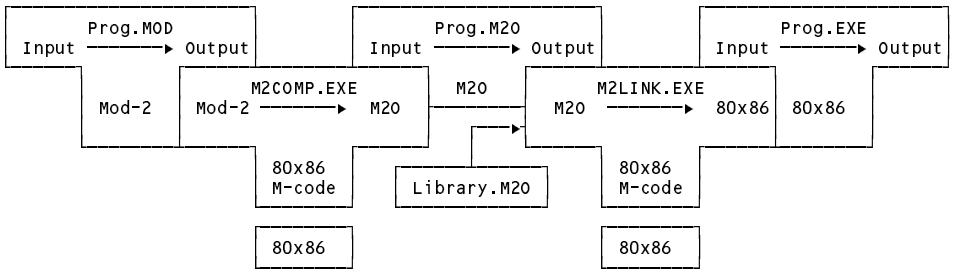


Figure 2.8 Compiling and linking Modula-2 program on the FST system

2.5 Phases in translation

Translators are highly complex programs and it is unreasonable to consider the translation process as occurring in a single step. It is usual to regard it as divided into a series of **phases**. The simplest breakdown recognizes that there is an **analytic phase** in which the source program is analyzed to determine whether it meets the syntactic and static semantic constraints imposed by the language. This is followed by a **synthetic phase** in which the corresponding object code is generated in the target language. The components of the translator that handle these two major phases are said to comprise the **front end** and the **back end** of the compiler. The front end is largely independent of the target machine, the back end depends very heavily on the target machine. Within this structure we can recognize smaller components or phases, as shown in Figure 2.9 (after Aho, Sethi, Ullman (1986)).

The **character handler** is the section that communicates with the outside world, through the operating system, to read in the characters that make up the source text. As character sets and file handling vary from system to system, this phase is often machine or operating system dependent.

The **lexical analyzer** or **scanner** is the section that fuses characters of the source text into groups that logically make up the **tokens** of the language - symbols like identifiers, strings, numeric constants, keywords like `while` and `if`, operators like `<=` and so on. Some of these symbols are very simply represented on delivery from the scanner, some need to be associated with various properties or **attributes** such as their names or values. Besides recognizing such tokens, a lexical analyzer is usually given the tasks of discarding extraneous inter-token white space (blanks), discarding comments and reacting to **pragmas** or **compile-time directives**. Pragmas merely give guidance to a compiler and are normally unrelated to the semantic meaning of the source program. They might indicate the exact processor for which code is to be generated in the situation where, as in the case of Intel processors, there is a range of similar processors to choose from, or the degree to which optimization is to be attempted, or whether a source listing is required and in what format this should be presented.

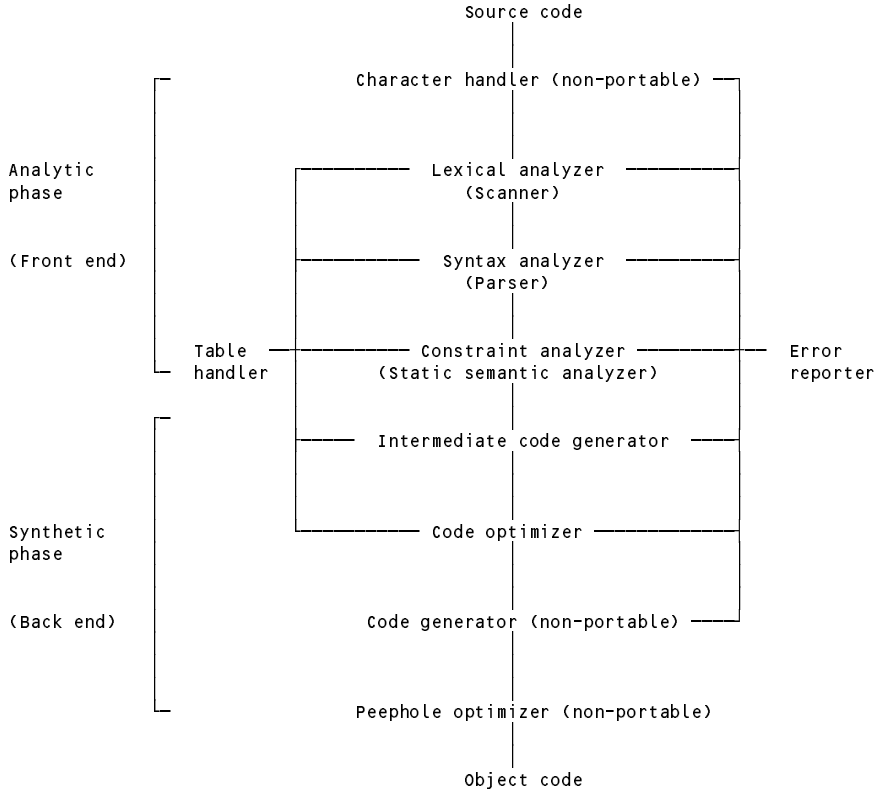


Figure 2.9 Structure and phases of a compiler

Lexical analysis is sometimes straightforward and at other times not. For example, the Java statement

```
while (1 < p && p < 9) p = p + q;
```

easily decodes into tokens

`while`	keyword	
`(`	left parenthesis	
`1`	integer literal	value 1
`<`	operator	comparison
`p`	identifier	name *p*
`&&`	operator	conjunction
`p`	identifier	name *p*
`<`	operator	comparison
`9`	integer literal	value 9
`)`	right parenthesis	
`p`	identifier	name *p*
`=`	operator	assignment
`p`	identifier	name *p*
`+`	operator	addition
`q`	identifier	name *q*
`;`	semicolon	

as we read it from left to right. But the Fortran statement

```
10      DO 20 I = 1 . 30
```

is more deceptive. Readers familiar with Fortran might see it as decoding into

`10`	label
`DO`	keyword
`20`	statement label
`I`	INTEGER identifier
`=`	assignment operator
`1`	INTEGER constant literal
`,`	separator
`30`	INTEGER constant literal

while those who enjoy perversity might like to see it as it really is:

`10`	label
`DO2OI`	REAL identifier
`=`	assignment operator
`1.30`	REAL constant literal

One has to look quite hard to distinguish the period from the "expected" comma. (Spaces are irrelevant in Fortran; one would, of course *be* perverse to use identifiers with unnecessary and highly suggestive spaces in them.) While virtually all modern languages have been designed so that lexical analysis can be clearly separated from the rest of the analysis, the same is not true of Fortran and other languages that do not have reserved keywords. In passing, there is a well-known tale that a Venus space probe is reputed to have been lost because of a mistyped DO statement like that just illustrated.

The **syntax analyzer** or **parser** determines whether and how the tokens produced by the scanner may be organised into syntactic structures. It does this by parsing expressions and statements in a manner analogous to the way a human might analyze the words in a sentence looking for components like "subject", "object" and "dependent clauses". Often the parser is combined with the **contextual constraint analyzer** whose job it is to determine that the components of the syntactic structures also satisfy such requirements as scope rules and type rules within the context of the program being analyzed. For example, in Modula-2 the syntax of a *while* statement is sometimes described as

`WHILE` *Expression* `DO` *StatementSequence* `END`

It is reasonable to think of a statement in the above form with any type of *Expression* as being syntactically correct, but as being devoid of real meaning unless the value of the *Expression* is constrained (in this context) to be of the Boolean type. No program really has any meaning until it is executed dynamically. However, it is possible with strongly typed languages to predict at compile-time that some source programs can have no sensible meaning (that is, statically, before an attempt is made to execute the program dynamically). Semantics is a term

used to describe "meaning" and so the constraint analyzer is sometimes called the **static semantic analyzer**, or simply the semantic analyzer.

The output of the syntax analysis and constraint analysis phases is sometimes expressed in the form of a decorated **abstract syntax tree** (AST). This is a very useful representation as it can be used in clever ways to optimize code generation at a later stage.

Whereas the **concrete syntax** of many programming languages incorporates many keywords and tokens, the **abstract syntax** is rather simpler, retaining only those aspects of the language needed to capture the real content and, ultimately, the meaning of a program. For example, whereas the concrete syntax of a Modula-2 *while* statement requires the presence of `WHILE`, `DO` and `END` as shown above, the essential components of the *while* statement are simply the (Boolean) *Expression* and the statements comprising the *StatementSequence*.

Thus the Modula-2 statement

```
WHILE  (1 < p)  AND  (p < 9)  DO  p := p + q  END
```

and its possibly more familiar Java, C# or C++ equivalents

```
while  (1 < p && p < 9)  p = p + q;
```

are all depicted by the common AST shown in Figure 2.10.

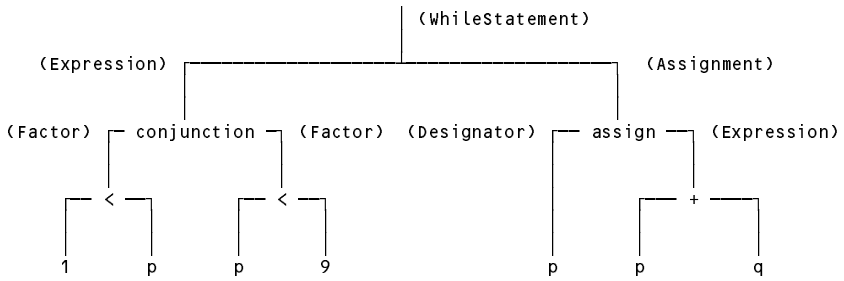


Figure 2.10 AST for the statement while (1 < p && p < 9) p = p + q;

An AST on its own is devoid of some semantic detail - the semantic analyzer has the task of adding "type" and other contextual information to the various nodes (hence the term "decorated" tree).

Sometimes, as for example in the case of most Pascal compilers and, indeed, the compiler we shall later develop in detail as a complete case study, the construction of such a tree is not explicit, but remains implicit in the sequence of (possibly recursive) calls to the procedures that perform the syntax and semantic analysis. Thus, in the rest of this text we shall frequently refer to the "underlying AST" for a program even though we have not built such a tree in the way readers may have encountered in courses in data structures.

Of course, it is also possible to construct concrete syntax trees. The Modula-2 form of the statement

```
WHILE  (1 < p)  AND  (p < 9)  DO  p := p + q  END
```

could be depicted in full and tedious detail by the tree shown in Figure 2.11. The reader may have to make reference to Modula-2 syntax diagrams and a knowledge of Modula-2 precedence rules to understand why the tree looks so complicated.

The phases just discussed are all analytic in nature. The ones that follow are more synthetic. The first of these might be an **intermediate code generator** which, in practice, may also be integrated with earlier phases, or omitted altogether in the case of some very simple translators like the ones we shall study in detail. It uses the data structures produced by the earlier phases to generate a form of code, perhaps in the form of simple code skeletons or macros, or ASSEMBLER or even high-level code for processing by an external assembler or separate compiler. The major difference between intermediate code and actual machine code is that intermediate code need not specify in detail such things as the exact machine registers to be used, the exact addresses to be referred to, and so on.

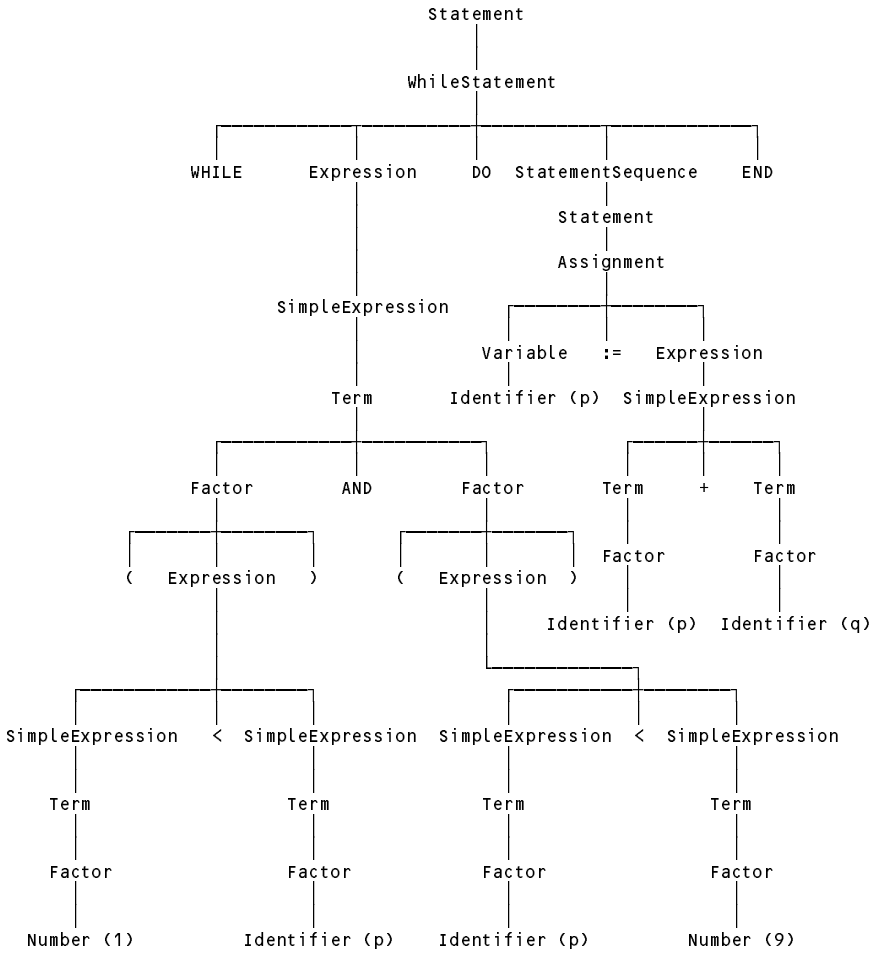


Figure 2.11 Concrete syntax tree for the Modula-2 statement
WHILE (1 < p) AND (p < 9) DO p := p + q END

Our example statement

```
while (1 < p && p < 9)  p = p + q;
```

might produce intermediate code equivalent to

```
L0      if 1 < p goto L1
        goto L3
L1      if p < 9 goto L2
        goto L3
L2      p = p + q
        goto L0
L3      continue
```

Then again, it might produce something like

```
L0      T1 = 1 < p
        T2 = p < 9
        if T1 and T2 goto L1
        goto L2
L1      p = p + q
        goto L0
L2      continue
```

depending on whether the implementors of the translator use the so-called *sequential conjunction* or *short-circuit* approach to handling compound Boolean expressions (as in the first case) or the so-called *Boolean operator* approach. The reader will recall that Modula-2, C, C++, C# and Java all require the short-circuit approach, which can be succinctly summarised in the two logical equivalences

A and B ≡ if A then B else false (or if !A then false else B)
A or B ≡ if A then true else B (or if !A then B else false)

Unfortunately, the very similar language Pascal did not specify that one approach be preferred above the other.

A **code optimizer** may optionally be provided, in an attempt to improve the intermediate code in the interests of speed or space or both. To use the same example as before, obvious improvement would lead to code equivalent to

```
L0      if 1 ≥ p goto L1
        if p ≥ 9 goto L1
        p = p + q
        goto L0
L1      continue
```

The most important phase in the back end is the responsibility of the **code generator**. In a real compiler this phase takes the output from the previous phase and produces the object code. If an explicit AST has been constructed, this code generator might "walk this tree" as it completes such tasks as deciding on the memory locations for data, generating code to access such locations, selecting registers to be used for intermediate calculations and indexing, and so on. Clearly this is a phase which calls for much skill and attention to detail if the finished product is to be at all efficient. Some translators go on to a further phase by incorporating a so-called **peephole optimizer**, in which attempts are made to reduce unnecessary operations still further by examining short sequences of generated code in closer detail.

As it proceeds with the translation of a program, a translator inevitably builds a complex data structure, traditionally known as the **symbol table**, in which it keeps track of the names or identifiers used by the program and associated properties for these, such as their type and their storage requirements (in the case of variables and fields), their values (in the case of constants) or the number and type of their parameters (in the case of functions and methods).

Figure 2.9 seems to imply that compilers work serially and that each phase communicates with the next by means of a suitable intermediate language, but in practice the distinction between the various phases often becomes a little blurred. Moreover, many compilers are actually constructed around a central parser as the dominant component, with a structure rather more like the one in Figure 2.12.

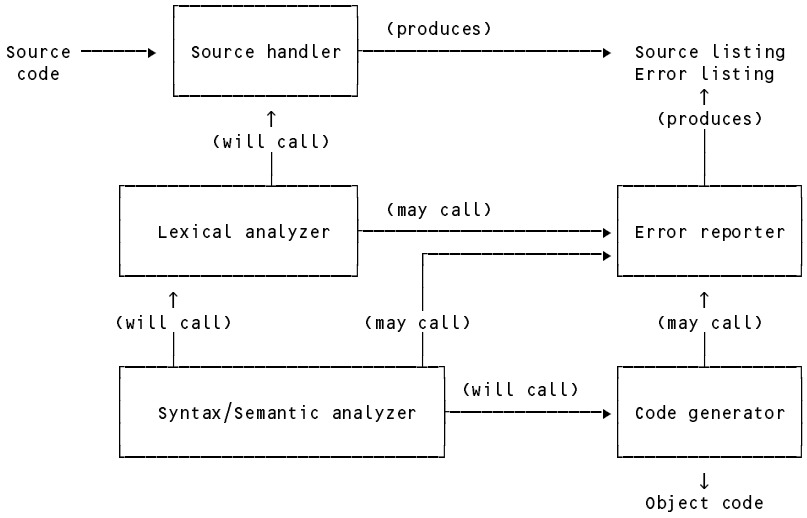


Figure 2.12 Structure of a parser-directed compiler

As is well known, users of high-level languages are apt to make many errors in the development of even quite simple programs. Thus the various phases of a compiler, especially the earlier ones, also communicate with an **error handler** and **error reporter** which are invoked when errors are detected. It is desirable that compilation of erroneous programs be continued, if possible, so that the user can clean several errors out of the source before recompiling. This raises very interesting issues regarding the design of **error recovery** and **error correction** techniques. (We speak of error recovery when the translation process attempts to carry on after detecting an error, and of error correction or error repair when it attempts to correct the error from context - usually a

contentious subject, as the correction may be nothing like what the programmer originally had in mind.)

Error detection at compile-time in the source code must not be confused with error detection at run-time when executing the object code. Many code generators are responsible for adding error-checking code to the object program (to check that subscripts for arrays stay in bounds, for example). This may be quite rudimentary or it may involve adding considerable code and data structures for use with sophisticated debugging systems. Such ancillary code can drastically reduce the efficiency of a program, and some compilers allow it to be suppressed.

Sometimes mistakes in a program that are detected at compile-time are known as *errors*, while errors that show up at run-time are known as *exceptions*. The latter term has come into prominence with the advent of languages like Java and C# in which the programmer is given access to language features for "throwing" and "catching" objects of this sort when run-time problems arise.

2.6 Multi-stage translators

Besides being conceptually divided into phases, translators are sometimes divided into **passes**, in each of which several phases may be combined or interleaved. In its simplest form, this technique was more common in days gone by - traditionally, a pass read the source program, or output from a previous pass, made some transformations and then wrote output to an intermediate file, whence it was rescanned on a subsequent pass.

Such passes may be handled by different integrated parts of a single compiler or they may be handled by running two or more separate programs. They may communicate by using their own specialized forms of intermediate language, they may communicate by making use of internal data structures (rather than files) or they may make several passes over the same original source code.

The number of passes used depends on a variety of factors. Certain languages seem to require at least two passes to be made if code is to be generated easily - for example, those where declaration of identifiers may occur after the first reference to the identifier, or where properties associated with an identifier cannot be readily deduced from the context in which it first appears. Although modern computers are usually blessed with far more memory than their predecessors of only a few years back, multiple passes may be an important consideration if one wishes to translate complicated languages within the confines of small systems. Multi-pass compilers may also allow for better provision of code optimization, error reporting and error handling. Lastly, they lend themselves to team development, with different members of the team assuming responsibility for different passes. However, if they need to keep track of several files, multi-pass compilers may be awkward to write and to use and are invariably slower than single-pass compilers. Compromises at the design stage often result in languages that are well suited to single-pass compilation, and improvements in compiler technology have led to the currently favoured model where the achievements of multiple passes may be realized by repeatedly walking internally constructed tree structures rather than processing a sequence of files. In passing, we note that the extreme examples of single-pass compilation are to be found in what some authors term **incremental compilers** (Gough 2002) in which the phase of code generation begins even before the first pass has been completed.

However, in practice considerable use is still made of two-stage translators in which the first stage is a high-level translator that converts the source program into ASSEMBLER, or even into some other relatively high-level language for which an efficient translator already exists. The compilation process would then be depicted as in Figure 2.13 - our example shows a Modula-3 program being prepared for execution on a machine that has a Modula-3 to C high-level compiler and a C to machine code compiler.

In recent years it was common to find compilers for high-level languages that were implemented using C, and which themselves produce C code as output. The success of these was based on the premises that "all modern computers come equipped with a C compiler" and "source code written in C is truly portable". Neither premise is, unfortunately, completely true. However, compilers written in this way came close to achieving the dream of themselves being portable. The way in which such compilers may be used is discussed further in Chapter 3.

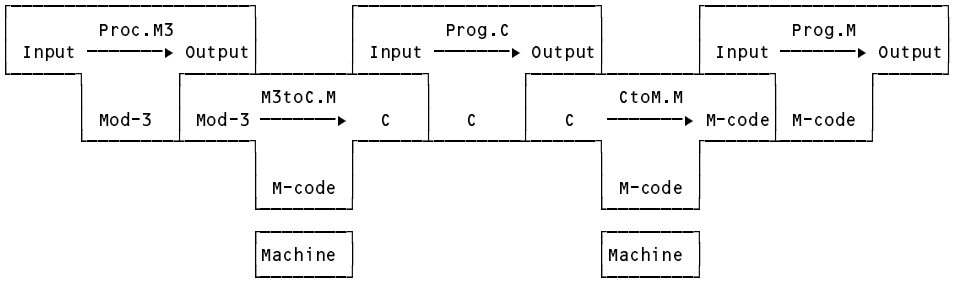


Figure 2.13 Compiling Modula-3 by using C as an intermediate language

2.7 Interpreters, interpretive compilers, and emulators

Compilers of the sort that we have been discussing have a few properties that may not immediately be apparent. Firstly, they usually aim to produce object code that can run at the full speed of the target machine. Secondly, they are usually arranged to compile an entire section of code before any of it can be executed.

In some interactive environments the need arises for systems that can execute part of an application without preparing all of it, or ones that allow users to vary their course of action on the fly. Typical scenarios involve the use of spreadsheets, on-line databases, or batch files or shell scripts for operating systems. With such systems it may be feasible (or even desirable) to exchange some of the advantages of speed of execution for the advantage of procuring results on demand.

Systems like these are often constructed so as to make use of an **interpreter**. An interpreter is a translator that effectively accepts a source program and executes it directly, without, seemingly, producing any object code first. It does this by fetching the source program instructions one by one, analyzing them one by one and then "executing" them one by one. Clearly, a scheme like this, if it is to be successful, places some quite severe constraints on the nature of the source program. Complex program structures, such as nested procedures or compound statements, may not lend themselves easily to such treatment. On the other hand, one-line queries made of a data base, or simple manipulations of a row or column of a spreadsheet, can be handled very effectively.

This idea is taken quite a lot further in the development of some translators for high-level languages, known as **interpretive compilers**. Such translators produce (as output) "pseudo code", which is intrinsically simple enough to satisfy the constraints imposed by a practical interpreter, even though it may still be quite a long way from the machine code of the system on which it is desired to execute the program being compiled. Rather than continue translation to the level of machine code, an alternative approach that may perform acceptably well is to use the intermediate pseudo-code as part of the input to a specially written interpreter. This in turn "executes" the original algorithm, by simulating a hypothetical machine for which the pseudo-code effectively *is* the machine code. The distinction between the machine code and pseudo-code approaches to execution is summarized in Figure 2.14.

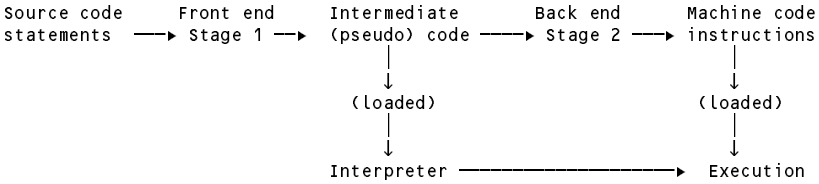


Figure 2.14 Differences between native code and pseudo-code compilers

It is not necessary to restrict interpreters to programs that process output from a translator or as substitutes for the back end of a "real" compiler. More generally, of course, even a real machine can be viewed as a highly specialized interpreter - one that executes the machine-level instructions in a program by fetching, analyzing and then interpreting them one by one. In a real machine this all happens "in hardware", and hence very quickly. By carrying on this train of thought, the reader should be able to see that a program could be written to allow one real machine to emulate any other real machine, albeit perhaps slowly, simply by writing an interpreter for the machine-level instructions of the second machine, giving rise to what is usually called an **emulator** or **virtual machine**.

For example, we might develop an emulator that runs on a Sun SPARC machine and makes it appear to be an IBM PC (or the other way around). Once we have done this, we are (in principle) in a position to execute any software developed for an IBM PC on the Sun SPARC machine - effectively the PC software becomes portable!

The interpreter/emulator approach is widely used in the design and development both of new machines themselves and the software that is to run on those machines. It has an instant appeal in several respects.

- It is far easier to generate code for an idealised hypothetical machine (which can be tailored towards the quirks of the original source language) than real machine code (which has to deal with the uncompromising quirks of real machines).

- A compiler written to produce (as output) well-defined pseudo-machine code capable of easy interpretation on a range of machines can be made highly portable, especially if it is written in a host language that is widely available (such as ANSI/ISO C), or even if it is made available already implemented in its own pseudo-code.

- It may be possible to implement a whole range of languages quickly and easily on a wide range of different machines. This is done by producing pseudo-code to a well-defined standard, for which a relatively efficient interpreter should be easy to implement on any particular real machine.

- It proves to be useful in connection with cross-translators such as were mentioned earlier. The code produced by such translators can sometimes be tested more effectively by simulated execution on the donor machine, rather than after transfer to the target machine - the delays inherent in the transfer from one machine to the other may be balanced by the degradation of execution time in an interpretive simulation.

- As we shall see, pseudo-code instructions are often very compact, allowing large programs to be handled, even on relatively small machines.

- Lastly, case studies of such compilers fit readily within the constraints of a textbook such as this, as we hope to demonstrate in later chapters where we illustrate the development of a compiler for a small C-like language, together with a suitable interpreter and virtual machine for the pseudo-code that it generates.

For all these advantages, interpretive systems carry fairly obvious overheads in execution speed, because execution of pseudo-code effectively carries with it the cost of virtual translation into machine code each time a hypothetical machine instruction is obeyed.

2.8 The P-system and the JVM

Perhaps the best known of the early **portable interpretive compilers** was the one developed in Zürich and known as the "Pascal-P" compiler (Nori *et al.* 1981). This was supplied in a kit of three components.

- The first component was the source code for a Pascal compiler, written in a very complete subset of the language, known as Pascal-P. The aim of this compiler was to translate Pascal-P source programs into a well-defined and well-documented intermediate language, known as P-code, which was the "machine code" for a hypothetical stack-based computer known as the P-machine.

- The second component was a compiled version of the first - the P-code that would be produced by the Pascal-P compiler, were it able to compile itself.

- Lastly, the kit contained an interpreter for the P-code language, supplied as a Pascal algorithm.

At first this may appear a bit baffling. Academics could get a kit to build a Pascal compiler, but which seemed to require them to possess a Pascal compiler already - which they did not have. This calls for explanation.

The interpreter served primarily as a model for writing a similar program for the target machine to allow it to emulate the hypothetical P-machine. As we shall see in chapter 4, emulators are relatively easy to develop -even, if necessary, in ASSEMBLER - so that creating such an interpreter was usually fairly painlessly achieved. Once one had plugged the interpreter - that is to say, the version of it tailored to execute on a local real machine - into a real machine, one was in a position to "execute" P-code, and in particular the P-code of the P-compiler. The compilation and execution of a user program could then be achieved in a manner depicted in Figure 2.15, where

we see the T-diagram notation extended to handle the concept of a virtual machine.

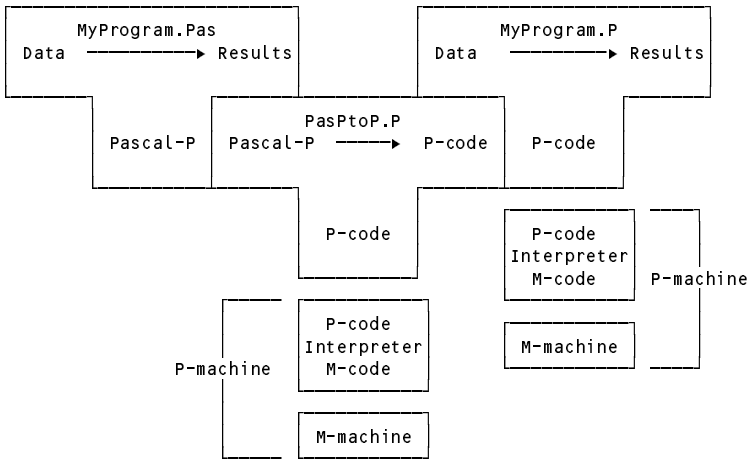


Figure 2.15 Compiling and executing a program with the P-compiler

Notice that, as yet, no mention has been made of any practical use for the first component of the kit. All will be revealed in time!

The Zürich P-compiler could be, and indeed was, used as a highly portable development system. It was employed to remarkable effect in developing the UCSD Pascal system at the University of California in San Diego. This represented the first serious attempt to implement Pascal on microcomputers. The UCSD Pascal team went on to provide the framework for an entire operating system, editors and other utilities - all written in Pascal and all compiled into a well-defined P-code object code that could fit within the small memory capacities of the time. Simply by providing an suitable interpreter, one could move the whole system to a new microcomputer system virtually unchanged.

A more recent exploitation of this form of compiler technology has been largely responsible for the spectacular rise to prominence of Java as a language, and of the associated Java platform as one that represents almost the last word in safe, reliable portability. The Java platform was originally designed in the early 1990s to support small, networked, embedded applications, but the coincidence of this development with the exploitation of the World Wide Web as a delivery mechanism for data and programs, and of the firm establishment of object-orientation as a design tool, have secured it a far greater role in the annals of computing history. As in the case of the P-system, Java compilers are written to target a virtual machine, the JVM (Java Virtual Machine), which represents a significant advance on earlier machines of this sort. The JVM is not defined in terms of an algorithm, but in terms of a detailed specification at a fairly abstract level, which is not restricted simply to describing the semantics of a set of low-level instructions or **bytecodes** for the machine. The specification also prescribes a strict format for a **class file**, which is a bytestream (effectively an object file) that a compiler should produce for each module of code (or in Java terms, each class) that it compiles. A class file incorporates not only the important bytecode instruction stream but also infrastructural information relating to the types, methods and fields of each class - in effect, a symbol table. The combination enables an important part of the virtual machine, the **class loader** and **bytecode verifier**, to ensure - before execution commences - that the class file as a whole represents a memory-safe section of program that is capable of being processed by the execution engine without disaster, and with comparatively little overhead in run-time checking.

Significantly, implementations of the JVM have found their way into web browsers, which allows for Java class files to be transmitted as "applet" components of web pages, and, if successfully loaded, executed all around the globe. And, lastly, the very wide range of APIs (Application Program Interfaces) that have been made widely and freely available have solved many of the problems of developing large systems from scratch. The combination of an implementation of the JVM and the Java API (which may include so-called native methods that give access to features of the underlying real machine in a controlled way) is known as a **Java platform**.

Mention has already been made of how Java was welcomed by programmers as a great improvement over C++, which by the time Java was released had accreted so many features that it had become unwieldy and, in the opinion of many, difficult to master. Some observers have noted that the most important reasons for the popularity of Java are not really properties of the language but of the underlying system (Gough 2002).

Understandably, therefore, the portability and safety features of the Java platform have made it a most attractive target for compiler writers for other languages. However, although the format of the class file is not tightly bound to the Java language, the bytecodes themselves and the underlying execution engine were designed with ease of implementation of Java programs as the primary objective. As a target for compilers for other source languages - which might incorporate features such as passing parameters by reference that are not found in Java - the instruction set of the JVM has some awkward omissions, although a number of compilers for type-safe languages such as Component Pascal have been produced (Gough and Corney 2000). Terry (2005) also explores the creation of a compiler for a small C#-like language (C# Minor) that targets the JVM. Rather than handle all the detailed low-level ramifications of the class file format, that study is limited to producing assembler-level code that can be processed further by a JVM assembler program such as *Jasmin* (Meyer and Downing 1997) or *Oolong* (Engel 1999).

2.9 JIT compilers and the .NET Framework

The specification of the JVM did not prescribe that its implementation should take the form of an interpreter, although several implementations were developed in this manner. More recent implementations have incorporated a **JIT** (just in time) component. Effectively this is a system for performing a translation from bytecodes to the machine language of the computer implementing the JVM execution engine, which, once performed, means that methods so translated will run at the full speed of the machine and not at the somewhat crippled speed of an interpreter. Very sophisticated implementations of this idea go so far as to undertake this compilation (and any optimizations that seem needed on the code) only as and when necessary. It is believed that, in time, systems developed to embrace such **adaptive optimization** techniques will be as efficient as those compiled in more traditional ways.

Another development that relies heavily on the JIT philosophy is to be found in the .NET Framework announced by Microsoft in 2000. This framework has many facets, but common to them all is the idea that compilers for languages supported by the system should all generate **assemblies** similar in concept to jar files for the JVM. These incorporate a stream of instructions for a well-defined virtual machine in what is called Common Intermediate Language (CIL), sometimes called Microsoft Intermediate Language (MSIL), along with **metadata** describing the classes, fields and methods from which the assemblies have been produced. As in the case of the JVM, the assemblies can be checked for safety at load time and subsequently be "managed" by the CLR system as they are executed. Finally, the vast Framework Class Library (FCL) provides a great many useful classes, on the lines of those found in the Java API.

The CLR and the JVM at first appear to have quite a lot in common. However, whereas the JVM was designed specifically with the needs of the Java programmer and compiler writer in mind, the philosophy of the CLR is that it should be able to handle code originating from a wide variety of languages. Not only does the CIL instruction set support operations that have no counterpart in the JVM and allow, for example, parameters easily to be passed by reference, but the system also makes provision for incorporating so-called "unmanaged code" that can bypass the verification algorithms and provide for operations that the JVM would regard as totally unsafe. To this end, the Common Type System (CTS) defines the types that can be components of metadata and the operations that CIL code can perform on these. A subset of this extensive type system, the Common Language Specification (CLS), can be used to ensure that assemblies compiled from one language can use library code produced by another language - even to the extent of their inheriting from one another's classes.

To enable all this generality, it turns out that an implementation of the CLR would find it almost impossible to follow an interpretive approach. Instead, the CLR relies on the presence of a JIT compiler to perform the last stage of compilation to native code as each assembly is loaded. The final machine code is, however, rarely saved in that form, the system preferring to repeat the JIT process each time the application is run.

As we shall discuss in later chapters, the underlying machine model is similar for both systems. The instruction sets of the virtual machines provide direct support for concepts such as exception handling. In particular, both require a run-time execution model that is object-oriented in that it treats machine memory as being a collection of objects rather than an array of bytes, and manages this collection with the ability to perform so-called **garbage collection** when an object is no longer required. This feature removes from the high-level programmer the responsibility for handling what had hitherto been one of the most prolific causes of elusive bugs.

Targeting the JVM or the CLR imposes other constraints on compiler writers that are not at first apparent. Not only is it necessary to produce low-level code that reflects the semantics of the source program exactly, but this code has to be be able to satisfy the verification process, which is fairly conservative in what it will sanction.

Terry (2005) also explores targeting the CLR for the simple C#-like language (C# Minor) - as in the case of the compiler for the JVM, it is content to produce assembler-level code that can be turned into true CLR assemblies by making use of the CLR assembler (*ilasm*).

Further reading

As we progress we shall revisit the material of this chapter on numerous occasions. Most books on compiler writing have introductory chapters similar to this one and the reader may find the discussion in books by Louden (1997), Grune *et al*. (2000), Watt and Brown (2000) and Parsons (1992) helpful. Material on the UCSD Pascal system can be found at the on-line Jefferson Computer Museum at `http://www.threedee.com/jcm/psystem/`. Good introductions to the features of the JVM can be found in the books by Engel (1999) and Venners (1999), while the specification of the JVM appears in the authoritative book by Lindholm and Yellin (1999). A list of compilers that target the JVM is maintained by Tolksdorf at `http://www.robert-tolksdorf.de/vmlanguages`. The inner workings of the .NET system are documented at `http://msdn.microsoft.com/net/ecma/`. A useful introduction to the .NET Framework is to be found in the book by Drayton, Albahari and Neward (2002). The excellent book by Mössenböck *et al*. (2004) discusses many aspects of application development for .NET, as well as providing a useful overview of C#. A superb guide to the development of compilers that target the CLR has been produced by Gough (2002). A list of languages for which compilers have been released for the .NET system can be found at `http://en.wikipedia.org/wiki/List_of_CLI_languages`.

3 COMPILER DEVELOPMENT AND BOOTSTRAPPING

By now the reader may have realized that developing translators is a decidedly non-trivial exercise. If one is faced with the task of writing a full-blown translator for a fairly complex source language - or an emulator for a new virtual machine, or an interpreter for a low-level intermediate language - one would probably prefer not to implement it all in machine code.

Fortunately one rarely has to contemplate such a radical step. Translator systems are now widely available and well understood. A fairly obvious strategy when a translator is required for an old language on a new machine, or a new language on an old machine (or even a new language on a new machine), is to make use of existing compilers on either machine and to do the development in a high-level language. This short chapter provides a few examples that should clarify this process

3.1 Using a high-level language to develop a compiler for a new high-level language

If, as is increasingly common, one's dream machine *M* is supplied with the machine coded version of a compiler for a well-established language like C, then the production of a compiler for one's dream language *Great* is achievable by writing the new compiler, say *GtoM*, in C and compiling its source (*GtoM.C*) with the C compiler (*CtoM.M*) running directly on *M* (see Figure 3.1). This produces the object version (*GtoM.M*) which can then be executed by plugging it into *M*.

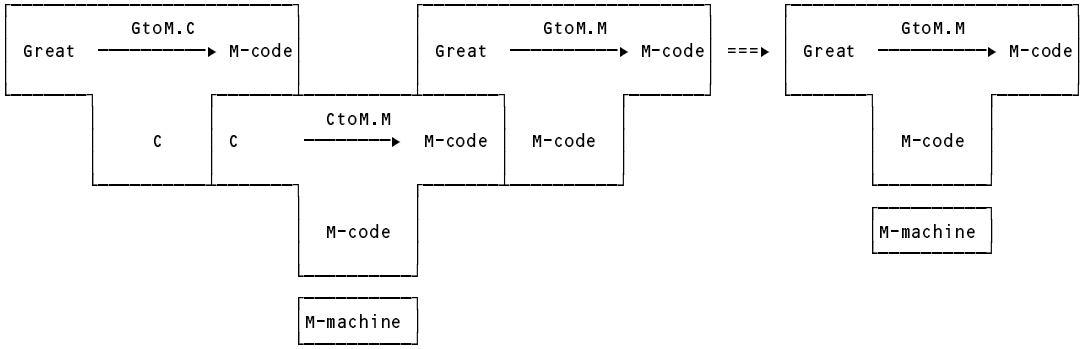


Figure 3.1 Use of C as an implementation language

Even though development in C is much easier than development in machine code, the process is still tedious. The hardest part of the development is probably that associated with the back end, since this is intensely machine dependent. If one has access to the source code of a compiler like *CtoM* one may be able to use this for inspiration. Although commercial compilers are rarely released in source form, source code is available for many compilers produced at academic institutions or as components of the GNU (GNU's not UNIX) project carried out under the auspices of the Free Software Foundation.

3.2 Using a high-level language to develop a two-stage compiler for a new language

Given that one has the *CtoM.M* compiler at one's disposal, rather than attempt to write a whole native code compiler for our Great new language, one might write a high-level translator to convert programs written in *Great* to C - again using C as a development host language. If this high-level compiler, say *GtoC.C*, is compiled with the native code C compiler *CtoM.M* we produce an executable *GtoC.M*, which can then be used as the first stage of a compiler system. The native code compiler *CtoM.M* is then used to complete the last stage of the compilation.

This is all conveniently represented in terms of T-diagrams chained together. Figure 3.2(a) shows the compilation of the Great to C compiler, and Figure 3.2(b) shows the two-stage compilation process needed to compile programs written in Great to *M*-code. Figure 3.2(c) shows a diagram for the effective compiler.

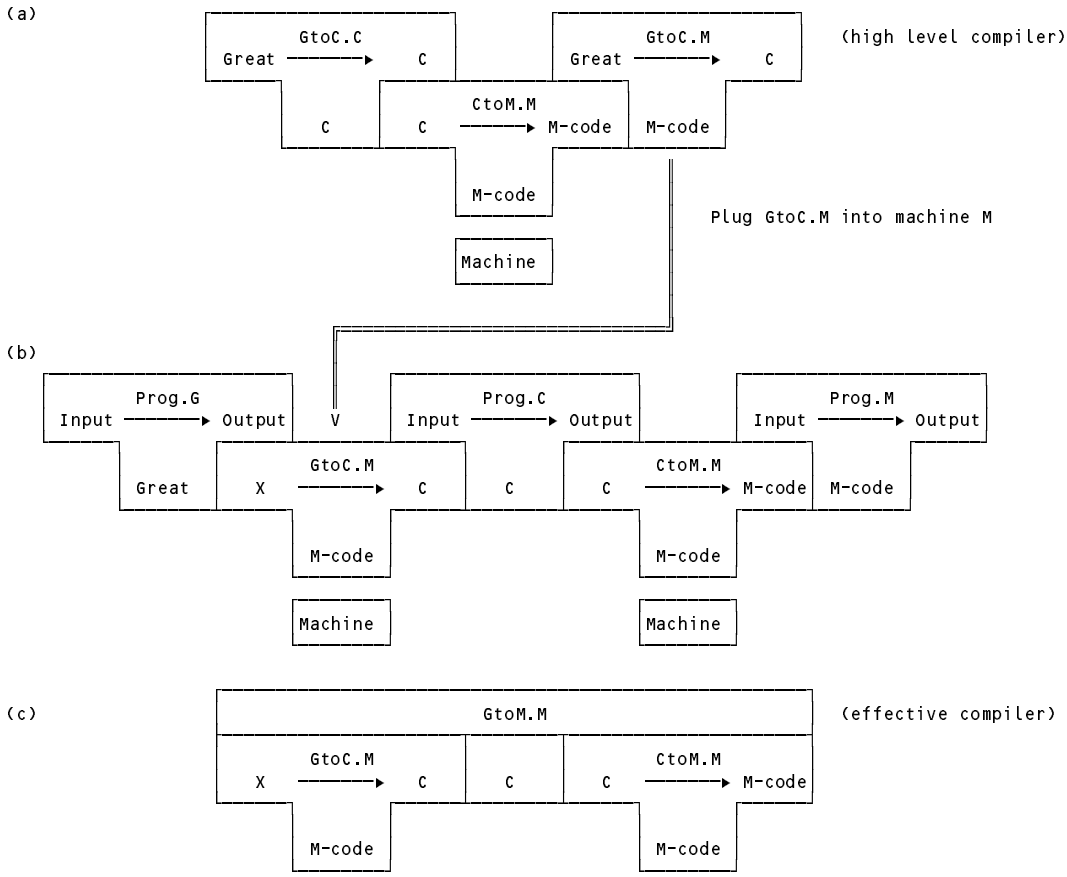


Figure 3.2 Creating a two stage compiler using C as an implementation language

The process of modifying an existing compiler to work on a new machine is often known as **porting** the compiler.

A compiler like *GtoC.C* is well worth developing - its portability is almost guaranteed, provided that it is itself written in "portable" C. This means that the Great compiler can easily be "ported" to any other machine *N* that has a C compiler. All that one would have to do is to recompile the *GtoC.C* component and use the *CtoN.N* compiler as the back end on machine *N*. In principle use of our Great language could spread like wildfire …

Unfortunately, or as Mr. Murphy would put it, "interchangeable parts don't" (more explicitly, "portable C isn't"). Some time may have to be spent in modifying the source code of *GtoC.C* before it is acceptable as input to another C compiler, *CtoN.N*, although it is to be hoped that the developers of *GtoC.C* will have used only standard C in their work and used pre-processor directives that allow for easy adaptation to other systems.

If there is an initial strong motivation for making a compiler portable to other systems it is, indeed, often first written so as to produce high-level code as output. More often, the original implementation of a language is refined slowly to yield a self-resident translator which directly produces machine code for the host system.

3.3 Bootstrapping

All this may seem to be skirting around a really nasty issue - how might this C compiler have been developed? Well, presumable we might have tried using an existing compiler for a language that preceded C, like FORTRAN. Yes, all right, but how was the FORTRAN compiler written. Probably very tediously, in ASSEMBLER. But then how was the assembler for ASSEMBLER produced? And so on!

A full assembler is itself a major piece of software, albeit rather simple when compared with a compiler for a really high-level language, as we shall see. It is, however, quite common to define one language as a subset of another, so that subset 1 is contained in subset 2 which in turn is contained in subset 3 and so on, that is

Subset 1 of ASSEMBLER	⊆	Subset 2 of ASSEMBLER	⊆	Subset 3 of ASSEMBLER

One might first write an assembler for subset 1 of ASSEMBLER in machine code, perhaps on a load-and-go basis (more likely one writes in ASSEMBLER and then hand translates it into machine code). This subset assembler program might, perhaps, do very little other than convert mnemonic opcodes into binary form. One might then write an assembler for subset 2 of ASSEMBLER using only the facilities in subset 1 of ASSEMBLER, and so on.

This process, by which a simple language is used to translate a slightly more complicated program - which in turn may handle an even more complicated program and so on - is known as **bootstrapping**, by analogy with the idea that it might be possible to lift oneself off the ground by tugging at one's boot-straps.

In the modern era there are plenty of sophisticated assemblers and higher level compilers available, so the probability of having to develop a compiler from scratch using machine code is very low indeed. Of course, for any new machine the compiler writer has to come to terms with its new architecture and machine-level instructions, but here too the use of interpreters and emulators can help simplify the task.

3.4 Self-compiling compilers

Another new idea that may take some time to sink in: once one has a working compiler for a given language on a given machine, one can start using it to improve itself (a variation on the bootstrapping idea introduced in the previous section). Many compilers for popular languages were first hosted in another implementation language, as implied in section 3.1, and then their source was manually recoded in the same source language as they were intended to compile. The rewrite gives source for a *new* compiler that can then be compiled with the *old* compiler - the one written in the original implementation language. This is illustrated in Figure 3.3.

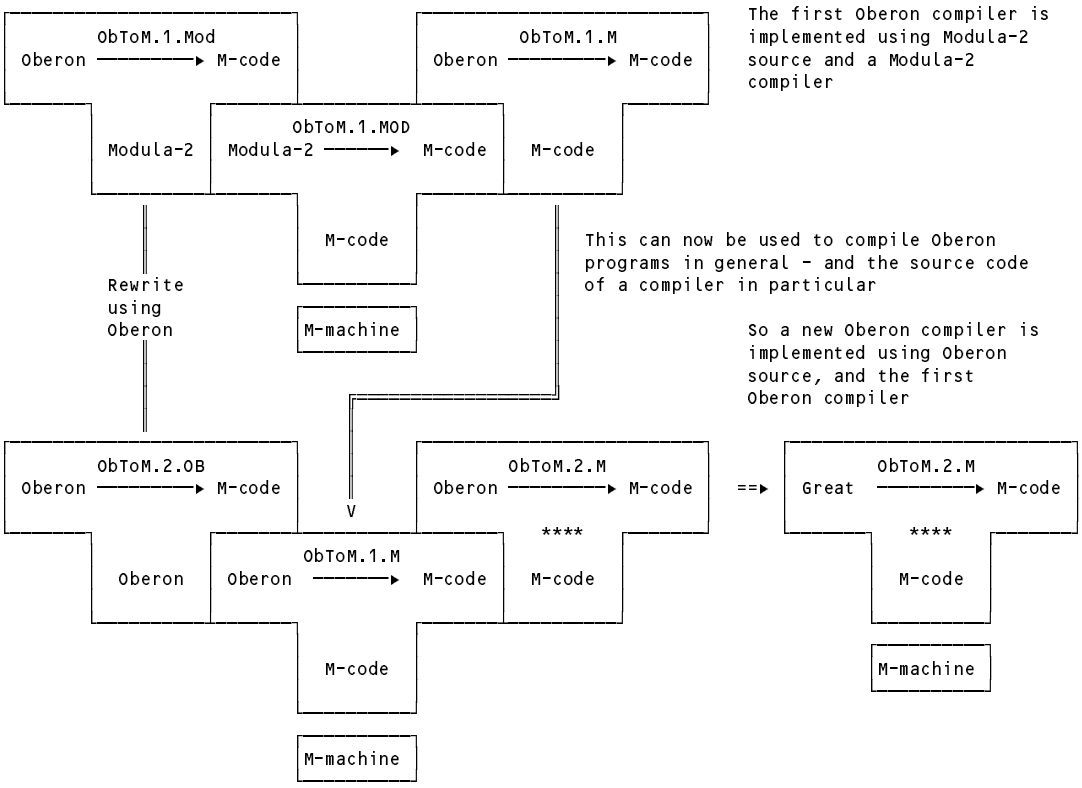


Figure 3.3 Steps in developing a self-compiling compiler

Clearly, writing a compiler by hand not once but twice is a non-trivial operation, unless the original implementation language is close to the source language that is to be implemented. This is not uncommon - Oberon compilers could be implemented in Modula-2; Modula-2 compilers, in turn, were first implemented in Pascal (all three are fairly similar languages); C++ compilers in C; Java and C# compilers in C++.

Developing a self-compiling compiler has several distinct points to recommend it. Firstly, it constitutes a non-trivial test of the viability of the language being proposed. Secondly, once it has been done, further development can be done without recourse to other translator systems. Thirdly, any improvements that can be made to its back end manifest themselves both as improvements to the object code it produces for general programs and as improvements to the compiler itself. Fourthly, it provides a fairly exhaustive self-consistency check - if the compiler is used to compile its own source code, it should, of course, be able to reproduce its own object code (see Figure 3.4). Lastly. no computer program (including a compiler) can hope to be entirely error-free. If an error is discovered in a self-compiling compiler it is usually possible to recompile an error-free version of the compiler with the buggy compiler and not have to resort to clumsy patches of machine-level code.

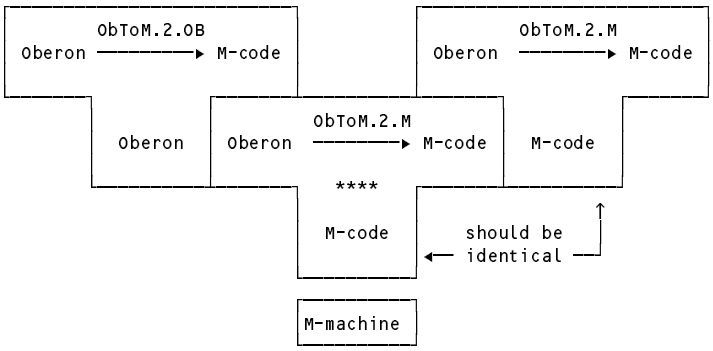


Figure 3.4 A self-compiling compiler must be self-consistent

Furthermore, given a working self-compiling compiler for a high-level language it is then very easy to produce compilers for specialized dialects of that language.

3.5 The half bootstrap - porting a compiler to a different machine

As a variation on what we have just discussed (developing a compiler for a new language on an old machine), let us consider how we write a compiler for an established language, but that must run on a new machine.

Self-resident compilers written to produce object code for a particular machine are not intrinsically portable. However, they are often used to assist in a porting operation in various ways. For example, by the time that the first Pascal compiler was required for ICL machines, the Pascal compiler available in Zürich (where Pascal had first been implemented on CDC mainframes) existed in two forms (Figure 3.5). The reader should recognise that the CDC Pascal compiler had reached the maturity of being a self-compiling compiler, of course.

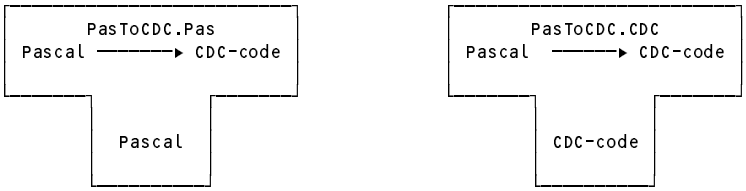


Figure 3.5 Two versions of the original Zürich self-compiling Pascal compiler

The first stage of the transportation process involved changing the back end of *PasToCDC.Pas* to generate ICL machine code - a **retargeting** operation, producing a cross-compiler. Since *PasToCDC.Pas* had been written in a high-level language, this was not too difficult to do, and resulted in the compiler *PasToICL.Pas*.

Of course, this compiler could not yet run on any machine at all. It was first compiled using *PasToCDC.CDC*, on the CDC machine (see Figure 3.6(a)). This gave a cross-compiler that could run on CDC machines, but still not, of course, on ICL machines. One further compilation of *PasToICL.Pas*, using the new cross-compiler *PasToICL.CDC* on the CDC machine, produced the final result, *PasToICL.ICL* (Figure 3.6(b)).

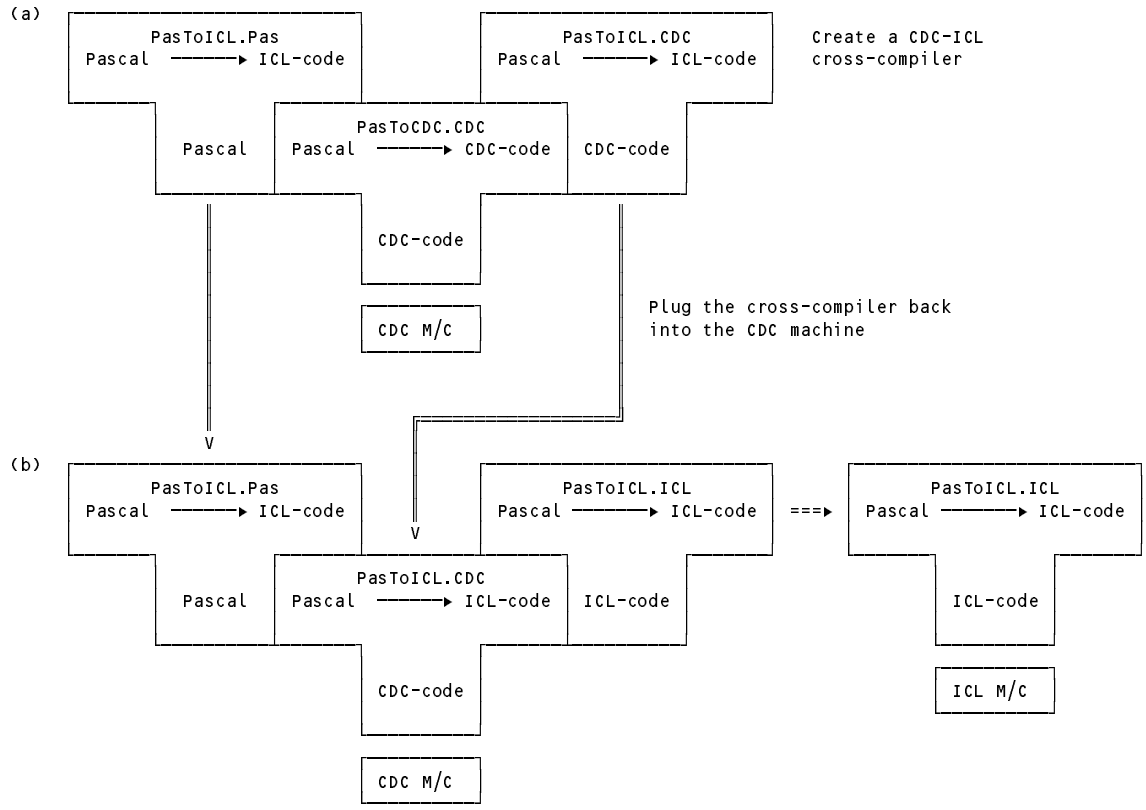


Figure 3.6 The production of the first ICL Pascal compiler by half bootstrap

The final product (*PasToICL.ICL*) was then transported on magnetic tape to the ICL machine, and loaded quite easily (this was long before the Internet!). Having obtained a working system, the ICL team could (and did) continue development of the system in Pascal itself.

This porting operation was an example of what is known as a **half bootstrap** system. The work of transportation is essentially done entirely on the *donor* machine, without the need for any translator on the target machine, but a crucial part of the original compiler (the back end, or code generator) has to be rewritten in the process. Clearly the method is hazardous - any flaws or oversights in writing *PasToICL.Pas* could have spelled disaster. Such problems can be reduced by minimizing changes made to the original compiler. Another technique is to write an emulator for the target machine that runs on the donor machine, so that the final compiler can be tested on the donor machine before being transferred to the target machine.

3.6 Bootstrapping from a portable interpretive compiler

Because of the inherent difficulty of the half bootstrap for porting compilers, a variation on the full bootstrap method described above for assemblers was successfully used in past years in the case of Pascal and other similar high-level languages. Here most of the development takes place on the *target* machine, after a lot of preliminary work has been done on the donor machine to produce an interpretive compiler that is almost portable. It may be helpful to illustrate with the well-known example of the Pascal-P implementation kit mentioned in section 2.8.

Users of this kit typically commenced operations by implementing an interpreter for the P-machine. The bootstrap process was then initiated by developing a compiler (*PasPtoM.PasP*) to translate Pascal-P source programs to the local machine code. This compiler could be written in Pascal-P source, development being guided by the source of the Pascal-P to P-code compiler supplied as part of the kit. This new compiler was then compiled with the interpretive compiler (*PasPtoP.P*) from the kit (Figure 3.7(a)) and the source of the Pascal to M-code compiler was then compiled by this new compiler, interpreted once again by the P-machine, to give the final product, *PasPtoM.M* (Figure 3.7(b)).

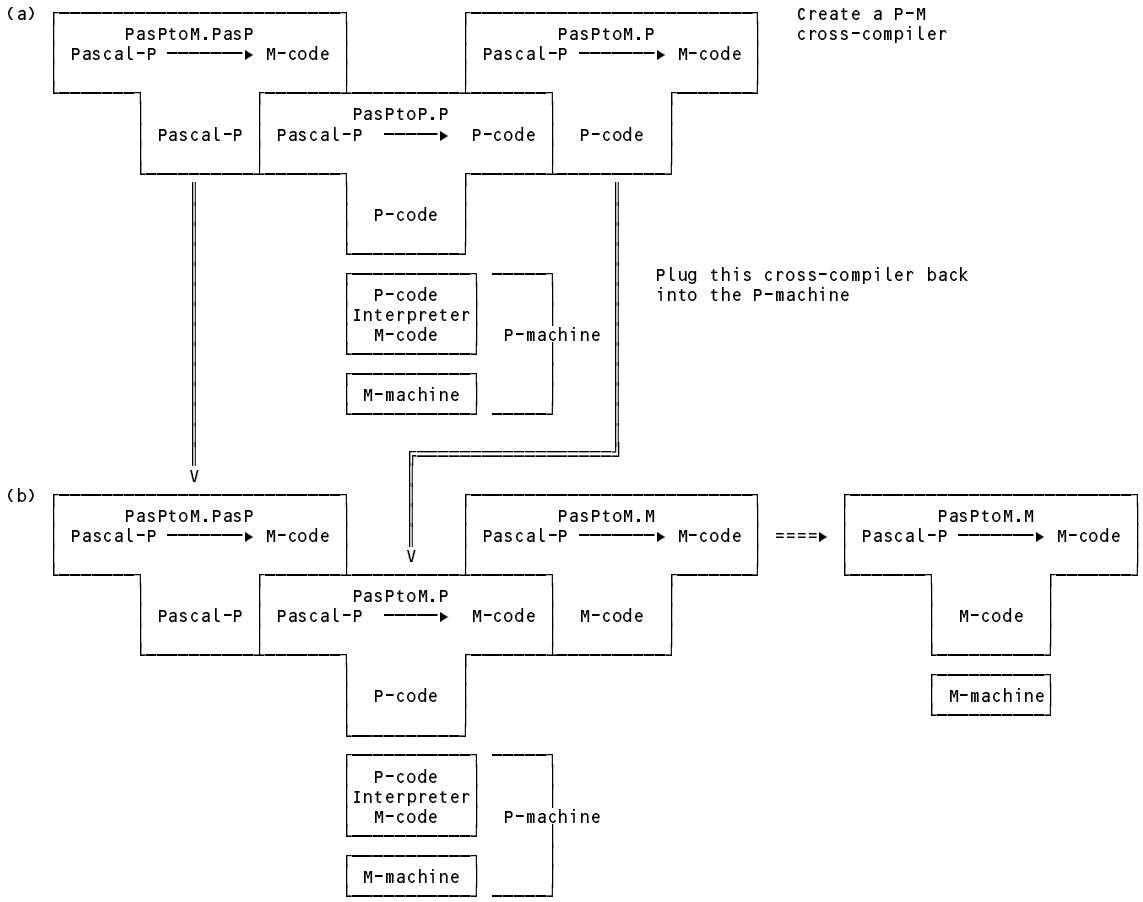


Figure 3.7 Developing a native code compiler from the P-compiler

3.7 A P-code assembler

There is, of course, yet another way in which a portable interpretive compiler kit might be used. One might commence by writing a P-code to M-code assembler, probably a relatively simple task. Once this has been produced one would have the assembler depicted in Figure 3.8. As an exercise, draw the T-diagrams showing how this would be done.

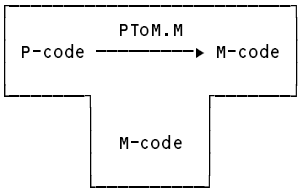


Figure 3.8 A P-code to M-code assembler

The P-code for the P-code compiler (from the kit) would then be assembled by this system to give another cross-compiler (Figure 3.9(a)), and the same P-code/M-code assembler could then be used as a back end to the cross-compiler (Figure 3.9(b)).

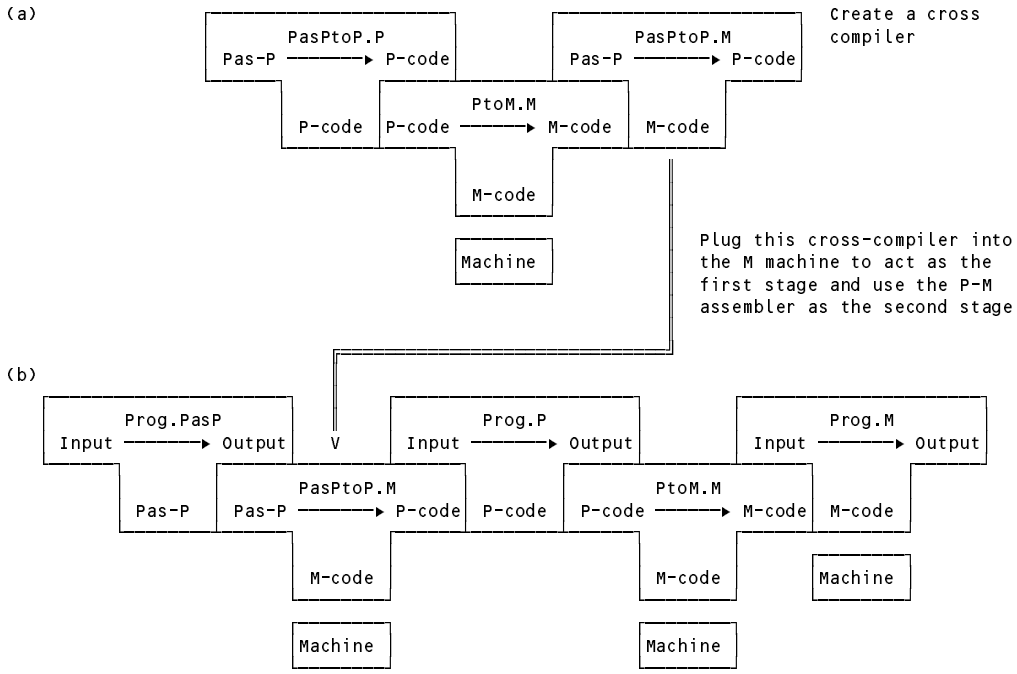


Figure 3.9 Two-pass compilation and assembly using a P-code system

3.8 The use of compiler generating tools

As was mentioned earlier, it may be possible to develop a large part of the compiler source using compiler generator tools - assuming, of course, that these are already available either in executable form or as source that can itself be compiled easily on the developer's machine.

These tools take as input a formal description of the language to be compiled, and construct from this at least the scanner and parser of a corresponding compiler, which can be linked together with code for the other phases such as code generation. In the later chapters of Terry (2005) we illustrate the use of such a tool (Coco/R) for constructing various compilers. Some of these are for a simple language Parva that is used as a case study in the later chapters of the present text. Another is for a compiler hosted in Java that will produce ILASM code - assembler code for the .NET framework - for programs written in a subset of C# (C#Minor). Figure 3.10 illustrates the chain of development in terms of T-diagrams.

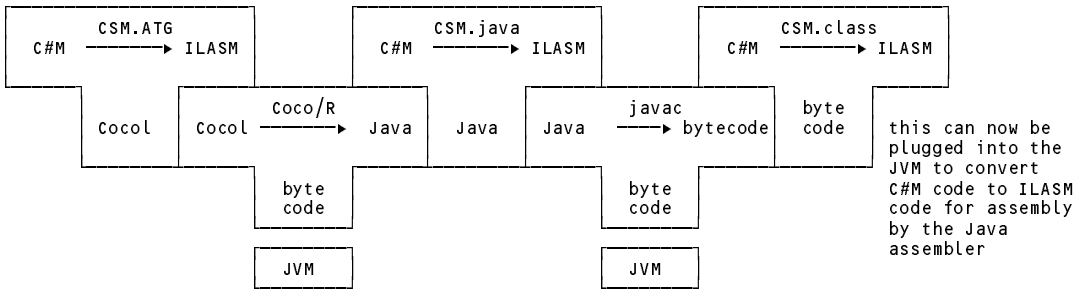


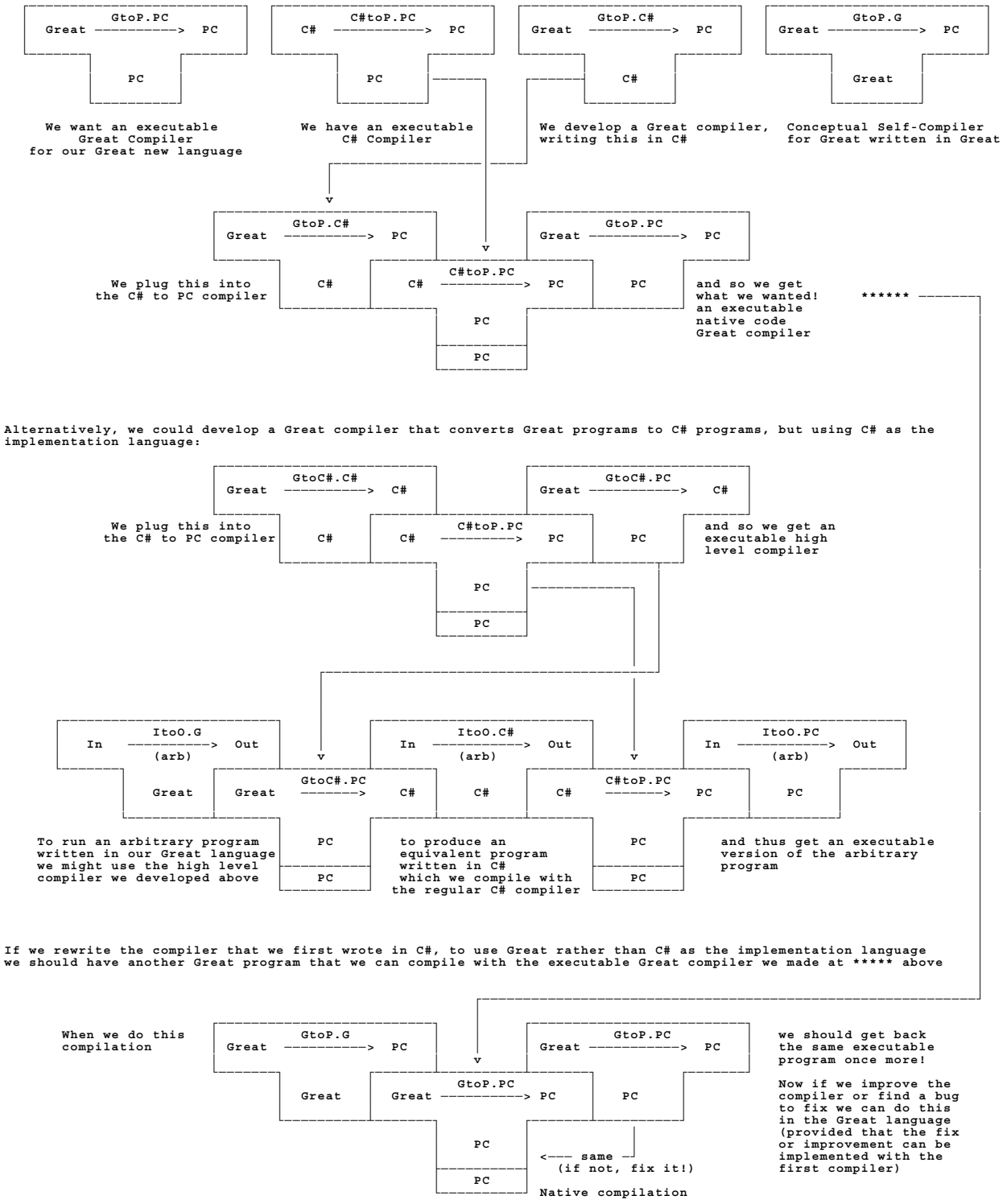
Figure 3.10 Development of a compiler with the aid of a compiler generator

Further reading

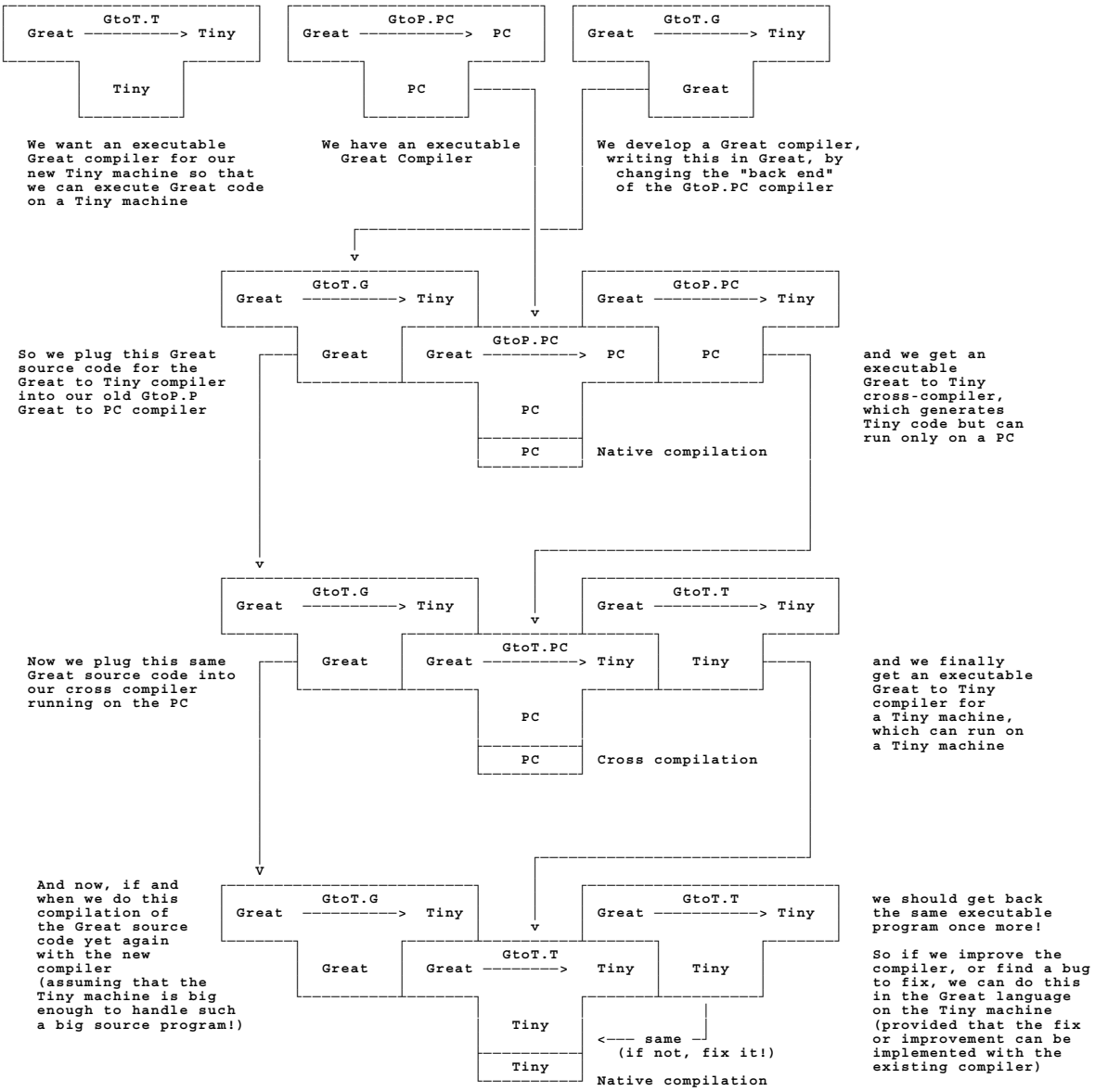
A very clear exposition of bootstrapping is to be found in the books by Watt (1993) and Watt and Brown (2000). The ICL bootstrap is further described by Welsh and Quinn (1972). Other early insights into bootstrapping are to be found in papers by Lecarme and Peyrolle-Thomas (1973), by Nori *et al.* (1981), and Cornelius, Lowman and Robson (1984).

3.9 Diagrams for some case studies

Creating a compiler for a new language on an existing machine



Porting a compiler to a different machine



4 STACK MACHINES

In Chapter 2 we discussed the use of emulation or interpretation as a tool for programming language translation. In this chapter we aim to discuss virtual machine languages and the emulation of virtual machines in more detail. Modern computers are among the most complex machines ever designed by the human mind. However, this is a text on programming language translation and not on electronic engineering, and our restricted discussion will focus mainly on stack-oriented machines and machine languages similar to those used in the JVM and CLR and suited to the simple translators to be discussed in later chapters.

4.1 Simple machine architecture

Many CPU (central processor unit) chips used in modern computers have one or more internal **registers** or **accumulators**, which can be regarded as highly local memory where simple arithmetic and logical operations are performed, and between which local data transfers may take place. These registers may be restricted to the capacity of a single byte (8 bits), or, as is typical of most modern processors, they may come in a variety of small multiples of bytes or machine words.

