Des Watson

A Practical Approach to Compiler Construction

Springer

Des Watson
Department of Informatics
Sussex University
Brighton, East Sussex
UK

ISSN 1863-7310 ISSN 2197-1781 (electronic)
Undergraduate Topics in Computer Science
ISBN 978-3-319-52787-1 ISBN 978-3-319-52789-5 (eBook)
DOI 10.1007/978-3-319-52789-5

Library of Congress Control Number: 2017932112

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

The study of programming languages and their implementations is a central theme of computer science. The design of a compiler—a program translating programs written in a high-level language into semantically equivalent programs in another language, typically machine code—is influenced by many aspects of computer science. The compiler allows us to program in high-level languages, and it provides a layer of abstraction so that we do not have to worry when programming about the complex details of the underlying hardware.

The designer of any compiler obviously needs to know the details of both the source language being translated and the language being generated, usually some form of machine code for the target machine. For non-trivial languages, the compiler is itself a non-trivial program and should really be designed by following a standard structure. Describing this standard structure is one of the key aims of this book.

The design of compilers is influenced by the characteristics of formal language specifications, by automata theory, by parsing algorithms, by processor design, by data structure and algorithm design, by operating system services, by the target machine instruction set and other hardware features, by implementation language characteristics and so on, as well as by the needs of the compilers' users. Coding a compiler can be a daunting software task, but the process is greatly simplified by making use of the approaches, experiences, recommendations and algorithms of other compiler writers.

Why Study Compiler Design?

Why should compiler design be studied? Why is this subject considered to be an important component of the education of a computer scientist? After all, only a small proportion of software engineers are employed on large-scale, traditional compiler projects. But knowing something about what happens within a compiler can have many benefits. Understanding the technology and limitations of a

compiler is important knowledge for any user of a compiler. Compilers are complex pieces of code and an awareness of how they work can very helpful. The algorithms used in a compiler are relevant to many other application areas such as aspects of text decoding and analysis and the development of command-driven interfaces. The need for simple domain-specific languages occurs frequently and the knowledge of compiler design can facilitate their rapid implementation.

Writing a simple compiler is an excellent educational project and enhances skills in programming language understanding and design, data structure and algorithm design and a wide range of programming techniques. Understanding how a high-level language program is translated into a form that can be executed by the hardware gives a good insight into how a program will behave when it runs, where the performance bottlenecks will be, the costs of executing individual high-level language statements and so on. Studying compiler design makes you a better programmer.

Why Another Book?

Why is there now yet another book on compiler design? Many detailed and comprehensive textbooks in this field have already been published. This book is a little different from most of the others. Hopefully, it presents key aspects of the subject in an accessible way, using a practical approach. The algorithms shown are all capable of straightforward implementation in almost any programming language, and the reader is strongly encouraged to read the text and in parallel produce code for implementations of the compiler modules being described. These practical examples are concentrated in areas of compiler design that have general applicability. For example, the algorithms shown for performing lexical and syntax analysis are not restricted for use in compilers alone. They can be applied to the analysis required in a wide range of text-based software.

The field of programming language implementation is huge and this book covers only a small part of it. Just the basic principles, potentially applicable to all compilers, are explained in a practical way.

What's in this Book?

This book introduces the topic of compiler construction using many programmed examples, showing code that could be used in a range of compiler and compiler-related projects. The code examples are nearly all written in C, a mature language and still in widespread use. Translating them into another programming language should not cause any real difficulty. Many existing compiler projects are written in C, many new compiler projects are being written in C and there are many compiler construction tools and utilities designed to support compiler

implementations in C. Character handling and dynamic data structure management are well-handled by C. It is a good language for compiler construction. Therefore, it may have seemed appropriate to choose the construction of a C compiler as a central project for this textbook. However, this would not have been sensible because it is a huge project, and the key algorithms of language analysis and translation would be overwhelmed by the detail necessary to deal with the numerous complexities of a "real" programming language, even one regarded as being simpler than many.

This book is primarily about compiler construction, and it is not specifically about the use of compiler-related algorithms in other application areas. Hopefully, though, there is enough information in the analysis chapters to show how these standard grammar-based techniques can be applied very much more widely.

Although many examples in this book are taken from code that may appear in a complete C compiler, the emphasis is on the development of a compiler for the DL language. This is a very simple language developed for the needs of this book from languages used in a series of practical exercises from various undergraduate and postgraduate compiler construction courses presented by the author. The syntax of DL is loosely based on a subset of C, but with many restrictions. In particular, there is just one data type (the integer), and although functions are supported, their functionality is rather restricted. Nevertheless, DL is sufficiently powerful to be usable for real problems. The syntax of DL is presented in the appendix.

The widely-available *flex* and *bison* tools are introduced, and their use in practical implementations is discussed, especially in the context of generating a compiler for DL. These particular packages provide a good insight into the benefits offered by the many powerful compiler generation tools now available.

The software examples in this book were developed and tested on systems running Fedora Linux on an x86-64 architecture. The C compiler used was GCC. Machine and operating system dependencies are probably inevitable, but any changes needed to move this code to a different computer or operating system should be comparatively minor.

The code examples are concentrated on the compiler's front-end. Code for intermediate code optimisation, target machine code generation and optimisation tends to be long, complex and often overwhelmed by target machine detail. Hence, code examples from the back-end are largely avoided in this book so that no introduction to or detailed discussion of assembly language programming is included. Instead, the text presents the basic principles of back-end design from which code generators for diverse target architectures can be developed. References are given to sources providing further algorithm examples.

The source code of a complete DL compiler is *not* presented in this book. The real reason for this is that there is an underlying assumption that one of the most important practical exercises of the book is to produce a complete compiler for DL. A large number of code examples taken from a compiler are included in the text to illustrate the principles being described so that the reader will not be coding from scratch.

How Should this Book be Used?

This book can be used to accompany taught courses in programming language implementation and compiler design, and it can also be used for self-study. There is an assumption that students using this book will have some programming skills but not necessarily significant experience of writing large software systems. A working understanding of basic data structures such as trees is essential. The examples in the book are coded in C, but a comprehensive knowledge of C is really not required to understand these examples and the accompanying text. A basic knowledge of computer hardware is also assumed, including just the rudiments of the principles of assembly-level programming.

Each chapter ends with a few exercises. They vary a great deal in complexity. Some involve just a few minutes of thought, whereas others are major programming projects. Many of the exercises are appropriate for group discussion and some may form the basis of group projects involving code implementation as well as research.

It is especially important to make the most of the practical aspects of this subject by coding parts of a compiler as the book is being read. This will help greatly to alleviate boredom and will hugely help with the process of understanding. For example, for the newcomer to recursive descent parsing, the power and elegance of the technique can only be fully appreciated when a working implementation has been written.

The obvious project work associated with this book is to write a complete compiler for the DL language. Assuming that a simple target machine is chosen, the project is of a reasonable size and can fit well into an average size university or college course. Extensions to such a compiler by including optimisation and register allocation can follow in a more advanced course. The project can be taken even further by developing the DL compiler into a complete C compiler, but the time required to do this should not be underestimated. Writing a simple compiler following the steps described in this book is not a huge task. But it is important not to abandon the standard techniques. I have seen some students getting into major difficulties with the implementation of their compilers, coded using a "much better algorithm" of their own devising! The correct approach is reliable and really does involve a careful and systematic implementation with extensive testing of each module before it is combined with others.

Although the DL language is used as an example in most of the chapters, this book is not intended to be a tutorial guide for writing DL compilers. Its aims are much broader than this—it tries to present the principles of compiler design and the implementation of certain types of programming language, and where appropriate, DL-targeted examples are presented. Should the reader want to accept the challenge of writing a complete DL compiler (and I would certainly recommend this), then the key practical information about lexical and syntax analysis is easy to find in Chaps. 3 and 5 and semantic analysis in Chap. 6. There is then some information about DL-specific issues of code generation in Chap. 8.

Turning the compiler construction project into a group project worked well. Programming teams can be made responsible for the construction of a complete compiler. The development can be done entirely by members of the team or it may be possible for teams to trade with other teams. This is a good test of well-documented interfaces. Producing a set of good test programs to help verify that a compiler works is an important part of the set of software modules produced by each team.

Generating standard-format object code files for real machines in an introductory compilers course may be trying to go a little too far. Generating assembly code for a simple processor or for a simple subset of a processor's features is probably a better idea. Coding an emulator for a simple target machine is not difficult—just use the techniques described in this book, of course. Alternatively, there are many virtual target architecture descriptions with corresponding emulator software freely available. The MIPS architecture, with the associated SPIM software [1], despite its age, is still very relevant today and is a good target for code generation. The pleasure of writing a compiler that produces code that actually runs is considerable!

Acknowledgement This book is loosely based on material presented in several undergraduate and postgraduate lecture courses at the University of Sussex. I should like to thank all the students who took these courses and who shared my enthusiasm for the subject. Over the years, I watched thousands of compilers being developed and discovered which parts of the process they usually found difficult. I hope that I have addressed those issues properly in this book.

Thanks also go to my colleagues at the University of Sussex—in particular to all the staff and students in the Foundations of Software Systems research group who provided such a supportive and stimulating work environment. Particular thanks go to Bernhard Reus for all his suggestions and corrections.

I'm really grateful to Ian Mackie, the UTICS series editor, and to Helen Desmond at Springer for their constant enthusiasm for the book. They always provided advice and support just when it was needed.

Finally, and most important, I should like to thank Wendy, Helen and Jonathan for tolerating my disappearing to write and providing unfailing encouragement.

Sussex, UK — Des Watson

Reference

1. Larus JR (1990) SPIM S20: a MIPS R2000 simulator. Technical Report 966. University of Wisconsin-Madison, Madison, WI, Sept 1990

Contents

Figures

Chapter 1
Introduction

The high-level language is the central tool for the development of today's software. The techniques used for the implementation of these languages are therefore very important. This book introduces some of the practicalities of producing implementations of high-level programming languages on today's computers. The idea of a *compiler*, traditionally translating from the high-level language source program to machine code for some real hardware processor, is well known but there are other routes for language implementation. Many programmers regard compilers as being deeply mysterious pieces of software—black boxes which generate runnable code—but some insight into the internal workings of this process may help towards their effective use.

Programming language implementation has been studied for many years and it is one of the most successful areas of computer science. Today's compilers can generate highly optimised code from complex high-level language programs. These compilers are large and extremely complex pieces of software. Understanding what they do and how they do it requires some background in programming language theory as well as processor design together with a knowledge of how best to structure the processes required to translate from one computer language to another.

1.1 High-Level Languages

Even in the earliest days of electronic computing in the 1940s it was clear that there was a need for software tools to support the programming process. Programming was done in *machine code*, it required considerable skill and was hard work, slow and error prone. Assembly languages were developed, relieving the programmer from having to deal with much of the low-level detail, but requiring an *assembler*, a piece of software to translate from assembly code to machine code. Giving symbolic names to instructions, values, storage locations, registers and so on allows the programmer

to concentrate on the coding of the algorithms rather than on the details of the binary representation required by the hardware and hence to become more productive. The *abstraction* provided by the assembly language allows the programmer to ignore the fine detail required to interact directly with the hardware.

The development of high-level languages gathered speed in the 1950s and beyond. In parallel there was a need for compilers and other tools for the implementation of these languages. The importance of formal language specifications was recognised and the correspondence between particular grammar types and straightforward implementation was understood. The extensive use of high-level languages prompted the rapid development of a wide range of new languages, some designed for particular application areas such as COBOL for business applications [1] and FORTRAN for numerical computation [2]. Others such as PL/I (then called NPL) [3] tried to be very much more general-purpose. Large teams developed compilers for these languages in an environment where target machine architectures were changing fast too.

1.1.1 Advantages of High-Level Languages

The difficulties of programming in low-level languages are easy to see and the need for more user-friendly languages is obvious. A programming notation much closer to the problem specification is required. Higher level abstractions are needed so that the programmer can concentrate more on the problem rather than the details of the implementation of the solution.

High-level languages can offer such abstraction. They offer many potential advantages over low-level languages including:

- Problem solving is significantly faster. Moving from the problem specification to code is simpler using a high-level language. Debugging high-level language code is much easier. Some high-level languages are suited to rapid prototyping, making it particularly easy to try out new ideas and add debugging code.
- High-level language programs are generally easier to read, understand and hence maintain. Maintenance of code is now a huge industry where programmers are modifying code unlikely to have been written by themselves. High-level language programs can be made, at least to some extent, self-documenting, reducing the need for profuse comments and separate documentation. The reader of the code is not overwhelmed by the detail necessary in low-level language programs.
- High-level languages are easier to learn.
- High-level language programs can be structured more easily to reflect the structure of the original problem. Most current high-level languages support a wide range of program and data structuring features such as object orientation, support for asynchronous processes and parallelism.
- High-level languages can offer software portability. This demands some degree of language standardisation. Most high-level languages are now fairly tightly defined

so that, for example, moving a Java program from one machine to another with different architectures and operating systems should be an easy task.
- Compile-time checking can remove many bugs at an early stage, before the program actually runs. Checking variable declarations, type checking, ensuring that variables are properly initialised, checking for compatibility in function arguments and so on are often supported by high-level languages. Furthermore, the compiler can insert runtime code such as array bound checking. The small additional runtime cost may be a small price to pay for early removal of errors.

1.1.2 Disadvantages of High-Level Languages

Despite these significant advantages, there may be circumstances where the use of a low-level language (typically an assembly language) may be more appropriate. We can identify possible advantages of the low-level language approach.

- The program may need to perform some low-level, hardware-specific operations which do not correspond to a high-level language feature. For example, the hardware may store device status information in a particular storage location—in most high-level languages there is no way to express direct machine addressing. There may be a need to perform low-level i/o, or make use of a specific machine instruction, again probably difficult to express in a high-level language.
- The use of low-level languages is often justified on the grounds of efficiency in terms of execution speed or runtime storage requirements. This is an important issue and is discussed later in this section.

These disadvantages of high-level languages look potentially serious. They need further consideration.

1.1.2.1 Access to the Hardware

A program running on a computer system needs to have access to its environment. It may input data from the user, it may need to output results, create a file, find the time of day and so on. These tasks are hardware and operating system specific and to achieve some portability in coding the high-level language program performing these actions, the low-level details have to be hidden. This is conventionally done by providing the programmer with a library acting as an interface between the program and the operating system and/or hardware. So if the program wants to write to a file, it makes a call to a library routine and the library code makes the appropriate operating system calls and performs the requested action.

To address the stated advantage of low-level languages mentioned above there is nothing to stop the construction of operating system and machine-specific libraries to perform the special-purpose tasks such as providing access to a particular storage

location or executing a particular machine instruction. There *may* be machine-specific problems concerned with the mechanism used to call and return from this library code with, for example, the use of registers, or with the execution time cost of the call and return, but such difficulties can usually be overcome.

A few programming languages provide an alternative solution by supporting inline assembly code. This code is output unchanged by the compiler, providing the high-level language program direct access to the hardware. This is a messy solution, fraught with danger, and reliable means have to be set up to allow data to be passed into and returned from this code. Such facilities are rarely seen today.

1.1.2.2 Efficiency

There are many programming applications where efficiency is a primary concern. These could be large-scale computations requiring days or weeks of processor time or even really short computations with severe real-time constraints. Efficiency is usually concerned with the minimisation of computation time, but other constraints such as memory usage or power consumption could be more important.

In the early development of language implementations, the issue of efficiency strongly influenced the design of compilers. The key disadvantage of high-level languages was seen as being one of poor efficiency. It was assumed that machine-generated code could never be as efficient as hand-written code. Despite some remarkable optimisations performed by some of the early compilers (particularly for FORTRAN), this remained largely true for many years. But as compiler technology steadily improved, as processors became faster and as their architectures became more suited to running compiler-generated code from high-level language programs, the efficiency argument became much less significant. Today, compiler-generated code for a wide range of programming languages and target machines is likely to be just as efficient, if not more so, than hand-written code.

Does this imply that justifying the use of low-level languages on the grounds of producing efficient code is now wrong? The reality is that there may be some circumstances where coding in machine or assembly code, very carefully, by hand, will lead to better results. This is not really feasible where the amount of code is large, particularly where the programmer loses sight of the large-scale algorithmic issues while concentrating on the small-scale detail. But it may be a feasible approach where, for example, a small function needs to run particularly quickly and the skills of a competent low-level programmer with a good knowledge of the target machine are available. Mixing high-level language programming with low-level language programming is perfectly reasonable in this way. But if the amount of code to be optimised is very small, other automated methods may be available (for example [4]).

When developing software, a valuable rule to remember is that there is no need to optimise if the code is already fast enough. Modern processors are fast and a huge amount can be achieved during the time taken for a human to react to the computer's output. However, this does not imply that compilers need never concern themselves

with code optimisation—there will always be some applications genuinely needing the best out of the hardware.

The case for using high-level languages for almost all applications is now very strong. In order to run programs written in high-level languages, we need to consider how they can be implemented.

1.2 High-Level Language Implementation

A simplistic but not inaccurate view of the language implementation process suggests that some sort of translator program is required (a compiler) to transform the high-level language program into a semantically equivalent machine code program that can run on the target machine. Other software, such as libraries, will probably also be required. As the complexity of the source language increases as the language becomes "higher and higher-level", closer to human expression, one would expect the complexity of the translator to increase too.

Many programming languages have been and are implemented in this way. And this book concentrates on this implementation route. But other routes are possible, and it may be the characteristics of the high-level language that forces different approaches. For example, the traditional way of implementing Java makes use of the Java Virtual Machine (JVM) [5], where the compiler translates from Java source code into JVM code and a separate program (an *interpreter*) reads these virtual machine instructions, emulating the actions of the virtual machine, effectively running the Java program. This seemingly contrary implementation method does have significant benefits. In particular it supports Java's feature of dynamic class loading. Without such an architecture-neutral virtual machine code, implementing dynamic class loading would be very much more difficult. More generally, it allows the support of *reflection*, where a Java program can examine or modify at runtime the internal properties of the executing program.

Interpreted approaches are very appropriate for the implementation of some programming languages. Compilation overheads are reduced at the cost of longer runtimes. The programming language implementation field is full of tradeoffs. These issues of compilers versus interpreters are investigated further in Chap. 2.

To make effective use of a high-level language, it is essential to know something about its implementation. In some demanding application areas such as *embedded systems* where a computer system with a fixed function is controlling some electronic or mechanical device, there may be severe demands placed on the embedded controller and the executing code. There may be real-time constraints (for example, when controlling the ignition timing in a car engine where a predefined set of operations has to complete in the duration of a spark), memory constraints (can the whole program fit in the 64 k bytes available on the cheap version of the microcontroller chip?) or power consumption constraints (how often do I have to charge the batteries in my mobile phone?). These constraints make demands on the performance of the hardware but also on the way in which the high-level language implementing

the system's functionality is actually implemented. The designers need to have an in-depth knowledge of the implementation to be able to deal with these issues.

1.2.1 Compilers

The compiler is a program translating from a *source language* to a *target language*, implemented in some *implementation language*. As we have seen, the traditional view of a compiler is to take some high-level language as input and generate machine code for some target machine. Choice of implementation language is an interesting issue and we will see later in Chap. 9 why the choice of this language may be particularly important.

The field of compilation is not restricted to the generation of low-level language code. Compiler technology can be developed to translate from one high-level language to another. For example, some of the early C++ compilers generated C code rather than target machine code. Existing C compilers were available to perform the final step.

The complexity of a compiler is not only influenced by the complexities of the source and target languages, but also by the requirement for optimised target code. There is a real danger in compiler development of being overwhelmed by the complexities and the details of the task. A well-structured approach to compiler development is essential.

1.2.2 Compiler Complexity

The tools and processes of programming language implementation cannot be considered in isolation. Collaboration between the compiler writer, the language designer and the hardware architect is vital. The needs of the end-users must be incorporated too. Compilers have responded to the trend towards increased high-level language complexity and the desire for aggressive optimisation by becoming significantly more complex themselves. In the early days of compilers the key concern was the generation of good code, rivalling that of hand coders, for machines with very irregular architectures. Machine architectures gradually became more regular, hence making it easier for the compiler writer. Subsequent changes (from the 1980s) towards the much simpler instruction sets of the reduced instruction set computers (RISC) helped simplify the generation of good code. Attention was also being focused on the design of the high-level languages themselves and the support for structures and methodologies to help the programmers directly. Today, the pressures caused by new languages and other tools to support the software development process are still there and also the rapid move towards distributed computing has placed great demands on program analysis and code generation. Parallelism, in its various forms, offers higher performance processing but at a cost of increased implementation complexity.

The language implementation does not stop at the compiler. The support of collections of library routines is always required, providing the environment in which code generated by the compiler can run. Other tools such as debuggers, linkers, documentation aids and interactive development environments are needed too. This is no easy task.

Dealing with this complexity requires a strict approach to design in the structuring of the compiler construction project. Traditional techniques of software engineering are well applied in compiler projects, ensuring appropriate modularisation, testing, interface design and so on. Extensive stage-by-stage testing is vital for a compiler. A compiler may produce highly optimised code, but if that code produces the wrong answers when it runs, the compiler is not of much use. To ease the task of producing a programming language implementation, many software tools have been developed to help generate parts of a compiler or interpreter automatically. For example, lexical analysers and syntax analysers (two early stages of the compilation process) are often built with the help of tools taking the formal specification of the syntax of the programming language as input and generating code to be incorporated in the compiler as output. The modularisation of compilers has also helped to reduce workload. For example, many compilers have been built using a target machine independent front-end and a source language-independent back-end using a standard intermediate representation as the interface between them. Then front-ends and back-ends can be mixed and matched to produce a variety of complete compilers. Compiler projects rarely start from scratch today.

Fortunately, in order to learn about the principles of language implementation, compiler construction can be greatly simplified. If we start off with a simple programming language and generate code for a simple, maybe virtual, machine, not worrying too much about high-quality code, then the compiler construction project should not be too painful or protracted.

1.2.3 Interpreters

Running a high-level language program using a compiler is a two-stage process. In the first stage, the source program is translated into target machine code and in the second stage, the hardware executes or a virtual machine interprets this code to produce results. Another popular approach to language implementation generates no target code. Instead, an *interpreter* reads the source program and "executes" it directly. So if the interpreter encounters the source statement `a = b + 1`, it analyses the source characters to determine that the input is an assignment statement, it extracts from its own data structures the value of `b`, adds one to this value and stores the result in its own record of `a`.

This process of source-level interpretation sounds attractive because there is no need for the potentially complex implementation of code generation. But there are practical problems. The first problem concerns performance. If a source statement is executed repeatedly it is analysed each time, before each execution. The cost of

possibly multiple statement analysis followed by the interpreter emulating the action of the statement will be many times greater than the cost of executing a few machine instructions obtained from a compilation of `a = b + 1`. However this cost can be reduced fairly easily by only doing the analysis of the program once, translating it into an intermediate form that is subsequently interpreted. Many languages have been implemented in this way, using an interpreted intermediate form, despite the overhead of interpretation.

The second problem concerns the need for the presence of an interpreter at runtime. When the program is "executing" it is located in the memory of the target system in source or in a post-analysis intermediate form, together with the interpreter program. It is likely that the total memory footprint is much larger than that of equivalent compiled code. For small, embedded systems with very limited memory this may be a decisive disadvantage.

All programming language implementations are in some sense interpreted. With source code interpretation, the interpreter is complex because it has to analyse the source language statements and then emulate their execution. With intermediate code interpretation, the interpreter is simpler because the source code analysis has been done in advance. With the traditional compiled approach with the generation of target machine code, the interpretation is done entirely by the target hardware, there is no software interpretation and hence no overhead. Looking at these three levels of interpretation in greater detail, one can easily identify tradeoffs:

Source-level interpretation—interpreter complexity is high, the runtime efficiency is low (repeated analysis and emulation of the source code statements), the initial compilation cost is zero because there is no separate compiler and hence the delay in starting the execution of the program is also zero.

Intermediate code interpretation—interpreter complexity is lower, the runtime efficiency is improved (the analysis and emulation of the intermediate code statements is comparatively simple), there is an initial compilation cost and hence there is a delay in starting the program.

Target code interpretation—full compilation—there is no need for interpreter software so interpreter complexity is zero, the runtime efficiency is high (the interpretation of the code is done directly by the hardware), there is a potentially large initial compilation cost and hence there may be a significant delay in starting the program.

The different memory requirements of the three approaches are somewhat harder to quantify and depend on implementation details. In the source-level interpretation case, a simplistic implementation would require both the text of the source code and the (complex) interpreter to be in main memory. The intermediate code interpretation case would require the intermediate code version of the program and the (simpler) interpreter to be in main memory. And in the full compilation case, just the compiled target code would need to be in main memory. This, of course, takes no account of the memory requirements of the running program—space for variables, data structures, buffers, library code, etc.

There are other tradeoffs. For example, when the source code is modified, there is no additional compilation overhead in the source-level interpretation case, whereas in the full compilation case, it is usual for the entire program or module to be recompiled. In the intermediate code interpretation case, it may be possible to just recompile the source statements that have changed, avoiding a full recompilation to intermediate code.

Finally, it should be emphasised that this issue of lower efficiency of interpreted implementations is rarely a reason to dismiss the use of an interpreter. The interpreting overhead in time and space may well be irrelevant, particularly in larger computer systems, and the benefits offered may well overwhelm any efficiency issues.

1.3 Why Study Compilers?

It is important to ask why the topic of compiler construction is taught to computer science students. After all, the number of people who actually spend their time writing compilers is small. Although the need for compilers is clear, there is not really a raging demand for the construction of new compilers for general-purpose programming languages.

One of the key motivations for studying this technology is that compiler-related algorithms have relevance to application areas outside the compiler field. For example, transforming data from one syntactic form into another can be approached by considering the grammar of the structure of the source data and using traditional parsing techniques to read this data and then output it in the form of the new grammar. Furthermore, it may be appropriate to develop a simple language to act as a user interface to a program. The simplicity and elegance of some parsing algorithms and basing the parsing on a formally specified grammar helps produce uncomplicated and reliable software tools.

Studying compiler construction offers the computer science student many insights. It gives a practical application area for many fundamental data structures and algorithms, it allows the construction of a large-scale and inherently modular piece of software, ideally suited for construction by a team of programmers. It gives an insight into programming languages, helping to show why some programming languages are the way they are, making it easier to learn new languages. Indeed, one of the best ways of learning a programming language is to write a compiler for that language, preferably writing it in its own language. Writing a compiler also gives some insight into the design of target machines, both real and virtual.

There is still a great deal of work to be done in developing compilers. As new programming languages are designed, new compilers are required, and new programming paradigms may need novel approaches to compilation. As new hardware architectures are developed, there is a need for new compilation and code generation strategies to make effective use of the machine's features.

Although there is a steady demand for new compilers and related software, there is also a real need for the development of new *methodologies* for the construction

of high-quality compilers generating efficient code. Implementing code to analyse the high-level language program is not likely to be a major challenge. The area has been well researched and there are good algorithms and software tools to help. But the software needed to generate target code, particularly high-quality code, is *much* harder to design and write. There are few standard approaches for this part of the compiler. And this is where there is an enormous amount of work still to be done. For example, generating code to make the best use of parallel architectures is hard, and we are a long way from a general, practical solution.

Is there really a need for heavily optimised target code? Surely, today's processors are fast enough and are associated with sufficiently large memories? This may be true for some applications, but there are many computations where there are, for example, essential time or space constraints. These are seen particularly in embedded applications where processor power or memory sizes may be constrained because of cost or where there are severe real-time constraints. There will always be a need to get the most from the combination of hardware and software. The compiler specialist still has a great deal of work to do.

1.4 Present and Future

Today's compilers and language tools can deal with complex (both syntactically and semantically) programming languages, generating code for a huge range of computer architectures, both real and virtual. The quality of generated code from many of today's compilers is astonishingly good, often far better than that generated by a competent assembly/machine code programmer. The compiler can cope well with the complex interacting features of computer architectures. But there are practical limits. For example, the generation of truly optimal code (optimising for speed, size, power consumption, etc.) may in practice be at best time consuming or more likely impossible. Where we need to make the best use of parallel architectures, today's compilers can usually make a good attempt, but not universally. There are many unsolved optimisation-related problems. Also, surely there must be better processor architectures for today's and tomorrow's programming languages?

Compilers are not just about generating target code from high-level language programs. Programmers need software tools, probably built on compiler technology, to support the generation of high-quality and reliable software. Such tools have been available for many years to perform very specific tasks. For example, consider the *lint* tool [6] to highlight potential trouble spots in C programs. Although huge advances have been made, much more is required and this work clearly interacts with programming language design as well as with diverse areas of software engineering.

1.5 Conclusions and Further Reading

A study of the history of programming languages provides some good background to the development of implementations. Detailed information about early languages appears in [7] and more recent articles are easily found on the web. Several pictorial timelines have been produced, showing the design connections between programming languages.

Similarly, the history of processor design shows how the compiler writer has had to deal with a rapidly changing target. Web searches reveal a huge literature, and it is easy to see how the development of hardware architectures in more recent years has been influenced by the compiler writer and indirectly by the needs of the high-level language programmer. A full and wide-ranging coverage of computer hardware is contained in [8]. A comprehensive coverage of modern architecture together with historical perspectives is found in [9].

To help put the material appearing in the rest of this book into context, it is worth looking at some existing compiler projects. But it is important not to be put off by the scale of some of these projects. Many have been developed over decades, with huge programming teams. Perhaps most famous is GCC (the GNU Compiler Collection), documented at https://gcc.gnu.org/ which "… includes front ends for C, C++, Objective-C, Fortran,[1] Java, Ada, and Go". This website includes links to numerous documents describing the project from the point of view of the user, the maintainer, the compiler writer and so on. The LLVM Compiler Infrastructure is documented at http://llvm.org — another collection of compilers and related tools, now in widespread use. The `comp.compilers` newsgroup and website (http://compilers.iecc.com/) is an invaluable resource for compiler writers.

Exercises

1.1 Try to find a compiler that has an option to allow you to look at the generated code (for example, use the `-S` option in GCC). Make sure that any optimisation options are turned off. Look at the code generated from a simple program (only a few source lines) and try to match up blocks of generated code with source statements. Details are not important, and you do not need an in-depth knowledge of the architecture or instruction set of the target machine. By using various source programs, try to get some vague idea of how different source language constructs are translated.
Now, turn on optimisation. It should be very much harder to match the input with the output. Can you identify any specific optimisations that are being applied?

1.2 Find the documentation for a "big" compiler. Spend some time looking at the options and features supported by the package.

1.3 Find the instruction set of the main processor contained in the computer you use most often. How much memory can it address? How many registers does

[1] Note the capitalisation. After FORTRAN 77, the language became known as Fortran.

it have? What can it do in parallel? How many bits can it use for integer and for floating point arithmetic? Approximately how long does it take to add two integers contained in registers?

1.4 Look back at the design of a much older processor (for example, the DEC PDP-11 architecture is well documented on the web). Answer the questions listed above for this processor.

1.5 Try to find out something about the optimisations performed by the early FORTRAN compilers. The work done by IBM on some early FORTRAN compilers is well documented and shows the efforts made by the company to move programmers away from assembly language programming. Try also to find out about more recent attempts with Fortran (note the capitalisation!) to make the most of parallel architectures.

1.6 Do some research to find out the key trends in processor design over the last few decades. Do the same for high-level language design. How have these trends affected the design of compilers and interpreters?

1.7 To prepare for the practical programming tasks ahead, write a simple program to read a text file, make some trivial transformation character by character such as swapping the case of all letters, and write the result to another text file. Keep this program safe. It will be useful later when it can be augmented to produce a complete lexical analyser.

References

1. American National Standards Institute, New York (1974) USA Standard COBOL, X3.23-1974
2. United States of America Standards Institute, New York (1966) USA Standard FORTRAN – USAS X3.9-1966
3. Radin G, Paul Rogoway H (1965) NPL: highlights of a new programming language. Commun ACM 8(1):9–17
4. Massalin H (1987) Superoptimizer – a look at the smallest program. In: Proceedings of the second international conference on architectural support for programming languages and operating systems (ASPLOS-II). Palo Alto, California. Published as ACM SIGPLAN Notices 22:10, pp 122–126
5. Lindholm T, Yellin F (1997) The Java virtual machine specification. The Java series. Addison-Wesley, Reading
6. Johnson SC (1978) Lint, a C program checker. Technical report. Bell Laboratories, Murray Hill, 07974
7. Sammet JE (1969) Programming languages: history and fundamentals. Prentice-Hall, Englewood Cliffs
8. Tanenbaum AS, Austin T (2013) Structured computer organization. Pearson, Upper Saddle River
9. Hennessy JL, Patterson DA (2012) Computer architecture – a quantitative approach, 5th edn. Morgan Kaufmann, San Francisco

Chapter 2
Compilers and Interpreters

Before looking at the details of programming language implementation, we need to examine some of the characteristics of programming languages to find out how they are structured and defined. A compiler, or other approach to implementation, is a large and complex software system and it is vital to have some clear and preferably formal structure to support its construction.

This chapter examines some of the approaches that can be used for high-level programming language implementation on today's computer hardware and provides some of the background to enable high-level to low-level language translation software to be designed in a structured and standard way.

2.1 Approaches to Programming Language Implementation

The traditional approach for the implementation of a programming language is to write a program that translates programs written in that language into equivalent programs in the machine code of the target processor. To make the description of this process a little easier, we shall assume that the source program is written in a language called `mylanguage` and we are producing a program to run on `mymachine`. This view is shown in Fig. 2.1.

This trivial diagram is important because it forces us to consider some important issues. First, what precisely is the nature of the source program? It is obviously a program written in `mylanguage`—the language we are trying to implement. But before we can contemplate an implementation of the translator software, we have to have a precise definition of the rules and structure of programs written in `mylanguage`. In Sect. 2.2, we consider the nature of such a definition.

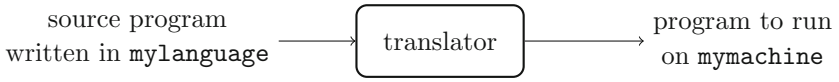


Fig. 2.1 A simple view of programming language implementation

Second, what sort of programming language is `mylanguage`? At this stage it may not really matter, but the nature of the language will of course affect the design of the translator software. For example, if `mylanguage` is a high-level programming language and `mymachine` is a hardware-implemented processor, then the translator program is usually called a *compiler*. This book concentrates on this particular configuration. If, however, `mylanguage` is an assembly language (for `mymachine`) then the translator is usually called an *assembler* and would be significantly easier to implement than a compiler.

Can programs in `mylanguage` be passed directly into the translator or does it make more sense to preprocess the programs first? For example, programs written in C can include language features best dealt with by a preprocessor. This stage could of course be regarded as being an integral part of the translation process.

Third, what sort of language is used to express the program to run on `mymachine`? Again if `mymachine` is a hardware-implemented processor, then we are probably looking at the generation of machine code programs encoded in some object file format dependent on the design of `mymachine` and probably on the operating system running on `mymachine`. Generating assembly language may be the right thing to do, requiring the existence of a separate assembler program to produce code directly runnable on `mymachine`. And there are other possibilities too. A popular approach to language implementation assumes that `mymachine` is a *virtual machine*. This is a machine for which there is no corresponding hardware and exists only as a consequence of a piece of software which emulates the virtual machine instructions. This approach is examined in Sect. 2.1.1. Furthermore, the translator could have a somewhat different role. It could be used to translate from `mylanguage` to a high-level language rather than to a low-level language for `mymachine`. This would result in a software tool which could be used to translate programs from one high-level language to another, for example from C++ to C. In this case, aspects of the internal design of the translator may be rather different to that of a conventional compiler and there is an assumption that `mymachine` somehow runs programs written in the target high-level language. There are many common principles and algorithms that can be used in all these language translation tasks. Whatever the form of `mymachine` and the generated code, a precise specification is required, just as for `mylanguage`.

At the beginning of this section is a statement that the translator generates "equivalent programs" for `mymachine`. This is an important issue. The translator should preserve the semantics of the `mylanguage` program in the running of the generated code on `mymachine`. The semantics of `mylanguage` may be specified formally or informally and the user of `mylanguage` should have a clear idea of what each valid program should "mean". For example, translating the statement `a = a + 2`

into code that increments the value of `a` by `3` is not right, certainly for a sensible programming language! The translator should translate correctly.

Assuming that `mylanguage` and `mymachine` are both non-trivial, the translator is going to be a complex piece of software. It is the role of this book to help explain how this software can be structured to make it feasible to produce a reliable translator in a reasonable time. We concentrate in this book on the structure of compilers and later in this chapter a traditional internal structure of a compiler is described. But it is helpful now to say that a compiler can be built of two distinct phases. The first is the *analysis phase*, reading the source program in `mylanguage`, creating internal data structures reflecting its syntactic and semantic structure according to the definition of `mylanguage`. The second is the *synthesis phase*, generating code for `mymachine` from the data structures created by the analysis phase. Thinking about a compiler in terms of these two distinct phases can greatly simplify both design and implementation.

2.1.1 Compile or Interpret?

Figure 2.1 illustrates the conventional view of a compiler used to generate code for a target machine from a source program written in a high-level language. This book concentrates on the design of this type of translator, how it can be structured and implemented. The term *compiler* is usually used for a translator from a high-level programming language to a low-level language such as the machine code of a target machine. However, as we have seen, the term is sometimes used to cover translation from and to a wider range of programming language types, such as high-level language to another high-level language.