One fundamental internal register is the **instruction register** (IR), through which moves the bitstrings (bytes) representing the fundamental machine-level instructions that the processor can obey. These instructions tend to be extremely simple - operations such as "clear a register" or "move a byte from one register to another" being the typical order of complexity. Some of these instructions can be completely defined by a single byte value. Others may need two or more bytes for a complete definition. Of these multi-byte instructions, the first usually denotes an operation and the rest relate either to a value to be operated upon or to the address of a location in memory at which can be found the value to be operated upon.

The simplest processors have only a few **data registers** (R_1 through R_n) and are very limited in what they can actually do with their contents. So processors invariably make provision for interfacing to the memory of the computer, and allow transfers to take place along so-called **bus** lines between the internal registers and the far greater number of external memory locations. When information is to be transferred to or from memory, the CPU places the appropriate address information on the address bus, and then transmits or receives the data itself on the data bus. This is illustrated in Figure 4.1 (after Wakerly (1981)).

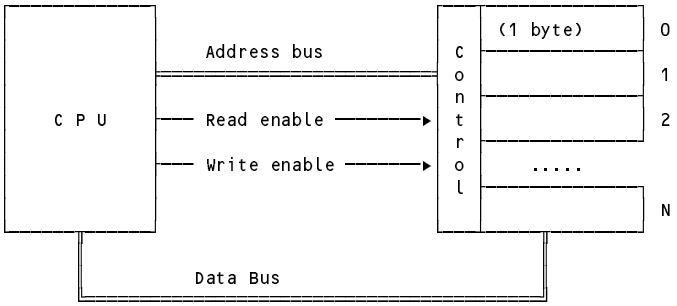


Figure 4.1 The CPU is linked to memory by address and data buses

The memory may simplistically be viewed as a one-dimensional array of byte values, analogous to what might be described in C# or Java terms by

```
byte[] mem = new byte[memSize];
```

Since the memory is used to store not only "data" but also "instructions", another important internal register in a processor, the so-called **program counter** or **instruction pointer** (denoted by PC or IP), is used to keep track of the address in memory of the next instruction to be fed to the processor's IR.

Perhaps it will be helpful to think of the processor itself in high-level terms. In the rather unrealistic situation where all the registers have single byte capacity we might think in terms of C# or Java declarations like

```
class Processor {
  byte IR;          // instruction register
  byte R1, R2, R3;  // data registers
  int PC;           // program counter
} // class Processor

Processor CPU = new Processor();
```

The operation of the machine is repeatedly to *fetch* a byte at a time from memory (along the data bus), place it in the IR, and then *execute* the operation which this byte represents. Multi-byte instructions may require the fetching of further bytes before the instruction itself can be decoded fully by the CPU. After the instruction denoted by the contents of IR has been executed, the value of PC will have been changed to point to the next instruction to be fetched. This **fetch-execute cycle** may be described by the following algorithm.

```
BEGIN
  CPU.PC := initialValue;    (* address of first instruction *)
  LOOP
    CPU.IR := mem[CPU.PC];   (* fetch *)
    Increment(CPU.PC);       (* bump PC in anticipation *)
    Execute(CPU.IR, CPU.PC); (* affects registers, memory, PC *)
  END
END
```

Normally the value of PC alters by small steps (since instructions are usually stored in memory in sequence). Execution of branch instructions may, however, have a rather more dramatic effect.

The algorithm just given reflects a view of the operation of a computer that suffices for a great many applications. However, modern computers (just like modern people) live in a world where unpredictable things can happen to which they may have to turn their attention, temporarily diverting themselves from the task originally planned. For example, a CPU may also be connected to various other hardware devices - such as printers, modems, ethernet network cards and so on. These might interact with it in an *asynchronous* way, that is to say, at fairly unpredictable times sending the CPU special signals, seeking attention, perhaps by setting an interrupt request bit, (IRQ). Should this signal be detected, the processor must react to it. This may be modelled by extending the fundamental fetch-execute cycle as shown below. Real processors, which perform each pass through the cycle very quickly indeed, are likely to handle interruptions relatively rarely.

```
BEGIN
  CPU.PC := initialValue;    (* address of first instruction *)
  LOOP
    CPU.IR := mem[CPU.PC];   (* fetch *)
    Increment(CPU.PC);       (* bump CPU.PC in anticipation *)
    Execute(CPU.IR, CPU.PC); (* affects registers, memory, PC *)
    IF CPU.IRQ               (* has process been interrupted? *)
      THEN HandleInterrupt   (* handle asynchronous interrupts *)
  END
END
```

We shall not discuss interrupt handling further.

4.2 ASSEMBLER languages

A program for such a machine as we have discussed consists, in the last resort, of a long string of byte values. Were these to be written on paper (as binary, decimal, or hexadecimal values), they would appear pretty meaningless to the human reader. We might, for example, find a section of program reading

```
25  45  21  34  34  30  45
```

Although it may not be obvious, this might be equivalent to a high-level statement like

```
Price = 2 * Price + MarkUp;
```

Machine-level programming is usually performed by associating *mnemonics* with the recognizable operations - like `HLT` for "halt" or `ADD` for "add to register". The above code is far more comprehensible written as

```
LDA  45   ; load accumulator with value stored at address 45
SHL       ; shift accumulator one bit left (multiply by 2)
ADI  34   ; add 34 to the accumulator
STA  45   ; store the value in the accumulator at address 45
```

The basic purpose of an assembler is to translate these mnemonics into binary or hexadecimal machine code. Some assemblers do little more than this, but most modern assemblers offer a variety of additional features and the boundary between assemblers and compilers has become somewhat blurred.

Programs written in ASSEMBLER invariably make use of other named entities besides the mnemonics used for the opcodes. A typical example of a program written in this way for a hypothetical machine is given below, along with its equivalent object code. We have, as is conventional, used hexadecimal notation for the object code; numeric values in the source have been specified in decimal.

```
00                BEG             ; Count the bits in a number
00   02           CLA             ; CPU.A := 0
01   21 14        STA     BITS    ; BITS := 0
03   13           INI             ; Read(CPU.A)
04         LOOP                   ; REPEAT
04   07           SHR             ;  CPU.A := CPU.A DIV 2
05   3E 0E        BCC     EVEN    ;  IF CPU.A MOD 2 # 0 THEN
07   10           PSH             ;    save CPU.A on stack
08   1C 14        LDA     BITS
0A   04           INC
0B   21 14        STA     BITS    ;    BITS := BITS + 1
0D   11           POP             ;    restore CPU.A
0E   3A 04  EVEN  BNZ     LOOP    ; UNTIL CPU.A = 0
10   1C 14        LDA     BITS    ;
12   17           OTI             ; Write(BITS)
13   01           HLT             ; terminate execution
14         BITS   DS      1       ; BYTE BITS
15                END
```

ASSEMBLER statements fall into two main classes.

Firstly, there are the **executable instructions** that correspond directly to executable code. These can be recognized immediately by the presence of a distinctive mnemonic for an **opcode**. These executable instructions divide further into two classes - there are those that require an **argument** as part of the instruction (as in `LDA BITS`) and occupy two or more bytes of object code, and there are those that stand alone (like `INI` and `HLT`). When it is necessary to refer to such statements elsewhere, they may be labelled with an introductory distinctive **label** identifier of the programmer's choice (as in `EVEN BNZ LOOP`), and may include a **comment**, typically extending from an introductory semicolon to the end of a line.

The argument for those instructions that require them is denoted most simply by either a numeric literal or by an identifier of the programmer's choice. Such identifiers usually correspond to the ones that are used to label statements - when an identifier is used to label a statement itself we speak of a **defining occurrence** of a label; when an identifier appears as an address or operand we speak of an **applied occurrence** of a label.

The second class of statement includes the **directives**. In source form these appear to be deceptively similar to executable instructions - they are often introduced by a label, terminated with a comment, and have what may appear to be mnemonic and argument components (as in `BITS DS 1`). However, directives have a rather different role to play. They do not generally correspond to operations that will form part of the code that is to be executed at *run-time*, but rather denote actions that direct the action of the assembler at *compile-time* - for example, indicating where in memory a block of code or data is to be located when the object code is later loaded, or indicating that a block of memory is to be preset with literal values, or that a name is to be given to a literal to enhance readability.

When we use code fragments such as these for illustration we shall make frequent use of commentary showing an equivalent fragment written in a high-level language.

4.3 Addressing modes

Within the broad spectrum of ASSEMBLER instructions we find a wide variety of semantic meaning.

An example of a simple operation expressed in a high-level language might be

```
AmountDue = Price + Tax;
```

Some machines and assembler languages provide for such operations in terms of so-called **three-address code** in which the opcode is followed by two *operands* and a *destination*. In general this takes the form

operation destination, operand$_1$*, operand*$_2$

as exemplified by

```
ADD     AmountDue, Price, Tax
```

We may also express this in a general sense as a function call

destination = *operation*(*operand*$_1$*, operand*$_2$)

which helps to stress the important idea that the *operands* really denote *values*, while the *destination* denotes a processor register, or an *address* in memory where the result is to be stored.

In many cases this generality is restricted (that is, the machine suffers from non-orthogonality in design). Typically the value of one *operand* is required to be the value originally stored at the *destination*. This corresponds to high-level statements like

```
Price = Price * InflationFactor;
```

and is mirrored at the low-level by so-called **two-address code** of the general form

operation destination, operand

as exemplified by

```
MUL     Price, InflationFactor
```

In passing, we can point out an obvious connection between some of the assignment operations in C and its derivatives and two-address code. In these languages the above assignment would probably have been written

```
Price *= InflationFactor;
```

which is surely a hint to a compiler to generate code of this form. (Perhaps this example may help you understand why C is regarded by some as the world's finest assembly language!)

In some real machines even general two-address code is not found at the machine level. One of *destination* and *operand* might be restricted to denoting a machine register. The other one might denote a machine register, or a constant, or a machine address. This is often called **one-and-a-half-address code**, and is exemplified by

```
MOV     R1, Value    ;  CPU.R1 := Value
ADD     Answer, R1   ;  Answer := Answer + CPU.R1
MOV     Result, R2   ;  Result := CPU.R2
```

In so-called *accumulator machines* we may be restricted still further to **one-address code**, where the destination is *always* a machine register, save for those operations that copy (store) the contents of a machine register into memory. In some assembler languages such instructions may still appear to be of the two-address form, as above. Alternatively they may use opcodes that have the register implicit in the mnemonic, as exemplified by

```
LDA     Value        ;  CPU.A := Value
ADA     Answer       ;  CPU.A := CPU.A + Answer
STB     Result       ;  Result := CPU.B
```

For many machine instructions multi-byte instructions are required. The first byte typically specifies the operation itself (and possibly the register or registers that are involved), while the remaining bytes specify the other values (or the memory addresses of the other values) involved. In such instructions there are several ways in which the ancillary bytes might be used. This variety gives rise to what are known as different **addressing**

modes for the processor, whose purpose it is to provide an **effective address** to be used in an instruction. Exactly which modes are available varies tremendously from processor to processor and we can mention only a few representative examples here.

In **inherent addressing** the operand is implicit in the opcode itself, and often the instruction is contained in a single byte. For example, to clear a machine register named `A` we might have a single byte instruction

```
CLA                 ;  CPU.A := 0
```

Some instructions of this form actually involve more than one operand. This is frequently the case for the very compact **zero-address code** found in stack machines, where the operands are implicitly found on a stack, the top of which is pointed to by an internal CPU register. Instructions of this form are fundamental to the operation of the JVM and CLR and we shall discuss them more fully in later sections.

In **immediate addressing** the ancillary bytes for an instruction typically give the *actual value* that is to be combined with a value in a register. An example, from the instruction set for Intel 80x86 processors, might be

```
MOV  AX, 34          ; CPU.AX := 34
```

In this addressing mode the use of the word "address" is almost misleading, as the value of the ancillary bytes may often have nothing to do with a memory address at all.

However, in **direct** or **absolute addressing** the ancillary bytes typically specify the *memory address* of the value that is to be retrieved or combined with the value in a register, or specify where a register value is to be stored. An example is

```
MOV  AX, [34]         ; CPU.AX := mem[34]
```

Beginners frequently confuse immediate and direct addressing, a situation not improved by the fact that there is no consistency in notation between different assembler languages - and there may even be a variety of ways of expressing a particular addressing mode.

Further reading

Most books on assembler level programming have far deeper discussions of the subject of addressing modes than we have presented. Readable accounts are to be found in the books by Wakerly (1981) and MacCabe (1993). A deeper discussion of machine architectures is to be found in the book by Hennessy and Patterson (2002).

4.4 The PVM - a simple stack machine

In later sections of this text we shall be looking at developing compilers that generate object code for a variety of "stack machines", ones that have no general data registers of the sort discussed previously but which function primarily by manipulating a stack pointer and associated stack. Such machines turn out to be ideally suited to the evaluation of complicated arithmetic or Boolean expressions, as well as to the implementation of high-level languages which support function calls, the concepts of global and local variables, recursion and the ability to create objects of various complexity as the execution proceeds. In this section we wish to illustrate the principles of such machines, using one of our own design. The machine is called the Parva Virtual Machine (PVM). ("Parva" is the feminine form of the Latin adjective meaning "small".)

4.4.1 Machine architecture

We assume that our stack machine (in common with most machines) stores code and data in a memory that can be modelled as a linear array. For simplicity the elements of memory will be treated as "words". A word might store a single integer - typically using a 16-bit or 32-bit two's-complement representation. It might also store an instruction, or a Boolean value, or a character, or even the address of a data item or machine instruction (that is, a word has the capacity to be somewhat abused, and regarded as a value of any convenient type). Diagrammatically we might represent the PVM as in Figure 4.2:

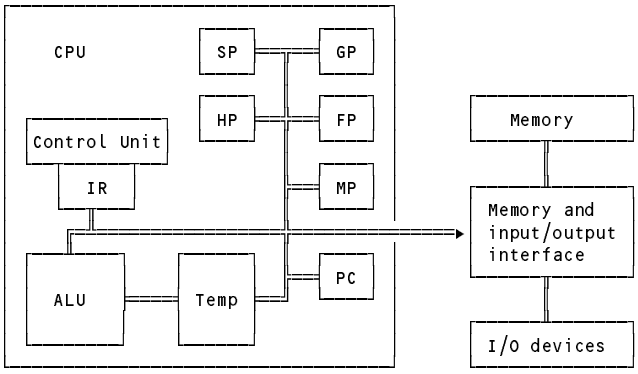


Figure 4.2 A simple stack-oriented CPU and computer

The symbols in this diagram refer to the following components of the PVM:

`ALU` is the *arithmetic logic unit* where arithmetic and logical operations are actually performed;

`Temp` is a set of registers for holding intermediate results needed during arithmetic or logical operations. - these registers cannot be accessed explicitly;

`SP` is the *stack pointer*, a register that points to the area in memory utilized as the main stack;

`HP` is the *heap pointer*, a register that points to an area in memory in which objects and arrays can be created;

`GP` is the *global pointer*, a register that points to the base of an area of memory within the stack, which is used to store the global variables accessible to all routines;

`FP` is the *frame pointer*, a register that points to the base of an area of memory within the stack, known as a *stack frame* or *activation record*, which is used to store the local variables and arguments for the currently active function or method;

`MP` is the *mark stack pointer*, a register used in handling function and method calls, whose use will become apparent only in later chapters;

`IR` is the *instruction register* in which is held the instruction currently being executed;

`PC` is the *program counter* which contains the address in memory of the instruction that is next to be executed.

A programmer's model of the PVM is suggested by C# declarations like

```
class Processor {
  public int sp;                    // Stack pointer
  public int hp;                    // Heap pointer
  public int gp;                    // Global frame pointer
  public int fp;                    // Local frame pointer
  public int mp;                    // Mark stack pointer
  public int ir;                    // Instruction register
  public int pc;                    // Program counter
} // class Processor

const int memSize = 512;            // Limit on memory
int[] mem = new int[memSize + 1];   // Plug in some memory
Processor cpu = new Processor();    // Plug in a processor
```

For simplicity, we shall assume that the code is stored in the low end of memory and that the top part of memory is used for storing data. We shall treat the topmost section of memory as a *literal pool* in which are stored constants, such as literal character strings. The rest of the memory is then used to set up various stacks for working storage. A typical memory layout might be as shown in Figure 4.3, where the markers `HeapBase` and `StackBase` will be useful for providing memory protection in an emulated system.

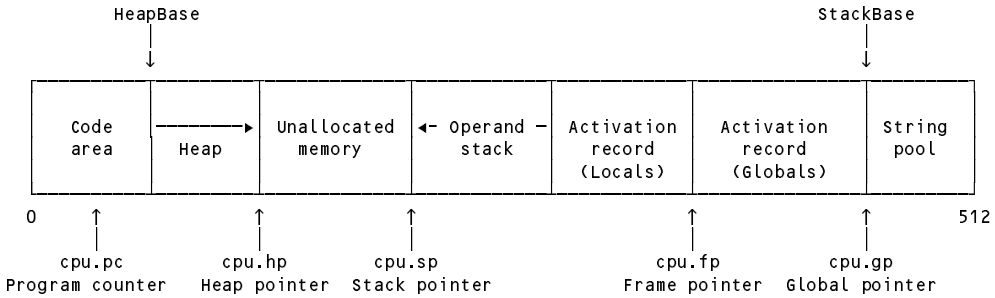


Figure 4.3 Usage of memory in a simple stack-oriented computer

We assume that the program loader will load the code at the bottom of memory, leaving the marker denoted by `HeapBase` pointing to the next word in memory above this code. It will also load the string literals into the literal pool, leaving the marker denoted by `StackBase` pointing to the low end of this pool. It will go on to initialize the stack pointer `SP`, the global frame pointer `GP` and the local frame pointer `FP` to the value of `StackBase`, and initialize the heap pointer `HP` to the value of `HeapBase`.

As each function in the program is activated it will be allocated an **activation record** from the (high end) stack area of the memory. In such activation records - sometimes called **stack frames** - storage is reserved for the local variables of each function, which can be recovered when the function has completed its work. A special activation record might be used to reserve storage for the global variables of the program. Allocation of such storage is achieved by machine operations that manipulate the various processor pointers. Memory may also be allocated from the (low end) heap area of memory, typically on demand when a method creates an array or some other object whose size may not have been known when the program was compiled.

Function activation is the subject of Chapter 14. For the moment it will suffice to discuss the simplest of programs - those having only a single (`main`) function.

The first instruction in such a program might have the responsibility of reserving further space in memory for its variables, simply by decrementing the stack pointer `SP` by the number of words needed for these variables. The stack grows downwards in memory, from high addresses towards low ones. Variables associated with the function do not have absolute memory addresses determined at compile time, but can be addressed by subtracting an offset from the frame pointer `FP`.

4.4.2 Instruction set

A minimal set of operations for the PVM is described informally below. (As we proceed we shall find it convenient to add more opcodes to this set.) We shall use the mnemonics introduced here to code programs for the machine in what appears to be a simple assembler language, albeit with addresses stipulated in absolute form.

Several of these operations belong to the category of zero-address instructions mentioned in section 4.3. Even though operands are clearly needed for operations such as addition and multiplication, their addresses or values are not specified by part of the instruction but are implicitly derived from the value of the stack pointer `SP`. The two operands are assumed to reside on the top and immediately below the top of the stack - in this context sometimes called the **operand stack** or **evaluation stack**. In our descriptions their values are denoted by `TOS` (for "top of stack") and `SOS` (for "second on stack").

`LDC N` Push the constant integer value `N` onto the stack to form new `TOS`

`LDA N` Push the value `FP - 1 - N` onto the stack to form new `TOS`. (This value is conceptually a *reference*, the *address* of the `N`th variable, stored at an offset `N` within the stack frame pointed to by the frame pointer `FP`.)

`LDV` Pop `TOS` and push the value of `mem[TOS]` to form new `TOS` (an operation called *dereferencing*)

`LDXA` Pop `TOS` and `SOS`, add `SOS` to `TOS`, push sum to form new `TOS`. (This value is conceptually a *reference*, the *address* of an array element within the heap area.)

`STO` Pop `TOS` and `SOS`, store `TOS` in `mem[SOS]`

`PRNI` Pop `TOS` and write it to the output as an integer value

`PRNB` Pop `TOS` and write it to the output as a Boolean value

PRNS N	Write the nul-terminated string that is stacked in the literal pool from mem[N]
PRNL	Write a newline (carriage-return-linefeed) sequence
INPI	Read an integer value, pop TOS, store the value that was read in mem[TOS]
INPB	Read a Boolean value, pop TOS, store the value that was read in mem[TOS]
ADD	Pop TOS and SOS, add SOS to TOS, push sum to form new TOS
SUB	Pop TOS and SOS, subtract TOS from SOS, push difference to form new TOS
MUL	Pop TOS and SOS, multiply SOS by TOS, push product to form new TOS
DIV	Pop TOS and SOS, divide SOS by TOS, push quotient to form new TOS
REM	Pop TOS and SOS, divide SOS by TOS, push remainder to form new TOS
CEQ	Pop TOS and SOS, push 1 to form new TOS if SOS = TOS, 0 otherwise
CNE	Pop TOS and SOS, push 1 to form new TOS if SOS ≠ TOS, 0 otherwise
CGT	Pop TOS and SOS, push 1 to form new TOS if SOS > TOS, 0 otherwise
CLT	Pop TOS and SOS, push 1 to form new TOS if SOS < TOS, 0 otherwise
CLE	Pop TOS and SOS, push 1 to form new TOS if SOS ≤ TOS, 0 otherwise
CGE	Pop TOS and SOS, push 1 to form new TOS if SOS ≥ TOS, 0 otherwise
NEG	Integer negation of TOS
NOT	Boolean negation of TOS
AND	Pop TOS and SOS, bitwise and SOS with TOS, push result to form new TOS
OR	Pop TOS and SOS, bitwise or SOS with TOS, push result to form new TOS
DSP N	Decrement value of stack pointer SP by N
ANEW	Pop TOS to yield size; allocate an array of length size on the heap, push heap pointer HP to form new TOS, and increment heap pointer HP by Size
HALT	Terminate execution
BRN N	Unconditional branch to instruction N
BZE N	Pop TOS, and branch to instruction N if TOS is zero
NOP	No operation

The instructions in the first group allow for the access of data in memory by means of manipulating addresses and the stack. Those in the second group, affording I/O facilities, are not typical of real machines but allow us to focus on the principles of emulation without getting lost in the trivia and overheads of handling real I/O systems.

The instructions in the third group are concerned with arithmetic and logical operations. Those in the fourth group are concerned with allocation of memory to variables, while those in the last group allow for control of flow of the program itself.

4.5 Programming the PVM

As our stack machine and its instruction set may be rather different from anything the reader has seen before, some samples of program code may help to clarify various points.

Example 4.5.1 - Manipulating local variables

To illustrate how the memory is allocated and simple transfers between variables are achieved, consider a simple section of program that corresponds to high-level code of the form

```
    X := -1; Y := X;

 0  DSP  2   ; reserve 2 variables - X is var 0, Y is var 1
 2  LDA  0   ; push address of X
 4  LDC  -1  ; push constant -1
 6  STO      ;        X := -1
 7  LDA  1   ; push address of Y
 9  LDA  0   ; push address of X
11  LDV      ; dereference - value of X now on top of stack
12  STO      ;        Y := X
```

Before we can perform a STO instruction it is necessary that we set up the stack to contain an address and a value to be stored at that address, in that order. Figure 4.4 shows in detail what the local variable area and stack might look like after the completion of each of the instructions in the above sequence. For illustration we have assumed

that the stack pointer SP has an initial value of 512. The effect of the DSP instruction is to create the activation record for the two variables. When the sequence is completed the stack below this will again be empty.

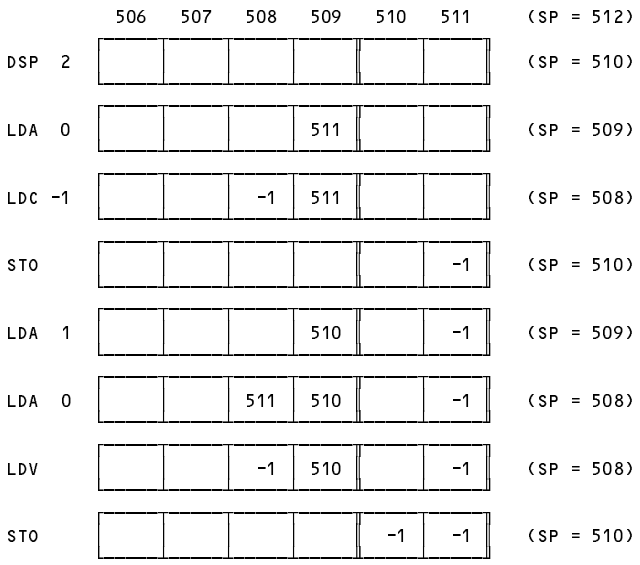


Figure 4.4 Behaviour of the stack during a simple sequence of assignments

Example 4.5.2 - Simple I/O instructions

Our second example shows how one might perform simple input/output (I/O).

```
 0   DSP  1   ; X is variable 0
 2   LDA  0   ; push address of X
 4   INPI     ; read and store at address popped from stack
 5   PRNS "X = " ; write string "X = "
 7   LDA  0   ; push address of X
 9   LDV      ; dereference - value of X now on top of stack
10   PRNI     ; pop value of X and print it
11   HALT
```

This program would be stored in memory as shown in Figure 4.5.

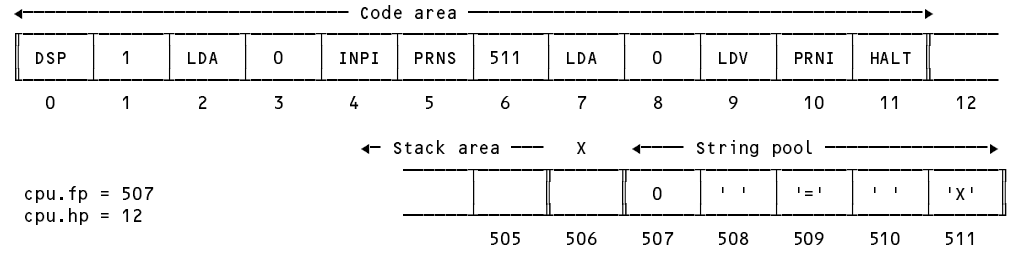


Figure 4.5 Memory use in a simple program, showing data, strings and code

Immediately after loading this program (but before executing the DSP instruction) the program counter PC would have the value 0, the frame pointer FP and stack pointer SP would each have the value 507, and the heap pointer HP would have the value 12.

Example 4.5.3 - Simple arithmetic

A binary operation is accomplished by popping two operands from the stack into (inaccessible) internal registers in the CPU, performing the operation, and then pushing the result back onto the stack. Such operations can be very economically encoded in terms of the storage taken up by the program code itself - the high density of stack-oriented machine code is a strong point in its favour so far as developing interpretive translators is concerned.

The third example shows how we might read two values X and Y and compute the value of X + 4 / Y.

```
 0   DSP  2   ; X is variable 0, Y is variable 1
 2   LDA  0   ; push address of X
 4   INPI     ; read and store at address popped from stack
 5   LDA  1   ; push address of Y
 7   INPI     ; read and store at address popped from stack
 8   LDA  0   ; push address of X
10   LDV      ; dereference - value of X now on top of stack
11   LDC  4   ; push constant 4 onto stack
13   LDA  1   ; push address of Y
15   LDV      ; dereference - value of Y now on top of stack
16   DIV      ; pop Y and 4 - push back 4/Y
17   ADD      ; pop 4/Y and X - push back X+4/Y
18   PRNI     ; pop value of X+4/Y and print it
19   HALT
```

Example 4.5.4 - Looping and decision making

Our examples so far have scarcely represented the epitome of the programmer's art! A more realistic program follows, as a translation of the simple algorithm below. The program shows how the comparison and branching instructions may be used to control loops and how high-level code may adequately be used as commentary.

```
BEGIN
  Total := 0;
  REPEAT  READ(X);  Total := X + Total   UNTIL X = 0;
  WRITE("Total is ", Total);
END
```

```
 0   DSP  2   ; X is variable 0, Total is variable 1
 2   LDA  1
 4   LDC  0
 6   STO      ; Total := 0
 7   LDA  0   ; REPEAT
 9   INPI     ;   Read(X)
10   LDA  1
12   LDA  0
14   LDV
15   LDA  1
17   LDV
18   ADD
19   STO      ;   Total := X + Total
20   LDA  0
22   LDV
23   LDC  0
25   CEQ      ;   (* check equality of X and 0 *)
26   BZE  7   ; UNTIL X = 0
28   PRNS "Total is"
30   LDA  1
32   LDV
33   PRNI     ; Write(Total)
34   HALT
```

The instructions that effect comparisons achieve their objective by popping two integer values from the stack and pushing back a Boolean value. Internally the machine can simply represent *false* and *true* by the integer values 0 and 1 respectively. In this particular example - where the loop is controlled by a test for zero - we could use the instruction set in another way, by substituting for the last section of the above code:

```
20   LDA  0
22   LDV
23   BZE  27  ; exit when X = 0
25   BRN  7   ; otherwise repeat the loop
27   PRNS "Total is"
29   LDA  1
31   LDV
32   PRNI     ; Write(Total)
33   HALT
```

Example 4.5.5 - Simple array handling

The examples given previously have shown how sometimes it is necessary to store addresses on the operand stack, as opposed to simple integers or Boolean values, although the variables in the programs have all been of integer type. However, in some situations it is useful to have variables that themselves store addresses. These are sometimes called **pointer variables**, **reference variables** or, simply, **references**.

One approach to manipulating arrays depends on this approach. Storage for an extended array might be allocated "statically", or it might be allocated "dynamically" as and when the required size of the array becomes known. Storage of this latter sort is often taken from the **heap** - the unused portion of memory above the code for the program itself. The PVM has an instruction (ANEW) for doing this. When an array is to be allocated, the size of the array is computed and pushed onto the stack, and the heap manager is invoked. The size is popped from the stack, and in its place is stored the current value of the heap pointer HP, which always "points" to the first currently unused word in memory in the heap area. The heap pointer is then "bumped" by the amount of storage requested so that future calls to the heap manager can allocate space beyond the area allocated at this time. The memory address left on the top of the stack can then be stored in a variable, effectively recording the address of the zeroth element of the array.

We may illustrate the process by considering PVM code equivalent to C# code of the form

```
int[] list = new int[3];
bool[] sieve = new bool[5];
```

```
 0   DSP  2   ; ref to list is var 0, ref to sieve is var 1
 2   LDA  0   ; push address of list on stack
 4   LDC  3   ; push number of elements needed for list array
 6   ANEW     ; allocate storage for list
 7   STO      ; store address of list[0] as value of var list
 8   LDA  1   ; push address of sieve on stack
10   LDC  5   ; push number of elements needed for sieve array
12   ANEW     ; allocate storage for sieve
13   STO      ; store address of sieve[0] as value of var sieve
```

Once this code has executed, the state of memory might be as shown in Figure 4.6.

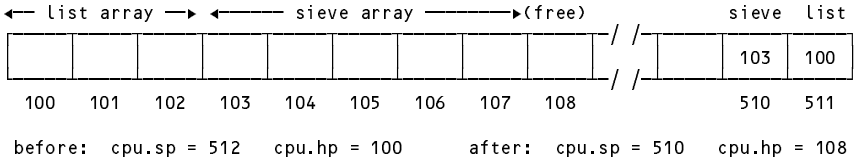


Figure 4.6 Memory allocation after dynamic allocation of two arrays

To access an element of the array requires that the machine perform "address arithmetic". Firstly, the value of the reference variable is pushed onto the stack - recall that this value is really that of an address in the heap area, previously allocated by the heap manager. Secondly, the value of the subscript is computed and left on the stack. Thirdly, these two values are popped off the stack and added together. The sum - which represents the address in the heap area where the element is to be found - is pushed back onto the stack, whence it may be dereferenced (to find the value of the array element) or used as the target of a STO instruction (to store a value at that address). This can be clarified by the code below which shows such manipulation following on from the allocation shown earlier.

```
14   LDA  0   ; push address of list reference onto stack
16   LDV      ; dereference - address of list[0] on top of stack
17   LDC  2   ; push value of subscript (2)
19   LDXA     ; calculate address of list[2] as new value of TOS
20   LDC  12  ; push constant 12 onto stack
22   STO      ;    list[2] = 12
23   LDA  1   ; push address of sieve reference onto stack
25   LDV      ; dereference - address of sieve[0] on top of stack
26   LDC  4   ; push value of subscript (4)
28   LDXA     ; calculate address of sieve[4] as new value of TOS
29   LDV      ; dereference - value of sieve[4] on top of stack
30   PRNB     ;    print(sieve[4])
```

4.6 An emulator for the PVM

Although a processor for our machine almost certainly does not exist "in silicon", its action may easily be simulated "in software". Essentially we need only to write an emulator that models the fetch-execute cycle of the processor, and we can do this in any suitable language for which we already have a compiler on a real machine. Languages like C# and Java are highly suited to this purpose. Not only do they have "bit-twiddling" capabilities for performing operations like "bitwise and", they have the advantage that one can implement the various phases

of translators and emulators as coherent, clearly separated modules or classes.

In modelling the PVM in C# it will be convenient to have a well-defined specification of a stack machine class. The main responsibility of the class will be to define an `Emulator` routine for interpreting the code stored in the memory of the machine. For expediency we have chosen to extend the specification to expose the (suitably enumerated) values of the opcodes, and the memory itself, and to provide various other useful public methods that will later help us develop an assembler or compiler targeting the machine. (In this, and in other such specifications, private members are not shown.)

```
class PVM {

// Machine opcodes

  public const int
    nop  =  1,  dsp  =  2,                        // others like this

// Memory

  public const int memSize = 5120;                // Limit on memory
  public static int[] mem = new int[memSize + 1]; // memory

// The interpreter and utility methods

  public static void Emulator(int initPC, int codeLen, int initSP,
                              InFile data, OutFile results,
                              bool tracing)
  // Emulates execution of the codeLen instructions stored in
  // mem[0..codeLen-1], with program counter initialized to initPC
  // stack pointer initialized to initSP. data and results are
  // used for I/O.  Tracing at the code level may be requested.

  public static void Interpret(int codeLen, int initSP)
  // Interactively opens data and results files.  Then
  // interprets the codeLen instructions stored in mem, with
  // stack pointer initialized to initSP.

  public static void ListCode(string fileName, int codeLen)
  // Lists the codeLen instructions stored in mem on named file

  public static int OpCode(string str)
  // Maps str to opcode, or to PVM.nul if no match can be found

  public static void Init()
  // Initializes stack machine

} // class PVM
```

The fetch-execute cycle of the processor is easily simulated by the repetitive execution of a large `switch` or `CASE` statement, and the whole system follows the lines of the algorithm given below.

```
BEGIN
  InitializeRegisters;
  InitializeProgramCounter;
  PS := running;
  REPEAT
    CPU.IR := mem[CPU.PC];     (* fetch *)
    Increment(CPU.PC)          (* bump PC in anticipation *)
    CASE CPU.IR OF             (* execute *)
        . . . .
    END
  UNTIL PS ≠ running;
END
```

The full implementation is to be found in the *Resource Kit* - the important parts are listed here for the reader to study.

```
  static int Next() {
  // Fetches next word of program and bumps program counter
    return mem[cpu.pc++];
  } // PVM.Next

  static void Push(int value) {
  // Bumps stack pointer and pushes value onto stack
    mem[--cpu.sp] = value;
  } // PVM.Push
```

```
static int Pop() {
// Pops and returns top value on stack and bumps stack pointer
  return mem[cpu.sp++];
} // PVM.Pop

public static void Emulator(int initPC, int codeLen, int initSP,
                            InFile data, OutFile results,
                            bool tracing)
// Emulates execution of the codeLen instructions stored in
// mem[0..codeLen-1], with program counter initialized to initPC
// stack pointer initialized to initSP. data and results are
// used for I/O.  Tracing at the code level may be requested.

  int loop;                   // internal loops
  int tos;                    // value popped from stack
  stackBase = initSP;
  heapBase = codeLen;         // initialize boundaries
  cpu.hp = heapBase;          // initialize registers
  cpu.sp = stackBase;
  cpu.gp = stackBase;
  cpu.mp = stackBase;
  cpu.fp = stackBase;
  cpu.pc = initPC;            // initialize program counter
  bool running = true;        // prepare to execute
  do {
    cpu.ir = Next();          // fetch
    switch (cpu.ir) {         // execute
      case PVM.nop:           // no operation
        break;
      case PVM.dsp:           // decrement stack pointer
        cpu.sp -= Next();     // (allocate space for variables)
        break;
      case PVM.ldc:           // push constant value
        Push(Next());
        break;
      case PVM.lda:           // push local address
        Push(cpu.fp - 1 - Next());
        break;
      case PVM.ldv:           // dereference
        Push(mem[Pop()]);
        break;
      case PVM.sto:           // store
        tos = Pop(); mem[Pop()] = tos;
        break;
      case PVM.ldxa:          // heap array indexing
        tos = Pop(); Push(Pop() + tos);
        break;
      case PVM.inpi:          // integer input
        mem[Pop()] = data.ReadInt();
        break;
      case PVM.prni:          // integer output
        results.Write(Pop(), 0);
        break;
      case PVM.inpb:          // boolean input
        mem[Pop()] = data.ReadBool() ? 1 : 0;
        break;
      case PVM.prnb:          // boolean output
        if (Pop() != 0) results.Write(" true  ");
          else results.Write(" false ");
        break;
      case PVM.prns:          // string output
        loop = Next();
        while (mem[loop] != 0) {
          results.Write((char) mem[loop]); loop--;
        }
        break;
      case PVM.prnl:          // newline
        results.WriteLine();
        break;
      case PVM.neg:           // integer negation
        Push(-Pop());
        break;
      case PVM.add:           // integer addition
        tos = Pop(); Push(Pop() + tos);
        break;
      case PVM.sub:           // integer subtraction
        tos = Pop(); Push(Pop() - tos);
        break;
      case PVM.mul:           // integer multiplication
        tos = Pop(); Push(Pop() * tos);
        break;
```

```
        case PVM.div:           // integer division (quotient)
          tos = Pop(); Push(Pop() / tos);
          break;
        case PVM.rem:           // integer division (remainder)
          tos = Pop(); Push(Pop() % tos);
          break;
        case PVM.not:           // logical negation
          Push(Pop() == 0 ? 1 : 0);
          break;
        case PVM.and:           // logical and
          tos = Pop(); Push(Pop() & tos);
          break;
        case PVM.or:            // logical or
          tos = Pop(); Push(Pop() | tos);
          break;
        case PVM.ceq:           // logical equality
          tos = Pop(); Push(Pop() == tos ? 1 : 0);
          break;
        case PVM.cne:           // logical inequality
          tos = Pop(); Push(Pop() != tos ? 1 : 0);
          break;
        case PVM.clt:           // logical less
          tos = Pop(); Push(Pop() <  tos ? 1 : 0);
          break;
        case PVM.cle:           // logical less or equal
          tos = Pop(); Push(Pop() <= tos ? 1 : 0);
          break;
        case PVM.cgt:           // logical greater
          tos = Pop(); Push(Pop() >  tos ? 1 : 0);
          break;
        case PVM.cge:           // logical greater or equal
          tos = Pop(); Push(Pop() >= tos ? 1 : 0);
          break;
        case PVM.brn:           // unconditional branch
          cpu.pc = Next();
          break;
        case PVM.bze:           // pop top of stack, branch if false
          int target = Next();
          if (Pop() == 0) cpu.pc = target;
          break;
        case PVM.anew:          // heap array allocation
          int size = Pop();
          Push(cpu.hp);
          cpu.hp += size;
          break;
        case PVM.halt:          // halt
          running = false;
          break;
      }
    } while (running);
  } // PVM.Emulator
```

The reader should note the following points.

- Pushing an item onto the stack is achieved by first decrementing the stack pointer `cpu.sp` and then storing the item at `mem[cpu.sp]`, the memory indexed by the pointer, using a so-called *pre-decrement* approach. Stacks may also be implemented by using *post-decrement*. The code here has the advantage that `cpu.sp` always points directly to the item last pushed onto the stack, but leads to possible confusion in that registers like `cpu.fp` point to a location one higher than one might at first associate with the base of an activation record. It is for the same reason that the memory array has been declared of size one greater than appears necessary.

- The code incorporates calls to the methods of a general purpose I/O library, details of which can be found in Appendix B.

- The emulation of the `LDXA` and `ADD` opcodes will be seen to be identical. Later sections will describe various extensions and modifications to the emulator, when the need to draw a distinction between the two will be more apparent.

4.7 A minimal assembler for PVM code

To be able to use this system we must, of course, have some way of loading or assembling code into memory. An assembler might conveniently be developed using the following specification.

```
class PVMAsm {

  public static bool Assemble(string sourceName)
  // Assembles source code from an input file sourceName and loads
  // codeLength words of code directly into memory
  // PVM.mem[0..codeLength-1], storing strings in the string pool
  // at the top of memory in PVM.mem[stackBase .. memSize-1].
  //
  // Returns true if assembly succeeded
  //
  // Instruction format :
  //  Instruction  = [ Label ] Opcode [ AddressField ] [ Comment ]
  //  Label        = Integer
  //  Opcode       = Mnemonic
  //  AddressField = Integer | "String"
  //  Comment      = String
  //
  // A string AddressField may only be used with a PRNS opcode
  // Instructions supplied one per line; terminated at end of file

  public static int CodeLength()
  // Returns the number of words of code assembled

  public static int StackBase()
  // Returns the address at which the run-time stack will commence

} // class PVMAsm
```

This specification would allow us to develop a range of assemblers. Code for a load-and-go assembler/interpreter using this class might be as simple as

```
class Assem {

  public static void Main(string[] args) {
    PVM.Init();
    if (PVMAsm.Assemble(args[0]))
      PVM.Interpret(PVMAsm.CodeLength(), PVMAsm.StackBase());
  }
} // class Assem
```

The objective of this chapter is to introduce the principles of machine emulation, and not to be too concerned about the problems of assembly. If, however, we confine ourselves to assembling code where the opcodes are denoted by their mnemonics but all the addresses and offsets are written in absolute form, as was done for the examples given earlier, a rudimentary assembler can be written relatively easily.

The essence of this is described informally by an algorithm like

```
BEGIN
  codeLength := 0;
  REPEAT
    SkipLabel;
    IF NOT EOF(SourceFile) THEN
      Extract(Mnemonic);
      Convert(Mnemonic, OpCode);
      mem[codeLength] := OpCode; Increment(codeLength);
      IF OpCode = PRNS THEN
        Extract(String); Store(String, Address);
        mem[codeLength] := Address; Increment(codeLength)
      ELSIF OpCode IN {LDA, LDC, DSP, BRN, BZE} THEN
        Extract(Address); mem[codeLength] := Address;
        Increment(codeLength)
      END;
      IgnoreComments
    END
  UNTIL EOF(SourceFile)
END
```

An implementation of this is to be found in the *Resource Kit*, where code is assumed to be supplied to the assembler in free format, one instruction per line. Comments and labels may be added, as in the examples given earlier, but these are simply ignored by the assembler. Since absolute addresses are required, any labels may be more of a nuisance than they are worth.

4.8 Enhancing the efficiency of the emulator

We have already had cause to remark that an interpretive approach to program compilation and execution results in execution times that can be orders of magnitude longer than would be achieved were the same program to be compiled to native machine code. Developers of interpreters usually attempt to write the code that simulates the fetch-execute cycle as tightly as possible. Although the code in section 4.6 follows the description of the PVM instruction set closely, the many calls to the very simple auxiliary routines `Next`, `Push` and `Pop` would have the effect of slowing interpretation considerably (for reasons that might be more apparent once the reader has understood Chapter 14). Execution speed would be enhanced considerably were the code for these methods simply to be "in-lined" at the points where they are called. In some languages it is possible to indicate to a compiler that such in-lining should take place, but this does not seem possible with the present generation of C# and Java compilers. However, it is not difficult to do this for oneself, as the following representative extracts from an emulator coded in this way will reveal.

```
do {
  cpu.ir = mem[cpu.pc];     // fetch
  cpu.pc++;                 // bump program counter in anticipation
  switch (cpu.ir) {         // execute
    ...
    case PVM.ldc:           // push constant value
      cpu.sp--;  mem[cpu.sp] = mem[cpu.pc]; cpu.pc++;
      break;
    case PVM.lda:           // push local address
      cpu.sp--;  mem[cpu.sp] = cpu.fp - 1 - mem[cpu.pc]; cpu.pc++;
      break;
    case PVM.ldv:           // dereference
      mem[cpu.sp] = mem[mem[cpu.sp]];
      break;
    case PVM.sto:           // store
      cpu.sp++;  mem[mem[cpu.sp]] = mem[cpu.sp - 1]; cpu.sp++;
      break;
    case PVM.add:           // integer addition
      cpu.sp++; mem[cpu.sp] += mem[cpu.sp - 1];
      break;
    case PVM.brn:           // unconditional branch
      cpu.pc = mem[cpu.pc];
      break;
    ...
  }
} while (running);
```

As usual, a complete implementation of the emulator embodying these ideas can be found in the *Resource Kit*. Note that such code could be written even more concisely - and thus probably compiled even more effectively - were we to incorporate the `++` and `--` operators into the various subscripting expressions. This has not been done for clarity; the modification is left as an easy exercise.

4.9 Error handling in the PVM

Although the emulators just described are easily understood and will be found to perform adequately for interpreting programs like those presented in section 4.5, they have several shortcomings that undermine their credibility as a piece of reliable software. In particular, they will react very badly, if at all, to any instruction streams that make no sense. As a fairly extreme example, consider the sequence

```
0   PRNI         ; no value exists on the stack prior to printing
1   DSP  400000  ; insufficient memory to allocate 400000 words
3   BRN  -45     ; -45 is not a valid memory address
```

This would be accepted as syntactically correct by an assembler and recognizable code would be loaded into memory, but the code is clearly devoid of any meaning and would cause trouble when the interpreter attempted to pop values from a corrupt or empty stack, or alter the processor registers to nonsensical values. In fact, it can get even worse than this. The interpreter we have developed has no guarantee that the "code" in lower memory has been produced by any sort of genuine assembler or compiler - it is possible that a totally meaningless array of numbers might have been loaded into the memory by some malicious agent!

There are several approaches that can be taken to alleviate such problems. The most sophisticated of these rely on the program loader attempting to verify, before interpretation commences, that the "code" does, in fact, represent a semantically valid stream of instructions. Such verification lies at the heart of the JVM and the CLR, but is

beyond the scope of the present discussion. Suffice it to comment here that one of the triumphs of modern language design and compiler implementation is that it has become possible for compilers to produce code which is capable of such verification at all (Gough 2002).

Further problems beset the interpretive process. Even if the code sequence were to pass a set of stringent criteria imposed by a critical loader, this still might not anticipate every disastrous situation. To illustrate this, recall Example 4.3 of section 4.5, which showed PVM code corresponding to

```
BEGIN
  Read(X, Y);
  Print(X + 4 / Y);
END
```

This code would have no meaning if the value read for `Y` happened to be zero.

Many of these situations can be detected by an interpreter, provided it is enhanced to include a great many run-time checks on the bounds of the processor registers and the values loaded and stored in "memory". Unfortunately these checks slow the interpretive process considerably. In the absence of a sophisticated loader, that is a price one has to pay. Failure to adopt this approach has the unfortunate effect that if the interpreter runs amok, the likely outcome is that it will eventually collapse and throw exceptions, or come up with run-time error messages that will make absolutely no sense to anyone thinking in terms of the original source code.

It will suffice to illustrate this approach with a few examples and leave the reader with the task of studying the code in the *Resource Kit* to obtain a more complete picture. To the interpreter, firstly, is added the concept that the machine can be in one of several states other than simply "running" or "not running". We can enumerate these states on the lines of

```
const int          // possible machine states
  running  =  0,
  finished =  1,
  badMem   =  2,
  badData  =  3,
  noData   =  4,
  divZero  =  5,
  badOp    =  6,   // and others like this
```

We then introduce a state variable to monitor the execution state

```
static int ps;     // program status
```

and provide a diagnostic method that can be invoked to abort interpretation with a suitable message if disaster strikes

```
static void PostMortem(OutFile results, int pcNow) {
// Reports run-time error and position
  results.WriteLine();
  switch (ps) {
    case badMem:  results.Write("Memory violation"); break;
    case badData: results.Write("Invalid data"); break;
    case noData:  results.Write("No more data"); break;
    case divZero: results.Write("Division by zero"); break;
    case badOp:   results.Write("Illegal opcode"); break;
    ...
```

At each point in the fetch-execute cycle where a processor register is used, it is checked to ensure that its value lies in an appropriate range. Our extract shows the modified form of the interpreter loop and some representative extended options in the `switch` statement:

```
      ps = running;               // prepare to execute
      do {
        pcNow = cpu.pc;           // retain for tracing/postmortem
        if (cpu.pc < 0 || cpu.pc >= codeLen) ps = badAdr;
        else if (mem[cpu.pc] < 0 || mem[cpu.pc] > PVM.nul) ps = badOp;
        else if (cpu.sp <= cpu.hp || cpu.sp > memSize) ps = badMem;
        else {
          cpu.ir = mem[cpu.pc];   // fetch
          cpu.pc++;               // bump cpu.pc in anticipation
          switch (cpu.ir) {       // execute
            ...
```

```
            case PVM.ldv:         // dereference
              if (InBounds(mem[cpu.sp]))
                mem[cpu.sp] = mem[mem[cpu.sp]];
              break;
            ...
            case PVM.div:         // integer division (quotient)
              cpu.sp++;
              if (mem[cpu.sp - 1] == 0) ps = divZero;
              else mem[cpu.sp] /= mem[cpu.sp - 1];
              break;
            ...
            case PVM.brn:         // unconditional branch
              if (mem[cpu.pc] < 0) ps = badAdr;
              else cpu.pc = mem[cpu.pc];
              break;
            ...
            default:              // unrecognized opcode
              ps = badOp;
              break;
          }
        }
      } while (ps == running);
      if (ps != finished) PostMortem(results, pcNow);
```

It may be helpful to pay more attention to errors that may arise when handling arrays. Code of the form

```
void main () {
  int[] list = new int[100000];
  bool[] sieve;
  list[-1] = 0;
  sieve[0] = true;
} // main
```

displays several of these. In particular, there may not be sufficient memory to allocate that many words to the array `list`, a reference to `list[-1]` takes one out of the bounds of the array (valid subscripts are confined to the range 0 … 99999), and the array `sieve` does not have storage allocated to it at the point where it is first accessed.

These situations can be handled in various ways. It turns out to be advantageous to extend the heap manager so that an array effectively becomes "self-describing" - the first word of the storage set aside for it recording the number of elements in the array. Using this scheme, the memory allocation of the arrays allocated in Example 4.5.5 would look as follows (compare this with the simpler scheme illustrated in Figure 4.6).

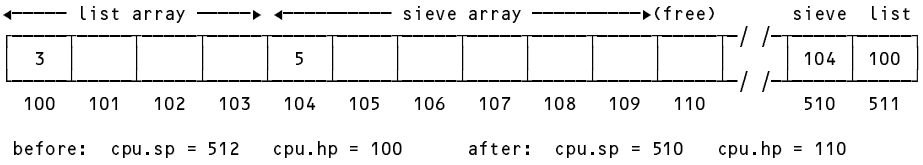


Figure 4.7 Memory allocation after dynamic allocation of two arrays

You will recall that the execution of the ANEW instruction expects to find the size to be allocated to the array on the top of the stack. Responsibly it should check that this size is positive and that there is sufficient storage between the current base of the heap and the current top of the stack for the allocation to be allowed. Code that achieves this follows.

```
          case PVM.anew:        // heap array allocation
            int size = mem[cpu.sp];
            if (size <= 0 || size + 1 > cpu.sp - cpu.hp)
              ps = badAll;
            else {
              mem[cpu.hp] = size;
              mem[cpu.sp] = cpu.hp;
              cpu.hp += size + 1;
            }
            break;
```

The address arithmetic for computing the run-time address of an array element like `list[i]` at first appears straightforward. We observed earlier that the LDXA instruction might appear to be superfluous, as its interpretation seems to be identical to that of ADD. It expects to find the value of the subscript `i` on the top of the stack, and the heap address of the storage allocated to `list` immediately above that. However, responsible indexing should check that the heap address is valid, and that the subscript lies within range. The heap address might be invalid

because storage for the array had never been claimed. In the best situation this would show up as a "null pointer reference" if one could somehow guarantee that all reference variables were initialized to some recognizable null value (the equivalent, say, of zero) as execution commenced. In other situations the best one might hope for would be to detect that the value purported to be a heap address actually lay somewhere in the heap area, although this is not totally reliable. Given that those conditions are met, the allocated size of the array can be deduced from its own description on the heap and the index value checked against this. Code for achieving all this follows, and is worthy of scrutiny.

```
case PVM.ldxa:        // heap array indexing
  int heapPtr = mem[cpu.sp + 1];
  if (heapPtr == 0) ps = nullRef;
  else if (heapPtr < heapBase || heapPtr >= cpu.hp)
    ps = badMem;
  else if (mem[cpu.sp] < 0 || mem[cpu.sp] >= mem[heapPtr])
    ps = badInd;
  else {
    cpu.sp++; mem[cpu.sp] += 1 + mem[cpu.sp - 1];
  }
  break;
```

Ensuring that all pointer references (and, indeed, all variables) are initialized to zero as execution commences might be the responsibility of an enhanced `DSP` instruction.

```
case PVM.dsp:             // decrement stack pointer
  cpu.sp -= mem[cpu.pc];  // allocate space for variables
  if (InBounds(cpu.sp)) {
    for (loop = 0; loop < mem[cpu.pc]; loop++)
      mem[cpu.sp + loop] = 0;  // initialize
    cpu.pc++;
  }
  break;
```

4.10 Enhancing the instruction set of the PVM

The PVM as so far defined falls into the category of RISC (Reduced Instruction Set Computer) machines. There are fewer than 40 instructions. In Chapter 14 we shall add a few instructions to deal with function calls and access to global variables, but it turns out that not many of these are needed if we are content to limit ourselves to an "integer only" machine.

For simplicity, the PVM has used the capacity of a full integer to encode an instruction. This is very wasteful of space. It is not difficult to develop a system that uses bytes as the fundamental unit of simulated memory, and we could encode up to 255 different instructions using the capacity of a single byte. Most of our instructions are of the zero-address form and the implications for densely packed code should be obvious. The question arises as to whether what we have is an optimal set of instructions - that is, whether we could dedicate some of the hitherto unused encodings to good effect. This section is intended to explore some of the possibilities.

4.10.1 Encoding addresses within instructions

Examination of the code examples given earlier will reveal that many instructions like `LDA N` and `LDC N` only specify small values for the parameter `N`. This suggests that we might use specialized opcodes to handle these situations more effectively. Introducing instructions with mnemonics like `LDA_n`, `LDC_n` or `LDC_Mn`, where *n* is a small integer, can achieve the desired effect, as the following variation on the code shown in Example 4.5.1 demonstrates.

```
    X := -1; Y := X;

0   DSP  2   ; reserve 2 variables - X is var 0, Y is var 1
2   LDA_0    ; push address of X
3   LDC_M1   ; push constant -1
4   STO      ;        X := -1
5   LDA_1    ; push address of Y
6   LDA_0    ; push address of X
7   LDV      ; dereference - value of X (-1) now on top of stack
8   STO      ;        Y := X
9   ...
```

The implication is that we use only 8 words of code, and not 12 as before - a considerable saving. This would

have the added advantage that the fetch-execute cycle would probably run faster, since there are fewer extra words to "fetch". Clearly there is a limit in how far one would want to exploit this idea.

4.10.2 Loading and storing variables

Examination of the code in earlier examples and in others will reveal that sequences like

```
LDA  N
LDV
```

and

```
LDA  N
(calculations)
STO
```

occur frequently. This might suggest that the machine could benefit from the introduction of special purpose load and store instructions, say `LDL` (load local) and `STL` (store local). In the notation used in section 4.4.2, we might define the effect of these by

`LDL  N` Push value of `mem[CPU.FP - 1 - N]` onto stack to form new `TOS`
`STL  N` Pop `TOS` and store `TOS` in `mem[CPU.FP - 1 - N]`

As an example, consider how one might handle a high-level statement like

```
Z := X + 4 / Y;

0   LDL  0   ; push value of X
2   LDC_4    ; push constant 4 onto stack
3   LDL  1   ; push value of Y
5   DIV      ; pop Y and 4 - push back 4/Y
6   ADD      ; pop 4/Y and X - push back X+4/Y
7   STL  2   ; store value of X+4/Y in variable Z
```

This idea could be combined with those of the last section, leading to special encodings in situations where the parameter `N` is a small integer, exemplified by `LDL_n` and `STL_n`.

In the case of array access the corresponding common sequences are

```
LDL  N
(compute and push value of subscript)
LDXA
LDV
```

and

```
LDL  N
(compute and push value of subscript)
LDXA
(calculations)
STO
```

which would be simplified were we to introduce two instructions `LDE` (load element) and `STE` (store element) described by

`LDE` Pop `TOS` and `SOS` and push `mem[TOS + SOS + 1]` onto stack to form new `TOS`
`STE` Pop `TOS`, `SOS` and `NOS`, and store `TOS` in `mem[SOS + NOS + 1]`

where the `+ 1` compensates for the fact that the array size is recorded in part of the storage allocated to the array. As usual, an example may clarify. The six words of highly compact code below correspond to the assignment

```
    Z[3] := Y[X];

0   LDL_0    ; push value of Z (reference to array Z)
1   LDC_3    ; push constant 3 onto stack (index to array Z)
2   LDL_1    ; push value of Y (reference to array Y)
3   LDL_2    ; push value of X (value of subscript expression)
4   LDE      ; pop reference Y and subscript X
             ; and push back value of Y[X]
5   STE      ; pop value of Y[X], subscript (3) and Z
             ; and assign Y[X] to Z[3]
```

4.10.3 Comparisons and branching

The instructions of the form Cxx introduced in section 4.4.2 are convenient for computing the values of very simple Boolean expression such as X < Y. Frequently these expressions are used in conjunction with conditional branch instructions, leading to code like

```
0   LDL_0    ; push value of X (variable 0)
1   LDL_1    ; push value of Y (variable 1)
2   CEQ      ; pop X and Y, push 1 if X = Y, push 0 if X ≠ Y
3   BZE 123  ; branch to 123 if X ≠ Y
```

As an alternative (or in addition) to the instructions already defined we might define a set where the comparisons are directly combined with the branching:

BEQ N Pop TOS and SOS, branch to N if SOS = TOS
BNE N Pop TOS and SOS, branch to N if SOS ≠ TOS
BGT N Pop TOS and SOS, branch to N if SOS > TOS
BLT N Pop TOS and SOS, branch to N if SOS < TOS
BLE N Pop TOS and SOS, branch to N if SOS ≤ TOS
BGE N Pop TOS and SOS, branch to N if SOS ≥ TOS

in terms of which the above example might have been coded

```
0   LDL_0    ; push value of X (variable 0)
1   LDL_1    ; push value of Y (variable 1)
2   BNE 123  ; branch to 123 if X ≠ Y
```

4.11 Another stack machine - the JVM

The PVM discussed in the last few sections is but one of a number of similar virtual machines that one could construct. The JVM, first mentioned in section 2.7 is, understandably, considerably more complex. For a start it supports the execution of programs that require a much wider range of types, and this alone necessitates the existence of many more opcodes. Discussion of all the ramifications of the JVM is outside the scope of this text but it will be helpful to give a broad overview for the purposes of comparison, especially as in later chapters (of the complete book (Terry, 2005)) we develop a simple compiler that generates assembler code for the JVM.

Discussion of the PVM has focused on the use of an array of words to simulate the memory of the machine and on how parts of this array are put to good effect to form a code area, a heap, a stack and so on, all of which can be manipulated by a simulated processor. Conceptually the JVM can also be regarded as having several data areas and an execution engine, as indicated in Figure 4.9.

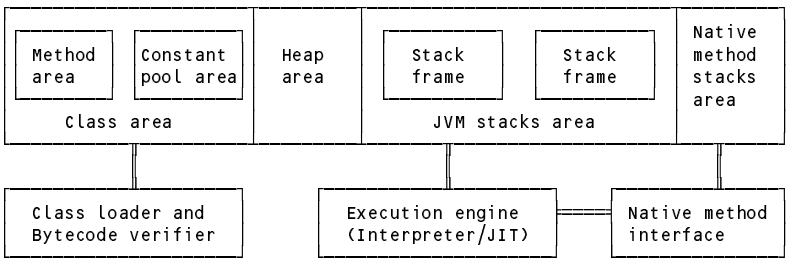


Figure 4.9 A programmer's model of the JVM

The **class area** and **heap area** are shared between all the threads of a program - each thread has its own **JVM stacks** area and its own PC. As each class is verified and loaded, the code for its methods is kept in the **method area**, and its constants and metadata (type information) are kept in the **constant pool area**. As and when an object or array is created, storage for the fields or elements is allocated from the heap area.

Whenever a method is called, storage for its local variables, arguments and other housekeeping data is allocated from a newly created **stack frame** in the JVM stacks area, along with an **operand stack** for the method to use in the computation of intermediate results.

The parallels with the description of the PVM should be fairly clear, but the JVM has additional features not found in the simpler system. For example, the PVM has no facilities for instantiating classes or for reclaiming storage from the heap when the arrays created there are no longer required. The JVM is required to provide automatic **garbage collection** to clear the heap area of objects and arrays that it can detect are no longer in use. The JVM also makes provision for sophisticated interaction between the methods coded in JVM code and methods coded in native code (so-called **native methods**) which may be necessary for using features of the "real machine".

As was the case in the PVM, the local variables of each method may be thought of as elements of a local array, indexed from zero (in the JVM the arguments supplied to the method are also stored in this space). Provision is made in the instruction set for transferring values between this area and the operand stack, and there is a close resemblance between some of the code for the JVM and the PVM. For example, the code corresponding to

```
int a, b, c;
c = 400000 + a * b;
```

is essentially identical, save for the mnemonics used:

```
LDC   400000          ldc     400000
LDL_1                 iload_1
LDL_2                 iload_2
MUL                   imul
ADD                   iadd
STL_0                 istore_0
```

Compilers for the JVM are required to be able to predict the extent of the local variable storage and the maximum depth to which the operand stack will grow. The elements of the operand stack and the "slots" in the local variable area are 32 bits wide. Some types - `long` and `double` - require 64 bits of storage, and these are catered for by using two slots or stack elements. Types like `boolean, byte` and `char` might seem to require only 1, 8 or 16 bits of storage (respectively), but the local variable slots and stack elements are still allocated in 32-bit units, with explicit conversions being introduced into the code where needed. However, arrays allocated from the heap are always managed rather more economically.

The instruction set for the JVM has about 250 opcodes and much of a stream of instructions will consist of single byte "zero address" code. Some instructions, clearly, have arguments, which in binary class files are often index values into the constant pool for the class, although at the level of assembler source this may not be apparent.

4.12 The CLR - a conceptual stack machine

Many of the features of the JVM, both those in common with the PVM and those that distinguish it from the simpler PVM, appear to have exact parallels in the virtual machine that forms the conceptual basis of the CLR for .NET, especially if one approaches the machine from the viewpoint of a programmer developing assembler level code for processing with the *ilasm* assembler.

As the CLR is intended to support the execution of programs developed in one of a wide range of languages, or even developed in a mixture of languages, the underlying type system is even more extensive than that of the JVM. By analogy with the JVM, we can think of assemblies being verified and having their CIL code and metadata loaded into a "class area", but the analogy is dangerous, as the .NET system relies on a JIT compiler to turn the CIL code into native code for the host platform. As in the case of the JVM, one can think in terms of an activation record being created as a method is called, and in terms of part of this being used to store the local variables and arguments of the method. An interesting difference is that in the CLR the arguments passed to a method are regarded as being stored in their own indexed section of the activation record, distinct from the indexed section used to store the local variables.

Although the some 220 instructions for the CLR do not match one-for-one with those in the JVM, there is often a close resemblance between code for the PVM, the JVM and the CLR. CIL code corresponding to the example in section 4.11

```
int a, b, c;
c = 400000 + a * b;
```

is essentially identical to the PVM and JVM code, save for the mnemonics used:

```
LDC   400000        ldc     400000      ldc.i4   400000
LDL_1               iload_1             ldloc.1
LDL_2               iload_2             ldloc.2
MUL                 imul                mul
ADD                 iadd                add
STL_0               istore_0            stloc.0
```

All of this is, in fact, somewhat illusory, although it has the advantage of conceptual simplicity. In particular, the slots in the local variable and parameter areas and on the evaluation stack (the counterpart of what the JVM calls the operand stack) are "virtual slots". Although they follow a simple logical numbering so far as the assembler programmer is concerned, they do not all have the same fixed size. This, and the apparent ability of the CLR to use generic or polymorphic instructions (like `mul` and `add`) for many operations, rather than the type-specific ones needed for the JVM (like `imul` and `iadd`), are further consequences of the reliance on a JIT compiler. Furthermore, although the illustrations above only demonstrate the use of simple integral arguments, some instructions employ operands that are considerably more complex references. By the time the CIL code above is actually executed it may have been translated into native code that is quite a long way removed from what appears here.

As for the JVM, compilers for the CLR must predict the logical extent of the local variable and parameter storage areas and the maximum depth to which the evaluation stack will grow. And as in the case of the JVM, arrays and other objects are created on demand to utilize storage efficiently from the heap area, storage which is automatically reclaimed by the garbage collector when it detects that the objects are no longer accessible.

Further reading

Appendix A of Terry (2005) contrasts the instruction sets and type models for the PVM, JVM and CLR in more detail, at least as far as the facilities of these machines are used in the case studies in this text. Detailed descriptions of the JVM and CLR are long and involved. The interested reader might find the definitive description of the JVM in the book by Lindholm and Yellin (1999) quite hard work at first - an easier introduction appears in the book by Engel (1999). The book by Venners (1999) is also very comprehensive. So far as the CLR is concerned, the definitive description is to be found in the ECMA standards originating from Microsoft and available at `http://msdn.microsoft.com/net/ecma`. The book by Gough (2002) is very full of insight, and the book by Lidin (2002) offers a complete reference for the ILASM programmer.

The very comprehensive stack-based interpreter for the Zürich Pascal-P system is fully described in the book by Pemberton and Daniels (1982). Another good discussion of stack-based interpreters is found in the book by Watt and Brown (2000).

5 LANGUAGE SPECIFICATION

A study of the syntax and semantics of programming languages can be made at many levels, and is an important part of modern computer science. One can approach it from a very formal viewpoint or from a very informal one. In this chapter we shall mainly be concerned with ways of specifying the concrete syntax of languages in general, and programming languages in particular. This forms a basis for the further development of the syntax-directed translation upon which much of the rest of this text depends.

5.1 Syntax, semantics, and pragmatics

People use languages in order to communicate. In ordinary speech they use natural languages like English or French; for more specialized applications they use technical languages like those of music or mathematics, for example:

$$\forall\ x\ \exists\ \varepsilon\ ::\ |\ x - \xi\ |\ <\ \varepsilon$$

We are mainly concerned with programming languages, which are notations for describing computations. (As an aside, the word "language" is regarded by many to be unsuitable in this context. The word "notation" is preferable; we shall, however, continue to use the traditional terminology.) A useful programming language must be suited both to *describing* and to *implementing* the solution to a problem, and it is difficult to find languages which satisfy both requirements - efficient implementation seems to require the use of low-level languages, while easy description seems to recommend the use of high-level languages.

Most people are taught their first programming language by example. This is admirable in many respects, and probably unavoidable, since learning the language is often carried out in parallel with the more fundamental process of learning to develop algorithms. But the technique suffers from the drawback that the tuition is incomplete - after being shown only a limited number of examples, one is inevitably left with questions of the "Can I do this?" or "How do I do this?" variety. In recent years a great deal of effort has been spent on formalizing programming (and other) languages, and in finding ways to describe them and to define them. Of course, a formal programming language has to be described by using another language. This language of description is called the **metalanguage**. Early programming languages were described using English as the metalanguage. A precise specification requires that the metalanguage be completely unambiguous. This is not a strong feature of English, although politicians and comedians rely heavily on ambiguity in spoken languages in pursuing their careers! Some beginner programmers find that the best way to answer the questions which they may have about a programming language is to ask them of the compilers which implement the language. This is highly unsatisfactory, as compilers are known to be error-prone and to differ in the way they handle a particular language.

Natural languages, technical languages and programming languages are alike in several respects. In each case the **sentences** of a language are composed of sets of **strings** of **symbols**, **tokens** or **words**, and the construction of these sentences is governed by the application of two sets of rules.

- **Syntax Rules** describe the *form* of the sentences in the language. For example, in English the sentence "They can fish" is syntactically correct, while the sentence "Can fish they" is incorrect. To take another example, the language of binary numerals uses only the symbols 0 and 1, arranged in strings formed by concatenation, so that the string "101" is a syntactically correct sentence for this language, while the string "1110211" is syntactically incorrect.

- **Semantic Rules**, on the other hand, define the *meaning* of syntactically correct sentences in a language. By itself the sentence "101" has no meaning without the addition of semantic rules to the effect that it is to be interpreted as the representation of some number using a positional convention. The sentence "They can fish" is more interesting, for it can have two possible meanings; a set of semantic rules would be even harder to formulate.

The formal study of syntax as applied to programming languages took a great step forward, in about 1960, with the publication of the *Algol 60 report* by Naur (1960, 1963), which used an elegant, yet simple, notation known as **Backus-Naur form** (sometimes called **Backus normal form**) which we shall study shortly. Simply understood notations for describing semantics have not been so forthcoming, and many semantic features of languages are still described informally or by example.

Besides being aware of syntax and semantics, the user of a programming language cannot avoid coming to terms with some of the pragmatic issues involved with implementation techniques, programming methodology and so on. These factors govern subtle aspects of the design of almost every practical language, often in a most irritating way. For example, in Fortran 66 and Fortran 77 the length of an identifier was restricted to a maximum of six characters - a legacy of the word size on the IBM computer for which the first Fortran compiler was written.

5.2 Languages, symbols, alphabets and strings

To understand how programming languages are specified rigorously one must be aware of some features of **formal language theory**. We start with a few abstract definitions.

- A **symbol** or **token** is an atomic entity, sometimes represented by a single character but sometimes by a reserved or key word, for example `+ , ; END`.

- An **alphabet** A is a non-empty, but finite, set of symbols. For example, the alphabets of Modula-2 and Pascal include the symbols

  ```
  -   /   *   a   b   c   A   B   C   BEGIN   IF   CASE   END
  ```

  while those for C++, C# or Java would include a corresponding set

  ```
  -   /   *   a   b   c   A   B   C   {   if   switch   }
  ```

- A **phrase**, **word** or **string** "over" an alphabet A is a sequence $\sigma = a_1 a_2 \ldots a_n$ of symbols from A.

- It is often useful to hypothesize the existence of a string of length zero, called the **null string** or **empty word**, usually denoted by ε (some authors use λ instead). This has the property that if it is concatenated to the left or right of any word, that word remains unaltered.

$$a\,\varepsilon = \varepsilon\,a = a$$

- The set of all strings of length n over an alphabet A is denoted by A^n. The set of all strings (including the null string) over an alphabet A is called its **Kleene closure** or, simply, **closure** and is denoted by A^*. The set of all strings of length at least one over an alphabet A is called its **positive closure**, and is denoted by A^+. Thus

$$A^* = A^0 \cup A^1 \cup A^2 \cup A^3 \ldots = A^0 \cup A^+$$

- A **language** L over an alphabet A is a subset of A^*. At the present level of discussion this involves no concept of meaning. A language is simply a set of strings that conform to some syntax rules. A language consisting of a finite number of strings can be defined simply by listing all those strings, or giving a rule for their derivation. This may even be possible for simple infinite languages. For example, we might have

$$L = \{\, (\,[a+\,)^n\,(\,b]\,)^n \mid n > 0 \,\}$$

  (the vertical stroke can be read "such that"), which defines exciting expressions like

$$[a + b]$$
$$[a + [a + b]\, b]$$
$$[a + [a + [a + b]\, b]\, b]$$

5.3 Regular expressions

Several simple languages - but by no means all - can be conveniently specified using the notation of **regular expressions**. Although we shall make very little use of this notation in the rest of the book, for completeness it is of interest to consider it briefly. A regular expression specifies the form that a string may take by using the symbols from the alphabet A in conjunction with a few other **metasymbols**, which *do* have meaning, and represent operations that allow for

- *concatenation*: symbols or strings may be concatenated by writing them next to one another, or by using the metasymbol · (dot) between them if further clarity is required;

- *alternation*: a choice between two symbols a and b is indicated by separating them by the metasymbol | (bar);

- *repetition*: a symbol a followed by the metasymbol ***** (star) indicates that a sequence of zero or more occurrences of a is allowable;

- *grouping*: a group of symbols may be surrounded by the metasymbols **(** and **)** (parentheses).

As an example of a regular expression, consider

$$R \quad = \quad 1 \; (\; 1 \; | \; 0 \;)^* \; 0$$

This generates the set of strings each of which has a leading 1, is followed by any number of 0s or 1s and is terminated with a 0. We may now speak of a language $L(R)$ as defined by the set

$$L(R) \quad = \quad \{ \, 10, \, 100, \, 110, \, 1000 \, \ldots \, \}$$

If a semantic interpretation is required, the reader will recognize this as the set of strings representing non-zero even numbers in a binary representation.

Formally, regular expressions may be defined inductively as follows:

- a regular expression R denotes a regular set of strings;

- $R = \varnothing$ is a regular expression denoting the empty set (that is, $L(\varnothing) = \varnothing = \{ \, \}$);

- $R = \varepsilon$ is a regular expression denoting the set that contains only an empty string (that is, $L(\varepsilon) = \{\varepsilon\}$);

- $R = \sigma$ is a regular expression denoting a set containing only the symbol σ (that is, $L(\sigma) = \{ \, \sigma \, \}$);

- if A and B are regular expressions, then $(\, A \,)$ and $A \mid B$ and $A \cdot B$ and A^* are also regular expressions. That is, $L(\, (A) \,) = L(A)$, $L(A|B) = L(A) \cup L(B)$, $L(A \cdot B) = L(A) \; L(B)$ and $L(A^*) = L(A)^*$.

Thus, for example, if σ and τ are strings generated by regular expressions, $\sigma\tau$ and $\sigma \cdot \tau$ are also generated by a regular expression.

The reader should take note of the following points.

- As in arithmetic, where multiplication and division take precedence over addition and subtraction, there is a precedence ordering between the operators within regular expressions. Parentheses take precedence over repetition, which takes precedence over concatenation, which in turn takes precedence over alternation. Thus, for example, the following two regular expressions are equivalent

  his | hers and h (i | er) s

  and both define the set of strings { his , hers } (and not, please note { hisers , hihers }).