When a program is passed through a compiler generating code for some target machine (say, the `mymachine` processor), the code can be run on the `mymachine` architecture and this has the effect of "running" the original program. The processor hardware *interprets* the machine instructions generated by the compiler and the cpu state is altered according to the nature of the sequence of instructions executed. However, other implementation routes are possible and in particular there is no fundamental necessity for the instructions generated by the compiler to be interpreted directly by the hardware.

There are many implementations of high-level programming languages where the compiler generates code for a *virtual machine* and then a separate program, the *interpreter*, reads the virtual machine code and emulates the execution of the virtual machine, instruction by instruction. At first sight this may seem a strange thing to do—why not generate target machine code directly? But this approach has several significant advantages, including the following.

- The design of the code generated by the compiler is not constrained by the architecture of the target machine. This can simplify the design of the compiler because it does not have to deal with the quirks and restrictions of the real hardware. For

example, it may be convenient to use a stack-based architecture for the virtual machine, not directly supported by the target hardware.
- Portability is enhanced. If the interpreter is written in a portable language, the interpreter and the virtual machine code can be shipped and easily run on machines with different architectures or operating systems. This ties in well with today's prevalence of heterogeneous networked environments.
- The virtual machine code can be designed to be particularly compact. There are application areas where this may be very important.
- Runtime debugging and monitoring features can be incorporated in the virtual machine interpreter allowing improved safety and security in program execution. The virtual machine code can run in a sandbox, preventing it from performing illegal operations. For example, the virtual machine can operate on typed data, and runtime type checking can provide helpful debugging information.

The obvious disadvantage of this approach concerns the question of efficiency. Interpreted code is likely to be slower than native execution. But for most applications this turns out not to be of real significance. The advantages usually easily outweigh this disadvantage and so many modern programming languages are implemented in this way.

The nature of the virtual machine poses interesting questions. The design of the virtual machine should not be too close to that of the hardware because the advantages of compiler simplification essentially disappear. But if the virtual machine's design is made very close or identical to the language that is being implemented, the compiler is made very simple, but the interpreter has to deal with the detail of decoding and analysing this potentially complex code. The functions of the compiler are being shifted into the interpreter. However, several languages have been implemented successfully in this way, where the interpreter does all the work, removing the necessity for the separate compiler. Some implementations of the BASIC language have been implemented in this way. Because of the need for repeated analysis of the source language statements in the interpreter, this is rarely a practical approach for a production system.

These implementation issues are examined again in Chap. 9.

2.2 Defining a Programming Language

The definition of a programming language is fundamentally important to users of the programming language as well as to the compiler writer. The definition has to provide information about how to write programs in the language. It has to specify the set, presumably infinite, of valid programs and also what each valid program "means". The specification of *syntax* is central here. Syntax defines the sequences of characters that could form a valid program. And the meaning of these programs is specified by the *semantics* of the language.

The language definition should be clear, precise, complete, unambiguous and preferably understandable by all language users. Specifying a programming language in an informal way using a language such as English makes the definition accessible, but precision can suffer. Aspects of many programming languages have been defined using natural language and ambiguities have been common, particularly in early revisions of the definitions. The definition of the language's syntax is usually done using a more formal approach and a range of *metalanguages* (languages used to define other languages) have been developed. A few of these metalanguages are described in Sect. 2.2.1. For most of the current and popular programming languages, the use of a simple metalanguage to define syntax results in a compact and largely complete syntactic specification. We shall see in Chap. 4, how this specification can be used as the starting point for the design of a syntax analyser for the language.

2.2.1 BNF and Variants

Backus–Naur Form or *Backus Normal Form* is a metalanguage which was popularised by its use in the definition of the syntax of ALGOL 60 [1]. It is a very simple yet powerful metalanguage and it has been used extensively to support the formal definitions of a huge range of languages. It has become one of the fundamental tools of computer science.

A BNF specification consists of a set of rules, each rule defining a symbol (strictly a *non-terminal symbol*) of the language. The use of BNF is best illustrated by some examples. This first example (see Fig. 2.2) makes use of some of the simpler rules and terminology of English grammar to define the syntax of some trivial sentences.

```
<sentence> ::= <subject> <verb> <object>
<subject> ::= <article> <noun>
<object> ::= <article> <noun> | <article> <adjective> <noun>
<verb> ::= watches | hears | likes
<article> ::= a | the
<noun> ::= man | woman | bicycle | book
<adjective> ::= red | big | beautiful
```

Fig. 2.2 A trivial language

Here, we define a structure called a <sentence> as a <subject> followed by a <verb> followed by an <object>. An <object> includes an optional <adjective>. The tokens on the right-hand sides of <verb>, <article>, <noun> and <adjective> are *terminal* tokens implying that they cannot be expanded further. Tokens enclosed in angle brackets in BNF are called *non-terminal* tokens. Terminal tokens are just treated as sequences of characters. The symbol ::= separates the token being defined

from its definition. The symbol | separates alternatives. It is the *alternation* operator. The symbols ::=, |, < and > are the *metasymbols* of BNF.

This set of rules can be used to generate random "sentences". Where there are alternatives, random choices can be made. For example, starting with the non-terminal <sentence>, expanding just a single non-terminal at each step:

```
<sentence>
<subject> <verb> <object>
<article> <noun> <verb> <object>
the <noun> <verb> <object>
the woman <verb> <object>
the woman watches <object>
the woman watches <article> <adjective> <noun>
the woman watches a <adjective> <noun>
the woman watches a beautiful <noun>
the woman watches a beautiful bicycle
```

Be aware that BNF is far from being sufficiently powerful to define the syntax of English or any other natural language. And we have not considered the consequences of semantics here. For example, this simple grammar can generate sentences with little sense such as `the bicycle hears the book`.

The power of BNF is better seen in a slightly more complicated example, shown in Fig. 2.3.

```
<expr> ::= <term> | <expr> + <term> | <expr> - <term>
<term> ::= <factor> | <term> * <factor> | <term> / <factor>
<factor> ::= <integer> | (<expr>)
<integer> ::= <digit> | <integer> <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Fig. 2.3 BNF for simple arithmetic expressions

These rules define the syntax of simple arithmetic expressions using integer values, the operators +, -, * and /, together with parentheses for grouping. Expressions such as `2+3+4/5` and `2+3*(44-567)` can be generated via this set of BNF rules.

Several important points should be highlighted.

- There is no upper limit to the length of expressions that can be generated by these particular rules. This is because these rules make use of *recursion*, allowing rule expansion to continue for an arbitrary number of steps. Strict BNF has no mechanism for simple iteration and recursion is used instead. For example, <expr> is defined in terms of itself.
- Consider the generation or *derivation* of the expression `1+2*3` using this set of rules (we omit here the steps between <factor> and a literal integer value).