- It is often useful to give a regular expression a descriptive name, for example

  owner = his | hers

  but care must be taken not to read too much into the apparent semantic meaning of such names. After names have been given to regular expressions it is convenient to use these as factors in writing further regular expressions, for example

  responsibility = *owner* | mine

In this case some notational trick has to be employed - such as the *italic* font used above - to distinguish named subexpressions from other strings where the characters stand for themselves. The above regular expression defines a language *L(responsibility)* = { his, hers, mine }, whereas

culpability = owner | mine

defines a language *L(culpability)* = { owner, mine }.

- Regular expressions have a variety of algebraic properties, among which we can draw attention to:

$A \mid B = B \mid A$	(commutativity for alternation)
$A \mid (B \mid C) = (A \mid B) \mid C$	(associativity for alternation)
$A \mid A = A$	(absorption for alternation)
$A \cdot (B \cdot C) = (A \cdot B) \cdot C$	(associativity for concatenation)
$A (B \mid C) = A B \mid A C$	(left distributivity)
$(A \mid B) C = A C \mid B C$	(right distributivity)
$A \varepsilon = \varepsilon A = A$	(identity for concatenation)
$A^* . A^* = A^*$	(absorption for closure)

- Further metasymbols are sometimes introduced. For example, the positive closure symbol $^+$ may be used to allow a^+ as an alternative representation for $a\ a^*$. A question mark is sometimes used to denote an optional instance of a, so that a? denotes $a \mid \varepsilon$. The period or fullstop is sometimes used to denote "any character", so that *L(a.b)* = { *aab, abb, acb, adb* …}. Brackets, hyphens and ellipses are often used in place of parentheses and bars, so that [*a-eBC*] denotes (*a*|*b*|*c*|*d*|*e*|*B*|*C*). The tilde ~ or carat ↑ is sometimes used to denote "any character other than", so that ~(*a*|*e*|*i*|*o*|*u*) or ↑[*aeiou*] would be regular expressions denoting consonants.

- Occasionally the metasymbols themselves are required to be members of the alphabet. One convention is to enclose them in quotes when they appear as simple symbols within the regular expression. For example, the regular expression

bullets = "(" (a | b | c | d) ")"

defines a limited language whose four sentences are easily seen to be:

L(bullets) = { (a) , (b) , (c) , (d) }

Regular expressions are of practical interest in programming language translation because they can be used to specify the structure of the tokens (like identifiers, literal constants and strings) whose recognition is the prerogative of the scanner (lexical analyzer) phase of a compiler.

For example, the set of integer literals in many programming languages is described by the regular expression

(0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)$^+$

or, more verbosely, by

(0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9) · (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)*

or, more concisely, by

[0-9]$^+$

and the set of permissable identifiers might be described by a similar regular expression

(a | b | c | ... | z) · (0 | 1 | ... | 9 | a | ... | z)*

or, more concisely, by

[a-zA-Z][a-zA-Z0-9]*

or, more descriptively, by the set of regular expressions

letter = `[a-zA-z]`
letterordigit = `[a-zA-z0-9]`
identifier = *letter letterordigit**

Regular expressions are also powerful enough to describe complete simple assembler languages of the sort illustrated in the last chapter, although the complete expression is rather tedious to write down and so is left as an exercise for the zealous reader.

It is important to appreciate that the regular expression has some severe limitations. For example, although we can describe the language $L = \{ a^m\ b\ a^n \mid m,n \geq 0 \}$ very easily by the regular expression a^*ba^*, we cannot describe the slightly more restrictive language $L = \{ a^n\ b\ a^n \mid n \geq 0 \}$. Use of the Kleene closure symbol does not imply any ability to count. Furthermore, although we have suggested that a complex regular expression may often conveniently be factorized and expressed in terms of simpler regular expressions, it must always be possible to combine the factors together to form a single regular expression by substitution. We must not succumb to the temptation to use these named factors recursively. Thus, for example, we cannot write

digit = `[0-9]`
number = *digit* | *digit number*

although this sort of expressive rule is allowed in the more powerful ideas of context-free grammars, to which the rest of this chapter is devoted.

Further reading

Material on regular expressions is to be found in all books on compilers and syntax analysis. Particularly good treatments are to be found in the books by Gough (1988) and Louden (1997).

5.4 Grammars and productions

Regular expressions have severe limitations in describing even some small languages that are little more than mathematical or text-book curiosities. It should come as little surprise that most practical languages are far more complicated than can be defined by regular expressions. In particular, regular expressions are not powerful enough to describe languages that manifest *self-embedding* in their descriptions. Self-embedding comes about, for example, in describing structured statements which have components that can themselves be statements, or expressions comprised of factors that may contain further parenthesized expressions, or variables declared in terms of types that are structured from other types, and so on.

Thus we move on to consider the notion of a **grammar**. This is essentially a set of rules for describing **sentences** - that is, choosing the subsets of A^* in which one is interested. Formally, a grammar G is defined by a quadruple $\{ N, T, S, P \}$ with the four components:

(a) N - a finite set of **non-terminal** symbols;
(b) T - a finite set of **terminal** symbols;
(c) S - a special **goal** or **start** or **distinguished** symbol;
(d) P - a finite set of **production rules** or, simply, **productions**.

We stress again that the word "set" is used here in the mathematical sense. A sentence is a string composed entirely of terminal symbols chosen from the set T. On the other hand, the set N denotes the **syntactic classes** of the grammar, that is, general components or concepts used in describing sentence construction.

The union of the sets N and T denotes the **vocabulary** V of the grammar

$$V = N \cup T$$

but the sets N and T are required to be disjoint, so that

$$N \cap T = \varnothing$$

where $\varnothing$ is the empty set.

When referring to the strings permitted by a grammar, use is often made of the closure operators. Thus, if a string α consists of zero or more terminals (and no non-terminals) we should write

$$\alpha \in T^*$$

while if α consists of one or more non-terminals (but no terminals)

$$\alpha \in N^+$$

and if α consists of zero or more terminals and/or non-terminals

$$\alpha \in (N \cup T)^* \qquad \text{that is,} \quad \alpha \in V^*$$

A convention often used when discussing grammars in a theoretical sense is to use lower-case Greek letters (α, β, γ, …) to represent strings of terminals and/or non-terminals, capital Roman letters from the start of the alphabet (*A, B, C* …) to represent single non-terminals, lower-case Roman letters from the start of the alphabet (*a, b, c* …) to represent single terminals, and lower-case Roman letters from the end of the alphabet (*x, y, z*) to represent strings of terminals. In terms of the components of G we should write

$$A,\ B,\ C\ \ldots \in N$$
$$a,\ b,\ c\ \ldots \in T$$
$$x,\ y,\ z\ \ldots \in T^*$$

Each author seems to have his or her own set of conventions, so the reader should be on guard when consulting the literature.

English words like *sentence* or *noun* are often used as the names of non-terminals. When describing programming languages, reserved or key words (like `do`, `while`, `switch`) are inevitably terminals. The distinction between these is sometimes made with the use of different type face - we shall use *italic font* for non-terminals and `monospaced font` for terminals where it is necessary to draw a firm distinction.

This probably all sounds very abstruse, so let us try to enlarge a little by considering English as a written language. The set T here would be one containing the 26 letters of the common alphabet, and punctuation marks. The set N would be the set containing syntactic descriptors - simple ones like *noun*, *adjective*, *verb*, as well as more complex ones like *noun phrase*, *adverbial clause* and *complete sentence*. The set P would be one containing syntactic rules, such as a description of a *noun phrase* as a sequence of *adjective* followed by *noun*. This set might become very large indeed - much larger than T or even N. The productions, in effect, tell us how we can *derive* sentences in the language. We start from the distinguished symbol *S* (which is always a non-terminal such as *complete sentence*) and, by making successive substitutions for the non-terminals *N* as allowed by the productions *P*, work through a sequence of so-called **sentential forms** towards a final **sentence**, a string which contains terminals only.

There are various ways of specifying productions. Essentially a production is a rule relating to a pair of strings, say γ and δ, specifying how one may be transformed into the other. Sometimes they are called **rewrite rules** or **syntax equations** to emphasize this property. One way of denoting a general production is

$$\gamma \rightarrow \delta$$

To move on towards completing these rather abstract definitions, let us suppose that σ and τ are two strings each consisting of zero or more non-terminals and/or terminals (that is, $\sigma,\ \tau \in V^* = (N \cup T)^*$).

- If we can obtain the string τ from the string σ by employing *one* of the productions of the grammar *G*, then we say that σ *directly produces* τ (or that τ *is directly derived from* σ), and express this as $\sigma \Rightarrow \tau$.

  That is, if $\sigma = \alpha\delta\beta$ and $\tau = \alpha\gamma\beta$, and $\delta \rightarrow \gamma$ is a production in *G*, then $\sigma \Rightarrow \tau$.

- If we can obtain the string τ from the string σ by applying n productions of *G*, with $n \geq 1$, then we say that σ *produces* τ *in a non-trivial way* (or that τ *is derived from* σ *in a non-trivial way*), and express this as $\sigma \Rightarrow^+ \tau$.

That is, if there exists a sequence $\alpha_o, \alpha_1, \alpha_2, \ldots \alpha_k$ (with $k \geq 1$), such that

$$\begin{array}{lcll} \sigma & = & \alpha_o, & \\ \alpha_{j-1} & \Rightarrow & \alpha_j & \text{(for } 1 \leq j \leq k\text{)} \\ \alpha_k & = & \tau, & \end{array}$$

then $\sigma \Rightarrow^+ \tau$.

- If we can produce the string τ from the string σ by applying n productions of G, with $n \geq 0$ (this includes the above and, in addition, the trivial case where $\sigma = \tau$), then we say that σ *produces* τ (or that τ *is derived from* σ), and express this $\sigma \Rightarrow^* \tau$.

In terms of all these ideas, a **sentential form** is the goal or start symbol S, or *any* string that can be derived from it - that is, any string σ such that $S \Rightarrow^* \sigma$. **Sentences** are strings that contain terminals only. Furthermore, we can now define a language $L(G)$ produced by a grammar G by the relation

$$L(G) = \{ w \mid S \Rightarrow^* w \wedge w \in T^* \}$$

There are a few other definitions that we need to mention.

A grammar is called *recursive* if it permits derivations of the form $A \Rightarrow^+ \omega_1 A \omega_2$, (where $A \in N$, and $\omega_1, \omega_2 \in V^*$). If both ω_1 and ω_2 appear in the derivation it is said to display *self-embedding*. If, as often happens, one or the other is absent, the grammar is called *left recursive* if $A \Rightarrow^+ A \omega$ and *right recursive* if $A \Rightarrow^+ \omega A$.

We have already mentioned the concept of syntax trees when discussing the phases of compilation. We shall refer to these trees on numerous occasions. As our last venture into formalism for the present, we shall define the concept of a **parse tree** over a grammar G as a rooted labelled tree such that

- each node is labelled with a terminal, a non-terminal, or with ε;
- each leaf node is labelled with a terminal or with ε;
- each interior node is labelled with a non-terminal - in particular the root node is labelled with the start symbol S;
- if an internal node with label $A \in N$ has n children with labels $X_i \in V^*$, $i = 1(1)n$ then the grammar must contain a production rule of the form $A \rightarrow X_1 X_2 \ldots X_n$.

5.5 Classic BNF notation for productions

As we have remarked, a production is a rule relating to a pair of strings, say γ and δ, specifying how one may be transformed into the other. This can be denoted $\gamma \rightarrow \delta$, and for simple theoretical grammars use is often made of this notation, using the conventions about the use of upper-case letters for non-terminals and lower-case for terminals. For more realistic grammars, such as those used to specify programming languages, the most common way of specifying productions for many years was to use an alternative notation invented by Backus, and first called Backus normal form. Later it was realized that it was not, strictly speaking, a "normal form", and was renamed Backus-Naur form. Backus and Naur were largely responsible for the *Algol 60 report* (Naur 1960 and 1963), which was the first major attempt to specify the syntax of a programming language using this notation. Regardless of what the acronym really stands for, the notation is now universally known as **BNF**.

In classic BNF a non-terminal is usually given a descriptive name and this is enclosed in angle brackets to distinguish it from a terminal symbol. (Remember that non-terminals are used in the construction of sentences, although they do not actually appear in the final sentence.) In BNF, productions have the form

leftside → *definition*

Here "→" can be interpreted as "is defined as" or "produces" (in some texts the symbol `::=` is used in preference to →). In such productions, both *leftside* and *definition* will consist of strings concatenated from one or more terminals and non-terminals. In fact, in terms of our earlier notation

$$leftside \in (N \cup T)^+$$

and

$$definition \in (N \cup T)^*$$

For the moment we shall restrict our discussion to productions whose *leftside* consists of a single non-terminal only (so-called *context-free* grammars). Even in the more general case of so-called *context-sensitive* grammars *leftside* must contain at least one non-terminal, so that we must also have

$$leftside \cap N \neq \varnothing$$

Frequently we find several productions with a common *leftside*, and these are almost invariably expressed by listing their *definitions* as a set of one or more alternatives, separated by the vertical bar symbol "|".

5.6 Simple examples

It will help to put the abstruse theory of the last two sections in better perspective if we consider two simple examples in some depth. Our first example shows a grammar for a tiny subset of English. In full detail we have

G = { N , T , S , P }
N = { <sentence> , <qualified noun> , <noun> , <pronoun> , <verb> , <adjective> }
T = { the , man , girl , boy , lecturer , he , she , talks , listens , mystifies , tall , thin , sleepy }
S = <sentence>
P = {

<sentence>	→ the <qualified noun> <verb>	(1)
	| <pronoun> <verb>	(2)
<qualified noun>	→ <adjective> <noun>	(3)
<noun>	→ man | girl | boy | lecturer	(4, 5, 6, 7)
<pronoun>	→ he | she	(8, 9)
<verb>	→ talks | listens | mystifies	(10, 11, 12)
<adjective>	→ tall | thin | sleepy	(13, 14, 15)

}

The set of 15 productions is grouped into 6 rules, each specifying the options for one non-terminal. Between them they define the non-terminal <sentence> as consisting of either the terminal "the" followed by a <qualified noun> followed by a <verb>, or as a <pronoun> followed by a <verb>. A <qualified noun> is an <adjective> followed by a <noun>, and a <noun> is one of the terminal symbols "man" or "girl" or "boy" or "lecturer". A <pronoun> is either of the terminals "he" or "she", while a <verb> is either "talks" or "listens" or "mystifies". Here <sentence>, <qualified noun>, <noun>, <pronoun>, <noun> and <adjective> are non-terminals. These do not appear in any sentence of the language, which includes such majestic prose as

the thin lecturer mystifies
he talks
the sleepy boy listens

From a grammar, one non-terminal is singled out as the so-called **goal** or **start symbol**. If we want to *generate* an arbitrary sentence we start with the goal symbol as our initial sentential form, replace it by the right-hand side of one of the eligible productions for the goal symbol, and then successively replace each non-terminal in the resulting sentential form by the right-hand side of one of the eligible productions for that non-terminal, continuing this process until all non-terminals have been removed and only terminals remain.

Thus, for example, we could start with <sentence> and from this derive the sentential form

the <qualified noun> <verb>

In terms of the definitions of section 5.4 we say that <sentence> *directly produces* "the <qualified noun> <verb>". If we now apply production 3 (<qualified noun> → <adjective> <noun>) we get the sentential form

the <adjective> <noun> <verb>

In terms of the definitions of section 5.4, "the <qualified noun> <verb>" directly produces "the <adjective> <noun> <verb>", while <sentence> has produced this sentential form in a non-trivial way. If we now follow this by

applying production 14 (<adjective> → thin) we get the form

the thin <noun> <verb>

Application of production 10 (<verb> → talks) gets to the form

the thin <noun> talks

Finally, after applying production 6 (<noun> → boy) we get the sentence

the thin boy talks

The end result of all this is often represented by a tree, as in Figure 5.1, which shows a **phrase structure tree** or **parse tree** for our sentence. In this representation, the order in which the productions were used is not readily apparent, but it should now be clear why we speak of "terminals" and "non-terminals" in formal language theory - the leaves of such a tree are all terminals of the grammar; each interior node is labelled by a non-terminal.

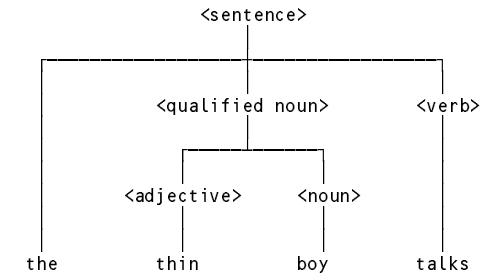


Figure 5.1 Parse tree for "the thin boy talks"

A moment's thought should reveal that there may be many possible derivation paths from the goal or start symbol to the final sentence, depending on the order in which the eligible productions are applied. It is convenient to be able to single out a particular derivation as being *the* derivation. This is generally called the **canonical derivation** and, although the choice is essentially arbitrary, the usual one is that where at each stage in the derivation the leftmost non-terminal is the next one that is to be replaced - this is called a **left canonical derivation**. (In a similar way we could define a **right canonical derivation**.)

Not only is it important to use grammars generatively in this way, but it is also important - perhaps more so - to be able to take a given sentence and determine whether it is a valid member of the language - that is, to see whether it could have been obtained from the goal symbol by a suitable choice of derivations. When mere recognition is accompanied by the determination of the underlying tree structure, we speak of **parsing** and we shall have a lot more to say about this in later chapters. For the moment note that there are several ways in which we can attempt to solve the problem. A fairly natural way is to start with the goal symbol and the sentence and, by reading the sentence from left to right, to try to deduce which series of productions must have been applied.

Let us try this on the sentence

the thin boy talks

If we start with the goal <sentence> we can derive a wide variety of sentences. Some of these will arise if we choose to continue by using production 1, some if we choose production 2. By reading no further than "the" in the given sentence we can be fairly confident that we should try production 1.

<sentence> → the <qualified noun> <verb>.

In a sense we now have a residual input string "thin boy talks" which somehow must match <qualified noun> <verb>. We could now choose to substitute for <verb> or for <qualified noun>. Again limiting ourselves to working from left to right, our residual sentential form <qualified noun> <verb> must next be transformed into <adjective> <noun> <verb> by applying production 3.

Similarly, we have now to match "thin boy talks" with a residual sentential form <adjective> <noun> <verb>. We could choose to substitute for any of <adjective>, <noun> or <verb>; if we read the input string from the left we see that by using production 14 we can reduce the problem of matching a residual input string "boy talks" to the

residual sentential form <noun> <verb>. And so it goes; we need not labour a very simple point here.

The parsing problem is not always as easily solved as this. It should be fairly obvious that the algorithms used to parse a sentence to see whether it can be derived from the goal symbol will be very different from algorithms that might be used to generate sentences (almost at random) starting from the start symbol. The methods used for successful parsing depend rather critically on the way in which the productions have been specified. For the moment we shall be content to examine a few sets of productions without worrying too much about how they were developed.

In BNF a production may define a non-terminal recursively, so that the same non-terminal may occur on both the left-hand and right-hand sides of the → sign. For example, if the production for <qualified noun> were changed to

<qualified noun> → <noun> | <adjective> <qualified noun> (3a, 3b)

this would define a <qualified noun> as either a <noun>, or an <adjective> followed by a <qualified noun> (which in turn may be a <noun>, or an <adjective> followed by a <qualified noun> and so on). In the final analysis a <qualified noun> would give rise to zero or more <adjective>s followed by a <noun>. Of course, a recursive definition can be useful only when there is some way of terminating it. The single production

<qualified noun> → <adjective> <qualified noun> (3b)

is effectively quite useless on its own - and it would be the alternative production

<qualified noun> → <noun> (3a)

that would provide the means for terminating the recursion.

As a second example, consider a simple grammar for describing a somewhat restricted set of algebraic expressions.

$G = \{ N , T , S , P \}$
$N = \{$ <goal> , <expression> , <term> , <factor> $\}$
$T = \{ a , b , c , - , * \}$
$S =$ <goal>
$P = \{$

<goal>	→ <expression>	(1)
<expression>	→ <term> | <expression> - <term>	(2, 3)
<term>	→ <factor> | <term> * <factor>	(4, 5)
<factor>	→ *a* | *b* | *c*	(6, 7, 8)

$\}$

It is left as an easy exercise to show that it is possible to derive the string *a - b * c* using these productions, and that the corresponding phrase structure tree takes the form shown in Figure 5.2.

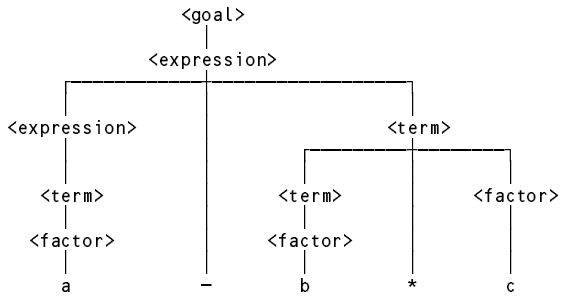


Figure 5.2 Parse tree for the expression a - b * c

A point that we wish to stress here is that the construction of this tree has, happily, reflected the relative precedence of the multiplication and subtraction operations - assuming, of course, that the symbols * and - are to have implied meanings of "multiply" and "subtract" respectively and that we extract the meaning of the whole expression by performing an post-order traversal of the tree. We should also point out that it is by no means

obvious at this stage how one goes about designing a set of productions that not only describe the syntax of a programming language, but also reflect some semantic meaning for the programs written in that language. Hopefully the reader can foresee that there will be a very decided advantage if such a choice *can* be made, and we shall have more to say about this in later sections.

5.7 Phrase structure and lexical structure

It should not take much to see that a set of productions for a real programming language grammar may conveniently be divided in two. We can distinguish a set of productions that specify **phrase structure** - the way in which the words or tokens of the language are combined to form components of programs that are known by concepts such as *Expression* or *Statement*. This specification can conveniently ignore such niceties as describing the form of the ignorable comments, white space and line breaks that separate tokens, and can even ignore the subtle technical differences involved in distinguishing a keyword like `case` from a word like `Case` that might be chosen as an identifier. While some tokens are easily specified by simple constant strings standing for themselves, we can also recognize that a further set of productions (or, in many cases, equivalent regular expressions) can be used to specify the **lexical structure** of whole classes of tokens - the way in which individual characters are combined to form identifiers, numbers or strings.

As we have already hinted, the recognition of tokens for a real programming language is usually done by a scanner (lexical analyzer) that returns these tokens to the parser (syntax analyzer) on demand. The productions involving only individual characters on their right-hand sides are thus the productions used by a subparser forming part of the lexical analyzer, while the others are productions used by the main parser in the syntax analyzer.

5.8 ε-productions

The alternatives for the right-hand side of a production usually consist of a string of one or more terminal and/or non-terminal symbols. At times it is useful to be able to derive an empty string - that is, one consisting of no symbols. This string is usually denoted by ε when it is necessary to reveal its presence explicitly. For example, the set of productions

⟨unsigned integer⟩ → ⟨digit⟩ ⟨rest of integer⟩
⟨rest of integer⟩ → ⟨digit⟩ ⟨rest of integer⟩ | ε
⟨digit⟩ → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

defines ⟨rest of integer⟩ as a sequence of zero or more ⟨digit⟩s, and hence ⟨unsigned integer⟩ is defined as a sequence of one or more ⟨digit⟩s. In terms of our earlier notation we should have

⟨rest of integer⟩ → ⟨digit⟩*

or

⟨unsigned integer⟩ → ⟨digit⟩$^+$

The production

⟨rest of integer⟩ → ε

is called a **null production**, or an ε-production, or sometimes a **lambda production** (from an alternative convention of using λ instead of ε for the null string). Applying a production of the form $N \rightarrow \varepsilon$ amounts to the erasure of the non-terminal N from a sentential form - for this reason such productions are sometimes called **erasures**. More generally, if for some string σ it is possible that

$$\sigma \Rightarrow^* \varepsilon$$

then we say that σ is **nullable**. In particular, a non-terminal N (usually one that possesses alternative definitions) is said to be nullable if it admits to at least one production whose right-hand side is nullable. In general, and to be useful, nullable non- terminals must have one or more alternative definitions that are not nullable.

5.9 Extensions to BNF

Various simple extensions are often employed with BNF notation for the sake of increased readability and for the elimination of recursion (which has a strange habit of confusing people brought up on iteration). Recursion is the standard method used in BNF to specify simple repetition, as for example

<unsigned integer> → <digit> | <digit> <unsigned integer>

(which uses right recursion) or

<unsigned integer> → <digit> | <unsigned integer> <digit>

(which uses left recursion).

Then we often find several productions used to denote alternatives which are very similar, for example

<integer> → <unsigned integer> | <sign> <unsigned integer>
<unsigned integer> → <digit> | <digit> <unsigned integer>
<sign> → + | -

using six productions (besides the omitted obvious ones for <digit>) to specify the form of an <integer>.

The extensions introduced to simplify these constructions lead to what is known as **EBNF** (Extended BNF). There have been many variations on this, most of them inspired by the metasymbols used for regular expressions. Thus we might find the use of the Kleene closure operators to denote repetition of a symbol zero or more times, and the use of parentheses ( ) to group items together.

Using these ideas we might define an integer by

<integer> → <sign> <unsigned integer>
<unsigned integer> → <digit> (<digit>)*
<sign> → + | - | ε

or even by

<integer> → (+ | - | ε) <digit> (<digit>)*

which is, of course, nothing other than a regular expression anyway. In fact, a language that can be expressed as a regular expression can always be defined by a single EBNF expression.

5.9.1 Wirth's EBNF notation

In defining Pascal and Modula-2, Wirth came up with one of these many variations on BNF which has now become rather widely used (Wirth 1977). Further metasymbols are used, so as to express more succinctly the many situations that otherwise require combinations of the Kleene closure operators and the ε string. In addition, further simplifications are introduced to facilitate the automatic processing of productions by parser generators such as we shall discuss in later chapters. In this notation for EBNF

`Non-terminals` are written as single words, as in *VarDeclaration* (rather than the <Var Declaration> of our previous notation);
`Terminals` are all written in quotes, as in `"void"` or `"while"` (rather than as themselves, as in BNF);
`|` is used, as before, to denote alternatives;
`( )` (parentheses) are used to denote grouping;
`[ ]` (brackets) are used to denote the optional appearance of a symbol or group of symbols;
`{ }` (braces) are used to denote optional repetition of a symbol or group of symbols;
`=` is used in place of the `::=` or → symbol;
`.` is used to denote the end of each production;
`(* *)` (or `/* .. */`) are used in some extensions to allow comments;
ε can be handled by using the [ ] notation;
`spaces` are essentially insignificant.

For example

```
Integer          = Sign UnsignedInteger .
UnsignedInteger  = digit { digit } .
Sign             = [ "+" | "-" ] .
digit            = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
```

The effect is that non-terminals are less "noisy" than in the earlier forms of BNF, while terminals are "noisier". Many grammars used to define programming language employ far more non-terminals than terminals, so this is often advantageous. Furthermore, since the terminals and non-terminals are textually easily distinguishable, it is usually adequate to give only the set of productions P when writing down a grammar and not the complete quadruple $\{ N , T , S , P \}$.

An excellent example of the use of this notation shows how to describe a set of EBNF productions in EBNF itself:

```
EBNF        = { Production } .
Production  = nonterminal "=" Expression "." .
Expression  = Term { "|" Term } .
Term        = Factor { Factor } .
Factor      =    nonterminal | terminal | "[" Expression "]"
               | "(" Expression ")" | "{" Expression "}" .
nonterminal = letter { letter } .
terminal    = "'" character { character } "'" | '"' character { character } '"' .
character   = (* implementation defined *)  .
letter      = (* implementation defined *)  .
```

Here we have chosen to spell *nonterminal* and *terminal* in lower-case throughout to emphasize that they are lexical non-terminals of a slightly different status from the others like *Production, Expression, Term* and *Factor*.

Strictly, BNF notation did not use meta-brackets. We shall refer to a set of productions that uses Wirth's notation but employs recursion and avoids using meta-brackets as a "BNF style" description.

5.9.2 Semantic overtones

Sometimes non-terminals are named and productions are developed to suggest semantic meaning to the reader. As we shall see in Chapter 9, this may lead more easily towards the possibility of extending or *attributing* the grammar to incorporate a formal semantic specification along with the syntactic specification. For example, in describing Modula-2, where expressions and identifiers fall into various categories at the static semantic level, we might find among a large set of productions

```
ConstDeclarations  = "CONST"
                       ConstIdentifier "=" ConstExpression ";"
                       { ConstIdentifier "=" ConstExpression ";" } .
ConstIdentifier    = identifier .
ConstExpression    = Expression .
```

5.9.3 Cocol

The reader will recall from Chapter 2 that compiler writers often make use of compiler generators to assist with the automated construction of parts of a compiler. Such tools usually take as input an augmented description of a grammar, one usually based on a variant of the EBNF notations we have just been discussing. We stress that far more is required to construct a compiler than a description of syntax - which is, essentially, all that EBNF can provide on its own. In later chapters we shall describe the use of a specific compiler generator, Coco/R (Rechenberg and Mössenböck 1989, Mössenböck 1990a,b). The name Coco/R is derived from "**Co**mpiler-**Co**mpiler/**R**ecursive descent". A variant of Wirth's EBNF known as Cocol/R is used to define the input to Coco/R and is the notation we shall prefer in the rest of this text (to avoid confusion between two very similar acronyms we shall simply refer to Cocol/R as Cocol). Cocol draws a clear distinction between lexical and phrase structure. It also provides for describing the character sets from which lexical tokens are constructed.

A simple example will show the main features of a Cocol description. The example describes a calculator that is

intended to process a sequence of simple four-function calculations involving decimal or hexadecimal whole numbers, for example `3 + 4 * 8 =` or `$3F / 7 + $1AF = .`

```
COMPILER Calculator

CHARACTERS
  digit      = "0123456789" .
  hexdigit   = digit + "ABCDEF" .

TOKENS
  decNumber  = digit { digit } .
  hexNumber  = "$" hexdigit { hexdigit } .

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Calculator = { Expression "=" } .
  Expression = Term { "+" Term  |  "-" Term } .
  Term       = Factor { "*" Factor |  "/" Factor } .
  Factor     = decNumber | hexNumber .
END Calculator.
```

The CHARACTERS section defines the sets of characters that can appear in decimal or hexadecimal number representations - note carefully that the right-hand side of each of these definitions is to be interpreted as defining a *set* of characters, and not a *string* of them. The TOKENS section describes the valid forms that decimal and hexadecimal numbers may take. Notice that we do not, at this stage, indicate how the values of these numbers are to be computed from the digits. The PRODUCTIONS section describes the phrase structure of the calculations themselves. Again there is no indication of how the results of the calculations are to be obtained.

At this stage it will probably come as no surprise to the reader to learn that Cocol, the language of the input to Coco/R, can itself be described by a grammar. Indeed, we may write this grammar in such a way that it can be processed by Coco/R itself. (Using Coco/R to process its own grammar is, of course, just another example of the bootstrapping techniques discussed in Chapter 3 - Coco/R is another good example of a self-compiling compiler.) A full description of Coco/R and Cocol appears later in this text. While the finer points of this may currently be beyond the reader's comprehension, the following simplified description will suffice to show the syntactic elements of most importance.

```
COMPILER Cocol

CHARACTERS
  letter       = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                 + "abcdefghijklmnopqrstuvwxyz" .
  digit        = "0123456789" .
  tab          = CHR(9) .
  lf           = CHR(10) .
  cr           = CHR(13) .
  backslash    = CHR(92) .
  control      = CHR(0) .. CHR(31) .
  graphic      = ANY - control .
  noQuote2     = graphic - '"' - backslash .
  noQuote1     = graphic - "'" - backslash .

TOKENS
  identifier   = letter { letter | digit } .
  string       =   '"' { noQuote2 | backslash graphic } '"'
                 | "'" { noQuote1 | backslash graphic } "'" .
  number       = digit { digit } .

COMMENTS FROM "/*" TO "*/"

IGNORE tab + cr + lf

PRODUCTIONS
  Cocol        = "COMPILER"  Goal
                    [ Characters ]
                    [ Tokens ]
                    { Comments }
                    { Ignorable }
                    Productions
                 "END" Goal "." .
  Goal         = identifier .

  Characters   = "CHARACTERS" { NamedCharSet } .
  NamedCharSet = SetIdent "=" CharacterSet "." .
```

```
  CharacterSet = SimpleSet { "+" SimpleSet | "-" SimpleSet } .
  SimpleSet    =   SetIdent | string | SingleChar [ ".." SingleChar ] | "ANY" .
  SingleChar   = "CHR" "(" number ")"  .
  SetIdent     = identifier .

  Tokens       = "TOKENS" { Token } .
  Token        = TokenIdent "=" TokenExpr "."  .
  TokenExpr    = TokenTerm { "|" TokenTerm } .
  TokenTerm    = TokenFactor { TokenFactor }
                 [ "CONTEXT" "(" TokenExpr ")" ] .
  TokenFactor  =   TokenSymbol | "(" TokenExpr ")"
                 | "[" TokenExpr "]" | "{" TokenExpr "}" .
  TokenSymbol  = SetIdent | string .
  TokenIdent   = identifier .

  Comments     = "COMMENTS" "FROM" TokenExpr "TO" TokenExpr [ "NESTED" ] .

  Ignorable    = "IGNORE" CharacterSet .

  Productions  = "PRODUCTIONS" { Production } .
  Production   = NonTerminal "=" Expression  "." .
  Expression   = Term { "|" Term } .
  Term         = [ Factor { Factor } ] .
  Factor       =   Symbol | "(" Expression ")"
                 | "[" Expression "]" | "{" Expression "}" .
  Symbol       = string | NonTerminal | TokenIdent .
  NonTerminal  = identifier .

END Cocol.
```

The following points are worth emphasizing.

- The productions in the TOKENS section specify identifiers, strings and numbers in the usual simple way.

- The first production (for Cocol) shows the overall form of a grammar description as consisting of five sections, the first four of which are all optional (although they are usually present in practice).

- The productions for *Characters* show how character sets may be given names (*SetIdents*) and values (in terms of *SimpleSets*).

- The production for *Ignorable* allows certain characters - typically line feeds and other unimportant characters - to be included in a set that will simply be ignored by the scanner when it searches for the next token.

- The productions for *Tokens* show how generic forms of token (terminal classes) may be named (*TokenIdents*) and defined by expressions in EBNF. Careful study of the semantic overtones of these productions will show that they are not self-embedding - that is, one token may not be defined in terms of another token, but only in terms of quoted strings or in terms of characters chosen from the named character sets defined in the *Characters* section. This amounts, in effect, to defining these tokens by means of regular expressions, even though the notation used is not the same as that given for regular expressions in section 5.3. As future examples will show, it is in fact unusual to give a name to a token that can be defined in terms of a unique quoted string (such as a keyword).

- The productions for *Productions* show how we define the phrase structure by naming *NonTerminals* and expressing their productions in EBNF. Notice that here we *are* allowed to have self-embedding and recursive productions. Although terminals may again be specified directly by strings, Cocol does not permit the names of character sets to appear as elements of the *Productions*.

- Although it is not specified by the grammar above, one non-terminal must have the same identifier name as the grammar itself to act as the goal symbol and, of course, all non-terminals must be defined properly - and may only appear on the left-hand side of exactly one *Production*.

- It is possible to write input in Cocol that is syntactically correct (in terms of the grammar above) but which cannot be fully processed by Coco/R because it does not satisfy other constraints. This topic will be discussed further in later sections.

- The production for *Term* permits a term to be "empty". This may seem a curiosity, but it does, for

example, allow a nullable production of the form

```
A = [ B C d ] .
```

to be expressed equivalently as

```
A = B C d  |  .
```

This may seem a little strange; it turns out in some applications to be extremely useful, as we shall see.

We stress again that a Cocol description really defines *two* grammars. One is the grammar that defines the form of tokens that the lexical analyzer must be able to recognise in a generic sense (TOKENS) and the other defines the non-terminals for the higher level phrase structure grammar used by the syntax analyzer (PRODUCTIONS). It is not always obvious to beginners where to draw the distinction between these. The following, in one sense, may appear to be equivalent:

```
COMPILER Sample   /* one */

CHARACTERS
  letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" .

TOKENS
  ident = letter { letter } .

PRODUCTIONS
  Sample = "BEGIN" ident ":=" ident "END" .

END Sample .
```

```
COMPILER Sample   /* two */

CHARACTERS
  letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" .

TOKENS
  Letter = letter .

PRODUCTIONS
  Sample = "BEGIN" Ident ":=" Ident "END" .
  Ident  = Letter { Letter } .

END Sample .
```

```
COMPILER Sample   /* three */

CHARACTERS
  letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" .

TOKENS
  ident = letter { letter } .
  begin = "BEGIN" .
  end   = "END" .
  becomes = ":=" .

PRODUCTIONS
  Sample = begin ident becomes ident end .

END Sample .
```

Actually they are not quite the same. Since Coco/R generally ignores spaces (other than in strings), the second one would accept either of the inputs

```
BEGIN  A C E := S P A D E  END
BEGIN  ACE := SPADE  END
```

while the first would accept only the input

```
BEGIN  ACE := SPADE  END
```

As a general rule, one should declare under TOKENS any class of symbol that has to be recognized as a contiguous string of characters, and of which there may be several instances (this includes entities like identifiers, numbers and strings), as well as special character terminals (like EOL) that cannot be graphically represented as quoted characters. Reserved keywords and symbols like ":=" are better introduced by literal strings embedded in the PRODUCTIONS section. Thus the first grammar above is probably optimal so far as using Cocol is concerned.

6 DEVELOPMENT AND CLASSIFICATION OF GRAMMARS

In this chapter we shall explore the underlying ideas behind grammars further, identify some potential problem areas in designing grammars, and examine the ways in which grammars can be classified. Designing a grammar to describe the syntax of a programming language is not merely an interesting academic exercise. The effort is, in practice, usually made so as to be able to aid the development of a translator for the language (and, of course, so that programmers who use the language may have a reference to consult when All Else Fails and they have to Read The Instructions). Our study thus serves as a prelude to the next chapter, where we shall address the important problem of parsing rather more systematically than we have done previously.

6.1 Equivalent grammars

As we shall see, not all grammars are suitable as the starting point for developing practical parsing algorithms and an important part of compiler writing is concerned with the ability to find **equivalent grammars**. Two grammars are said to be equivalent if they describe the same language - that is, can generate exactly the same set of sentences (not necessarily using the same set of sentential forms or parse trees).

In general we may be able to find several equivalent grammars for any language. A distinct problem in this regard is a tendency to introduce far too few non-terminals or, alternatively, far too many. It should not have escaped attention that the names chosen for non-terminals usually convey some semantic implication to the reader and the way in which productions are written (that is, the way in which the grammar is factorized) often serves to emphasize this still further. Choosing too few non-terminals means that semantic implications are very awkward to discern at all. Choosing too many means that one runs the risk of ambiguity and of hiding the semantic implications in a mass of hard-to-follow alternatives.

It may be of some interest to view approximate counts of the numbers of non-terminals and productions that have been used in the definition of a few languages. It would, however, be dangerous to conclude that the inherent complexity of a language is directly proportional to these quantities.

Language		Non-terminals	Productions
Pascal	(Jensen + Wirth report)	110	180
Pascal	(ISO standard)	160	300
C		75	220
C++		110	270
ADA		180	320
Modula-2	(Wirth)	74	135
Modula-2	(ISO standard)	225	306
Parva	(Chapter 14)	34	73
C#Minor	(Chapter 15)	57	112

6.2 Case study - equivalent grammars for describing expressions

One problem with the grammars found in textbooks is that, like many complete programs found in textbooks, their final presentation often hides the thought which has gone into their development. To try to redress the balance, let us look at a typical language construct - arithmetic expressions - and explore several grammars which seem to define them.

Consider the following EBNF descriptions of simple algebraic expressions. The first set (G1) is left recursive, while the second set (G2) is right recursive.

```
(G1)    Expression  = Term | Expression "-" Term .        (1, 2)
        Term        = Factor | Term "*" Factor .          (3, 4)
        Factor      = "a" | "b" | "c" .                   (5, 6, 7)

(G2)    Expression  = Term | Term "-" Expression .        (1 ,2)
        Term        = Factor | Factor "*" Term .          (3, 4)
        Factor      = "a" | "b" | "c" .                   (5, 6, 7)
```

Either of these grammars can be used to derive the string *a - b * c*, and we show the corresponding phrase

structure or parse trees in Figure 6.1 below.

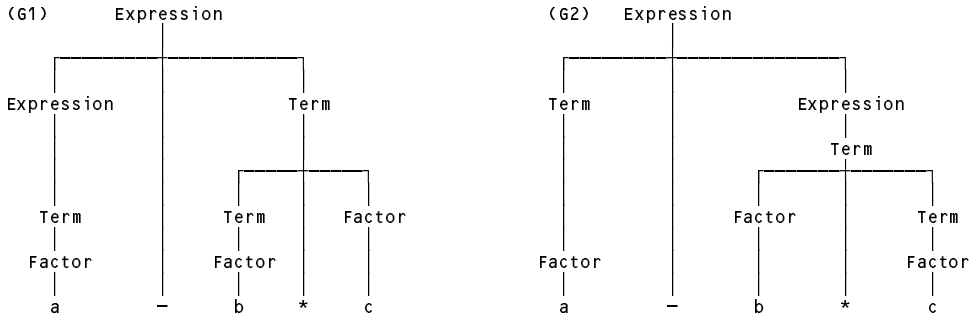


Figure 6.1 Parse trees for the expression a - b * c arising from two grammars

We have already commented that it is frequently the case that the semantic structure of a sentence is reflected in its syntactic structure and that this is a very useful property for programming language specification. The terminals - and * fairly obviously have the "meaning" of subtraction and multiplication. We can reflect this by drawing the abstract syntax tree (AST) equivalents of the above diagrams - ones constructed essentially by eliding out the names of the non-terminals, as depicted in Figure 6.2. In this case both grammars lead to the same AST.

Figure 6.2 Abstract syntax trees for the expression a - b * c

The appropriate meaning can then be extracted from this AST by performing a post-order or left-right-node (LRN) tree walk.

While the two sets of productions lead to the same sentences, the left recursive set of productions corresponds to the usual implied semantics of "left to right" associativity of the operators - and *, but the right recursive set has the awkward implied semantics of "right to left" associativity. We can see this by considering the parse trees for each grammar for the string *a - b - c*, depicted in Figure 6.3.

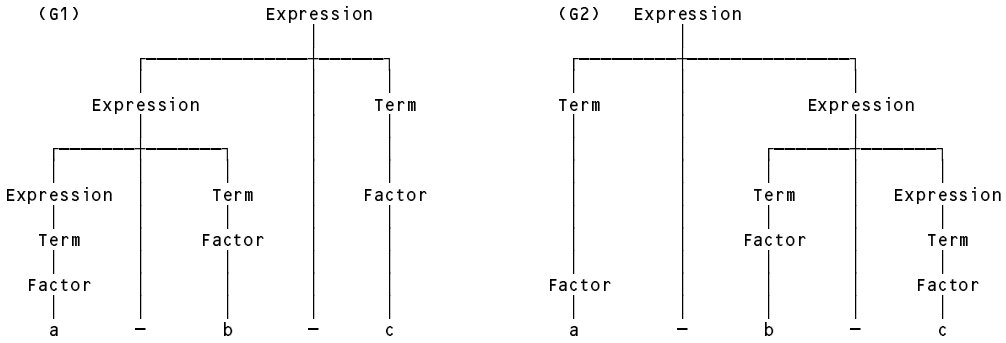


Figure 6.3 Parse trees for the expression a - b - c arising from two grammars

Another attempt at writing a grammar for this language is of interest:

(G3)	*Expression*	= *Term* | *Term* "*" *Expression* .	(1, 2)
	Term	= *Factor* | *Factor* "-" *Term* .	(3, 4)
	Factor	= "*a*" | "*b*" | "*c*" .	(5, 6, 7)

Here we have an unfortunate situation. Not only is the associativity of the operators wrong, but the relative precedence of multiplication and subtraction has also been inverted from the norm. This can be seen from the parse tree for the expression *a - b * c* shown in Figure 6.4.

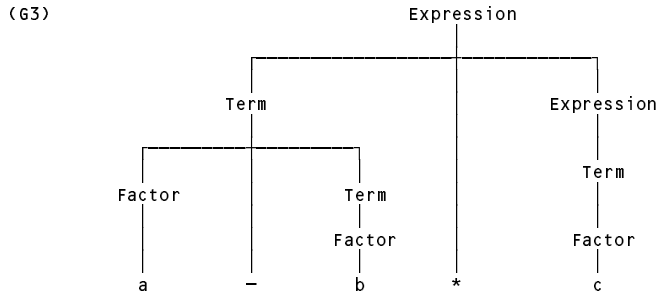


Figure 6.4 Parse tree for the expression a - b * c arising from grammar G3

The reason for the incorrect precedence is not hard to see. Reflection on this, and on the way in which operator associativity is reflected in the various parse trees, should suggest to the reader how best to go about constructing a grammar for expressions involving binary (infix) operators in general. We have structured the productions in a hierarchical fashion, and it appears that the operators of higher precedence need to appear lower down in this hierarchy. Furthermore, if left associativity is required we use left recursive productions - if right associativity is required we use right recursive productions.

One further example may be valuable. Suppose we wish to extend the grammar to incorporate an "exponentiation" operator, denoted by ↑, to allow for expressions like $a^b - b^c$ to be written as $a\uparrow b - b\uparrow c$. Exponentiation is conventionally deemed to be of higher precedence than multiplication, so we need an extra level deep in our hierarchy. Furthermore, exponentiation is conventionally right associative, so that $a\uparrow b\uparrow c$ is interpreted as equivalent to $a\uparrow(b\uparrow c)$ and not to $(a\uparrow b)\uparrow c$. With these observations a suitable grammar seems to be

(G4)	*Expression*	= *Term* | *Expression* "-" *Term* .	(1, 2)
	Term	= *Factor* | *Term* "*" *Factor* .	(3, 4)
	Factor	= *Primary* | *Primary* "↑" *Factor* .	(5, 6)
	Primary	= "*a*" | "*b*" | "*c*" .	(7, 8, 9)

and the reader is invited to draw a few parse trees to verify this claim.

If we use the EBNF metasymbols it is possible to write grammars without using recursive productions. One such grammar, incorporating exponentiation, multiplication and subtraction is

(G5)	*Expression*	= *Term* { "-" *Term* } .	(1)
	Term	= *Factor* { "*" *Factor* } .	(2)
	Factor	= *Primary* ["↑" *Factor*] .	(3)
	Primary	= "*a*" | "*b*" | "*c*" .	(4, 5, 6)

It may not be quite so apparent how to use this grammar to construct a parse tree, or that it correctly reflects the left associativity of multiplication and subtraction and the right associativity of exponentiation, but it turns out to have these properties. The trick is to stretch the imagination a little and interpret each production as a prescription for applying its operators as soon as the values of the operands on either side of these operators are known. So, for example, evaluation of $a\uparrow b\uparrow c$ requires the leading *Primary* (*a*) to be raised to the value of a *Factor* (corresponding to production 3). But we cannot do this until we know the value of this *Factor* - which we get by noting that it must be the value of a further *Primary* (*b*) raised to a *Factor* (which in this last case is simply the final *Primary* (*c*)).

6.3 Some simple restrictions on grammars

Had he looked at our grammars, George Orwell (1984) might have been tempted to declare that while they might be equal, some are more equal than others. Even with rather limited experience we have seen that some grammars display characteristics that will call for great care if we are to use them as the basis of compiler development. Apart from the sort of considerations already illustrated, there are several standard restrictions which are called for by different parsing techniques, among which are some fairly obvious ones.

6.3.1 Useless productions and reduced grammars

For a grammar to be of practical value, especially in the automatic construction of parsers and compilers, it

should not contain superfluous rules that cannot be used in parsing a sentence. Detection of useless productions may seem a waste of time, but it may also point to a clerical error (perhaps an omission) in writing the productions. An example of a grammar with useless productions is

$S \rightarrow A$ (1)
$A \rightarrow aA$ (2)
$A \rightarrow D$ (3)
$A \rightarrow B$ (4)
$D \rightarrow aD$ (5)
$B \rightarrow a$ (6)
$C \rightarrow aa$ (7)

The useful productions are (1), (2), (4) and (6). Production (7) ($C \rightarrow aa$) is useless, because C is **non-reachable** or **non-derivable** - there is no way of introducing C into a sentential form (that is, $S \not\Rightarrow^* \alpha C \beta$ for any α, β). Productions (3) and (5) are useless, because D is **non-terminating** - if D appears in a sentential form then this cannot generate a terminal string (that is, $D \not\Rightarrow^* \alpha$ for any $\alpha \in T^*$).

A **reduced grammar** is one that does not contain superfluous rules of these two types (non-terminals that can never be reached from the start symbol, and non-terminals that cannot produce terminal strings).

More formally, a context-free grammar is said to be reduced if, for each non-terminal A, we can write

$$S \Rightarrow^* \alpha A \beta$$

for some strings α and β, and where

$$A \Rightarrow^* \gamma$$

for some $\gamma \in T^*$.

In fact, non-terminals that cannot be reached in any derivation from the start symbol are sometimes added so as to assist in describing the language - an example might be to write, for C# or Java

```
Comment       = "/*" CommentString "*/" .
CommentString = character | CommentString character .
```

6.3.2 ε-free grammars

Intuitively we might expect that detecting the presence of "nothing" would be a little awkward, and for this reason certain compiling techniques require that a grammar should contain no ε-productions (those which generate the null string). Such a grammar is referred to as an ε-free grammar.

ε-productions are usually used in BNF as a way of terminating recursion and are often easily removed. For example, the productions

```
Integer       = digit RestOfInteger .
RestOfInteger = digit RestOfInteger | ε .
digit         = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
```

can be replaced by the ε-free equivalent

```
Integer       = digit | Integer digit .
digit         = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
```

Such replacement may not always be quite so easy - the reader might like to look at an attempt to write a Parva description (at the end of this chapter) which uses ε-productions to express *StatementSequence, Initializer, Subscript, ExpTail* and *ArrayFlag*, and try to eliminate them.

6.3.3 Cycle-free grammars

A production in which the right-hand side consists of a single non-terminal

$$A \rightarrow B \qquad (\text{where } A, B \in N)$$

is termed a **single production**. Fairly obviously, a single production of the form

$$A \rightarrow A$$

serves no useful purpose, and should never be present. It could be argued that it causes no harm, for it presumably would be an alternative which was never used (so being useless, in a sense not quite that discussed above). A less obvious example is provided by the set of productions

$$\begin{aligned} A &\rightarrow B \\ B &\rightarrow C \\ C &\rightarrow A \end{aligned}$$

Not only is this useless in this new sense, it is also highly undesirable from the point of obtaining a unique parse, and so all parsing techniques require a grammar to be **cycle-free** - it should not permit a derivation of the form

$$A \Rightarrow^{+} A$$

6.4 Ambiguous grammars

An important property which one looks for in programming languages is that every sentence that can be generated by the language should have a unique parse tree, or, equivalently, a unique left (or right) canonical parse. If at least one sentence that can be generated by a grammar has two or more parse trees then the grammar is said to be *ambiguous*. To demonstrate ambiguity all tht is needed is to find *one* such sentence An example of ambiguity is provided by another attempt at writing a grammar for simple algebraic expressions - this time apparently simpler than before -

(G9)	*Expression*	=	*Expression* "-" *Expression*	(1)
			| *Expression* "*" *Expression*	(2)
			| *Factor* .	(3)
	Factor	=	"*a*" | "*b*" | "*c*" .	(4, 5, 6)

With this grammar the sentence *a - b * c* has two distinct parse trees and two canonical derivations. We refer to the numbers to show the derivation steps.

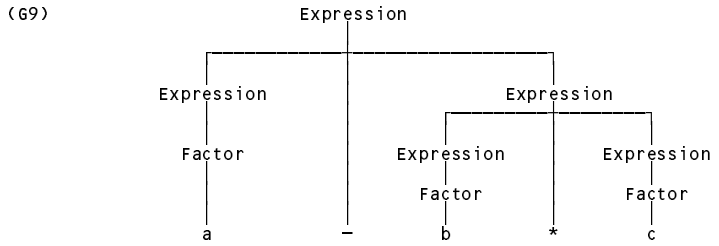


Figure 6.5 One parse tree for the expression a - b * c using grammar G9

The parse tree shown in Figure 6.5 corresponds to the derivation

Goal	→ *Expression*	
	→ *Expression - Expression*	(1)
	→ *Factor - Expression*	(3)
	→ *a - Expression*	(4)
	→ *a - Expression * Expression*	(2)
	→ *a - Factor * Expression*	(3)
	→ *a - b * Expression*	(5)
	→ *a - b * Factor*	(3)
	→ *a - b * c*	(6)

while the second derivation

Goal → *Expression*
→ *Expression * Expression* (2)
→ *Expression - Expression * Expression* (1)
→ *Factor - Expression * Expression* (3)
→ *a - Expression * Expression* (4)
→ *a - Factor * Expression* (3)
→ *a - b * Expression* (5)
→ *a - b * Factor* (3)
→ *a - b * c* (6)

corresponds to the parse tree depicted in Figure 6.6.

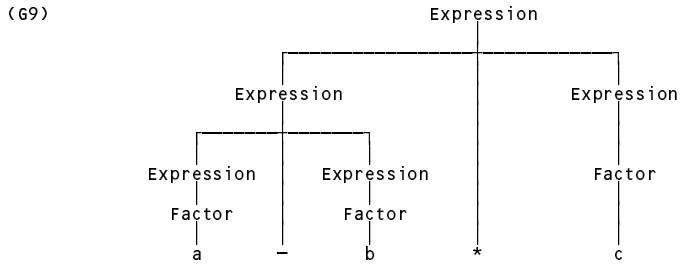


Figure 6.6 Another parse tree for the expression a - b * c using grammar G9

If the only use for grammars was to determine whether a string belonged to the language, ambiguity might be of little consequence. However, if the meaning of a program is to be tied to its syntactic structure, then ambiguity must be avoided. In the example above, the two trees correspond to two different evaluation sequences for the operators * and - . In the first case the meaning would be the usual mathematical one, namely *a* - (*b* * *c*), but in the second case the meaning would effectively be (*a* - *b*) * *c* .

We have already seen various examples of unambiguous grammars for this language in an earlier section and in this case, fortunately, ambiguity is quite easily avoided. In many cases where a well-meaning attempt to describe a language results in an ambiguous grammar one can find an unambiguous equivalent grammar, possibly with some difficulty. In the worst situations it may not be possible to find an unambiguous grammar for a language however hard one tries. In this case the language is said to be **inherently ambiguous**.

A rather more interesting ambiguous grammar for expressions is the following variation on grammar G10. Can you demonstrate that it is indeed ambiguous, by finding at least one expression that can be parsed in two different ways?

(G10) *Expression* = *Term* { "-" *Term* } . (1)
Term = *Factor* { "*" *Factor* } . (2)
Factor = *Primary* ["↑" *Expression*] . (3)
Primary = "*a*" | "*b*" | "*c*" . (4, 5, 6)

The most famous example of an ambiguous grammar probably derives from the IF ... THEN ... ELSE statement in simple Algol-like languages, where it leads to what is called the *dangling else* problem. We may demonstrate this by considering the following simple grammar incorporating this construct.

Program = *Statement* .
Statement = *Assignment* | *IfStatement* .
Assignment = *Variable* ":=" *Expression* .
Expression = *Variable* .
Variable = "*i*" | "*j*" | "*k*" | "*a*" | "*b*" | "*c*" .
IfStatement = "IF" *Condition* "THEN" *Statement*
| "IF" *Condition* "THEN" *Statement* "ELSE" *Statement* .
Condition = *Expression* "=" *Expression*
| *Expression* "≠" *Expression* .

In this grammar the string

```
IF i = j  THEN IF i = k THEN a := b ELSE a := c
```

has two possible parse trees. The reader is invited to draw these out as an exercise. The essential point is that we can parse the string to correspond either to

```
IF i = j  THEN ( IF i = k THEN a := b ELSE a := c )
          ELSE ( nothing )
```

or to

```
IF i = j  THEN ( IF i = k THEN a := b ELSE nothing )
          ELSE ( a := c )
```

Faced with a situation like this, a language designer or compiler writer must try to resolve it. One approach is to introduce an *ad hoc* **disambiguating rule** outside of the grammar itself. For example, one might disallow certain statement forms from following the `THEN` of an *IfStatement*, as was done in Algol (Naur 1963). In Pascal, C#, Java and C++, as is hopefully well known, an `ELSE` is deemed to be attached to the most recent `IF`, and the problem is solved that way.

If one has the freedom to choose the syntax of the language (at the design stage), a clean solution is simply to introduce closing symbols like `ENDIF` and `ELSIF`, as was done in Ada and Modula-2.

Lastly, one might persevere with trying to find an unambiguous grammar. It *is* possible to write productions that *are* unambiguous (Aho, Sethi and Ullman 1986):

```
Statement   =  Matched | Unmatched .
Matched     =  "IF" Condition "THEN" Matched "ELSE" Matched
             | Assignment .
Unmatched   =  "IF" Condition "THEN" Statement
             | "IF" Condition "THEN" Matched "ELSE" Unmatched .
```

In the general case, unfortunately, no algorithm exists (or can exist) that can take an arbitrary grammar and determine with certainty and in a finite amount of time whether it is ambiguous or not. All that one can do is to develop fairly simple but non-trivial conditions which, if satisfied by a grammar, assure one that it is unambiguous. Fortunately, other than in this classic example, ambiguity does not seem to be a problem in practical programming languages.

6.5 The Chomsky hierarchy

Until now all our practical examples of productions have had a single non-terminal on the left-hand side, although grammars may be more general than that. Based on pioneering work by a linguist (Chomsky 1956, 1959), computer scientists now recognize four classes of grammar. The classification depends on the format of the productions, and can be summarized as follows.

6.5.1 Type 3 Grammars (regular, either right-linear or left-linear)

The most highly constrained form of grammar is known as a type 3 or **regular** grammar. Here productions have the form

$$\alpha \rightarrow \beta \qquad \text{with } |\alpha| \leq |\beta| \; ; \; \alpha \in N \; ; \; \beta \in (N \cup T)^+$$

(where $|\alpha|$ denotes the length of α), but with the further constraint that β is very tightly constrained to take one or other of two forms (but not both in a single grammar). We speak of a **right-linear regular grammar** if the left-hand side of every production is a single non-terminal and the right-hand side consists of one terminal symbol, optionally followed by a single non-terminal, so that productions have the form

$$A \rightarrow a \text{ or } A \rightarrow aB \qquad \text{with } a \in T \; ; \; A, B \in N$$

We speak of a **left-linear regular grammar** if the right-hand side of every production consists of one terminal optionally preceded by a single non-terminal, so that productions have the form

$$A \rightarrow a \text{ or } A \rightarrow Ba \qquad \text{with } a \in T \ ; \ A, B \in N$$

Strictly, the constraint that $| \alpha | \leq | \beta |$ means that ε-productions are forbidden. This is often overlooked, or else a concession is made to allow the grammar to contain at most one ε-production for the goal symbol.

A simple example of such a grammar is one for describing binary integers:

$$BinaryInteger = \text{"0"}\ BinaryInteger \mid \text{"1"}\ BinaryInteger \mid \text{"0"} \mid \text{"1"} \ .$$

A more interesting example of a regular grammar is the following

(G11)	$A \rightarrow aA \mid aB$	(1, 2)
	$B \rightarrow bC$	(3)
	$C \rightarrow aC \mid a$	(4, 5)

which generates the language

$$L(G11) = \{ a^n b a^m \mid n, m \geq 1 \}$$

Regular grammars are rather restrictive - local features of programming languages like the definitions of integer numbers and identifiers can be described by them, but not much more. Any language that can be described by a regular grammar can be described by a single regular expression (and *vice versa*), which makes them of theoretical interest from that viewpoint as well. Regular grammars have the property that their sentences can be efficiently parsed by so-called **finite state automata**. Automata theory is an important part of computer science, but apart from a brief description of a finite state automaton in section 8.6, a full treatment of automata is beyond the scope of this text.

6.5.2 Type 2 Grammars (context-free)

Slightly relaxing the constraints on the form that the right-hand sides of productions may assume leads to the class known as type 2 or **context-free** grammars. A grammar is context-free if the left-hand side of every production consists of a single non-terminal, and the right-hand side consists of a non-empty sequence of terminals and non-terminals, so that productions have the form

$$\alpha \rightarrow \beta \qquad \text{with } | \alpha | \leq | \beta | \ ; \ \alpha \in N \ ; \ \beta \in (N \cup T)^+$$

that is

$$A \rightarrow \beta \qquad \text{with } A \in N \ ; \ \beta \in (N \cup T)^+$$

Strictly, as before, no ε-productions should be allowed, but this is often relaxed to allow $\beta \in (N \cup T)^*$. The reason for calling such productions context-free is easily understood - if A occurs in any sentential form, say $\gamma A \delta$, then we may effect a derivation step $\gamma A \delta \Rightarrow \gamma \beta \delta$ without any regard for the particular context (prefix or suffix) in which A occurs. Type 3 grammars are also context-free in this sense, but if all productions in a grammar are of the regular form the grammar is usually said to be regular rather than context-free. The presence of one or more productions with a more general right-hand side means that the grammar can *only* be described as context-free.

General context-free grammars have the property that their sentences can be parsed by so-called **push-down automata** which are, of necessity, more complex than the finite state automata that suffice for strictly regular languages. They are also considerably more expressive and more powerful than regular grammars. For example, the language *L(G11)* of section 6.5.1 can be described more succinctly by the grammar with productions

$$A \rightarrow BbB$$
$$B \rightarrow aB \mid a$$

while the set of productions

$$A \rightarrow aBa$$
$$B \rightarrow aBa \mid b$$

is all that is needed to describe the balanced language

$$L(G) \;=\; \{\, a^n\, b\, a^n \mid n \geq 1 \,\}$$

which cannot be described by a regular grammar at all.

6.5.3 Type 1 Grammars (context-sensitive)

If we relax the restriction on type 2 grammars to require only that the number of symbols in the string on the left of any production is less than or equal to the number of symbols on the right-hand side of that production we get the subset of grammars known as type 1 or **context-sensitive**. As before, a context-free grammar also satisfies this constraint. What is really being suggested is that the left-hand side of some productions might now contain more than one symbol, and the presence of one or more productions of this form will cause the grammar to be termed context-sensitive rather than context-free.

Productions in type 1 grammars can be described to be of the general form

$$\alpha \rightarrow \beta \qquad \text{with } \mid \alpha \mid \leq \mid \beta \mid ; \quad \alpha \in (N \cup T)^* N (N \cup T)^* \; ; \quad \beta \in (N \cup T)^+$$

where $\mid \alpha \mid > 1$ for at least one α. Once again, it follows that the null string would not be allowed as a right-hand side of any production. However, this is sometimes overlooked, as ε-productions are often needed to terminate recursive definitions. Indeed, the exact definition of "context-sensitive" differs from author to author. Another way of describing such grammars is to say that at least one production must be of the form described by

$$\alpha A \beta \rightarrow \alpha \gamma \beta \qquad \text{with } \alpha, \beta \in (N \cup T)^* ; \; A \in N ; \; \gamma \in (N \cup T)^+$$

although examples are often given where productions are of a more general form, namely

$$\alpha A \beta \rightarrow \gamma \qquad \text{with } \alpha, \beta \in (N \cup T)^* ; \; A \in N ; \; \gamma \in (N \cup T)^+$$

(It can be shown that the two definitions are equivalent.) Here we can see the meaning of context-sensitive more clearly - A can be replaced by γ when A is found in the context of (that is, surrounded by) α and β.

A much quoted simple example of such a grammar is as follows (Tremblay and Sorenson 1985).

(G12)	$A \rightarrow aABC \mid abC$	(1, 2)
	$CB \rightarrow BC$	(3)
	$bB \rightarrow bb$	(4)
	$bC \rightarrow bc$	(5)
	$cC \rightarrow cc$	(6)

Let us derive a sentence using this grammar. A is the start string - let us choose to apply production (1)

$$A \rightarrow aABC$$

and then in this new string choose another production for A, namely (2), to derive

$$A \rightarrow a\, abC\, BC$$

and follow this by the use of (3). (We could also have chosen (5) at this point.)

$$A \rightarrow aab\, BC\, C$$

We follow this by using (4) to derive

$$A \rightarrow aa\, bb\, CC$$

followed by the use of (5) to get

$$A \rightarrow aab\, bc\, C$$

followed finally by the use of (6) to give

$$A \rightarrow aabbcc$$

This example may appear to be little more than a curiosity. As it happens, the language described by the grammar is deceptively simple

$$L(G) = \{ a^n b^n c^n \mid n \geq 1 \}$$

although this is not immediately apparent from a study of the productions themselves - a drawback of context-sensitive grammars in general. Simple though it appears, this language cannot be described by a context-free language either.

A further difficulty with trying to use this grammar is that it is possible to derive a sentential form to which no further productions can be applied. For example, after deriving the sentential form

$$aabCBC$$

if we were to apply (5) instead of (3) we would obtain

$$aabcBC$$

but no further production can be applied to this string. The consequence of such a failure to obtain a terminal string is simply that we must try other possibilities until we either find those that yield terminal strings or exhaust all possibilities. The consequences for the inverse problem, namely parsing, are that we may have to resort to considerable *backtracking* to decide whether a string is a sentence in the language.

Another context-sensitive grammar that is slightly more than a curiosity is the following (Pittman and Peters 1992).

(G13)	$S \rightarrow cDC$	(1)
	$D \rightarrow aDA \mid bDB \mid cc$	(2, 3, 4)
	$C \rightarrow c$	(5)
	$Aa \rightarrow aA$	(6)
	$Ab \rightarrow bA$	(7)
	$Ba \rightarrow aB$	(8)
	$Bb \rightarrow bB$	(9)
	$AC \rightarrow aC$	(10)
	$BC \rightarrow bC$	(11)

This generates sentences of the form $cVccVc$ where V is any string composed of a and b. The grammar is of interest because it captures the idea of declaring a variable named V (in the first appearance of V) before it is referenced (in the second appearance of V). It is not possible to find a context-free grammar to enforce this constraint.

Theoretically, the sentences of a language described by a context-sensitive language can be parsed using so-called **linear-bounded automata**, but these are beyond the scope of this text and are of little practical interest.

It is, however, worth a brief digression to consider ambiguity as it relates to context-sensitivity. The ambiguities that we considered earlier in this chapter arose from a poor choice of context-free syntax. Similar examples can be found in spoken or written English. For example, it is not clear from the sentence, "I greeted the student with a smile", which of the parties involved was smiling, even though each of the individual words has a fairly unambiguous meaning. In other cases ambiguities may arise from situations where it is not immediately clear (without further clues being given) which of several meanings should be attached to a word. We can think of this as another manifestation of the concept of context-sensitivity. Spoken and written language is full of such examples, which the average person parses with ease, discerning any necessary clues from within a particular cultural context or idiom. For example, the sentences

Time flies like an arrow

and

Fruit flies like a banana

in one sense have identical construction

Noun Verb Adverbial phrase

but, unless we were preoccupied with aerodynamics, in listening to them we would probably subconsciously parse the second along the lines of

Adjective Noun Verb Noun phrase

Examples like this can be found in programming languages too. In Fortran a statement of the form

A = B(J)

(when taken out of context) could imply a reference either to the Jth element of array B, or to the evaluation of a function B with integer argument J. Mathematically there is little difference - an array can be thought of as a mapping, just as a function can, although programmers may not often think that way.

6.5.4 Type 0 Grammars (unrestricted)

An **unrestricted** grammar is one in which there are virtually no restrictions on the form of any of the productions, which have the general form

$\alpha \rightarrow \beta$ with $\alpha \in (N \cup T)^+$, $\beta \in (N \cup T)^*$

(thus the only restriction is that there must be at least one symbol on the left-hand side of each production). As before, the above relation does not at first seem prescriptive - to qualify as being of type 0 rather than one of the more restricted types the grammar must contain at least one production $\alpha \rightarrow \beta$ with $|\alpha| > |\beta|$. Such a production can be used to "erase" symbols - for example, $aAB \rightarrow aB$ erases A from the context aAB. Although the sentences of languages describable only in this way can be parsed by **Turing machines**, this type is so rare in computer applications that we shall consider it no further here. Practical grammars need to be far more restricted if we are to base translators on them.

6.5.5 The relationship between grammar type and language type

It should be clear from the preceding discussions that type 3 grammars are a subset of type 2 grammars, which themselves form a subset of type 1 grammars, which in turn form a subset of type 0 grammars (see Figure 6.7).

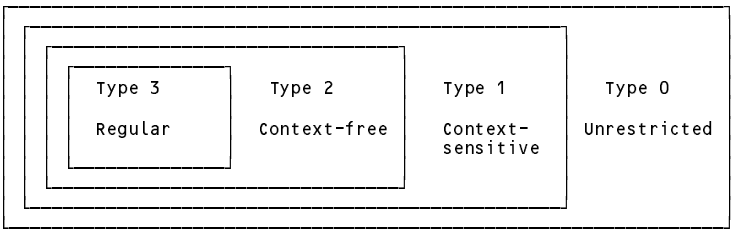


Figure 6.7 The Chomsky hierarchy of grammars

A language $L(G)$ is said to be of type k if it *can* be generated by a type k grammar. Thus, for example, a language is said to be context-free if a context-free grammar can be used to define it. Note that if a non context-free definition is given for a particular language, it does not necessarily imply that the language is not context-free - there may be an alternative (possibly yet-to-be-discovered) context-free grammar that describes it. Similarly, the fact that a language can, for example, most easily be described by a context-free grammar does not necessarily preclude our being able to find an equivalent regular grammar.

Grammars for modern programming languages are invariably defined by context-free grammars that capture all but a few context-sensitive features. These are then handled with a few extra *ad hoc* rules and by using so-called **attribute grammars**, rather than by taking on the formidable task of finding suitable context-sensitive grammars. Among these features are the following:

- the declaration of a variable must precede its use;
- the number of formal and actual parameters in a procedure call must be the same;
- the number of index expressions or fields in a variable designator must match the number specified in its declaration.

Further reading

The material in this chapter is standard, and good treatments of it can be found in many books. The keen reader might do well to look at the alternative presentation in the books by Gough (1988), Watson (1989), Watt (1991), Pittman and Peters (1992), Aho, Sethi and Ullman (1986), Louden (1997) or Tremblay and Sorenson (1985). An interesting discussion of Chomsky's work and its implications can be found in the book by Parsons (1992).

An attempt to use context-sensitive productions in an actual computer language was made by Lee (1972), who gave such productions for the PRINT statement in BASIC.

Addendum - a "BNF" style set of productions for Parva

The reader might like to study the following attempt to write a description of Parva and to compare it with the description in section 7.4. Are the two grammars equivalent? Could you find a ε-free grammar that uses BNF style productions (see section 6.3.2)? Where does this grammar use left-recursion, where does it use right-recursion and could this be altered? Do you think it is a good description of the syntax of Parva? Why (or why not)?

```
PRODUCTIONS
  Parva            = "void" identifier "(" ")" Block .
  Block            = "{" StatementSequence "}" .
  StatementSequence = StatementSequence Statement | .
  Statement        =   Block | ConstDeclarations | VarDeclarations | Assignment
                     | IfStatement | WhileStatement | ReturnStatement | HaltStatement
                     | ReadStatement | WriteStatement
                     | ";" .
  ConstDeclarations = "const" ConstSequence ";" .
  ConstSequence    = OneConst | ConstSequence "," OneConst .
  OneConst         = identifier "=" Constant .
  Constant         = number | charLit | "true" | "false" | "null" .
  VarDeclarations  = Type VarSequence ";" .
  VarSequence      = OneVar | VarSequence "," OneVar .
  OneVar           = identifier Initializer .
  Initializer      = "=" Expression | .
  Assignment       = Designator "=" Expression ";" .
  Designator       = identifier Subscript .
  Subscript        = "[" Expression "]" | .
  IfStatement      = "if" "(" Condition ")" Statement .
  WhileStatement   = "while" "(" Condition ")" Statement .
  ReturnStatement  = "return" ";" .
  HaltStatement    = "halt" ";" .
  ReadStatement    = "read" "(" ReadSequence ")" ";" .
  ReadSequence     = ReadElement | ReadSequence "," ReadElement .
  ReadElement      = stringLit | Designator .
  WriteStatement   = "write" "(" WriteSequence ")" ";" .
  WriteSequence    = WriteElement | WriteSequence "," WriteElement .
  WriteElement     = stringLit | Expression .
  Condition        = Expression .
  Expression       = AddExp ExpTail .
  ExpTail          = RelOp AddExp | .
  AddExp           = Term | AddOp Term | AddExp AddOp Term .
  Term             = Factor | Term MulOp Factor .
  Factor           =   Designator | Constant
                     | "new" BasicType "[" Expression "]"
                     | "!" Factor | "(" Expression ")" .
  Type             = BasicType ArrayFlag .
  ArrayFlag        = "[]" | .
  BasicType        = "int" | "bool" .
  AddOp            = "+" | "-" | "||" .
  MulOp            = "*" | "/" | "&&" .
  RelOp            = "==" | "!=" | "<" | "<=" | ">" | ">=" .
```

7 DETERMINISTIC TOP-DOWN PARSING

In this chapter we build on the ideas developed in the preious one, and discuss the relationship between the formal definition of the syntax of a programming language and the methods that can be used to parse programs written in that language. As with so much else in this text, our treatment is introductory, but detailed enough to make the reader aware of certain crucial issues. As we have already observed, modern programming languages are invariably defined by context-free grammars that adequately describe all but a few features of the language, where the context-sensitivity can be handled easily in other ways as we shall see. This chapter limits discussion entirely to context-free grammars (that is, where the *leftside* of every production is a single non-terminal).

7.1 Deterministic top-down parsing

The task of the front end of a translator is, of course, not the generation of sentences in a source language, but the recognition of them. This implies that the generating steps which led to the construction of a sentence (or program) must be deducible from an analysis of the finished sentence. How difficult this is to do depends on the complexity of the production rules of the grammar. For Pascal-like languages it is, in fact, not too bad, but in the case of languages like Fortran and C++ it becomes quite complicated, for reasons that may not at first be apparent.

Many different methods for parsing sentences have been developed. In this text we concentrate on a rather simple (yet quite effective) one known as **top-down parsing** by **recursive descent**, which can be applied to Pascal, Modula-2 and many similar languages, including Parva, a simple C-like one to be introduced in section 7.5, and for which we shall later construct a compiler targetting the PVM.

The reason for the phrase "by recursive descent" will become apparent later. For the moment we note that top-down methods try to verify that a string of terminals forms a valid sentence by expanding a sentential form emanating from the goal symbol, replacing the next non-terminals that occur in that sentential form by choosing from the productions identified by looking at the next terminal in the string that they have been given to parse.

To illustrate top-down parsing, consider the small language defined by the productions

$A \rightarrow abaB$ (1)
$B \rightarrow bB$ (2)
$B \rightarrow c$ (3)

Let us try to parse the string *ababbbbc*, which clearly is formed from the terminals of this grammar, to see whether it qualifies as a valid sentence. We start with the goal symbol A and the input string

Sentential form $S = A$ Input string *ababbbbc*

To the sentential form A we apply the only possible production (1) to get

Sentential form *abaB* Input string *ababbbbc*

So far we are obviously doing well. The leading terminals in both the sentential form and the input string are the same (*a*), and we can effectively discard them from both - and we can continue in turn to discard the next two terminals (*b* and then *a*), until what remains implies that from the residual non-terminal B we must be able to derive the residual string *bbbbc*.