```
<expr>
<expr> + <term>
<term> + <term>
```

<factor> + <term>
1 + <term>
1 + <term> * <factor>
1 + <factor> * <factor>
1 + 2 * <factor>
1 + 2 * 3

Note particularly that in the expansion 1+ <term>, there is an implication that the 2*3 part of the expression is grouped as a single entity called a <term>. The BNF rules can be used to support the idea of *operator precedence* where here the precedence of the * operator is higher than the precedence of the + operator—the multiplication is "done before" the addition. This of course coincides with the rules of traditional arithmetic and algebra. The phrasing of the BNF rules allow the specification of the precedence of operators. This notion will arise repeatedly in the techniques used for compiler construction.

- Similarly, the BNF rules can be used to express the *associativity* of operators. For example, consider the generation of the expression 1+2+3. Here, the 1+2 part of the expression is grouped as a single entity called a <term>. The implication is, therefore, that the expression 1+2+3 is interpreted as (1+2)+3. The + operator is *left-associative*.
  If different associativity or precedence rules are required, then the BNF could be modified to express these different rules. It is perhaps surprising that such a simple metalanguage can do all this.

BNF has been used in the definition of many programming languages and over the years, many extensions to BNF have been proposed and subsequently used. There are several different variants of the low-level syntax of BNF-like metalanguages, but one variant became popular after its use in the ISO Pascal Standard [2]. This variant was called *Extended Backus–Naur Form* (*EBNF*). It retains the basic principles of BNF but the syntactic detail is a little different. The BNF example above can be translated easily into EBNF as follows:

expr = *term* | *expr* "+" *term* | *expr* "-" *term*.
term = *factor* | *term* "*" *factor* | *term* "/" *factor*.
factor = *integer* | "(" *expr* ")".
integer = *digit* | *integer digit*.
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".

The terminal symbols (the symbols that cannot be expanded any further) are all enclosed in double quotation marks, each production rule is terminated with a full stop and the token = (rather than BNF's ::=) separates the non-terminal token from its definition. The < and > brackets have disappeared. There are some other key additional features.

- Parentheses can be used to indicate grouping in the rule.
- There is a specific feature to indicate optionality in a rule: [X] specifies zero or one instance of X, in other words specifying that X is optional.
- Repetition (not by recursion) is supported too: {X} implies zero or more instances of X.

We can therefore write an essentially equivalent set of rules:

expr = *term* | *expr* ("+" | "-") *term*.
term = *factor* | *term* ("*" | "/") *factor*.
factor = *integer* | "("*expr*")".
integer = *digit* {*digit*}.
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".

EBNF offers a more compact and probably clearer route towards the definition of the syntax of a programming language. It ties in well with the hand construction of syntax analysers as will be seen in Chap. 4.

Finally, a brief mention should be made of the use of *syntax diagrams* in the definition of programming languages. This approach was popularised in the original definition of Pascal [3] and uses a pictorial notation to represent the syntax of each of the non-terminal tokens of the language. This too proves to be an easily understood and compact notation.

2.2.1.1 Limitations

The formal syntax of most common programming languages, expressed in the form of BNF or equivalent, will almost certainly lack a specification for some of the rules for the construction of "well-formed" programs. This is likely to result from the fact that a BNF specification cannot express contextual constraints. For example, the programming language being defined may be such that all variables have to be declared appropriately before they are used. The C statement `i = 2;` is only valid in the context of a valid declaration for `i` such as `int i;`. This constraint does not appear in the usual syntax specification for C. We will see exactly why this is a problem caused by the limitations of metalanguages such as BNF a little later in this chapter.

It is tempting, therefore, to rethink the metalanguage used for language specification so that these additional rules can be incorporated somehow. Such metalanguages do exist; an example is the two-level grammar introduced in the definition of ALGOL 68 [4]. This approach essentially uses two distinct rule sets in two different metalanguages, the first set being used to generate the second (infinite) set which in turn is used to generate valid programs in ALGOL 68. This infinite set of production rules has the effect of allowing the specification of context-dependent constraints in the language.

This and similar approaches have not proved to be popular because of their complexity. The preferred approach is to stick with the simple context-free rules of BNF, or equivalent, and rely on other sets of rules, formal or informal, to define the additional constraints. As we will see, one of the key advantages of retaining this simple form of syntax specification is that the generation of the corresponding analysis phase of the compiler can be made very simple.

2.2.2 Semantics

Unfortunately, the specification of semantics is much harder than the specification of syntax. Formal approaches may be possible, based on mathematical formalisms. These definitions may prove to be long and complex, inaccessible to the casual user of the programming language. Formal semantics opens up the possibility of *proving* program correctness and removes the possibility of semantic ambiguity. Several approaches to the specification of programming language semantics have been developed—operational semantics, denotational semantics and axiomatic semantics, basing a formal description of semantics on the language's syntax. There are many good textbooks in this area, for example see [5].

Attribute grammars can be used to help define aspects of the semantics of a programming language, allowing the specification of context-sensitive aspects by augmenting a context-free grammar. Here, grammar symbols are associated with *attributes*. These are values that can be passed to both parent and child of the grammar node in which the symbol appears. This approach allows the formal specification of the language's operational semantics (how the program is interpreted as a sequence of computational steps), supporting semantic checks such as requiring the definition of a name before its use [6].

An alternative approach is to specify semantics somewhat more indirectly by producing a *reference implementation*. Here, a particular implementation is selected to define how all other implementations should behave. A program running on any of the implementations should behave as if it is running on the reference implementation. The simplicity of this approach is attractive. However, there are potential problems that may arise because of software or even hardware errors in that reference implementation which, strictly speaking, should be followed by all other implementations.

A third approach, and an approach used widely in the specification of popular languages, is to specify the semantics using a natural language. Here, text in a natural language such as English is used to describe the semantic rules of the programming language. Care is needed to avoid omission or ambiguity and to prevent the specification from becoming overly long. There is a real danger of assuming that the semantics of programming language constructs are "obvious". This is far from being true—there are many examples of real programming language features that are often misinterpreted by the programmer and sometimes mis-implemented by the compiler writer.

This book takes the easy route and avoids issues concerned with the formal specification of semantics. There will be many semantics-related issues discussed, but using an informal (English language) notation. We, in common with many other compiler writers, follow this third approach for semantics specification.

2.3 Analysis of Programs

Before looking at practical approaches for the analysis phase of programming language translation, we have to cover just a little theory. We need a formal structure on which to base the process of analysis. It just cannot be done reliably in an ad hoc way. We need to look first at the idea of formal grammars and the notations associated with them. These grammars form the rock on which we can build code for programming language analysis.

2.3.1 Grammars

The term "grammar" has a wide range of definitions and nuances and it is hardly surprising that we need a tight and formal definition for it when used in the context of programming languages. The idea of a set of BNF rules to represent the grammar of a language has already been used in this chapter, but formally a little more is required.

The *grammar* (G) of a language is defined as a 4-tuple $G = (N, T, S, P)$ where:

- N is the finite set of non-terminal symbols.
- T is the finite set of terminal symbols (N and T are disjoint.)
- S is the *starting symbol*, $S \in N$. The starting symbol is the unique non-terminal symbol that is used as the starting point for the generation of all the strings of the language.
- P is the finite set of *production rules*. A production rule defines a string transformation and it has the general form $\alpha \rightarrow \beta$. This rule specifies that any occurrence of the string α in the string to be transformed is replaced by the string β.
  There are constraints on the constitution of the strings α and β. If U is defined by $U = N \cup T$ (i.e. U is the set of all non-terminal and terminal symbols), then α has to be a member of the set of all non-empty strings that can be formed by the concatenation of members of U, and it has to contain at least one member of N. β has to be a member of the set of all strings that can be formed by the concatenation of members of U, including the empty string (i.e. $\beta \in U^*$).

Looking back at the BNF definition of the simple arithmetic expressions in Fig. 2.3, it is easy to see that this forms the basis of the formal grammar of the language. Here N is the set {expr, term, factor, integer, digit}, T is the set {+, -, *, /, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, S is expr and P is simply the set of rules in Fig. 2.3, translating the BNF syntax into whatever syntax is used to represent the rules, traditionally ::= being replaced by $\rightarrow$. In practice, we can afford to be a bit sloppy in the definition and use of the term grammar since the specification of N, T and S are usually obvious from the BNF (or equivalent) production rules. There is a convention that the starting symbol is the non-terminal defined in the first production rule.

If BNF, EBNF or syntax diagrams are used to specify the production rules, all rules have a particular characteristic that the left-hand sides of productions are always single non-terminal symbols. This is certainly allowed by the rules defining a grammar, but this restriction gives these grammars certain important features which will be examined in Sect. 2.3.2.

A grammar gives the rules for deriving strings of characters conforming to the syntactic rules of the grammar. A *sentential form* is any string that can be derived from S, the starting symbol. And a *sentence* is a sentential form not containing any non-terminal symbols. A sentence is something final, it cannot be expanded any further. In the context of grammars for programming languages, a sentence is a complete program, containing just terminal symbols (i.e. the characters of the language).

2.3.2 Chomsky Hierarchy

Looking at the definition of a grammar in the last section, it is clear that the important and potentially problematic component is P, the set of production rules. A production rule has the form $\alpha \rightarrow \beta$, loosely translated as "anything can be transformed to anything" (although we have already stated some restrictions on the content of α and β). The key question then is to remove this generality by restricting the forms of the production rules to see whether less general rules can be useful for defining and analysing computer programming languages.

In the 1950s, Noam Chomsky produced a hierarchical classification of formal grammars which provides a framework for the definition of programming languages as well as for the analysis of programs written in these languages [7]. This hierarchy is made up of four levels, as follows:

- A *Chomsky type 0* or a *free grammar* or an *unrestricted grammar* contains productions of the form $\alpha \rightarrow \beta$. The restrictions on α and β are those already mentioned in the section above. This was our starting point in the definition of a grammar and as suggested, these grammars are not sufficiently restricted to be of any practical use for programming languages.
- A *Chomsky type 1* or a *context-sensitive grammar* has productions of the form $\alpha A \beta \rightarrow \alpha \gamma \beta$ where $\alpha, \beta, \gamma \in U^*$, γ is non-null and A is a single non-terminal symbol. The left context is α, the right context is β and in this particular context, A is transformed to γ.
  This type of grammar turns out to have significant relevance to programming languages. The concept of context is central to programming language definition—"… this statement is only valid in the context of an appropriate declaration of i …", for example. However, in practice, these grammars do not turn out to be particularly helpful because defining the context-sensitive aspects of a programming language in terms of a type 1 grammar can turn into a nightmare of complexity. So we need to simplify further and specify context-sensitive aspects by resorting to other means, such as English language descriptions.

- A *Chomsky type 2* or a *context-free grammar* has productions of the form $A \rightarrow \gamma$ where A is a single non-terminal symbol. These productions correspond directly to BNF rules. In general, if BNF can be used in the definition of a language, then that language is no more complex than Chomsky type 2.
  These grammars are central to the definition and analysis of the majority of programming languages. Despite the simplicity of the productions they are capable of defining powerful and complex program syntax. Chapters 4 and 5, in the discussion of syntax analysis, are based on this grammar type. Programming languages are generally defined using type 2 grammars and these grammars are used directly in the production of code for program analysis.
- A *Chomsky type 3* or a *regular grammar* or a *finite-state grammar* puts further restrictions on the form of the productions. Here, all productions are of the form $A \rightarrow a$ or $A \rightarrow aB$ where A and B are non-terminal symbols and a is a terminal symbol. These grammars turn out to be far too restrictive for defining the syntax of conventional programming languages, but do have a key place in the specification of the syntax of the basic lexical tokens dealt with by the lexical analysis phase of a compiler (see Chap. 3). They are the grammars for languages that can be specified using *regular expressions* and programs in these languages can be recognised using a *finite-state machine*.

These grammar types form a hierarchy, such that all type 3 languages are also type 2, 1 and 0, all type 2 languages are also type 1 and 0 and all type 1 languages are also type 0.

2.3.3 Parsing

Suppose we have a set of BNF (or equivalent) production rules defining the grammar of a programming language. We have already seen how by expanding these rules, programs can be generated. This is a simple process. We can use these grammar rules as a reference while writing programs in that language to help ensure that what is written is syntactically correct. Because the BNF specification lacks the power to define the context-sensitive aspects of the language, we will need additional advice about, for example, making sure that names are declared, that types have to match, and so on. This collection of information serves to define the programming language and should offer enough to allow the writing of *syntactically correct programs*.

The reverse process of going from a program to some data structure representing the structure and details of the program, also checking that the program is indeed syntactically correct, is unfortunately much harder. This is the process of program analysis or *parsing* and is one of the key tasks performed by a compiler.

Why is parsing so much harder? Consider a simple example based on the grammar presented in Fig. 2.3 and on its subsequent use to generate the expression `1+2*3`. Let us try using this grammar to work backwards from the expression `1+2*3` to the starting symbol <expr>. We know that this should be possible because it can be

achieved by simply reversing the steps used in its generation. Again, to simplify, we ignore the steps between a literal integer value and <factor>.

`1 + 2 * 3`	
<factor> + 2 * 3	(using <factor>→ 1)
<term> + 2 * 3	(using <term>→<factor>)
<expr> + 2 * 3	(using <expr>→<term>)
<expr> + <factor> * 3	(using <factor>→ 2)
<expr> + <term> * 3	(using <term>→<factor>)
<expr> * 3	(using <expr>→<expr> + <term>)
<expr> * <factor>	(using <factor>→ 3)
<expr> * <term>	(using <term>→<factor>)
<expr> * <expr>	(using <expr>→<term>)

At this stage we seem to be stuck, implying that `1+2*3` is syntactically incorrect. What has gone wrong?

The process of parsing repeatedly matches substrings with the right-hand sides of productions, replacing the matched substrings with the corresponding production's left-hand side. Problems arise when there is more than one substring that can be matched or *reduced* at any stage. It turns out that the choice of substring to be matched is important. By trying this parsing process again using a different set of reductions, we get a different result.

`1 + 2 * 3`	
`1 + 2 *` <factor>	(using <factor>→ 3)
`1 +` <factor> `*` <factor>	(using <factor>→ 2)
`1 +` <term> `*` <factor>	(using <term>→<factor>)
`1 +` <term>	(using <term>→<term> * <factor>)
<factor> + <term>	(using <factor>→ 1)
<term> + <term>	(using <term>→<factor>)
<expr> + <term>	(using <expr>→<term>)
<expr>	(using <expr>→<expr> + <term>)

In this case, we end with the starting symbol so that the parse has succeeded.

But how do we determine the proper set of reductions? It may be that we can reach the starting symbol via several different sets of reduction operations. In this case, we conclude that the grammar is *ambiguous* and it needs repair, either by altering the set of productions or (not so desirable) by adding additional descriptive explanation to indicate which particular set of reductions is correct. Choosing the set of reductions to be applied on a sentence is the central issue in parsing. In Chaps. 4 and 5 algorithms are proposed for tackling this problem.

2.3.3.1 The Output of the Parser

The parser obtains a stream of tokens, from the lexical analyser in a conventional compiler, and matches them with the tokens in the production rules. As well as indicating whether the input to the parser forms a syntactically correct sentence, the parser must also generate a data structure reflecting the syntactic structure of the input. This can then be passed on to later stages of the compiler.

This data structure is traditionally a tree. The *parse tree* is constructed as the parser performs its sequence of reductions and the form of the parse tree directly reflects the syntactic specification of the language. The root node of the parse tree corresponds to the starting symbol of the grammar. For example, the tree generated by running the parser on the `1+2*3` example could take the general form shown in Fig. 2.4a.

This form of tree accurately reflects the formal syntactic definition of the language, and much of the tree may turn out to be redundant. Therefore it may be adequate to generate a tree closer to the form shown in Fig. 2.4b. This is an *abstract syntax tree* where not every detail of the sequence of reductions performed by the parser is reflected in the tree. Nevertheless, the data in the tree is sufficient for later stages of compilation.

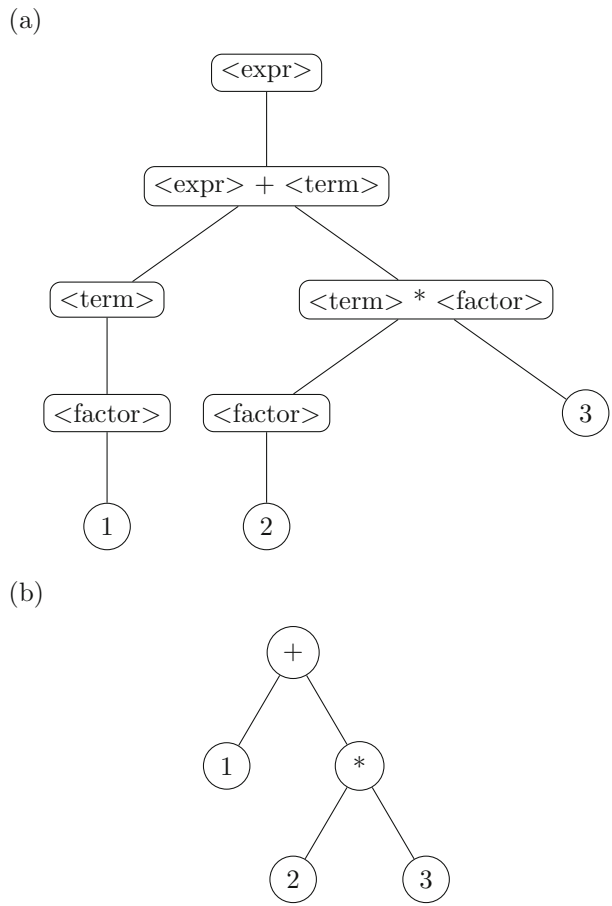


Fig. 2.4 Syntactic structure of the expression `1 + 2 * 3`

These trees and the algorithms used for their construction will be examined in detail in later chapters.

2.3.3.2 Parsing Strategies

There are two broad approaches for the construction of an algorithm for parsing. Most parsers can be classified as being either *top-down parsers* or *bottom-up parsers*. The parsing process takes the string to be parsed and repeatedly matches substrings with the right-hand sides of productions, replacing those substrings with the corresponding left-hand sides. If we start off with a syntactically correct sentence, the parsing process should transform the sentence to the starting symbol via a sequence of sentential forms. We have already seen that the choice of which reductions are made is important and a correctly written parser gets that choice right. Practical parsers rarely take this approach of repeatedly manipulating a potentially very long character string, but the principle applies.

The *top-down parser* starts with the starting symbol of the grammar and hence with the root of the parse tree. Its goal is to match the input available with the definition of the starting symbol. So if the starting symbol S is defined as $S \rightarrow AB$, the goal of recognising S will be achieved by recognising an instance of an A followed by recognising an instance of a B. Similarly, if S is defined as $S \rightarrow A|B$, then the goal of recognising S will be achieved by recognising an instance of an A or by recognising an instance of a B. The subgoals of recognising A and B are then dealt with according to subsequent rules in the grammar. When the right-hand side of a production that is being matched with the input contains *terminal* symbols, these symbols can be matched with the input string. If the matching fails, then the parsing process fails too. But if the matching succeeds then the process continues until, hopefully, all characters in the input have been matched, at which point the parse succeeds. It is hard to visualise how this top-down process corresponds to the process described above of parsing using repeated reductions on the original and then transformed input string, but the top-down parser *is* making repeated reductions, the order and choice being controlled by the structure of the set of productions. Chapter 4 examines this whole process in detail.

The *bottom-up parser* perhaps reflects a more obvious way of thinking about parsing, where, instead of starting with the starting symbol, we start with the input string, choose a substring to be matched with the right-hand side of a production, replace that substring with the corresponding left-hand side, and repeat until just the starting symbol remains (indicating success) or until no valid reduction can be performed (indicating failure). The parse tree is being constructed upwards from the leaves, finally reaching the starting symbol at the root. The key problem here is of course one of determining which reductions to apply and in which order. Again, we return to this issue in Chap. 4.

2.4 Compiler and Interpreter Structure

Having looked at some of the issues of programming language definition and analysis, we have to step back and examine the overall structure of the programming language translation process. The implementation of a programming language is potentially a huge software project. The GNU Compiler Collection (GCC) now contains well over 10 million lines of source code. This is, admittedly, an extreme example, but it does illustrate the need for good software engineering principles for compiler or interpreter projects. In order to start thinking about such a project, it is essential to consider the structure of a compiler or an interpreter in terms of a collection of logically separate modules so that a large task can be viewed as a collection of somewhat simpler tasks.

We start with the trivial view of a compiler or interpreter shown in Fig. 2.1. In the case of a compiler, the translator is generating code to run on a real or virtual machine. In the case of an interpreter, the translator is generating code which is interpreted by the interpreter program. There is no profound difference between these two approaches (a compiler can generate code that is interpreted by the hardware) and hence some of the internal structures of compilers and interpreters can be similar. In this section, we discuss specifically the modular structure of a compiler generating code for a real machine.

The first subdivision, we can make is to consider the compiler as having to perform two distinct tasks, as shown in Fig. 2.5. The analysis phase and the synthesis phase are often referred to as the *front-end* and the *back-end* respectively.

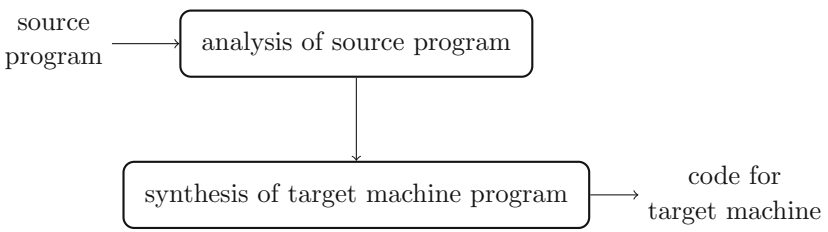


Fig. 2.5 The analysis/synthesis view of compilation

This subdivision, although simple, has major consequences for the design of compilers. The interface between these two phases is some intermediate language, loosely "mid-way" between the source and target languages. If this intermediate language is designed with care, it may then be possible to structure the compiler so that the analysis phase becomes *target machine independent* and the synthesis phase becomes *source language independent*. This, in theory, allows great potential savings in implementation effort in developing new compilers. If a compiler structured in this way needs to be retargeted to a new machine architecture, then only the synthesis phase needs to be modified or rewritten. The analysis phase should not need to be touched. Similarly, if the compiler needs to be modified to compile a different source language (targeting the same machine), then only the analysis phase needs

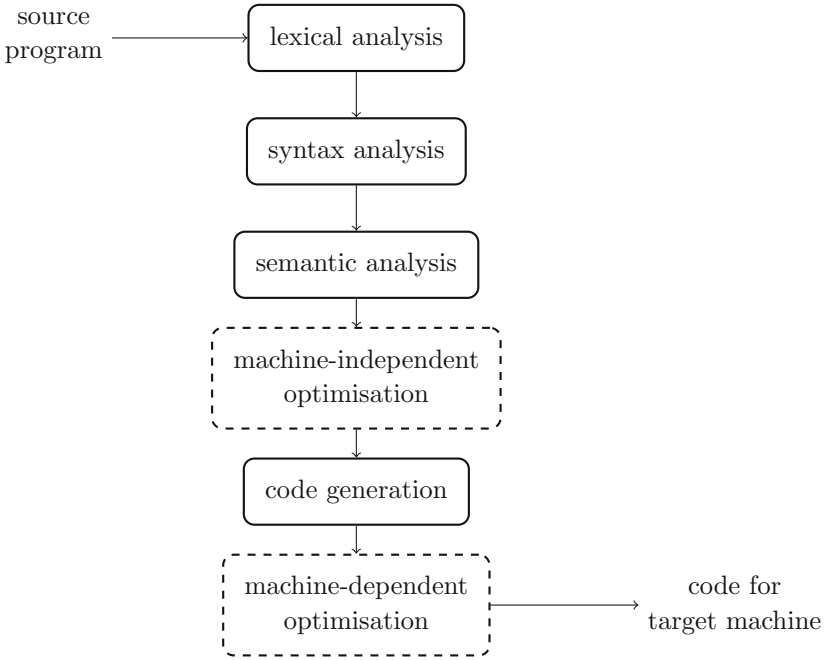


Fig. 2.6 Phases of compilation

to be modified or rewritten. We look at this issue from the view of the intermediate representation in Chap. 6.

But the subdivision into phases needs to be taken further to be of any use in the practical construction of a compiler. A traditional view of compiler structuring, referred to repeatedly in this book, is shown in Figure 2.6.

The lexical analysis, syntax analysis, semantic analysis and the machine-independent optimisation phases together form the front-end of the compiler and the code generation and machine-dependent phases form the back-end. These phases all have specific and distinct roles. And to support the design of these individual modules, the interfaces between them have to be defined with care.

2.4.1 Lexical Analysis

This first phase of compilation reads the characters of the source program and groups them together into a stream of *lexical tokens*. Each lexical token is a basic syntactic component of the programming language being processed. These are tokens such as numbers, identifiers, punctuation, operators, strings, reserved words and so on. Comments can be ignored unless the language defines special syntactic components

encoded in comments that may be needed later in compilation. White space (spaces, tabs, newlines, etc.) may be ignored except, again, where they have some syntactic significance (where spacing indicates block structure, where white space may occur in character strings and so on).

For example, this C program fragment:

```
sum = 0;
for (i=0; i<=99; i++) sum += a[i];    /* sum array */
```

will be read by the lexical analyser and it would generate this stream of tokens:

`sum` (identifier), `=`, `0` (integer constant), `;`, `for` (reserved word), `(`, `i` (identifier), `=`, `0` (integer constant), `;`, `i` (identifier), `<=`, `99` (integer constant), `;`, `i` (identifier), `++`, `)`, `sum` (identifier), `+=`, `a` (identifier), `[`, `i` (identifier), `]`, `;`

The syntax of these basic lexical tokens is usually simple, and the expectation is that the syntax can be specified formally in terms of a Chomsky type 3 grammar (i.e. in terms of regular expressions). This considerably simplifies the coding of the lexical analyser.

The output of the lexical analyser is a stream of tokens, passed to the syntax analyser. The interface could be such that the lexical analyser tokenises the entire input file and then passes the whole list of tokens to the syntax analyser. Alternatively, the tokens could be passed on to the syntax analyser one at a time, when demanded by the syntax analyser.

2.4.2 Syntax Analysis

The syntax analyser groups and structures the lexical tokens according to the syntax rules of the programming language. It performs the parsing process, as outlined above in Sect. 2.3.3, repeatedly grouping together components by performing reductions according to the production rules. Assuming that the sequence of tokens is syntactically correct, the parse should succeed. If the sequence is not syntactically correct, then the syntax analyser should report an error and then perform some appropriate recovery action.

The syntax analyser constructs a data structure representing the syntactic structure of the input. This is usually based on some form of tree where the nodes represent syntactic components defined by the grammar. This is the parse tree or abstract syntax tree. This data structure should contain or link to all the information needed by later phases of compilation. So, for example, a node corresponding to the occurrence of a constant value in the original program should contain or link to information defining that constant such as its type, value and so on.

It is clear that the lexical and syntax analysers are doing similar things. They are both grouping together characters or tokens into larger syntactic units. So there is an issue about whether a particular syntactic structure should be recognised by the lexical analyser or by the syntax analyser. The traditional approach, and it is

an approach that works well, is to recognise the simpler structures in the lexical analyser, specifically those that can be expressed in terms of a Chomsky type 3 grammar. Syntactic structures specified by a type 2 or more complex grammar are then left for resolution by the syntax analyser. In theory, the syntax analyser could deal with the lexical tokens using a type 2 grammar parsing approach, but this would add significantly to the complexity of the syntax analyser. Furthermore, by leaving the lexical analyser to deal with these tokens improves compiler efficiency because simpler and faster type 3 parsing techniques can be used.

2.4.3 Semantic Analysis

The analysis phase is not quite complete even after the syntax tree has been constructed. Traditional type 2 grammars used to build the syntax analyser cannot deal directly with contextual issues such as type checking, declaration and scopes of names, choice of overloaded operators and so on. This is the role of the semantic analysis phase. Traversing the tree, inserting and checking type information is done here. Typed languages may require that all or almost all the nodes in the tree be labelled with a data type. Complexity is increased when the language being compiled allows user-defined types. Rules for type compatibility have to be applied here too. For example, does the language allow an integer value to be assigned to a real (floating point) variable?

A second task of the semantic analysis phase is to *flatten* the parse tree to produce some form of *intermediate code*. The nature of this code is discussed in Chap. 6. It should be straightforward to generate this intermediate code by traversing the tree. The type information is preserved so that the intermediate code is functionally equivalent to the original source program. This intermediate representation can be regarded as the machine code for a carefully designed virtual machine.

2.4.4 Machine-Independent Optimisation

Generating intermediate code by simple tree traversal will yield code with opportunities for improvement. This optimisation phase, together with the optimisation performed during and after code generation, can make a dramatic difference to the quality of the code generated by the compiler. In Fig. 2.6 these phases are enclosed in dashed lines, indicating that they are optional. The compiler will still work without them, but the quality of generated code may be poor.

Post-semantic analysis is a good stage of compilation for code optimisation. The intermediate representation will have been designed with optimisation in mind and many optimisation techniques can be applied. These result, in some cases, in dramatic performance improvements. Removal of redundant code, function inlining, loop unrolling, dependence and flow analysis and so on can all be done here. This is

a target machine-independent phase because the optimisations being performed are making no assumptions about the low-level design of the target machine.

The output of this phase is a representation of the program being compiled in an intermediate form. It is likely, but not essential, that the input and the output of this phase are expressed in the same intermediate representation.

2.4.5 Code Generation

At this point, attention shifts away from the nature of the source language and moves towards the design and characteristics of the target machine. The code generation phase reads the (optimised) intermediate code and outputs functionally equivalent target machine instructions. This is easy to specify but the implementation requires the handling of complex detail.

The code generator has to select appropriate machine instructions, decide how the target machine registers are to be used, deal with a storage allocation scheme for all the variables and structures needed by the program as it runs, generate code to interface with libraries and the operating system. This is all being done in the context of the need to generate high-quality code.

2.4.6 Machine-Dependent Optimisation

At least partially incorporated into the code generation phase is the process of optimisation specifically geared towards the characteristics of the target machine. Use of special-purpose instructions and addressing modes, making use of target machine parallelism, using the target machine registers effectively and so on can make a significant impact on the quality of the target code. There are some optimisations that are best done as code is actually generated, whereas there are other techniques that are best run as a separate pass over the generated code.

There are some general issues concerning optimisation. First, the term "optimisation" is used in the compiler context in a somewhat unconventional way. It is not taken to mean "generate the *best* code possible", but instead it implies "generate *better* code". Furthermore, the aims of optimisation need to be clear when developing the compiler. Is the aim of optimisation to generate code that will run *fast* on the target machine? Or is it to generate *compact* code (maybe more appropriate for embedded systems)? Or is it some combination of the two? Is the aim to minimise the power consumption as the code runs? Managing these tradeoffs may be difficult and the aims should be clear as the optimisation phases are being developed.

The final output of the compiler is a program that can be run on the target machine, maybe after some further processing. The output may be some form of object file requiring processing by a linker or loader before it can actually run or maybe an

assembly language file requiring processing by an assembler to produce loadable target code.

2.4.7 Symbol Tables

The names (identifiers, symbols) used in the source program need to be stored during the compilation process. Further information relating to types, scope, declarations, values or locations and other source language-dependent features will need to be stored too. This information allows the compiler, for example, to ensure that variables are appropriately declared, perform type checking, generate appropriate intermediate code instructions and include symbolic names in the code generator output to allow symbolic debugging at runtime. Therefore, the symbol table in a typical compiler is a complex data structure, supporting efficient name lookup, accessible by any of the compilation phases.

Symbols may be inserted into the symbol table by the lexical analyser, but it may be better to perform this task in the syntax analyser where more context information is known. The syntax analyser can distinguish between the declaration and the use of a name and this is important when accessing the symbol table. The semantic analysis phase makes heavy use of the symbol table, and it may generate intermediate code that implicitly includes enough of the symbol table information to allow the code generation and optimisation phases to be free of the need to access the symbol table.

2.4.8 Implementation Issues

Finally, there are many practical issues to consider in designing the implementation plan of a compiler. In which programming language should the compiler or interpreter be written? What sort of testing strategy should be adopted? Are there techniques to simplify the implementation process? Are there good software tools to use? Can we make use of software that is already freely available by incorporating it in the compiler?

It may be that more software than just the compiler or interpreter needs to be written. Is there a need for a runtime debugger, linker or loader? Are there program development tools needed to be integrated with the compiler? Do we need a runtime system providing an interface between the running compiled program and the operating system and/or hardware? The task can easily get out of control but there are many standard implementation routes, some of which are examined in Chap. 9.

2.5 Conclusions and Further Reading

This chapter has shown that the design of the implementation of a programming language is by no means a trivial task but has also shown that the task may have become tractable by imposing a solid structure on which an implementation can be built. Good planning is vital in this sort of project especially if a team is involved. Starting from accurate definitions is essential. Formalising and automating the process makes the production of a reliable implementation so much more straightforward.

Compilers and interpreters are now very rarely constructed from scratch. Making use of already available software may well be crucial to make the project feasible. The separation of the compilation process into clearly defined modules and the use of standardised or pre-existing interfaces makes this process of modular construction very much easier.

This may be completely obvious, but it is worth stating nevertheless. It pays to start off with an excellent, in-depth knowledge of both the source language and the target machine. Conversely, writing a compiler may be one of the best ways to learn a programming language and a target machine!

The documents formally defining programming languages form a valuable resource for anyone involved in the task of language implementation. Early language definitions, such as the FORTRAN standard [8], are well worth examining to see how far we have come in both programming language design and also in techniques for programming language definition. The ALGOL 60 definition [1] is also a key historical document, particularly for its use of BNF. It is also worth taking a look at the definition of ALGOL 68 [4], again a key historical document but it clearly shows the importance of having an *accessible* language definition. The definitions of most more modern languages are easily found on the web, some simple and others of astonishing complexity.

Routes towards programming language implementation are sometimes complex. This is usually and paradoxically the case because of the need to reduce the amount of programming effort required. Understanding the stages required in such implementations is often difficult and *T-diagrams* were introduced as a simple visualisation mechanism [9, 10]. Deciding on whether to compile or interpret is a key question and picking out the language features (such as reflection in Java) that push towards an interpreted implementation is a helpful task.

A great deal has been written about the design of virtual machines (for example, see [11]) and documents easily accessible via the web provide designs for general-purpose and domain-specific virtual machines. The *Java Virtual Machine* (see [12]) is perhaps the most famous virtual machine and has been used in the implementations of a wide range of programming languages.

This is not a textbook about the more formal aspects of grammars and parsing. There are so many high-quality published resources in this area. A classic text is [13] and much useful background information is contained in [14, 15].

Finally, an excellent source of design information is the source code and documentation of existing compilers and interpreters. For example, the GNU GCC project is

well documented and the compiler source code is freely available. A good indication of the functionality offered by the compiler is given by the range of options available when running the compiler.

Exercises

2.1. The Java Virtual Machine has been used as a route to the implementation of many programming languages. Produce a list of some of these languages and implementations. Why was the JVM often chosen? Are there programming language features that do not map well onto the JVM?

2.2. Suppose you had to write a program to count the number of `if` statements used in a C program. Explain why the obvious approach of counting the number of matches with the character string "if" may produce the wrong answer. Why can the use of lexical analyser techniques help? Are aspects of syntax analysis required too?

2.3. Write a program to read grammar productions expressed in BNF or EBNF and generate random sentences from the grammar. To make the sentences more interesting it may help to be able to alter the relative probabilities of choice where alternatives are specified by the grammar.
Try this program on the grammar of a real programming language and if possible put it through a compiler for that language. Did you expect it to compile?

2.4. Write the syntax of BNF in EBNF (and the other way round).

2.5. Produce the grammar for a simple language specifying conventional arithmetic expressions involving integer constants, brackets and the four binary operators `+`, `-`, `*` and `/`. Make sure that expressions of the form `1+2*3` are interpreted correctly by the grammar. Then extend the grammar to allow the unary operators `+` and `-`. Make sure that the grammar correctly interprets expressions of the form `-1-2`. Maybe produce an implementation, although this will be *much* easier once material in later chapters has been read!

2.6. Produce a grammar for simple arithmetic expressions with unconventional rules of precedence so that, for example, the expression `3*2+1` is interpreted as `3*(2+1)`.

2.7. Check how your grammar interprets expressions of the form `1-2-3`. Change the grammar to make all four operators right-associative so that the value of `1-2-3` is `2` rather than `-4`.

References

1. Naur P (1960) Report on the algorithmic language ALGOL 60. Commun ACM 3(5):299–314
2. Jensen K, Wirth N (1985) Pascal user manual and report – ISO Pascal standard, 3rd edn. Springer, New York
3. Jensen K, Wirth N (1975) The Pascal user manual and report, 2nd edn. Springer, New York

4. van Wijngaarden A, Mailloux BJ, Peck JEL, Coster CHA, Sintzoff M, Lindsey CH, Meertens LGLT, Fisker RG (1975) Revised report on the algorithmic language ALGOL 68. Acta Inform 5:1–236
5. Winskel G (1993) The formal semantics of programming languages. The MIT Press, Cambridge
6. Cooper KD, Torczon L (2011) Engineering a compiler, 2nd edn. Morgan Kaufmann, San Francisco
7. Chomsky N (1956) Three models for the description of language. IRE Trans Inf Theory 2:113–124
8. United States of America Standards Institute, New York (1966) USA Standard FORTRAN – USAS X3.9-1966
9. McKeeman WM, Horning JJ, Wortman DB (1970) A compiler generator. Prentice Hall, Englewood Cliffs
10. Terry PD (1986) Programming language translation: a practical approach. International computer science series. Addison-Wesley Publishing Company, Reading
11. Wilhelm R, Seidl H (2010) Compiler design: virtual machines. Springer, Berlin
12. Lindholm T, Yellin F (1997) The Java virtual machine specification. The Java series. Addison-Wesley, Reading
13. Hopcroft JE, Ullman JD (1979) Introduction to automata theory, languages and computation. Addison-Wesley Publishing Company, Reading
14. Aho AV, Lam MS, Sethi R, Ullman JD (2007) Compilers - principles, techniques and tools, 2nd edn. Pearson Education, Upper Saddle River
15. Mogensen TÆ (2011) Introduction to compiler design. Undergraduate topics in computer science. Springer, Berlin

Chapter 3
Lexical Analysis

It is appropriate to start the details of compiler implementation by considering the lexical analyser. The place of the lexical analyser in the complete compiler has already been discussed in Chap. 2. Because it is the first phase of source code analysis, the format of its input is governed by the specification of the programming language being compiled. The output of the lexical analyser has to satisfy the needs of the next phase of compilation (syntax analysis) and details of this interface will be examined later in this chapter.

Making a clear separation between lexical analysis and syntax analysis is important. Regarding and implementing them as separate modules has several advantages. It increases the degree of *modularity* of the compiler—the principles of software engineering emphasise that this is a good thing. It makes the *implementation* easier with the coding of the syntax analyser not having to worry about the low-level details of the lexical structure of the source language. This is particularly noticeable when attempting to produce a formal definition (for example, in BNF) of a programming language to include lexical tokens, optional white space between tokens, and so on. Implementation from this inevitably complicated grammar may be difficult. Furthermore, leaving the more complex syntactic structures for the syntax analyser makes the coding of the lexical analyser much easier. *Debugging* is helped by this separation too. For example, if the examination of the interface between the two phases indicates that the tokens generated by the lexical analyser are correct, the problem should lie in the syntax analyser. Finally, the separation can offer benefits of *efficiency* since the algorithms used to analyse tokens within the lexical analyser are simpler than those used to analyse syntactic structures in the syntax analyser. Simpler grammars, specifically Chomsky type 3 grammars, can be used to define the syntax of lexical tokens. These efficiency gains in terms of both execution time and space requirements may not be of paramount importance for today's hardware but they have certainly influenced the historical development of compiler design.

This chapter examines two general approaches to lexical analyser design, presenting some code examples and highlighting some of the pitfalls. The first approach shows how a hand-crafted lexical analyser can be structured by following some simple guidelines. The second approach follows a more formal path, starting with a

syntactic specification of the lexical tokens. This specification, expressed in the form of regular expressions, can always be transformed via the construction of finite-state automata to the code for a lexical analyser. We concentrate on the practicalities of lexical analyser construction, including only just a little background theory.

These approaches have their individual advantages and disadvantages and in some cases there are aspects of the language being analysed that cause difficulty for both approaches. The use of a specification based on regular expressions forms the basis of a range of software tools that can help greatly in the construction of a lexical analyser.

Writing a lexical analyser for simple languages usually turns out to be a uncomplicated task. Code complexity is low because the lexical analyser should not need to store much state information as it runs. The analysis can usually be done essentially in a single pass with very little or no backtracking. However in a real compiler, efficiency may be particularly important and this may influence the algorithms used for lexical token recognition.

The lexical analyser is constrained by the syntax of the source language and by the details of the stream of tokens it has to pass to the next phase of compilation—the syntax analyser. So it makes sense to look next at the nature of these lexical tokens.

3.1 Lexical Tokens

The role of the lexical analysis phase of a compiler is to read the program being compiled, breaking up the input into a sequence of *tokens*. Each token represents a basic syntactic component of the language being compiled. For example, a compiler for C would be recognising tokens such as reserved words, identifiers, numerical constants, single character tokens such as '+', multiple character tokens such as '<=', strings, character constants, and so on. Some of the characters read from the input can be thrown away by the lexical analyser. For example, white space (spaces, tabs, newlines, etc.) may, in most but not all circumstances, be ignored. In general, comments can also be ignored.

3.1.1 An Example

Consider this fragment of a C program:

```
while (i <= 100) {
   tot += a[i];   /* form vector total */
   i++;
}
```

The lexical analyser would read this input combining adjacent characters to form syntactically correct tokens, ignoring irrelevant input and passing a stream of tokens to the next phase of compilation. In this case, the tokens generated would be

`while` (reserved word), `(`, `i` (identifier), `<=`, `100` (integer constant), `)`, `{`, `tot` (identifier), `+=`, `a` (identifier), `[`, `i` (identifier), `]`, `;`, `i` (identifier), `++`, `;`, `}`

This simple example raises some important issues.

- For many programming languages, the lexical analyser can easily distinguish between identifiers (or other names) and reserved words. This is because in these languages the language specification simply lists a set of reserved words (such as `while` in C) that cannot be used as identifiers. We will look later at more awkward languages where the rules are not so straightforward and context has to be taken into account.
- White space and comments have not been passed on as tokens. It is tempting to say that white space and comments are *ignored* but this is not strictly accurate. There are circumstances where white space must be used in order to separate tokens. For example, in the declaration `int a;` the space character must not be ignored. In other circumstances, such as `a+1`, white space is not necessary to separate the tokens and if white space appears, it can be ignored.
- There are three cases here of two-character tokens where there is potential for confusion in their interpretation. These tokens are <=, += and ++. We are interpreting each of these to be single tokens although, for example <= could be considered incorrectly as < followed by =. Doing the work of combining these character pairs in the lexical analyser rather than in the syntax analyser makes good sense, and again, we will be returning to this issue later in this chapter.
- The lexical analyser makes no attempt to verify that this example contains a well-formed `while` statement. The lexical analyser does not need to worry whether the stream of tokens it is generating is syntactically correct because that is done later in the compiler. Beginners often try to do too much in the lexical analyser and attempt to include tests on the sequencing of tokens that should be left for the syntax analyser. It is important to realise the limits of the tasks of the lexical analyser.

Hopefully at this stage the coding of a lexical analyser should not feel too daunting and there may be a temptation to tackle it straight away. However, there are some potential pitfalls in the process and it is worth taking the time to examine these before routes to implementations are described.

3.1.2 Choosing the List of Tokens

When designing a lexical analyser the first task is to use the specification of the input language to help produce a list of all the lexical tokens that the lexical analyser should be recognising, together with their (preferably formal) definitions. The language

definition is unlikely to provide a neatly separated list of lexical tokens. The choice of tokens is largely the responsibility of the language implementer.

As an example, here is an outline of a list of tokens for writing a lexical analyser for C. Further detail is clearly required.

- Identifiers, starting with a letter followed by an arbitrary sequence of letters or digits or underscores, such as `a`, `Ab123`, `next_one`.
- Integer constants, such as `12345`. The C language definition allows a wider range of integer constant types such as octal (`0123`) and hexadecimal (`0x12ff`) constants, and optional suffices to specify that the constant is *unsigned* and/or *long*.
- Character constants also can take several forms. The simple form is a single character enclosed between single quote marks (`'a'`). Escape sequences can be used (`'\n'`) and characters can be expressed as octal or hexadecimal values.
- Floating constants too can be expressed in several ways. The traditional representation is an integer part followed by a full stop followed by a fractional part (`123.45`), but an exponent can be added (`1.234e10`). The *type* of the constant (`float`, `double`, `long double`, determining its internal representation) can also be specified.
- String constants are sequences of characters enclosed by double quotes (`"Hello"`). There are also minor complications here with the inclusion of escaped characters within the string, the use of an extended character set and the automatic concatenation of adjacent strings.
- The operator tokens are numerous but somewhat simpler in structure. They take the form of a single character (such as `+`, `*`, `(`, `=`, `<`, `?`) or a pair of characters (`<<`, `+=`, `++`, `<=`) or in a few cases three characters (`...`, `<<=`).
- Keywords and reserved words have special meanings in the syntax of the language. Reserved words cannot be used as identifiers. Examples from C include `while`, `int`, `if`, `typedef`. Reserved words like these are assumed to be keywords of the language. But some languages allow keywords that are not reserved and it is the context of their use which determines whether or not they are being used as a keyword. This issue is discussed later in this chapter.
- Finally, although not tokens as far as the rest of the compiler is concerned, comments have to be parsed by the lexical analyser so that they can be ignored. So, for example, the lexical analyser recognises the pair of characters `/*` and continues skipping the input until encountering the character following the character pair `*/`. Similarly, *white space* (spaces, horizontal and vertical tabs, newlines and formfeeds) can also be ignored, except that it can be used to separate tokens.

In some respects, this list of tokens is simple. The number of token types is comparatively small and none of them appear to have a particularly complex syntax. But there *is* complexity here and dealing with this detail can cause serious implementation difficulties. This complexity largely arises from the fact that C is a real programming language, requiring extensive functionality and expressivity for its use in diverse application areas. Other popular programming languages are even more complex, even at the lexical token level, and some of these complications will be examined later in this chapter.

This complexity can be tamed by an understanding of the detail, requiring access to an accurate and preferably formal definition of the language. And this can help to achieve a lexical analyser implementation that is well structured and more likely to be correct.

In Chap. 2 the use of a Chomsky type 3 or regular grammar for the specification of lexical tokens was described. This results in a compact, clear and accurate statement of the syntax of these tokens. But the key advantage as far as we are concerned here is that there are clear methods to take these formal specifications and generate a lexical analyser directly from them, as shown later in Sect. 3.4. This implies that there is considerable benefit in ensuring that all the lexical tokens should be definable in terms of regular expressions. For example, the syntax of all the lexical tokens for C presented above can be expressed in terms of regular expressions. More complex syntactic structures and in particular nested structures cannot be represented in terms of regular expressions and are therefore not appropriate for handling within the lexical analyser. Instead, they are tackled by the more powerful analysis algorithms in the syntax analyser.

3.1.3 Issues with Particular Tokens

Once we have this list of lexical tokens, precisely defined, we can prepare for an implementation by looking at some of the things that can go wrong.

The concept of an *identifier* or *name* is widespread in programming languages. Often identifiers are described as "starting with a letter, followed by a sequence of letters or digits". We need to know more. For example, we should ask detailed questions about the programming language we are trying to compile:

- What is the set of characters that can appear in an identifier? Is this set restricted for the first character of the identifier? For example, does the language allow underscore characters (_) in an identifier and can that character appear at the start of an identifier? Can characters from an extended character set such as Unicode appear? Is white space allowed in an identifier?
- What about the case of letters? Are both upper and lower case letters allowed? Are upper case letters translated automatically to lower case (or vice versa), making identifiers such as `ABC123` and `Abc123` equivalent?
- Is there any limit on the length of identifiers? This will have an effect on the way in which identifiers are stored within the compiler. For example, the rules may state that "an identifier can be arbitrarily long but only the first six characters are significant", implying that, for example, `abc123x` and `abc123y` are both allowed but are equivalent.

Context sensitivity issues arise in a lexical analyser when the nature of a lexical token may depend not only on its composition in terms of characters but also on its

context. For example, in C, identifiers declared by the `typedef` declaration may need special treatment in the lexical analyser. Suppose `t` is defined as