Sentential residue B String residue *bbbbc*

We might be tempted to choose either of productions (2) or (3) in replacing the non-terminal B - but simply by looking at the leading character in the residual string we see that (2) is the better choice. If we apply this production we get

Sentential residue *bB* String residue *bbbbc*

which implies that from the non-terminal B we must be able to derive *bbbc*.

Sentential residue B String residue *bbbc*

Again we are led to use production (2) and so on until, after matching all the *b* terminals, we are left with

Sentential residue B String residue c

which implies that from the non-terminal B we hope to be able to derive the terminal c directly - which of course we can do by applying production (3).

The reader can easily verify that a string composed entirely of the terminal a (such as *aaaa*) could not be derived from the goal symbol, nor could one with b as the rightmost symbol, such as *ababbb*.

The method we are using is a special case of so-called **LL(*k*) parsing**. The terminology comes from the notion that as we scan the input string from **L**eft to right (the first L) we apply productions to the **L**eftmost non-terminal in the residual sentential form we are manipulating (the second L), and look ahead only as far as the next k terminals in the input string to help decide which production to apply at any stage. In our example, fairly obviously, $k = 1$: LL(1) parsing is the most common form of LL(*k*) parsing in practice.

Parsing in this way is not always as easy, as is evident from the following grammar where, as before, A is the goal symbol.

$A \rightarrow aB$	(1)
$A \rightarrow aC$	(2)
$B \rightarrow aB$	(3)
$B \rightarrow b$	(4)
$C \rightarrow aC$	(5)
$C \rightarrow c$	(6)

An attempt to parse the sentence *aaac* might proceed as follows

Sentential form $S = A$ Input string *aaac*

In manipulating the sentential form A we must make a choice between productions (1) and (2). We do not get any real help from looking at the first terminal in the input string, but let us try production (1). This leads to

Sentential form aB Input string *aaac*

which implies that we must be able to derive the residual *aac* from B. We now have a much clearer choice. Of the productions for B it is (3) which will yield an initial *a*, so we apply it and get to

Sentential residue aB String residue *aac*

which implies that we must be able to derive *ac* from B. If we apply (3) again we get

Sentential residue aB String residue *ac*

which implies that we must be able to derive c directly from B, which we cannot do. We must conclude either that we cannot derive the string, or that we must have made a wrong decision somewhere along the line. In this case, fairly obviously, we went wrong right at the beginning. Had we used production (2) and not (1) we should have matched the string quite easily.

When faced with this sort of dilemma, a parser might adopt the strategy of simply proceeding according to one of the possible options, being prepared to retreat along the chosen path if no further progress is possible. Any **backtracking** action is clearly inefficient, and even with a grammar as simple as this there is almost no limit to the amount of backtracking one might have to be prepared to do. This leads to advice to language designers: syntactic structures which can be described *only* by productions that run the risk of requiring backtracking algorithms should be identified, and avoided.

This may not be possible after the event of defining a language, of course - Fortran is full of examples where it seems backtracking might be needed. A classic example is provided by the pair of statements

```
DO 10 I = 1 , 2
```

and

```
DO 10 I = 1 . 2
```

These are distinguishable as examples of two totally different statement types (`DO` statement and `REAL` assignment) only by the period/comma. This kind of problem is minimized in modern languages by the introduction of reserved keywords, and by an insistence that white space appear between some tokens (neither of which is a feature of Fortran, but neither of which causes difficulties for programmers who have never known otherwise).

The consequences of backtracking for full-blooded translators are far more severe than our simple example might suggest. Typically, translators do not simply read single characters (even "unreading" characters is awkward enough for a computer); they also construct explicit or implicit trees, generate code, create symbol tables and so on - all of which may have to be undone, perhaps just to be redone in a very slightly different way. In addition, backtracking makes the detection of malformed sentences more complicated. All in all, it is best avoided.

It might occur to the reader that some of these problems - including some real ones too, like the Fortran example just given - could be resolved by looking ahead more than one symbol in the input string. Perhaps to solve this particular problem we should have been prepared to scan four symbols ahead? A little more reflection shows that even this may be quite futile. The language which this grammar generates can be described by:

$$L(G) = \{ a^n p \mid n > 0, \quad p \in \{b , c\} \}$$

or, if the reader prefers less formality

"at least one, but otherwise as many *a*s in a row as you like, followed by a single b or c"

We note that being prepared to look more than one terminal ahead is a strategy which can work well in some situations (Parr and Quong 1996), although, like backtracking, it will clearly be more difficult to implement.

To be able to rid ourselves of such complications, we should try to confine ourselves to the use of *deterministic* parsing methods - those where at each stage we can be sure of which production to apply next - or where, if we cannot find a production to use, we can be sure that the input string is malformed. In most cases this simply means finding the most suitable form of grammar to use. As we shall shortly be able to justify, each of the following equivalent EBNF style grammars also describes this language:

$$\begin{aligned} A &\rightarrow a\,(A \mid B) \\ B &\rightarrow b \mid c \end{aligned}$$

or

$$\begin{aligned} A &\rightarrow a\ B \\ B &\rightarrow A \mid b \mid c \end{aligned}$$

or

$$A \rightarrow a\,\{\,a\,\}\,(b \mid c)$$

and are perfectly suited to LL(1) parsing.

7.2 Restrictions on grammars so as to allow LL(1) parsing

The top-down approach to parsing looks so promising that we should consider what restrictions have to be placed on a grammar so as to allow us to use the LL(1) approach (and its close cousin, the method of recursive descent). Once these have been established we shall pause to consider the effects they might have on the design or specification of "real" languages.

A little reflection on the examples above will show that problems might arise when we encounter alternative productions for the leading (left-most) non-terminal in the sentential form, and should lead to the insight that the top-down technique will succeed only if the initial *terminal* symbols that can be derived from any alternative

right-hand sides of the production for this non-terminal are distinct (that is, are mutually exclusive).

Hence, for example, if we have reached a sentential form like $wA\alpha$ where w is a sequence of terminals that matches the leading part of the string being parsed, and find that the leading non-terminal A has five possible productions

$$A \rightarrow aBx \mid aCy \mid cD \mid cE \mid fG$$

then we would be able to accept one of a, c or f as the next terminal of the string, but would not immediately know which production to apply if this terminal happened to be a (either of the first two alternatives might apply) or if it happened to be c (either the third or fourth alternatives might apply).

There are various ways to overcome this problem, of course. We might try writing an equivalent grammar

$$\begin{aligned} A &\rightarrow aX \mid cY \mid fG \\ X &\rightarrow Bx \mid Cy \\ Y &\rightarrow D \mid E \end{aligned}$$

The productions for A now look as though they avoid the previous problem - but we must now consider whether the alternatives for X and Y are also clear of similar problems, and this may not be immediately obvious, as these alternatives have been defined in terms of further leading non-terminals rather than terminals.

Note that these complications do not necessarily mean that the grammar is *ambiguous*, merely that our suggested parsing method might be *indeterminate*. All of the grammars in section 7.1 permit each of the sentences of the language to be derived in only one way. How do we ensure *determinacy*?

7.2.1 Terminal starters and the FIRST function

For the moment let us suppose that each of the alternative right hand sides for the productions for a given non-terminal A is composed simply of a string of non-terminals and terminals, and that neither A or any other of these non-terminals is nullable. Furthermore, let us assume that the productions are not expressed using any of the meta-brackets suggested in our preferred EBNF notation, that is, that we are restricting our discussion to "BNF style" productions.

To enhance the discussion, we introduce the concept of the **terminal starters** of a non-terminal. The set FIRST(A) of the non-terminal A is defined to be the set of all terminals with which a string that can be derived from A might start, that is

$$a \in \text{FIRST}(A) \qquad \text{if } A \Rightarrow^{+} a\beta \qquad (A \in N \ ; \ a \in T \ ; \ \beta \in (N \cup T)^{*} \ )$$

Furthermore, given that the right hand sides for the alternatives for A are often strings we shall also introduce the related concept of the terminal starters of a general string σ in a similar way, as the set of all terminals with which σ or a string derived from σ can start, that is

$$a \in \text{FIRST}(\sigma) \qquad \text{if } \sigma \Rightarrow^{*} a\beta \qquad (a \in T \ ; \ \beta \in (N \cup T)^{*} \ )$$

It is important to distinguish between FIRST(σ) and FIRST(A). The string σ might consist of a single non-terminal A, but in general it could be a concatenation of several symbols.

7.2.2 Computation of FIRST sets for non-nullable non-terminals and strings

These definitions for set membership do not of themselves seem to indicate how one might actually compute the sets, but this is not difficult to do. If, as usual, α and β denote arbitrary strings, a denotes an arbitrary terminal and A and B denote arbitrary (but non-nullable) non-terminals, then we may compute any FIRST set we might need by applying the following relationships recursively until closure is obtained:

For the strings σ on the right side of a production:

- if σ is of the form $\sigma = a\beta$ then FIRST(σ) = { a };

- if σ is of the form $\sigma = A\beta$ then FIRST(σ) = FIRST(A);

For the non-terminals on either side:

- If A is defined by a single production $A = a$ then FIRST(A) =$\{ a \}$;
- if A is defined by a single production of the form $A = a\beta$ then FIRST(A) = $\{ a \}$;
- if A is defined by a single production of the form $A = B\beta$ then FIRST(A) = FIRST(B);

For the non-terminal A on the left side of the set of productions for $L(G)$:

- if A is defined by the alternative productions $A = \alpha_1 \mid \alpha_2 \mid \ \ldots \alpha_n$
  then FIRST(A) = FIRST(α_1) $\cup$ FIRST(α_2) $\ldots$ $\cup$ FIRST(α_n).

7.2.3 First LL(1) condition for ε-free context-free grammars (CFGs)

With the aid of these set definitions and relationships we may express a rule that easily allows us to determine whether an ε-free grammar $G = \{ N , T , S , P \}$ is LL(1), that is, can be parsed deterministically in the way described in section 7.1:

Rule 1

For each and every non-terminal $A_i \in N$ that admits alternatives

$$A_i \rightarrow \ \alpha_{i1} \mid \alpha_{i2} \mid \ \ldots \ \ \alpha_{in}$$

the FIRST sets of all strings that can be generated from each of the alternative α_{ik} must be disjoint, that is

$$\text{FIRST}(\alpha_{ij}) \ \cap \ \text{FIRST}(\alpha_{ik}) \ = \ \varnothing \qquad \text{for all } j \neq k$$

It is important to realize at the outset that this rule applies in a *pairwise* fashion to the FIRST sets that are derived from alternative *right-hand sides* of a production. It says nothing about the intersection of the FIRST sets derived for non-terminals that appear on the *left-hand sides* of the complete set of productions for the grammar. That is, it does not prescribe anything about FIRST(A_p) $\cap$ FIRST(A_q) with $p \neq q$. Another common misconception is to assume that the rule is equivalent to demanding that

$$\text{FIRST}(\alpha_{i1}) \ \cap \ \text{FIRST}(\alpha_{i2}) \ \cap \ \text{FIRST}(\alpha_{i3}) \ \ldots \cap \ \text{FIRST}(\alpha_{in}) \ = \ \varnothing \qquad \text{(wrong!)}$$

If all the alternative right-hand sides for a particular non-terminal A were simply of the form

$$\alpha_k \ = \ a_k \beta_k \qquad (a_k \in T \ ; \ \alpha_k , \beta_k \in (N \cup T)^+ \ )$$

it would be easy to check the grammar very quickly, but it is a little restrictive to expect that we can write or rewrite all productions with alternatives in this form. More likely we shall find several alternatives of the form

$$\alpha_k \ = \ B_k \beta_k$$

where B_k is another non-terminal. In this case to find FIRST($B_k\beta_k$) we shall have to consider the closure rules suggested earlier, by considering possible alternatives for B_k, and looking at the first terminals which can arise from those (and so it goes on, because there may be alternatives all down the line). All of these must then be included in the set FIRST(α_k).

As a simple illustration, consider the set of productions below, which generates exciting sentences with any number of *a*s, followed by *ba* or by *c*.

$$\begin{array}{lll} A \rightarrow Ba \mid C & & (A_1, A_2) \\ B \rightarrow aB \mid b & & (B_1, B_2) \\ C \rightarrow aC \mid c & & (C_1, C_2) \end{array}$$

All three non-terminals A, B and C admit to alternatives, but it takes only a moment to see that the alternatives for B and C satisfy Rule 1. However, on looking at the alternatives for the non-terminal A we see that

$$\text{FIRST}(A_1) = \text{FIRST}(Ba) = \text{FIRST}(B) = \text{FIRST}(aB) \cup \text{FIRST}(b) = \{ a , b \}$$
$$\text{FIRST}(A_2) = \text{FIRST}(C) = \text{FIRST}(aC) \cup \text{FIRST}(c) = \{ a , c \}$$

so that Rule 1 is violated as both FIRST(A_1) and FIRST(A_2) have a as a member. If we were to set out to parse a string starting $a\ldots$ we should be in trouble.

It is important to be quite clear as to when and how one should compute and use FIRST sets. Rule 1 applies only to those productions that have alternatives, but then must be applied to each pair of alternatives for each of those productions. In the above example A, B and C each have two alternatives, so it is necessary firstly to consider FIRST(A_1) with FIRST(A_2), secondly to consider FIRST(B_1) with FIRST(B_2) and thirdly to consider FIRST(C_1) with FIRST(C_2) as three separate applications of the rule. A frequent mistake made by beginners is simply to construct the FIRST sets for each non-terminal and see whether they appear to have anything in common. The fact that FIRST(B) and FIRST(C) both have a as a member would be irrelevant but for the fact that Ba and C are the alternatives for A, and that FIRST(Ba) happens to reduce to FIRST(B).

As an exercise, try to find an equivalent set of productions that describe this language and also satisfy Rule 1.

One further example may help to clarify still further. Consider a grammar in which the non-terminals A, B, W, X, Y, Z appear in the productions

$$A \rightarrow aX \mid bY \mid cZ \qquad (A_1, A_2, A_3)$$
$$B \rightarrow A \mid W \qquad (B_1, B_2)$$

Here the three FIRST sets associated with the right sides of A , namely $\{ a \}$, $\{ b \}$ and $\{ c \}$ are pairwise disjoint so that A satisfies Rule 1, but it is the union of those sets, namely $\{ a, b, c \}$ (that is, FIRST$\{A\}$) that must be disjoint from FIRST(W) in order for B to satisfy Rule 1.

7.2.4 Computation of FIRST sets for arbitrary non-terminals and strings

It is somewhat unreasonable to expect that a grammar G will never contain ε-productions, or to demand that these be eliminated before parsing can be attempted. To handle ε-productions it is necessary to extend our definitions of the FIRST sets slightly. This can be done in various ways. The most common seems to be to permit ε to be a member of FIRST(A) for a non-terminal A that is nullable, even though ε is not usually regarded as a member of the terminal vocabulary T of the grammar. Like this, if A is nullable then $\varepsilon \in$ FIRST(A), and it is the set FIRST(A) - $\{ \varepsilon \}$ that will contain any and all elements of FIRST(A) that are strictly terminals. Similarly, ε might be regarded as a member of FIRST(σ) if σ were a general but nullable string.

However, it turns out to be slightly simpler to follow those authors who restrict the definition of the FIRST sets to contain elements of the terminal vocabulary T only, and so that is what we shall illustrate here.

The closure rules given earlier have to be modified slightly:

For the strings σ on the right side of a production:

- if σ is of the form $\sigma = a\beta$ then FIRST(σ) = $\{ a \}$;
- if σ is of the form $\sigma = A\beta$
  then if A is non-nullable, FIRST(σ) = FIRST(A);
  else if A is nullable, FIRST(σ) = FIRST(A) $\cup$ FIRST(β);
- if $\sigma = \varepsilon$ then FIRST(σ) = $\varnothing$;

For the non-terminals on either side:

- if A is defined by the single production $A = \varepsilon$ then FIRST(A) = $\varnothing$;
- if A is defined by the single production $A = a$ then FIRST(A) = $\{ a \}$;

- if A is defined by a single production of the form $A = a\beta$ then FIRST(A) = { a }.

- If A is defined by a single production of the form $A = B\beta$
  then if B is non-nullable, FIRST(A) = FIRST(B);
  else if B is nullable, FIRST(A) = FIRST(B) ∪ FIRST(β).

For the non-terminal A on the left side of the set of productions for $L(G)$:

- If A is defined by the production $A = \alpha_1 \mid \alpha_2 \mid \ \ldots \alpha_n$
  then FIRST(A) = FIRST(α_1) ∪ FIRST(α_2) … ∪ FIRST(α_n).

As a simple illustration, given that

$\sigma = Abc$
$A \rightarrow x \mid y \mid z \mid \varepsilon$
FIRST(A) = { x, y, z }
FIRST(σ) = { x, y, z } ∪ FIRST(bc) = { x, y, z, b }

7.2.5 First LL(1) condition for context-free grammars

With these modifications, Rule 1 can be applied even in the presence of nullable productions.

We can demonstrate this with another grammar, rather similar to the one in section 7.2.3. Suppose we have

$A \rightarrow d \mid Cd$ $\quad (A_1, A_2)$
$B \rightarrow aB \mid b$ $\quad (B_1, B_2)$
$C \rightarrow aC \mid c \mid \varepsilon$ $\quad (C_1, C_2, C_3)$

The three alternatives for non-terminal C cause no problems. FIRST(C_1) = { a }, FIRST(C_2) = { c } and FIRST(C_3) = ∅, and these are all pairwise disjoint. The two alternatives for A bear closer scrutiny. FIRST(A_1) = { d }, and it might at first seem that FIRST(A_2) = FIRST(C) where FIRST(C) = { a , c }, and that Rule 1 was satisfied. However, proper application of the rules above will show that, as C is nullable, FIRST(A_2) = FIRST(C) ∪ FIRST(d) = { a, c } ∪ { d } = { a , c , d }, and so Rule 1 is broken.

7.2.6 Terminal successors of nullable non-terminals; the FOLLOW function

We have already noticed how ε-productions cause slight difficulties in parsing and in applying Rule 1. Sadly, Rule 1 on its own is not strong enough to detect another source of trouble which may arise if such productions are used. Consider the very simple grammar

$A \rightarrow Ba$ $\quad (A_1)$
$B \rightarrow ab \mid \varepsilon$ $\quad (B_1, B_2)$

In terms of the discussion above, Rule 1 is satisfied. Of the alternatives for the non-terminal B, we see that

FIRST(B_1) = FIRST(ab) = { a }
FIRST(B_2) = FIRST(ε) = ∅

which are disjoint. However, if we try to parse the string a we may come unstuck:

Sentential form	$S = A$	Input string	a
Sentential form	Ba	Input string	a

As we are working from left to right and have a non-terminal on the left, we would be tempted to substitute for B, using B_1, to get

Sentential form	aba	Input string	a

which is clearly wrong. We should have used (3), not (2). How do we detect this, if we are trying to parse a string by looking only at the next terminal in the input?

This situation arises only for productions which can generate the null string. One might try to rewrite the grammar so as to avoid ε-productions, but in fact that is not always necessary, and, as we have commented, it is sometimes highly inconvenient.

A further simple grammar may be used to cast further light on the problem. Consider the grammar

$$\begin{array}{lll} A \rightarrow Bef & & (A_1) \\ B \rightarrow ab \mid cd \mid \varepsilon & & (B_1, B_2, B_3) \end{array}$$

Once again, Rule 1 is satisfied for the alternatives for the non-terminal B. If we are asked to parse the string *abef* we would be led to

Sentential form	$S = A$	Input string	*abef*
Sentential form	*Bef*	Input string	*abef*

and of the alternatives for B we would be directed to use B_1 and successfully match the input string. But if we were asked to parse the string *ef* we would be led to

Sentential form	$S = A$	Input string	*ef*
Sentential form	*Bef*	Input string	*ef*

As we are working from left to right and have a non-terminal on the left, we presumably should try to substitute for B - and, sadly, and unlike the previous example, neither alternative matches the terminal c at the start of the string being parsed.

However if, in this case, we were to erase B by applying the ε-production we should be left with a sentential form which *does* match the sentence being parsed.

With a little insight we should be able to see that if a non-terminal A is nullable, and any non-nullable alternative right sides do not allow the parse to continue, then examining the terminals that might legitimately follow A may help decide that the ε-production is the one to be applied - or the string rejected. But of course this decision cannot be made if the terminals that can help select the non-nullable alternatives for A and the terminals that might follow the nullable option are not distinct.

With this in mind, it is convenient to define the **terminal successors** of a non-terminal A as the set of all terminals that can follow A in any sentential form derived from S, that is

$$a \in \text{FOLLOW}(A) \qquad \text{if } S \Rightarrow^* \alpha A a \beta \qquad (A, S \in N \; ; \; a \in T \; ; \; \alpha, \beta \in (N \cup T)^*)$$

7.2.7 Second LL(1) rule for context-free grammars

Over and above Rule 1 (which still applies even if productions are nullable), if we are to ensure easy top-down parsing, a critical restriction that we must impose on a context-free grammar if it incorporates ε-productions is covered by a second rule.

Rule 2

For each non-terminal $A_i \in N$ that admits alternatives

$$A_i \rightarrow \alpha_{i1} \mid \alpha_{i2} \mid \ldots \alpha_{in}$$

but where $\alpha_{ik} \Rightarrow \varepsilon$ for some k, the FIRST sets that can be generated from each of the α_{ij} for $j \neq k$ must be disjoint from the set FOLLOW(A_i) of symbols that may follow any sequence generated from A_i, that is

$$\text{FIRST}(\alpha_{ij}) \cap \text{FOLLOW}(A_i) = \varnothing, \qquad j \neq k$$

or, rather more loosely,

$$\text{FIRST}(A_i) \cap \text{FOLLOW}(A_i) = \varnothing$$

where, as before,

$$\text{FIRST}(A_i) = \text{FIRST}(\alpha_{i1}) \cup \text{FIRST}(\alpha_{i2}) \cup \ldots \text{FIRST}(\alpha_{in})$$

We note in passing that, while FIRST is a function that may be applied to a string (in general) and to a non-terminal (in particular) and may yield an empty set, FOLLOW is a function that is almost invariably applied to a nullable non-terminal (only) and must yield a non-empty set if it is to be of any use in applying Rule 2. Although we can define FOLLOW(A) for any non-terminal, the calculation is only ever useful for nullable non-terminals.

7.2.8 Computation of FOLLOW sets for non-terminals

When determining the FIRST set of a non-terminal A we focused attention on those productions in which A appeared on the *left-hand* side of productions. In contrast, the set FOLLOW(A) is computed by considering every production where A appears on the *right-hand* side. That is, we look for any and all productions P_k of the form

$$P_k \rightarrow \alpha_k A \beta_k$$

The terminals that follow A are then found by considering the sets FIRST(β_k), when it might appear that

$$\text{FOLLOW}(A) = \text{FIRST}(\beta_1) \cup \text{FIRST}(\beta_2) \cup \ldots \cup \text{FIRST}(\beta_n)$$

and, indeed, this would be the case were it not for two complications. Firstly, if any β_k *is* nullable, then the set FOLLOW(P_k) has to be included in the set FOLLOW(A), as in this case anything that could have followed A must be present by virtue of earlier applying the production P_k in whatever context one had chosen to do so.

Secondly, the calculation of a FOLLOW set serves really to predict when to apply an ε-production, and for this purpose it is not helpful if it turns out to be empty. This problem will arise if a nullable non-terminal A does not appear on the right-hand side of *any* production, as might happen in the case of a grammar with which it is possible to derive a completely empty string. A trivial example of this is provided by

$$\begin{array}{lll} A & \rightarrow & ab \quad (1) \\ A & \rightarrow & \varepsilon \quad (2) \end{array}$$

Here the nullable non-terminal A admits to two alternatives, but does not appear on the right-hand side of either production. For completeness, situations like this may be handled by constructing a so-called **augmented grammar** by adding a new terminal symbol (denoted, say, by \$), a new goal symbol, and a new single production. For the above example we would create an augmented grammar on the lines of

$$\begin{array}{lll} S & \rightarrow & A\,\$ \quad (1) \\ A & \rightarrow & ab \quad (2) \\ A & \rightarrow & \varepsilon \quad (3) \end{array}$$

The new terminal \$ amounts to an explicit end-of-file or end-of-string symbol; here FOLLOW(A) = (ε). Since realistic parsers and scanners must always be able to detect and react to an end-of-file in a sensible way, augmenting a grammar in this way really carries no practical overheads. For simplicity it might be easiest at the outset simply to add \$ to FOLLOW(S), where S is the goal symbol for the grammar.

7.2.9 A further example

In the very trite example given at the start of section 7.2.6, Rule 2 is clearly violated, because

$$\text{FIRST}(B_1) = \text{FIRST}(ab) = \{ a \} = \text{FOLLOW}(B)$$

It may be worth studying a more demanding example, so as to explore these rules further. Consider the language defined by the grammar below where, as usual, A is the goal symbol.

$A \rightarrow CB \mid BD$	(1, 2)
$B \rightarrow b\,B\,d \mid c \mid E$	(3, 4, 5)
$C \rightarrow a \mid d$	(6, 7)
$D \rightarrow b \mid d$	(8, 9)
$E \rightarrow e \mid \varepsilon$	(10, 11)

All five non-terminals admit to alternatives, and B and E are capable of generating the empty string ε. Rule 1 is clearly satisfied for the alternative productions for B, C, D and E, since these alternatives all produce sentential forms that start with distinctive terminals.

To check Rule 1 for the alternatives for A requires a little more work. We need to examine the intersection of FIRST(CB) and FIRST(BD).

Since C is not nullable, FIRST(CB) is simply FIRST(C) = $\{ a \} \cup \{ d \} = \{ a , d \}$.

FIRST(BD) is not simply FIRST(B), since B is nullable. Applying our rules to this situation leads to the result that FIRST(BD) = FIRST(B) $\cup$ FIRST(D) = $\{ b , c , e \} \cup \{ b, d \} = \{ b , c , d , e \}$.

Since FIRST(CB) $\cap$ FIRST(BD) = $\{ d \}$, Rule 1 is broken. If we were given a string to parse that began with d we could not determine easily whether to apply production 1 or 2 to continue the parse.

For completeness we might check Rule 2 for the productions for B and for E. We have already noted that FIRST(B) = $\{ b , c , e \}$. To compute FOLLOW(B) we need to consider all productions where B appears on the right-hand side. These are productions (1), (2) and (3). This leads to the result that

$$\begin{aligned} \text{FOLLOW}(B) &= \text{FOLLOW}(A) && \text{(from the rule } A \rightarrow CB\text{)} \\ &\quad \cup \text{FIRST}(D) && \text{(from the rule } A \rightarrow BD\text{)} \\ &\quad \cup \text{FIRST}(d) && \text{(from the rule } B \rightarrow bBd\text{)} \\ &= \{ \$ \} \cup \{ b , d \} \cup \{ d \} = \{ \$, b , d \} \end{aligned}$$

Since FIRST(B) $\cap$ FOLLOW(B) = $\{ b , c , e \} \cap \{ \$, b , d \} = \{ b \}$, Rule 2 is broken as well. Checking Rule 2 for the non-terminal E leads us to consider the only production (5) where E appears on the right-hand side. FIRST(E) = $\{ e \}$ and FOLLOW(E) = FOLLOW(B) = $\{ \$, b , d \}$; Rule 2 is satisfied in this case.

7.2.10 Director sets - an alternative formulation of the LL(1) conditions

By now the reader should be comfortable with the idea that topdown parsing involves a process of starting from the goal symbol of the grammar, using the production rules to substitute for the non-terminals that appear in successive sentential forms, and being "directed" at each stage where a choice might be made by the next terminal in the input string. For each alternative right side of a context-free productions there may be a set of one or more acceptable terminals from which to choose. We introduce the concept of **director sets**:

The sets DS(A_i, α_{ik}) for each alternative right side α_{ik} for the productions for non-terminal A_i are the terminal sets which effectively prescribe whether to apply the alternative α_{ik}. That is, when the input string contains the terminal a, we choose α_{ik} such that $a \in$ DS(A_i, α_{ik}).

If we use the concept of director sets, the two rules for determining whether a grammar is LL(1) may be succinctly combined into one rule as follows:

Combined LL(1) Rule

A grammar $G = \{ N , T , S , P \}$ is LL(1) if for every non-terminal $A_i \in N$ that admits alternatives

$$A_i \rightarrow \alpha_{i1} \mid \alpha_{i2} \mid \ldots \alpha_{in}$$

$$\text{DS}(A_i, \alpha_{ij}) \cap \text{DS}(A_i, \alpha_{ik}) = \varnothing, \qquad j \neq k$$

where

$$\begin{aligned} \text{DS}(A_i, \alpha_{ik}) &= \text{FIRST}(\alpha_{ik}) && \text{if } \alpha_{ik} \not\Rightarrow^* \varepsilon \\ &= \text{FIRST}(\alpha_{ik}) \cup \text{FOLLOW}(A_i) && \text{if } \alpha_{ik} \Rightarrow^* \varepsilon \end{aligned}$$

For the example discussed in the last section we would have

$$\begin{array}{lll} \text{DS}(A,\ CB) & = \text{FIRST}(CB) & = \{\ a\ ,\ d\ \} \\ \text{DS}(A,\ BD) & = \text{FIRST}(BD) & = \{\ b\ ,\ c\ ,\ d\ ,\ e\ \} \\ \text{DS}(B,\ bBd) & = \text{FIRST}(bBd) & = \{\ b\ \} \\ \text{DS}(B,\ c) & = \text{FIRST}(c) & = \{\ c\ \} \\ \text{DS}(B,\ E) & = \text{FIRST}(E)\ \cup\ \text{FOLLOW}(B) & = \{\ \$\ ,\ b\ ,\ d\ ,\ e\ \} \end{array}$$

and so on. The first two of these show that the LL(1) rule is violated for the productions for A and the third and fifth show that it is violated for the productions for B.

7.3 LL(1) restrictions for grammars defined using EBNF notation

The rules derived in this chapter have been expressed in terms of regular BNF notation, and we have so far avoided discussing whether they might need modification to handle grammars where the productions are expressed in terms of the grouping, option and repetition (closure) metasymbols (( ) , [ ] and { } respectively). Productions that are expressed in terms of these symbols are easily rewritten into standard BNF style by the introduction of extra non-terminals. For example, the production

$$A\ \rightarrow\ \xi\ (\ \mu\ |\ \nu\ )\ \zeta$$

is readily seen to be equivalent to

$$\begin{array}{l} A\ \rightarrow\ \xi\ B\ \zeta \\ B\ \rightarrow\ \mu\ |\ \nu \end{array}$$

and the requirement to be met for LL(1) compliance is that

$$\text{FIRST}(\mu)\ \cap\ \text{FIRST}(\nu)\ =\ \varnothing$$

provided that neither μ nor ν are nullable. If μ is nullable the requirement would become

$$(\ \text{FIRST}(\mu)\ \cup\ \text{FOLLOW}(\mu)\ )\ \cap\ \text{FIRST}(\nu)\ =\ \varnothing$$

with a similar condition if ν is nullable. Note that μ and ν cannot both be nullable - in that case the grammar would be ambiguous, as there would be two alternative ways to handle the ε component!

In most cases FOLLOW(μ) and FOLLOW(ν) would simply be FIRST(ζ) - but of course if ζ were nullable or absent we should have to add FOLLOW(A) as well.

As a simple example, within the production

$$A\ \rightarrow\ a\ (\ bC\ |\ cD\ )\ e\ (\ f\ |\ g\ )$$

there are two applications of the grouping parentheses, but it is readily seen that, were an analysis to be performed on this production, no violations of the LL(1) rules would seem possible. This is true for this production, but the grammar as a whole is still subject to an analysis of the productions for C and D.

The analysis needs care whenever nullable non-terminals appear - or more generally, whenever what we shall call **nullable sections** appear within productions. For example, a production like

$$A\ \rightarrow\ \alpha\ [\ \beta\ ]\ \gamma$$

is readily seen to be equivalent to

$$\begin{array}{l} A\ \rightarrow\ \alpha\ C\ \gamma \\ C\ \rightarrow\ \beta\ |\ \varepsilon \end{array}$$

to which the rules, as given earlier, are again easily applied. Note once again that β cannot be nullable without rendering the grammar ambiguous. In effect, of course, the requirement that

$$\text{FIRST}(C) \cap \text{FOLLOW}(C) = \varnothing$$

amounts to saying that

$$\text{FIRST}(\beta) \cap \text{FOLLOW}(\ [\beta]\ ) = \varnothing$$

where FOLLOW(C) is normally FIRST(γ) unless γ is nullable, in which case one has to include FOLLOW(A) as usual. In this case, FOLLOW(β) and FOLLOW([β]) are the same.

Things get more interesting still when one uses the repetition braces. The production

$$A \rightarrow \rho\ \{\ \sigma\ \}\ \tau$$

(where σ is non-nullable) can easily be expanded to

$$\begin{aligned} A &\rightarrow \rho\ D\ \tau \\ D &\rightarrow \sigma\ D\ \mid \varepsilon \end{aligned}$$

to which the rules, as given earlier, are again easily applied (note that the production for D is right recursive). In effect, of course, the requirement

$$\text{FIRST}(D) \cap \text{FOLLOW}(D) = \varnothing$$

amounts in this case to saying that

$$\text{FIRST}(\sigma) \cap \text{FOLLOW}(\ \{\sigma\}\ ) = \varnothing$$

Notice that we are mentally no longer applying the FOLLOW function to non-terminals only and that care must be taken with constructs involving optional repetition. FOLLOW($\{\sigma\}$) is not the same as FOLLOW(σ). The production above allows repetition of its σ component, so that, if τ is non-nullable

$$\text{FOLLOW}(\ \{\sigma\}\ ) = \text{FIRST}(\tau)$$

but

$$\text{FOLLOW}(\sigma) = \text{FIRST}(\sigma) \cup \text{FIRST}(\tau) = \text{FIRST}(\sigma) \cup \text{FOLLOW}(\ \{\sigma\}\ )$$

This may be exemplified by considering the following grammar, which might at first glance appear to satisfy the LL(1) constraints:

$$\begin{aligned} A &\rightarrow a\ \{\ B\ \}\ d && (1) \\ B &\rightarrow b\ [\ C\ ]\ \mid c && (2, 3) \\ C &\rightarrow c && (4) \end{aligned}$$

This incorporates two nullable sections - { B } and [C]. Since FIRST($\{B\}$) = FIRST(B) = { b, c }, while FOLLOW($\{B\}$) = { d } the production for A passes the test easily.

When we consider the nullable section in the productions for B we find that FIRST([C]) = FIRST(C) = { c } and FOLLOW([C]) = FOLLOW(B) = FIRST(B) $\cup$ FOLLOW($\{B\}$) = { b , c, d }, and so the presence of the common element c renders the grammar non-LL(1).

With a bit of practice in critically looking at the nullable components in EBNF style productions, the reader will become quite adept at spotting violations of the LL(1) constraints. But it is reasonable to assume that in a really large grammar one might have to make many iterations over the productions in forming all the FIRST and FOLLOW sets and in checking the applications of all these rules. Fortunately, software tools are available to help in this regard and the reader is urged to learn how to use them. Any reasonable LL(1) compiler generator like Coco/R will, of course, incorporate checks that the grammars it is asked to handle satisfy the LL(1) constraints.

7.4 Language design and the consequences of the LL(1) conditions

There are some immediate implications which follow from the rules of the last section as regards language design and specification. If we embark on trying to define a computer language we shall sensibly be led to trying to define its syntax using one or other notation - which these days inevitably means one or other variation on BNF or EBNF. Concepts like *Statement*, *Expression* and *Declaration* can be captured by defining non-terminals and productions for these, and inevitably we shall find that alternative right-hand sides for productions are very common - we cannot hope to avoid their use in practice. Our first attempts at writing a grammar may well turn up one that is not LL(1). There are other parsing methods than the one we have discussed that may be able to handle these - but suppose we want to take advantage of the fact that the top-down one we have described is so attractive. Can we use what we have to write an equivalent grammar that *is* LL(1) conformant? This brief section considers some common situations where problems might arise and examines the possibilities.

At the outset, we should note that we cannot hope to transform every non-LL(1) grammar into an equivalent LL(1) grammar. To take an extreme example, an ambiguous grammar must have two parse trees for at least one input sentence. If we *really* want to allow this we shall not be able to use a parsing method that is capable of finding only one parse tree, as deterministic parsers must do. We can argue that an ambiguous grammar is of little interest, but the reader should not go away with the impression that it is just a matter of trial and error before an equivalent LL(1) grammar is found for an arbitrary grammar.

7.4.1 Refactoring productions with a common leading term

The classic case of productions that flaunt the LL(1) condition may be described by

$$A \rightarrow \alpha\,\beta \mid \alpha\,\gamma$$

that is, two alternatives that both start with the component α, where α is some non-null string of terminals and non-terminals. In this case, re-factorization will resolve that problem - provided that β and γ do not have common leading components:

$$A \rightarrow \alpha\ (\beta \mid \gamma)$$

For example, it is almost trivially easy to find an LL(1) grammar for the problematic language of section 7.1. Once we have seen the types of strings the language allows, then all we have to do is to find productions that sensibly deal with leading strings of *a*s, but delay introducing *b* and *c* for as long as possible. This insight leads to recursive productions of the form

$$A \rightarrow a\,(A \mid B)$$
$$B \rightarrow b \mid c$$

or, if we prefer to use EBNF

$$A \rightarrow a\,\{\,a\,\}\,(b \mid c)$$

or, perhaps not quite so obviously, we could make the transformation

$$A \rightarrow a\,A \mid b \mid c$$

Productions with alternatives are often found in specifying the kinds of *Statement* that a programming language may have. Rule 1 suggests that if we wish to parse programs in such a language by using LL(1) techniques we should design the language so that each statement type begins with a different reserved keyword. This is what is attempted in several languages, but it is not always convenient, and we may still have to solve various problems by careful and annoying factorization. In C, for example, although the use of keywords like `while`, `switch`, `return` allows several statement types to be recognized immediately, declarations of variables and of functions can both start with key words like `int` or `bool`, as exemplified by

```
int i, j, k;
int function (int i, int j) {
  return i + j;
}
```

Although a complete grammar is more complicated than we wish to discuss here, using the "factor out the common leading part of the right sides" idea might suggest using productions like

```
DeclarationStatement  = "int" Identifier RestOfDeclaration .
RestOfDeclaration     = { "," Identifier } ";"
                      | "(" Parameters ")" FunctionBody .
```

This would allow LL(1) parsing to proceed quite happily. However, we have delayed recognizing the underlying statement type until rather later than might have been convenient - the "kind" of the leading *Identifier* is not the same as one embarks on accepting one or other alternative *RestOfDeclaration*, even though from a lexical point of view there is no difference.

Another example is provided by an attempt to describe an *IfStatement* in a Pascal-like manner using productions of the form

```
Statement     = IfStatement | OtherStatement .
IfStatement   = "IF" Condition "THEN" Statement
              | "IF" Condition "THEN" Statement "ELSE" Statement .
```

Factorization on the same lines as for the *DeclarationStatement* is less successful. We might be tempted to try

```
Statement     = IfStatement | OtherStatement .                  (1, 2)
IfStatement   = "IF" Condition "THEN" Statement IfTail .        (3)
IfTail        = "ELSE" Statement | ε .                          (4, 5)
```

but then we run foul of Rule 2. The production for *IfTail* is nullable - a little reflection shows that

FIRST("ELSE" *Statement*) = { "ELSE" }

while to compute FOLLOW(*IfTail*) we consider the production (3) (which is where *IfTail* appears on the right-hand side) and obtain

```
FOLLOW(IfTail) = FOLLOW(IfStatement)    (production 3)
               = FOLLOW(Statement)      (production 1)
```

which clearly includes ELSE.

The reader will recognize this as the "dangling else" problem again. We have already remarked (in section 6.4) that we can find a way of expressing this construct unambiguously. That way turns out to be non-LL(1). But in fact the more usual solution is just to impose the semantic meaning that the ELSE is attached to the most recent unmatched THEN which, as the reader will discover in the next chapter, is handled trivially easily by a recursive descent parser. (*Ad hoc* disambiguating rules are quite often used to handle tricky points in recursive descent parsers).

7.2.9 Eliminating left recursion

Perhaps not quite so obviously, Rule 1 eliminates the possibility of using left recursion to specify syntax. This is a very common way of expressing a repeated pattern of symbols if one is restricted to the use of BNF only. For example, the two productions

$$A \rightarrow A\beta \mid \beta$$

describe the set of sentential forms β , $\beta\beta$, $\beta\beta\beta$ … . These productions contravene Rule 1, because

$$\text{FIRST}(A_1) = \text{FIRST}(A\beta) = \text{FIRST}(A) = \text{FIRST}(A\beta) \cup \text{FIRST}(\beta)$$
$$\text{FIRST}(A_2) = \text{FIRST}(\beta)$$
$$\text{FIRST}(A_1) \cap \text{FIRST}(A_2) \neq \varnothing$$

Direct left recursion can be avoided by using right recursion. Care must be taken, as sometimes the resulting grammar is still unsuitable. For example, the productions above are equivalent to

$$A \rightarrow \beta \mid \beta A$$

but this violates Rule 1 even more clearly. In this case, success may often be achieved by deliberately introducing extra non-terminals. A non-terminal which admits to left recursive productions can always be defined in terms of only two alternative productions, expressed in BNF in the form

$$A \rightarrow A\gamma \mid \delta$$

By expansion we can see that this leads to sentential forms like

$$\delta \ , \ \delta\gamma \ , \ \delta\gamma\gamma \ , \ \delta\gamma\gamma\gamma$$

and these can easily be derived from the equivalent recursive grammar (at the expense of introducing an ε production) given by

$$\begin{aligned} A &\rightarrow \delta B \\ B &\rightarrow \gamma B \mid \varepsilon \end{aligned}$$

The example given earlier is easily dealt with in this way (substitute β for both γ and δ):

$$\begin{aligned} A &\rightarrow \beta B \\ B &\rightarrow \beta B \mid \varepsilon \end{aligned}$$

The reader might complain that the limitation on two alternatives for A is too severe. This is not really true, as suitable factorization can allow γ and δ to have alternatives, none of which starts with A. For example, the set of productions

$$A \rightarrow Ab \mid Ac \mid d \mid e$$

can obviously be recast as

$$\begin{aligned} A &\rightarrow AB \mid C \\ B &\rightarrow b \mid c \\ C &\rightarrow d \mid e \end{aligned}$$

(Indirect left recursion, for example

$$\begin{aligned} A &\rightarrow B \\ B &\rightarrow C \ldots \\ C &\rightarrow A \ldots \end{aligned}$$

is harder to handle, and is, fortunately, not very common in practice.)

All this might not be quite as useful as it first appears. For example, the problem with

Expression = *Expression* "-" *Term* | *Term* .

can readily be removed by using right recursion

Expression = *Term* *RestExpression* .
RestExpression = "-" *Term* *RestExpression* | ε .

but, as we have already noted in Chapter 6, this may have the side effect of altering the implied order of evaluation of an *Expression*. For example, adding the productions

Term = "x" | "y" | "z" .

to the above would mean that with the former production for *Expression*, a string of the form $x - y - z$ would be evaluated as $(x - y) - z$. With the latter production it might be evaluated as $x - (y - z)$, which would result in a very different answer (unless z were zero).

The way to handle this situation and still preserve the desirable left-associativity of the operator seems to be to write the productions to use iteration, as suggested earlier. For example, we might define

Expression = *Term* { "-" *Term* } .

which may even be easier for the reader to follow.

It might be tempting to conclude that the ability to use the iterative constructs in EBNF removes all the problems associated with recursion. But, as we have seen, even here care must be taken. Careful study of this last production will reveal that our manipulations to describe an *Expression* would come to naught if "-" could follow *Expression* in other productions of the grammar.

7.5 Case study - Parva

Toy grammars of the sort we have illustrated in this chapter are all very well, but it is time to consider a rather larger example. What follows is a complete syntactic specification of a small programming language which will be used as the basis for discussion and enlargement at several points in the future, in particular to develop a compiler targeting the PVM of Chapter 4. The language is called Parva (after the Latin feminine adjective for "small"). It deliberately contains only a limited mixture of features drawn from languages like Pascal and C#, and should be immediately comprehensible to programmers familiar with those languages.

7.5.1 EBNF description of Parva

An EBNF description is quite concise.

```
COMPILER Parva $CN
/* Parva level 1 grammar  - Coco/R for C# (EBNF)
   P.D. Terry, Rhodes University, 2003
   Grammar only */

CHARACTERS
  lf         = CHR(10) .
  backslash  = CHR(92) .
  control    = CHR(0) .. CHR(31) .
  letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
               + "abcdefghijklmnopqrstuvwxyz" .
  digit      = "0123456789" .
  stringCh   = ANY - '"' - control - backslash .
  charCh     = ANY - "'" - control - backslash .
  printable  = ANY - control .

TOKENS
  identifier = letter { letter | digit | "_" } .
  number     = digit { digit } .
  stringLit  = '"' { stringCh | backslash printable } '"' .
  charLit    = "'" ( charCh   | backslash printable ) "'" .

COMMENTS FROM "//" TO lf
COMMENTS FROM "/*" TO "*/"
IGNORE CHR(9) .. CHR(13)

PRODUCTIONS
  Parva             = "void" identifier "(" ")" Block .
  Block             = "{" { Statement } "}" .
  Statement         =   Block | ";"
                      | ConstDeclarations | VarDeclarations
                      | Assignment | IfStatement | WhileStatement
                      | ReturnStatement | HaltStatement
                      | ReadStatement | WriteStatement .
  ConstDeclarations = "const" OneConst { "," OneConst } ";" .
  OneConst          = identifier "=" Constant .
  Constant          =   number | charLit | "true" | "false" | "null" .
  VarDeclarations   = Type OneVar { "," OneVar } ";" .
  OneVar            = identifier [ "=" Expression ] .
  Assignment        = Designator "=" Expression ";" .
```

```
  Designator        = identifier [ "[" Expression "]" ] .
  IfStatement       = "if" "(" Condition ")" Statement .
  WhileStatement    = "while" "(" Condition ")" Statement .
  ReturnStatement   = "return" ";" .
  HaltStatement     = "halt" ";" .
  ReadStatement     = "read" "(" ReadElement { "," ReadElement } ")" ";" .
  ReadElement       = stringLit | Designator .
  WriteStatement    = "write" "(" WriteElement { "," WriteElement } ")" ";" .
  WriteElement      = stringLit | Expression .
  Condition         = Expression .
  Expression        = AddExp [ RelOp AddExp ] .
  AddExp            = [ "+" | "-" ] Term { AddOp Term } .
  Term              = Factor { MulOp Factor } .
  Factor            =   Designator | Constant | "!" Factor | "(" Expression ")" .
                      | "new" BasicType "[" Expression "]"
  Type              = BasicType [ "[]" ] .
  BasicType         = "int" | "bool" .
  AddOp             = "+" | "-" | "||" .
  MulOp             = "*" | "/" | "&&" .
  RelOp             = "==" | "!=" | "<" | "<=" | ">" | ">=" .
END Parva.
```

7.5.2 A sample program

It is fairly common practice to illustrate a programming language description with an example of a program illustrating many of the language's features. To keep up with tradition, we follow suit.

```
void main () {
  const votingAge = 18;
  int age, eligible = 0, total = 0;
  bool allEligible = true;
  int[] voters = new int[100];
  read(age);
  while (age > 0) {
    bool canVote = age > votingAge;
    allEligible = allEligible && canVote;
    if (canVote) {
      voters[eligible] = age;
      eligible = eligible + 1;
      total = total + voters[eligible - 1];
    }
    read(age);
  }
  write(eligible, " voters.  Average age ", total/eligible, "\n");
  if (allEligible) write("Everyone was above voting age");
} // main
```

Note that the rather obtuse way in which `eligible` is incremented before being used in a subscripting expression in line 13 is simply to illustrate that a subscript can be a fairly general expression.

Further reading

Good treatments of the material in this chapter may be found at a comprehensible level in the books by Wirth (1976b, 1996), Gough (1988), Parsons (1992), Louden (1997) and Tremblay and Sorenson (1985). Pittman and Peters (1992) have a good discussion of what can be done to transform non-LL(k) grammars into LL(k) ones.

Algorithms exist for the detection and elimination of useless productions. For a discussion of these the reader is referred to the books by Gough (1988), Rechenberg and Mössenböck (1989), and Tremblay and Sorenson (1985).

Our treatment of the LL(1) conditions may have left the reader wondering whether the process of checking them - especially the second one - ever converges for a grammar with anything like the number of productions needed to describe a real programming language. In fact, a little thought should suggest that, even though the number of sentences which they can generate might be infinite, convergence should be guaranteed since the number of productions and the terminal alphabets are finite. The process of checking the LL(k) conditions can be automated and algorithms for doing this, and further discussion of convergence, can be found in the books mentioned above.

8 PARSER AND SCANNER CONSTRUCTION

In this chapter we aim to show how parsers and scanners can be synthesized once appropriate grammars have been written. Our treatment covers the manual construction of these important components of the translation process, as well as an introduction to the use of software tools that help automate the process.

8.1 Construction of simple recursive descent parsers

For the kinds of language that satisfy the rules discussed in the last chapter, parser construction turns out to be remarkably easy. The syntax of these languages is governed by production rules of the form

non-terminal → allowable string

where the allowable string is a concatenation derived from:

- the basic tokens or terminal symbols of the language;
- other non-terminal symbols of the language;
- the actions of metasymbols such as { }, [ ], and | .

We express the effect of applying each production by writing a procedure (or *void method* in C# or Java terminology) to which we give the name of the non-terminal that appears on its left-hand side. The purpose of this routine is to analyze a sequence of terminals, which will be supplied on request from a suitable scanner (lexical analyzer), and to verify that it is of the correct form, reporting errors if it is not. To ensure consistency the routine corresponding to any non-terminal *A*:

- may assume that it has been called *after* some (globally accessible) object *sym* has been created that represents one of the terminals in FIRST(*A*);
- will then parse a complete sequence of terminals which can be derived from *A*, reporting an error if no such sequence is found (in doing this it may have to call on similar routines to handle sub-sequences);
- will relinquish parsing after leaving *sym* representing the first terminal that it finds which cannot be derived from *A* - that is to say, a *sym* should be a member of the set FOLLOW(*A*).

We shall discuss scanners in more detail shortly, but it is as well to have some concept of how a token will be represented. In C# or Java terms it is convenient to define a `Token` class

```
class Token {
  public int kind;    // abstract representation of token
  public string val;  // actual representation in source text
} // class Token
```

In many cases it is only the `kind` field that has any real significance to the higher-level parser, the role of the scanner being simply to map a sequence of source characters, such as `switch` or `while`, to a distinct integer value held in this field. These integer values are often given descriptive internal names such as `switchSym` or `whileSym`. In other cases the representation of a token in the program source cannot be discarded: identifiers such as `DateOfBirth` or numeric literals such as `12345` readily map onto abstract concepts denoted by `identSym` or `numSym`, but the actual spelling of an identifier and the digits in a number retain a deeper significance, and can conveniently be held in the `val` field. This string is sometimes called a **lexeme**.

The shell of each parsing routine follows a pattern suggested by

```
(* A  →  string *)
  BEGIN
    (* we assert  sym.kind ∈ FIRST(A) *)
    Parse(string)
    (* we assert  sym.kind ∈ FOLLOW(A) *)
  END
```

where the transformation *Parse(string)* is governed by the following rules.

(a) If the production yields a single terminal, then the action of *Parse* is to report an error if an unexpected terminal is detected, or (more optimistically) to accept it and then request the next token.

```
Parse (terminal) →
  IF IsExpected(terminal)
    THEN Get(sym)
    ELSE ReportError
  END
```

(b) If we are dealing with a "single" production (that is, one of the form $A = B$), then the action of *Parse* is a simple invocation of the corresponding routine

```
Parse(SingleProduction A) → B
```

This is a rather trivial case, just mentioned here for completeness. Single productions do not really need special mention, except where they arise in the treatment of longer strings, as discussed below.

(c) Very often, a component of the right-hand side of a production is formed of a sequence of terminals and non-terminals. The action is then a sequence derived from (a) and (b), namely

```
Parse (α1 α2 ... αn ) →
  Parse(α1); Parse(α2); ... Parse(αn)
```

(d) If the production allows for a number of non-nullable alternatives, then the action can be expressed as a selection

```
Parse (α1 | α2 | ... αn ) →
  IF sym.kind IN
    FIRST(α1) : Parse(α1);
    FIRST(α2) : Parse(α2);
    ......
    FIRST(αn) : Parse(αn);
    ELSE ReportError
  END
```

in which we see immediately the relevance of Rule 1. In situations where Rule 2 has to be invoked we must go further. To the above we must then add the explicit lack of action to be taken if one of the alternatives of *Parse* is nullable. In that case we must do nothing to advance *sym* - an action which should leave *sym*, as we have seen, as one of the set FOLLOW(*A*). In short, we may sometimes have to augment the above to read

```
Parse (α1 | α2 | ... αn | ε ) →
  IF sym.kind IN
    FIRST(α1) : Parse(α1);
    FIRST(α2) : Parse(α2);
    ......
    FIRST(αn) : Parse(αn);
    FOLLOW(A) :  (* do nothing *)
    ELSE ReportError
  END
```

(e) If a component of a production allows for a nullable option, the transformation involves a decision

```
Parse ( [ α ] ) →
  IF sym.kind ∈ FIRST(α) THEN Parse(α) END
```

(f) If a component of a production allows for possible repetition, the transformation involves a loop, often of the form

```
Parse ( { α } ) →
  WHILE sym.kind ∈ FIRST(α) DO Parse(α) END
```

Note the importance of Rule 2 here again. Some repetitions are of the form

$$\alpha \{ \alpha \}$$

which transforms to

```
Parse(α);  WHILE sym.kind ∈ FIRST(α) DO Parse(α)  END
```

On occasions this may be better expressed

```
REPEAT  Parse(α)  UNTIL sym.kind ∉ FIRST(α)
```

8.2 Case study - a parser for assignment sequences

To illustrate these ideas further, let us consider a concrete example. The grammar below, chosen to illustrate the various options discussed above, describes a sequence of assignment statements separated one from the next by semicolons and terminated with a period, as exemplified by

```
Mark[Top] = 100;
CanSleepLate = (Saturday || Sunday) && EssayCompleted;
PassedAll = Passed[Test1] && Passed[Test2] && Passed[Exam].
```

The grammar is described in Cocol as follows.

```
COMPILER Sequence
/* Grammar for a sequence of Boolean assignment statements */

CHARACTERS
  letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
               + "abcdefghijklmnopqrstuvwxyz" .
  digit      = "0123456789" .

TOKENS
  identifier = letter { letter | digit } .
  number     = digit { digit } .

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Sequence   = Assignment { ";" Assignment } "." .
  Assignment = Variable "=" Expression .
  Variable   = identifier [ "[" Expression "]" ] .
  Expression = Term { "||" Term } .
  Term       = Factor { "&&" Factor } .
  Factor     = Variable | number | "(" Expression ")" .

END Sequence.
```

While the grammar introduces operators that appear to be Boolean in nature there are, of course, no constraints that these have been used correctly - it says nothing about the type of the *Variable* operands, or whether any sensible meaning could be associated with an assignment like

```
X[A && B] = 123 || 34;
```

A first draft of the parsing routines, developed from the earlier suggestions, follows.

```
static void Sequence() {
// Sequence = Assignment { ";" Assignment } "."
  Assignment();
  while (sym.kind == semicolonSym) {
    Accept(semicolonSym, "; expected");
    Assignment();
  }
  Accept(periodSym, ". expected");
} // Sequence

static void Assignment() {
// Assignment = Variable "=" Expression .
  Variable();
  Accept(assignSym, "= expected");
  Expression();
} // Assignment
```

```
static void Variable() {
// Variable = identifier [ "[" Expression "]" ] .
  Accept(identSym, "identifier expected");
  if (sym.kind == lBrackSym) {
    Accept(lBrackSym, "[ expected");
    Expression();
    Accept(rBrackSym, "] expected");
  }
} // Variable

static void Expression() {
// Expression = Term { "||" Term } .
  Term();
  while (sym.kind == orSym) {
    Accept(orSym, "|| expected");
    Term();
  }
} // Expression

static void Term() {
// Term = Factor { "&&" Factor } .
  Factor();
  while (sym.kind == andSym) {
    Accept(andSym, "&& expected");
    Factor();
  }
} // Term

static void Factor() {
// Factor = Variable | number | "(" Expression ")" .
  switch (sym.kind) {
    case identSym:
      Variable();
      break;
    case numSym:
      Accept(numSym, "number expected");
      break;
    case lParenSym:
      Accept(lParenSym, "( expected");
      Expression();
      Accept(rParenSym, ") expected");
      break;
    default:
      Abort("Invalid start to Factor");
      break;
  }
} // Factor
```

Besides the routines corresponding to each non-terminal it is necessary to provide some auxiliary methods to deal with the acceptance or rejection of any incorrect terminals that are encountered. Code for these might be as follows.

```
static void ReportError(string errorMessage) {
// Displays errorMessage on standard output and reflected output
  Console.WriteLine(errorMessage);
  output.WriteLine(errorMessage);
} // ReportError

static void Abort(string errorMessage) {
// Abandons parsing after issuing error message
  ReportError(errorMessage);
  output.Close();
  System.Environment.Exit(1);
} // Abort

static void Accept(int wantedSym, string errorMessage) {
// Checks that lookahead sym.kind is wantedSym
  if (sym.kind == wantedSym) GetSym(); else Abort(errorMessage);
} // Accept
```

where we have decided simply to abort the process completely (giving a reason) if the input is incorrect.

Examination of the code will show that we have used the method `GetSym()` to represent the scanner. For this first example, the various token kinds can adequately be represented by distinctive integer values, enumerated on the lines of

```
const int
  noSym     = 0,
  assignSym = 1,
  lBrackSym = 2,
   ...
  EOFSym    = 12;
```

In this enumeration, `noSym` is the value used to denote that a token could be detected, but could not be recognized. In contrast to this, `EOFSym` is used to denote that no further token could be detected in the input at all.

A simple driver for a complete system might then take the form below. Note that it is necessary to call the scanner to provide the initial lookahead token before the first parser routine is called, in accordance with the principles established in the last section.

```
public static void Main(string[] args) {
  // Open input and output files from command line arguments
  input = new InFile(args[0]);
  output = new OutFile(newFileName(args[0], ".out"));
  GetChar();       // Lookahead character
  GetSym();        // Lookahead token
  Sequence();      // Start to parse from the goal symbol
  // If we get back here everything must have been satisfactory
  Console.WriteLine("Parsed correctly");
  output.Close();
} // Main
```

It should now be clear why this method of parsing is called *recursive descent*, and that such parsers might easily be implemented in any of the many imperative languages which directly support recursive programming. With those that strictly require "declaration before use" (like Pascal), care will have to be taken with the relative ordering of the declaration of the parsing methods which in this example, and in general, are recursive in nature. If one were developing this parser in C++, for example, it would be necessary to use the concept of the "function prototype", at least for the routine for *Expression*.

Languages like Modula-2, C# and Java are all very well suited to the task, although they each have their own particular strengths and weaknesses. For example, in Modula-2 one can take advantage of other organizational strategies, such as the use of nested procedures (which are not permitted in C# or Java) and the very tight control offered by encapsulating a parser in a module with a very thin interface (only the routine for the goal symbol need be exported), while in C# and Java one can take advantage of object-orientation (both to encapsulate the parser in a class with a thin public interface and to create hierarchies of specialized parser classes).

8.3 Other aspects of recursive descent parsing

A little reflection on the code just illustrated should suggest that, although we have simply rigorously followed the guidelines suggested in section 8.1, it is not always necessary to invoke the `Accept` method each time a terminal appears in a sentential form. Several of the calls to `Accept` appear in contexts where they cannot possibly fail, and in such situations it makes sense to substitute a simple call to `GetSym`. An example of this is provided by the `Term` method, which is more efficiently written as

```
static void Term() {
// Term = Factor { "&&" Factor } .
  Factor();
  while (sym.kind == andSym) {
    GetSym();  // no further test; we already know it is an andSym
    Factor();
  }
} // Term
```

Code for a complete program (incorporating such refinements) can be found in the *Resource Kit*.

While the discussion in Section 8.1 made explicit use of FIRST sets, the reader may have noticed that the code given in the last example seems not to have done so at all. The reason is that the FIRST sets in each case happen to be singleton sets (that is, have one member only) so that a set membership test is easily achieved by a test for equality. For more extensive grammars it may make better sense to develop recursive descent parsers that make explicit use of larger FIRST and FOLLOW sets.

In languages like Modula-2 and Pascal, where set operations are directly supported, implementation of these ideas is very straightforward. C# and Java do not have "built-in" set types. Their implementation in terms of a user-

defined class is easily achieved. Part of the specification of such a class, suited to the C# applications in this text, is summarized below, and further details can be found in Appendix B and in the source code in the *Resource Kit*.

```
class IntSet {
  public IntSet()                          // Empty set constructor
  public IntSet(params int[] members)      // General constructor
  public void Incl(int i)                  // Set inclusion
  public void Excl(int i)                  // Set exclusion
  public bool Contains(int i)              // Set membership test
  public bool Contains(IntSet that)        // true if that is a subset of this set
  public bool IsEmpty()                    // true if empty set
  public IntSet Union(IntSet that)         // Returns new set this OR that
  public IntSet Intersection(IntSet that) // Returns new set this AND that
  public IntSet Difference(IntSet that)    // Returns new set this - that
  public IntSet SymDiff(IntSet that)       // Returns new set this XOR that
} // class IntSet
```

Following on from this, note that we can conveniently provide an overloading of the `Accept` method introduced in the last section.

```
static void Accept(IntSet acceptable, string errorMessage) {
// Checks that lookahead sym.kind is in acceptable set of tokens
  if (acceptable.Contains(sym.kind)) GetSym();
  else Abort(errorMessage);
} // Accept
```

To illustrate how sets might be used in practice, suppose the productions in our grammar were changed to

```
Sequence   = Assignment { ";" Assignment } "." .
Assignment = Variable "=" Relation .
Variable   = identifier [ "[" Expression "]" ] .
Relation   = Expression [ RelOp Expression ] .
Expression = Term { "||" Term } .
Term       = Factor { "&&" Factor } .
Factor     = Variable | number | "(" Expression ")" .
RelOp      = "==" | "!=" | "<" | "<=" | ">" | ">=" .
```

A parsing routine for *Relation* might then take the form

```
static void Relation() {
// Relation = Expression [ RelOp Expression ] .
  Expression();
  if (FirstRelOp.Contains(sym.kind)) {
    GetSym();   // no further test needed
    Expression();
  }
} // Relation
```

where we would have initialized `FirstRelOp` within the parser class as

```
static IntSet FirstRelOp = new IntSet(eqlSym, neqSym, lssSym, leqSym, gtrSym, geqSym);
```

In this case we need not provide a separate parsing routine for the non-terminal *RelOp*, since it appears on the right-hand side of one production only, and it is itself defined in terms of a choice of single terminals. Frequently, simplifications of this sort can be introduced into recursive descent parsers in what amounts technically to writing a parser for an equivalent grammar, rather than for the original grammar.

Other modifications of the basic suggestions are sometimes worth consideration. To illustrate one of these, suppose the production for our goal symbol were replaced by

Sequence = { *Assignment* ";" } EOF .

There is a very natural temptation to write the parsing routine as

```
static void Sequence() {
  while (sym.kind != EOFSym) {
    Assignment();
    Accept(semicolonSym, "; expected");
  }
} // Sequence
```

While this goes against the spirit of the suggestion that recursive descent parsing should be driven by tokens delivered "from the left" rather than by anticipating what will eventually come "on the extreme right", it can be argued that it might be more effective, especially when one develops techniques for recovering from syntactic errors, rather than simply aborting the process. If we were to follow the earlier suggestions we should, of course, come up with a routine more like

```
static void Sequence() {
  while (sym.kind == identSym) {
    Assignment();
    Accept(semicolonSym, "; expected");
  }
  Accept(EOFSym, "EOF expected");
} // Sequence
```

Acceptance of `EOFSym` might seem a strange thing to suggest - how might any token be found to follow it? The scanner must, of course, be programmed simply to return `EOFSym` again after it is first detected.

Although recursive descent parsers are eminently suitable for handling languages which satisfy the LL(1) conditions, they may often be used, perhaps with simple modifications, to handle languages which, strictly, do not satisfy these conditions. The classic example of a situation like this is provided by the `IF ... THEN ... ELSE` statement. Suppose we have a language in which statements are defined by

Statement = *IfStatement* | *OtherStatement* .
IfStatement = "IF" *Condition* "THEN" *Statement* ["ELSE" *Statement*] .

which, as we have already discussed, is actually ambiguous as it stands. A grammar defined like this is easily parsed deterministically with code like

```
static void Statement() {
  switch(sym.kind) {
    case ifSym : IfStatement(); break;
    default    : OtherStatements(); break;
  }
} // Statement

static void IfStatement() {
  GetSym(); Condition();
  accept(thenSym, "THEN expected");
  Statement();
  if (sym.kind == elseSym) { GetSym(); Statement(); }
} // IfStatement

static void OtherStatement()
// Handle parsing of other statements - not necessary to show this in detail
```

The reader who cares to trace the function calls for an input sentence of the form

`IF` *Condition* `THEN` `IF` *Condition* `THEN` *OtherStatement* `ELSE` *OtherStatement*

will note that this parser recognizes and handles an `ELSE` clause as soon as it can - effectively forcing an *ad hoc* resolution of the ambiguity by coupling each `ELSE` to the closest unmatched `THEN`. Indeed, it would be far more difficult to design a parser that implemented the other possible disambiguating rule - no wonder that the semantics of this statement are those which correspond to the solution that becomes easy to parse!

In earlier sections we discussed the development of equivalent grammars in some detail, pointing out that the careful choice of the best alternative was crucial to the successful development of a parser. As a further example that highlights some practical difficulties in arriving at the best choice, consider how one might try to describe variable designators of the kind found in many languages to denote elements of classes, records, structures and arrays, possibly in combination, for example `A[B.C.D]`. One set of productions that describes some (although by no means all) of these constructions might appear to be:

Designator	=	*identifier Qualifier* .	(1)
Qualifier	=	*Subscript* | *FieldSpecifier* .	(2, 3)
Subscript	=	"[" *Designator* "]" | ε .	(4, 5)
FieldSpecifier	=	"." *Designator* | ε .	(6, 7)

This grammar is not LL(1); it is actually ambiguous. At first it might appear that all is well, since

FIRST(*Qualifier*$_1$) = FIRST(*Subscript*) = { "[" }
FIRST(*Qualifier*$_2$) = FIRST(*FieldSpecifier*) = { "." }

but we must be more careful. As we have expressed the grammar, both *Subscript* and *FieldSpecifier* are nullable. A better approach for those keen to see how the LL(1) rules might react to this is to use the Director Set concept discussed in section 7.2.10, when it will be found that

DS(*Qualifier, Subscript*) = FIRST(*Subscript*) ∪ FOLLOW(*Qualifier*) = { "[" , "]" }
DS(*Qualifier, FieldSpecifier*) = FIRST(*FieldSpecifier*) ∪ FOLLOW(*Qualifier*) = { "." , "]" }

These two director sets have "]" in common, and so the grammar is non-LL(1) (of course).

The reader will probably complain that this looks ridiculous and immediately suggest rewriting the grammar in the form

Designator	=	*identifier Qualifier* .	(1)
Qualifier	=	*Subscript* | *FieldSpecifier* | ε .	(2, 3, 4)
Subscript	=	"[" *Designator* "]" .	(5)
FieldSpecifier	=	"." *Designator* .	(6)

This leads easily to an LL(1) grammar (readers should verify this to their own satisfaction). Once again, a recursive descent parser is easily written, probably on the lines of

```
static void Designator() {
  Accept(identSym, "identifier expected");
  Qualifier();
} // Designator

static void Qualifier() {
  switch (sym.kind) {
    case lBrackSym : Subscript(); break;
    case periodSym : FieldSpecifier(); break;
    case rBrackSym : break; // FOLLOW(Qualifier) is { "]" }
    default :        Abort("unrecognizable Qualifier");
  }
} // Qualifier

static void Subscript() {
  GetSym(); Designator();
  Accept(rBrackSym, "] expected");
} // Subscript

static void FieldSpecifier() {
  GetSym(); Designator();
} // FieldSpecifier
```

but even this may not really be what the language requires. Since *Qualifier* is optional, the best grammar yet would be

Designator	=	*identifier* [*Subscript* | *FieldSpecifier*] .
Subscript	=	"[" *Designator* "]" .
FieldSpecifier	=	"." *Designator* .

which suggests that the method for *Designator* should be

```
static void Designator() {
  Accept(identSym, "identifier expected");
  switch (sym.kind) {
    case lBrackSym : Subscript(); break;
    case periodSym : FieldSpecifier(); break;
    default :        break;
  }
} // Designator
```

In this case there are easy, if not even obvious ways to define a suitable grammar, and to develop the parser. However, a more realistic version of this problem leads to a situation that cannot as easily be resolved. In

Modula-2 a *Designator* is better described by the productions

```
Designator          = QualifiedIdentifier { Selector } .
QualifiedIdentifier = identifier { "." identifier } .
Selector            = "." identifier | "[" Expression "]" | "↑" .
```

It is left as an exercise to demonstrate that this is not LL(1). It is left as a harder exercise to come to a formal conclusion that one cannot find an LL(1) grammar that describes *Designator* as well as we would like. The underlying reason is that "." is used in one context to separate a module identifier from the identifier that it qualifies (as in `Scanner.sym`) and in a different context to separate a record identifier from a field identifier (as in `sym.kind`). When these are combined (as in `Scanner.sym.kind`) the problem becomes more obvious. The problem exists in Java and C# too, of course, in designators like `NameSpace.Class.Field`.

The reader may have wondered at the fact that the parsing methods we have advocated all look "ahead", and never seem to make use of what has already been achieved - that is, of information which has become embedded in the previous history of the parse. All LL(1) grammars are, of course, context-free, yet we pointed out in Chapter 6 that there are features of programming languages which cannot be specified in a context-free grammar (such as the requirement that variables must be declared before use, and that expressions may only be formed when terms and factors are of the correct types). In practice, of course, a parser is usually combined with a semantic analyzer - in a sense some of the past history of the parse is recorded in such devices as symbol tables which the semantic analysis needs to maintain. The examples given here are not as serious as may at first appear. By making recourse to the symbol table, a compiler will be able to resolve the potential ambiguity in a static semantic way (rather than in an *ad hoc* syntactic way as is done for the "dangling else" situation).

8.4 Syntax error detection and recovery

Up to this point our parsers have been content merely to stop when a syntactic error is detected. In the case of a real compiler this is probably unacceptable. However, if we modify the parser as given above so as simply not to stop after detecting an error, the result is likely to be chaotic. The analysis process will quickly get out of step with the sequence of tokens being scanned, and in all likelihood will then report a plethora of spurious errors.

One useful feature of the compilation technique we are using is that the parser can detect a syntactically incorrect structure after being presented with its first "unexpected" terminal. This will not necessarily be at the point where the error really occurred. For example, if it were presented with the sequence

```
{ if (A > 6) else B = 2; C = 5 } }
```

we might hope that a C# compiler would give a sensible error message when `else` is found where a statement is expected. Even if parsing does not get out of step, we would probably get a less helpful message when the second } is found - the compiler can have little idea where a missing { might have occurred, or whether a superfluous closing brace should be deleted.

A production quality compiler should aim to issue appropriate diagnostic messages for all the "genuine" errors, and for as few "spurious" errors as possible. This is only possible if it can make some likely assumption about the nature of each error and the probable intention of the author, or if it skips over some part of the malformed text, or both. Various approaches can be made to handling the problem. Some compilers go so far as to try to correct the error and continue to produce object code for the program. Error correction is a little dangerous, except in some trivial cases, and we shall discuss it no further here. Many systems confine themselves to attempting **error recovery**, which is the term used to describe the process of simply trying to get the parser back into step with the source code presented to it. The art of doing this for handcrafted compilers is rather intricate and relies on a mixture of fairly well-researched techniques and intuitive experience, both of the language being compiled and of the potential misconceptions of its users.

Since recursive descent parsers are constructed as a set of routines, each of which tackles a sub-goal on behalf of its caller, fairly obvious places to try to regain lost synchronization are at the entry to and exit from these routines, where the effects of getting out of step can be confined to examining a small range of known FIRST and FOLLOW tokens. To enforce synchronization at the entry to the routine for a non-terminal A we might try to employ a strategy like

```
IF sym.kind ∉ FIRST(A) THEN
  ReportError; SkipTo(FIRST(A))
END
```

where *SkipTo* is an operation which simply calls on the scanner until it returns a value for *sym* that is a member of FIRST(*A*). Unfortunately this is not quite adequate - if the leading terminal has been omitted we might then skip over tokens that should have been processed later, that is, by the routine which called *A*.

At the exit from *A*, we have postulated that *sym* should be a member of FOLLOW(*A*). This set may not be known to *A* but it should be known to the routine which calls *A*, so that it may conveniently be passed to *A* as a parameter. This suggests that we might employ a strategy like

```
IF sym.kind ∉ FOLLOW(A) THEN
  ReportError; SkipTo(FOLLOW(A))
END
```

The use of FOLLOW(*A*) also allows us to avoid the danger mentioned earlier of skipping too far at routine entry, by developing parsing routines on the lines of

```
IF sym.kind ∉ FIRST(A) THEN
  ReportError; SkipTo(FIRST(A) ∪ FOLLOW(A))
END;
IF sym.kind ∈ FIRST(A) THEN
  Parse(A);
  IF sym.kind ∉ FOLLOW(A) THEN
    ReportError; SkipTo(FOLLOW(A))
  END
END
```

Although the FOLLOW set for a non-terminal is quite easy to determine, the legitimate follower may itself have been omitted, and this may lead to too many tokens being skipped at routine exit. To prevent this, a parser using this approach usually passes to each sub-parser a *followers* parameter which is constructed so as to include:

- the minimally correct set FOLLOW(*A*); augmented by
- tokens that have already been passed as *followers* to the calling routine (that is, tokens that require investigation after the *calling* routine itself completes); and also
- so-called **beacons** which are on no account to be passed over, even though their presence might be quite out of context. In this way the parser can often avoid skipping large sections of possibly important code.

On return from sub-parser *A* we can then be fairly certain that *sym* represents a terminal which was either expected (if it is in FOLLOW(*A*)), or can be used to regain synchronization (if it is one of the beacons, or is in FOLLOW(*Caller(A)*). The caller may need to make a further test to see which of these conditions has arisen.

Syntax error recovery is then conveniently implemented by defining methods on the lines of

```
static bool errors = false;

static void ReportError(string errorMessage) {
  errors = true;
  Console.WriteLine(errorMessage);
  output.WriteLine(errorMessage);
} // ReportError

static void Test(IntSet allowed, IntSet beacons, string errorMessage) {
  if (allowed.Contains(sym.kind)) return;
  ReportError(errorMessage);
  IntSet stopSet = allowed.Union(beacons);
  while (!stopSet.Contains(sym.kind)) GetSym();
} // Test

static void Accept(int wantedSym, string errorMessage) {
  if (sym.kind == wantedSym) GetSym();
  else ReportError(errorMessage);
} // Accept
```

where we note that the amended *Accept* routine does not try to regain synchronization in any way.

One must not be overenthusiastic in applying this technique. In the code below we show how it can be used to improve the parser developed in section 8.2. It is really necessary only to incorporate calls to *Test* into the subparsers for *Assignment* and *Factor*. Parsing of an *Expression* always results in calls down the hierarchy to the parser for *Factor*, which thus forms a convenient focus point for synchronization.

```
static IntSet
  FirstFactor     = new IntSet(identSym, numSym, lParenSym),
  FirstAssignment = new IntSet(identSym);

static void Sequence(IntSet followers) {
// Sequence = Assignment { ";" Assignment } "."
  IntSet punctuate = new IntSet(semicolonSym, periodSym);
  Assignment(followers.Union(punctuate));
  while (sym.kind == semicolonSym) {
    GetSym();
    Assignment(followers.Union(punctuate));
  }
  Accept(periodSym, ". expected");
} // Sequence

static void Assignment(IntSet followers) {
// Assignment = Variable "=" Expression .
  Test(FirstAssignment, followers, "Invalid start to Assignment");
  if (FirstAssignment.Contains(sym.kind)) {
    Variable(followers.Union(new IntSet(assignSym)));
    Accept(assignSym, "= expected");
    Expression(followers);
    Test(followers, new IntSet(), "Invalid sym after Assignment");
  }
} // Assignment

static void Variable(IntSet followers) {
// Variable = identifier [ "[" Expression "]" ] .
  Accept(identSym, "identifier expected");
  if (sym.kind == lBrackSym) {
    GetSym();
    Expression(followers.Union(new IntSet(rBrackSym)));
    Accept(rBrackSym, "] expected");
  }
} // Variable

static void Expression(IntSet followers) {
// Expression = Term { "||" Term } .
  Term(followers.Union(new IntSet(orSym)));
  while (sym.kind == orSym) {
    GetSym();
    Term(followers.Union(new IntSet(orSym)));
  }
} // Expression

static void Term(IntSet followers) {
// Term = Factor { "&&" Factor } .
  Factor(followers.Union(new IntSet(andSym)));
  while (sym.kind == andSym) {
    GetSym();
    Factor(followers.Union(new IntSet(andSym)));
  }
} // Term

static void Factor(IntSet followers) {
// Factor = Variable | number | "(" Expression ")" .
  Test(FirstFactor, followers, "Invalid start to Factor");
  if (FirstFactor.Contains(sym.kind)) {
    switch (sym.kind) {
      case identSym:
        Variable(followers);
        break;
      case numSym:
        GetSym();
        break;
      case lParenSym:
        GetSym();
        Expression(followers.Union(new IntSet(rParenSym)));
        Accept(rParenSym, ") expected");
        break;
    }
    Test(followers, new IntSet(), "Invalid sym after Factor");
  }
} // Factor
```

```
// +++++++++++++++++++++ Main driver function ++++++++++++++++++

public static void Main(string[] args) {
  // Open input and output files from command line arguments
  input = new InFile(args[0]);
  output = new OutFile(newFileName(args[0], ".out"));
  GetChar();                    // Lookahead character
  GetSym();                     // Lookahead token
  Sequence(new IntSet(EOFSym)); // Start to parse from goal symbol
  if (!errors) Console.WriteLine("Parsed correctly");
  output.Close();
} // Main
```

There are many variations on using these ideas, and in handcrafted parsers they are often adapted in other ways. For example, it might not always be deemed necessary to attempt synchronization at the start of a parser routine but to leave this until the end. An acceptable variation on the code just shown might be to simplify the parsing routine for *Assignment*

```
static void Assignment(IntSet followers) {
// Assignment = Variable "=" Expression .
  Variable(followers.Union(new IntSet(assignSym)));
  Accept(assignSym, "= expected");
  Expression(followers);
  Test(followers, new IntSet(), "Invalid sym after Assignment");
} // Assignment
```

A similar modification could be made to the routine for *Factor*

```
static void Factor(IntSet followers) {
// Factor = Variable | number | "(" Expression ")" .
  switch (sym.kind) {
    case identSym:
      Variable(followers);
      break;
    case numSym:
      GetSym();
      break;
    case lParenSym:
      GetSym();
      Expression(followers.Union(new IntSet(rParenSym)));
      Accept(rParenSym, ") expected");
      break;
    default:
      ReportError("Invalid start to Factor");
      break;
  }
  Test(followers, new IntSet(), "Invalid sym after Factor");
} // Factor
```

However, notice that attempts to synchronize before handling alternatives must be done carefully in cases where one of the alternatives may legitimately be empty.

As mentioned earlier, one gains from experience when dealing with learners, and some concession to likely mistakes is, perhaps, a good thing. For example, beginners are likely to confuse operators like `":="`, `"="` and `"=="`, and also `THEN` and `DO` after `IF`, and these may call for special treatment.

In a production like that for *Sequence* the semicolon provides an example of a so-called *weak separator* in an iterative construct. In the alternative parsing routine below, the iteration is started as long as the parser detects the presence of the weak separator *or* a valid symbol that would follow it in that context (of course, appropriate errors are reported if the separator has been omitted). This has the effect of "inserting" missing semicolons into the stream of tokens being parsed. Treating weak separators in this way in the many places where they are found in grammars proves to be a highly effective enhancement to the basic technique.

```
static IntSet
  assignSynchSet = new IntSet(semicolonSym, identSym, periodSym),
  continueAssign = new IntSet(semicolonSym, identSym);
```

```
static void Sequence(IntSet followers) {
// Sequence = Assignment { ";" Assignment } "."
  Assignment(followers.Union(AssignSynchSet));
  while (continueAssign.Contains(sym.kind)) {
    Accept(semicolonSym, "; expected");
    Assignment(followers.Union(AssignSynchSet));
  }
  Accept(periodSym, ". expected");
} // Sequence
```

Clearly it may be impossible to recover cleanly from all possible incorrect contortions of code, but one should guard against the cardinal sins of not reporting errors when they are present, or of collapsing completely when trying to recover from an error, either by giving up prematurely or by getting the parser caught in an infinite loop reporting the same error.

The code above deliberately highlights a weakness of the follower-set approach to error recovery, namely that it is quite expensive. Each call to a parsing routine is effectively preceded by two time-consuming operations - the dynamic construction of a set object, and the parameter-passing operation itself - which turn out not to have been required if the source being translated is correct. It can be improved considerably if as many as possible of the sets are constructed once only and held in static variables.

Further reading

Error recovery is an extensive topic, and we shall have more to say on it in later chapters. Good treatments of the material of this section can be found in the books by Wirth (1976b, 1986, 1996), Gough (1988) and Elder (1994). Papers by Pemberton (1980), Topor (1982), Stirling (1985) and Grosch (1990) are also worth exploring, as is the bibliographical review article by van den Bosch (1992).

8.5 Construction of simple scanners

In a sense, a scanner or lexical analyzer can be thought of as just another syntax analyzer. It handles a grammar with productions relating non-terminals such as *identifier*, *number* and *Relop* to terminals supplied, in effect, as single characters of the source text. When used in conjunction with a higher-level parser a subtle shift in emphasis comes about - there is, in effect, no special goal symbol. Each invocation of the scanner ends when it has reduced a string of characters to a token, without preconceived ideas of what that should be. These tokens or non-terminals are then regarded as terminals by the higher-level recursive descent parser that analyzes the phrase structure of *Block*, *Statement*, *Expression* and so on.

There are at least five reasons for wishing to decouple the scanner from the main parser.

- The productions involved are usually very simple. Very often they amount to regular expressions, and then a scanner can be programmed without recourse to methods like recursive descent.
- A keyword token like a reserved word is lexically equivalent to an *identifier* - the distinction may sensibly be made as soon as the basic token has been synthesized.
- The character set may vary from machine to machine, a variation easily isolated in this phase.
- The semantic analysis of a numeric literal constant (deriving the internal representation of its value from the characters) may conveniently be performed in parallel with lexical analysis.
- The scanner can be made responsible for screening out superfluous separators, like blanks and comments, which are rarely of interest in the formulation of the higher-level grammar.

In common with the parsing strategy suggested earlier, development of the routine or method responsible for token recognition:

- may assume that it is always called *after* some (globally accessible) variable *ch* has been arranged to contain the next character to be handled in the source;
- will then read the longest complete sequence of characters that form a recognizable token - so, for

example, != will be treated as a "not equal" relational operator, rather than as two tokens "not" and "becomes";

- will relinquish scanning after leaving *ch* with the first character that does not form part of this token - so as to satisfy the precondition for the next invocation of the scanner.

A scanner is necessarily a top-down parser. For ease of implementation it may seem desirable that the productions defining the token grammar also obey the LL(1) rules. However, token grammars are almost invariably regular, do not display self-embedding, and lend themselves to the construction of simple scanners even if they are not themselves LL(1) compliant.

There are two main strategies that are employed in scanner construction.

- Rather than being decomposed into a set of possibly recursive routines, simple scanners are often written in an *ad hoc* manner, controlled by a large `switch` statement, since the underlying task is one of choosing between a number of tokens which are often distinguishable on the basis of their initial characters.

- Alternatively, since they frequently have to read a number of characters, scanners are often written in the form of a **finite state automaton** (FSA) controlled by a loop, on each iteration of which a single character is absorbed, the machine moving between a number of "states" determined by the character just read. This approach has the advantage that the construction can be formalized in terms of an extensively developed automata theory, leading to algorithms from which scanner generators can be constructed automatically.

A proper discussion of automata theory is beyond the scope of this text, but in the next section we shall demonstrate both approaches to scanner construction by means of some case studies.

8.6 Case study - a scanner for Boolean assignments

To consider a concrete example, suppose that we wish to write a scanner to complement a parser for a language described in Cocol as follows.

```
COMPILER Assignment
/* Grammar describing a simple assignment */

CHARACTERS
  letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
               + "abcdefghijklmnopqrstuvwxyz" .
  digit      = "0123456789" .

TOKENS
  identifier = letter { letter | digit } .
  number     = digit { digit } .

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Assignment = identifier "=" Relation .
  Relation   = Operand [ Operator Operand ] .
  Operand    =   identifier | number
               | "!" Operand | "(" Relation ")" .
  Operator   =   "==" | "!=" | "<" | "<=" | ">" | ">=" | "&&" | "||" .
END Assignment.
```

The token kinds we need are enumerable as

```
KINDS = { noSym, EOFSym, lssSym, leqSym, gtrSym, geqSym,
          assignSym, eqlSym, notSym, neqSym, andSym, orSym,
          lParenSym, rParenSym, numSym, identSym }
```

It should be easy to see that these tokens are not uniquely distinguishable on the basis of their leading characters alone, but it is not difficult to write a set of productions for the token grammar that obeys the LL(1) rules.

```
kind  =  EOF                          (* EOFSym *)
       | "<" [ "=" ]                  (* lssSym, leqSym *)
       | ">" [ "=" ]                  (* gtrSym, geqSym *)
       | "=" [ "=" ]                  (* assignSym, eqlSym *)
       | "!" [ "=" ]                  (* notSym, neqSym *)
       | "&" "&"                      (* andSym *)
       | "|" "|"                      (* orSym *)
       | "("                          (* lParenSym *)
       | ")"                          (* rParenSym *)
       | digit { digit }              (* numSym *)
       | letter { letter | digit }    (* identSym *) .
```

from which an *ad hoc* scanner method follows very easily on the lines shown below. For simplicity we have assumed that there is a distinctive EOF character marking the end of the input.

```
static void GetSym() {
// Scans for next sym from input
  while (ch != EOF && ch <= ' ') GetChar();  // skip whitespace
  StringBuilder symLex = new StringBuilder();
  int symKind = noSym;
  if (Char.IsLetter(ch)) {
    do {
      symLex.Append(ch); GetChar();
    } while (Char.IsLetterOrDigit(ch));
    symKind = identSym;
  }
  else if (Char.IsDigit(ch)) {
    do {
      symLex.Append(ch); GetChar();
    } while (Char.IsDigit(ch));
    symKind = numSym;
  }
  else {
    symLex.Append(ch);
    switch (ch) {
      case EOF:
        symLex = new StringBuilder("EOF"); // special case
        symKind = EOFSym; break;           // no need to GetChar
      case '<':
        symKind = lssSym; GetChar();
        if (ch == '=') {
          symLex.Append(ch); symKind = leqSym; GetChar();
        }
        break;
      case '>':
        symKind = gtrSym; GetChar();
        if (ch == '=') {
          symLex.Append(ch); symKind = geqSym; GetChar();
        }
        break;
      case '=':
        symKind = assignSym; GetChar();
        if (ch == '=') {
          symLex.Append(ch); symKind = eqlSym; GetChar();
        }
        break;
      case '!':
        symKind = notSym; GetChar();
        if (ch == '=') {
          symLex.Append(ch); symKind = neqSym; GetChar();
        }
        break;
      case '|':
        symKind = noSym; GetChar();
        if (ch == '|') {
          symLex.Append(ch); symKind = orSym; GetChar();
        }
        break;
      case '&':
        symKind = noSym; GetChar();
        if (ch == '&') {
          symLex.Append(ch); symKind = andSym; GetChar();
        }
        break;
```

```
        case '(':
          symKind = lParenSym; GetChar(); break;
        case ')':
          symKind = rParenSym; GetChar(); break;
        default:
          symKind = noSym; GetChar(); break;
      }
    }
    sym = new Token(symKind, symLex.ToString());
  } // GetSym
```

Here we have assumed that each call to the method `GetChar()` will store in *ch* the next character in the source text and that a distinctive `EOF` character will be returned repeatedly once the end of the source text is reached.

A characteristic feature of this algorithm - and of most scanners constructed in this way - is that they are governed by a selection statement, within the alternatives of which sometimes finds loops that consume sequences of characters. To illustrate the FSA approach - in which the algorithm is inverted to be governed by a single loop - let us write our grammar in a slightly different way, in which the comments have been placed to reflect the state that a scanner can be thought to have reached at the point where a character has just been read.

```
kind  =  (* noSym *) EOF (* EOFSym *)
       | (* noSym *) "<" (* lssSym *) [ "=" (* leqSym *) ]
       | (* noSym *) ">" (* gtrSym *) [ "=" (* geqSym *) ]
       | (* noSym *) "=" (* assignSym *) [ "=" (* eqlSym *) ]
       | (* noSym *) "!" (* notSym *) [ "=" (* neqSym *) ]
       | (* noSym *) "&" (* noSym *) "&" (* andSym *)
       | (* noSym *) "|" (* noSym *) "|" (* orSym *)
       | (* noSym *) "(" (* lParenSym *)
       | (* noSym *) ")" (* rParenSym *)
       | (* noSym *) digit (* numSym *) { digit (* numSym *) }
       | (* noSym *) letter (* identSym *) { ( letter | digit ) (* identSym *) }
```

Another way of representing this information is in terms of a transition diagram like that shown in Figure 8.1, where, as is customary, the states have been labelled with small integers, and where the arcs are labelled with the characters whose recognition causes the automaton to move from one state to another.

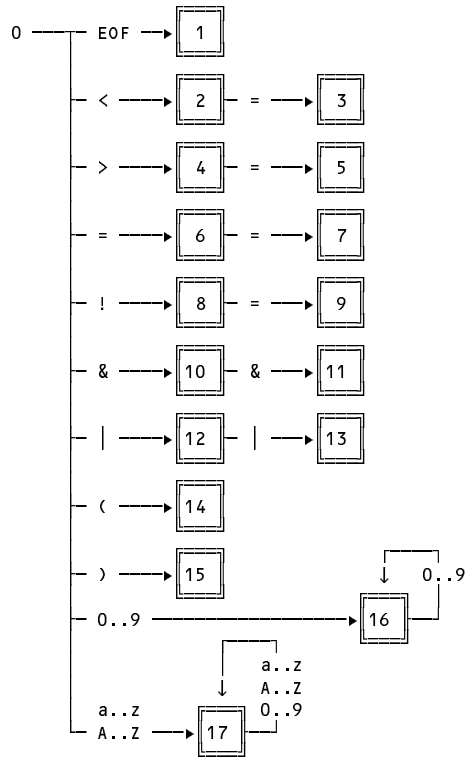


Figure 8.1 A transition diagram for a simple FSA

There are many ways of developing a scanner from these ideas. One approach, using a table-driven scanner, is suggested below. To the set of states suggested by the diagram we add one more, denoted by `finished`, to allow the postcondition to be easily realized.

```
TOKENKINDS FUNCTION GetSym;
(* Preconditions: ch is already available,
                  NextState, Token kind mappings defined
   Postcondition: ch is left as the character following token *)
  BEGIN
    state := 0;
    WHILE state ≠ finished DO
      lastState := state;
      state := NextState[state, ch];
      Get(ch)
    END;
    RETURN Token[lastState]
  END
```

Here we have made use of various mapping functions, expressed in the form of arrays:

`Token[s]`	is defined to be the token recognized when the machine has reached state *s*
`NextState[s, x]`	indicates the transition that must be taken when the machine is currently in state *s*, and has just recognized character *x*.

For our example, the arrays `Token` and `NextState` would be set up as in Figure 8.2. For clarity, the many transitions to the `finished` state have been left blank.

	ch											
State	a..z A..Z	0..9	<	>	=	!	&	|	(	)	EOF	
0	17	16	2	4	6	8	10	12	14	15	1	noSym
1												EOFSym
2					3							lssSym
3												leqSym
4					5							gtrSym
5												geqSym
6					7							assignSym
7												eqlSym
8					9							notSym
9												neqSym
10							11					noSym
11												andSym
12								13				noSym
13												orSym
14												lParenSym
15												rParenSym
16		16										numSym
17	17	17										identSym
	NextState[state, ch]											Token kind

Figure 8.2 Table of the states and the transitions between states in a simple FSA

A table-driven algorithm is efficient in time, and effectively independent of the token grammar and thus highly suited to automated construction. However it should not take much imagination to see that it is very hungry and wasteful of storage. A complex scanner might run to dozens of states, and many machines use an ASCII character set, with 256 values. For each character a column would be needed in the matrix, yet most of the entries (as in the example above) would be identical. And although we may have given the impression that this method will always succeed, this is not necessarily so. If the underlying token grammar were non-deterministic it might not be possible to define an unambiguous transition matrix - some entries might appear to require two or more values. In this situation we speak of requiring a **non-deterministic finite automaton** (NDFA) as opposed to the **deterministic finite automaton** (DFA) that we have been considering up until now.

Small wonder that considerable research has been invested in developing variations on this theme. The code below shows one possible variation, for our specimen grammar, in the form of a complete C# method. In this case it is necessary to have but one static array (denoted by `state0`), initialized so as to map each possible character into a single first state.

```
static int[] state0 = new int[256];

static void InitScannerStates() {
  int i;
  for (i = 0; i <= 255; i++) state0[i] = 0;
  for (i = '0'; i <= '9'; i++) state0[i] = 16;
  for (i = 'a'; i <= 'z'; i++) state0[i] = 17;
  for (i = 'A'; i <= 'Z'; i++) state0[i] = 17;
  state0['<'] =  2;  state0['>'] =  4;  state0['='] =  6;
  state0['!'] =  8;  state0['&'] = 10;  state0['|'] = 12;
  state0['('] = 14;  state0[')'] = 15;  state0[EOF] =  1;
} // InitScannerStates

static void GetSym() {
// Scans for next sym from input
  while (ch != EOF && ch <= ' ') GetChar();  // skip whitespace
  StringBuilder symLex = new StringBuilder();
  int symKind = noSym;
  int state = state0[ch];
  bool finished = false;
  while (!finished) {
    symLex.Append(ch); GetChar();
    switch (state) {
      case  1:
        symLex = new StringBuilder("EOF"); symKind = EOFSym;
        finished = true; break;
      case  2:
        if (ch == '=') state = 3; else { symKind = lssSym; finished = true; }
        break;
      case  3:
        symKind = leqSym; finished = true; break;
      case  4:
        if (ch == '=') state = 5; else { symKind = gtrSym; finished = true; }
        break;
      case  5:
        symKind = geqSym; finished = true; break;
      case  6:
        if (ch == '=') state = 7; else { symKind = assignSym; finished = true; }
        break;
      case  7:
        symKind = eqlSym; finished = true; break;
      case  8:
        if (ch == '=') state = 9; else { symKind = notSym; finished = true; }
        break;
      case  9:
        symKind = neqSym; finished = true; break;
      case 10:
        if (ch == '&') state = 11; else { symKind = noSym; finished = true; }
        break;
      case 11:
        symKind = andSym; finished = true; break;
      case 12:
        if (ch == '|') state = 13; else { symKind = noSym; finished = true; }
        break;
      case 13:
        symKind = orSym; finished = true; break;
      case 14:
        symKind = lParenSym; finished = true; break;
      case 15:
        symKind = rParenSym; finished = true; break;
      case 16:
        symKind = numSym; finished = !Char.IsDigit(ch); break;
      case 17:
        symKind = identSym; finished = !Char.IsLetterOrDigit(ch);
        break;
      default:
        symKind = noSym; finished = true; break;
    }
  }
  sym = new Token(symKind, symLex.ToString());
} // GetSym
```

It may be worth a final comment as regards the efficiency of the scanner routines illustrated here. Both of them appear to be simply and tightly coded. Both make frequent recourse to a character handler method, represented by `GetChar()`, and much will depend on how this method is implemented. The specimen code in the *Resource Kit* adopts a very simple approach, calling on a fundamental single-character reading routine in an I/O library each time a character is required. A routine like that may have been implemented very inefficiently, and efficiency can often be dramatically improved by using a single library routine to read the entire source that is to be scanned into

an internal buffer array, and then implementing the `GetChar` routine as one that simply selects the next character from this buffer when it is requested.

Further reading

Automata theory and the construction of finite state automata are discussed in most texts on compiler construction. A particularly thorough treatment is is to be found in the book by Gough (1988). Those by Watson (1989), Parsons (1992), Louden (1997) and Fischer and LeBlanc (1988, 1991) are also highly readable.

8.7 Keywords and literals

Our scanner algorithms are as yet immature. Earlier we claimed that scanners often incorporate such tasks as the recognition of keywords (which usually resemble identifiers), the evaluation of constant literals and so on. There are various ways in which these results can be achieved.

For example, suppose that our grammar was extended to recognize *"and"* as a keyword alternative for `&&`. We could accomplish this by extending the last part of the transition diagram given earlier to that shown in Figure 8.3.

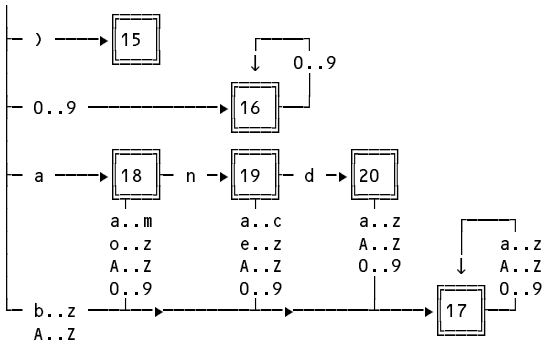


Figure 8.3 Part of a transition diagram for an extended FSA

In principle one could, of course, handle any number of keywords in a similar fashion. The number of states would grow very rapidly to the stage where manual construction of the table would become very tedious and error-prone.

Since in most languages a keyword is lexically equivalent to an identifier, and since the scanner already has the responsibility of constructing the lexeme for any identifier it encounters, an alternative technique is simply to search through a table of keywords before deciding finally on the token kind of an apparent identifier. In our *ad hoc* scanner this could be achieved by amending the section of code that recognizes identifiers as follows.

```
StringBuilder symLex = new StringBuilder();
if (Char.IsLetter(ch)) {
  do {
    symLex.Append(ch); GetChar();
  } while (Char.IsLetterOrDigit(ch));
  symKind = LiteralKind(symLex);
}
...
```

while the corresponding change to the FSA scanner would be

```
case 17:
  if (!Char.IsLetterOrDigit(ch)) {
    finished = true; symKind = LiteralKind(symLex);
  }
  break;
```

Here `LiteralKind` is to be developed in terms of an appropriate table-searching algorithm. In its simplest form this might be coded as follows (later versions of C# and Java allow the switch statement to be used).

```
static int LiteralKind(StringBuilder lex) {
  string s = lex.ToString();
  if (s.Equals("and")) return andSym;
  if (s.Equals("or")) return orSym;
  ...
  return identSym;

} // LiteralKind
```

```
switch lex.ToString() {
  case "and" : return andSym;
  case "or"  : return orSym;
  ...
  default    : return identSym;
} // switch
```

However, the subject of keyword recognition is important enough to warrant further comment. Since string comparisons are tedious, and since typically 50-70% of program text consists of either identifiers or keywords, it may make sense to perform this search as quickly as possible. This has to be weighed up against other efficiency considerations. In some cases it might be found that a simplistic implementation of the fundamental `GetChar` routine of the scanner was responsible for a far greater loss in overall efficiency than a poorly developed keyword recognition algorithm.

When there are many keywords to be distinguished, a simple linear search in an inherently unordered list, as is implicit in the above code, should be avoided unless the number of keywords is very small. Many efficient methods are known for searching tables or lists. In particular:

- the keywords can be stored in a table in alphabetic order and a binary search of order $O(\log_2 N)$ used;
- the keywords can be stored in a table in length order and a sequential search used among those that have the same length as the word just assembled;
- the keywords can be stored in alphabetic order and a sequential search used among those that have the same initial letter as the word just assembled - this is the technique employed in the Java version of Coco/R;
- a "perfect hash function" can be derived for the keyword set, allowing for a single string comparison to distinguish between all identifiers and keywords.

8.8 Comment handling

Generally speaking, comments in computer programs are intended for the benefit of a human reader and are of little interest to the parser. They effectively have the status of "white space" and can appear in source text anywhere that white space characters can appear - that is to say, between tokens that *are* of interest to the parser. It makes sense, therefore, for the scanner to be given the responsibility of stripping comments out of the source text. In those cases where comments are also used to supply compiler directives, or **pragmas**, the scanner may, of course, have a greater responsibility.

Comments in programming languages are generally introduced by a distinctive character or character sequence and terminated in the same way. Discarding comments would at first appear to be a trivial process but it calls for great care.

The simplest comments are introduced by a single character that is not otherwise part of the token alphabet, and are also terminated by a single character. Comments in assembler languages tend to be of this form, being introduced by a semicolon or # and terminated at the end of line. While it appears that such comments are discarded simply by adding another loop to the start of the scanner as so far developed,

```
while (ch != EOF && ch <= ' ') GetChar();    // skip whitespace
if (ch == startComment)                      // skip comment
  do GetChar(); while (ch != stopComment);
```

this neglects two fundamental aspects of the problem. Firstly, if the comment is not correctly terminated the loop will not terminate before the end-of-file is detected. Secondly, since the aim of the scanner is to detect a token, if a comment is discarded the scanner must nevertheless continue its search for a token - possibly discarding other comments in the process.

A better solution is as follows

```
static void GetSym() {
// Scans for next sym from input
  while (ch != EOF && ch <= ' ') GetChar();  // skip whitespace
  if (ch == startComment) {                  // skip comment
    do GetChar(); while (ch != stopComment && ch != EOF);
    if (ch != EOF) {                         // try again (recurse)
      GetChar(); GetSym(); return;
    }
    else {                                   // give up
      sym = new Token(EOFSym, "EOF"); return;
    }
  }
  ....  // as before
} // GetSym
```

Most programmers have written so-called *runaway comments* at some stage in their careers. Rather than return `EOFSym` as we have suggested here, it might be even more helpful to abort the program with a message to the effect that an unterminated comment had been detected. Sophisticated scanners will go further than we have indicated, and try to relate the error - which can be detected only at the end of the text - to the starting position of the runaway comment.

In many cases comments are introduced by a pair of characters and the leading character of the pair may itself be a member of the token alphabet. To illustrate one possible way of handling such comments, consider how we might extend the FSA scanner of the last section to discard Pascal style comments of the form `(* comment *)`, while still being able to recognize an opening parenthesis correctly.

The part of Figure 8.1 dealing with recognition of a left parenthesis is extended to include further transitions between three new states, as shown in Figure 8.4. After recognizing the opening sequence `(*` the FSA moves to state 18, where it remains until it detects an `*`, which may signal that the end of the comment is imminent. However, if this `*` is not followed by `)` - as would be the case during the analysis of a comment of the form `(**AB***)` - the FSA has to revert to state 18 or remain in state 19. As before, the scanner must handle the error condition that arises if the comment has not been correctly terminated.

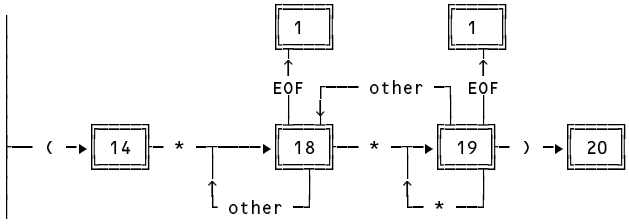


Figure 8.4 Part of a transition diagram for an extended FSA to handle (* comments *)

The last section of the scanner code given earlier could be replaced by the following

```
      case 14:
        if (ch == '*') state = 18; else { symKind = lParenSym; finished = true; }
        break;
      ...
      case 18:
        if (ch == EOF) state = 1; else if (ch == '*') state = 19;
        break;
      case 19:
        if (ch == EOF) state = 1;
        else if (ch == ')') state = 20;
        else if (ch != '*') state = 18; break;
      case 20:
        symKind = comSym; finished = true; break;
      default:
        symKind = noSym; finished = true; break;
    }
  }
  if (symKind == comSym) GetSym();
  else sym = new Token(symKind, symLex.ToString());
} // GetSym
```

Here we have introduced a pseudo token kind `comSym` to keep the code simple, though the recursive call to `GetSym` ensures that no token of this kind will ever be seen by the higher-level parser.

9 SYNTAX-DIRECTED TRANSLATION

In this chapter we build on the ideas developed in the previous two, and continue towards our goal of developing translators for computer languages by discussing how syntax analysis can form the basis for driving a translator or similar programs which process input strings that can be described by a grammar. Our discussion will be limited to methods suited to the top-down approach studied so far, and we shall make the further simplifying assumption that the sentences to be analyzed are essentially syntactically correct.

9.1 Embedding semantic actions into syntax rules

The primary goal of the types of parser studied in the last chapter - or, indeed, of any parser - is the recognition or rejection of input strings that claim to be valid sentences of the language under consideration. However, it does not take much imagination to see that, once a parser has been constructed, it might be enhanced to perform specific actions whenever various syntactic constructs have been recognized.

As usual, a simple example will help to crystallize the concept. We turn again to the grammars that can describe simple algebraic expressions, and in this case to a variant that can handle operands in the form of parenthesized sub-expressions, in addition to simple variables and numbers, using the usual four operators.

Expression = *Term* { "+" *Term* | "-" *Term* } .
Term = *Factor* { "*" *Factor* | "/" *Factor* } .
Factor = *identifier* | *number* | "(" *Expression* ")" .

It is easily verified that this grammar is LL(1). A simple recursive descent parser is readily constructed, with the aim of accepting a valid input expression or aborting, with an appropriate message if the input expression is malformed.

```
static void Expression() {
// Expression = Term { "+" Term | "-" Term } .
  Term();
  while (sym.kind == addSym || sym.kind == subSym) {
    GetSym(); Term();
  }
} // Expression

static void Term() {
// Term = Factor { "*" Factor | "/" Factor } .
  Factor();
  while (sym.kind == mulSym || sym.kind == divSym) {
    GetSym(); Factor();
  }
} // Term

static void Factor() {
// Factor = identifier | number | "(" Expression ")" .
  switch (sym.kind) {
    case identSym:
    case numSym:
      GetSym(); break;
    case lParenSym:
      GetSym(); Expression();
      Accept(rParenSym, ") expected"); break;
    default:
      Abort("invalid start to Factor"); break;
  }
} // Factor
```

Note that in this and subsequent examples we have assumed the existence of a lower level scanner *GetSym()* that recognizes fundamental terminal symbols, and constructs a globally accessible object *sym* of the *Token* class suggested in section 8.1. For simplicity we have also assumed the existence of simple auxiliary methods *Accept* and *Abort*, similar to those used in the last chapter, and we remind the reader that an initial call to *GetSym()* would be needed before calling *Expression()* for the first time.

Now consider the problem of reading a valid string in this language, and translating it into a string that has the same meaning but which is expressed in *postfix* (that is, "reverse Polish") notation. Here the operators follow the pairwise operands and there is no need for parentheses. For example, the infix expression

(*a* + *b*) * (*c* - *d*)

is to be translated into its postfix equivalent

a b + *c d* - *

This is a well-known problem, allowing a fairly straightforward solution. As we read the input string from left to right we immediately copy all the *operands* to the output stream as soon as they are recognized, but we delay copying the *operators* until we can do so in an order that relates to the familiar precedence rules for the operations they imply. With a little thought the reader should see that the grammar and the parser given above capture the spirit of these precedence rules. Given this insight, it is not difficult to see that the augmented routines below not only parse input strings - the execution of the carefully positioned output statements also effectively produces the required postfix translation.

```
static void Expression() {
// Expression = Term { "+" Term | "-" Term } .
  Term();
  while (sym.kind == addSym || sym.kind == subSym) {
    switch (sym.kind) {
      case addSym:
        GetSym(); Term(); output.Write(" +"); break;
      case subSym:
        GetSym(); Term(); output.Write(" -"); break;
    }
  }
} // Expression

static void Term() {
// Term = Factor { "*" Factor | "/" Factor } .
  Factor();
  while (sym.kind == mulSym || sym.kind == divSym) {
    switch (sym.kind) {
      case mulSym:
        GetSym(); Factor(); output.Write(" *"); break;
      case divSym:
        GetSym(); Factor(); output.Write(" /"); break;
    }
  }
} // Term

static void Factor() {
// Factor = identifier | number | "(" Expression ")" .
  switch (sym.kind) {
    case identSym:
    case numSym:
      output.Write(" " + sym.val); GetSym(); break;
    case lParenSym:
      GetSym(); Expression();
      Accept(rParenSym, ") expected"); break;
    default:
      Abort("invalid start to Factor"); break;
  }
} // Factor
```

In a very real sense we have moved from a parser to a compiler in one easy move! What we have illustrated is a simple example of a syntax-directed program - one in which the governing algorithm is readily developed from an understanding of an underlying syntactic structure. Compilers are obvious candidates for this sort of development, although the technique is more generally applicable as, hopefully, will become clear.

The reader might wonder whether this idea could somehow be reflected back to the formal grammar from which the parser was developed. Various schemes have been proposed for doing this. Many of these use the idea of adding **semantic actions** into context-free BNF or EBNF production schemes.

Unfortunately there is no clear winner among the notations proposed for this purpose. Most, however, incorporate the actions by writing statements in some implementation language (for example, C# or Java) between suitably chosen metabrackets that are not already reserved in that language. For example, Coco/R uses EBNF for expressing the productions, and brackets the actions with "`(.`" and "`.)`", as in the example below.

```
Expression
= Term
   {   "+" Term            (. output.Write(" +"); .)
     | "-" Term            (. output.Write(" -"); .)
   } .

Term
= Factor
   {   "*" Factor          (. output.Write(" *"); .)
     | "/" Factor          (. output.Write(" /"); .)
   } .

Factor
=   ( identifier | number )    (. output.Write(" " + sym.val); .)
  | "(" Expression ")" .
```

The **yacc** parser generator on UNIX systems uses unextended BNF for the productions, and uses braces, "{" and "}", around actions expressed in C.

9.2 Attribute grammars

A little reflection will show that, although an algebraic expression clearly has a semantic meaning (in the sense of its *value*), this was not brought out when developing the last example. While the incorporation of actions into the context-free productions of a grammar is a powerful tool for documenting and developing syntax-directed programs, what we have seen so far is still inadequate for handling the many situations where some deeper semantic meaning is required.

We have seen how a context-free grammar can be used to describe many features of programming languages. Such grammars effectively define a derivation or parse tree for each syntactically correct program in the language, and we have seen that with care we can construct the grammar so that this parse tree in some way reflects the meaning of the program as well.

As an example, consider the usual old chestnut language, albeit described by a slightly different (non-LL(1)) grammar

```
Goal       = Expression .
Expression = Term | Expression "+" Term | Expression "-" Term.
Term       = Factor | Term "*" Factor | Term "/" Factor .
Factor     = identifier | number | "(" Expression ")" .
```

and consider the phrase structure tree for $x + y * z$, shown in Figure 9.1.

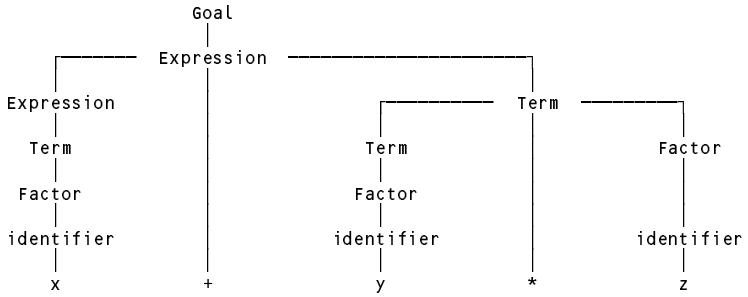


Figure 9.1 Phrase structure tree for x + y * z

Suppose x, y and z had associated numerical values of 3, 4 and 5, respectively. We can think of these as **semantic attributes** of the leaf nodes x, y and z. Similarly we can think of the nodes '+' and '*' as having attributes of "add" and "multiply". Evaluation of the whole expression can be regarded as a process where these various attributes are passed "up" the tree from the terminal nodes and are semantically transformed and combined at

higher nodes to produce a final result or attribute at the root - the value (23) of the *Goal* symbol. This is illustrated in Figure 9.2.

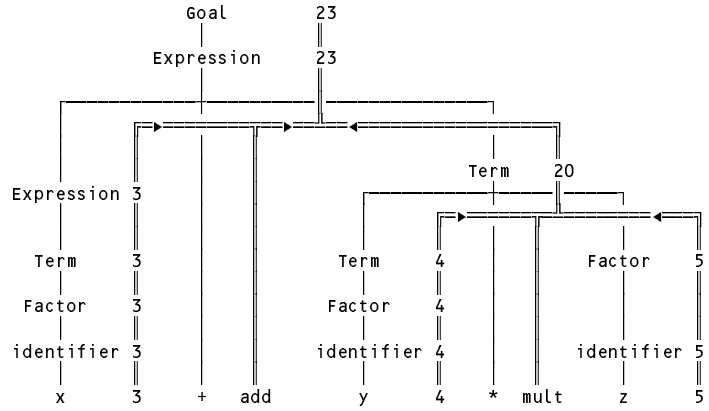


Figure 9.2 Passing attributes up a parse tree

In principle, and indeed in practice, parsing algorithms can be written whose embedded actions explicitly construct such trees as the input sentences are parsed, and also *decorate* or *annotate* the nodes with the semantic attributes. In fact, it would be unusual to construct a "concrete syntax" tree exactly like that shown in Figure 9.1. It would be far more likely that an AST would be constructed, of the sort mentioned in section 2.5 (the *tree* would of course be "concrete" - it is the *syntax* that is abstract). Associated tree-walking algorithms can then be invoked to process the semantic information stored in the nodes in a variety of ways, possibly making several passes over the tree before the evaluation is complete. This approach lends itself well to the construction of optimizing compilers, where repeatedly walking the tree can be used to prune or graft nodes in a way that a simpler compiler cannot hope to do.

The parser for recognizing this language, as developed in the last section, did not construct an explicit parse tree. The grammar we have now employed seems to map immediately to parse trees in which the usual associativity and precedence of the operators is correctly reflected, but it is left recursive and thus unsuitable as the basis on which to construct a recursive descent parser. (It is possible to construct other forms of parser that can handle grammars that employ left recursion, although this is beyond the scope of this book). For the moment we shall not pursue the interesting problem of whether or how a recursive descent parser could be modified to generate an explicit tree. We shall content ourselves with the observation that the execution of such a parser effectively walks an implicit structure, whose nodes correspond to the various calls made to the sub-parsers as the parse proceeds.

Notwithstanding any apparent practical difficulties, our notions of formal grammars can be extended to try to capture the essence of the attributes associated with the nodes, by enhancing the notation still further. In one scheme, attribute rules are associated with the context-free productions in much the same way as we have already seen for actions, giving rise to what is known as an **attribute grammar**. As usual, an example will help to clarify.

```
Goal
=  Expression                (. Goal.Value := Expr.Value .) .
Expression
=  Term                      (. Expr.Value := Term.Value .)
  | Expression "+" Term      (. Expr.Value := Expr.Value + Term.Value .)
  | Expression "-" Term      (. Expr.Value := Expr.Value - Term.Value .) .
Term
=  Factor                    (. Term.Value := Fact.Value .)
  | Term "*" Factor          (. Term.Value := Term.Value * Fact.Value .)
  | Term "/" Factor          (. Term.Value := Term.Value / Fact.Value .) .
Factor
=  identifier                (. Fact.Value := identifier.Value .)
  | number                   (. Fact.Value := number.Value .)
  | "(" Expression ")"       (. Fact.Value := Expr.Value .) .
```

Here we have employed the familiar "dot" notation that many imperative languages use in designating the fields of object or record structures. Were we to employ a parsing algorithm that constructed an explicit tree, this notation

would immediately be consistent with that needed to manipulate the tree nodes used for these structures.

It is important to note that the semantic rules for a given production specify the relationships between attributes of other symbols in the *same* production, and are essentially "local". Furthermore, the assignment statements are to be interpreted in a familiar way. An action such as

```
Expr.Value := Expr.Value + Term.Value;
```

implies that a new attribute for an *Expression* is to be computed from the attribute values of an extant *Expression* and its succeeding *Term*.

It is not necessary to have a left recursive grammar to be able to provide attribute information. We could write an iterative LL(1) grammar in much the same way.

```
Goal
= Expression                (. Goal.Value := Expr.Value .) .
Expression
= Term                      (. Expr.Value := Term.Value .)
  { "+" Term                (. Expr.Value := Expr.Value + Term.Value .)
  | "-" Term                (. Expr.Value := Expr.Value - Term.Value .)
  } .
Term
= Factor                    (. Term.Value := Fact.Value .)
  { "*" Factor              (. Term.Value := Term.Value * Fact.Value .)
  | "/" Factor              (. Term.Value := Term.Value / Fact.Value .)
  } .
Factor
= identifier                (. Fact.Value := identifier.Value .)
| number                    (. Fact.Value := number.Value .)
| "(" Expression ")"        (. Fact.Value := Expr.Value .) .
```

Our notation does not yet lend itself immediately to the specification and construction of those parsers that do *not* construct explicit structures of decorated nodes. However, it is not difficult to develop a suitable extension. We have already seen that the construction of parsers can be based on the idea that expansion of each non-terminal is handled by an associated routine. These routines can be parameterized, and the parameters can transmit the attributes to where they are needed. Using this idea we might express our expression grammar as follows (where we have introduced yet more metabrackets, this time denoted by "<" and ">", to enclose the parameters):

```
Goal < Value >
= Expression < Value > .
Expression < Value >
= Term < Value >
  { "+" Term < TermValue >      (. Value := Value + TermValue .)
  | "-" Term < TermValue >      (. Value := Value - TermValue .)
  } .
Term < Value >
= Factor < Value >
  { "*" Factor < FactorValue >  (. Value := Value * FactorValue .)
  | "/" Factor < FactorValue >  (. Value := Value / FactorValue .)
  } .
Factor < Value >
= identifier < Value >
| number < Value >
| "(" Expression < Value > ")" .
```

9.3 Synthesized and inherited attributes

A little contemplation of the parse tree in our earlier example, and of the attributes as given here, should convince the reader that (in this example at least) we have a situation in which the attributes of any particular node depend only on the attributes of nodes in the subtrees of the node in question. In a sense, information is always passed "up" the tree, or "out" of the corresponding routines. The grammar above maps into C# code of the form shown

below, where the parameters must be passed "by reference". For the moment we side-step the issue of how one attributes a variable *identifier* with a numeric value, simply calling an unspecified *ValueOf* method to do this.

```
static void Expression(out int value) {
// Expression = Term { "+" Term | "-" Term } .
  int termValue;
  Term(out value);
  while (sym.kind == addSym || sym.kind == subSym) {
    switch (sym.kind) {
      case addSym:
        GetSym();
        Term(out termValue); value += termValue; break;
      case subSym:
        GetSym();
        Term(out termValue); value -= termValue; break;
    }
  }
} // Expression

static void Term(out int value) {
// Term = Factor { "*" Factor | "/" Factor } .
  int factorValue;
  Factor(out value);
  while (sym.kind == mulSym || sym.kind == divSym) {
    switch (sym.kind) {
      case mulSym:
        GetSym();
        Factor(out factorValue); value *= factorValue; break;
      case divSym:
        GetSym();
        Factor(out factorValue); value /= factorValue; break;
    }
  }
} // Term

static void Factor(out int value) {
// Factor = identifier | number | "(" Expression ")" .
  switch (sym.kind) {
    case identSym:
      value = ValueOf(sym.val); GetSym(); break;
    case numSym:
      value = Convert.ToInt32(sym.val); GetSym(); break;
    case lParenSym:
      GetSym(); Expression(out value);
      Accept(rParenSym, ") expected"); break;
    default:
      value = 0; Abort("invalid start to Factor"); break;
  }
} // Factor
```

This idea has to be modified slightly if one is to write such parsers in Java, where parameters cannot as easily be passed by reference. Here, rather than write a *void* method for each production, as we have done until now, it may be necessary to develop some methods in the form of functions that return the synthesized attribute or attributes, possibly necessitating the introduction of some special classes to package attributes together. In the example just described we could write each of the three methods to return a simple `int` result, and it will suffice to show the form that the *Factor* parser might assume (the others would follow similar lines).

```
static int factor() { // Java version
// Factor = identifier | number | "(" expression ")" .
  int value = 0;   // local value for intermediate computation
  switch (sym.kind) {
    case identSym:
      value = valueOf(sym.val); getSym(); break;
    case numSym:
      value = Integer.parseInt(sym.val); getSym(); break;
    case lParenSym:
      getSym(); value = expression();
      accept(rParenSym, ") expected"); break;
    default:
      value = 0; abort("invalid start to Factor"); break;
  }
  return value;
} // factor (Java version)
```

Attributes that travel in this way are known as *synthesized attributes*. Given a context-free production rule

$$A = \alpha \ B \ \gamma$$

then an associated semantic rule of the form

$$A.\text{attribute}_i = f(\alpha, B, \gamma)$$

is said to specify a **synthesized attribute** of A.

Attributes do not always travel up a tree. As a rather grander example, consider the very small Parva program.

```
void main () {
  const bonus = 4;
  int pay;
  write(pay + bonus);
} // main
```

which has the concrete phrase structure tree shown in Figure 9.3.

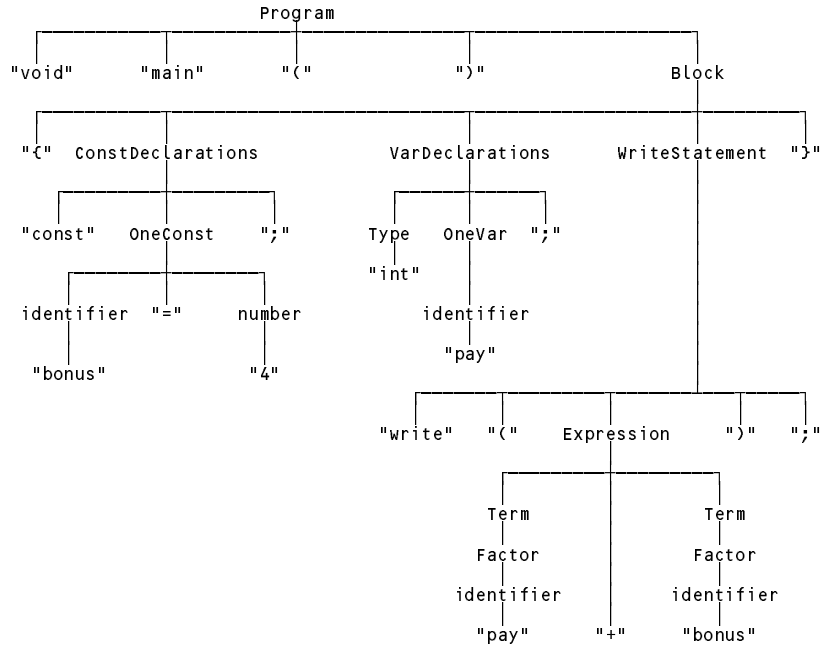


Figure 9.3 Parse tree for a complete small program

In this case we can think of the Boolean *IsConstant* and *IsInteger* attributes of the nodes `const` and `int` as being passed up the tree (*synthesized*), and then later passed back down and *inherited* by other nodes like `bonus` and `pay` (see Figure 9.4). In a sense, the context in which the identifiers were declared is being remembered - the system is providing a way of handling context-sensitive features of an otherwise context-free language.

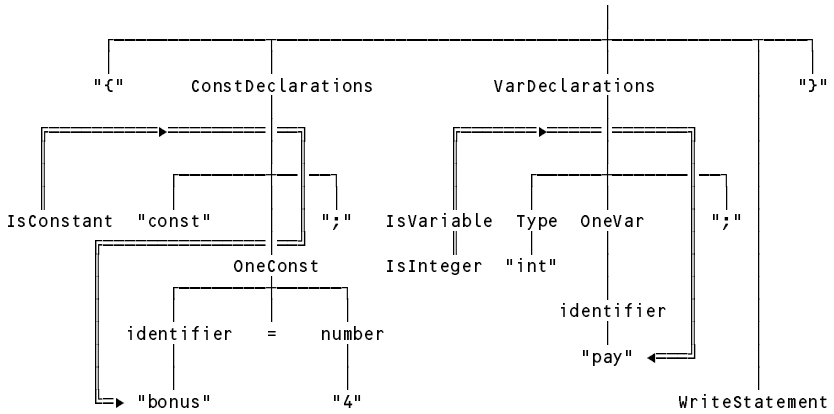


Figure 9.4 Attributes passed up and down a parse tree

Of course, this idea must be taken much further. Attributes like this form part of what is usually termed an **environment**. Compilation or parsing of programs in a language like Pascal or Modula-2 generally begins in a "standard" environment in which pervasive identifiers like TRUE, FALSE, ORD, CHR and so on are already incorporated. This environment - maintained in some form of symbol table - is augmented by *import* or *using* declarations in languages like Java and C#. It is passed to the program where (in our demonstration) it is inherited by *Block* and then by *ConstDeclarations*, which augments it and passes it back up, to be inherited in its augmented form by *VarDeclarations* which augments it further and passes it back, so that it may then be passed down to the remainder of the statements. We may try to depict this as shown in Figure 9.5.

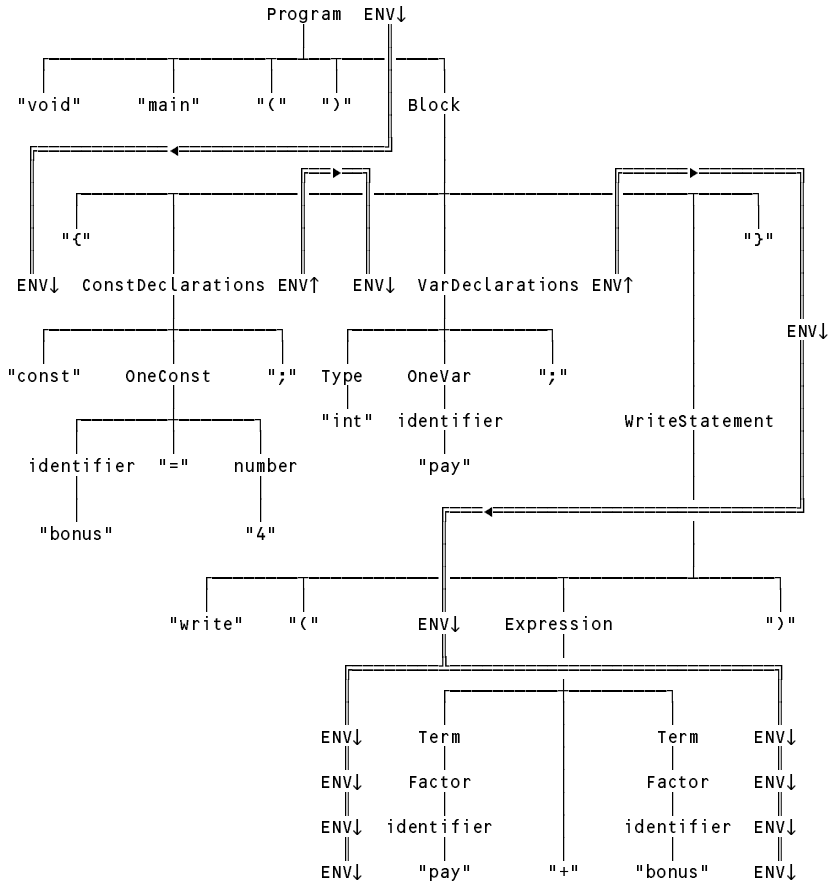


Figure 9.5 Modification of the parsing environment

More generally, given a context-free production rule of the form

$$A = \alpha B \gamma$$

an associated semantic rule of the form

$$B.\text{attribute}_i = f(\alpha, A, \gamma)$$

is said to specify an **inherited attribute** of B. The inherited attributes of a symbol are computed from information held in the environment of the symbol in the parse tree.

As before, our formal notation needs modification to reflect the different forms and flows of attributes. A notation often used employs arrows ↑ and ↓ in conjunction with the parameters mentioned in the < > metabrackets. Inherited attributes are marked with ↓, and synthesized attributes with ↑. In terms of actual coding, ↑ attributes correspond to "reference" parameters, while ↓ attributes correspond to "value" parameters. In practice, reference parameters may also be used to manipulate features (such as an environment) that are inherited, modified, and then returned - these are sometimes called **transmitted attributes**, and are marked with ↓↑ or ⇕.

9.4 Classes of attribute grammars

Attribute grammars have other important features. If the action of a parser is in some way to construct a tree whose nodes are decorated with semantic attributes relevant to each node, then "walking" this tree after it has been constructed should provide a possible mechanism for developing the synthetic aspects of a translator, such as code generation. If this is the case, then the order in which the tree is walked becomes crucially important, since attributes can depend on one another. The simplest tree walk (depth-first, left-to-right) will not suffice, for example, if we can "use" a variable or method before it has been "declared". In principle we can, of course, perform multiple tree walks, just as we can perform multiple-pass compilation. There are, however, two types of attribute grammars for which this is not necessary.

- An **S-attributed grammar** is one that uses only synthesized attributes. For such a grammar the attributes can obviously be correctly evaluated using a bottom-up walk of the parse tree. Furthermore, such a grammar is easily handled by parsing algorithms (such as recursive descent) that do not explicitly build the parse tree.

- An **L-attributed grammar** is one in which the inherited attributes of a particular symbol in any given production are restricted in that, for each production of the general form

  $$A \rightarrow B_1 B_2 \ldots B_n$$

  the inherited attributes of B_k may depend only on the inherited attributes of A or synthesized attributes of $B_1, B_2 \ldots B_{k-1}$. For such a grammar the attributes can be correctly evaluated using a left-to-right depth-first walk of the parse tree, and such grammars are usually easily handled by recursive descent parsers, which implicitly walk the parse tree in this way.

We have already pointed out that there are various aspects of computer languages that involve context sensitivity, even though the general form of the syntax might be expressed in a context-free way. Context-sensitive constraints on such languages - often called **context conditions** - are often conveniently expressed by conditions included in an attribute grammar, specifying relations that must be satisfied between the attribute values in the parse tree of a valid program. For example, we might have a production like

```
Assignment = VarDesignator < TypeV↑ > ":=" Expression < TypeE↑ >
             (. where AssignmentCompatible(TypeV↓, TypeE↓) .) .
```

Alternatively, and more usefully in the construction of real parsers, the context conditions might be expressed in the same notation as for semantic actions. For example,

```
Assignment = VarDesignator < TypeV↑ > ":=" Expression < TypeE↑ >
             (. if (Incompatible(TypeV↓, TypeE↓))
                  SemError("Incompatible types"); .) .
```

10 COCO/R: A DETAILED GUIDE

One of the main reasons for developing attributed grammars like those discussed in the last chapter is to be able to use them as input to compiler generator tools, and so construct complete programs. It is the aim of this chapter and the next to illustrate how this process is achieved with Coco/R and to discuss the Cocol specification language in greater detail than before. Our discussion will, as usual, focus mainly on C# applications, but a study of the documentation and examples in the *Resource Kit* should allow easy development of Java applications as well.

10.1 Coco/R - a brief history

A table-driven parser generator called Coco was developed in 1983 in Linz, Austria. The derivative now known as Coco/R was developed in Oberon in 1989 by Hanspeter Mössenböck at ETH Zürich, who continued to develop and maintain the system after his return to the Institute of Practical Computer Science at Johannes Kepler University in Linz. Highly compatible ports to Modula-2 and Pascal soon followed. Frankie Arzu developed first version that generated applications in C or C++ in the early 1990s. This was described and used in an earlier book by Terry (1997). Since then, Mössenböck has released Java and C# versions, the latest of which can employ LL(*k*) lookahead to allow LL(1) conflicts in grammars to be accommodated. Since the system is distributed as "open source" software, other versions have been developed from the official Linz releases, including ones for Delphi, Icon and Ruby. The many variants of Coco/R can now be obtained from several sites on the Internet. The official Coco/R site is at `http://www.ssw.uni-linz.ac.at/Coco/`

The evolution of the system has inevitably been such that the various versions and releases have a close resemblance one to another, but also have a number of fairly significant and even annoying differences (for example, in the way in which escape sequences in strings are treated). In the *Resource Kit* that accompanies this book can be found implementations of Coco/R that can generate applications in C# or Java. These have been modified from the official Linz releases to include some features that are useful in a tutorial text like this one, and are configured for easy use on MS-Windows based systems (and on UNIX systems, in the case of the Java version). The rest of this chapter is limited to describing features that are common to both these implementations.

10.2 Installing and running Coco/R

The implementations of Coco/R do not make use of GUI interfaces and are most conveniently run in command line mode. The installation and execution of Coco/R is rather system-specific, but a brief overview of the process can usefully be given here from the viewpoint of a user running a Command Window on an MS-Windows system.

10.2.1 Installation

The various versions of Coco/R are supplied as compressed files, and for these the installation process requires a user to:

- create a system directory to store the system files [`mkdir c:\coco`];
- make this the active directory [`cd c:\coco`];
- copy the distribution file to the system directory [`copy d:cocowin.zip c:\coco`];
- start the decompression process [`unzip cocowin.zip`] - this process will extract the files, and create further subdirectories to contain Coco/R and its support files and library modules;
- add the system directory to the command path - this may often most easily be done by using the "environment variables" option under the "advanced" properties of the system configuration tool in the MS-Windows control panel;
- (for some versions) set an environment variable so that Coco/R can locate its frame files - this can be done by adding `CRFRAMES` to the list of environment variables, and setting it to `c:\coco\frames`);
- (for the Java version) add the system directory to the class path - usually by modifying the entry for the environment variable `CLASSPATH` to something like `.;c:\coco;c:\j2sdk1.4.1_01\jre\lib\rt.jar`.

10.2.2 Input file preparation

For each application, the user has to prepare a text file to contain the attributed grammar. Points to be aware of are that:

- it is sensible to work within a "project directory" (say `c:\work`) and not within the "system directory" (`c:\coco`);
- text file preparation must be done with an ASCII editor and not with a word processor;
- by convention the file of the attributed grammar (the Cocol source) is named with a primary name that is based on the grammar's goal symbol, and with an "`ATG`" extension - for example `Calc.ATG`.

Besides the grammar, Coco/R needs to be able to read **frame files**. These contain outlines of the scanner, parser, and driver classes, to which will be added statements derived from an analysis of the attributed grammar. Frame files for the scanner and parser are of a highly standard form - those supplied with the distribution are suitable for use in many applications without the need for any customization. However, a complete compiler consists of more than just a scanner and parser - in particular it requires a driver program to call the parser. A basic driver frame file (`Driver.frame`) comes with the distribution kit. This will allow simple applications to be generated immediately, but it is usually necessary to copy this basic file to the project directory and then to edit it to suit the application. The resulting file should be given the same primary name as the grammar file, and a `.frame` extension - for example `Calc.frame`.

10.2.3 Execution

Once the grammar file has been prepared, generation of the application is started with a command like

```
coco  Calc.ATG
```

A number of compiler options may be specified after a command line parameter `-options`, for example

```
coco  Calc.ATG   -options mc
```

The options depend on the particular version of Coco/R in use. A summary of those available can be obtained by issuing the `coco` command with no parameters at all. Commonly used options are:

`c`	generate a compiler driver;
`f`	list FIRST/FOLLOW sets for all non-terminals;
`m`	merge error messages with source code to form output listing;
`n`	generate names for terminal tokens;
`t`	test grammar but do not generate code;
`x`	give a cross-reference listing of the grammar.

Compiler options may also be selected by **pragmas** of the form `$CN` embedded in the attributed grammar itself, and this is probably the preferred approach for serious applications. Examples of such pragmas can be found in the case studies later in this chapter.

Coco/R needs to be able to locate the frame files as well as the file containing the attributed grammar. The standard frame files can be copied to the project directory. However, to prevent unnecessary proliferation, they are better left in the `frames` sub-directory of the system directory. The versions of Coco/R described here can find them if the path to the frame files is specified after a command line parameter `-frames`, as exemplified by

```
coco  Calc.ATG   -options mc   -frames C:\coco\frames\
```

10.2.4 Output from Coco/R

Assuming that the attributed grammar appears to be satisfactory, and depending on the compiler options specified, execution of Coco/R will typically result in the production of source code files for various classes, including:

- an FSA scanner class (`Scanner.cs` or `Scanner.java`);
- a recursive descent parser class (`Parser.cs` or `Parser.java`);
- a driver class (for example, `Calc.cs` or `Calc.java`).

All generated classes belong to a *namespace* (or *package* in Java terminology) having the name of the grammar's goal symbol (for example `namespace Calc` or `package Calc`). In the Java implementation these files must be created in (or moved to) a correspondingly named sub-directory, a process simplified by the use of an appropriate batch

file such as is supplied with the distribution kit. In general, a large application will involve the use of other source files - for example, defining code generators or symbol table handlers. These must also belong to the appropriate namespaces or packages and/or make use of appropriate `using` or `import` statements to ensure that they have access to all classes and methods involved in the application.

10.2.5 Assembling the generated system

After they have been generated, the various parts of an application can be compiled and linked with one another, and with any other components that they need. The way in which this is done depends very much on the host compiler. For a very simple C# application one might issue a command like

```
csc /out:calc.exe Calc.cs Scanner.cs Parser.cs
```

or, for more complex applications

```
csc /out:parva.exe Parva.cs Scanner.cs Parser.cs PVM.cs Library.cs CGen.cs Table.cs /warn:2
```

but for larger applications the use of a carefully defined batch script is probably to be preferred.

For a simple Java application one might use a command like

```
javac -d . -deprecation -nowarn -Xstdout Calc\*.java
```

but, once again, this can be simplified by the use of carefully defined batch files.

10.3 Overall form of a Cocol description

Preparation of completely attributed grammars suitable as input to Coco/R requires an in-depth understanding of the Cocol specification language, including many features that we have not yet encountered. Sections 10.4 to 10.5 discuss these aspects in some detail, and owe much to the original description by Mössenböck (1990a).

In section 5.9.3 an example was given of an unattributed Cocol description of a simple calculator. For ease of reference we give it again:

```
COMPILER Calculator

CHARACTERS
  digit      = "0123456789" .
  hexdigit   = digit + "ABCDEF" .

TOKENS
  decNumber  = digit { digit } .
  hexNumber  = "$" hexdigit { hexdigit } .

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Calculator = { Expression "=" } .
  Expression = Term { "+" Term  |  "-" Term } .
  Term       = Factor { "*" Factor |  "/" Factor } .
  Factor     = decNumber | hexNumber .
END Calculator.
```

Cocol grammars like this can be described in EBNF by

Cocol = [*LibraryAccess*]
"COMPILER" *GoalIdentifier*
[*ArbitraryText*]
[*ScannerSpecification*]
ParserSpecification
"END" *GoalIdentifier* "." .

We note immediately that the identifier after the keyword `COMPILER` gives the grammar name and must match the name after the keyword `END`. The grammar name must also match the name chosen for the non-terminal that

defines the goal symbol of the phrase structure grammar.

Each of the productions in the *ParserSpecification* leads to the generation of a corresponding parsing routine, each one forming a private static method of the overall *Parser* class. It should not take much imagination to see that the complete system will also need to incorporate operations for:

- converting the string that defines a decimal or hexadecimal token into a corresponding numerical value - thus we need mechanisms for extracting the attributes of various tokens from the scanner that recognizes them;
- storing such values in variables, and passing these values between the routines - thus we need mechanisms for declaring parameters and local variables in the generated routines, and for incorporating arithmetic and logic operations;
- displaying the values of the variables on an output device - thus we need mechanisms for interfacing our parsing routines to external I/O library routines;
- reacting sensibly to input data that does not conform to the proper syntax - thus we need mechanisms for specifying how error recovery should be accomplished;
- reacting sensibly to data that is syntactically correct, but still meaningless, as might happen if one were asked to divide by zero - thus we need mechanisms for reporting semantic and constraint violations.

These mechanisms are all incorporated into the grammar by attributing it with extra information, as discussed in the next sections.

Two points follow immediately.

- The *LibraryAccess* component consists of appropriate `using` (C#) or `import` (Java) declarations that give the methods in the *Parser* class access to facilities in other namespaces or packages.

- Arbitrary text may follow the *GoalIdentifier*, preceding the *ScannerSpecification*. This is not checked by Coco/R, but is simply incorporated directly in the generated *Parser* class. This offers the facility for declaring fields and methods in this class that may be needed by later semantic actions, or even by other classes in the application.

10.4 Scanner specification

A scanner has to read source text, skip meaningless characters and recognize tokens that can be handled by the parser. Clearly there has to be some way for the parser to retrieve information about these tokens. The most fundamental information can be returned in the form of a simple integer, unique to the kind of token recognized. While a moment's thought will confirm that the members of such an enumeration will allow a parser to perform syntactic analysis, semantic properties (such as the numeric value of a `decNumber` that appears in our example grammar) may require a token to be analyzed in more detail. To this end, in a manner familiar from Chapter 8, the generated scanner supplies the parser with objects of a *Token* class, which allows for the retrieval of the **lexeme** or textual representation of a token.

Tokens can be classified either as literals or generic tokens. As we have already seen, literals (like "`while`" and "`!=`") can be introduced directly into productions as strings, and do not need to be named. Generic token specifiers (such as *identifier* or *number*) must be named and have structures that could be defined by regular expressions, although in Cocol they are defined in EBNF "style".

In Cocol, a scanner specification consists of seven optional parts, strictly in the order:

ScannerSpecification = [*IgnoreCase*]
[*CharacterSets*]
[*Tokens*]
[*UserNames*]
[*Pragmas*]
{ *Comments* }
{ *WhiteSpace* } .

10.4.1 Character sets

The *CharacterSets* component is *optional*. If present it defines and names the character sets from which tokens

can be defined in the later TOKENS section. It does nothing more; in particular it does not define tokens themselves, even though at first it might seem to do so. While the definitions seem to introduce strings, this is illusory - the quotation marks are used in the way that mathematicians would use curly braces.

```
CharacterSets  = "CHARACTERS" { NamedCharSet } .
NamedCharSet   = SetIdent "=" CharacterSet "." .
CharacterSet   = SimpleSet { ( "+" | "-" ) SimpleSet } .
SimpleSet      = SetIdent  |  string | SingleChar [ ".." SingleChar ] | "ANY" .
SingleChar     = "CHR" "(" number ")" .
SetIdent       = identifier .
```

Simple character sets are denoted by one of

string	the set (not a string!) consisting of all characters in that string
CHR(*i*)	the singleton set of one character with ordinal value *i*
CHR(*i*) .. CHR(*j*)	the set consisting of all characters whose ordinal values are in the closed range *i* … *j*
ANY	the set of all characters acceptable to the implementation of Coco/R
SetIdent	a previously declared character set with that name

As examples we might have

```
digit     = "0123456789" .      // The set of decimal digits
hexdigit  = digit + "ABCDEF" .  // The set of hexadecimal digits
eol       = CHR(10) .           // Line feed character
space     = CHR(32) .           // Familiar white space character
backSlash = CHR(92) .           // C# escape character \
```

There are a few subtleties in this connection.

- Any number of sets can be defined, and they can have elements in common or not, as is convenient.
- The + and - operators can be used to form the union or differences of sets, and the .. notation can be used to simplify the specification of a range. This is especially useful where this range includes control characters.
- ANY is a keyword that in this context means "all characters acceptable to the implementation of Coco/R". Although Java itself uses Unicode, Coco/R for Java and C# in this version are restricted to the low end (8 bit) subset of UNICODE. The first 128 elements of ASCII and Unicode are the same.
- The strings that can be used to define a *SimpleSet* are not really strings in the usual sense. This is one of the areas where different variants of Coco/R differ in their treatment of what appear to a C# or Java programmer to be familiar "escape sequences" like \n (newline) or \t (tab).
- One cannot put spaces into literal strings in Coco/R (see later), where these strings are being used in the CHARACTERS section to denote a set of characters, or in the TOKENS or PRODUCTIONS section to denote a terminal.
- It is recommended that escape sequences be avoided and that use be made of the CHR(*i*) mechanism. This will also simplify matters should it be required to port a Cocol specification easily to another variant.
- In particular, control characters like CR (CHR(13)) and LF (CHR(10)) are best represented using the CHR(n) notation. So is SP (space, or CHR(32)) if for some strange reason it is needed as an *internal* character in a token. Extra spaces *between* tokens are almost always ignored.

By way of example, here are some correct CHARACTERS declarations. Many applications simply use pseudo "strings":

```
CHARACTERS
  ULetter    = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" .  /* set of all upper case letters */
  LLetter    = "abcdefghijklmnopqrstuvwxyz" .  /* set of all lower case letters */
  digit      = "0123456789" .                  /* set of all decimal digits */
  hexDigit   = "0123456789ABCDEF" .            /* set of all hexadecimal digits */
  AnyVowel   = "AEIOUaeiou" .                  /* set of all English vowels in either case */
```

Here are some others that are valid, illustrating ranges, set unions and differences:

```
ULetter    = "A" .. "Z" .                    /* set of all upper case letters */
LLetter    = "a" .. "z" .                    /* set of all lower case letters */
letter     = ULetter + LLetter .             /* set of all possible letters */
control    = CHR(0) .. CHR(31) .             /* set of all ASCII control characters */
inString   = ANY - control - '"' - CHR(92) . /* set of all easy characters within a string */
printable  = ANY - control .                 /* set of all easily printable characters */
hexDigit   = digit + "ABCDEFabcdef" .        /* set of all hex digits, either case acceptable */
noLetter   = ANY - letter .                  /* set of all characters except letters */
ctrlChars  = CHR(0) .. CHR(31) .             /* set of all the ASCII control characters */
```

Here are some "singleton" ones used to get round the problem of representing certain characters in the "string" version:

```
LF         = CHR(10) .                       /* line feed */
CR         = CHR(13) .                       /* carriage return */
SP         = CHR(32).
backslash  = CHR(92) .                       /* the dreaded \ used in escape sequences */
WhiteSpace = CHR(9) .. CHR(13) + SP .
inString   = ANY - '"' - CR - LF .           /* strings may not cross line ends */
```

Here are some invalid ones:

```
BadLetter = "A..Z" .                                  /* bad notation */
BadDigit1 = { "0"  | "1"  | "2"  | "3"  | "4"  | "5"  } . /* bad notation */
BadDigit2 = { "0" | "1" | "2" | "3" | "4" | "5"  } . /* bad notation */
BadDigit3 = { "0123456789" } .                        /* bad notation */
BadDigit4 = { 0 1 2 3 4 5 6 7 8 9   } .               /* bad notation */
CRLF      = CR LF  .        /* cannot define a TOKEN in this section */
```

One cannot put the names of CHARACTER sets, or the notations using CHR into the PRODUCTIONS, no matter how tempting this might seem:

```
CHARACTERS
  CR = CHR(13);
...
PRODUCTIONS
  Line
  = "SOMETHING"    /* Here SOMETHING is a string representing a TOKEN, and is allowed */
    CR             /* Wrong: CR is a named singleton CHARACTER set, not a token */
    CHR(10) .      /* Wrong: CHR(n) is restricted to defining CHARACTER sets or TOKENS */
```

10.4.2 Tokens

The important *optional* TOKENS section is typically used to define the patterns of characters that entities like identifiers, strings and numbers are expected to match. That is, it specifies how generic tokens may be constructed. Such tokens are effectively regarded by the parser as terminal symbols, of which there might be many instances. These might all be described in the same way, even though they are, obviously, lexically distinct.

Tokens = "TOKENS" { *Token* } .
Token = *TokenSymbol* ["=" *TokenExpr* "."] .
TokenExpr = *TokenTerm* { "|" *TokenTerm* } .
TokenTerm = *TokenFactor* { *TokenFactor* } ["CONTEXT" "(" *TokenExpr* ")"] .
TokenFactor = *SetIdent*
| *string*
| "(" *TokenExpr* ")"
| "[" *TokenExpr* "]"
| "{" *TokenExpr* "}" .
TokenSymbol = *TokenIdent* | *string* .
TokenIdent = *identifier* .

Tokens may be declared in any order. A token declaration defines a *TokenSymbol* together with the pattern of its structure. Usually the symbol on the left-hand side of each declaration is an identifier, which is then used in other parts of the grammar to denote an instance of the structure defined by the right-hand side expression.

This expression may contain literals denoting themselves (for example "while") or the names of character sets (for example *letter*). It is expressed using the Wirth EBNF notation, but restricted effectively to rules equivalent to regular expressions. Any name that appears in these definitions must identify a CHARACTER *set*, and in this context implies acceptance of "any one character that is a member of that set".

There is one pre-declared token, EOF, which is returned by the scanner when it detects the end of the source file (and returned any time it might be called again thereafter). This is often explicitly used in the PRODUCTIONS section to ensure that a grammar describes all the input up to the end of file (see later examples).