```
typedef int t;
```

and later in the program the code fragment `(t*)` appears. This is valid in C, being a type cast operation. One would expect the lexical analyser to return the tokens *open bracket*, *identifier*, *asterisk*, *close bracket* and these tokens would be dealt with appropriately by the syntax analyser. But the grammar of C could also interpret this sequence of tokens as a malformed expression (a multiplication without the second operand). To avoid this ambiguity, the lexical analyser may be expected to return the token *type identifier* instead of *identifier* when `t` has already been defined in a `typedef` declaration. And to do this, there must be some communication back from the lexical analyser to the syntax analyser to allow the lexical analyser to determine the nature of `t`. This is messy, and it is a consequence of the way in which the C grammar is defined. We will revisit this problem in Chap. 5.

Integer and floating point constants may appear fairly straightforward but there are some important issues to address.

- Check the syntax, and ensure that all possible forms of numerical constants are being dealt with appropriately. It depends on the details of the language being analysed but handling a leading `+` or `-` as being part of the constant is rarely a good idea. If the language allows hexadecimal constants, are there restrictions on the case of the letters `a` to `f`?
- Are there restrictions on the length or numerical range of constants? Are both `1` and `123456789012` acceptable? Is there a rule which says that there is a limit on the range of integer constants, but if an out of range constant is input, there should be an automatic conversion to floating point format. And this leads on to …
- Reading and analysing numeric constants for most languages will require lookahead. For example, the input of `12345678.1` will require `12345678.` to be read before the lexical analyser can determine whether this is an integer or a floating point number. This issue is examined later in this chapter when implementation techniques are described.
- If there is a limit on the range of integer constants then the lexical analyser should really check for out of range constants. For example, Java specifies that the largest integer literal is `2147483648` (that's 2^{31}) when preceded by a unary minus and `2147483647` ($2^{31} - 1$) otherwise. Code to check this in a lexical analyser is not trivial.
- Floating point constants may have a more complex syntax with a range of alternative representations. Writing a recogniser by hand may be hard work so generating recognising code automatically from a formal specification may be an easier and safer route.

The *operator tokens* are often just single characters and pose little difficulty. But with multi-character operator tokens, the lexical analyser recognising code has to

combine the characters as appropriate. So, for example, the input ... <=... should be, if that is what the language definition says, a single "less-than-or-equal" token rather than a "less-than" token followed by an "equals" token. But with the input ... < =... (with a space between the two characters), does the space force the recognition of two separate tokens, or is the space ignored? The language definition should tell you.

Keywords and *reserved words* should be fairly easy to deal with. Again, does the case of the letters matter? In languages where all keywords are reserved words (where identifiers and reserved words cannot be confused), they can be dealt with by the lexical analyser regarding them as special cases of identifiers—read a token that can be an identifier or a reserved word and consult a table of reserved words to determine whether or not the string is a reserved word. If instead a lexical analyser is being written for a language where a keyword can be an identifier in certain contexts, then the identity of the keyword must be passed on to the syntax analyser where context information is available and a decision on the nature of the token can be made.

Comments appear in programming languages in a wide variety of forms. Some languages support comment markers where the marker and all that follows to the end of the line can be ignored. Comment brackets are also often found, where the comment is enclosed between a pair of matched tokens. In theory these comment brackets could support the notion of nested comments but this rarely offered, for good reason: if the syntax of lexical tokens is being specified in terms of a Chomsky type 3 grammar, then nested/recursive lexical structures cannot be supported. For example, comments in C cannot be nested and it *may* be helpful for the lexical analyser, on encountering the character sequence `/*` within a comment, to issue a warning. This does not immediately indicate a C syntax error, but may be a useful warning and help the resolution of trouble ahead.

Can comments be inserted *anywhere* in the source program? Some languages may impose restrictions by stating that a comment should be replaced by a single space character in the lexical analysis process. For example, in C, is `verylong/* comment */identifier = 0;` valid?

Some language implementations support special comments having a particular syntax with the purpose of controlling compilation options. For example, the programming language and/or implementation may support a form of comment that tells the compiler to turn on or off the generation of special code to perform array bound checking in that selected part of the source program. Representing these special comments as lexical tokens and passing them on to later stages of compilation is the appropriate action.

The role of *white space* is not the same in different programming languages. In some languages, white space can largely be ignored and this is trivially done in a lexical analyser. But there may be contexts, such as in character strings, where it would be wrong to ignore white space. White space is often used as a token separator or terminator.

In other languages, white space has a syntactic significance. For example in Python the indentation (i.e. white space at the beginning of a line) denotes block structure,

in a similar style to the { and } tokens of C. Indenting starts a block and unindenting ends it:

```
n = 2
print 'n =', n
if n>1:
    print 'Greater'
    print 'than 1'
else:
    n = 1
print 'Finally n =', n
```

The lexical analyser can deal with such indentation by noting whenever there is a change in indentation level and generating appropriate "start block" and "end block" tokens.

3.1.4 Internal Representation of Tokens

Now that the constraints placed on the lexical analyser by the syntax of the source language have been outlined, it is time to look at the structure of the stream of tokens it has to pass to the next phase of compilation—the syntax analyser. For each token read by the lexical analyser the syntax analyser has to be able to determine the token's identity (it is a *numerical constant*, an *identifier*, a *plus symbol* and so on) and in the case of some tokens, the "value" of the token (the value of the numerical constant, the string of characters making up the identifier, etc.). A simple approach which fulfils these requirements is to represent tokens by their type or identity together with a string containing the characters of the token. The technique used to represent the identity of the token depends on the features offered by the implementation language. This could be some sort of enumerated type or symbolic constant, typically resulting in an integer value being used to specify the identity of the lexical token. Including this identity information in the data structure representing the token is important because it saves the syntax analyser from having to re-analyse the string representations of the tokens.

The handling of constants, particularly numerical constants, needs special care. On recognising a numerical constant, the lexical analyser has to return an indication of the type of the constant (*integer constant*, *floating point constant*, etc.) together with the value of the constant. The obvious way of returning the value is for the lexical analyser to convert the string of characters representing the constant into a standard binary form (for example, a 32-bit or maybe 64-bit 2's complement integer or a 32 or 64-bit IEEE 754 floating point representation), passing it on to the next phase of compilation. However, in some circumstances, this approach may be inappropriate. Consider a compiler for a language whose definition states that the integer constant range is determined by the architecture of the target system, and the target system uses

64-bit integers. If the compiler is running on a 32-bit machine, the lexical analyser would have to make special provision to pass 64-bit integers to subsequent phases of compilation. In particular the lexical analyser's code would be affected by the architecture of the target machine. But in Chap. 2 we saw the huge benefits of having a target machine-independent front-end. An easy resolution is to delay the conversion of the text representation of the constant into some binary internal form until much later in the compilation process, where target machine dependencies are already being dealt with. So numerical constants can be returned from the lexical analyser represented as a pair—the type of the constant and its text (string) representation.

Conversion from a text representation to a binary representation is fairly easy for integers, but accurate conversion of floating point values is much harder [1, 2].

The compiler uses the *symbol table* to store information about identifiers. So it is important to consider whether the lexical analyser should have the responsibility for placing identifiers into the symbol table. To some extent this is an issue influenced by the design of the source language. But in practice the manipulation of symbol table data is only feasible when full context information is available, for example distinguishing between the appearance of an identifier in a declaration and its appearance in an expression. This is most easily done later in the compiler, typically in the syntax analyser.

Source line numbers can be included in the data returned by the lexical analyser. This is easy to do and can make the later production of accurate and helpful error messages much easier.

3.2 Direct Implementation

The previous section outlined the nature of lexical tokens in the source language and made suggestions about the representation of tokens emitted by the lexical analyser. To construct a correct and reliable lexical analyser, we need some structure on which to hang the details of the analysis of individual token types. We also need a clear and unambiguous definition of the structure and meaning of each lexical token.

This is a practically oriented book about compiler construction with the aim of presenting techniques that will work. There are then two broad approaches to lexical analyser construction. The first is essentially just to tackle the problem directly by coding in an appropriate implementation language. Code is written that reads the input, matches strings according to the syntax of the lexical tokens and passes those tokens on. The lexical analysers in many compilers are written in this way. The second approach takes a more formal view of lexical tokens by working directly with the syntactic definition of the tokens, expressed as regular expressions. Later in this chapter, we will see how regular expressions can be converted into recognising code, thus forming the main part of the lexical analyser. This transformation from regular expressions to code is best performed by software and there are now many lexical analyser generator packages available, enabling a complete lexical analyser to be generated from a formal specification of the tokens. The use of these tools

is described in Sect. 3.4. Hybrid approaches are of course possible too, where part of the lexical analyser can be written by hand and the rest generated from formal specifications (such as the code to recognise more complex lexical tokens).

We look first at a direct implementation of lexical analysers by programming them from scratch and we start with a plan for an overall structure.

3.2.1 Planning a Lexical Analyser

It is convenient to regard the lexical analyser as a function (or procedure or method), callable by the syntax analyser. This enables the syntax analyser to call for the next token from the input. A natural approach to the source code analysis is to read the entire source file into a buffer directly accessible by the lexical analyser and then make maybe multiple passes over any part of the buffer to determine where each token starts and ends, in effect tokenising the entire input. This demands the allocation of a buffer sufficiently large to hold the entire source program together with another buffer to hold the identities of all the tokens. Having the entire source program directly available makes tokenising and error reporting somewhat easier. The more common alternative is for the lexical analyser to read characters from the input as they are required. In the discussion that follows we will assume that we are following this conventional approach. Each time the lexical analyser is called it will examine characters from the input. Once the identity of the token has been recognised, the lexical analyser can read to the end of the token, storing it for return to its caller. It returns the identity of the token and where appropriate the text of the token itself, and is then in a state ready to accept the next call.

Turning now to a practical example, we can start the design of a lexical analyser for the DL language described in the appendix. Our C implementation of this lexical analyser has the following characteristics:

- The set of possible lexical tokens returned by the lexical analyser is defined as an `enum` type, and so an individual token is represented as an `int` value, returned by the `lex` function.
- To make the code a little simpler, the lexical analysis function, `lex`, returns a single integer value rather than a structure containing in addition the source line number and the string representation of the token. The only tokens that require additional information to be passed on to the next phase of compilation are those representing identifiers and numeric constants. We can achieve all we need here by maintaining a couple of global variables, one storing the string representation of the last identifier read (`identifier`) and the other storing the last numerical value read (`ival`).
- To make the coding of the lexical analyser manageable, we will make use of a *one character lookahead*. Here, we adopt the rule that each time the lexical analyser is entered via the `lex` function, the next character of the input to be analysed has

already been read and is stored in a global variable. This removes the need for backtracking in the lexical analyser and makes its coding very much simpler.

So the key declarations in our lexical analyser look something like this:

```
#define MAXIDLEN 30

typedef enum {
  elsesym, ifsym, intsym, printsym, readsym, returnsym, whilesym,
  semicolonsym, commasym, opencurlysym, closecurlysym,
  opensquaresym, closesquaresym, openbracketsym, closebracketsym,
  dividesym, multiplysym, plussym, minussym, assignsym,
  eqsym, nesym, ltsym, lesym, gtsym, gesym, eofsym,
  constantsym, identifiersym, errorsym} lextokens;

lextokens lex();

char identifier[MAXIDLEN+1];
char ch;
```

The definition of the constant `MAXIDLEN` provides an upper limit on the length of an identifier and makes it possible to use fixed length arrays for identifier storage (we are assuming that DL allows us to place some implementation restriction on identifier length). The `enum` declaration lists the complete set of lexical tokens. The order is not significant, neither is the actual assignment of names to integer values which is done automatically. The code for the function doing the lexical analysis (`lex()`) appears later. This is just the prototype declaration indicating that the function takes no arguments and returns an `lextokens` result, selected from the values declared in the `enum` type declaration. Finally we have the global declarations for the last identifier read and for the single character lookahead.

3.2.2 Recognising Individual Tokens

The next step in the coding of a lexical analyser by hand is to tackle the recognition of the individual token types. On entry to the lexical analysis function, the global variable `ch` contains the next character to be analysed and it will typically be the first character of the next token to be returned.

3.2.2.1 White Space

In DL, *white space* (spaces, tabs, newlines) can be inserted freely between lexical tokens. So the first action of the `lex` function has to be to skip over these white space characters so that `ch` contains a "significant" character:

```
while (white(ch)) ch=getchar();
```

For the majority of programming languages it is not feasible nor sensible to rely on the existence of white space to separate all tokens.

3.2.2.2 Single Character Tokens

At this stage, the value of `ch` will determine, but not quite uniquely, the nature of the token being recognised. So the main body of the `lex` function consists of an
`if .. then .. else if .. then .. else if ...`
structure or a
`switch .. case .. case ...`
construct. In this implementation, we will use the `switch` statement.

Many of the lexical tokens of DL are single characters, so the next step is easy:

```
switch (ch) {
  case ';': lextoken = semicolonsym; ch = getchar(); break;
  case ',': lextoken = commasym; ch = getchar(); break;
  case '+': lextoken = plussym; ch = getchar(); break;
  case '-': lextoken = minussym; ch = getchar(); break;
  case '/': lextoken = dividesym; ch = getchar(); break;
  case '*': lextoken = multiplysym; ch = getchar(); break;
  case '(': lextoken = openbracketsym; ch = getchar(); break;
  case ')': lextoken = closebracketsym; ch = getchar(); break;
  case '{': lextoken = opencurlysym; ch = getchar(); break;
  case '}': lextoken = closecurlysym; ch = getchar(); break;
  case '[': lextoken = opensquaresym; ch = getchar(); break;
  case ']': lextoken = closesquaresym; ch = getchar(); break;
  case EOF: lextoken = eofsym; break;
```

Note the use of `ch = getchar();` to maintain the one character lookahead before exiting from the `lex` function. But there is no need for lookahead after finding end of file because there are no further characters left and the lexical analyser can assume that the rest of the compiler will not attempt to call `lex` again after the input has all been read. If `lex` is called again, it will just return end of file again. Also, note that the code to deal with the token `'/'` will need to be redesigned because the character `/` can start a comment. This is dealt with in Sect. 3.2.2.6.

3.2.2.3 Other Short Tokens

Looking at the syntax specification for DL, it is easy to see that there are a few more single character tokens, but since they can also start a double character token, the recognising code has to be slightly more sophisticated. Specifically, we have to deal

with the tokens <, =, >, <=, ==, >= and !=. The one character lookahead helps here.

```
cases'<':
  ch = getchar();
  if (ch=='=') {
    lextoken = lesym;
    ch = getchar();
  }
  else lextoken = ltsym;
  break;

cases'>':
  ch = getchar();
  if (ch=='=') {
    lextoken = gesym;
    ch = getchar();
  }
  else lextoken = gtsym;
  break;

cases'=':
  ch = getchar();
  if (ch=='=') {
    lextoken = eqsym;
    ch = getchar();
  }
  else lextoken = assignsym;
  break;

cases'!':
  ch = getchar();
  if (ch !='=') {
    fprintf(stderr,"***Error - expected = after ! (%c)\n",ch);
    lextoken = errorsym;
  }
  else {
    ch = getchar();
    lextoken = nesym;
  }
  break;
```

Looking first at the code to recognise < and <=, it can be seen that once a < character is found, the next character is read and if it is not = there is no need to call `getchar()` again because *the lookahead has already been done*. The code for recognising `!=` is very similar, except that we know that a = character *must* directly follow the ! character. The code here complains if the = is not found and an error token is returned for the pair ! and the character following it (not =). Note that white space between the two characters is not allowed.

3.2.2.4 Identifiers and Reserved Words

The order in which tokens are recognised is not particularly important. We can tackle identifiers and reserved words next.

```
case 'a': case 'b': case 'c': case 'd': case 'e': case 'f':
case 'g': case 'h': case 'i': case 'j': case 'k': case 'l':
case 'm': case 'n': case 'o': case 'p': case 'q': case 'r':
case 's': case 't': case 'u': case 'v': case 'w': case 'x':
case 'y': case 'z':
  {
    int i = 0;
    while (islower(ch) || isdigit(ch)) {
      if (i < MAXIDLEN) identifier[i++] = ch;
      ch = getchar();
    }
    identifier[i] = '\0';

    if (strcmp(identifier,"else")==0) lextoken = elsesym;
    else if (strcmp(identifier,"if")==0) lextoken = ifsym;
    else if (strcmp(identifier,"int")==0) lextoken = intsym;
    else if (strcmp(identifier,"print")==0) lextoken = printsym;
    else if (strcmp(identifier,"read")==0) lextoken = readsym;
    else if (strcmp(identifier,"return")==0) lextoken = returnsym;
    else if (strcmp(identifier,"while")==0) lextoken = whilesym;
    else lextoken = identifiersym;
    break;
  }
```

Identifiers and reserved words in this language have to start with a lower case letter but can continue with lower case letters or digits. The string of characters of the identifier or reserved word is accumulated in the array `identifier` and there is a simple check for string equality with the various reserved words of DL. When there are many reserved words to check it would make sense to use a more efficient algorithm to determine whether a string is a reserved word, but this approach is fine for now. Note that upper case letters do not feature at all in DL. The identity of the lexical token just recognised is stored in the variable `lextoken` and this value will be passed back to the caller as the result of the call to the `lex` function.

In this code, identifiers longer than `MAXIDLEN` characters are silently *truncated*. This may not be a sensible or correct thing to do in all circumstances.

3.2.2.5 Integer Constants

In a first attempt, integer constants are handled in a standard way by converting a stream of decimal digit characters into a binary internal `int` representation:

```
case '0': case '1': case '2': case '3': case '4': case '5':
case '6': case '7': case '8': case '9':
```

```
    {
      ival = 0;
      while (isdigit(ch)) {
        ival = ival*10 + ch - '0';
        ch = getchar();
      }
      lextoken = constantsym;
      break;
    }
```

It would also be possible to use the C library function `atoi` by passing it a string of the decimal digit characters accumulated in an array, just as was done in the case of reading an identifier. Recall that `ival` is a global variable containing the value of the last integer constant read.

However, there is an important problem here. What happens if the number read by this code is larger than can be represented in an `int` variable in C, our compiler's implementation language? And how does this interact with any limits placed on the maximum value of an integer in DL? This issue was introduced in Sect. 3.1.3 and the resolution requires knowing the language's specification. Some languages specify that numerical limits are implementation-defined, probably influenced by the architecture of the target hardware. In this case, since there are advantages in making the compiler's front-end target machine independent, it would be sensible to delay the conversion of the string representing the number into some internal binary form until later in the compilation. So the lexical analyser can return a character string representation of the number. But let us assume in this case that the upper limit of an `int` value in DL is the same as the upper limit imposed by the use of C as an implementation language. We can then use code of the form:

```
case '0': case '1': case '2': case '3': case '4': case '5':
case '6': case '7': case '8': case '9':
  {
    int overflow = 0;
    ival = 0;
    while (isdigit(ch)) {
      if (ival > INT_MAX/10) overflow = 1;
      else if (ival*10 > (INT_MAX - ch + '0')) overflow = 1;
      else ival = ival*10 + ch - '0';
      ch = getchar();
    }

    if (overflow) {
      lextoken = constantsym;
      ival = INT_MAX;
      fprintf(stderr,"integer constant overflow\n");
    }
```

```
      else lextoken = constantsym;
      break;
    }
```

The constant `INT_MAX` contains the maximum value that can be stored in a C integer and the C standard header file `limits.h` defines this value. Note that an attempt to check for overflow by including code of the form `if(ival > INT_MAX)...` is not correct and may give unexpected results. In this code a constant token is returned even if overflow is detected. This may not be the right thing to do even though an error message is generated.

In some languages and implementations based on 2's complement arithmetic, the absolute value of the largest negative number is likely to be larger than the largest positive number. This may need care both with the interpretation of the language definition as well as with the implementation.

Handling overflow like this in the lexical analyser is not always the right thing to do. Some languages make explicit statements in their definitions about size limits on constants and under those circumstances performing the checks in the lexical analyser is sensible. But particularly with languages where limits are implementation-defined, it makes more sense to delay the checking and conversion of constants into some machine-dependent form until much later in compilation.

3.2.2.6 Comments

The DL language supports comments bracketed by `/*` and `*/`. The general strategy to skip comments in the lexical analyser is straightforward (ignore everything between the comment brackets), but coding this may need some care, particularly to maintain the one-character lookahead. To deal with comments in DL, the code to handle the `/` token can be modified:

```
case '/':
  ch = getchar();
  if (ch=='*') {              /* start of comment */
    incomment = 1;
    ch = getchar();
    while (incomment) {
      if (ch==EOF) {
        fprintf(stderr,"warning - end of file in comment\n");
        incomment = 0;
      }
      else if (ch=='*') {
        ch = getchar();
        if (ch=='/') {
          ch = getchar();
          incomment = 0;
```

```
        }
      }
      else ch = getchar();
    }
    lextoken = lex();          /* recursive call */
  }
  else lextoken = dividesym;
  break;
```

Once the `/` character has been found, a comment starts if the following character is `*`. The variable `incomment` is set to TRUE. The comment is ended if the end of the input is detected (and a warning is output) or if the closing comment bracket is found. When the `while (incomment)` loop exits after finding the closing comment bracket, `ch` contains the character immediately following the comment, ready for a recursive call to `lex`. This call will return the lexical token following the comment.

This code does not and should not handle nested comments, so that the lexical analyser will deal with input of the form:

```
/* comment /* comment */ */
```

by treating the input as a comment terminated by the *first* `*/` and then returning a `multiplysym` and then a `dividesym`.

3.2.2.7 Errors

In the code examples above, there are three occasions where the lexical analyser is directly outputting an error/warning message. This is unlikely to be an acceptable strategy when the lexical analyser is integrated within the complete compiler.

The lexical analyser can detect a number of errors such as reading an unexpected character, not in the alphabet of the language. Clearly such errors have to be communicated to the user of the compiler. But also, the compiler has to *recover* from the error so that the analysis of the input can continue. The actions taken by this recovery may depend on the syntactic context of the error. The syntax analysis phase of the compiler has to deal with error reporting and error recovery and so it is sensible to allow the syntax analyser to deal with errors detected by the lexical analyser too. A conventional way of doing this is to pass back a special lexical token when an error is detected, and the syntax analyser can then handle the error. For example, if our lexical analyser encounters a character not in DL's alphabet:

```
default:
  fprintf(stderr,"***Unexpected character %c\n",ch);
  lextoken = errorsym;
  break;
```

Once the syntax analyser has been integrated with the lexical analyser and is dealing with errors satisfactorily all the `fprintf(stderr...)` calls can be removed. But they should stay while the lexical analyser is being developed and debugged.

3.2.3 More General Issues

The last section has illustrated how a simple enclosing structure containing individual pieces of code to deal with each of the lexical tokens can be developed. But to produce a lexical analyser for a real language requires further design decisions and careful attention to coding details.

In the example code above, a one character lookahead simplifies the programming. For example, analysing the input `123+` starts with the character `1` having already been read. This leading character indicates that an integer constant has just started. The `2`, `3` and `+` characters are read and the existence of the non-numeric `+` indicates that the numeric string has ended. The number 123 can then be returned, but the character `+` has already been read, ready for the next call to the lexical analyser. Even for this simple language DL, more lookahead is sometimes required. For example, to distinguish the token types of `print` and `print1` (reserved word and identifier) requires several characters to be read and analysed. Storing the whole of an identifier or reserved word in an array of characters simplifies this process. Similarly, if the lexical analyser detects an overly large constant, at least part of the constant has to have been read before being able to make that decision.

Even for more complex languages it should be possible to use the first character of the token (the character used as the `switch` expression) as a fair indicator of the identity of the complete token. This suggests that structuring the lexical analyser as a large `switch` statement is probably a good idea. And as already advised, starting off with a comprehensive list of token types makes the implementation easier.

In this section various constructs from popular programming languages are examined to see how they may influence the lexical analysis process.

3.2.3.1 Identifiers and Reserved Words

The syntax of identifiers and reserved words/keywords is easy to specify and the rules have to be followed in the lexical analyser. Issues which sometimes cause mistakes include:

- Are upper case and lower case letters equivalent? Do the identifiers `ABC` and `abc` refer to the same thing?
- If the language has keywords or reserved words, is the case of letters significant? Are `while`, `WHILE`, `While` and `whilE` all the same keyword or reserved word?
- Is there a limit on the length of an identifier, and are all characters significant?
- Is white space allowed within an identifier?

In some cases, the language specification may use the term "implementation-defined". The C language treats at least the first 31 characters of an internal identifier to be significant [3]. Java and Python allow identifiers of unlimited length. Fortran 90 allows identifiers up to 31 characters long, they are case-insensitive and Fortran keywords can be used as identifiers, although not recommended for readable programs. Allowing arbitrarily long identifiers requires some sort of dynamic data structure for the storage of the characters of the identifiers.

The lexical analyser clearly needs access to a data structure containing the language's keywords so that keywords can be identified and returned as keywords, identifiers or reserved words as appropriate. Rapid lookup is desirable to help maintain compilation efficiency and some compilers use part of the symbol table for this purpose. Hash tables are often used.

3.2.3.2 Numerical Constants

The issue of size limits on integer constants has already been covered. The complexity of the analysis code is further increased by having to deal with a wide range of numeric constant types supported by most programming languages. For example, ANSI C allows integer constants expressed in various forms: decimal, octal (if the number begins with a zero), hexadecimal (if preceded by `0x` or `0X`), unsigned (followed by `u` or `U`) or long (followed by `l` or `L`). Floating point constants have a more complex syntax and require a systematic approach to the coding of their lexical analysis otherwise messy and unreliable code results. It is sensible to disconnect the reading and analysis of a numerical constant from its conversion to an internal binary form. The lexical analyser can pass on a string representation and the conversion is done later in compilation. Care has to be taken with this conversion process, particularly for floating point values, to maintain accuracy [2].

3.2.3.3 Testing

Writing a compiler can be a huge task, requiring a good grasp of the principles of software engineering. Testing should of course be a key aspect of the project. Beginners to compiler writing often fall into the trap of inadequate testing. They will start a compiler construction project, perfectly reasonably, by coding a lexical analyser, but only testing it on a handful of small test programs. Then the syntax analyser is coded and much time is wasted in attempting to debug the syntax analyser while, in fact, the problems are caused by problems with the lexical analyser.

The lexical analyser should not be overly difficult to test, but the testing has to be systematic and thorough. Write a trivial main program that repeatedly calls the lexical analysis function until end of input is detected. The main program can output the identity of the token returned at each call. Making use of a large library of test programs as input to the lexical analyser is a good first step. The lexical analyser must also be tested with erroneous input such as test programs with carefully crafted

syntactic errors, random data, files containing non-text data, and so on. Limiting cases are essential too. For example, identifiers just shorter than the length limit, at the limit and just over the limit can help reveal "out-by-one" errors.

Such testing with random data can also highlight problems with error handling. The lexical analyser should be capable of dealing with *any* input and report error tokens when necessary. The next phases of compilation can produce helpful error messages for the user and recover appropriately.

3.2.3.4 Difficult Languages

Most programming languages include some "features" that pose particular problems for the lexical analyser. For example, there are some languages where distinguishing between keywords and identifiers can only be done by examining the context in which they are used. A nightmare example is the PL/I language where the language's keywords are not reserved words, allowing statements of the form

```
IF THEN THEN THEN = ELSE; ELSE ELSE = THEN.
```

Similar difficulties arise in FORTRAN where white space is in general insignificant. A famous example of the difficulties this causes is

```
DO 5 I = 1,10
```

compared with

```
DO 5 I = 1.10
```

The former starts a `DO` loop iterating to statement number `5` and the latter sets the value of the variable `DO5I` to the value `1.10`.

Dealing with these awkward cases is not easy. Some form of lookahead is essential here, and practical solutions may have to resort to multi-pass lexical analysis. Fortunately languages requiring such treatment are becoming less common.

3.2.3.5 Evaluation

Implementing a lexical analyser in the direct way outlined in this section has much to recommend it. The overall structure of the lexical analyser is intuitive, with clear separation between the sections of code dealing with each individual token. The code for most tokens seems straightforward. However, some aspects of the design of most programming languages can cause implementation difficulties and may require annoying-to-program lookahead. Furthermore, some lexical tokens may have a complicated syntax (the floating point number is the often-quoted example) and ensuring that the hand-written code dealing with these tokens is reliable may turn out to be a taxing task.

To make dealing with these complex tokens easier, a more structured approach to lexical analysis is needed. This approach is described in the next section. We need a way of generating a formal specification of the syntax of the tokens and then transform that specification into recognising code. The more that this process can be automated the better.

3.3 Regular Expressions

In Chap. 2 we saw how Chomsky type 3 (finite-state or regular) grammars have a special significance for lexical analysis. We aim to specify the syntax of lexical tokens in terms of a finite-state grammar and then somehow transform that specification into the actual code for the lexical analyser.

3.3.1 Specifying and Using Regular Expressions

Regular expressions arise from Chomsky type 3 grammars. A Chomsky type 3 grammar has productions of the form:

$A \rightarrow a$ or $A \rightarrow aB$

Directly expressing the syntax of lexical tokens using this formulation is not really a practicable task. Instead, we use a different notation that is formally equivalent to a type 3 grammar expressed using productions of the form shown above. This notation makes use of *regular expressions*.

A regular expression is made up of symbols of the language being defined together with operators that support:

- concatenation (traditionally specified by symbol adjacency in the regular expression),
- alternation (symbols or groups of symbols are separated by the | operator) and
- repetition (symbols or groups of symbols are followed by the $*$ operator to signify zero or more repetitions).

Parentheses can also be used to group symbols. Conventionally, the repetition operator has highest precedence, followed by concatenation, with alternation having the lowest precedence.

For example:

- abc denotes the set of strings with the single member $\{abc\}$.
- $a|b|c$ denotes the set $\{a, b, c\}$.
- $a*$ denotes $\{\varepsilon, a, aa, aaa, \ldots\}$. ε is the empty string.
- $ab*$ denotes the infinite set $\{a, ab, abb, abbb, \ldots\}$.
- $(a|b)*$ denotes the set of strings made up of zero or more a's or b's.
- $a(bc|d)*e$ denotes the set of strings including $\{ae, abce, abcde, ade, \ldots\}$.

Regular grammars and regular expressions are equivalent in the sense that a language expressed in one can be transformed to the same language expressed in the other. Note also that a specification in terms of a regular expression is not necessarily unique. For example $a(b|c)$ and $ab|ac$ are equivalent regular expressions.

The regular expression notation is simple yet powerful. It is compact and unambiguous. And it is well-suited to the specification of the syntax of lexical tokens. For example, we can define an integer constant as follows:

digit → 0|1|2|3|4|5|6|7|8|9
intconstant → digit digit∗

Similarly, we can define an identifier as an initial letter followed by a sequence, possibly empty, of letters or digits:

digit → 0|1|2|3|4|5|6|7|8|9
letter → a|b|c| . . . |z|A|B|C| . . . |Z
identifier → letter (letter | digit)∗

3.3.2 *Recognising Instances of Regular Expressions*

Transforming a set of regular expressions into code that recognises instances of those regular expressions is our aim. We start with a set of regular expressions defining the syntax of all the lexical tokens of our language and then transform the regular expressions into recognising code, yielding the basis of a complete lexical analyser.

Suppose we wish to recognise instances of the regular expression $(ab|c)*d$. This regular expression can be expressed in the form of a syntax diagram, as shown in Fig. 3.1.

Generating this form of directed graph from a regular expression is uncomplicated (it is just a different way of presenting the regular expression) and by following the arrows, the graph can be used to generate instances of the regular expression. But we really have the reverse problem: that of using this graph to parse strings to determine whether they are capable of being generated from the regular expression. It turns out that this is possible and an algorithm described in [4] can be used. Unfortunately this algorithm proves to be too inefficient for practical lexical analyser construction, particularly for complex regular expressions. A different approach is needed.

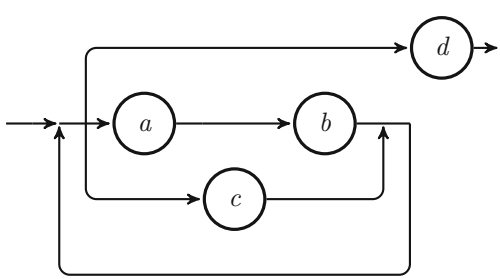


Fig. 3.1 Directed graph representation of $(ab|c)*d$

3.3.3 *Finite-State Machines*

The directed graph presented in the previous section is ideal for *generating* instances of the regular expression but not so good for *recognising* instances. An alternative structure is needed. This is the *transition diagram* which is another directed graph but with labelled branches. The transition diagram for the regular expression $(ab|c)*d$ is shown in Fig. 3.2. Each node in the transition diagram is called a *state*. One or more states are enclosed in double circles to signify they are *accepting states*.

In this form of transition diagram there is a unique *starting state* (state 1 in Fig. 3.2). In this simple example there is just one accepting state (state 3). Each *edge* or *transition* in the diagram is labelled with a character. These characters are matched with characters read from the input during the recognition process.

To illustrate the actions of this *finite-state machine* while operating as a parser for strings from the regular expression, we consider the input $abccd$.

- Start in state 1. Read a. Find the edge labelled a from state 1 and follow that transition. Move to state 2.
- Read b. Move to state 1.
- Read c. Stay in state 1.
- Read c. Stay in state 1.
- Read d. Move to state 3.
- Because we end in an accepting state, the parse has succeeded, showing that $abccd$ is an instance of the regular expression represented by this finite-state machine.

This approach provides an efficient parser for instances of regular expressions. The time complexity is clearly $O(n)$ where n is the number of characters in the input—the cost of parsing tends to grow linearly with the length of the input.

If erroneous input is input to this machine, the actions are different. For example, consider the input ad.

- Start in state 1. Read a. Move to state 2.
- Read d. There is no edge labelled d from state 2, so an error is reported. Usefully, the error message can provide the information that a b was expected at this point.

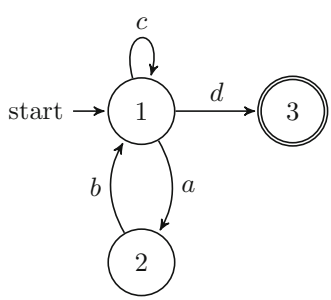


Fig. 3.2 Transition diagram for the regular expression $(ab|c) * d$

The transition diagram can be represented in other ways. In particular, representing it as a *transition table* may be helpful. Figure 3.2 can be represented as:

	input character			
state	a	b	c	d
1	2	—	1	3
2	—	1	—	—
3	finished			

This is looking encouraging. Representing regular expressions as finite-state machines allows efficient parsing, and the code of the parser is simple, controlled by a directed graph or a transition table. Assuming that it is manageable to transform regular expressions into finite-state machines, this approach looks as though it might be a good, practical route for the implementation of lexical analysers.

To give an overview of the process of going from regular expressions to finite-state machines, we need to examine the characteristics of a finite-state machine a little more carefully. The finite-state machines described so far in this section are called *deterministic finite-state machines* because they have the characteristic that for any state, there is at most one possible next state for each input symbol. In other words, the entries in the transition table are either empty or contain a single next state. No two edges from a node representing a state can be labelled with the same input symbol. These machines are contrasted with *non-deterministic finite-state machines* where an input symbol can trigger the machine into more than one next state *simultaneously*. There may be states having two or more edges labelled with the same input symbol. A non-deterministic finite-state machine can also have multiple starting states. The inherent parallelism of these machines clearly causes implementation difficulties. However they are important because there is a straightforward algorithm for transforming a regular expression into a non-deterministic finite-state machine.

There are several steps in this implementation process.

- Start with the regular expression. This can be transformed into a non-deterministic finite-state machine.
- Because the implementation of a non-deterministic machine is complicated and potentially inefficient, the non-deterministic finite-state machine is transformed into an equivalent deterministic finite-state machine.
- The deterministic finite-state machine thus generated may well have many more states than strictly necessary, and a process of state minimisation is performed to ensure that the machine is the simplest possible for the parsing of instances of the original regular expression.

These algorithms will not be detailed here. They are standard, well-understood algorithms described on the web and in many textbooks. For example, see [5].

This results in a powerful process for the construction of a lexical analyser. The syntax of the individual tokens is expressed in terms of a set of regular expressions, these regular expressions are transformed into state-minimised deterministic finite-state machines which can then be implemented by transforming the finite-state machine into code. This results in an efficient lexical analyser. The resulting lexical analyser has implementation issues that have to be considered, particularly concerning the representation of a potentially large but sparse transition matrix, but these issues can be overcome without too much difficulty.

Unfortunately, this process is really not appropriate for carrying out by hand. The individual steps may appear simple but as soon as there is any complexity in the regular expressions, the detail becomes overwhelming and it is very hard and time-consuming to produce a lexical analyser with any hope for correctness. On the other hand, carrying out this process by machine is a much more attractive proposition and there are now many software tools that are capable of generating lexical analysers, in a wide variety of implementation languages, starting from a set of regular expression specifications. And this gives us a second, practical approach to the construction of lexical analysers with the key advantage of being able to handle arbitrarily complex regular expressions, generating recognisers that are likely to be correct as well as efficient, using little human effort.

3.4 Tool-Based Implementation

Techniques for the automated production of lexical analysers are not new. The Unix-based program *lex* was described in [6] and became very popular soon after its introduction, being used for a wide range of applications, not all by any means in the field of compiler construction. *Lex* takes as input a set of regular expressions defining the tokens, each being associated with some C code indicating the action to be performed when each token is recognised. The output of the *lex* program is itself a C program that recognises instances of the regular expressions and acts according to the corresponding action specifications.

Many descendants of *lex* were then developed, generating code in a wide variety of languages. Tools such as *flex* [7] (again, generating C) and *JLex* (generating Java) have become very popular. Most of these tools share a largely common format for the specification of the regular expressions. Other compiler-construction tools combine lexical analyser generation with syntax analyser generation and maybe other compiler phases too.

These software tools have been refined over the years so that they are now capable of generating very efficient lexical analysers, in terms of both execution time and runtime storage requirements, with minimum effort from the programmer. They often contain functionality which addresses many of the irritating practical difficulties of lexical analysis.

3.4.1 Towards a Lexical Analyser for C

As an example of the use of software tools for the generation of lexical analysers, this section examines the use of the *flex* tool, ending with its use in constructing various parts of a lexical analyser for the C language.

Although lexical analyser generating tools are large and complex pieces of software, their use is not difficult. For example, *flex* takes as input a specification file that contains amongst other things a set of regular expressions, each associated with a corresponding piece of C code. These are the *rules* which specify the actions of the lexical analyser. *Flex* will generate C code defining a function `yylex()`. This is the lexical analyser itself and it is subsequently linked with the remainder of the code for the compiler. Each time the function `yylex()` is called, it finds the next token from the input, the tokens being defined by the regular expressions originally used as the specification file for *flex*. The C code associated with the matching regular expression is executed, and it is this code that has the responsibility in a conventional lexical analyser for returning the identity of the token just recognised. The automated generation of the recognising code makes the construction of a lexical analyser much easier.

The high-level structure of the input is simple. It consists of three sections, separated by lines containing just the characters `%%`.