Here are some common examples, using named character sets only:

```
TOKENS
  identifier     = letter { letter | digit } .  /* For example Pat or Terry or Bond007 */
  unsignednumber = digit { digit } .            /* For example 1234 or 51 */
  CRLF           = CR LF .                      /* CR and LF are actually singleton sets! */
```

One can also define tokens by a mixture of characters drawn one at a time from named character sets, as well as explicit strings or characters. Within a token definition these strings represent themselves. Here are some simple examples:

```
TOKENS
  signedNumber   = ( "+" | "-" ) digit { digit } .  /* mandatory + or - followed by digit sequence */
  hexNumber      = digit { hexDigit } "H" .
  string         = '"' { instring | backslash printable } '"' .
  dimension      = digit "-D" .
  ellipsis       = "..." .
```

Token definitions can get quite involved. For example, consider the following:

```
TOKENS
  floatnumber    = [ "+" | "-" ] digit { digit }    /* whole part */
                   "." digit { digit }              /* digits after the mandatory point */
                   [ ( "E" | "e" )                  /* optional exponent part */
                     [ "+" | "-" ]                  /* optional sign within this */
                     digit { digit }                /* mandatory digit sequence */
                   ] .
```

This definition demands that there must be at least one digit before the point, there must be a point, and at least one digit after the point, and can match numbers like +12.3 or 12.4E+5 or 12.4E3 or 1.4e26.

Here is a variation that will match integer or float numbers:

```
TOKENS
  floatnumber    = [ "+" | "-" ] digit { digit }      /* whole part */
                   [                                  /* optional fraction */
                     "." digit { digit }              /* digits after the mandatory point */
                     [ ( "E" | "e" )                  /* optional exponent part */
                       [ "+" | "-" ]                  /* optional sign within this */
                       digit { digit }                /* mandatory digit sequence */
                     ]
                   ] .
```

This will not match numbers like 123E45 or 123E-45. As an exercise, work out what would be needed to allow such representations, as well as the ones we have seen so far.

While token specification is usually straightforward, there are some interesting subtleties.

- The token names introduced on the left of the TOKENS production rules may not be used on the right side of the TOKENS rules in a direct or indirect way. Put another way, one cannot define one token in terms of another token, or recursively in terms of itself. So the following attempt to define a float number is invalid:

  ```
  TOKENS
    integer        = digit { digit } .              /* so far so good */
    floatnumber    = integer "." integer .          /* looks seductive! */
  ```

  The only "names" one can use within the definition part, following the = sign, are names of character sets declared in the CHARACTERS section. However, the token names themselves invariably appear somewhere on the right sides of productions in the PRODUCTIONS section.

- All the token definitions must be given by uniquely distinguishable regular expressions. So one cannot write

```
TOKENS
  firstName = ULetter { LLetter } .
  surname   = ULetter { LLetter } .
  longName  = Uletter LLetter { LLetter } .       /* at least two letters */
```

because all of these amount to describing the same patterns of characters. Note, however, that one could write (if it is appropriate) something like

```
TOKENS
  firstName = ULetter { LLetter } .
  surname   = ULetter { LLetter } "." .
  integer   =   digit { digit }
              | digit { hexDigit } "H" .
```

where the full stop "." is sufficient to distinguish a `surname` (where it must appear) from a `firstName` (where it must not appear), and where an "H" can distinguish an implicitly decimal integer number from an explicitly hexadecimal one.

To sum up, the production rules in the TOKENS section are expressed in EBNF, but they do not have to be in LL(1) form (although they often are).

- Sometimes a token must be described in terms of a single element from a character set. This is a useful device - and in fact the only real way - for defining a terminal token that consists of a control character. By extension one might need to specify a simple sequence of awkward control characters:

```
TOKENS
  EOL  = LF .
  CRLF = CR LF .
```

- A token definition specifies a complete token as a string of contiguous characters. A common mistake is to try to put too much into a token definition - for example

```
TOKENS
  Header = "void main() {" .
```

This sort of construction should be specified in the PRODUCTIONS section in terms of discrete tokens:

```
PRODUCTIONS
  Header = "void"     identifier     "("     ")"     "{" .
```

- In any event, spaces cannot appear within literal strings used to define tokens. If a token really needs to incorporate an internal space (very unusual!), this must be done using something like

```
TOKENS
  Devil = "Pat" SP "Terry" .        /*  not "Pat Terry"  */
```

- Key words are best introduced directly into the PRODUCTIONS section. Thus one would prefer

```
TOKENS
  identifier = letter { letter }.
PRODUCTIONS
  Goal = "keyword" identifier ":=" identifier "anotherKeyword" .
```

to

```
TOKENS
  identifier     = letter { letter }.
  keyword        = "keyword" .          /* a unique keyword, given a token name */
  anotherKeyWord = "anotherkeyWord" .   /* another keyword, given a token name */
  assigned       = ":=" .               /* an operator, given a token name */

PRODUCTIONS
  Goal = keyword identifier assigned identifier anotherKeyword .
```

although the second form is permissible. Of course, a key word is usually nothing other that the sort of

word that might otherwise have been denoted an identifier. Coco/R is clever in this respect, and will have no difficulty telling them apart if the "keyword" is introduced in the PRODUCTIONS section.

- However, if a token *is* to be named and declared as a simple literal that also matches an instance of a more general token, the specific literal has to be declared *after* the more general token in the TOKENS section, as in the example just shown. By contrast, the following is incorrect:

```
TOKENS
  keyword        = "keyword" .          /* a unique keyword */
  anotherKeyWord = "anotherkeyWord" .   /* another keyword */
  identifier     = letter { letter }.   /* too late, too late, too late */
  assigned       = ":=" .               /* an operator */
```

- Although literal tokens like `while` or `switch` do not have to be declared in the `TOKENS` section, it sometimes makes sense to declare some of them explicitly, perhaps to generate a token name for them that can be used in conflict resolver methods (see section 10.5.4).

- As in the case of the *CharacterSets* component, to avoid complications arising from differences in Coco/R implementations, it is recommended that escape sequences not be introduced into any string used as a *TokenFactor*. Use should rather be made of the `CHR(`*i*`)` mechanism as illustrated previously.

- Some characters used in programming languages are overworked. Suppose that there is a need to develop a scanner that can distinguish the tokens in input like

```
3 .. 5.4  +  5.4..16.4  + 50..80
```

where in some contexts a period forms part of a `float` literal, while in others it forms part of an ellipsis. This sort of situation arises quite frequently, and Cocol makes special provision for it. An optional `CONTEXT` phrase in a *TokenTerm* specifies that this term can be matched only when its right-hand context in the input stream matches the *TokenExpr* specified within parentheses. Hence

```
TOKENS
  integer  =   digit { digit }
             | digit { digit } CONTEXT ( ".." ) .
  float    =   digit { digit } "." { digit } .
  ellipsis = ".." .
```

where a digit sequence followed by any character other than a period will be recognized an an `integer`, as will a digit sequence followed immediately by a double period. But a digit sequence followed by a single period and then a non-period will be recognised as a `float` (this `float` may have further digits).

The reader may be intrigued by all this, as the productions here are distinctly non-LL(1). But, as mentioned in section 8.5, the algorithms for parsing a set of regular expressions are not limited to recursive descent, and those used in Coco/R to create the scanner automaton are quite complex.

- The grammar for tokens allows for empty right-hand sides. No scanner is generated if the right-hand side of a declaration is missing. This facility is used if the user wishes to supply a handcrafted scanner, rather than the one generated by Coco/R (perhaps to handle the ability for one source file to "include" another one). In such (rare) cases, all such identifiers and literal tokens must be listed in this section, as exemplified by

```
COMPILER Calculator

TOKENS
  decNumber hexNumber  "="  "+"  "-"  "*"  "/"

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Calculator = { Expression "=" } .
  Expression = Term { "+" Term  |  "-" Term } .
  Term       = Factor { "*" Factor |  "/" Factor } .
  Factor     = decNumber | hexNumber .
END Calculator.
```

- Tokens specified without right-hand sides are numbered consecutively starting from 1 (0 being reserved for the implicit EOF token). A handcrafted scanner has to return token codes matching this numbering scheme.

10.4.3 User names

The scanner and parser produced by Coco/R use small integer values to distinguish token kinds. This makes their code harder to understand by a human reader (some would argue that humans should never need to read such code anyway). When used with appropriate options (typically a `$N` pragma), Coco/R can generate code that uses names for the tokens. By default these names have a rather stereotyped form (for example "..." would be named "pointpointpointSym"). The *UserNames* section may be used to prefer user-defined names, or to help resolve name clashes (for example, between the default names that would be chosen for "point" and ".").

UserNames = "NAMES" { *UserName* } .
UserName = *TokenIdent* "=" (*identifier* | *string*) "." .

As examples we might have

```
NAMES
  period   = "." .
  ellipsis = "..." .
```

(The *UserNames* section is not part of the official Linz releases of Coco/R).

10.4.4 Pragmas

A pragma, like a comment, is a pseudo-token that may occur between any two other tokens in the input stream, but, unlike a comment, it often cannot be ignored completely. Since it becomes impractical to modify a phrase structure grammar to handle this, a special mechanism is provided for the recognition and treatment of pragmas. In Cocol they are declared like tokens but may have an associated semantic action that is executed whenever they are recognized by the scanner.

Pragmas = "PRAGMAS" { *Pragma* } .
Pragma = *Token* [*Action*] .
Action = "(." *arbitraryText* ".)" .

Pragmas might be used to allow programmers to select compiler switches. As an example, we might have in a system that we were testing:

```
PRAGMAS
  debugOn  = "$D+" .  (. Parser.debugMode = true;  .)    // Note: the action follows the period!
  debugOff = "$D-" .  (. Parser.debugMode = false; .)
```

10.4.5 Comments

Comments are difficult to specify with the regular expressions used to denote tokens - indeed, nested comments may not be specified at all in this way. Since comments are usually discarded by a parsing process, and may typically appear in arbitrary places in source code, it makes sense to have a special construct for them.

The *optional* COMMENTS section allows one to specify the format of comments:

Comments = "COMMENTS" "FROM" *TokenExpr* "TO" *TokenExpr* ["NESTED"] .

where the optional keyword `NESTED` should have an obvious meaning. It is possible to define more than one structure for comments within a single grammar, for example, for C# or Java

```
COMMENTS FROM "(*" TO "*)" NESTED  /* Modula-2 style nested comments */
COMMENTS FROM "/*" TO "*/"         /* Java style comments - 1 */
COMMENTS FROM "//" TO LF           /* Java style comments - 2 */
COMMENTS FROM ";"  TO LF           /* Assembler style comments */
```

In the last two examples `LF` would have been defined under the TOKENS section as shown earlier.

As before, there are some subtleties to observe.

- Comment delimiters may not be longer than 2 characters.

- If one wishes to use delimiters with more than 2 characters one can make use of a PRAGMA definition instead (for example, in the case of HTML).

- Similarly, if for some reason one needs to process the comment text rather than merely discard it, one can use a PRAGMA definition instead.

10.4.6 Whitespace and ignorable aspects

Coco/R *optionally* allows one to specify that a set of characters is "ignorable", meaning that members of this set will be skipped over each time the scanner looks for the *next* token. The basic space character (SP, CHR(32)) is always in this set. Usually all the control characters - or at least the whitespace control characters - are also ignorable, meaning that they can be freely used to separate different tokens in the manner familiar to all Java and C# programmers.

Case-insensitive parsers/scanners can be constructed by using the keyword `IGNORECASE`, as shown below. If this is required, the `IGNORECASE` directive must appear before any others.

IgnoreCase = "`IGNORECASE`" .
WhiteSpace = "`IGNORE`" *CharacterSet* .

Examples

```
COMPILER Example
  IGNORECASE                          /* case insensitive scanner */
  CHARACTERS
    Control = CHR(0) .. CHR(31) .

  ...  /* CHARACTERS, TOKENS sections, followed by */

  IGNORE Control .                    /* control characters as above */
  IGNORE CHR(9) + CHR(11) .. CHR(13) .  /* white space, but not LF */
  IGNORE CHR(9) .. CHR(13) .          /* all white space */
```

Note that

- The effect of `IGNORECASE` is that all literals that occur in the productions or in the token declarations are considered to be case insensitive. For example, after introducing `IGNORECASE`, the production

  ```
  WhileStatement = "while" "(" Expression ")" Statement .
  ```

  will recognize a *WhileStatement* that starts with `while` or `While` or `WHILE`.

- However, the casing of letters that occur in identifiers, strings or numbers is never changed by the scanner, even if `IGNORECASE` is specified. If users want to treat them in a case insensitive way they have to convert the lexemes to all upper-case (or lower-case) themselves.

- Because a character is ignorable does not mean that it cannot form an *internal* part of a token, but an ignorable character cannot be the *initial* character of a token.

10.4.7 Interfacing to the Scanner and Token classes

The specification of the *Scanner* class generated by Coco/R declares various `public static` methods.

```
public class Scanner {

  public static void Init(string fileName)
  // Open and read source file specified by fileName

  public static Token Scan()
  // Returns next token from source file
```

```
  public static Token Peek()
  // Returns next token from source file but does not consume it

  public static Token ResetPeek()
  // Resets the scan position after calling Peek

} // class Scanner
```

Normally one makes little direct use of this specification, as the generated parser incorporates all the necessary calls to the scanner routines automatically. Of more interest is the specification of the *Token* class, since the fields here are sometimes accessed by semantic actions in the parser routines.

```
public class Token {
  public int kind;   // token kind
  public int pos;    // token position in source (starting at 0)
  public int col;    // token column (starting at 0)
  public int line;   // token line (starting at 1)
  public string val; // token lexeme
} // class Token
```

As already mentioned, Coco/R enumerates the various token kinds internally. Besides the lexeme itself, a `Token` object also stores information about its position in the source file and its location as the user would see it. The line and column number can be used to associate error messages with an offending token very accurately.

If users supply their own scanner algorithms - as discussed in section 10.4.2 - this must be done in such a way that *Scanner* and *Token* classes meet the specifications given above.

10.5 Parser specification

The parser specification is the main part of the input to Coco/R. It contains the productions of an attributed grammar specifying the syntax of the language to be recognized, as well as the action to be taken as each phrase or token is recognized.

10.5.1 Productions

The form of the parser specification may itself be described in EBNF as follows.

ParserSpecification = "PRODUCTIONS" { *Production* } .
Production = *NonTerminal* [*FormalAttributes*]
[*Initialization*]
"=" *Expression* "." .
FormalAttributes = "<" *arbitraryText* ">"
| "<." *arbitraryText* ".>" .
Initialization = "(." *arbitraryText* ".)" .
NonTerminal = *identifier* .

All Cocol grammars must have a PRODUCTIONS section, and there must be a production for the `Goal` symbol, which is also used as the name of the grammar. The shortest possible Cocol specification (which, of course, does nothing useful) would thus be

```
COMPILER Goal
PRODUCTIONS
  Goal = .
END Goal.
```

Any identifier appearing in a production that was not previously declared as a terminal token is considered to be the name of a *NonTerminal*, and there must be exactly one *Production* for each *NonTerminal* that is used in the specification (this may, of course, specify a list of alternative right-hand sides).

Each production may be considered as a specification for creating a method that parses the *NonTerminal*. This method will constitute its own scope for parameters and other local components like variables and constants. The left-hand side of a *Production* specifies the name of the *NonTerminal* as well as its *FormalAttributes* (which effectively specify the formal parameters of the method). The optional *Initialization* allows for the declaration of local variables and other statements to precede those generated by analyzing the *Expression* of the production.

Cocol can only be used for context-free grammars, so that the left side of each production must always be a single non-terminal. The right side *can* include

- Terminals represented directly by strings - for example `"void"`
- The names of terminal tokens named under the TOKENS section - for example `Number`
- The names of other non-terminals - for example `Expression`
- The meta-characters | { } [ ] and ( ) .
- In various contexts, code fragments enclosed in metabrackets < … >, <. … .> and (. … .). Of course, these fragments are by no means completely arbitrary!

The right side *cannot* include

- The names of character sets as defined in the CHARACTERS section, however tempting it might be to do so. Use an intermediate token definition, as illustrated previously.

A typical simple production might be

```
Factor = number | [ "sqrt" ] "(" Expression ")" .
```

As in the case of tokens, some subtleties in the specification of productions should be emphasized.

- The productions can be given in any order.

- A production must be given for a *GoalIdentifier* that matches the name used for the grammar.

- The *FormalAttributes* enclosed in angle brackets "<" and ">" (or "<." and ".>") simply consist of parameter declarations in the host language. Similarly, when used, the *Initialization* takes the form of host language statements enclosed in "(." and ".)" brackets. However, the syntax of these components is not checked by Coco/R - this is the responsibility of the compiler that will actually compile the generated application.

- In the C# version of Coco/R, all productions give rise to *void methods* (in C# terminology). Any synthesized or transmitted attributes must be handled by using the *out* or *ref* parameter mechanisms.

- In Java, parameters can only be passed by value. To overcome this rather nasty restriction, the formal attributes in the Java version of Cocol may be of the form

  FormalAttributes = "<" ["↑" | "out"] *arbitraryText* ">"
  | "<." ["↑" | "out"] *arbitraryText* ".>" .

  The first attribute in an *FormalAttribute* declaration is considered to be an *output attribute* if it is preceded by the caret character '↑' or the keyword `out` . This output attribute is translated into a function return value, thus providing a mechanism for a production to return a single synthesized value (which can be a reference to an object of a suitable class if necessary). For example, the production

  ```
  SomeNonTerminal <out SomeType t, String name, int i> = ... .
  ```

  is translated into

  ```
  static SomeType SomeNonTerminal(String name, int i) {
    SomeType t;
    ...
    return t;
  }
  ```

- The goal symbol may not have any *FormalAttributes*. Any information that must be passed between the parser and the main driver method must be handled in other ways, perhaps by declaring `public static` fields in the parser class as discussed in section 10.3.

- It may happen that an identifier chosen as the name of a *NonTerminal* will clash with one of the internal names used in the rest of the system. Such clashes will only become apparent when the application is compiled and linked, and may require the user to redefine the grammar to use other identifiers.

- A production like

```
Goal = { Something } EOF .
```

  is often (but not always) preferable to the simpler, less explicit

```
Goal = { Something } .
```

  The first form will signal an error if the sequence of `Something`s does not reach the end of file, the second form might just stop prematurely and leave the user wondering what happened.

The *Expression* on the right-hand-side of each *Production* defines the context-free structure of some part of the source language, together with the attributes and semantic actions that specify how the parser must react to the recognition of each component. The syntax of an *Expression* may itself be described in EBNF (albeit not in LL(1) form - can you see why not?) as

```
Expression  =  Term { "|" Term } .
Term        =  [ [ Resolver ] Factor { Factor } ] .
Factor      =    [ "WEAK" ] TokenSymbol
               | NonTerminal [ Attributes ]
               | Action
               | "ANY"
               | "SYNC"
               | "(" Expression ")"
               | "[" Expression "]"
               | "{" Expression "}" .
Attributes  =    "<"  arbitraryText ">"
               | "<." arbitraryText ".>" .
Action      =  "(." arbitraryText ".)" .
Resolver    =  "IF" "(" arbitraryText ")" .
```

The *Attributes* (enclosed in angle brackets) that may follow a *NonTerminal* in this context effectively denote the actual parameters that will be used in calling the corresponding method. If a *NonTerminal* is defined on the left-hand side of a *Production* to have *FormalAttributes*, then every occurrence of that *NonTerminal* in a right-hand side *Expression* must have a list of actual *Attributes* that correspond to the *FormalAttributes* according to the parameter compatibility rules of the host language. However, the conformance is checked only when the generated parser is itself compiled.

An *Action* is an arbitrary sequence of host language statements enclosed in `"(."` and `".)"`. These are simply incorporated into the generated parser *in situ* - once again, no syntax is checked at that stage.

In the Java version the *Attributes* associated with a *NonTerminal* may take the form

```
Attributes  =    "<"  [ "↑" | "out" ] arbitraryText ">"
               | "<." [ "↑" | "out" ] arbitraryText ".>" .
```

where the leading ↑ or the keyword `out` is required if it appears in the corresponding *FormalAttributes*.

These points may be made clearer by considering a further development of our simple calculator, which, hopefully, needs little further explanation.

```
PRODUCTIONS
  Calculator                      (. int value; .)
  = { Expression<out value> "="   (. IO.Write(value); .)
    } EOF .

  Expression<out int expVal>      (. int termVal; .)
  = Term<out expVal>
    {   "+" Term<out termVal>     (. expVal += termVal; .)
      | "-" Term<out termVal>     (. expVal -= termVal; .)
    } .
```

```
Term<out int termVal>           (. int factVal; .)
= Factor<out termVal>
  {   "*" Factor<out factVal>   (. termVal *= factVal; .)
    | "/" Factor<out factVal>   (. termVal /= factVal; .)
  } .

Factor<out int factVal>         (. int factVal = 0; .)
=   decNumber                   (. factVal = Convert.ToInt32(token.val, 10); .)
  | hexNumber                   (. factval = Convert.ToInt32(token.val.Substring(1), 16); .) .

END Calculator.
```

Although the input to Coco/R is free-format, it is recommended that a grammar file be set out with the regular EBNF appearing on the left and the actions lined up on the right, as in the example above.

Many aspects of parser specification are straightforward, but there are some subtleties that call for comment.

- The WEAK and SYNC keywords are used in error recovery, as discussed in the next section.

- Coco/R allows for attributes to be demarcated by "<." and ".>" as an alternative to the "<" and ">" brackets, to allow for situations like the following, where it is necessary to use the > character as part of an operator within the expression that forms an actual argument of the parsing routine.

  ```
  SomeNonTerminal<. first > second .>
  ```

- Productions may be recursive, but left recursive grammars are intrinsically non-LL(1) and thus rarely of any use. In general, Coco/R only generates usable parsers if the PRODUCTIONS section defines an LL(1) grammar. There are a few exceptions to this - for example the "dangling else" construction is non-LL(1), but a Coco/R generated parser does the sensible thing.

- Be careful of the precedence rules - alternation | is "weaker" than concatenation, so that

  ```
  A = B C | D E .
  ```

  means

  ```
  A = ( B C )       |       ( D E ) .
  ```

  and not

  ```
  A = B       ( C | D )       E .
  ```

  It is easy to make such mistakes and, as with forgetting to put braces round blocks of statements in Java programs, it can lead to very-hard-to-find bugs. The production below is acceptable to Coco/R, but is (hopefully obviously) *wrong*:

  ```
  IfStatement = "if" "(" Condition ")" Statement "else" Statement | .
  ```

- Be careful to avoid introducing optional parts in too many ways. This sometimes happens accidentally, often indirectly. For example, do not fall into the trap of writing productions like

  ```
  Complete      = [ { Part1 } { Part2 } ] .
  Something     = [ [ Inner ] ] .
  LotsOfOptions = { [ First ] [ Second ] [ Third ] } .
  WholePart     = [ Part1 Part2 ] .
  Part1         = { Something } .
  Part2         = [ Part3 ] .
  ```

- Empty alternatives can be defined using square brackets (the usual way), or by using a | followed by nothing, for example:

  ```
  IfStatement = "if" "(" Condition ")" Statement [ "else" Statement ] .
  IfStatement = "if" "(" Condition ")" Statement ( "else" Statement | ) .
  ```

  This feature is often useful. Close perusal of the grammar for *Expression* will reveal that it is legal to write

a *Production* in which an *Action* appears to be associated with an alternative for an *Expression* that contains no terminals or non-terminals at all. For example, we might have

```
Option =   "push" (. stack[++top] = item; .)
         | "pop"  (. item = stack[top--]; .)
         |        (. MonitorStack(); .) .
```

where `MonitorStack()` will be invoked if `Option` is derived to an empty string.

- It may happen that an identifier chosen as a local variable or parameter in the actions and attributes will clash with one of the internal names used in the rest of the system. Such clashes will become apparent only when the application is compiled and linked, and may require the user to redefine the grammar to use other identifiers.

- The keyword `ANY` can be used as a *Factor* to denote any terminal that cannot be an alternative to `ANY` in that context. For example, one could write

```
A = "a" | ANY .
```

where `ANY` denotes any terminal symbol except `"a"`, or

```
A = [ "a" | ANY ] "b" .
```

where `ANY` denotes any terminal symbol except `"a"` or `"b"`.

This feature can conveniently be used to parse structures that contain arbitrary text. For example, if we wished to be able to mark the source of the attributes within a grammar resembling the one for Coco itself, we might proceed as follows.

```
ArbitraryText<out TextMark marker>
= "<"     (. int start = token.pos + 1; .)
  { ANY }
  ">"     (. marker = new TextMark(start, token.pos - start); .) .
```

Here the closing angle bracket `">"` is an implicit alternative of the `ANY` symbol in braces. The effect is that `ANY` matches any terminal except `">"`. The `token` field in the generated parser contains the most recently recognized token (see section 10.5.6).

10.5.2 Syntax error recovery

Compiler generators vary tremendously in the way in which they provide for recovery from syntactic errors, a subject that was discussed in section 8.4.

The follower set technique described there, although systematically applicable, slows down error-free parsing, inflates the parser code and is relatively difficult to automate. Coco/R uses a simpler technique, as suggested by Wirth (1986), that has proved to be almost as effective and is very easily understood. Recovery takes place only at a rather small number of **synchronization points** in the grammar. Errors at other points are reported, but cause no recovery - parsing simply continues up to the next synchronization point. One consequence of this simplification is that many spurious errors are then likely to be detected for as long as the parser and the input remain out of step. Coco/R uses an effective technique for handling this - errors are simply not reported if they follow too closely upon one another (that is, a minimum amount of text must be correctly parsed after one error is detected before the next can be reported).

In the simplest approach to using this technique, the designer of the grammar is required to specify synchronization points explicitly. As it happens, this rarely turns out to be a difficult task - the usual heuristic is to choose locations in the grammar where especially safe terminals are expected that are hardly ever missing or mistyped, or appear so often in source code that they are bound to be encountered again at some stage. In many languages, good opportunities for synchronization points are found at the beginning of a statement (where keywords like `if` and `while` are expected), the beginning of a declaration sequence (where keywords like `int` and `bool` are expected) and the beginning of a type definition (where keywords like `class` are expected).

In Cocol, a synchronization point is specified by the keyword SYNC, and the effect is to generate code for a loop that is prepared simply to consume source tokens until one is found that would be acceptable at that point. The sets of such terminals can be precomputed at parser generation time. They are always extended to include the end-of-file symbol, guaranteeing that, if all else fails, synchronization will succeed at the end of the source text.

As a simple, if somewhat contrived, example consider the following:

```
Folly = Speed { "+" Speed } SYNC ( "crash" | "roll" ) .
```

This would have the effect of generating code for the Folly parser that corresponds to the algorithm

```
BEGIN
  Speed;
  WHILE sym.kind = plusSym DO GetSym; Speed END;
  WHILE sym.kind ∉ { crashSym, rollSym, EOFSym } DO
    ReportError("unexpected symbol"); GetSym
  END;
  IF sym.kind ∈ { crashSym, rollSym }
    THEN GetSym
    ELSE ReportError("invalid Folly")
  END
END
```

The union of all the synchronization sets (which we shall denote by *AllSyncs*) is also computed by Coco/R and is used in further refinements on this idea. A terminal can be designated to be *weak* in a certain context by preceding its appearance in the phrase structure grammar with the keyword WEAK. A weak terminal is one that might often be mistyped or omitted, such as the semicolon between statements. When the parser expects (but does not find) a weak terminal, it adopts the strategy of consuming source tokens until it recognizes either a legal successor of the weak terminal or one of the members of *AllSyncs*. Since terminals expected at synchronization points are considered to be very "strong", it makes sense that they never be skipped in any error recovery process.

As an example of how this could be used, consider another rather artificial example:

```
ShoppingSpree = WEAK "trolley" Load { Load } WEAK "swipe" "pay" .
Load          = "coffee" | "noodle" .
```

This would give rise to code for the ShoppingSpree parser that could be described algorithmically as

```
BEGIN
  ExpectWeak(trolleySym, FIRST(Load));
      (* ie { coffeeSym, noodleSym } *)
  Load;
  WHILE sym.kind ∈ { coffeeSym, noodleSym } DO Load END;
  ExpectWeak(swipeSym, { paySym, EOFSym })
END
```

where the ExpectWeak routine could be described algorithmically as

```
PROCEDURE ExpectWeak (ExpectedTerminal, WeakFollowers);
  BEGIN
    IF sym.kind = ExpectedTerminal
      THEN GetSym
      ELSE
        ReportError(ExpectedTerminal);
        WHILE sym.kind ∉ (WeakFollowers ∪ AllSyncs) DO GetSym END
    END
  END
```

Weak terminals give the parser another chance to synchronize in case of an error. The WeakFollower sets can be precomputed at parser generation time, and the technique causes no run-time overhead if the input is error-free.

Frequently iterations start with a weak terminal which, if omitted in the text being parsed, would result in a loss of synchronization probably confusingly. Marking such a terminal WEAK, as exemplified by

Sequence = *FirstPart* { WEAK *ExpectedTerminal* *IteratedPart* } *LastPart* .

introduces the concept of *weak separators*, which are handled in a special way. If the *ExpectedTerminal* cannot be recognized, source tokens are consumed until a terminal is found that is contained in one of the following sets

FOLLOW(*ExpectedTerminal*) (that is, FIRST(*IteratedPart*))
FIRST(*LastPart*)
AllSyncs

As an example of this, suppose we were to modify our case study grammar to read

```
Folly = Speed { WEAK "+" Speed } ( "crash" | "roll" ) .
Speed = "faster" | "turbo" .
```

The algorithm for the `Folly` parser would be

```
BEGIN
  Speed;
  WHILE WeakSeparator(plusSym,
                      { fasterSym, turboSym },
                      { crashSym, rollSym } ) DO
    Speed
  END;
  IF Sym ∈ {crashSym, rollSym } THEN GetSym END
END
```

where the algorithm for the `WeakSeparator` routine would be

```
BOOLEAN FUNCTION WeakSeparator (Expected,
                                WeakFollowers,
                                IterationFollowers);
  BEGIN
    IF sym.kind = Expected THEN GetSym; RETURN TRUE
      ELSIF sym.kind ∈ IterationFollowers THEN RETURN FALSE
      ELSE
        ReportError(Expected);
        WHILE sym.kind ∉ (WeakFollowers ∪ IterationFollowers ∪ AllSyncs) DO
          GetSym
        END;
        RETURN sym.kind ∈ WeakFollowers
    END
  END
```

Once again, all the necessary sets can be precomputed at generation time. Occasionally, in highly embedded grammars, the inclusion of *AllSyncs* (which tends to be "large") may detract from the efficacy of the technique, but with careful choice of the placing of `WEAK` and `SYNC` keywords it can work remarkably well.

Note that any occurrence of the keyword `WEAK` must immediately precede a **terminal**. An easy mistake is to try to insert `WEAK` before a non-terminal, or before an action or attribute.

10.5.3 Grammar checks

Coco/R performs several tests to check that the grammar submitted to it is well-formed. In particular, it checks that:

- each non-terminal has been defined by exactly one production;
- there are no useless productions (in the sense discussed in section 6.3.1);
- the grammar is cycle-free (in the sense discussed in section 6.3.3);
- all tokens can be distinguished from one another (that is, no ambiguity exists in the specification of the token classes and literal terminals).

If any of these tests fail, no code generation takes place. In other respects the system is more lenient. Coco/R issues warnings if analysis of the grammar reveals that:

- a non-terminal is nullable (this occurs frequently in correct grammars, but may sometimes be indicative of an error);
- the LL(1) conditions are violated, either because at least two alternatives for a production have FIRST sets with elements in common, or because the FIRST and FOLLOWER sets for a nullable string have elements in common.

10.5.4 LL(1) conflict resolution

If Coco/R reports an LL(1) warning for a construct that involves alternatives or iterations, users should be aware that the generated parser is highly likely to behave in a way that they may not have foreseen. As simple examples, productions like the following

```
P = "a" A | "a" B .
Q = [ "c" B ] "c"  .
R = { "d" C } "d" .
```

result in generation of syntactically acceptable code that can be described algorithmically as

```
IF Sym = "a" THEN Accept("a"); A
  ELSIF Sym = "a" THEN Accept("a"); B
END;

IF Sym = "c" THEN Accept("c"); B END;
Accept("c");

WHILE Sym = "d" DO Accept("d"); C END;
Accept("d");
```

In the case of an LL(1) warning, the parser generated by Coco/R will always select the first matching alternative. In situations like the second of the above examples, which corresponds to the problem of the "dangling else", this may be what the user expects. In others - as exemplified by the first and last of these examples - it is usually necessary to redesign the grammar to eliminate the LL(1) conflict.

As we have already commented in an earlier chapter, many LL(1) conflicts can be resolved by factorization. A typical example is to be found in a Pascal-like language, where the most natural way to define statements might tempt one to write a non-LL(1) production like

```
AssignmentOrCall =   Identifier "=" Expression  |  Identifier .
```

This is easily recast as

```
AssignmentOrCall = Identifier [ "=" Expression ] .
```

although, clearly, the semantics of the two statement forms are very different. In a situation like this, where recourse to a symbol table is somewhat inevitable, a feature that can sometimes be exploited in Coco/R is the ability of an *Action* to drive the parsing process "semantically". For example, we might proceed as follows.

```
AssignmentOrCall             (. string name; int value; .)
= Identifier<out name>       (. if (IsProcedure(name)) {
                                   Call(name); return;
                                } .)
  "="
  Expression<out value>      (. Assignment(name, value); .) .
```

but this is not always convenient or possible, as can easily be seen if the grammar were to be replaced by

```
AssignmentOrCall = Identifier [ "=" Expression ] ";" .
```

but even here it is easy to write Cocol code like the following - note the interesting example of associating an action with an "empty" alternative, a trick that can be usefully employed in many situations.

```
AssignmentOrCall             (. string name; int value; .)
= Identifier<out name>
  (   "="
      Expression<out value>  (. Assignment(name, value); .)
    |                        (. Call(name); .)
  ) ";" .
```

The latest version of Coco/R from Linz incorporates other features for resolving LL(1) conflicts by using so-called **conflict resolvers** to look ahead in the input (Wöß *et al.* 2003). Although we shall make little use of them in the remainder of this text, some examples showing their use should be of interest.

A conflict resolver incorporates a Boolean expression and is inserted into the grammar at the beginning of the first

of two conflicting alternatives to decide, using a multi-symbol lookahead or a semantic check, whether this alternative matches the actual input. If the resolver yields `true`, the alternative prefixed by the conflict resolver is selected, otherwise the next alternative will be checked. For the same grammar we might use

```
AssignmentOrCall            (. string name; int value; .)
= ( IF (IsAssignStatement())
      Identifier<out name>
      "="
      Expression<out value> (. Assignment(name, value); .)
    | Identifier<out name>  (. Call(name); .)
  ) ";" .
```

In most cases the conflict resolver will incorporate a call to a static Boolean function defined local to the parser. For the above example this might be

```
static bool IsAssignStatement() {
  Token next = Scanner.Peek();
  return la.kind == _ident && next.kind == _assign;
} // IsAssignStatement

TOKENS
  ident  = letter { letter } .
  assign = "=" .
```

where it will be noted that the `IsAssignmentStatement` method has made use of the `Peek` method of the scanner to look two tokens ahead at the point where the conflict occurs. The `la` field of the parser contains the look ahead token (see section 10.5.6), and the code has also made use of the fact that Coco/R generates alias names (each prefixed by an underscore) for all tokens named in the `TOKENS` section of the scanner description.

This example may not be very convincing, as it is just as easily handled by factorising the grammar. Another example may demonstrate the power of the system better.

Consider the following LL(1) conflict between type casts and nested expressions, which can be found in many programming languages.

```
Factor =   '(' Identifier ')' Factor   /* type cast */
         | '(' Expression ')'          /* nested expression */
         | Identifier .
```

An *Expression* can start with an *Identifier* - there is no way to resolve the conflict by simple factorization. However, the conflict can be resolved by checking whether the identifier denotes a type or some other entity.

```
Factor =   IF (IsCast())
           '(' Identifier ')' Factor /* type cast */
         | '(' Expression ')'        /* nested expression */
         | Identifier .
```

Here the method `IsCast()` is prepared to look up the identifier in the symbol table and returns `true` if it is recognized as the name of a type:

```
static bool IsCast() {
// la.val must be ( or an identifier at this stage
  Token next = Scanner.Peek();       // look past it
  if (next.kind == _ident) {         // was an identifier - do we recognize it?
    Entry e = Table.Find(next.val);  // check symbol table
    return e != null && IsTypeName(e.name);
  }
  else return false;                 // not an identifier, so not a cast!
} // IsCast
```

Further examples of the use of conflict resolvers and the precautions that must be taken in applying them may be found in the documentation for the Linz release of Coco/R in the *Resource Kit* and in Chapter 14.

10.5.5 Semantic errors

The parsers generated by Coco/R handle the reporting of syntax errors automatically. The default driver

programs can summarize these errors at the end of compilation, along with source code line and column references, or produce source code listings with the errors clearly marked with explanatory messages (an example of such a listing appears in section 10.5.8). Pure syntax analysis cannot reveal static semantic errors, but Coco/R supports a mechanism whereby the grammar designer can arrange for such errors to be reported in the same style as is used for syntactic errors. The parser class includes a static method that can be called from within the semantic actions, using an explanatory string as an argument that describes the error.

In the grammar of our simple calculator, for example, it might make sense to introduce a semantic check into the actions for an attempt to divide by zero.

```
PRODUCTIONS
  ...

  Term<out int termVal>           (. int factVal; .)
  = Factor<out termVal>
    {   "*" Factor<out factVal)   (. termVal *= factVal; .)
      | "/" Factor<out factVal)   (. if (factVal == 0
                                       SemError("Division by zero");
                                     else termVal /= factVal; .)
    } .
```

10.5.6 Interfacing to the Parser class

The *Parser* class generated by Coco/R is placed in a namespace or package named by the goal symbol of the grammar. As well as several private fields and methods, it defines various public methods that may be called from an application. The interface is conveniently summarized below.

```
using System;

namespace Goal {

  public class Parser {

    public static void Parse()
    // Parses the source

    public static bool Successful()
    // Returns true if no errors occurred while parsing

    public static void SemError(string s)
    // Reports semantic error denoted by s

    public static void Warning(string s)
    // Reports warning message denoted by s

    private static Token token;   // last recognized token
    private static Token la;      // look ahead token

    public static string LexString()
    // Returns lexeme token.val

    public static string LookAheadString()
    // Returns lexeme la.val

  } // class Parser

} // end namespace
```

The functionality provides for an application to:

- initiate the parse for the goal symbol by calling `Parse()`;
- investigate whether the parse succeeded by calling `Successful()`;
- report on the presence of semantic errors by calling `SemError(errorMessage)`;
- issue warning messages (as opposed to error messages) by calling `Warning(warningMessage)`;
- obtain the lexeme value of a particular token (`LexString`, `LookAheadString`).

The methods for retrieving a lexeme are provided mainly for compatibility with earlier versions of Coco/R which were not object-oriented and did not define a `Token` class.

10.5.7 Interfacing to support classes

It will not have escaped the reader's attention that code specified in the actions of an attributed grammar will frequently need to make use of routines that are not defined by the grammar itself. Two typical situations are exemplified in our case study.

Firstly, it has seen fit to make use of the `IO.WriteLine` method from the `Library` namespace (or the equivalent method in the Java version). To make use of such routines - or ones defined in other support libraries that the application may need - it is necessary simply to prefix the grammar with the appropriate `using` or `import` clauses, as discussed in section 10.4.

Secondly, the need arises in routines like `Factor` to be able to convert a lexeme, recognized by the scanner as corresponding to a number, into a numerical value that can be passed back via a formal parameter to the calling routine (`Speed`). This is conveniently accomplished by accessing the private field `token`, as exemplified by the code below.

Our case study is deliberately very small. More realistic applications usually require the development of other classes - for example, to handle code generation in a complete compiler. These classes are normally placed in the same namespace or package as is defined by the goal symbol.

10.5.8 A complete example

To place all of the ideas of the last sections in context, we present a complete C# version of the attributed grammar for our case study:

```
using Library;

COMPILER Calculator  $CN
/* Simple four function calculator */

CHARACTERS
  digit      = "0123456789" .
  hexdigit   = "ABCDEF" + "abcdefg" + digit .

TOKENS
  decNumber  = digit { digit } .
  hexNumber  = "$" hexdigit { hexdigit } .

IGNORE CHR(0) .. CHR(31)

  PRODUCTIONS
    Calculator                      (. int value; .)
    = { Expression<out value>
        WEAK "="                    (. IO.Write(value); .)
      } EOF .

    Expression<out int expVal>      (. int termVal; .)
    = Term<out expVal>
      {   "+" Term<out termVal>     (. expVal += termVal; .)
        | "-" Term<out termVal>     (. expVal -= termVal; .)
      } .

    Term<out int termVal>           (. int factVal; .)
    = Factor<out termVal>
      {   "*" Factor<out factVal>   (. termVal *= factVal; .)
        | "/" Factor<out factVal>   (. if (factVal == 0)
                                        SemError("Division by zero");
                                      else termVal /= factVal; .)
      } .

    Factor<out int factVal>         (. int intBase = 10;
                                       string str = null;
                                       factVal = 0; .)
    = (   decNumber                 (. intBase = 10; str = token.val; .)
        | hexNumber                 (. intBase = 16; str = token.val.Substring(1); .)
      )                             (. try {
                                         factVal = Convert.ToInt32(str, intBase);
                                       } catch (Exception) {
                                         SemError("Number out of range");
                                       } .) .
  END Calculator.
```

One further point can be made about the above code. Initializing `str` to `null` and `factVal` to zero may seem unnecessary, as a value for *factVal* should surely always be computed! However, if the scanner fails to recognize either a syntactically correct *decNumber* or *hexNumber*, no such value will be computed. This would be detected by the C# or Java compiler, which will refuse to compile code that has the possibility of not assigning a value to an `out` parameter, or of reaching the end of a non-void method before executing a `return` statement that defines the value to be returned.

To show how errors are reported, we show the output from applying the generated system to input that is fairly obviously incorrect.

```
   1   1 + 2 =
   2   3 - * 4 =
****       ↑ invalid Factor
   3   $12 / 0 =
****         ↑ Division by zero
   4   3 + 4
   5   5 + 6 =
****   ↑ "=" expected
   6   $FfFfffFFFFFFFAB =
****   ↑ Number out of range

4 errors detected
```

10.6 The driver program

The most important tasks that Coco/R has to perform are the construction of the scanner and parser. However, these must be incorporated into a complete program before they become useful.

10.6.1 Essentials of the driver program

Any main routine for a driver program must be a refinement of ideas that can be summarized as

```
BEGIN
  ProcessCommandLineParameters;
  IF Okay THEN
    InstallErrorHandler;
    InitializeScanner;
    InitializeSupportModules;
    Parse;
    SummarizeErrors;
    IF Successful() THEN ApplicationSpecificAction END
  END
END
```

Much of this can be automated, of course, and Coco/R can generate such a driver class, consistent with its other components. To do so requires the use of an appropriate frame file - a generic version of this is supplied with the distribution. Although it may be suitable for constructing simple prototypes, it acts best as a model from which an application-specific frame file can easily be derived. A user is not bound to use the driver class that Coco/R offers - in many cases it might be preferable to develop a driver class in some other way.

10.6.2 Customizing the driver frame file

A customized driver frame file generally requires at least three simple additions.

- It may be necessary to add application-specific `using` or `import` clauses, so that the necessary library support will be provided.

- The default driver file shows one example of how the command-line option `-l` may be detected and used to choose between reporting errors in a very simple form to standard output, or doing so in the way suggested in section 10.5.8 where error messages were merged with the source text in a convenient way. Other command-line parameters may be needed and can be processed in similar ways.

- At the end of the default frame file can be found code like

```
Scanner.Init(inputName);
Errors.Init(inputName, dir, mergeErrors);
//  ---------------------- add other initialization if required:
Parser.Parse();
Errors.Summarize();
//  ---------------------- add other finalization if required:
```

the intention of which should be almost self-explanatory. For example, in the case of a compiler/interpreter such as we shall discuss in Chapter 14, we might modify this to read something like

```
Scanner.Init(inputName);
Errors.Init(inputName, dir, mergeErrors);
PVM.Init();
Table.Init();
Parser.Parse();
Errors.Summarize();

bool compiledOK = Parser.Successful();
int initSP = CodeGen.GetInitSP();
int codeLength = CodeGen.GetCodeLength();
PVM.ListCode(codeName, codeLength);
if (!compiledOK || codeLength == 0) {
  Console.WriteLine("Unable to interpret code");
  System.Environment.Exit(1);
}
else {
  Console.WriteLine("Interpreting code ...");
  PVM.Interpret(codeLength, initSP);
}
```

10.7 Developing token definitions

It is sometimes quite difficult to get token definitions correct. The following simple grammar may prove to be of use. It constructs a system that can read a file of tokens and report on the type and the spelling of each one. So, for example, if it were presented with a file reading

```
1234  abcdef  567   "a string"   'c'   ++  --
```

it should report that it finds a number, an identifier, a string, a character and then some unrecognizable tokens. One can adapt this quite easily, for example by adding token definitions for float numbers, key words from a programming language and so on.

```
using Library;

COMPILER TokenTests $CN
/* Test scanner construction and token definitions - C# version
   The generated program will read a file of words, numbers, strings etc
   and report on what characters have been scanned to give a token,
   and what that token is (as a magic number).  Useful for experimenting
   when faced with the problem of defining awkward tokens!

   P.D. Terry, Rhodes University, 2015 */

/* Some code to be added to the parser class */

  static void Display(Token token) {
  // Simply reflect the fields of token to the standard outpul
    IO.Write("Line ");
    IO.Write(token.line, 4);
    IO.Write(" Column");
    IO.Write(token.col, 4);
    IO.Write(":  Kind");
    IO.Write(token.kind, 3);
    IO.WriteLine("  Val |" + token.val.Trim() + "|");
  } // Display

CHARACTERS
  sp         = CHR(32) .
  backslash  = CHR(92) .
  control    = CHR(0) .. CHR(31) .
  letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit      = "0123456789" .
  stringCh   = ANY - '"' - control - backslash .
  charCh     = ANY - "'" - control - backslash .
  printable  = ANY - control .
```

```
  /* you may like to introduce others */

TOKENS

  ident      = letter { letter | digit } .
  number     = digit { digit } .
  stringLit  = '"' { stringCh | backslash printable } '"' .
  charLit    = "'" ( charCh   | backslash printable ) "'" .

  /* you may like to introduce others */

IGNORE control

PRODUCTIONS

   TokenTests
   = { (    ANY
          | "."
          | ".."
          | "ANY"
       )                        (. Display(token); .)
     } EOF                      (. Display(token); .)
     .

END TokenTests.
```

10.8 Cocol Pragmas

A Cocol grammar can include pragmas - generally placed right near the start. These start with a $, followed by a few key letters, for example $CNF. Letters of interest are

- C - generate a driver program as well as the scanner and parser;
- N - generate names for the tokens (like `pointSym` or `commaSym`) rather than cryptic numbers;
- F - list the FIRST and FOLLOW sets for each non-terminal (in the file `listing.txt`);
- T - test the grammar but do not generate any scanner, parser or driver.

11 COCO/R: A SIMPLE ASSEMBLER FOR THE PVM

This chapter aims to show how we might use Coco/R to construct an assembler for programs written in PVM ASSEMBLER, as introduced in Chapter 4 - more sophisticated than the assembler given in that chapter.

11.1 Case study - A simple assembler for the PVM

As was pointed out in earlier chapters, assembler languages are usually quite simple, and the essence of an assembler is simply to map the set of mnemonics onto the corresponding opcodes. For a stack machine like the PVM many of the opcodes are of the zero-address form, so that assembly is almost trivial, especially if we can assume that the assembled code can be built up and stored in an array that corresponds to the memory of the interpreter that will later execute it.

From the human viewpoint the most awkward part of coding in a very low-level language is concerned with branching instructions. At the machine level these take a machine address as a parameter, but computing these addresses manually involves tedious and error-prone counting exercises, all of which have to be redone whenever the source program is altered. All realistic assemblers will offer some kind of labelling facility, so that rather than writing code of the form shown in Figure 11.3(a), one can write equivalent code of the form shown in Figure 11.3(b), and leave the assembly process to keep track of the values (memory addresses) associated with labels like `WHILE:` and `EXIT:`

```
LDA    0                  WHILE:  LDA   0
LDV                               LDV
LDC    0                          LDC   0
CGT                               CGT
BZE    18                         BZE   EXIT
LDA    0                          LDA   0
INPI                              INPI
BRN    5                          BRN   WHILE
HALT                      EXIT:   HALT
```

(a) Absolute addressing (b) Use of labels

Figure 11.3 Examples of stack machine code

To implement this we shall need a table structure in which we can record the names of the labels and the corresponding addresses. Unfortunately, constructing this table is complicated by the inevitable presence of forward references. A label like `WHILE` in the code in Figure 11.3 is easily handled, because its defining occurrence in the first line precedes its application in the penultimate instruction. By the time the assembler comes to handle the `BRN WHILE` instruction, all the necessary information is to hand and can be retrieved from the table. By contrast, when the assembler tries to handle the `BZE EXIT` instruction it has not yet come across the definition of the `EXIT` label. And, indeed, in a large program there might be numerous forward references to some labels before they are eventually defined, which will complicate matters still further.

There are several ways in which the problem can be solved. To illustrate one possibility, let us introduce the idea of building up a searchable table whose entries are defined by

```
class Entry {
  public string name;
  public Label label;
  public Entry(string name, Label label) {
    this.name = name; this.label = label;
  }
} // class Entry
```

The specification of a table handler is conceptually simple.

```
class Table {

  public static void Insert(Entry labelEntry)
  // Inserts labelEntry into label table

  public static Entry Find(string name)
  // Searches table for entry matching name.
  // If found then returns entry, if not found, returns null
} // class Table
```

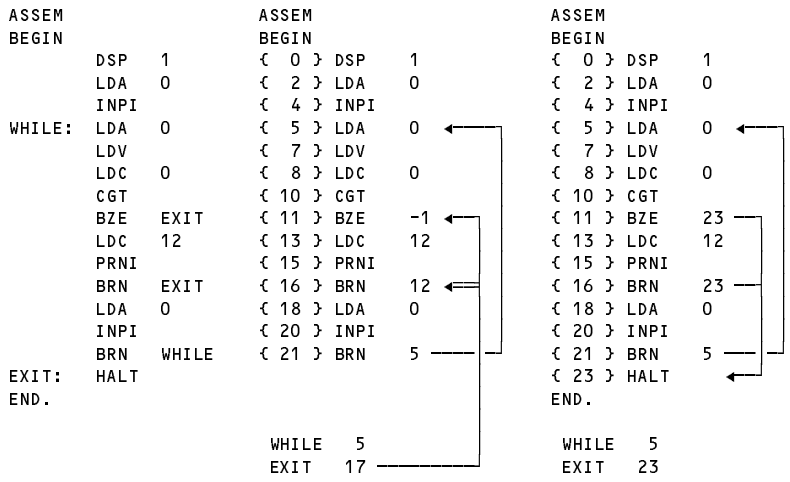
The concept of a *Label* is modelled by introducing another class

```
class Label {
  private int memAdr;      // address if this.defined,
                           // else last forward reference
  private boolean defined; // true once this.memAdr is known

  public Label(bool known)
  // Constructor for label, possibly at already known location

  public int Address()
  // Returns memAdr if known, otherwise effectively adds to a
  // forward reference chain that will be resolved if and when
  // Here() is called and in that case returns the address of the
  // most recent forward reference

  public void Here()
  // Defines memAdr of this label to be at the current location
  // counter after fixing any outstanding forward references

  public bool IsDefined()
  // Returns true if location of this label has been established
} // class Label
```

When an *applied reference* is made to a named label as the target of a BRN or BZE instruction, a search is made of the label table, using the name as a key. If the label has already found its way into the table, the corresponding entry can be retrieved. If not, a new instance of an "unknown" *Label* can be created and inserted into the table. In either case we now have a label to pass to the code generator, which can insert the value obtained from invoking the Address() method into the code array.

When a *defining occurrence* of a label is encountered, a search is made of the label table using the name as a key. If the label has not already found its way into the table, a "known" *Label* can be created and inserted into the table. If it is already in the table and has a known location, an erroneous attempt to redefine a label can be flagged. If it is already in the table but still has an "unknown" location, the table entry can be updated by invoking Here().

The implementation of the Address() and Here() methods achieves more than simply retrieving or setting the values of private fields. When an unknown label is created, it is assigned a recognizable (but impossible) value for memAdr and the defined field is set to false. Whenever Address() is invoked on a label object, it returns the current value of memAdr. However, if defined is false it also resets memAdr to point to the location in the code array corresponding to the address field of a BRN or BZE instruction that is, in fact, not yet known. This has the effect of building a linked chain of forward references within the code array. Until it is finally defined, the memAdr field of a label always points to the most recent location abused in this way.



(a) Source Code; (b) As assembled before EXIT:; (c) Fully assembled.

Figure 11.4 Stages in the assembly of a simple Parva ASSEMBLER program

An example may make this clearer. Figure 11.4(a) shows a section of code in which two forward references are made to the label EXIT. Immediately before assembly of the line labelled EXIT begins, the "value" of this label is 17 - pointing to the address field of the most recent BRN instruction aimed at EXIT. In this location has temporarily been stored the value 12 - pointing to the address field of the earlier BZE instruction that was also aimed at EXIT

(see Figure 11.4(b)). Similarly, the marker value -1 has temporarily been stored in this location.

When the label `EXIT` is encountered and `Here()` is invoked, it calls a simple routine that follows through this list changing the address fields to the now known value of 23, leaves `memAdr` set to this value and, at last, sets `defined` to `true`. This results in the final situation depicted in Figure 11.4(c).

The implementation of the *Label* class is concisely achieved by the following code.

```
class Label {
  private int memAdr;      // address if this.defined,
                           // else last forward reference
  private bool defined;    // true once this.memAdr is known

  public Label(bool known) {
    if (known) this.memAdr = CodeGen.GetCodeLength();
    else // mark end of forward reference chain
      this.memAdr = CodeGen.undefined;
    this.defined = known;
  } // constructor

  public int Address() {
    int adr = memAdr;
    if (!defined) memAdr = CodeGen.GetCodeLength();
    return adr;
  } // Address

  public void Here() {
    CodeGen.BackPatch(memAdr);
    memAdr = CodeGen.GetCodeLength();
    defined = true;
  } // Here

  public bool IsDefined() {
    return defined;
  } // IsDefined
} // class Label
```

Full details of the very simple code generator appear in the source code in the *Resource Kit*. It will suffice to list the `Branch` method to show how an address for a label is retrieved before being inserted into the code, and the `BackPatch` method that traces through the list of forward references.

```
public static void Branch(string mnemonic, Label lab) {
// Inline assembly of two word branch style instruction to adr
  Emit(PVM.OpCode(mnemonic)); Emit(lab.Address());
} // Branch

public static void BackPatch(int adr) {
// Stores the current location counter as the address field of
// the branch or call instruction currently holding a forward
// reference to adr and repeatedly works through a linked list
// of such instructions making the same adjustment
  while (adr != undefined) {
    int nextAdr = PVM.mem[adr];
    PVM.mem[adr] = codeTop;
    adr = nextAdr;
  }
} // BackPatch
```

The complete attributed grammar follows. Note the use of `IGNORECASE` (for key words) and `ToLower()` (for labels), and the use of the `Substring` method for extracting the portion of a string between its surroundin quotes, and for discarding the colon at the end of a defining occurrence of a label token.

```
COMPILER Parva $NC
/* Simple assembler for the PVM - C# version */

const bool known = true;

IGNORECASE

CHARACTERS
  lf       = CHR(10) .
  control  = CHR(0) .. CHR(31) .
  letter   = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit    = "0123456789" .
  stringCh = ANY - '"' - control .
```

```
TOKENS
  identifier = letter { letter | digit } .
  number     = digit { digit } .
  label      = letter { letter | digit } ":" .
  stringLit  = '"' { stringCh } '"' .

COMMENTS FROM "{" TO "}"
COMMENTS FROM ";" TO lf

IGNORE CHR(9) .. CHR(13)

PRODUCTIONS
  Parva
  = "ASSEM" "BEGIN"
       { Statement }
    "END" "."                  (. Table.CheckLabels(); .) .

  Statement
  = OneWord | TwoWord | WriteString | Label | Branch .

  OneWord
  = (   "ADD"  | "AND"  | "ANEW" | "CEQ"  | "CGE"  | "CGT"
      | "CLE"  | "CLT"  | "CNE"  | "DIV"  | "HALT" | "INPB"
      | "INPI" | "LDV"  | "LDXA" | "MUL"  | "NEG"  | "NOP"
      | "NOT"  | "OR"   | "PRNB" | "PRNI" | "PRNL" | "REM"
      | "STO"  | "SUB" )       (. CodeGen.OneWord(token.val); .) .

  TwoWord                      (. int value; .)
  = ( "DSP" | "LDC" | "LDA" )  (. string mnemonic = token.val; .)
    SignedNumber<out value>    (. CodeGen.TwoWord(mnemonic, value); .) .

  SignedNumber<out int value>  (. int sign = 1; .)
  = [   "+"
      | "-"                    (. sign = -1; .)
    ] IntConst<out value>      (. value = sign * value; .) .

  IntConst<out int value>
  = number                     (. try {
                                    value = Convert.ToInt32(token.val);
                                  } catch (Exception) {
                                    value = 0;
                                    SemError("number too large");
                                  } .) .

  WriteString                  (. string str; .)
  = "PRNS"
    StringConst<out str>       (. CodeGen.WriteString(str); .) .

  StringConst<out string str>
  = stringLit                  (. str = token.val.Substring(1, token.val.Length-2); .) .

  Label
  = label                      (. string name = token.val.Substring(0, token.val.Length-1).ToLower();
                                  Entry tableEntry = Table.Find(name);
                                  if (tableEntry == null)
                                    Table.Insert(new Entry(name, new Label(known)));
                                  else if (tableEntry.label.IsDefined())
                                    SemError("redefined label");
                                  else tableEntry.label.Here(); .) .

  Branch                       (. int target;
                                  string name;
                                  Label lab; .)
  = ( "BRN" | "BZE" )          (. string mnemonic = token.val; .)
    (   IntConst<out target>   (. CodeGen.TwoWord(mnemonic, target); .)
      | Ident<out name>        (. Entry tableEntry = Table.Find(name);
                                  if (tableEntry == null) {
                                    lab = new Label(!known);
                                    Table.Insert(new Entry(name, lab));
                                  }
                                  else lab = tableEntry.label;
                                  CodeGen.Branch(mnemonic, lab); .)
    ) .

  Ident<out string name>
  = identifier                 (. name = token.val.ToLower(); .) .

END Parva.
```

12 A PARVA COMPILER: THE FRONT END

The remainder of this book is devoted to the construction of compilers for simple programming languages. The next three chapters explore a compiler for Parva, a simple C-like programming language, developed (and later extended) from the grammar introduced in section 7.5. This compiler targets the PVM discussed in Chapter 4.

In a text of this nature it is impossible to discuss full-blown compilers for large languages, and the value of our treatment may arguably be reduced by the fact that in dealing with toy languages and toy compilers we shall be evading some of the real issues that a compiler writer has to face. However, we hope the reader will find the ensuing discussion of interest and that it will serve as a useful preparation for the study of much larger compilers. The technique we shall follow is one of slow refinement, supplementing the discussion with various asides on the issues that would be raised in compiling larger languages. Clearly, we could follow the example of several authors and develop a completely handcrafted compiler, but we prefer to illustrate the use of Coco/R, which removes from the compiler writer the tedium of developing the scanner and parser, and will allow us to concentrate on the more interesting aspects of constraint analysis (in this chapter) and code generation (in the next one).

12.1 Overall compiler structure

In Chapter 2 we commented that a compiler is often developed as a sequence of phases, of which syntactic analysis is only one. Although a recursive descent parser is easily written by applying the ideas of earlier chapters or using a tool like Coco/R, it should be clear that consideration will have to be given to the relationship of this to the other phases. It may be helpful to think of a compiler with a recursive descent parser at its core as having the structure depicted in Figure 12.1.

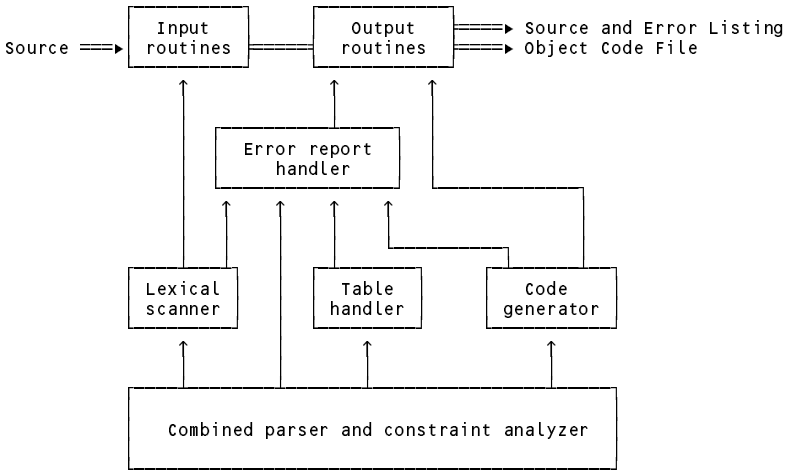


Figure 12.1 Relationship between the main components of a simple compiler

The C# and Java versions of Coco/R generate classes for a scanner, a parser, an error report handler, an input buffer manager and a driver that closely match Figure 12.1. In other situations a compiler might not look exactly like this. In some versions of Coco/R the scanner and the source handling section are combined into one module and the routines for producing the error listing are integrated with the main driver module. The C++ version of Coco/R made use of a standard class hierarchy involving parser, scanner and error reporter classes, and established links between the various instances of these classes as they were constructed. This gives the flexibility of having multiple instances of parsers or scanners within one system, although this is probably rarely exploited.

We emphasize that phases need not be sequential, as passes would be. In a recursive descent compiler it is common to find that the phases of syntax analysis, semantic analysis and code generation are interleaved. Nevertheless, it is useful to think of developing modular components to handle the various phases, with clear simple interfaces between them. If the source language - like Parva - is suitably restricted, the onset of the code generation phase does not have to wait until the phases of syntax and constraint analysis have been completed, but can begin almost immediately. Following Gough (2002) we shall refer to a compiler that employs this technique as an *incremental one-pass compiler*, although we should mention that the term "incremental compiling" is used differently by some other authors.

12.2 File handling

Reading the source file and writing the object code may, in some cases, take an appreciable amount of the total time to compile a program - it helps to make these parts of the system as efficient as possible.

Among the file handling routines of a compiler is to be found one that has the task of transmitting the source, character by character, to the scanner or lexical analyzer (which assembles it into symbols for subsequent parsing by the syntax analyzer). Since there are far more characters than symbols, one should try to touch each character as few times as possible - suggesting that the lookahead character should be stored in a static field of the scanner, rather than be repeatedly passed around as a parameter. Ideally the source handler should have to scan the program text only once, from start to finish, and in a one-pass compiler this should always be possible.

A source handler can improve on efficiency rather dramatically if it is able to read the entire source file into a large memory buffer. Since modern systems are often blessed with relatively huge amounts of RAM, this is usually quite feasible and the standard frame files supplied with Coco/R adopt this approach. The use of a buffer also allows for part or all of the source text to be rescanned easily when necessary, and can be useful in the production of compiler listings in which the source is merged with error messages in a convenient manner. However, the use of such a buffer precludes the easy development of interactive terminal-driven applications.

The source file has to be identified and opened before the buffer can be filled and the scanner initialized. In a Coco/R generated compiler this is handled automatically by the generated driver class.

12.3 Error reporting

As can be seen from Figure 12.1, most components of a compiler have to be prepared to signal that something has gone awry in the compilation process. To allow all of this to take place in a uniform way, Coco/R defines a base class error handler with a very small interface for "storing" syntactic and semantic errors, which a user might choose to extend in various ways. As part of the parser generation process, this error handler is extended automatically to handle those syntactic error messages that Coco/R can derive from the grammar definition - the default storage mechanism is simply to reflect these messages on the standard output file with an indication of the source line and column where they were detected. This simple mechanism can be used to good effect in conjunction with sophisticated editors that can use this output to track the position of each error in the source file. The parser frame files supplied with the distribution of Coco/R in the *Resource Kit* also provide an alternative option of storing the errors in a list which is later merged into a text file, along with the source, for the convenience of a user who does not have access to such editors.

A parser might choose simply to abort compilation altogether as soon as any error is detected. Although at least one highly successful microcomputer Pascal compiler used this strategy (Turbo Pascal, from Borland International), it tends to become very annoying when one is developing large systems.

12.4 Scanning and parsing

Lexical analysis was discussed in section 8.5 and presents few problems for languages as simple as Parva or Mikra. If required, a scanner for either one could be readily programmed in an *ad hoc* manner, driven by a selection statement. Similarly, a recursive descent parser could be readily programmed for either language following the ideas developed in section 8.1, since it may be easily verified that the basic grammars are LL(1). Adding syntax error recovery would be a little tedious and, by the time one had added the constraint analysis that is the main thrust of this chapter, the code would start to become quite cluttered.

A scanner and parser for Parva can be generated immediately from the Cocol grammar presented in section 7.5.1. That grammar made no attempt to incorporate syntax error recovery, constraint analysis or code generation - topics to which we must now turn.

12.5 Syntax error recovery

The way in which a Cocol description of a grammar is augmented to indicate where synchronization should be attempted was discussed in section 10.5.2. To be able to achieve optimal use of the basic facilities offered by the use of the SYNC and WEAK directives calls for some ingenuity. If too many SYNC directives are introduced, the error

recovery achievable with the use of WEAK can actually deteriorate, since the union of all the SYNC symbol sets tends to become the entire universe. Below we show a modification of part of the grammar in section 7.5.2 that has been found to work quite well.

```
PRODUCTIONS /* some omitted to save space */
  Parva             = "void" identifier "(" ")" Block .
  Block             = WEAK "{" { Statement } WEAK "}" .
  Statement         = SYNC (  Block | ConstDeclarations
                            | VarDeclarations | Assignment
                            | IfStatement | WhileStatement
                            | ReturnStatement | HaltStatement
                            | ReadStatement | WriteStatement
                            | ";" ) .
  ConstDeclarations = "const" OneConst
                        { WEAK "," OneConst } WEAK ";" .
  VarDeclarations   = Type OneVar { WEAK "," OneVar } WEAK ";" .
  Assignment        = Designator "=" Expression WEAK ";" .
  ReturnStatement   = "return" WEAK ";" .
  HaltStatement     = "halt" WEAK ";" .
  ReadStatement     = "read"  "(" ReadElement
                        { WEAK "," ReadElement  } ")" WEAK ";" .
  WriteStatement    = "write" "(" WriteElement
                       { WEAK "," WriteElement } ")" WEAK ";" .
```

In essence, synchronization has been attempted at points where a statement should begin, and the commas and semicolons that act as separators in lists or terminators of statements are all marked WEAK. This might have been overdone - the reader is invited to explore other variations on this theme. Sadly, no amount of effort dedicated to error recovery can make any sense of source programs that are grotesquely incorrect.

12.6 Constraint analysis

We have remarked on several occasions that the boundary between syntactic and semantic errors can be rather vague, and that some features of real computer languages are not readily described by context-free grammars.

12.6.1 The need for a symbol table

Once we start to include semantic analysis, and to attach meaning to our identifiers, the advantages of simple one-pass compilation are most easily gained by demanding that the "declaration" parts of a program come before the "statement" parts. This is easily enforced by a context-free grammar, all very familiar to most programmers, and even seems quite natural after a while. But it is only part of the story. Even if we insist that declarations precede statements, a context-free grammar is still unable to specify that those identifiers which have been declared may be used only in other statements which are within the scope of those declarations. Nor is a context-free grammar powerful enough to specify such simple constraints as insisting that only a variable identifier can be used to denote the target of an assignment statement, or that an integer constant cannot be assigned to a Boolean variable. We might be tempted to write productions that seem to capture some of these ideas.

```
OneConst   = constIdentifier "=" Constant .
Constant   = number | charLit | "true" | "false" | "null" .
OneVar     = varIdentifier [ "=" Expression ] .
Assignment = Variable "=" Expression ";" .
Variable   = varIdentifier [ "[" Expression "]" ] .
Expression = AddExp [ RelOp AddExp ] .
Factor     =   Variable | constIdentifier | Constant
             | "new" BasicType "[" Expression "]" .
             | "!" Factor | "(" Expression ")" .
```

While this might be suggestive to a human reader, it will not really get us very far, since all identifiers are lexically equivalent. We might be tempted to find a context-sensitive grammar to overcome such problems, but that turns out to be unnecessarily complicated, for the problems are easily solved by leaving the grammar as it was, adding attributes in the form of context conditions, and using a **symbol table**.

Demanding that identifiers be declared in such a way that their names and static semantic attributes can be

recorded in a table, whence they can be retrieved at a future stage of the analysis, is not nearly as tedious as users might at first imagine. It is clearly a semantic activity, made easier by a syntactic association with keywords like `const`, `int` and `bool`.

Setting up a symbol table can be done in many ways, but we need to take note of three important properties of all identifiers in the sorts of languages we are considering.

12.6.2 Identifiers have scope

Languages like Pascal, C#, Java - and even Parva - are said to be **block-structured**. In Pascal and Modula-2 the basic block structure applies to a complete module, procedure or function unit only. In Parva, as in C# and Java, the block structure extends down to a statement level, as is reflected in the concrete syntax given earlier. A *Block* allows for the intermingling of statements that handle the declaration of constants and variables, as well as statements that manipulate them. Since statements such as *IfStatements* and *WhileStatements* can incorporate further blocks as associated statements, blocks can be nested, in principle, to any depth. This introduces the concept of **scope**, which should be familiar to readers experienced in developing code in block-structured languages, although it causes confusion to some beginners. In such languages, the "visibility" or "accessibility" of an identifier declared within a *Block* is limited to that block, and to blocks themselves nested within that block. Some rule has to be applied when an identifier declared in one block is *redeclared* in one or more nested blocks. This rule differs from language to language, but in many cases such redeclaration is allowed, on the understanding that it is the innermost accessible declaration that will apply to any particular use of that identifier.

Scope rules like these can be easily implemented in a number of ways, all of which rely on some sort of stack structure. The simplest approach is to build the entire symbol table as a stack of entries, pushing an entry onto this stack each time an identifier is declared, and popping several entries off again whenever we complete parsing a *Block*, thereby ensuring that the names that were declared local to that block disappear from the table and thus go out of scope. A stack structure also ensures that if two identifiers with the same names are declared in nested blocks, the first to be found when searching the table will be the most recently declared. The discussion may be clarified by considering the shell of a simple program.

```
void main () {          // outer block begins here
  int G1, G2 = 10;
  while (G2 > 0) {        // inner block begins here
   int L1, L2;
   ...                    //     point (a)
  }                       // inner block ends here
  int G3;               //        point (b)
  ...
}                       // outer block ends here
```

For this program, either of the approaches suggested by Figure 12.2 or Figure 12.3 might appear to be suitable for constructing a symbol table. It should be clear that the stack of identifier entries must be augmented in some way to keep track of the divisions between blocks, either by introducing a special category of entry, or by constructing an ancillary stack of special purpose nodes.

In these structures, an extra "sentinel" node has been inserted at the bottom of the stack. This allows a search of the table to be implemented very simply (if perhaps inefficiently), by inserting a copy of the identifier that is being sought into this node before the (linear) search begins. The figures depict the state of the table when the parser reaches the points marked (a) and (b) as it analyzes the above skeleton.

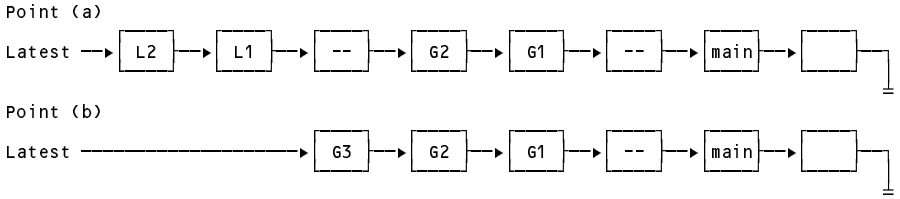


Figure 12.2 Stack-based symbol table with extra nodes marking scope boundaries

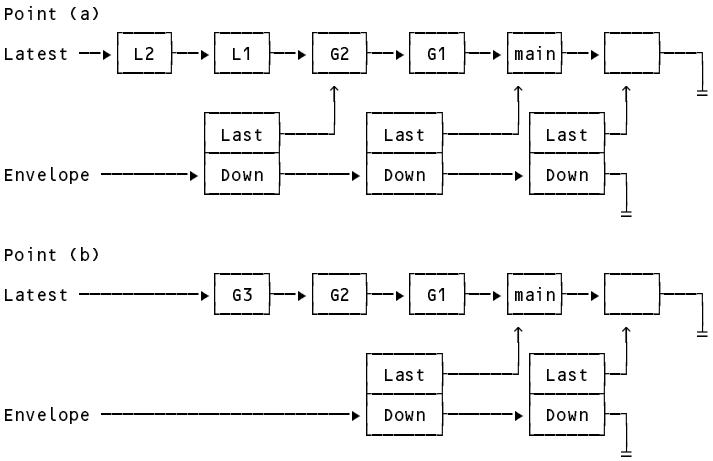


Figure 12.3 Stack-based symbol table with ancillary stack marking scope boundaries

As it happens, this sort of structure becomes rather more difficult to adapt when one extends the language to incorporate functions that must handle parameters, and so we shall promote the idea of having a stack of *scope nodes*, each of which contains a reference (pointer) to the scope node corresponding to an outer scope, as well as a reference (pointer) to a structure of *identifier nodes* pertinent to its own scope. This latter structure could be held as a stack, queue, tree or even hash table. Figure 12.4 shows the use of queues each of which is terminated by a common sentinel node - a structure that makes for easy search algorithms to be implemented.

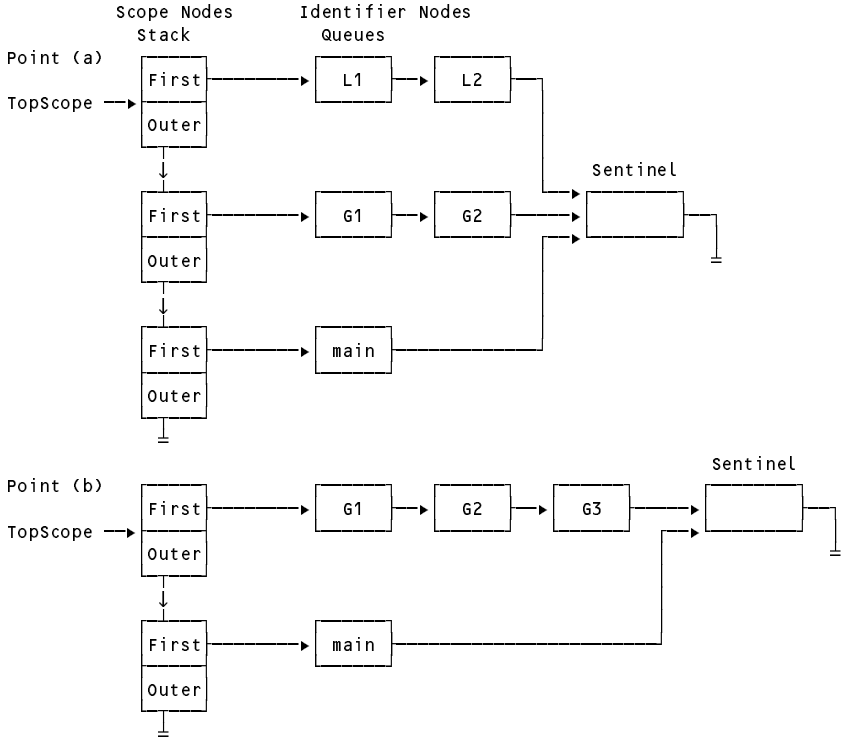


Figure 12.4 Symbol table based on a set of queues, with ancillary stack marking scope boundaries

Parva (like Java and C#) allows for statements and declarations to be intermingled. As we have already implied, single-pass incremental compilation will be difficult unless (as we shall do) we impose a "declare before use" rule. For a single function program this normally causes no real hardship, but if we were to allow an identifier declared in one block to be redeclared in a nested block we could get into trouble, so we shall follow the example of Java and forbid such redeclaration. Not only does this reduce the possibility of hard-to-find bugs if an identifier is redeclared by mistake rather than by design, but it also solves the problem of trying to explain the meaning of code like the following.

```
void main () {
  int i = 10;
  while (i > 0) {
    write(i);
    int i = i - 1;
  }
} // main
```

At the point where the `write` statement is encountered, the first `i` is in scope and code should presumably be generated to print the value of this variable. However, if the scope of an identifier is to extend over the whole of the block in which it is declared, one could argue that perhaps the second `i` should be the one that is in scope. However, in a one-pass compiler the second declaration would not have been encountered by the time that the code for the `write(i)` statement might have been generated. Formulating the scope rules so that the scope of an identifier extends from the point of declaration to the end of the block in which it is declared makes for simple one-pass compilation, but this example would surely be confusing to a programmer who expected some other interpretation. Furthermore, a combined declaration and assignment like `int i = i - 1;` seems to imply that one should be able to generate code to decrement the value of a variable before it had been assigned an initial value. In passing, we may observe that a certain breed of "computer language lawyer" often delights in inventing examples like the above, which serve little purpose other than to point out ambiguities or inconsistencies in the semantic aspects of language specification!

12.6.3 Identifiers come in specific kinds

Even for a language as simple as Parva, we note that identifiers designate entities that are restricted to one of three simple kinds - namely *constants*, *variables* and *functions*. In larger languages there might be other kinds of identifiers, for example denoting *classes* or other user-named *types*.

12.6.4 Identifiers are associated with specific types

Modern imperative languages allow users a great deal of flexibility and versatility in defining (and naming) their own types and manipulating entities and objects of these types - all of which are ultimately defined in terms of fundamental machine-level types, of course. A full discussion of the concept of **type** is inappropriate at this stage. It will suffice to note that even in our simple language we must distinguish between variables that are uniquely associated with scalar values of integer or Boolean type, and make provision for vectors or arrays whose elements are of these types, by allowing for variables whose values are associated with references (pointers) to those arrays. Later we shall see that it is expedient to introduce the concept of a pseudo-type `void` that can be associated with functions that do not return a computed result (these would be thought of as *regular procedures* by Pascal or Modula-2 programmers). It is also useful to introduce two other pseudo-types - one for entities that were never properly declared and one for the type of the generic `null` reference constant that is compatible with (and assignable to) a variable of any array reference type.

The reader should note that:

- Parva is a language simple enough for the distinction between types to be drawn by using a simple integer enumeration - this is rather unrealistic and more complex languages demand that types be represented by more complex objects;

- the concept of enumerating the different *kinds* of identifier must not be confused with the concept of enumerating the different *types* associated with identifiers;

- it is not only the identifiers in a program that have a type attribute - expressions must also be regarded as yielding values of a type which depends on the types of the operands and the effects of the operators of which they are composed. (We shall return to this important topic in section 12.6.8.)

12.6.5 Classes for supporting a symbol table and static semantic analysis

We are now in a position to suggest the form that classes suitable for constructing a symbol table might assume, at least so far as their fields are concerned.

```
class Scope {
  public Scope outer;        // link to enclosing (outer) scope
  public Entry firstInScope; // link to first identifier entry
                             // in this scope
} // class Scope

class Kinds {
  public const int
    Con = 0,                 // identifier kinds
    Var = 1,
    Fun = 2;
} // class Kinds

class Types {
  public const int
    noType   = 0,            // identifier (and expression) types.
    nullType = 2,            // The numbering is significant as
    intType  = 4,            // corresponding array reference types
    boolType = 6,            // are denoted by these numbers + 1
    voidType = 8;
} // class Types

class Entry {
  public int    kind;        // Kinds
  public int    type;        // Types
  public string name;        // lexeme
  public Entry  nextInScope; // link to next entry in current scope
  public bool   declared;    // true for all except the sentinel entry
} // class Entry
```

The suggested specification of the symbol table handler is as follows.

```
class Table {

  public static void Insert(Entry entry)
  // Adds entry to symbol table

  public static Entry Find(string name)
  // Searches table for an entry matching name.  If found then
  // return that entry; otherwise return a sentinel entry
  // (marked as not declared)

  public static void OpenScope()
  // Opens a scope record at the start of parsing statement block

  public static void CloseScope()
  // Closes a scope record at the end of parsing statement block

  public static void PrintTable(OutFile lst)
  // Prints symbol table for diagnostic purposes

  public static void Init()
  // Clears table and sets up sentinel entry

} // class Table
```

Full details of the implementation (which uses standard algorithms for manipulating linked lists) can be found in the *Resource Kit*. The following points are worth noting.

- The `Find` operation begins its search in the top scope list, working its way down through the outer scopes if necessary. Each of these lists is terminated by the sentinel node. Before the search starts, the name being matched is copied to the name field of the sentinel node to simplify the search.

- The implementation is such that a `Find` operation always succeeds to some extent - if a matching entry is not found by the time all scope lists have been exhausted then the sentinel entry is returned, masquerading as an entry for a variable of the pseudo-type `noType`.

- The sentinel entry is created when the table is initialized, and its `declared` field is set to `false`; all other entries will have their `declared` fields set to `true`. This affords an effective (if somewhat inelegant) way for the caller of the `Find` operation to detect that the search has really failed.

- The `Insert` operation checks for an extant entry with the same name before adding the entry to the list for the top scope. Parva, like Java, objects to redeclaration of an identifier that is already in scope.

- Although the use of unique integer constants to enumerate kinds and types is simple enough, it is error prone. At the time when this book and its Parva compilers were first developed, neither C# nor Java supported *enumeration type* like those found in Pascal and Modula-2. In the interests of continuity the treatment here has retained the use of integers, but it is a useful and informative exercise to modify the Parva system to use the enumerations that are now supported in Java and C# as well.

12.6.6 Identifier declarations

In terms of the symbol table mechanism that we have instituted, the semantic actions that we must add to our grammar to handle the declarations of identifiers now follow quite easily. The reader should note that we have introduced a *single production* for a new non-terminal *Ident* that takes the place of the *identifier* token in the grammar as previously defined - this isolates the operation of obtaining the lexeme for an identifier. Similar single productions for non-terminals *IntConst, StringConst* and *CharConst* aid in the isolation of the *number, stringLit* and *charLit* tokens of the original grammar.

```
Block
=                   (. Table.OpenScope(); .)
   WEAK "{" { Statement }
   WEAK "}"         (. Table.CloseScope(); .) .

OneConst            (. Entry constant = new Entry(); .)
=  Ident<out constant.name>
                    (. constant.kind = Kinds.Con; .)
   AssignOp
   Constant<out constant.type>
                    (. Table.Insert(constant); .) .

Constant<out int type>
                    (. type = Types.noType; .)
=    ( IntConst | CharConst )
                    (. type = Types.intType; .)
   | ( "true" | "false" )
                    (. type = Types.boolType; .)
   | "null"         (. type = Types.nullType; .) .

VarDeclarations     (. int type; .)
=  Type<out type>
   OneVar<type>
   { WEAK "," OneVar<type> }
   WEAK ";" .

Type<out int type>
=  BasicType<out type>
   [ "[]"           (. if (type != Types.noType) type++; .)
   ] .

BasicType<out int type>
                    (. type = Types.noType; .)
 =   "int"          (. type = Types.intType; .)
   | "bool"         (. type = Types.boolType; .) .

OneVar<int type>    (. int expType; .)
=                   (. Entry var = new Entry(); .)
   Ident<out var.name>
                    (. var.kind = Kinds.Var;
                       var.type = type; .)
   [ AssignOp Expression<out expType>
                    (. if (!Compatible(var.type, expType))
                         SemError("incompatible types"); .)
   ]                (. Table.Insert(var); .) .

Ident<out string name>
=  identifier       (. name = token.val; .) .
```

12.6.7 Identifier applications

We turn now to a consideration of those other parts of the grammar for Parva in which identifiers play a role. Our grammar has been written to use the concept of a *Designator* to cover all of these situations, as a look at the relevant productions will show.

```
Assignment  = Designator "=" Expression ";" .
ReadElement = StringConst | Designator .
Factor      =   Designator | Constant
              | "new" BasicType "[" Expression "]"
              | "!" Factor | "(" Expression ")" .
Designator  = Ident [ "[" Expression "]" ] .
Constant    = IntConst | CharConst | "true" | "false" | "null" .
Ident       = identifier .
```

In its simplest form a *Designator* is simply an identifier, which must already have been made known to the parser and stored in the symbol table at the time it was declared. As the entries in the symbol table have kinds and types, so too we can attribute a *Designator* with these properties, and a little reflection will show that only particular varieties of *Designator* will be acceptable at the points where they are found in the context-free productions just quoted. For example, an *Assignment* can have meaning only if its *Designator* is associated with a variable (not with a constant) and this applies also to the *Designator* of a *ReadElement*, which is still further restricted to being associated with a simple type such as `int` and not with a reference type such as `int[]`. The *Designator* in a *Factor* can, however, be associated with either a constant or a variable. Finally, although the context-free production for a *Designator* allows for an optional bracketed *Expression* to follow the identifier, this device for selecting an array element has semantic meaning only if the associated identifier denotes an entity declared to be a variable of an array type. Some of these semantic constraints are easily handled by actions defined in the parser for a *Designator* - some are more easily checked by the parsing methods that call on *Designator*. Accordingly, we arrange for the designator parser to synthesize an object of a simple *DesType* class, which these methods can examine before accepting or rejecting a *Designator*.

The *DesType* class is defined by

```
class DesType {
// Objects of this type are associated with l-value and
// r-value designators
  public Entry entry;          // the identifier properties
  public int type;             // designator type
                               // (not always the entry type)
  public DesType(Entry entry) {
    this.entry = entry; this.type = entry.type;
  }
} // class DesType
```

and the attributed production for a *Designator* becomes

```
Designator<out DesType des>
                    (. string name;
                       int indexType; .)
=  Ident<out name>  (. Entry entry = Table.Find(name);
                       if (!entry.declared)
                         SemError("undeclared identifier");
                       des = new DesType(entry); .)
   [  "["           (. if (IsRef(des.type)) des.type--;
                       else SemError("unexpected subscript");
                       if (entry.kind != Kinds.Var)
                         SemError("unexpected subscript"); .)
         Expression<out indexType>
                    (. if (!IsArith(indexType))
                         SemError("invalid subscript type"); .)
      "]"
   ] .
```

where we draw attention to the fact that the type of an array element designated by a construction like `Array[X]` is not the type of `X` nor the (reference) type of `Array`, but a type that can be deduced easily from the type of `Array` using the carefully chosen enumeration of the acceptable types. The way in which this synthesized *DesType* object can assist in semantic checking is demonstrated in the attributed productions for *Assignment* and *ReadElement*.

```
Assignment          (. int expType;
                       DesType des; .)
=  Designator<out des>
                    (. if (des.entry.kind != Kinds.Var)
                         SemError("invalid assignment"); .)
   AssignOp
   Expression<out expType>
```

```
                    (. if (!Compatible(des.type, expType))
                         SemError("incompatible types"); .)
    WEAK ";" .

ReadElement         (. DesType des; .)
=   StringConst
  | Designator<out des>
                    (. if (des.entry.kind != Kinds.Var)
                         SemError("wrong kind of identifier");
                       switch (des.type) {
                         case Types.intType:
                         case Types.boolType:
                           break;
                         default:
                           SemError("cannot read this type");
                           break;
                       } .) .
```

12.6.8 Type checking

As already mentioned, in strongly-typed languages like Pascal, C#, Java and Parva we must recognize that not only do the operands in an *Expression* possess a distinctive type, but also that an expression as a whole is really a prescription for computing a value that has an associated type which can be inferred from the types of its operands and the kinds of operators used in its construction. Not all combinations of operands and operators lead to valid types, nor are all types of expression valid in all contexts. Obvious examples here are that the *Expression* that denotes the *Condition* required by the context-free syntax for an *IfStatement* or *WhileStatement* must be of Boolean type, while the *Expression* that is used as a selecting subscript in a *Designator*, or to specify the size of an array when one is created, must be of integer type.

The productions of those parts of our grammar that deal with the specification of an *Expression* must be attributed in such a way that they synthesize an output attribute that denotes the type of that expression. In Parva, as we have seen, a simple integer enumeration suffices to distinguish types. Those productions that make use of an *Expression* can then be attributed to perform an appropriate check that the synthesized type is acceptable in their context. We must be prepared to deal with situations where expressions are incorrectly coded. This affords another reason why it is convenient to add to the enumeration a value (`noType`) that can denote that an expression appears to be of indeterminate type, perhaps because one of its operands is an undeclared identifier.

Operands in expressions are combined by the action of operators. To check on the semantics of these operations it is necessary to introduce the concept of **type compatibility** - an operation will only be acceptable if its operand or operands are selected from a small set of types for which the operation can be performed, and for which it can yield a result. The acceptable operand/operator combinations for Parva are tabulated in Figure 12.5, where we note that the result of an operation may be of a different type from its operand or operands and where, for convenience, we have included the assignment operator as well.

Operator OP (infix)	Type of A	Type of B	Type of A OP B
+ - * / %	int	int	int
&& ||	bool	bool	bool
< <= > >=	int	int	bool
== !=	int bool int[] or null bool[] or null	int bool int[] or null bool[] or null	bool bool bool bool
Operator OP (unary)	**Type of A**		**Type of OP A**
+ - (unary)	int		int
! (unary)	bool		bool
= (assign)	int bool int[] bool[]	int bool int[] or null bool[] or null	

Figure 12.5 Acceptable operand/operator combinations in Parva

Any attempt to combine operands and operators that does not match those allowed by Figure 12.5 should be flagged as an error. However, it is expedient to relax this slightly and not to complain if an operand is of the `noType` pseudo-type. Operands of this type occur as a result of earlier errors such as mismatching types or using undeclared identifiers, and these transgressions will already have been detected in the actions associated with the production for *Designator*.

We are now in a position to understand the fully attributed productions for *WriteElement, Condition, Expression, AddExp, Term* and *Factor*. The complete set can be studied in the source code in the *Resource Kit* and it will suffice here to show only three extracts - the productions for *Factor* (which is where the operands of an expression ultimately are recognized) and for *Expression* and *Term*.

```
Expression<out int type>
                    (. int type2;
                       int op;
                       bool comparable; .)
=  AddExp<out type>
   [ RelOp<out op> AddExp<out type2>
                    (. switch (op) {
                         case eql:
                         case neq:
                           comparable = Compatible(type, type2);
                           break;
                         default:
                           comparable = IsArith(type)
                                        && IsArith(type2);
                           break;
                       }
                       if (!comparable)
                         SemError("incomparable operands");
                       type = Types.boolType; .)
   ] .

Term<out int type>  (. int type2;
                       int op; .)
=  Factor<out type>
   { MulOp<out op> Factor<out type2>
                    (. switch (op) {
                         case and:
                           if (!IsBool(type) || !IsBool(type2))
                             SemError("bool operands needed");
                           type = Types.boolType;
                           break;
                         default:
                           if (!IsArith(type) || !IsArith(type2)) {
                             SemError("arith operands needed");
                             type = Types.noType;
                           }
                           break;
                       } .)
   } .

Factor<out int type>
                    (. type = Types.noType;
                       int size;
                       DesType des; .)
=    Designator<out des>
                    (. type = des.type;
                       switch (des.entry.kind) {
                         case Kinds.Var:
                         case Kinds.Con:
                           break;
                         default:
                           SemError("wrong kind of identifier");
                           break;
                       } .)
   | Constant<out type>
   | "new" BasicType<out type>
                    (. type++; .)
     "[" Expression<out size>
                    (. if (!IsArith(size))
                         SemError("size must be integer"); .)
     "]"
   | "!" Factor<out type>
                    (. if (!IsBool(type))
                         SemError("bool operand needed");
                       type = Types.boolType; .)
   | "(" Expression<out type> ")" .
```

The semantic actions here call upon further simple methods declared within the parser that check the (somewhat relaxed) attributes of operands needed for rigorous type checking, while at the same time avoiding a plethora of unhelpful messages related to expressions of indeterminate type.

```
static bool IsArith(int type) {
  return type == Types.intType || type == Types.noType;
} // IsArith

static bool IsBool(int type) {
  return type == Types.boolType || type == Types.noType;
} // IsBool

static bool IsRef(int type) {
  return (type % 2) == 1;
} // IsRef

static bool Compatible(int typeOne, int typeTwo) {
// Returns true if typeOne is compatible with typeTwo
  return    typeOne == typeTwo
         || typeOne == Types.noType || typeTwo == Types.noType
         || IsRef(typeOne) && typeTwo == Types.nullType
         || IsRef(typeTwo) && typeOne == Types.nullType;
} // IsCompatible
```

As an example of how these ideas combine in the reporting of incorrect programs we present a source listing produced by a parser derived from our attributed grammar.

```
   1 void main () {
   2   const max = 111000000000000000 +
****               ↑ number too large
   3   int x, y = true;
****              ↑ incompatible types in assignment
   4   bool b = main;
****            ↑ wrong kind of identifier
   5   read("supply x, x);
****         ↑ invalid ReadElement
   6   while (x) {
****           ↑ bool expression needed
   7     int y, z;
****          ↑ earlier declaration of y still in scope
   8     y = true || 4;
****                 ↑ bool operands needed
   9     write(x = a);
****              ↑ == intended?
  10   }
  11   y = z;
****        ↑ undeclared identifier
  12   write(max, "badstring);
****              ↑ invalid WriteElement
  13 } // main
  14
**** ↑ ) expected
```

13 A PARVA COMPILER: THE BACK END

After the front end has analyzed the source code, the back end of a compiler is responsible for synthesizing object code. The critical reader will have realized that code generation, of any form, implies that we consider the semantics of our language and of our target machine, and the interaction between them, in a different light and in more detail than we have done up to now. In this chapter we consider how we might enhance the parser developed in the last chapter to allow for generation of code (specifically for the PVM of Chapter 4, but our treatment will suggest how we might target other machines as well). In doing so we shall take advantage of three features that make this relatively simple. Firstly, the Parva language has been designed so that it is easily defined by an LL(1) grammar and, in particular, can be compiled using an single-pass compiler in which code generation can be carried out hand-in-hand with syntactic analysis. Secondly, the PVM has been designed to provide just the sorts of opcodes and features that the Parva language needs. Lastly, because the PVM is a virtual machine, any extensions that we might wish to make can, if necessary, be accomplished by extending either or both of the Parva language and the PVM to suit our fancy.

The reader might wonder whether taking such apparent liberties is realistic, but there is ample precedent for doing so. We draw attention to the discussion in section 2.8 where it was noted that the JVM was designed explicitly to support the features needed by Java. Similarly, the virtual stack machine postulated as the basis for the .NET CLR also has instructions for creating and manipulating objects and throwing exceptions that do not correspond directly to the traditional opcode set of most real machines. In later chapters of Terry (2005) are developed simple compilers for the JVM and CLR where, of course, we shall have to conform to the restrictions their designers have already imposed.

13.1 Extending the symbol table entries

Any interface between source and object code must take cognizance of data-related concepts like *storage*, *addresses* and *data representation*, as well as control-related ones like *location counter, sequential execution* and *branch instruction*, which are fundamental to nearly all machines on which programs in our imperative high-level languages execute. Typically, machines allow some operations which simulate arithmetic or logical operations on data bit patterns which simulate numbers or characters, these patterns being stored in an array-like structure of *memory*, whose elements are distinguished by *addresses*. In high-level languages these addresses are usually given mnemonic names. The context-free syntax of many high-level languages, as it happens, rarely seems to draw a distinction between the "address" for a variable and the "value" associated with that variable, and stored at its address. Hence we find statements like

```
x = x + 4;
```

in which the `x` on the left of the assignment operator actually represents an address (sometimes called the **L-value** of `x`), while the `x` on the right (sometimes called the **R-value** of `x`) actually represents the value of the quantity currently residing at the same address. Small wonder that mathematically trained beginners sometimes find the assignment notation strange! After a while it usually becomes second nature - by which time notations in which the distinction is made clearer possibly only confuse still further; witness the problems beginners often have with pointer types in C++ or Modula-2, where `*p` or `p↑` (respectively) denotes the explicit value residing at the explicit address `p`. If we were to relate this back to the productions used in our grammar, we should find that each `x` in the above assignment was syntactically a *Designator*. Semantically these two designators are very different - we shall refer to the one that represents an address as a *Variable Designator*, and to the one that represents a value as a *Value Designator*.

To perform its task, the code generation interface will require the extraction of further information associated with user-defined identifiers that is best kept in the symbol table. In the case of constants we need to record the associated values, and in the case of variables we need to record the associated addresses and storage demands (structures typically occupy several words forming a contiguous block in memory, and even a simple integer variable occupies two, four or even more bytes). If we can assume that our machine incorporates a "linear array" model of memory, this information is easily added as the declarations are encountered.

Handling the different sorts of entries that need to be stored in a symbol table can be done in various ways. In an object-oriented class-based implementation one might define an abstract base class to represent a generic type of entry, and then derive classes from this to represent entries for variables or constants (and, in due course, records, structures, functions, methods, classes and any other forms of entry that seem to be required). Another,

traditional way, still required if one is hosting a compiler in a language that does not support inheritance as a concept, is to make use of a **variant record** (in Modula-2 terminology) or **union** (in C++ terminology).

Parva, as so far defined, does not warrant all of this mechanism however, especially since we are targeting a machine as simple as the PVM where the underlying "word" of memory is deemed capable of storing a datum of any sort, such as a Boolean, an integer or a reference (address). For the moment, we shall simply add two fields to the *Entry* class that we first encountered in section 12.6.5 - one to record the memory address associated with an entry if it corresponds to a variable, and the other to record the value associated with an entry if it corresponds to a constant.

```
class Entry {
  public int    kind;        // Kinds
  public int    type;        // Types
  public string name;        // lexeme
  public int    value;       // constants
  public int    offset;      // variables
  public Entry  nextInScope; // link to next entry in current scope
  public bool   declared;    // true for all except sentinel entry
} // class Entry
```

(We could even make one field serve both purposes, but in the interests of making the code in this chapter more readable we have chosen not to do this.)

The specification of the *Table* class of section 12.6.5 does not have to change, but the productions that handle declarations must be attributed to enable the extraction of the information needed to set up the new fields. To handle the attributes of literal constants we introduce a simple class.

```
class ConstRec {
// Objects of this type are associated with literal constants
  public int value; // value of a constant literal
  public int type;  // constant type (determined from the literal)
} // class ConstRec
```

In Chapter 4 we introduced the concept of reserving storage for the variables of a simple program in an activation record or stack frame. This is, effectively, a sub-array indexed by the offset associated with each variable. Accordingly, we introduce a simple class that makes provision for recording the size of such frames.

```
class StackFrame {
// Objects of this type are used to keep track of the total space
// needed by the variables as they are declared in nested blocks
  public int size = 0;  // current frame size; also next offset
} // class StackFrame
```

In passing, the introduction of the `StackFrame` class is mandated by the inability of Java to pass simple parameters "by reference"; the best one can do is pass a reference to an object ("by value") to a method and allow the method to alter a field of that object. There is a much neater way of keeping track of the stack fram size if one can program in C#, but the details of this are left as an insightful exercise.

The way in which objects of these types are manipulated should be apparent from the amended productions for *OneConst, Constant* and *OneVar*, which are the ones that handle the creation and addition of symbol table entries. We note that the size of a stack frame is increased by one unit each time a variable declaration is encountered and that the current size also serves to predict the offset of the next variable to be declared. This is only the case because each of our variables - including array reference variables - can be accommodated in a single word. More generally, we might have to increment the frame size by an amount that would depend on the storage requirements prescribed by the variable's type. However, the principle would still be the same as we have illustrated.

```
OneConst          (. Entry constant = new Entry();
                     ConstRec con; .)
=  Ident<out constant.name>
                  (. constant.kind = Kinds.Con; .)
   AssignOp Constant<out con>
                  (. constant.value = con.value;
                     constant.type = con.type;
                     Table.Insert(constant); .) .
```

```
Constant<out ConstRec con>
                  (. con = new ConstRec(); .)
=   IntConst<out con.value>
                  (. con.type = Types.intType; .)
  | CharConst<out con.value>
                  (. con.type = Types.intType; .)
  | "true"        (. con.type = Types.boolType;
                     con.value = 1; .)
  | "false"       (. con.type = Types.boolType;
                     con.value = 0; .)
  | "null"        (. con.type = Types.nullType;
                     con.value = 0; .) .

OneVar<StackFrame frame, int type>
                  (. int expType; .)
=                 (. Entry var = new Entry(); .)
   Ident<out var.name>
                  (. var.kind = Kinds.Var;
                     var.type = type;
                     var.offset = frame.size;
                     frame.size++; .)
   [ AssignOp     (. CodeGen.LoadAddress(var); .)
     Expression<out expType>
                  (. if (!Compatible(var.type, expType))
                        SemError("incompatible types");
                     CodeGen.Assign(var.type); .)
   ]              (. Table.Insert(var); .) .
```

13.2 The code generation interface

In considering the interface between analysis and code generation it will again pay to aim for some degree of machine independence. Generation of code should take place without too much, if any, knowledge of how the analyzer works. Even in more general situations than we seem to be considering, a common technique for achieving this seemingly impossible task is to define a hypothetical machine (such as a stack machine) with instruction set and architecture convenient for the execution of programs of the source language, but without being too far removed from the actual system for which the compiler is required. The action of the interface methods is then conceptually to translate the source program into an equivalent sequence of operations for the hypothetical machine. Calls to these methods can be embedded in the parser without overmuch concern for how the final generator will turn the operations into "real" object code for the target machine. Of course, if the target machine really *is* a stack machine, the code generation methods will be very simple - as we shall see.

A little reflection on this theme might suggest that the specification of such a code generation class could take the form below.

```
class CodeGen {
  public const int
    undefined  = -1,  // for forward reference chains

    nop =  1, add =  2, sub =  3, mul =  4, div =  5, rem =  6,
    and =  7, or  =  8, ceq =  9, cne = 10, clt = 11, cge = 12,
    cgt = 13, cle = 14;

  public static void NegateInteger()
  // Generates code to negate integer value on top of stack

  public static void NegateBoolean()
  // Generates code to negate boolean value on top of stack

  public static void BinaryOp(int op)
  // Generates code to pop two values A,B from stack
  // and push value A op B

  public static void Comparison(int op)
  // Generates code to pop two values A,B from stack
  // and push Boolean value A op B

  public static void Read(int type)
  // Generates code to read a value of specified type
  // and store it at the address found on top of stack

  public static void Write(int type)
  // Generates code to output value of specified type from
  // top of stack
```

```
    public static void WriteLine()
    // Generates code to output line mark

    public static void WriteString(string str)
    // Generates code to output string str stored at known location

    public static void LoadConstant(int number)
    // Generates code to push number onto evaluation stack

    public static void LoadAddress(Entry var)
    // Generates code to push address of variable var onto stack

    public static void Index()
    // Generates code to index an array on the heap

    public static void Dereference()
    // Generates code to replace top of evaluation stack by the value
    // found at the address currently stored on top of the stack

    public static void Assign(int type)
    // Generates code to store value currently on top-of-stack on the
    // address given by next-to-top, popping these two elements

    public static void Allocate()
    // Generates code to allocate an array on the heap

    public static void OpenStackFrame()
    // Generates incomplete code to reserve space for variables

    public static void FixDSP(int location, int size)
    // Fixes up DSP instruction at location to reserve size space
    // for variables

    public static void LeaveProgram()
    // Generates code needed to leave a program (halt)

    public static void Branch(Label destination)
    // Generates unconditional branch to destination

    public static void BranchFalse(Label destination)
    // Generates branch to destination, conditional on the Boolean
    // value currently on top of the stack, popping this value

    public static void BackPatch(int adr)
    // Stores the current location counter as the address field of a
    // branch or call instruction currently holding a forward
    // reference to adr and repeatedly works through a linked list
    // of such instructions

    public static void Stack() {
    // Generates code to dump the current state of the evaluation stack

    public static void Heap() {
    // Generates code to dump the current state of the runtime heap

    public static int GetCodeLength()
    // Returns length of generated code

    public static int GetInitSP()
    // Returns position for initial stack pointer

} // class CodeGen
```

13.3 Using the code generation interface

The source code in the *Resource Kit* shows all the places at which calls to the methods suggested in the previous section are added to the attributed grammar. It will suffice here to give several extracts for, as usual, there are several points that benefit from further comment and explanation.

- The code generation methods have been given names that might suggest that they actually *perform* operations like `Branch`. They only generate *code* for such operations, of course.

- There is an unavoidable interaction between this class and the machine for which code is to be generated. We have seen fit to define a method (`GetCodeLength`) that will allow the compiler to determine the amount of code generated, and another (`GetInitSP`) that will return an initial value that can be assigned to the hypothetical machine register SP after computing the demands of the string pool in high memory.

- The concepts of the meaning of an expression, and of assignment of the "values" of expressions to locations in memory labelled with the "addresses" of variables are probably well understood by the reader. As we have seen in section 4.5, such operations are very easily handled by translating the normal infix notation used in describing expressions into a postfix or Polish equivalent (similar to that discussed in section 9.1). These are then easily handled with the aid of an evaluation stack, the elements of which are either addresses of storage locations, or the values found at such addresses.

  Code for these operations can be generated by making calls on methods like `LoadConstant, LoadAddress, Index` and `Dereference` for storage access and by calls to methods like `NegateInteger, NegateBoolean` and `BinaryOp` to generate code to perform simple arithmetic and logic. Generation of code for comparisons needed within Boolean expressions is achieved by calls on `Comparison`, suitably parameterized according to the test to be performed. Finally, calls to `Assign` handle the familiar assignment process.

  An example will clarify this. Compilation of the Parva assignment statement

```
Result = List[5] + MAX;
```

  should give rise to the following sequence of code generator method calls (and the production of code like that illustrated for the PVM).

```
LoadAddress(Result)          LDA  0
  LoadAddress(List)          LDA  1
  Dereference                LDV
  LoadConstant(5)            LDC  5
  Index                      LDXA
  Dereference                LDV
  LoadConstant(MAX)          LDC  400
  BinaryOp(add)              ADD
Assign(intType)              STO
```

  In understanding the mechanism for putting this all together, note that the above example has three examples of what the context-free syntax has described as a *Designator*. The designator for `Result` is a *variable designator*, and this implies that when the production for *Assignment*

```
Assignment       (. int expType;
                    DesType des; .)
=  Designator<out des>
                 (. if (des.entry.kind != Kinds.Var)
                      SemError("invalid assignment"); .)
   AssignOp
   Expression<out expType>
                 (. if (!Compatible(des.type, expType))
                      SemError("incompatible types");
                    CodeGen.Assign(des.type); .)
   WEAK ";" .
```

  calls upon *Designator* it will be the latter's responsibility to generate the code that will push the appropriate address onto the hypothetical machine stack. By contrast, the designator for `List[5]` is a *value designator*, and this implies that when the production for *Factor* calls upon *Designator*, code must be generated that will compute the address of `List[5]`, followed by code that will perform the dereferencing operation. Since the production for *Factor* will also call upon *Designator* when it has to handle another value designator - that for the constant `MAX` - some care has to be taken with the way in which the *Designator* production is attributed. One possibility is shown below.

```
Designator<out DesType des>
                 (. string name;
                    int indexType; .)
=  Ident<out name>
                 (. Entry entry = Table.Find(name);
                    if (!entry.declared)
                      SemError("undeclared identifier");
                    des = new DesType(entry);
                    if (entry.kind == Kinds.Var)
                      CodeGen.LoadAddress(entry); .)
   [  "["        (. if (IsRef(des.type)) des.type--;
                    else SemError("unexpected subscript");
                    if (entry.kind != Kinds.Var)
                      SemError("unexpected subscript");
                    CodeGen.Dereference(); .)
         Expression<out indexType>
```

```
                  (. if (!IsArith(indexType))
                       SemError("invalid subscript type");
                     CodeGen.Index(); .)
      "]"
   ] .
```

This has the effect that any calls to *Designator* that involve a variable have the effect of generating code to compute and push onto the stack the address of that variable (which may be an array element). Calls that involve a constant check the constraints and return the output attribute `des` but do not generate any code. A value designator appears semantically as a component of a *Factor*, so that the production for *Factor* must follow up a call to *Designator* with an appropriate call either to the `Dereference` or `LoadConstant` method.

```
Factor<out int type>
                  (. int value = 0;
                     type = Types.noType;
                     int size;
                     DesType des;
                     ConstRec con; .)
=    Designator<out des>
                  (. type = des.type;
                     switch (des.entry.kind) {
                       case Kinds.Var:
                         CodeGen.Dereference();
                         break;
                       case Kinds.Con:
                         CodeGen.LoadConstant(des.entry.value);
                         break;
                       default:
                         SemError("wrong kind of identifier");
                         break;
                     } .)
   | Constant<out con>
                  (. type = con.type;
                     CodeGen.LoadConstant(con.value); .)
   | "new" BasicType<out type>
                  (. type++; .)
     "[" Expression<out size>
                  (. if (!IsArith(size))
                       SemError("array size must be integer");
                     CodeGen.Allocate(); .)
     "]"
   | "!" Factor<out type>
                  (. if (!IsBool(type))
                       SemError("bool operand needed");
                     else CodeGen.NegateBoolean();
                     type = Types.boolType; .)
   | "(" Expression<out type> ")" .
```

The *Factor* production also has the responsibility for generating code that will create new array objects dynamically when the program runs, which it does by calling the `Allocate` method when required.

- The reverse Polish (postfix) form of expression manipulation is achieved simply by delaying the calls for code generation of an "operation" until code generation has occurred for the second "operand". This is completely analogous to the system developed in section 9.1 for converting the textual form of an infix expression to its reverse Polish equivalent. It will suffice to illustrate one of the productions, that for *Term*.

```
Term<out int type>
                  (. int type2;
                     int op; .)
=  Factor<out type>
   { MulOp<out op> Factor<out type2>
                  (. switch (op) {
                       case CodeGen.and:
                         if (!IsBool(type) || !IsBool(type2))
                           SemError("bool operands needed");
                         type = Types.boolType;
                         break;
                       default:
                         if (!IsArith(type) || !IsArith(type2)) {
                           SemError("arith operands needed");
                           type = Types.noType;
                         }
                         break;
                     }
                     CodeGen.BinaryOp(op); .)
   } .
```

(This does not provide "short-circuit" semantics for AND and OR operations - see section 13.5.2.)

- To generate code to handle simple I/O operations we can call on the methods `Read, Write, WriteString` and `WriteLine`. As an example, consider the production for *WriteElement*.

```
WriteElement      (. int expType;
                     string str; .)
=   StringConst<out str>
                  (. CodeGen.WriteString(str); .)
  | Expression<out expType>
                  (. switch (expType) {
                       case Types.intType:
                       case Types.boolType:
                         CodeGen.Write(expType); break;
                       default:
                         SemError("cannot write this type");
                         break;
                     } .) .

StringConst<out string str>
=  stringLit      (. str = token.val;
                     str = Unescape(str.Substring(1, str.Length-2)); .) .
```

  The associated production for *StringConst* extracts a quoted string from the source, discards the surrounding quotes, and then calls on a method `Unescape` that replaces any internal escape sequences like `\t` by their correct values.

- Control statements are a little more interesting. In the type of machine being considered it is assumed that machine code will be executed in the order in which it was generated, except where explicit branching operations occur. Although our simple language does not incorporate the somewhat despised `GOTO` statement, this maps very closely onto real machine code - and must form the basis of code generated by higher level control statements. The transformation is very easily automated, save for the problem of forward references. In our case there are two statement forms that give rise to these. Source code of the conceptual form

  `IF` *Condition* `THEN` *Statement*

  should lead to object code of the more fundamental form

  code for *Condition*
  `IF NOT` *Condition* `THEN GOTO LAB END`
  code for *Statement*
  `LAB` continue

  but when we get to the stage of generating `GOTO LAB` we do not know the address that will apply to `LAB`. Similarly, source code of the conceptual form

  `WHILE` *Condition* `DO` *Statement*

  could lead to object code of the form

  `LAB` code for *Condition*
  `IF NOT` *Condition* `THEN GOTO EXIT END`
  code for *Statement*
  `GOTO LAB`
  `EXIT` continue

  Here we should know the address of `LAB` as we start to generate the code for *Condition*, but we shall not know the address of `EXIT` when we get to the stage of generating `GOTO EXIT`.

  In general the solution to this problem might require the use of a two-pass system. However, since we are developing a one-pass compiler for which the generated code can all be held temporarily in memory, or at worst on a random access file, later modification of addresses in branch instructions can easily be effected. This affords us an excellent chance to reuse the *Label* class introduced in section 11.1 For convenience we show the important parts of the specification again.

```
class Label {
  public Label(bool known)
  // Constructor for label, possibly at already known location

  public int Address()
  // Returns memAdr if known, otherwise effectively adds to a
  // forward reference chain that will be resolved if and when
  // Here() is called and returns the address of the most recent
  // forward reference

  public void Here()
  // Defines memAdr of this label to be at current location
  // counter after fixing any outstanding forward references
} // class Label
```

The use of objects of this class solves the problem very easily, as the following productions reveal. A little thought should show that a consequence of allowing only the structured *WhileStatement* and *IfStatement* are that we need the same number of implicit labels for each instance of such constructs. These labels can be handled by declaring appropriate variables local to parsing methods like *IfStatement*. Each time a recursive call is made to *IfStatement* new variables will come into existence, and remain there for as long as it takes to complete parsing of the construction, after which they will be discarded.

Clearly, all variables associated with handling implicit forward references must be declared locally, or chaos will ensue.

```
IfStatement<StackFrame frame>
                  (. Label falseLabel = new Label(!known); .)
=  "if" "(" Condition ")"
                  (. CodeGen.BranchFalse(falseLabel); .)
   Statement<frame>
                  (. falseLabel.Here(); .) .


WhileStatement<StackFrame frame>
                  (. Label startLoop = new Label(known); .)
=  "while" "(" Condition ")"
                  (. Label loopExit = new Label(!known);
                     CodeGen.BranchFalse(loopExit); .)
   Statement<frame>
                  (. CodeGen.Branch(startLoop);
                     loopExit.Here(); .) .
```

In the above code the identifier `known` is synonymous with `true` and has been introduced simply to make for greater understanding.

- We shall need to generate special housekeeping code as we enter or leave a function. This may not be apparent in the case of a single function program - which is all our language allows at present - but will certainly be the case when we extend the language to support multiple functions. This code can be generated by the methods `OpenStackFrame` (for code needed as we start the program) and `LeaveProgram` (for code needed as we leave it to return, perhaps, to the charge of some underlying operating system). The code below shows how it is expedient to isolate the *Block* of the overall program from the production that handles nested blocks, for ease of adding the housekeeping code.

```
Parva
=  "void"         (. Entry program = new Entry(); .)
   Ident<out program.name> "(" ")"
                  (. program.kind = Kinds.Fun;
                     program.type = Types.voidType;
                     Table.Insert(program);
                     StackFrame frame = new StackFrame();
                     Table.OpenScope();
                     Label DSPLabel = new Label(known);
                     CodeGen.OpenStackFrame(); .)
   WEAK "{" { Statement<frame> }
   WEAK "}"       (. CodeGen.FixDSP(DSPLabel.Address(),
                                    frame.size);
                     CodeGen.LeaveProgram();
                     Table.CloseScope(); .) .
```

Note that the code generation encounters another forward reference problem - at the point where we call `OpenStackFrame`, an incremental one-pass compiler cannot know how many variables have yet to be declared. By the time the *Block* that constitutes the program has been parsed, the frame size will be known. Note how

the `frame` object is passed to the *Statement* parser for this purpose, allowing a later backpatching operation to be effected by the `FixDSP` method.

- It turns out to be useful for debugging purposes, and for a full understanding of the way in which our machine works, to be able to print out the evaluation stack and/or heap at any point in the program. This we might do by introducing keywords like `stackdump` and `heapdump`, which can appear as simple statements, and whose code generation is handled by the `Stack` and `Heap` methods.

13.4 Code generation for the PVM

The problem of code generation for a real machine is, in general, complex, and very specialized. In this section we shall content ourselves with completing our first-level Parva compiler on the assumption that we wish to generate code for the PVM stack machine described in section 4.4. Such a machine does not exist, but, as we saw, it may readily be emulated by a simple interpreter. Indeed, if the `Emulate` method of section 4.6 is invoked from the driver program after completing a successful parse, an implementation of Parva quite suitable for experimental work is readily produced.

An implementation of an "on-the-fly" code generator for this machine is very straightforward and can be found among the source code for this chapter in the *Resource Kit*. In studying this code the reader should note the following.

- The main part of the code generation is done in terms of calls to a private method `Emit`, which does some error checking to ensure that the capacity of the code array used to simulate the memory of the PVM has not overflowed. Storing a string in the literal pool in high memory is done by method `WriteString`, which also carries out overflow checking. As usual, errors are reported through the `SemError` method of the *Parser* class. The code generator suppresses further attempts to generate code if memory overflow occurs, while still allowing syntactic and constraint analysis to proceed.

```
static bool generatingCode = true;
static int codeTop = 0, stkTop = PVM.memSize;

private static void Emit(int word) {
// Code generator for single word
  if (!generatingCode) return;
  if (codeTop >= stkTop) {
    Parser.SemError("program too long");
    generatingCode = false;
  }
  else {
    PVM.mem[codeTop] = word; codeTop++;
  }
} // CodeGen.Emit

public static void WriteString(string str) {
  int L = str.Length, first = stkTop - 1;
  if (stkTop <= codeTop + L + 1) {
    Parser.SemError("program too long");
    generatingCode = false;
    return;
  }
  for (int i = 0; i < L; i++) {
    stkTop--; PVM.mem[stkTop] = str[i];
  }
  stkTop--; PVM.mem[stkTop] = 0;
  Emit(PVM.prns); Emit(first);
} // CodeGen.WriteString
```

In terms of `Emit`, many of the other methods are almost trivial, as can be seen from a representative selection below.

```
public static void Read(int type) {
  switch (type) {
    case Types.intType:  Emit(PVM.inpi); break;
    case Types.boolType: Emit(PVM.inpb); break;
  }
} // CodeGen.Read
```

```
public static void LoadAddress(Entry var) {
  Emit(PVM.lda); Emit(var.offset);
} // CodeGen.LoadAddress

public static void NegateBoolean() {
  Emit(PVM.not);
} // CodeGen.NegateBoolean

public static void BinaryOp(int op) {
  switch (op) {
    case CodeGen.mul:  Emit(PVM.mul); break;
    case CodeGen.div:  Emit(PVM.div); break;
    case CodeGen.rem:  Emit(PVM.rem); break;
    case CodeGen.and:  Emit(PVM.and); break;
    case CodeGen.add:  Emit(PVM.add); break;
    case CodeGen.sub:  Emit(PVM.sub); break;
    case CodeGen.or :  Emit(PVM.or);  break;
  }
} // CodeGen.BinaryOp

public static void Branch(Label destination) {
  Emit(PVM.brn); Emit(destination.Address());
} // CodeGen.Branch
```

- Since many of the methods in the code generator class are very elementary interfaces to `Emit`, the reader may feel that we have taken modular decomposition too far. Code generation as simple as this could surely be made more efficient if the parser simply evoked `Emit` directly. This is certainly true and many recursive descent compilers follow this route.

- Code generation is very easy for a stack-oriented machine. It is much more difficult for a machine with no stack and only a few registers and addressing modes. However, as the discussion in later sections will reveal, the specification we have developed is, in fact, capable of being used with only minor modification for code generation for more conventional machines. Developing the system in a highly modular way has many advantages if one is striving for portability and for the ability to construct improved back ends easily.

13.5 Towards greater efficiency

In the introduction to this chapter, the point was made that extensions to a language and improvements to its compiler are quite easily made if one is targeting a virtual machine and can take the liberty of extending the machine to match the language extensions, or to allow for the generation of shorter or more efficient code. This section illustrates that principle for a few language features that are encountered so commonly that it is worth making special provision for them, and for some features that would be very difficult to handle if the target machine had only the opcode set introduced in section 4.4, but which are quite easily handled if we can introduce further opcodes like those suggested in section 4.10.

13.5.1 Variations on assignment statements

As a first example, consider the common assignment statement of the form

```
Variable = Variable + Constant;
```

In many cases this would lead to simple PVM code like

```
LDA  Variable
LDA  Variable
LDV
LDC  Constant
ADD
STO
```

However, if the *Variable* were addressed by a more complex *VariableDesignator*, as illustrated by

```
List[A * B + C] = List[A * B + C] + Expression;
```

it is not hard to see that the code generated would be of the form

```
<Push address of Variable>
<Push address of Variable>
LDV
<Push value of Expression>
ADD
STO
```

in which the (extensive) section for computing the address of `Variable` appears twice. By the time the `LDV` instruction is obeyed at run-time the stack will contain two copies of this address. If the machine set includes an opcode (`DUP`) that pushes another copy of the element on top of the stack onto the stack, we could achieve the same effect with far less effort by generating code like

```
<Push address of Variable>
DUP
LDV
<Push value of Expression>
ADD
STO
```

Adding such an opcode to our virtual machine is extremely easy, as is adding the code generation interface to the *CodeGen* class. Being able to recognize when to take advantage of this optimization when presented with a general assignment statement is a topic that is beyond the scope of the present discussion, but the reader should easily see that if Parva were extended to provide variations on the assignment operator like those found in C# and Java, it would be very easy to compile a statement of a form exemplified by

```
Variable += Expression;
```

into PVM code of the form just illustrated. Furthermore, it would become correspondingly more difficult to generate code of the original form if the *Variable* appeared only once on the left of such an assignment operator.

We can take this still further. Another source code abbreviation is common in the special case where the *Expression* has the value 1. Rather than write

```
Variable = Variable + 1;
```

or even

```
Variable += 1;
```

C# and Java programmers would probably prefer

```
Variable++;
```

This, and the complementary form with the `--` operator occur, so frequently that it may be worth introducing special machine opcodes (`INC` and `DEC`) to allow the generation of code like

```
<Push address of Variable>
INC
```

where the semantics of `INC` and `DEC` could be described, in the notation of section 4.4.2, as

```
INC     Pop TOS and replace mem[TOS] by mem[TOS] + 1
DEC     Pop TOS and replace mem[TOS] by mem[TOS] - 1
```

13.5.2 Short-circuit semantics

Although we have introduced a Boolean type into Parva, along with `&&` (AND) and `||` (OR) operators, a study of the code generated for expressions involving these operators will reveal that they do not provide so-called **short-circuit** semantics, but rather the easier "Boolean operator" approach. In the short-circuit approach the operators AND and OR are defined to have semantic meanings such that

```
A AND B      means      IF A THEN B ELSE FALSE END
A OR  B      means       IF A THEN TRUE ELSE B END
```

A reader who attempts to generate code matching these semantics using only the very limited opcode set we have encountered so far will find this to be a non-trivial exercise. Fortunately the problem is neatly side-stepped if we simply introduce two new opcodes.

`BFALSE N`	Branch to instruction `N` if `TOS` is false (zero), otherwise pop and discard `TOS`
`BTRUE N`	Branch to instruction `N` if `TOS` is true (non-zero), otherwise pop and discard `TOS`

The use of these opcodes is exemplified in the following two fragments of code:

```
      A && B || C && D                A || B && (C || D)

      <Push value of A>               <Push value of A>
      BFALSE  L1                      BTRUE  L3
      <Push value of B>               <Push value of B>
L1    BTRUE   L2                      BFALSE L3
      <Push value of C>               <Push value of C>
      BFALSE  L2                      BTRUE  L3
      <Push value of D>               <Push value of D>
L2    ...                       L3    ...
```

The interesting way in which they are generated shows, perhaps, a rather more obscure way of attributing the grammar than we have previously encountered. It suffices to show the production for *Term* (the modifications to *AddExp* are very similar).

```
Term<out int type>
                    (. int type2;
                       int op;
                       Label shortcircuit = new Label(!known); .)
= Factor<out type>
  { MulOp<out op>
                    (. if (op == CodeGen.and)
                         CodeGen.BooleanOp(shortcircuit,
                                           CodeGen.and); .)
    Factor<out type2>
                    (. switch (op) {
                         case CodeGen.and:
                           if (!IsBool(type) || !IsBool(type2))
                             SemError("bool operands needed");
                           type = Types.boolType;
                           break;
                         default:
                           if (!IsArith(type) || !IsArith(type2)) {
                             SemError("arith operands needed");
                             type = Types.noType;
                           }
                           CodeGen.BinaryOp(op);
                           break;
                       } .)
  }                 (. shortcircuit.Here(); .) .
```

where, once again, we have to make use of a local object of the *Label* class to handle a forward reference problem, and where we have introduced a further interface to the code generator.

```
public static void BooleanOp(Label branch, int op)
// Generates code for short-circuit Boolean operator op
```

Notice that the `BFALSE` opcode is (incompletely) generated before the code for the second (and any subsequent) *Factor* is processed, and not after it, as is the case for the other multiplicative operators.

14 A PARVA COMPILER: FUNCTIONS AND PARAMETERS

Our simple system has so far not provided for the *procedure* or *function* concept in any real way. It is the aim of this chapter to show how Parva, its compiler and the PVM can be extended to support multi-function programs, using a model based on that found in C-like languages. The extended Parva language provides for the use of global variables and constants, and also functions with parameters, local variables and local constants. These functions are able to call one another, or even themselves recursively. Implementation of these features involves a deeper treatment of the concepts of storage allocation and management than we have needed previously.

In the accompanying *Resource Kit* will be found the full source code for the Cocol grammars and support classes covering this refinement, and the reader is encouraged to study this code in detail as he or she reads the text. As before, we shall develop a single-pass incremental compiler, and we shall do so in stages. We start by considering extensions to Parva to support programs consisting only of a set of parameterized void functions.

14.1 Void functions

The language features we wish to support should all be familiar to the reader, but it may help the discussion to present a simple example of a program that illustrates most of them : it sets up a list of values of the well-known function *n!* and then displays this in reverse order:

```
void display(int[] list, int n) {       // point c
  while (n >= 0) {
    write(n, list[n], "\n");
    n = n - 1;
  }
} // display

void compute(int[] fact, int limit) {
  int i, j, k = 1;
  fact[0] = 1;
  while (k <= limit) {
    fact[k] = fact[k-1] * k;
    k = k + 1;
  }                                     // point b
  display(fact, limit);                 // first call
} // compute

void main() {
  int max;
  read("Supply limit ", max);
  int[] list = new int[max + 1];        // point a
  compute(list, max);
  display(list, max);                   // second call
} // main
```

Within this program we can note several points of interest.

- There is a distinguished `main` function which declares and allocates storage for an array `list`, the reference to which is passed as one argument when `main` makes a call to the function `compute`.

- Function `main` is a so-called **void function**, as are the other two functions. Unlike the "typed" functions that we shall consider later, a void function does not compute and return a value. The declaration of a void function can be seen as a process whereby a *Block* is given a name. Quoting this name at later places in the program in the guise of a *Statement* gives rise to a **function call** and implies execution of that block. For greater flexibility the block may, as in the case of `compute` and `display`, be parameterized, so that its execution may differ appropriately from call to call.

- Function `compute` has two formal parameters, `fact` and `limit`, and three local variables `i`, `j` and `k`. The scope of all of these is constrained to the *Block* associated with `compute`. Similarly, function `display` has two formal parameters `list` and `n`, whose scopes are constrained to the *Block* associated with `display`.

14.1.1 Syntactic extensions to Parva to support void functions

At first glance the syntactic extensions we shall make to Parva appear straightforward. A Parva program will, in general, consist of a set of declarations of globally accessible functions. We introduce a new *Statement* form to

handle the invocation of void functions. Besides the definition of the associated *Block* (which is isolated in a single production for *Body* to assist with later code generation), the *declaration* of a function specifies a list of **formal parameters**, while a *call* to a function specifies a list of **actual parameters** or **arguments** to be passed to the function when the call takes place. Normally a void function returns control to the caller when execution reaches the end of its *Block*, but a simple *ReturnStatement* can effect an early exit if needed. These additions are readily described in a context-free way by the productions below.

```
PRODUCTIONS /* some omitted to save space */
  Parva               = { VoidFuncDeclaration } .
  VoidFuncDeclaration = "void" identifier "(" FormalParameters ")" Body .
  FormalParameters    = [ OneParam { "," OneParam } ] .
  OneParam            = Type identifier .
  Body                = Block .
  Statement           = Assignment | VoidFunctionCall | ...
  Assignment          = Designator "=" Expression .
  VoidFunctionCall    = FunctionCall ";" .
  FunctionCall        = identifier "(" Arguments ")" .
  Arguments           = [ OneArg { "," OneArg } ] .
  OneArg              = Expression .
  Designator          = identifier [ "[" Expression "]" ] .
  ReturnStatement     = "return" ";" .
```

14.1.2 An LL(1) grammar for Parva

Inspection will show that the descriptive grammar of the previous section is badly non-LL(1). Two alternatives in *Statement* both begin with an identifier. Previous grammars for Parva have been written so that all applied references to identifiers are handled initially by the production for *Designator*. At first it might not seem particularly difficult to refactorize the grammar to overcome the LL(1) problem as below:

```
PRODUCTIONS /* some omitted to save space */
  Statement           = AssignmentOrCall | ...
  AssignmentOrCall    = Designator
                        (   "=" Expression
                          | "(" Arguments ")"
                        ) ";" .
```

In the context of a *Statement*, a *Designator* associated with a variable must be followed by an assignment operator, while a *Designator* associated with a function must be followed by a parenthesized argument list, and the associated function must have been declared of `void` type. Presumably, therefore, we can achieve the desired effect by careful semantic checks (and in the published version of this book (Terry, 2005) that is what was done). However, this suggestion is somewhat sloppy. because it has relaxed the syntax too far. It is better to make use of the idea of a **conflict resolver**, first mentioned in Chapter 10, and which will be fully illustrated later in this chapter. (Conflict resolvers were not part of Cocol at the time the first draft of Terry (2005) was produced.)

```
  Statement           = AssignmentOrCall | ...
  AssignmentOrCall    = ( IF (IsCall(...))
                            identifier "(" Arguments ")"
                          | Designator "=" Expression
                        ) ";" .
```

14.2 Constraint analysis

It should be apparent that the grammar as suggested in the last section will require the attributes for each production to be carefully constructed to distinguish what we might think of as elements of syntax from aspects of semantics. Note the following in particular.

- `main` is not a keyword, and the main function is syntactically and semantically similar to other void functions. However, it will be necessary to check that such a function has been declared, to give it a special status and to set up an automagic call to it to commence execution..

- When a call is made to a function, the mandatory list of arguments must match the list of formal parameters associated with that function - there must be the same number of each, and the type of each actual argument must be compatible with the type of the corresponding formal parameter. Providing an "overloading" capability is beyond the scope of the present study.

14.2.1 Symbol table structures

The need to check semantic constraints on the use of function identifiers (and, later, to be able to generate code for function calls) requires that we extend our earlier definition of the *Entry* class used by the symbol table.

So far as symbol table entries for function identifiers are concerned, there are three aspects to consider. We shall consider the code generation phase later, but it should be apparent that this will require each function to be labelled with the starting address for its low-level code. The need to check that formal parameter lists match actual argument lists implies that we also record the number of parameters, although this in itself is insufficient. Somewhat more subtly, we shall need extra functionality if type checking is to be possible even when the identifiers for the formal parameters are no longer in scope. One possibility is to add (to the fields specific to function identifiers) an extra reference field. This can persistently point to the queue of formal parameter and local variable entries that is created as the parameters are encountered in the normal course of parsing the function declaration.

No special provision has to be made for the formal parameters of functions, as in most respects (including scope) they are similar to locally declared variables.

A revised definition of the *Entry* class follows, which should be compared with that given earlier in section 13.1.

```
class Entry {
  public int    kind;        // Kinds
  public int    type;        // Types
  public string name;        // lexeme
  public int    value;       // constants
  public int    offset;      // variables
  public int    nParams;     // functions
  public Label  entryPoint;
  public Entry  firstParam = null;
  public Entry  nextInScope; // link to next entry in current scope
  public bool   declared;    // true for all except sentinel entry
} // class Entry
```

Here, as before, we note that the fields `value`, `offset` and `nParams` are mutually exclusive and that we could economize by using a single common field for all three purposes.

One way in which objects of the *Entry* class could be linked together to form a symbol table is demonstrated in Figure 14.1. This shows the state of the table as it would be constructed for the sample program given earlier, at three points in the parsing process. Figure 14.1(b) shows that by the end of parsing the body of function `compute`, the globally accessible identifiers `display` and `compute` would be linked together in the outermost scope, while the top level of scope would link together the formal parameters `fact` and `limit` and the local variables `i`, `j` and `k` in the order in which they were encountered by the parser. Although the symbol table entries for `list` and `n` (the formal parameters of `display`) cannot now be found by a symbol table search that starts from either of the current scope nodes, they can still be reached by following the link maintained by the `firstParam` field of the entry for `display`.

By the end of parsing the body of function `main`, the symbol table would appear as in Figure 14.1(c). The link from the `firstParam` field of the entry for `compute` still points to the entries for the parameters and local variables of this function, but these entries cannot be reached by a symbol table search starting from any of the current scope nodes.

Although it might appear that the distinction between formal parameters and local variables is blurred in this structure, independently recording the number of formal parameters for each function will allow the semantic checking needed when parsing argument lists to proceed with relative ease.

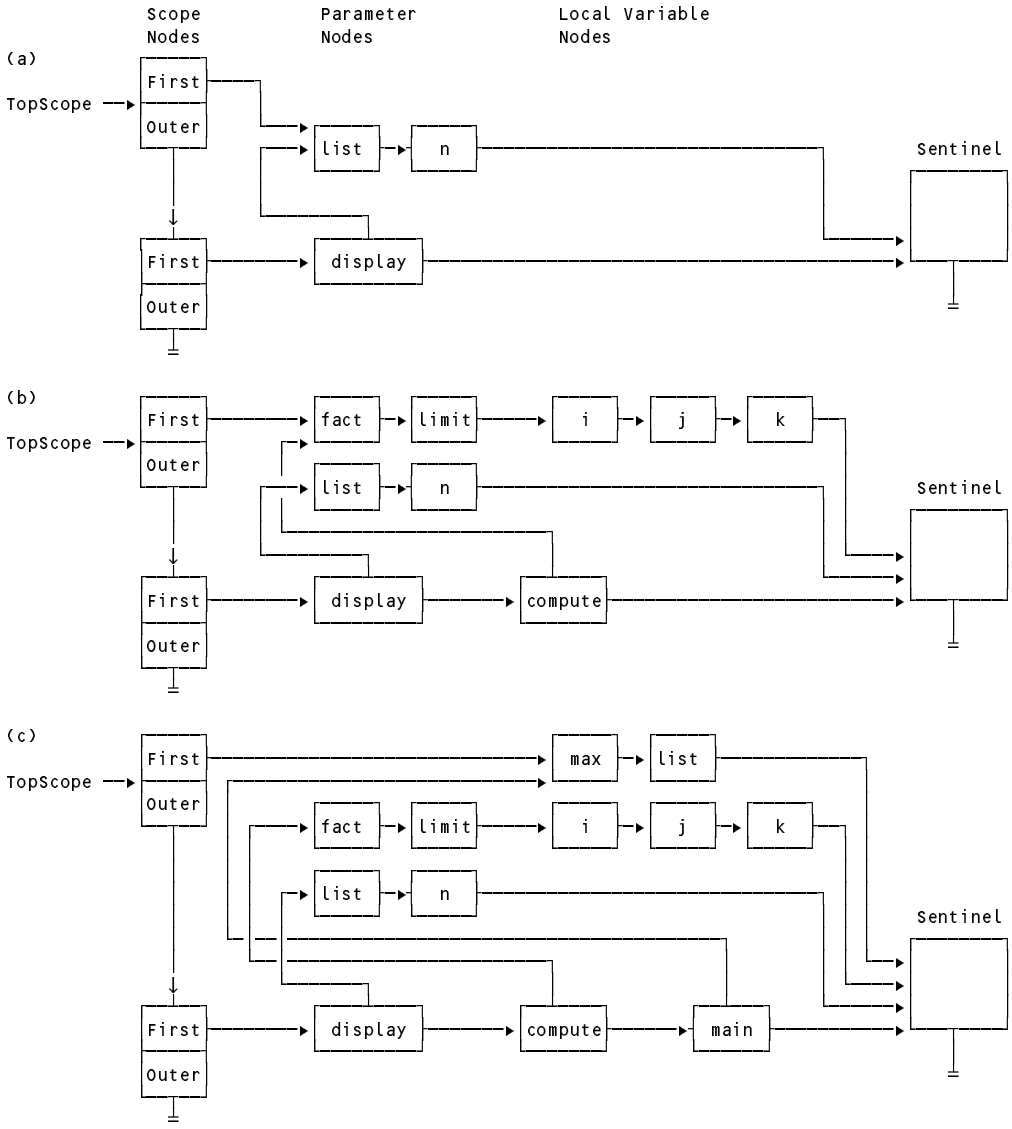


Figure 14.1 A symbol table structure with links from function entries to formal parameter entries: (a) at the stage of parsing the block for function display; (b) at the stage of parsing the block for function compute; (c) at the stage of parsing the block for function main.

14.3 Run-time storage management

If we wish functions to be able to call one another - possibly recursively - we shall have to think carefully about code generation and storage management. At run-time there may at any given time be several functions that have been called, but which are still pending completion, as only the most recently called function can effectively be "active" in a single-threaded program of the sort we are considering. A recursive program may even have several *instances* of its recursively defined functions pending completion. For each of these pending functions the corresponding instances of any arguments and local variables must be distinct. This has a rather complicating effect at compile-time, for a compiler cannot associate a simple unique address with any variable as it is declared (except, perhaps, for the global variables in what amounts to a surrounding program block, as we shall discuss later). Other aspects of code generation are not quite so problematic, although we must be on our guard always to generate only so-called **re-entrant code**, which executes without ever modifying itself.

14.3.1 Activation records and the stack frame concept

Just as the stack concept turns out to be useful for dealing with the compile-time accessibility aspects of *scope* in block-structured languages, so too do stack structures provide a solution for dealing with the run-time aspects of

extent or *existence*. Each time a function is called, it acquires a region of free store for its local variables and arguments - an area which can later be freed when control returns to the caller. On a stack machine this becomes almost trivially easy to arrange, although it may be more obtuse on other architectures. Since function activations strictly obey a first-in-last-out scheme, the areas needed for their local working store can be carved out of a single large stack. Such areas are usually called **activation records** or **stack frames**.

This may be made clearer by a simple example that serves only to illustrate these points.

```
void two(int x) {
  if (x > 1) two(x-1);  // recursive call
} // two

void one() {
  two(1);
} // one

void main() {
  one();
  two(3);
} // main
```

The dynamic execution of this fascinating program would result in an activation record history as shown below, where each line represents the relative layout of the activation records as the functions are entered and left.

	Stack grows ——————►			
start `main` function	`main`			
call `one()`	`main`	`one`		
call `two(1)`	`main`	`one`	`two`	
`two(1)` completes	`main`	`one`		
`one()` completes	`main`			
call `two(3)`	`main`	`two`		
call `two(2)` (recurse)	`main`	`two`	`two`	
call `two(1)` (recurse)	`main`	`two`	`two`	`two`
`two(1)` completes	`main`	`two`	`two`	
`two(2)` completes	`main`	`two`		
`two(3)` completes	`main`			

We first encountered the concept of an activation record in section 4.4.1 as it related to the architecture of the PVM. For the single function programs discussed previously it was necessary only for the single activation record to reserve space for the variables of that function. To support function activation of the sort now being considered it is usual to extend the role of the activation record. Although it does not contain any code, it is typical for each one to store some standard information that can be set up and used by the code that is executed when a function is called, and again when it has discharged its duty. This includes the **return address** through which control will eventually pass back to the calling function, as well as information that can later be used to reclaim the storage for the frame when it is no longer required, and provision for transmitting the value (if any) returned by the function to the caller. This housekeeping section of the frame is called the **frame header** or **linkage area** and can take on a standard layout for each function. In addition to the storage needed for the frame header, appropriate space is reserved in the allocation record for local variables and arguments. A variation on this idea will later be used to allocate storage for globally declared variables in a global activation record, which differs from those just described in that it will require little (if any) in the way of housekeeping components.

The run-time activation records for those functions that have been activated are conveniently maintained in a linked list. The pointer to the top of this linked structure - that is, to the base of the most recently created activation record - is usually given special status and is known as the current **base pointer** or **frame pointer**. Many modern architectures provide a special machine register especially suited to this role. The PVM is no exception and provides the `FP` register for exactly this purpose, as well as a similar register `GP` that points to the base of the activation record in which storage is allocated for any globally declared variables before the `main` function is activated.

As it happens, this is almost all the architectural support we need to implement the function model in Parva. It is inadequate for implementing languages like Pascal and Modula-2, which allow for functions to be nested within one another (a topic which is discussed in section 14.6 of Terry (2005), but which is beyond the scope of this derived extract). For the moment we note that at *run-time* the actual address of a variable or argument somewhere in memory will have to be found by subtracting an offset (which, fortunately, *can* be determined at *compile-time*)

from the address of the base of the appropriate activation record (an address which, naturally but unfortunately, *cannot* be predicted at compile-time but which can be relied on to be stored in either the FP or GP register at run-time). As applied to local variables this is, of course, exactly the addressing mechanism we have already employed in the implementation of Parva.

An activation record can be set out in various ways. Since we have the freedom to extend the PVM as best suits the back end of our compiler, we shall choose a layout that simplifies both the passing of parameters and the retrieval of the value that is returned by a non-void function, and require each activation record to assume the form shown in Figure 14.2.

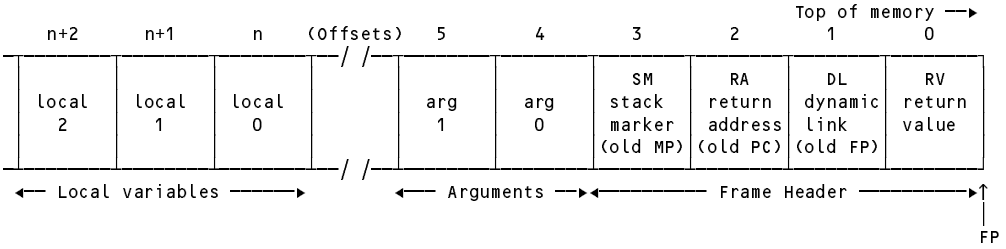


Figure 14.2 Activation record layout for the PVM.

It should be stressed that this is not the only arrangement possible. Another very common variation places the storage for the arguments above what we have denoted as the frame header. The arrangement shown here has the advantage of providing simple sequential numbering of the arguments and local variables.

14.3.2 Calling a void function

By now it should not take much imagination to see that calling a void function is not handled by simply executing a machine level JSR instruction, but rather by the execution of a complex activation and calling sequence.

Function *activation* is that part of the sequence that reserves storage for the frame header and evaluates the actual parameters needed for the call. The nuances of parameter handling will be discussed in section 14.3.5, but in anticipation we shall postulate that the calling routine initiates activation by executing code that:

- saves the current stack pointer SP in a special register known as the **mark stack pointer** MP;
- decrements SP so as to reserve storage for the frame header; before
- dealing with the arrangements for transferring the arguments, if any, to the area reserved for them below the frame header.

Once the arguments have been set up, the function is *called*. Code is then executed that saves, in the appropriate elements of the activation record that has just been allocated:

- a *dynamic link* DL - a copy of the current FP, forming a pointer to the base of the activation record of the calling routine;
- a *return address* RA - the code address of the instruction that follows the call, and to which control must finally return in the calling routine.

Thereafter the FP register can be reset to the value that was previously saved in MP and control transferred to the main body of the called function, which assumes responsibility for decrementing the SP register still further so as to reserve storage for its local variables.

In anticipation of extending the system to handle value-returning functions in a later section we have reserved the first location in an activation record (that is, at an invariant offset from the frame pointer FP) for a function's return value RV. This location is not really needed for void functions but it makes for easier code generation to keep all frame headers a constant size. Although not at first obvious, in this arrangement we also need to reserve an element SM for saving the old mark stack pointer at function activation, so that it can be restored once a function has run its course. The reason for this is that the computation of the values of one function's arguments may involve further function calls, as exemplified by code of the form

```
result = function(f(x), g(x+y));
```

These points may be clarified by a study of Figure 14.3, which depicts the state of the run-time activation records for the sample program given earlier at two points during its execution. Figure 14.3(a) shows the frame set up at the point where the void function `main` has been called (as a special case) and has read a value 6 for `max` and allocated space for an array of 7 elements on the heap (which starts from location 146). Figure 14.3(b) shows the two frames set up at the stage where `compute` has been activated and called, and has reached the point where the table has been set up (local variable `k` has by now reached the value of 7). Figure 14.3(c) shows what has taken place after function `display` has been called and its formal parameters `list` and `n` initialized to the values 146 and 6. When `display` terminates, control passes back to `compute` and then immediately back to `main` again - `main` then reactivates and calls `display` for a second time; in this case the stack frame for `display` occupies the space recovered when `compute` terminated.

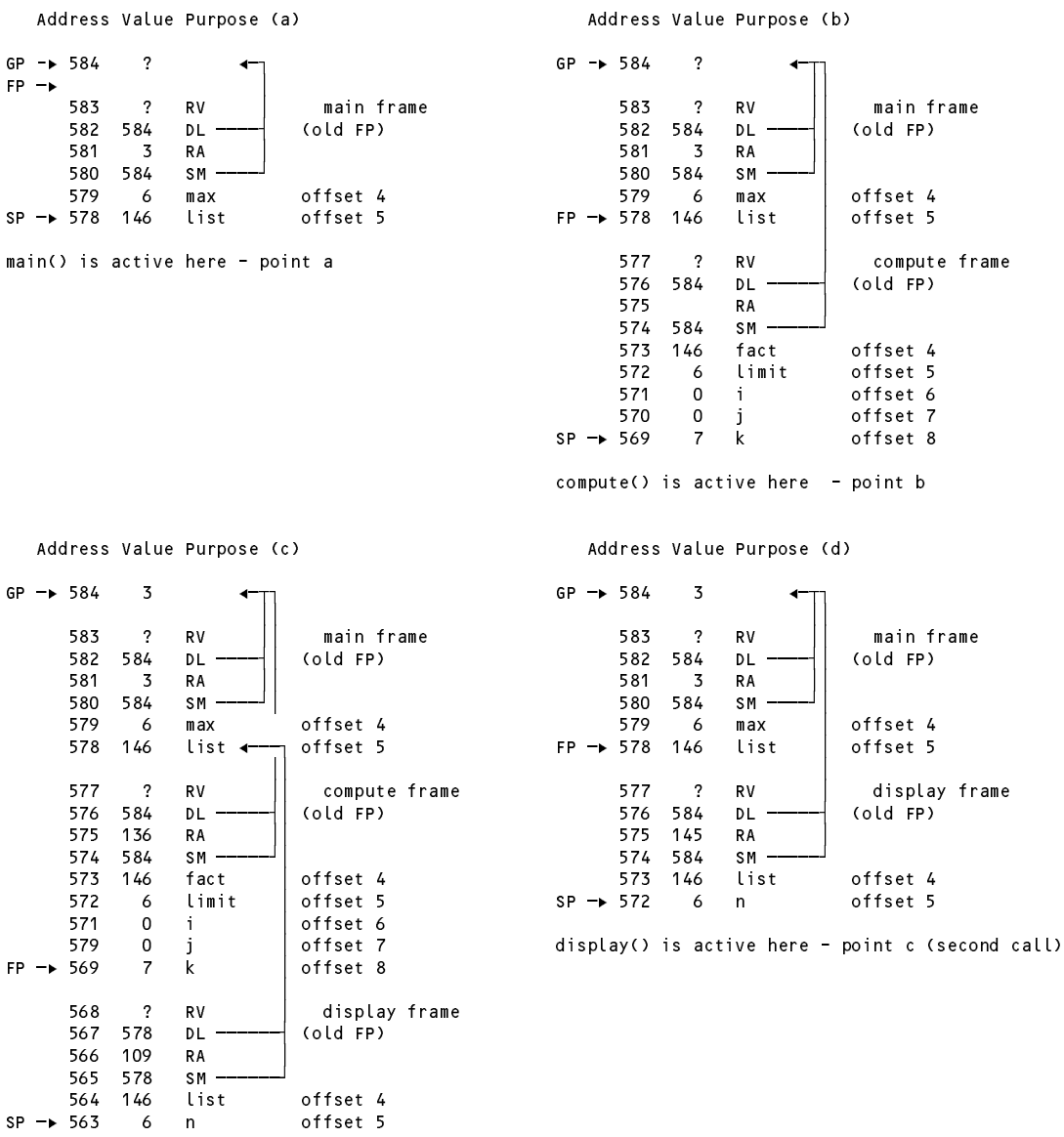


Figure 14.3 Contents of run-time activation records:
(a) after calling main() but before calling compute(list, max);
(b) after calling main() and calling compute(list, max);
(c) after calling main() and calling compute(list, max) and calling display(fact, limit);
(b) after making the call from main to display(list, max)

(It may be useful to remind the reader that the pre-decrement implementation of the run-time stack is the reason

why registers like `cpu.fp` seem to point to locations one higher than one might at first expect.)

14.3.3 Returning from a void function

Once a function has completed execution of the statements in its associated *Body*, it must return control to the function that called it and restore the registers to the values that they held when the function was called. This is accomplished quite easily by:

- restoring the stack pointer `SP` to the value `FP` so as to reclaim the storage used for the activation record;
- restoring the mark stack pointer `MP` to the value previously saved in the stack mark element (`SM`) of the activation record;
- restoring the program counter `PC` to the value previously saved in the return address element (`RA`) of the activation record.
- restoring the frame pointer `FP` to the value previously saved in the dynamic link element (`DL`) of the activation record;

14.3.4 Extending the PVM

The sequence of low-level operations needed to implement calling and returning from a function can, in principle, be associated with either the caller or the called routine. Were we to be generating code for a "real" machine the amount of this code might be considerable. Since a routine is defined once, but possibly called from many places, it is usual to associate most of the actions with the called routine. Once again, given the freedom to extend the instruction set of the PVM we can do so in a way that relegates all the finer details to the emulator itself.

It turns out that we need only three new opcodes to implement void functions. In the notation used previously, the intent of these can be summarized as follows.

`FHDR`	Create standard activation record or frame header
`CALL N`	Call the function whose code starts at instruction `N`
`RETV`	Return from `void` function

and the way in which one would handle these as enhancements to an interpreter like that presented in sections 4.6 and 4.8 should demonstrate their semantics quite clearly.

```
case PVM.fhdr:   // create frame header before evaluating args
  mem[cpu.sp - headerSize] = cpu.mp; // save mp
  cpu.mp = cpu.sp;                   // set mp
  cpu.sp = cpu.sp - headerSize;      // allocate header
  break;
case PVM.call:   // call function
  mem[cpu.mp - 2] = cpu.fp;          // set DL
  mem[cpu.mp - 3] = cpu.pc + 1;      // set RA
  cpu.fp = cpu.mp;                   // set FP
  cpu.pc = mem[cpu.pc];              // transfer to function
  break;
case PVM.retv:   // return from void function
  cpu.sp = cpu.fp;                   // discard frame and RV
  cpu.mp = mem[cpu.fp - 4];          // restore MP
  cpu.pc = mem[cpu.fp - 3];          // ready to return to caller
  cpu.fp = mem[cpu.fp - 2];          // restore FP
  break;
```

14.3.5 Parameter passing

Our discussion so far has made frequent mention of parameters and arguments, without going into detail. This subject is fairly extensive, as the reader may have realized. In the development of programming languages, several models of parameter passing have been proposed and those actually employed vary semantically from language to language, while syntactically often appearing deceptively similar. In most cases, declaration of a subprogram is accompanied by the declaration of a list of **formal parameters**, which appear to have a status within the subprogram rather like that of local variables. Invocation of the subprogram is accompanied by a corresponding list of **actual parameters** (sometimes called **arguments**), and it is invariably the case that the relationship between formal and actual parameters is achieved by positional correspondence rather than by lexical

correspondence in the source text. Thus it would be quite legal, if a little confusing to another reader, to declare

```
void function AnyName ( int A , int B )
```

and then to invoke it with a statement of the form

```
AnyName ( B , A );
```

when the `A` in the function would be associated with the `B` in the calling routine, and the `B` in the function would be associated with the `A` in the calling routine. It may be the lack of name correspondence that is at the root of a great deal of confusion in parameter handling amongst beginners.

The correspondence of formal and actual parameters goes deeper than mere position in a parameter list. Of the various ways in which it might be established, the two most widely used and familiar parameter passing mechanisms are those known as **call-by-reference** and **call-by-value**. In developing the case studies in this text we have, of course, made frequent use of both methods - we turn now to a discussion of how they might be implemented.

The semantics and the implementation of the two mechanisms are quite different:

- In call-by-reference an actual parameter usually takes the form of a *VariableDesignator*. Within the subprogram, a reference to the formal parameter results, at run-time, in a direct reference to the variable designated by the actual parameter, and any change to that formal parameter results in an immediate change to the corresponding actual parameter. In a very real sense, a formal parameter name can be regarded as an *alias* for the actual parameter name. The alias lasts as long as the subprogram is active and may be transmitted to other subprograms with parameters passed in the same way. Call-by-reference is usually accomplished by passing the address associated with the actual parameter to the subprogram for processing.

- In call-by-value, an actual parameter takes the form of an *Expression*. Formal parameters in a subprogram (when declared in this way) are effectively variables local to that subprogram which start their lives initialized to the values of the corresponding actual parameter expressions. However, any changes made to the values of the formal parameter variables are confined to the subprogram and cannot be transmitted back via the formal parameters to the calling routine. Fairly obviously, it is the run-time value of the expression which is handed over to the subprogram for processing, rather than an explicit address of a variable.

Call-by-value is preferred for many applications - for example, it is useful to be able to pass expressions to routines like `write` without having to store their values in otherwise redundant variables. However, in some languages, if a structure like an array is to be passed by value, a complete copy of the array must be passed to the subprogram. This is expensive, both in terms of space and time, and thus many programmers pass all array parameters by reference, even if there is no need for the contents of the array to be modified.

Some languages derived from C, including Java and Parva, support call-by-value only, which at first might appear to be very restrictive. However, since a parameter may be of a reference type, this means that one can nearly always achieve the effect of call-by-reference by passing a reference to a variable (using the call-by-value mechanism to do so) and then allowing the called function to access and even modify the variable to which this reference relates. An example of this is to be found in the sample program given earlier, where the `compute` function is passed a reference to the array `list` declared and created locally to the `main` function, and then proceeds to store values in the elements of the array - without, of course, changing the reference itself. Critics of languages like Java are quick to point out that there is no simple way to write some functions in which the use of true call-by-reference would be appropriate. For example, the Java method given below does not achieve what its author probably intended.

```
static void exchange(int x, int y) {
// Attempts to exchange values of x and y
  int z = y; y = x; x = z;
} // exchange
```

Languages that support true call-by-reference mechanisms usually do so by marking the formal parameters in some way when the subprogram is declared. For example, in C# the desired effect of the method just illustrated would be obtained by coding it

```
static void exchange(ref int x, ref int y) {
  int z = y; y = x; x = z;
} // exchange
```

while the corresponding headers in C++ and Modula-2 would be of the forms

```
void exchange(int& x, int& y)
PROCEDURE exchange(VAR x, y : INTEGER)
```

respectively. In passing, we may comment that these notational devices are open to the criticism that the accidental omission of the ampersand (&) or keyword `VAR` can lead to hard-to-find bugs. C# is exempt from this criticism - `ref` and `out` parameters have to be explicitly marked as such both when the method is declared and when it is called.

Some languages permit call-by-reference to take place with actual parameters that are expressions in the general sense; in this case the value of the expression is stored in a temporary variable and the address of that variable is passed to the subprogram.

Although our discussion has focussed on a particular implementation strategy, we should point out that there are other ways in which actual parameter values may be transmitted to a subprogram. Most of these involve pushing them onto a stack as part of the activation sequence that is executed before transferring control to the function which is to use them. Similarly, to allow a function value to be returned it is convenient to reserve a stack item for this just before the arguments are set up, and for the function subprogram to access this reserved location using a suitable offset. The arguments do not have to be stored *below* the frame header - that is, within the activation record - as we have suggested. Some implementations favour storing them *before* (above) the frame header. We shall discuss this latter possibility no further here but leave the details as an exercise for the curious reader (see Terry (1986), Wirth (1996), Louden (1997) or Grune *et al.* (2000)).

14.4 A complete compiler for a set of void functions

We are now in a position to understand the way in which our Cocol specification of Parva must be attributed in order to develop a compiler on the lines suggested by the preceding discussion. As usual, the full grammar can be found in the *Resource Kit* and we shall be content here to study only some extracts, focusing first on those productions concerned with handling declarations.

14.4.1 Declarations and constraint analysis

The set of declarations must include one for a void function with the distinctive name `main`. Since our system still enforces "declare before use" semantics, this function will, naturally, be the last to be declared, but the parser can be written more generally with an eye to removing this restriction later on.

The parser for recognizing a function must create an appropriate symbol table entry for this, and then open a new scope to cover the declaration of any formal parameters and local variables. Offsets for these will be computed as the declarations are encountered, and the productions are required, as before, to keep track of the overall size of the activation record that will be needed if and when the function is activated.

```
VoidFuncDeclaration                  (. StackFrame frame = new StackFrame();
                                        Entry function = new Entry(); .)
=  "void" Ident<out function.name>   (. function.kind = Kinds.Fun;
                                        function.type = Types.voidType;
                                        function.nParams = 0;
                                        function.firstParam = null;
                                        function.entryPoint = new Label(known);
                                        Table.Insert(function);
                                        Table.OpenScope(); .)
   "(" FormalParameters<function> ")" (. frame.size = CodeGen.headerSize + function.nParams; .)
   Body<frame>                       (. Table.CloseScope(); .) .
```

Parsing a formal parameter list is straightforward. Note that a reference to the first formal parameter is maintained in the entry for the function itself, as discussed in section 14.2.1.

```
FormalParameters<Entry func>           (. Entry param; .)
= [ OneParam<out param, func>          (. func.firstParam = param; .)
    { WEAK "," OneParam<out param, func> }
  ] .

OneParam<out Entry param, Entry func>  (. param = new Entry();
                                          param.kind = Kinds.Var;
                                          param.offset = CodeGen.headerSize + func.nParams;
                                          func.nParams++; .)
=  Type<out param.type>
   Ident<out param.name>               (. Table.Insert(param); .) .
```

14.4.2 Code generation

For our compiler it is straightforward to generate the code responsible for handling void function calls with parameter passing by value.

Firstly, consider the implicit code that must be added to ensure that the program as a whole can commence execution. It is usual to regard the `main` function as being called from within an operating system. Since the `main` function is syntactically similar to any other void function, it makes good sense at the start of compilation to set up a normal frame header followed by a normal call to this function, arranging that when it returns the program as a whole can terminate. Of course, at the start of parsing, the entry point of the `main` function is not known - another forward reference situation that should hold little terror for the reader. It is convenient to introduce a globally accessible field into our parser:

```
static Label mainEntryPoint = new Label(!known);  // entry point of main function
```

Once all functions have been parsed a check can be made that this entry point has been defined (and the call backpatched):

```
Parva                                  (. CodeGen.FrameHeader();          // no arguments
                                          CodeGen.Call(mainEntryPoint);   // incomplete
                                          CodeGen.LeaveProgram(); .)      // return to OS
= { VoidFuncDeclaration } EOF          (. if (!mainEntryPoint.IsDefined())
                                            SemError("missing Main function"); .) .
```

The *VoidFuncDeclaration* parser has to look out for the main method:

```
VoidFuncDeclaration                    (. StackFrame frame = new StackFrame();
                                          ....
                                          Table.OpenScope(); .)
   "(" FormalParameters<function> ")"  (. frame.size = CodeGen.headerSize + function.nParams;
                                          if (function.name.Equals("main")
                                              && !mainEntryPoint.IsDefined()
                                              && function.nParams == 0) {
                                            mainEntryPoint.Here();
                                          } .)
   Body<frame>                         (. Table.CloseScope(); .) .
```

The body of a function is handled in much the same way as in the last chapter, with a few subtle changes. We are still required to generate code that will reserve space for any local variables that have been declared, but this must take into account that, when this occurs, storage for the frame header and the arguments will already have been allocated by the activation sequence. At the end of parsing a void function we can arrange to generate appropriate code for leaving that function (this does not preclude the appearance of any other *ReturnStatement* within that function).

```
Body<StackFrame frame>                 (. Label DSPLabel = new Label(known);
                                          int sizeMark = frame.size;
                                          CodeGen.OpenStackFrame(); .)
=  "{" { Statement<frame> } WEAK "}"   (. CodeGen.FixDSP(DSPLabel.Address(),
                                                         frame.size - sizeMark);
                                          CodeGen.LeaveVoidFunction(); .)
```

Assignment statements and void function calls both start with an identifier. As mentioned in section 14.2, this LL(1) violation in the grammar is best handled by a simple lookahead, using the facility provided in the latest versions of Cocol of using a *conflict resolver*. At the point where an identifier has signalled the need to distinguish the statement kinds, the `IsCall()` function uses the value of the lookahead token `la.val` to perform a search of the symbol table and returns *true* if it corresponds to a function name. Note that an object of the `DesType` class will also be returned; if the identifier corresponds to a function this object will suffice to establish

the call to it but, if not, the variable designator must be constructed from the call to the *Designator* parser.

```
static bool IsCall(out DesType des) {
  Entry entry = Table.Find(la.val);
  des = new DesType(entry);
  return entry.kind == Kinds.Fun;
} // IsCall
```

The *AssignmentOrCall* parser then follows as:

```
AssignmentOrCall                          (. int expType;
                                             DesType des; .)
=  (  IF (IsCall(out des))                   /* use resolver to handle LL(1) conflict */
       identifier                         (. CodeGen.FrameHeader(); .)
       "(" Arguments<des> ")"             (. CodeGen.Call(des.entry.entryPoint); .)
    |  Designator<out des>                (. if (des.entry.kind != Kinds.Var)
                                               SemError("cannot assign to "
                                                        + Kinds.kindNames[des.entry.kind]); .)
       AssignOp Expression<out expType>   (. if (!Compatible(des.type, expType))
                                               SemError("incompatible types in assignment");
                                             CodeGen.Assign(des.type); .)
   ) WEAK ";" .
```

Notice how the code for allocating a frame header is generated before generating the code for processing the argument list and that this is followed by generating code for the call itself.

Processing the actual argument list calls for care in writing the attributed grammar. Not only is it necessary to check that the number of arguments is the same as the number of formal parameters, the check for type compatibility requires a scan of the linked list of these formal parameters. If an error has been made by the programmer this list may end prematurely, which is why the scanning algorithm takes the precaution of avoiding `null` references.

```
Arguments<DesType des>                    (. int argCount = 0;
                                             Entry fp = des.entry.firstParam; .)
=  [ OneArg<fp>                           (. argCount++;
                                             if (fp != null) fp = fp.nextInScope; .)
      { WEAK "," OneArg<fp>               (. argCount++;
                                             if (fp != null) fp = fp.nextInScope;.)
      }
   ]                                      (. if (argCount != des.entry.nParams)
                                               SemError("wrong number of arguments"); .) .

OneArg<Entry fp>                          (. int argType; .)
=  Expression<out argType>                (. if (fp != null && !Compatible(argType, fp.type))
                                               SemError("argument type mismatch"); .) .
```

14.5 Globally accessible constants and variables

In the compiler as so far developed, the names of all functions are placed in the lowest (global) scope. It is sometimes convenient to have variables and constants declared at this level as well, as exemplified by the following variation on our sample program:

```
const LIMIT = 12;           // constant with global scope
int[] list;                 // variables with global scope and existence
int limit;

void display(int n) {
  while (n >= 0) {
    write(n, list[n], "\n");
    n = n - 1;
  }
} // display

void compute() {
  int k = 1;
  list[0] = 1;
  while (k <= limit) {
    list[k] = list[k-1] * k;
    k = k + 1;
  }
  display(limit);
} // compute
```

```
void main() {
  read("Supply limit ", limit);
  if (limit > LIMIT) limit = LIMIT;
  list = new int[limit + 1];
  compute();
  display(limit);
} // main
```

14.5.1 Syntactic extensions to Parva to support global constants and variables

The syntactic extensions we need in Parva are almost trivial (they get more complicated when one wishes to introduce value-returning functions, as we shall do later). Devoid of attributes we might write:

```
PRODUCTIONS /* some omitted to save space */
  Parva                 = {   GlobalVarDeclarations
                            | ConstDeclarations
                            | VoidFuncDeclaration } EOF .
  GlobalVarDeclarations = Type identifier { "," identifier } ";" .
```

14.5.2 Symbol table considerations

Global variables have a scope that extends over all of the source text after the point of declaration; similarly at run-time they have an extent or lifetime for the entire time of execution. In keeping with our earlier practice, we shall reserve space for these variables in a small stack frame at the top end of memory. At run-time the base of this frame will be addressed via the global pointer register GP; the base of the stack frame of the latest function to have been activated and called will, as before, be addressed via the frame pointer register FP. Given that "local" and "global" variables have to be addressed in different ways, it follows that some record must be kept of the *level* at which every identifier is declared. In Parva as so far developed there are only two levels, but were we to extend the system to incorporated nested functions we should need further levels. Looking ahead in this way, the suggested addition to the symbol table *Entry* class is

```
class Entry {

  public const int
    global = 0,                          // declaration levels
    local  = 1;
  ...                                    // rest as before

} // end Entry
```

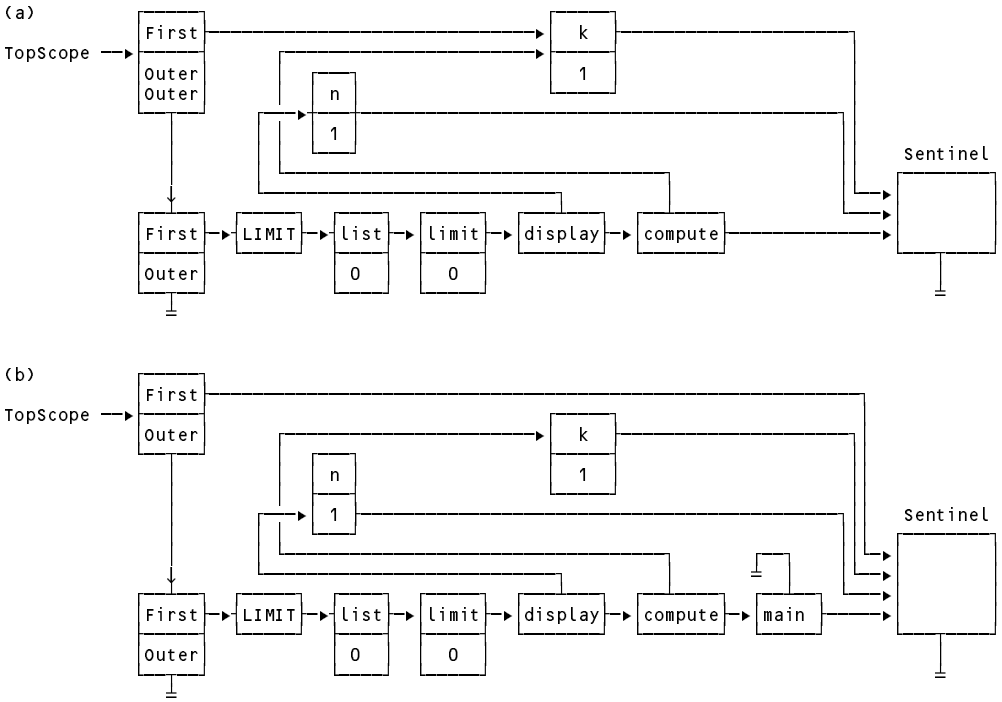


Figure 14.4 A symbol table structure with global constants and variables and links from function entries to local parameter and variable entries: (a) at the stage of parsing the block for function compute(); (b) at the stage of parsing the block for function main();

For the sample program given earlier the symbol table at various stage of the parsing process can be seen in Figure 14.4, where the levels have been indicated in each variable node. All nodes have a level field, of course, but at present there is little need for this in the entries for constants and functions.

14.5.3 Run-time storage management

Figure 14.5 displays part of the history of the run-time frames as the program given earlier is executed. The reader is urged to study this in detail.

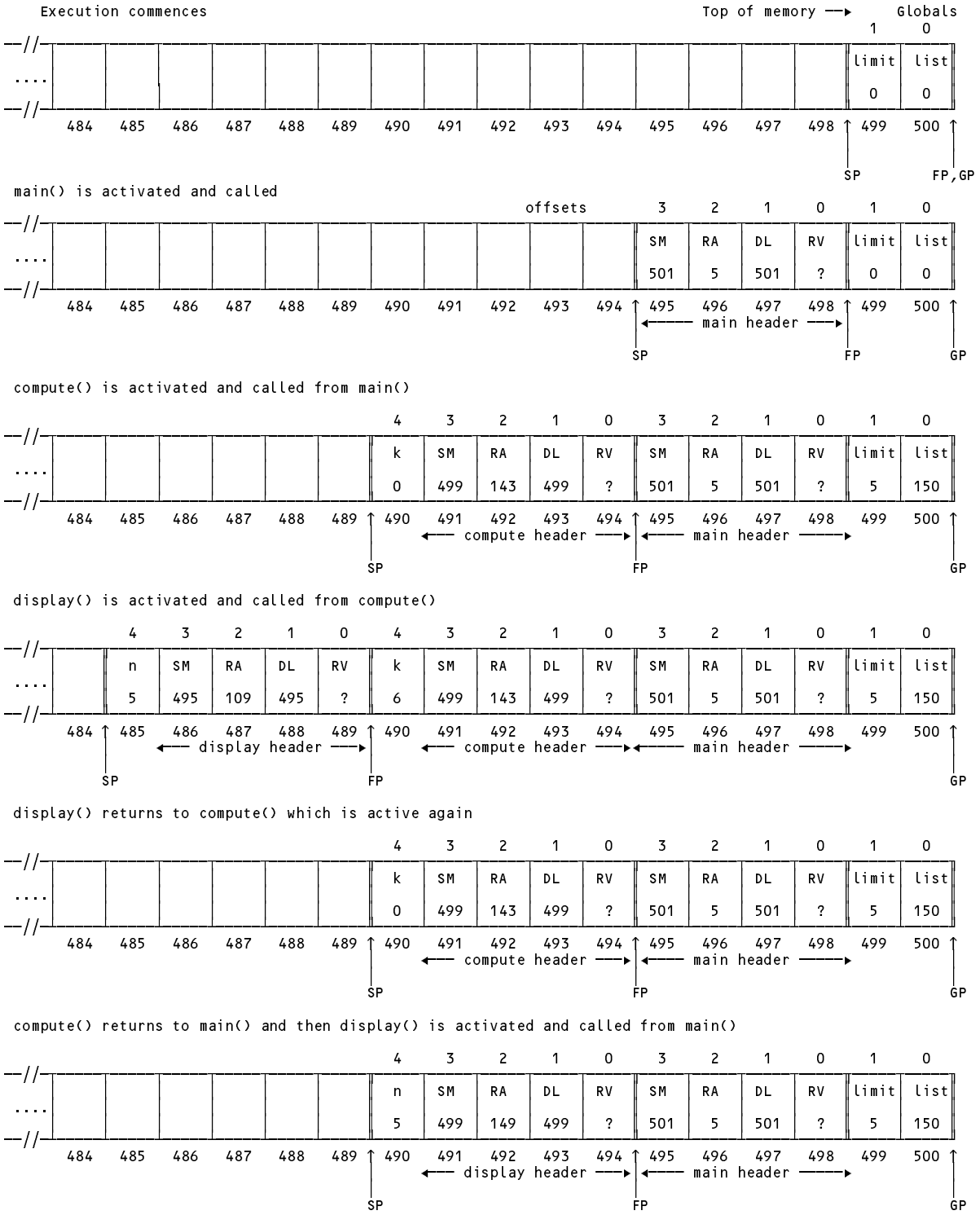


Figure 14.5 Activation record activity on the PVM for our sample program.

14.5.4 Putting it all together

Little is needed to extend the front end of the compiler to handle our extensions. The main driving routine has to incorporate code for reserving the level 0 stack frame using the familiar `DSP` operation, and the compiler is responsible for generating this operation incompletely and then back patching it when the number of global variables (if any) is known.

The reader should note a quirk of the design - we have allowed local variables to be initialized at the point of declaration, but have not done the same for global variables. Why do you suppose this has been done, and is it possible to remove this anomaly easily? If not, why not?

```
Parva                                    (. StackFrame frame = new StackFrame();
                                            Label DSPLabel = new Label(known);
                                            CodeGen.OpenStackFrame(0);
                                            CodeGen.FrameHeader();         // no arguments
                                            CodeGen.Call(mainEntryPoint);  // forward, incomplete
                                            CodeGen.LeaveProgram(); .)     // return to O/S
= {   GlobalVarDeclarations<frame>
    | ConstDeclarations
    | VoidFuncDeclaration } EOF           (. CodeGen.FixDSP(DSPLabel.Address(), frame.size);
                                            if (!mainEntryPoint.IsDefined())
                                              SemError("missing Main function"); .) .
```

The routines that deal with the creation of symbol table entries for variables must assign the appropriate value to the `level` field.

```
GlobalVarDeclarations<StackFrame frame>  (. int type; .)
=  Type<out type>
   OneGlobal<frame, type>
   { WEAK "," OneGlobal<frame, type> }
   WEAK ";" .

OneGlobal<StackFrame frame, int type>    (. Entry var = new Entry(); .)
=  Ident<out var.name>                   (. var.kind = Kinds.Var;
                                            var.type = type;
                                            var.level = Entry.global;
                                            var.offset = frame.size;
                                            frame.size++;
                                            Table.Insert(var); .) .

OneVar<StackFrame frame, int type>       (. int expType;
                                            Entry var = new Entry(); .)
=  Ident<out var.name>                   (. var.kind = Kinds.Var;
                                            var.type = type;
                                            var.level = Entry.local;
                                            var.offset = frame.size;
                                            frame.size++; .)
   [ AssignOp                            (. CodeGen.LoadAddress(var); .)
     Expression<out expType>             (. if (!Assignable(var.type, expType))
                                              SemError("incompatible types in assignment");
                                            CodeGen.Assign(var.type); .)
   ]                                     (. Table.Insert(var); .) .
```

All that is needed in the way of alterations to the code generator is to incorporate a new version of the `LoadAddress` method:

```
public static void LoadAddress(Entry var) {
// Generates code to push address of variable var onto evaluation stack
  if (var.level == Entry.global) Emit(PVM.ldga); else Emit(PVM.lda);
  Emit(var.offset);
} // CodeGen.LoadAddress
```

while the matching extension to the opcode despatcher in the PVM is equally straightforward:

```
case PVM.lda:           // push local address
  adr = cpu.fp - 1 - Next();
  if (InBounds(adr)) Push(adr);
  break;
case PVM.ldga:        // push global address
  adr = cpu.gp - 1 - Next();
  if (InBounds(adr)) Push(adr);
  break;
```

14.6 Value-returning (non-void) functions

For our next refinement we consider the addition of non-void value-returning functions, often called *typed functions*, to our Parva language and implementation . This concept is surely familiar to all readers by now, and we need not give a complete exposition, but leave readers to check the finer points for themselves. As usual, we begin with an example to illustrate the features we need.

Function `nfact` demonstrates a traditional (if somewhat inefficient) way to compute the factorial function in a recursive way.

```
const M1 = 123;

int nfact(int n) {
  if (n <= 1) return 1;
  return n * nfact(n-1);
} // nfact

void main() {
  int m = M1;
  m = 5 * nfact(2);
  write(m);
} // main
```

This function has a return type (`int`), takes a single parameter (`n`) and derives from this the integer value that is to be returned.

A value-returning function like `nfact` is called from within the context of a *Factor* that forms part of an *Expression*, while the value to be returned is computed by the associated *Expression* attached to the *ReturnStatement* within the function. There may (as here) be more than one *ReturnStatement* in a function, although only one will be executed on any given call of that function.

14.6.1 Syntactic extensions to Parva to support non-void functions

The syntactic extensions we need in Parva seem almost trivial. Devoid of attributes we might start to write:

```
PRODUCTIONS /* some omitted to save space */
  Parva                = {   ConstDeclarations
                           | GlobalVarDeclarations
                           | FuncDeclaration } .
  FuncDeclaration      = ( "void" | Type ) identifier
                         "(" FormalParameters ")" Body .
  FormalParameters     = [ OneParam { "," OneParam } ] .
  GlobalVarDeclarations = Type identifier { "," identifier } ";" .
  ConstDeclarations    = "const" OneConst { "," OneConst } ";" .
  OneConst             = identifier "=" Constant .
  Statement            = Assignment | VoidFunctionCall | ...
  Assignment           = Designator "=" Expression .
  VoidFunctionCall     = FunctionCall ";" .
  FunctionCall         = identifier "(" Arguments ")" .
  Designator           = identifier [ "[" Expression "]" ] .
  Factor               = Designator | TypedFunctionCall | ...
  TypedFunctionCall    = FunctionCall ";" .
  ReturnStatement      = "return" [ Expression ] ";" .
```

This raises some interesting challenges for the eould-be LL(1) grammar writer. We have seen the conflict between *Assignment* and *VoidFunctionCall* previously and shown how it can be handled with a conflict resolver. The same approach can be used to resolve the syntactic conflict between a *Designator* and a *TypedFunctionCall* in the context of a *Factor*. A further LL(1) conflict occurs between *GlobalVarDeclarations* and *FuncDeclarations* as each of these must start with an example of a *Type*. Once again, this could be handled with a conflict resolver, although it is easily handled by refactoring the grammar, which is what we illustrate below, leaving the use of a conflict resolver as an interesting exercise for the reader. With these points in mind, revised productions are:

```
PRODUCTIONS /* some omitted to save space */
  Parva                       = {   ConstDeclarations
                                  | FuncOrGlobalVarDeclarations ).
  FuncOrGlobalVarDeclarations = ( "void" | Type ) identifier
                                ( GlobalVars | Function ) .
  Function                    = "(" FormalParameters ")" Body .
  GlobalVars                  = { "," identifier } ";" .
  Statement                   = ( IF(IsCall()) FunctionCall | Assignment ) | ...
  Factor                      = ( IF(IsCall()) FunctionCall | Designator ) | ...
  ReturnStatement             = "return" [ Expression ] ";" .
```

14.6.2 Semantic complications introduced by non-void functions

Although the syntactic LL(1) conflicts just mentioned are easily handled with conflict resolvers, we shall have to go further, because only void functions can be called as stand-alone statements, and only non-void functions can be called as factors.

To this end, it is expedient to extend the notion of type to include `void` as an apparent type - this will allow various other productions to ensure that void and non-void functions are not misused:

```
Type<out int type>                        (. type = Types.noType; .)
=    "void"                               (. type = Types.voidType; .)
   | BasicType<out type>
     [ "[]"                               (. type++; .)
     ] .
```

The fully attributed part of the compiler that processes global declarations now takes the form:

```
Parva                                     (. StackFrame frame = new StackFrame();
                                             Label DSPLabel = new Label(known);
                                             CodeGen.OpenStackFrame(0);
                                             CodeGen.FrameHeader();         // no arguments
                                             CodeGen.Call(mainEntryPoint);  // forward, incomplete
                                             CodeGen.LeaveProgram(); .)     // return to O/S
= {   FuncOrGlobalVarDeclarations<frame>
    | ConstDeclarations } EOF             (. CodeGen.FixDSP(DSPLabel.Address(), frame.size);
                                             if (!mainEntryPoint.IsDefined())
                                               SemError("missing Main function"); .) .

FuncOrGlobalVarDeclarations<StackFrame frame>
                                          (. int type;
                                             string name; .)
=  Type<out type>
   Ident<out name>
   (   GlobalVars<type, name, frame>
     | Function<type, name>
   ) .

GlobalVars<int type, string name, StackFrame frame>
=                                         (. if (type == Types.voidType)
                                               SemError("variables may not be of void type");
                                             EnterGlobalVar(type, name, frame); .)
   { WEAK "," Ident<out name>             (. EnterGlobalVar(type, name, frame); .)
   } WEAK ";" .
```

It is convenient to introduce a further static void method into the parser to deal with the insertion of global variable entries into the symbol table:

```
static void EnterGlobalVar(int type, string name, StackFrame frame) {
  Entry var  = new Entry();
  var.name   = name;
  var.kind   = Kinds.Var;
  var.level  = Entry.global;
  var.type   = type;
  var.offset = frame.size;
  Table.Insert(var);
  frame.size++;
} // EnterGlobalVar
```

The parser for compiling a *Factor* follows as:

```
Factor<out int type>                        (. type = Types.noType;
                                               int size;
                                               DesType des;
                                               ConstRec con; .)
=  (  IF (IsCall(out des))                     // /* use resolver to handle LL(1) conflict */
       identifier                           (. if (des.type == Types.voidType)
                                                 SemError("void function call not allowed here");
                                               CodeGen.FrameHeader(); .)
       "(" Arguments<des> ")"               (. CodeGen.Call(des.entry.entryPoint); .)
    |  Designator<out des>                  (. switch (des.entry.kind) {
                                                 case Kinds.Var:
                                                   CodeGen.Dereference();
                                                   break;
                                                 case Kinds.Con:
                                                   CodeGen.LoadConstant(des.entry.value);
                                                   break;
                                                 default:
                                                   SemError("wrong kind of identifier");
                                                   break;
                                               } .)
   )                                        (. type = des.type; .)
   | Constant<out con>                      (. type = con.type;
                                               CodeGen.LoadConstant(con.value); .)
   | "new" BasicType<out type>              (. type++; .)
     "[" Expression<out size>               (. if (!IsArith(size))
                                                 SemError("array size must be integer");
                                               CodeGen.Allocate(); .)
     "]"
   | "!" Factor<out type>                   (. if (!IsBool(type)) SemError("boolean operand needed");
                                               else CodeGen.NegateBoolean();
                                               type = Entry.boolType; .)
   | "(" Expression<out type> ")" .
```

There is a further syntactic difficulty. It is tradtional in modern languages to use a *Return* statement to indicate that control should be passed back to the caller of the function. But the *Expression* that appears in this production is "context sensitive", not optional. In the context of a void function an explicit *ReturnStatement* is optional, but if it appears it may not be associated with an *Expression*, while in the context of a non-void function the *Expression* becomes mandatory and must yield a value compatible with the type of the function. In this case explicit *ReturnStatements* must not only be present, the flow of execution must be such that one of them must be executed in order to return to the caller. Of course, some of these complications could be better handled were there to be two distinct keywords, but that option is not open to us (except as an exercise).

All is not lost: the semantics can be handled by parameterizing the *ReturnStatement* parser as shown here. To do so means that the type of the enveloping function (once known) must be passed down as an argument to several other parser routines responsible for parsing the body of the function. All these details can be left as a useful exercise for the reader.

```
ReturnStatement<int funcType>               (. int expType; .)
=  "return"
   (                                        (. if (funcType == Types.voidType)
                                                 SemError("a void function cannot return a value");
                                               CodeGen.LoadReturnAddress(); .)
       Expression<out expType>              (. if (!Compatible(funcType, expType))
                                                  SemError("return type must match expression type");
                                                CodeGen.Assign(expType); .)
     |                                      (. if (funcType !== Types.voidType)
                                                 SemError("return expression needed"); .)
   )                                        (. CodeGenLeaveFunction(funcType); .)
   WEAK ";" .
```

This production will handle an explicit *ReturnStatement*. Unfortunately it will not handle the explicit return that appears at the end of each void function and, worse still, it will not handle the more awkward case of a badly constructed non-void function in which it is possible to reach the "end" of the function and never encounter a *ReturnStatement*, as illustrated in the following poor code:

```
bool isOdd(int n) {
  if (n/2 * 2 != n) return true;
} // isOdd
```

Probably the best that one can do with our simple compiler is to arrange that at the end of parsing the *Block* of a non-void function a trap is set that will generate a run-time error if it is sprung, while at the end of a void function an appropriate return is generated:

```
Body<StackFrame frame, int funcType>        (. Label DSPLabel = new Label(known);
                                               int sizeMark = frame.size;
                                               CodeGen.OpenStackFrame(0); .)
=  "{" { Statement<frame, funcType> }
   WEAK "}"                                 (. CodeGen.FixDSP(DSPLabel.Address(), frame.size - sizeMark);
                                               if (funcType == Types.voidType)
                                                 CodeGen.LeaveVoidFunction();
                                               else CodeGen.FunctionTrap(); .) .
```

14.6.3 Code generation and execution for non-void functions

When returning from a non-void function, restoring the stack pointer is done in a way that ensures that the return value then appears on the top of the working stack, where it naturally forms an operand for further manipulation in the expression of which it was a component *Factor*.

```
case PVM.ret:         // return from non-void function
  cpu.sp = cpu.fp - 1;
  cpu.mp = mem[cpu.fp - 4];
  cpu.pc = mem[cpu.fp - 3];
  cpu.fp = mem[cpu.fp - 2];
  break;
case PVM.trap:        // trap falling off end of function
  ps = badFun;
  break;
```

Code generation for these new opcodes is trivial:

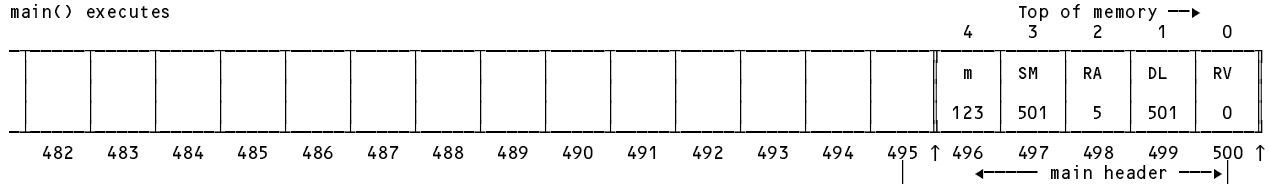
```
public static void LeaveFunction(int funcType) {
// Generates code to leave a function
  if (funcType == Types.voidType) Emit(PVM.retv); else Emit(PVM.ret);
} // CodeGen.LeaveFunction

public static void FunctionTrap() {
// Generates code at end of non-void function to trap failure to return
  Emit(PVM.trap);
} // CodeGen.FunctionTrap
```

The code generated by our compiler for our sample program is as follows:

```
{ 0 } DSP    0   no global variables            {33 } SUB          n-1
{ 2 } FHDR       activate main()                {34 } CALL   6     call nfact(n-1)
{ 3 } CALL   40  call main()                    {36 } MUL          n * (n-1)!
{ 5 } HALT       exit program                   {37 } STO            RV = n * (n-1)!
{ 6 } DSP    0   int nfact() { // nfact entry   {38 } RET          return n * (n-1)!
{ 8 } LDA    4                                  {39 } TRAP       } trap bad function
{10 } LDV                                       {40 } DSP    1   void main() { // main entry
{11 } LDC    1                                  {42 } LDA    4     adr(m)
{13 } CLE                                       {44 } LDC    123
{14 } BZE    22    if (n <= 1)                  {46 } STO          m = 123
{16 } LDA    0       //  adr(RV)               {47 } LDA    4     adr(m)
{18 } LDC    1                                  {49 } LDC    5     5
{20 } STO            RV = 1                     {51 } FHDR         activate nfact
{21 } RET          return 1                     {52 } LDC    2
{22 } LDA    0       // adr(RV)                 {54 } CALL   6     call nfact(2)
{24 } LDA    4                                  {56 } MUL
{26 } LDV          value n                      {57 } STO          m = 5 * nfact(2)
{27 } FHDR         activate nfact               {58 } LDA    4
{28 } LDA    4     param = n - 1                {60 } LDV
{30 } LDV                                       {61 } PRNI         write(m)
{31 } LDC    1                                  {62 } RETV       } return (to exit program)
```

14.6.5 Run time behaviour for a simple recursive program

It may help the reader assimilate all this discussion by a careful study of the following trace of our sample program.



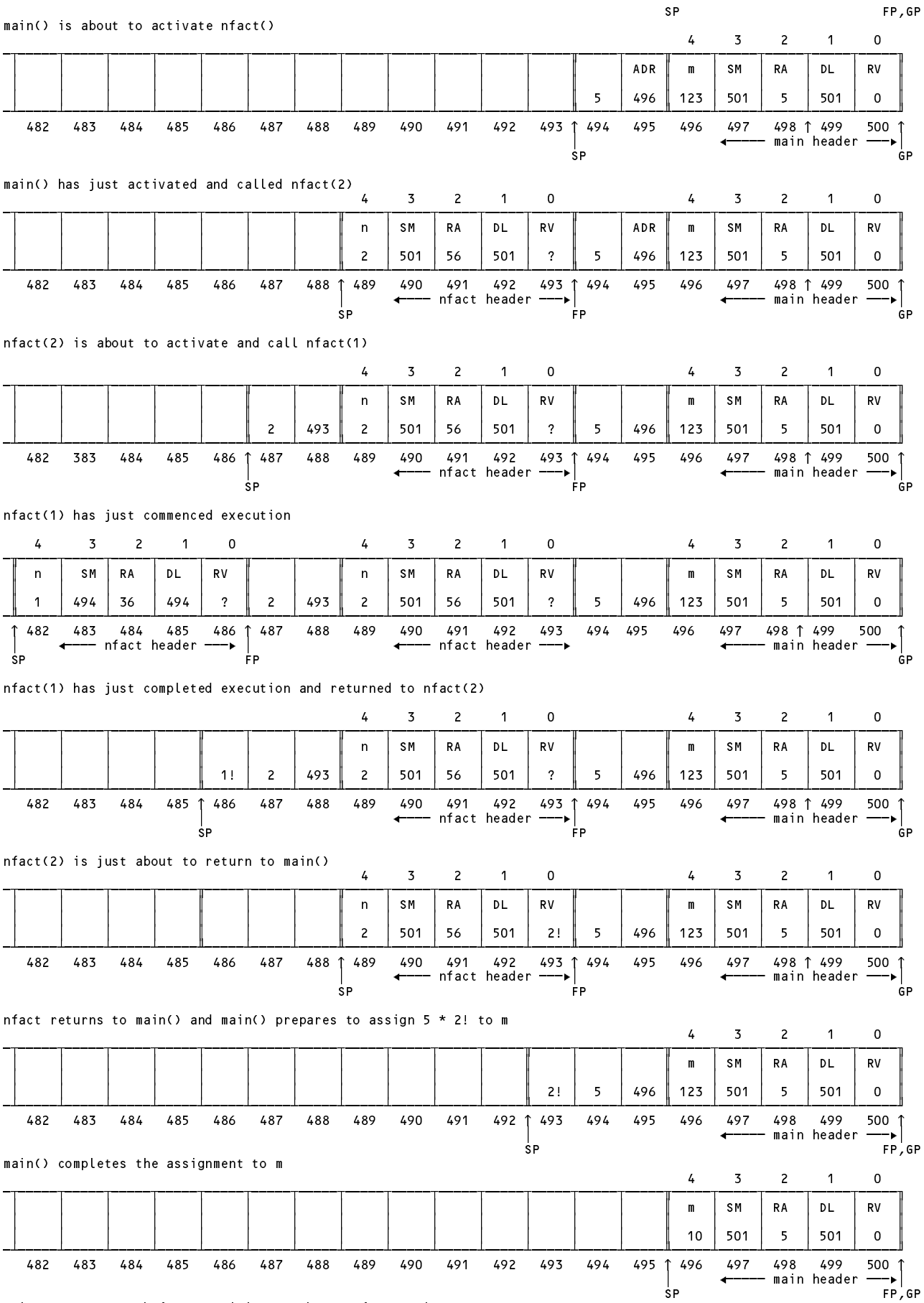


Figure 14.6 Stack frame activity on the PVM for sample program.

14.7 Mutually recursive functions

Earlier in this chapter we remarked that the limitations of our simple incremental compiling technique would become more apparent as the development of the compiler proceeded. A relatively simple change to the definition of Parva to insist that global variables and constants could only be declared ahead of function declarations would help the compiler writer, of course, but imposing artificial constraints on the order of declarations is somewhat annoying. Programmers schooled on languages that impose a policy of "declare before use" - which is really for the benefit of the compiler writer - learn to accept this as natural after a while, but there are situations where it becomes impossible to write code that conforms to this requirement. The classic example is provided by situations in which functions or methods are mutually recursive - function `one` wishes to call function `two`, which in turn must be able to call function `one` recursively.

Handling mutually recursive functions is not particularly difficult if the compiler first builds an AST for the whole program. Rather than pursuing this approach it may be of interest to examine an alternative strategy that is often adopted. It relies on a syntactic device, the so-called **forward declaration** of a function. The programmer is required to provide just enough information about the function so that it can be referenced even before it is fully defined. The device will be familiar to C, C++, Modula-2 and Pascal programmers, as all these languages provide for it. In C a forward declaration is known as a **function prototype**, and the idea may be illustrated by a variation on the simple program of section 14.3.1.

```
void one();       // function prototype
void two(int x);  // function prototype

void main() {
  one();
  two(3);
} // main

void one() {
  two(1);
} // one

void two(int x) {
  if (x > 1) two(x-1);  // recursive call
} // two
```

The function prototypes declared ahead of function `main` can lead to the construction of symbol table entries that provide enough information for a single-pass compiler to generate code for evaluating and checking arguments and generating calls to these functions in all respects other than supplying the address needed in the actual call - a minor complication that can be handled in the same way as for the forward branches needed in the compilation of an *IfStatement* or *WhileStatement*.

The syntactic change we would make to the Parva grammar to allow for forward declarations of this sort is almost trivially simple.

```
FuncOrGlobalVarDeclarations = Type identifier
                              ( Function | GlobalVars ) .
Function                    = "(" FormalParameters ")" ( Body | ";" ) .
FormalParameters            = [ OneParam { "," OneParam } ] .
```

The implementation of this extension will provide the reader with a useful exercise, but we can reveal the general idea here. The *Entry* class is extended to include a Boolean field `defined` that can be used to signal whether the declaration of a function has incorporated the associated *Body* or has stopped short at the prototype stage. The *Function* parser, as before, prepares a symbol table entry in much the same way as was illustrated in section 14.4.1, with the exception that it does not immediately define the `entryPoint` field. Before inserting the entry into the symbol table a search is made to see whether an `oldEntry` for a function of the same name exists. If this is the case, a check of the `oldEntry.defined` field will highlight any invalid redeclaration, while if no such entry exists the new entry can be added to the symbol table. Analysis of the formal parameter list can take place as before. After this, if no *Body* is detected the `entry.defined` field can be set to `false`, in anticipation of the check needed when the later complete declaration is made. However, if the *Body* is present, it is necessary to update any extant entry to mark it as now fully defined and to define the `entryPoint` (which for our system will have the convenient side effect of backpatching any partially generated calls to the function). Finally, if it is discovered that an earlier entry had been created then it is also necessary to check that the formal parameter list given previously matches that of the later complete declaration, in respect of there being the same number of parameters in each list and having the same types for corresponding parameters.

Appendix B:

Library routines

The following C# classes provide library facilities used in the case studies in this text. They may also be of more general usefulness. Full source is available in the *Resource Kit* for equivalent C# and Java implementations.

```
public class IntSet {
// Simple set class
// P.D. Terry (p.terry@ru.ac.za)

  public IntSet()
  // Empty set constructor

  public IntSet(params int[] members)
  // Variable args constructor  - eg IntSet(a)  IntSet(a,b) etc

  public IntSet Copy()
  // Value copy

  public bool Equals(IntSet that)
    // Value comparison - true if this and that sets contain same elements

  public void Incl(int i)
  // Includes i in this set

  public void Excl(int i)
  // Excludes i from this set

  public bool Contains(int i)
  // Returns true if i is a member of this set

  public bool Contains(IntSet that) {
  // Returns true if that is a subset of this set

  public bool IsEmpty()
  // Returns true if this set is empty

  public void Fill()
  // Creates a full universe set for this set

  public void Fill(int max) {
  // Creates a full universe set for this set { 0 .. max - 1 }

  public void Clear() {
  // Clear this set

  public int Members()
  // Returns number of members in this set

  public IntSet Union(IntSet that)
  // Returns set union of this and that sets

  public IntSet Intersection(IntSet that)
  // Returns set intersection of this and that sets

  public IntSet Difference(IntSet that)
  // returns set difference of this and that sets

  public IntSet SymDiff(IntSet that)
  public IntSet Xor(IntSet that)
  // Returns set symmetric difference (xor) of this and that sets

  public void Write()
  // Simple display of this set on StdOut

  public override string ToString()
  // Returns string representation of set

  public string ToCharSetString() {
  // Returns string representation of this set as set of chars

  ------------------------------------------------- */

} // IntSet
```

```
public class OutFile {
// Provide simple facilities for text output
// P.D. Terry (p.terry@ru.ac.za)

  public static OutFile StdOut = new OutFile(Console.Out);
  public static OutFile StdErr = new OutFile(Console.Error);

  // Error handler

  public bool OpenError()
  // Returns true if the open operation on this file failed

  // Constructors - choice of three

  public OutFile()
  // Creates an OutFile to StdOut

  public OutFile(string fileName)
  // Creates an OutFile to named disk file
  // (reverts to StdOut if it fails)

  public OutFile(TextWriter s)
  // Creates an OutFile from extant stream
  // (reverts to StdOut if it fails)

  // Output routines (overloaded)

  public void Write(string s)
  public void Write(object o)
  public void Write(int o)
  public void Write(long o)
  public void Write(bool o)
  public void Write(float o)
  public void Write(double o)
  public void Write(char o)
  // Output of the parameter, prefixed by one space if numeric

  public void WriteLine()
  public void WriteLine(string s)
  public void WriteLine(object o)
  public void WriteLine(int o)
  public void WriteLine(long o)
  public void WriteLine(bool o)
  public void WriteLine(float o)
  public void WriteLine(double o)
  public void WriteLine(char o)
  // Output of the parameter, prefixed by one space if numeric,
  // followed by linemark

  public void Write(string o, int width)
  public void Write(object o, int width)
  public void Write(int o,    int width)
  public void Write(long o,   int width)
  public void Write(bool o,   int width)
  public void Write(float o,  int width)
  public void Write(double o, int width)
  public void Write(char o,   int width)
  // Output of the parameter, in required field width
  // Left justified if width < 0, right justified if width > 0,
  // one leading space if width = 0

  public void WriteLine(string o, int width)
  public void WriteLine(object o, int width)
  public void WriteLine(int o,    int width)
  public void WriteLine(long o,   int width)
  public void WriteLine(bool o,   int width)
  public void WriteLine(float o,  int width)
  public void WriteLine(double o, int width)
  public void WriteLine(char o,   int width)
  // Output of the parameter, in required field width, followed
  // by linemark
  // Left justified if width < 0, right justified if width > 0,
  // one leading space if width = 0

  public void Close()
  // Closes the file

} // OutFile
```

```
public class InFile {
// Provide simple facilities for text input
// P.D. Terry (p.terry@ru.ac.za)

  public static InFile StdIn = new InFile(Console.In);

  // Error handlers

  public bool OpenError()
  // Returns true if the open operation on this file failed

  public int ErrorCount()
  // Returns number of errors detected on input

  public static bool Done()
  // Simple error checking - reports result for last method
  // called regardless of which file was accessed

  public void ShowErrors()
  // Allows user to switch error reporting on (off by default)

  public void HideErrors()
  // Allows user to switch error reporting off (default setting)

  // Constructors - choice of three

  public InFile()
  // Creates an InFile attached to StdIn

  public InFile(string fileName)
  // Creates an InFile from named file
  // (reverts to StdIn if it fails)

  public InFile(TextReader s)
  // Creates an inFile from extant stream
  // (reverts to StdIn if it fails)

  // Utility "getters"

  public bool EOF()
  // Returns true if the last operation terminated by reaching EOF

  public bool EOL()
  // Returns true if the last operation terminated by reaching EOL

  public bool Error()
  // Returns true if the last operation on this file failed

  public bool NoMoreData()
  // Returns true if the last operation on this file failed
  // because an attempt was made to read past the end of file

  public string Name()
  // Returns the file name for the file

  public char ReadChar()
  // Reads and returns a single character

  public void ReadAgain()
  // Allows for a single character probe in code like
  // do ch = I.ReadChar(); while(notDigit(ch)); I.ReadAgain();

  public void SkipSpaces()
  // Allows for the most elementary of lookahead probes

  public void ReadLn()
  // Consumes all characters to end of line.

  // String based input

  public string ReadString()
  // Reads and returns a string.  Incorporates leading white space,
  // and terminates with a control character, which is not
  // incorporated or consumed.  Thus you might need code like
  //       s = I.ReadString(); I.ReadLn();
  // (Alternatively use ReadLine() below )
```

```
  public string ReadString(int max)
  // Reads and returns a string.  Incorporates leading white space,
  // and terminates with a control character, which is not
  // incorporated or consumed, or when max characters have been
  // read (useful for fixed format data).  Thus you might need
  // code like
  //       s = I.ReadString(10); I.ReadLn();

  public string ReadLine()
  // Reads and returns a string.  Incorporates leading white
  // space, and terminates when EOL or EOF is reached.  The EOL
  // character is not incorporated, but is consumed.

  public string ReadWord()
  // Reads and returns a word - a string delimited at either
  // end by a control character or space (usually the latter).
  // Leading spaces are discarded, and the terminating
  // character is not consumed.

  // Numeric input routines.  These always return, even if the
  // data is incorrect.  Users may probe the status of the
  // operation, or use ShowErrors to get error messages.

  public int ReadInt()
  // Reads and returns an integer.  Leading spaces are discarded.
  // Errors may be reported or quietly ignored (when 0 is
  // returned)

  public int ReadInt(int radix)
  // Reads and returns an integer (base radix, eg 10, 16, 8).
  // Leading spaces are discarded.  Errors may be reported or
  // quietly ignored (when 0 is returned)

  public long ReadLong()
  // Reads and returns a long integer.  Leading spaces are
  // discarded.
  // Errors may be reported or quietly ignored (when 0 is
  / returned)

  public int ReadShort()
  // Reads and returns a short integer.  Leading spaces are
  // discarded.
  // Errors may be reported or quietly ignored (when 0 is
  // returned)

  public float ReadFloat()
  // Reads and returns a simple float.  Leading spaces are
  // discarded.
  // Errors may be reported or quietly ignored (when 0 is
  // returned)

  public double ReadDouble()
  // Reads and returns a double number.  Leading spaces are
  // discarded.
  // Errors may be reported or quietly ignored (when 0 is
  // returned)

  public bool ReadBool()
  // Reads a word and returns a boolean, based on the first
  // letter.
  // Typically the word would be T(rue) or Y(es) or F(alse) or
  // N(o)

  public void Close()
  // Closes the file

} // InFile
```

The IO library handles input from "standard input" and generates output to "standard output", and is effectively made up of a subset of the *InFile* and *OutFile* routines.

Appendix C:

Context-free grammars and I/O facilities for Parva

The following LL(1) grammar describes the Parva language for which a compiler is developed in chapter 14.

```
COMPILER Parva
/* LL(1) grammar for Parva level 2 - syntax only */

CHARACTERS
  cr          = CHR(13) .
  lf          = CHR(10) .
  backslash   = CHR(92) .
  control     = CHR(0) .. CHR(31) .
  letter      = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                + "abcdefghijklmnopqrstuvwxyz" .
  digit       = "0123456789" .
  stringCh    = ANY - '"' - control - backslash .
  charCh      = ANY - "'" - control - backslash .
  printable   = ANY - control .

TOKENS
  identifier  = letter { letter | digit | "_" } .
  number      = digit { digit } .
  stringLit   = '"' { stringCh | backslash printable } '"' .
  charLit     = "'" ( charCh   | backslash printable ) "'" .

COMMENTS FROM "//" TO lf
COMMENTS FROM "/*" TO "*/"

IGNORE CHR(9) .. CHR(13)

PRODUCTIONS
  Parva                                 = {   FuncOrGlobalVarDeclarations
                                            | ConstDeclarations
                                          } .
  FuncOrGlobalVarDeclarations           = Type Ident ( Function | GlobalVars ) .
  Type                                  = "void" | BasicType [ "[]" ] .
  BasicType                             = "int" | "bool" .
  Function                              = "(" FormalParameters ")" Block .
  GlobalVars                            = { "," Ident } ";" .
  FormalParameters                      = [ OneParam { "," OneParam } ] .
  OneParam                              = Type Ident .
  Block                                 = "{" { Statement } "}" .
  Statement                             =   Block | AssignmentOrCall | ";"
                                          | ConstDeclarations | VarDeclarations
                                          | IfStatement | WhileStatement
                                          | HaltStatement | ReturnStatement
                                          | ReadStatement | WriteStatement .
  ConstDeclarations                     = "const" OneConst { "," OneConst } ";" .
  OneConst                              = Ident "=" Constant .
  Constant                              =   IntConst | CharConst
                                          | "true" | "false" | "null" .
  VarDeclarations                       = Type OneVar { "," OneVar } ";" .
  OneVar                                = Ident [ "=" Expression ] .
  AssignmentOrCall                      = ( IF (IsCall())
                                              Ident "(" Arguments ")"
                                            | Designator "=" Expression
                                          ) ";" .
  Designator                            = Ident [ "[" Expression "]" ] .
  Arguments                             = [ OneArg { "," OneArg } ] .
  OneArg                                = Expression .
  IfStatement                           = "if" "(" Condition ")" Statement .
  WhileStatement                        = "while" "(" Condition ")" Statement .
  HaltStatement                         = "halt" ";" .
  ReturnStatement                       = "return" [ Expression ] ";" .
  ReadStatement                         = "read"
                                          "(" ReadElement { "," ReadElement }
                                          ")" ";" .
  ReadElement                           = StringConst | Designator .
```

```
  WriteStatement                          = "write"
                                            "(" WriteElement { "," WriteElement }
                                            ")" ";" .
  WriteElement                            = StringConst | Expression .
  Condition                               = Expression .
  Expression                              = AddExp [ RelOp AddExp ] .
  AddExp                                  = [ "+" | "-" ] Term { AddOp Term } .
  Term                                    = Factor { MulOp Factor } .
  Factor                                  = ( IF (IsCall())
                                                Ident "(" Arguments ")"
                                              | Designator
                                            )
                                            | Constant
                                            | "new" BasicType "[" Expression "]"
                                            | "!" Factor | "(" Expression ")" .
  RelOp                                   = "==" | "!=" | ">" | ">=" | "<" | "<=" .
  AddOp                                   = "+" | "-" | "||" .
  MulOp                                   = "*" | "/" | "%" | "&&" .
  Ident                                   = identifier .
  StringConst                             = stringLit .
  CharConst                               = charLit .
  IntConst                                = number .
END Parva.
```

```
class IO {
  public static int ReadInt()
  // Reads a word as a textual representation of an integer
  // and returns the corresponding value.
  // Errors may be reported or quietly ignored (when 0 is
  // returned).

  public static bool ReadBool()
  // Reads a word and returns a Boolean value, based on the
  // first letter.  Typically the word would be T(rue) or Y(es)
  // or F(alse) or N(o).

  public static char ReadChar()
  // Reads and returns a single character.

  public static string ReadString()
  // Reads and returns a string.  Incorporates leading white space,
  // and terminates on a control character, which is not
  // incorporated or consumed.

  public static string ReadLine()
  // Reads and returns a string.  Incorporates leading white space,
  // and returns when EOL or EOF is reached.  The EOL character
  // is not incorporated, but is consumed.

  public static string ReadWord()
  // Reads and returns a word - a string delimited at either
  // end by a control character or space (typically the
  // latter). Leading spaces are discarded, and the terminating
  // character is not consumed.

  public static void Write(int x, int w)
  public static void Write(bool x, int w)
  public static void Write(char x, int w)
  public static void Write(string x, int w)
  // Writes representation of the value of x to standard output.
  // If w = 0, x is preceded by exactly one space
  // If w > 0, x is right justified in a field of at least
  //    w characters
  // If w < 0, x is left justified in a field of at least
  //    -w characters

  public static void Write(int x)         { Write(x, 0); }
  public static void Write(bool x)        { Write(x, 0); }
  public static void Write(char x)        { Write(x, 1); }
  public static void Write(string x)      { Write(x, 1); }
}
```

BIBLIOGRAPHY

Addyman, A.M., Brewer, R., Burnett Hall, D.G., De Morgan, R.M., Findlay, W., Jackson M.I., Joslin, D.A., Rees, M.J., Watt, D.A., Welsh, J. and Wichmann, B.A. (1979) A draft description of Pascal, *Software - Practice and Experience*, **9**(5), 381-424.

Aho, A.V., Sethi, R. and Ullman, J.D. (1986) *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, MA.

Alblas, H. and Nymeyer, A. (1996) *Practice and Principles of Compiler Building with C*, Prentice-Hall, Hemel Hempstead, England.

Bailes, P.A. (1984) A rational Pascal, *Australian Computer Journal*, **16**(4), 155-176.

Bal, H.E. and Grune, D. (1994) *Programming Language Essentials*, Addison-Wesley, Wokingham, England.

Barron, D.W. (ed) (1981) *Pascal - the Language and its Implementation*, Wiley, Chichester, England.

Bennett, J.P. (1990) *Introduction to Compiling Techniques: a First Course using ANSI C, LEX and YACC*, McGraw-Hill, London.

Bernstein, R.L. (1985) Producing good code for the case statement, *Software - Practice and Experience*, **15**(10), 1021-1024.

Bjorner, D. and Jones, C.B. (eds) (1982) *Formal Specification and Software Development*, Prentice-Hall, Hemel Hempstead, England.

Bock, J. (2002) *CIL Programming: under the Hood of .NET*, Apress, New York.

Bratman, H. (1961) An alternate form of the Uncol diagram, *Communications of the ACM*, **4**(3), 142.

Brinch Hansen, P. (1983) *Programming a Personal Computer*, Prentice-Hall, Englewood Cliffs, NJ.

Cailliau, R. (1982) How to avoid getting schlonked by Pascal, *ACM SIGPLAN Notices*, **17**(12), 31-40.

Chomsky, N. (1956) Three models for the description of language, *IRE Transactions on Information Theory*, **IT-2**, 113-124.

Chomsky, N. (1959) On certain formal properties of grammars, *Information and Control*, **2**(2), 137-167.

Cichelli, R.J. (1979) A class of easily computed, machine independent, minimal perfect hash functions for static sets, *Pascal News*, **15**, 56-59.

Cichelli, R.J. (1980) Minimal perfect hash functions made simple, *Communications of the ACM*, **23**(1), 17-19.

Cooper, D. (1983) *Standard Pascal Reference Manual*, Norton, New York.

Cormack, G.V., Horspool, R.N.S. and Kaiserwerth, M. (1985) Practical perfect hashing, *Computer Journal*, **28**(1), 54-58.

Cornelius, B.J., Lowman. I.R. and Robson, D.J. (1984) Steady-state compilers, *Software - Practice and Experience*, **14**(8), 705-709.

Dobler, H. and Pirklbauer, K. (1990) Coco-2 - a new compiler-compiler, *ACM SIGPLAN Notices*, **25**(5), 82-90.

Dobler, H. (1991) Top-down parsing in Coco-2, *ACM SIGPLAN Notices*, **26**(3), 79-87.

Drayton, P., Albahari, B. and Neward, T. (2002) *C# in a Nutshell*, O'Reilly & Associates, Sebastopol, CA.

Earley, J. and Sturgis, H. (1970) A formalism for translator interactions, *Communications of the ACM*, **13**(10), 607-617.

Elder, J. (1994) *Compiler Construction: a recursive descent model*, Prentice-Hall, Hemel Hempstead, England.

Ellis, M.A. and Stroustrup, B. (1990) *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, MA.

Engel, J. (1999) *Programming for the Java Virtual Machine*, Addison-Wesley, Reading MA.

Fischer, C.N. and LeBlanc, R.J. (1988) *Crafting a Compiler*, Benjamin Cummings, Menlo Park, CA.

Fischer, C.N. and LeBlanc, R.J. (1991) *Crafting a Compiler with C*, Benjamin Cummings, Menlo Park, CA.

Gosling, J., Joy, W., Steele, G. and Bracha, G. (2000) *The Java™ Language Specification* (2nd edn), Addison-Wesley, Reading, MA.

Gough, K.J. (1988) *Syntax Analysis and Software Tools*, Addison-Wesley, Wokingham, England.

Gough, K.J. (2002) *Compiling for the .NET Common Language Runtime (CLR)*, Prentice-Hall PTR, Upper Saddle River, NJ.

Gough, K.J. and Corney, D. (2000) Evaluating the Java Virtual Machine as a target for languages other than Java, *Proc JMLC*, Zurich, Switzerland.

Gough, K.J. (2001) Stacking them up: A Comparison of virtual machines, *Proc Australian Computer Systems and Architecture Conference, ACSAC-2001*, Gold Coast, Australia.

Gough, K.J. and Mohay, G.M. (1988) *Modula-2: A Second Course in Programming*, Prentice-Hall, Sydney, Australia.

Grosch, J. (1990) Efficient and comfortable error recovery in recursive descent parsers, *Structured Programming,* **11**, 129-140.

Grune, D. and Jacobs, C.J.H. (1988) A programmer-friendly LL(1) parser generator, *Software - Practice and Experience*, **18**(1), 29-38.

Grune, D. and Jacobs, C.J.H. (1990) *Parsing Techniques: a Practical Guide*, Ellis Horwood, Chichester.

Grune, D., Bal, H.E., Jacobs, C.J.H and Langendoen, K.G. (2000) *Modern Compiler Design*, John Wiley, New York.

Hennessy, J.L. and Mendelsohn, N. (1982) Compilation of the Pascal case statement, *Software - Practice and Experience*, **12**(9), 879-882.

Hennessy, J.L. and Patterson, D.A. (2002) *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Mateo, CA.

Hoare, C.A.R. and Wirth, N. (1973) An axiomatic definition of the programming language Pascal, *Acta Informatica*, **2**, 335-355.

Holmes, J. (1995) *Object-Oriented Compiler Construction*, Prentice-Hall, Englewood Cliffs, NJ.

ICPC (1994) Exercise 5.16 is based on one set at the 1994 East-Central Regionals of the ACM International Collegiate Programming Contest held at the University of Waterloo 11-12 November 1994, to be found at `http://www.acm.inf.ethz.ch/ProblemSetArchive/B_US_EastCen/1994/EastCentral1994.html`

Johnson, S.C. (1975) Yacc - Yet Another Compiler Compiler, *Computing Science Technical Report 32*, AT&T Bell Laboratories, Murray Hill, NJ.

Jones, R. and Lins, R.D. (1996) *Garbage Collection - Algorithms for Automatic Dynamic Memory Management*, Wiley, New York.

Kannan, S. and Proebsting, T.A. (1994) Correction to "Producing good code for the case statement", *Software - Practice and Experience*, **24**(2), 233.

Kernighan, B.W. (1981) Why Pascal is not my favorite programming language, *Computer Science Technical Report 100*, AT&T Bell Laboratories, Murray Hill, NJ.

Kernighan, B.W. and Ritchie, D.M. (1988) *The C Programming Language* (2nd edn), Prentice-Hall, Englewood Cliffs, NJ.

King, K.N. (1996) *C Programming - a Modern Approach*, W.W. Norton, New York.

Lecarme, O. and Peyrolle-Thomas, M.C. (1973) Self compiling compilers: an appraisal of their implementations and portability, *Software - Practice and Experience*, 8(2), 149-170.

Lee, J.A.N. (1972) The formal definition of the BASIC language, *Computer Journal*, **15**, 37-41.

Lee, J.A.N. and Sammet, J.E. (1978) History of Programming Languages Conference HOPL I, *ACM SIGPLAN Notices*, **13**(8).

Lee, J.A.N. and Sammet, J.E. (1993) History of Programming Languages Conference HOPL II, *ACM SIGPLAN Notices*, **28**(3).

Lesk, M.E. (1975) Lex - a lexical analyzer generator, *Computing Science Technical Report* 39, AT&T Bell Laboratories, Murray Hill, NJ.

Levine, J.R., Mason, T. and Brown D. (1992) *Lex and Yacc* (2nd edn), O'Reilly and Associates, Sebastapol, CA.

Lidin, S. (2002) *Inside Microsoft .NET IL Assembler*, Microsoft Press, Redmond, WA.

Lindholm, T. and Yellin, F. (1999) *The Java™ Virtual Machine Specification* (2nd edn), Addison-Wesley, Reading, MA.

Louden, K.C. (1997) *Compiler Construction: Principles and Practice*, PWS Publishing Company, Boston, MA.

MacCabe, A.B. (1993) *Computer Systems: Architecture, Organization and Programming*, Irwin, Boston, MA.

Mak, R. (1991) *Writing Compilers and Interpreters: an Applied Approach*, John Wiley, New York.

Mak, R. (1996) *Writing Compilers and Interpreters: an Applied Approach* (2nd edn), John Wiley, New York.

McGettrick, A.D. (1980) *The Definition of Programming Languages*, Cambridge University Press, Cambridge, England.

Meek, B.M. (1990) The static semantics file, *ACM SIGPLAN Notices*, **25**(4), 33-42.

Meyer, J. and Downing, T. (1997) *The Java Virtual Machine*, O'Reilly and Associates, Sebastopol, CA.

Mössenböck, H. (1986) Alex: a simple and efficient scanner generator, *ACM SIGPLAN Notices*, **21**(12), 139-148.

Mössenböck, H. (1990a) *Coco/R: A generator for fast compiler front ends*, Report 127, Departement Informatik, Eidgenössische Technische Hochschule, Zürich.

Mössenböck, H. (1990b) *A generator for production quality compilers*, in Proceedings of the Third International Workshop on Compiler-Compilers, Lecture Notes in Computer Science 471, Springer, Berlin.

Mössenböck, H., Beer W., Birngruber, D. and Wöß, A. (2004) *.NET Application Development : with C#, ASP.NET, ADO.NET, and Web Services*, Pearson Addison-Wesley, London, England.

Naur, P. (1960) Report on the algorithmic language Algol 60, *Communications of the ACM*, **3**, 299-314.

Naur, P. (1963) Revised report on the algorithmic language Algol 60, *Communications of the ACM*, **6**(1), 1-17.

Nori, K.V., Ammann, U., Jensen, K. *et al*. (1981) Pascal-P implementation notes, in *Pascal - the Language and its Implementation*, Barron, D.W. (ed) Wiley, Chichester, England.

Panti, M. and Valenti, S. (1992) A modulus oriented hash function for the construction of minimal perfect tables, *ACM SIGPLAN Notices*, **27**(11), 33-38.

Parr, T.J., Dietz, H.G. and Cohen W.E. (1992) PCCTS 1.00: The Purdue Compiler Construction Tool Set, *ACM SIGPLAN Notices*, **27**(2), 88-165.

Parr, T.J. and Quong, R.W. (1995) ANTLR: A predicated-LL(k) parser generator, *Software - Practice and Experience*, **25**(7), 789-810.

Parr, T.J. and Quong, R.W. (1996) LL and LR Translators need k > 1 lookahead, *ACM SIGPLAN Notices*, **31**(2), 27-34.

Parr, T.J. (1996) *Language translation using PCCTS and C++ (a Reference Guide)*, Automata Publishing, San Jose, CA.

Parsons, T.W. (1992) *Introduction to Compiler Construction*, W.H. Freeman, New York.

Pemberton, S. (1980) Comments on an error-recovery scheme by Hartmann, *Software - Practice and Experience*, **10**(3), 231-240.

Pemberton, S. and Daniels, M. (1982) *Pascal Implementation - the P4 Compiler*, Ellis Horwood, Chichester.

Pittman, T. and Peters, J. (1992) *The Art of Compiler Design*, Prentice-Hall, Englewood Cliffs, NJ.

Rechenberg, P. and Mössenböck, H. (1989) *A Compiler Generator for Microcomputers*, Prentice-Hall, Hemel Hempstead, England.

Rees, M. and Robson, D. (1987) *Practical Compiling with Pascal-S*, Addison-Wesley, Wokingham, England.

Reiser, M. and Wirth, N. (1992) *Programming in Oberon*, Addison-Wesley, Wokingham, England.

Sale, A.H.J. (1979) A note on scope, one-pass compilers, and Pascal, *Australian Computer Science Communications*, **1**(1), 80-82. Reprinted in *Pascal News*, **15**, 62-63.

Sale, A.H.J. (1981) The implementation of case statements in Pascal, *Software - Practice and Experience*, **11**(9), 929-942.

Sebesta, R.W. and Taylor, M.A. (1985) Minimal perfect hash functions for reserved word lists, *ACM SIGPLAN Notices*, **20**(12), 47-53.

Stirling, C. (1985) Follow set error recovery, *Software - Practice and Experience*, **15**(8), 239-257.

Stroustrup, B. (1990) *The C++ Programming Language* (2nd edn), Addison-Wesley, Reading, MA.

Stroustrup, B. (1993) *The Design and Evolution of C++*, Addison-Wesley, Reading, MA.

Terry, P.D. (1986) *Programming Language Translation*, Addison-Wesley, Wokingham, England.

Terry, P.D. (1995) Umbriel: another minimal programming language, *ACM SIGPLAN Notices*, **30**(5), 11- 17.

Terry, P.D. (1997) *Compilers and Compiler Generators: an Introduction With C++*, International Thomson, London.

Terry, P.D. (2005) *Compiling with C# and Java*, Pearson/Addison Wesley, London.

Topor, R.W. (1982) A note on error recovery in recursive descent parsers, *ACM SIGPLAN Notices*, **17**(2), 37-40.

Tremblay, J.P. and Sorenson, P.G. (1985) *Theory and Practice of Compiler Writing*, McGraw-Hill, New York.

Trono, J.A. (1995) A comparison of three strategies for computing letter-oriented, minimal perfect hashing functions, *ACM SIGPLAN Notices*, **30**(4), 29-35.

Ullmann, J.R. (1994) *Compiling in Modula-2 - a First Introduction to Classical Recursive Descent Compiling*, Prentice-Hall, Hemel Hempstead, England.

van den Bosch, P.N. (1992) A bibliography on syntax error handling in context free languages, *ACM SIGPLAN Notices*, **27**(4), 77-86.

Venners, B. (1999) *Inside the Java Virtual Machine*, McGraw-Hill, New York.

Waite, W.M. and Goos, G. (1984) *Compiler Construction*, Springer, New York.

Wakerly, J.F. (1981) *Microcomputer Architecture and Programming*, Wiley, New York.

Watson, D. (1989) *High-level Languages and their Compilers*, Addison-Wesley, Wokingham, England.

Watt, D.A. (1990) *Programming Language Concepts and Paradigms*, Prentice-Hall, Hemel Hempstead, England.

Watt, D.A. (1991) *Programming Language Syntax and Semantics*, Prentice-Hall, Hemel Hempstead, England.

Watt, D.A. (1993) *Programming Language Processors*, Prentice-Hall, Hemel Hempstead, England.

Watt, D.A. (2004) *Programming Language Design Concepts*, Wiley, Chichester, England.

Watt, D.A. and Brown, D.F. (2000) *Programming Language Processors in Java: Compilers and Interpreters*, Prentice-Hall, Hemel Hempstead, England.

Welsh, J. and Quinn, C. (1972) A Pascal compiler for ICL 1900 series computers, *Software - Practice and Experience*, **2**(1), 73-78.

Welsh, J., Sneeringer, W.J. and Hoare, C.A.R. (1977) Ambiguities and Insecurities in Pascal, *Software - Practice and Experience*, **7**(6), 685-696. (Reprinted in Barron (1981))

Wilson, I.P. and Addyman, A.M. (1982) *A Practical Introduction to Pascal - with BS6192*, MacMillan, London.

Wirth, N. (1974) *On the design of programming languages*, Proc IFIP Congress 74, 386-393, North-Holland, Amsterdam.

Wirth, N. (1976a) *Programming languages - what to demand and how to assess them*, Proc. Symposium on Software Engineering, Queen's University, Belfast.

Wirth, N. (1976b) *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, NJ.

Wirth, N. (1977) What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, **20**(11), 822-823.

Wirth, N. (1981) Pascal-S: A subset and its implementation, in *Pascal - the Language and its Implementation*, Barron, D.W. (ed), Wiley, Chichester, England.

Wirth, N. (1985) *Programming in Modula-2* (3rd edn), Springer, Berlin.

Wirth, N. (1986) *Compilerbau*, Teubner, Stuttgart.

Wirth, N. (1988) From Modula to Oberon, *Software - Practice and Experience*, **18**(7), 661-670.

Wirth, N. (1996) *Compiler Construction*, Addison-Wesley, Wokingham, England.

Wöß, A., Löberbauer, M., Mössenböck, H. (2003) LL(1) Conflict Resolution in a Recursive Descent Compiler Generator, *Joint Modular Languages Conference* (JMLC'03), Klagenfurt, 2003, reprinted in *Lecture Notes in Computer Science* 2789, Springer-Verlag, 2003.