```
definitions
%%
rules
%%
user code
```

The `definitions` section consists of a set of name/definition pairs, used primarily to associate mnemonic names with regular expressions used in the `rules` section. The `rules` section is a list of regular expression/corresponding C action pairs. Finally, the `user code` section is copied directly to the output generated by *flex* and contains C functions which call or are called by the code in the `rules` section.

The details of *flex* syntax are easily available. There are many internet resources and textbooks, such as [7]. The power of *flex* is illustrated here by some examples of its use.

3.4.1.1 A Simple Example

Before tackling the lexical analysis of C and to illustrate the structure of the input to *flex*, an example of a very simple lexical analyser may be helpful. The language for which this lexical analyser is being written includes just three distinct lexical tokens:

1. A “word”, consisting of a letter (all letters can be upper or lower case), followed by an arbitrary number (can be zero) of letters or digits.
2. A “number”, consisting of one or more decimal digits.
3. Any other character. White space (consisting of any number of spaces, tabs and newlines) is ignored. But note that white space can be used to separate adjacent words or numbers.

This lexical analyser could form the starting point for the front-end of a compiler for a very simple programming language or for a command language interface to a software package.

Flex generates a function `yylex` which is called to get the next token from the input. We would like `yylex` to return an integer value to indicate the identity of the token, 1 for a “word”, 2 for a “number” and 3 for any other character. There are no illegal tokens in this language.

Here is a file for input to *flex*, complete with a main program to repeatedly call the lexical analyser generated by *flex*.

```
letter            [a-zA-Z]
digit             [0-9]
letter_or_digit   [a-zA-Z0-9]
white_space       [ \t\n]
other             .

%%

{white_space}+                ;
{letter}{letter_or_digit}*    return 1;
{digit}+                      return 2;
{other}                       return 3;

%%

int main() {
   int lextoken;
   while (lextoken = yylex())
      printf("%d - %s\n",lextoken, yytext);
}

int yywrap()
{
   return 1;
}
```

This file is input to *flex*, thus generating a C program, in default, a file called `lex.yy.c`. This C program is compiled and the resulting executable program

extracts the tokens from its input, outputting the identities of the tokens. If this lexical analyser were to be included in a complete compiler, the main program would obviously have to change and the syntax analyser would probably have the responsibility of calling `yylex`. But this structure for a main program offers a simple way to help verify that the lexical analyser is operating correctly.

The first section of the *flex* code (the definitions) gives symbolic names to regular expressions used later in the rules section. The `letter` expression matches a single (upper or lower case) letter. The `digit` expression matches a single decimal digit. `letter_or_digit` matches a single letter or decimal digit and `white_space` matches a single space, tab or newline. Finally, `other` matches *any* single character except newline. These definitions are not strictly necessary, but they allow the use of symbolic names in the rules section to improve readability.

The rules section consists of just four rules. Each rule consists of a regular expression/pattern followed by the action in C code to be taken on matching that regular expression. The first rule matches white space of any length on the input and the null C statement as the action causes the white space to be ignored. The pattern in the second rule matches a "word" as defined at the beginning of this section (a letter followed by an arbitrary number of letters or digits). The action on recognising a "word" is for the lexical analyser (i.e. the function `yylex`) to return the integer value 1. The third rule matches a "number" as a string of one or more decimal digits and returns the value 2. The final rule is a catch-all so that any other character still unmatched causes the integer value 3 to be returned by `yylex`.

Finally, the user code section of the input to *flex* defines a main program to repeatedly call `yylex()` outputting the value returned by `yylex` (the identity of the lexical token) together with a string representation of the token. *flex* provides a special variable `yytext` that contains the text that matched the regular expression pattern in the rule. The `while` loop repeatedly calls `yylex()` and this loop terminates when `yylex()` returns the value 0, signifying end of file.

The `yywrap()` function is also defined here as always returning the value 1. `yywrap` is automatically called when `yylex` encounters the end of the input file. If `yywrap` returns 1, then `yylex` assumes that its job is done and there are no more characters to analyse. If, however, `yywrap` returns 0, this indicates that `yylex` should continue and `yywrap` will have opened a new file for processing. This is a mechanism for allowing input from multiple files, not needed in this simple example.

The rules and the user code can be seen in action in this sample run. When the program generated by *flex* runs and is presented with the input:

```
a
abc
ABc123a

abc 123+45
1.2 * 3
```

it produces the output:

```
1 - a
1 - abc
1 - ABc123a
1 - abc
2 - 123
3 - +
2 - 45
2 - 1
3 - .
2 - 2
3 - *
2 - 3
```

This simple example illustrates the issue of *ambiguity* in the set of regular expression patterns. The first line of the sample input consists of a single `a` and the lexical analyser correctly identifies this as a "word" (lexical token 1). But this `a` also matches the final rule (lexical token 3). The *flex*-generated lexical analyser has a feature ensuring that if two or more rules match the same string, the earlier rule takes precedence. So the ordering of patterns in the rules section may be important. Furthermore, the input `ABc123a` is also recognised here as a single "word". It could also match the given rules as a "word" (`ABc`), followed by a "number" (`123`), followed by another "word" (`a`). Because *flex* has the feature that the longest match is the one chosen, the action here is to return a single "word". These two rules for resolving ambiguity greatly help simplify the construction of lexical analysers for programming languages. If the rules input to *flex* had to be formally unambiguous, they would become unmanageably complex.

3.4.1.2 A Lexical Analyser for DL

DL is a simple language with a small set of easily-defined lexical tokens. We have already seen how a lexical analyser for DL can be coded directly in C but it is certainly worth examining a different approach, using the *flex* tool. As ever, the path to an implementation starts with producing a list of the tokens to be recognised, and generating regular expression definitions for each token. This leads directly to the *flex* specification.

```
%{
    .
    .
    .
int lexnumval;
```

```
char lexident[MAXIDLEN+1];

%}

letter          [a-z]
digit           [0-9]
letter_or_digit [a-z0-9]
white_space     [ \t\n]
other           .

%%

"=="    return EQSYM;
"<="    return LESYM;
">="    return GESYM;
"!="    return NESYM;

else    return ELSESYM;
if      return IFSYM;
int     return INTSYM;
print   return PRINTSYM;
read    return READSYM;
return  return RETURNSYM;
while   return WHILESYM;

{letter}{letter_or_digit}*      return IDENTIFIER;

{digit}+        {  lexnumval = atoi(yytext);
                   return CONSTANT;
                }

"/*"            { comment(); }

{white_space}+  ;

{other}         return yytext[0];

%%
   .
   .
   .
```

This example code is not quite complete. It needs a short preamble to include header files providing a link to the syntax analyser and defining token names and so

on. It also needs a few lines of code to read to the end of a comment (the `comment()` function has to be written).

The definitions of `letter`, `digit`, `letter_or_digit`, `white_space` and `other` are simply made to improve readability in the rules section. The two-character tokens are defined first, followed by the reserved words, followed by identifiers and numerical constants. Comments and white space are then ignored and finally all the single character tokens are handled by the simple rule matching all remaining single characters.

The order of the rules has to be checked. This is a common source of error. The reserved words have to be matched before the identifiers to prevent an input such as `else` being recognised as an identifier rather than a reserved word. Here, both rules match the same number of characters and the first rule of the two takes precedence. Using the numerical value of the character for single character tokens is a useful trick, saving coding effort. The traditional way of managing the encoding of tokens is to make sure that the defined values of all the non-single character tokens are greater than 255 (the maximum numerical value of a single character, assuming the use of an 8-bit character set). This is handled almost automatically if these token names (`EQSYM`, `LESYM`, etc.) are declared in the *bison* file defining the syntax analyser. Examples of this integration are given in Chap. 5.

Finally, the code associated with integer constants would need a little enhancement to handle errors such as overflow. The function `atoi` needs to be replaced by something that can handle these errors. The value of the constant is placed in the global variable `lexnuval` to make it available to the syntax analyser.

3.4.1.3 Towards a C Lexical Analyser

Flex and other lexical analyser generating tools have many more features to support the specification of programming language-oriented regular expressions and actions. To illustrate this, we now look at some of the techniques that can be used to construct a lexical analyser for C. Some of the more interesting lexical tokens from C are selected for implementation using *flex*.

It is tempting to start off by coding the regular expressions required to recognise the individual tokens, but some careful planning is required first. It is essential to begin with an accurate list of all the lexical tokens to be recognised, obtained from a language definition document. Next, various design issues should be considered, including:

- What is the interface between the lexical analyser and the syntax analyser? Typically, the token identity is encoded as an integer value and the "values" of tokens such as constants and identifiers are returned as strings.
- How are source lines beginning with # to be treated? Are they all handled by the preprocessor? Presumably the `#line` directive has to be handled in the lexical analyser.

- There is a particularly nasty problem in the analysis of C in the handling of `typedef` names (see Sect. 3.1.3). It is probably advisable for the lexical analyser to ignore this particular issue and leave it to the syntax analyser.
- How is it best to handle reserved words? Having a separate regular expression for each reserved word (e.g. `if  return t_if;`) is certainly possible, but it *may* be better to handle reserved words and names using the same rule (letter or underscore followed by any number of letters, underscores or digits) together with some simple code to check whether the matched string is in fact a reserved word.
- Are there constraints on the ordering of patterns in the rules section? Is there any potential ambiguity?
- Are there any regular expressions that could usefully be included in the *flex* definition section to make the rules section more readable? This should become clear as the rules are developed.

We can now examine the coding implications of some of the issues involved with writing a lexical analyser for C. The full source is not presented here because there are good examples of such code available on the internet. A simple search will reveal a great deal of useful information. We can follow the design of the lexical analyser presented in Sect. 3.4.1.1, adding more rules and providing functions in the C code section to support the recognition of particular tokens. Some of the more interesting aspects of an implementation are as follows.

- The identities of the tokens recognised by the lexical analyser are returned as integers, but to make the code a little more readable and maintainable, the tokens are given symbolic names:

```
typedef enum {t_constant = 1000, t_string_literal,
t_right_assign, t_left_assign, t_add_assign, t_sub_assign,
t_mul_assign, t_div_assign,
.
.
.
```

  All the tokens can be defined in this way. The starting value of `1000` is prompted by a simple trick for the representation of all the single character tokens in the language. These are all returned as their character values assuming that they can all be represented as conventional ASCII values, starting the representation of other tokens at `1000` can cause no conflict.
- A simple main program is required to support the testing of the lexical analyser. Something like this will probably suffice:

```
int main()
{
  int lextoken;
  while (lextoken = yylex())
```

```
    printf("%d is \"%s\"\n", lextoken, yytext);
}
```

Here, the lexical analyser (`yylex()`) is called repeatedly until the end of the input file, each time outputting the identity of the token read as an integer and the text actually recognised (`yytext`).

- Comments (starting with `/*` and ending with `*/`) are best recognised by a rule of the form:

```
"/*"                    { comment(); }
```

The function `comment()` reads the input, discarding characters until it has read a `*` followed by a `/` or end of file. Implementing it this way is much simpler than puzzling over the details of a regular expression to represent a complete comment which can, of course, run over multiple lines.

- Identifiers and reserved words can be dealt with together. Assuming that `LETTERS` is defined as `[a-zA-Z_]` and `DIGITS` as `[0-9]`, the rule for identifiers and reserved words is:

```
{LETTERS}({LETTERS}|{DIGITS})*    { return(classify()); }
```

Here, `classify()` is a function returning the integer identity of the word just recognised—is it a reserved word (if so, which one) or an identifier (if so, return the value `t_identifier`)? A simple binary or even linear search or a hash table would be appropriate for `classify()`.

- Numerical constants have a variety of forms. For example, a decimal integer constant consists of a nonzero decimal digit followed by zero or more decimal digits followed by an optional *integer-suffix*. The *integer-suffix* is either `l` or `u` or `ul` or `lu`, and the `l`'s and `u`'s can be in upper or lower case. A possible rule is:

```
[1-9][0-9]*([uU][lL]?)|([lL][uU]?)?   { return t_constant; }
```

There are many other constant types to deal with. At first sight, these regular expressions may seem overwhelmingly complex, but constructing them step by step from a formal definition of the language is tractable.

- String constants are based on a sequence of characters enclosed between double quotes. A pattern of the form `\"(\\.|[^\\"])*\"` deals with a sequence of escape characters (a \ followed by another character) and ordinary characters (except for \ or " – the ^ specifies that the set of characters to be matched *excludes* those in the square brackets), all enclosed between double quotes. Dealing with escape characters in their various forms is handled elsewhere in the lexical analyser.

- Simple single and multi-character tokens are much easier. Rules of the form:

```
">>="                    { return t_right_assign; }
.
.
.
"<="                     { return t_le_op; }
">="                     { return t_ge_op; }
"=="                     { return t_eq_op; }
.
.
.
"="                      { return'='; }
"<"                      { return'<'; }
">"                      { return'>'; }
.
.
.
```

  can be used. Note that the rules of the *flex*-generated lexical analyser will result in `t_le_op` rather than `'<'` and `'='` being recognised when presented with the input `<=`.
- White space can be ignored, and finally any remaining unrecognised characters can also be flagged as an error and then ignored:

```
[ \t\v\f]                ;
[\n]                     { lineno++; }
.                        { yyerror("Illegal character"); }
```

It would be better in the complete compiler to report the illegal character to the syntax analyser as an error token, but for testing a stand-alone lexical analyser this simple error message will suffice.

Attention to detail is the key message. The syntax of the lexical tokens of most programming languages can be surprisingly complex, and ensuring that the rules in *flex* correspond precisely to the rules in the language definition needs care.

As can be seen, specifying more complex regular expressions can be challenging. The syntax of *flex*'s regular expression includes many powerful constructs, and this flexibility results in a regular expression language that needs careful study [7].

3.4.2 Comparison with a Direct Implementation

This chapter has examined two distinct approaches for the construction of a lexical analyser. Both have advantages and both are used today for real compilers. Hybrid

approaches are also possible, where some of the tokens are recognised by hand-written code and the others via the regular expression rules of a lexical analyser generator.

Writing a complete lexical analyser by hand has several advantages.

- Programming by hand may be the only option because there may be no lexical analyser generating tools available that are compatible with the programming language being used for the compiler's implementation.
- The process of compiler construction is less dependent on the availability of other software tools, and there is no need to learn the language of a new software tool. For example, the newcomer may find specifying complex regular expressions in *flex* (and other, similar tools) rather difficult.
- There are no real constraints on the techniques used for recognising tokens because it is easy to add ad hoc code to deal with awkward analysis tasks.
- The memory requirements of the lexical analyser can be modest. A lexical analyser generated automatically *may* be more memory hungry despite the effect of techniques for internal table compression.
- Performance of the lexical analyser produced in this way can be very good.

However, using a software tool purposely made for the job offers some significant benefits.

- The code is more likely to be correct, particularly for lexical tokens with a complex structure. For example, writing code by hand for recognising a floating point constant as found in traditional programming languages is a daunting task. But once a regular expression has been constructed, the software tool should generate accurate code without a problem.
- Regular expression specifications for the lexical tokens may be available in the language specification or documentation. Transcribing them into a *flex*-acceptable form should be easy.
- The task of writing the lexical analyser is potentially much simpler because much less code has to be written.
- The lexical analyser is easy to modify.
- Performance of the lexical analyser produced in this way can also be very good.

Lexical analysers for DL have been written by hand and also by using *flex*. The hand-written lexer is approximately 140 lines of code, the *flex* file is approximately 70 lines of code, including all the preamble and associated functions. The C code generated by *flex* has approximately 1870 lines (but not designed or intended for human reading!). One would expect to find increased proportional savings in lines of source code as the complexity of the lexical analyser increases. When execution times are compared, it is clear that there is very little difference between the two approaches, with the *flex*-generated parser just marginally ahead in most tests.

There is no clear winner here. Nevertheless the advice of "using the right tool for the job" is worth bearing in mind.

3.5 Conclusions and Further Reading

The lexical analyser is a good place to start when undertaking a compiler construction project. The algorithms and data structures that have to be used do not need to be complex, and their use should result in code that is efficient in terms of both space and computation time.

Starting with a precise definition of all the lexical tokens is essential. This implies a good understanding of the interface between the lexical and syntax analysers and their division of responsibilities. Thinking in terms of a regular expression formalism is undoubtedly the right way of managing tokens.

Testing is essential too. Leaving testing of the lexical analyser until it is connected to the syntax analyser will make debugging very much more difficult. Testing the lexical analyser as a stand-alone program is clearly the right thing to do.

There are many compiler textbooks explaining the theory of finite-state automata and discussing how these machines may be implemented in software. For example, [5, 8–10] all contain much relevant background material to the construction of lexical analysers. A good reference for the compiler construction tools *lex* and *yacc* (and *flex* and *bison* too) is [7].

There are now many high-quality open-source examples of lexical analyser code available on the web, easily found with any search engine. These form a superb learning resource. Compilers today are rarely written from scratch and freely available examples can be used for ideas and as the basis for further development. There are also good tutorials available for the compiler construction tools. The archives of the *comp.compilers* newsgroup and mailing list form an invaluable compiler construction resource.

Exercises

3.1. Write a lexical analyser for a simple programming language (such as DL) firstly by hand and then by using a lexical analyser generator, generating code in the same language as you used for the hand-written version. Compare the sizes of the two sources. Also compare code sizes and execution times.

3.2. Gather together the specification of several high-level programming languages and extract precise information about:

- The characters that can appear in an identifier/name. Are there length restrictions? Are upper and lower case letters treated as being identical?
- Comments—exactly where can they appear?
- White space—where and how is it significant?

3.3. Write some code to read and analyse a floating point constant. Assign its value to a floating point variable and print it. How "accurate" is your conversion. Maybe [1] will help with the understanding of these issues.

3.4. A floating point constant in a hypothetical programming language consists of one or more decimal digits followed by a full stop, followed by zero or more

decimal digits. Express this in terms of a regular expression. Try it out using *flex* or a similar alternative.

3.5. What high-level language characteristics make lexical analysis difficult? Why should a particular construct be recognised in the syntax analyser rather than in the lexical analyser?

3.6. Write a piece of software to count the number of times the reserved word `if` appears in a C program source file. Consider modifying this program to count the number of distinct identifiers used in the program. Why is this harder?

3.7. Write a complete lexical analyser for a real programming language. Make sure that it works.

References

1. Goldberg D (1991) What every computer scientist should know about floating-point arithmetic. ACM Comput Surv 23(1):5–48
2. Clinger WD (1990) How to read floating point numbers accurately. In: Proceedings of the ACM SIGPLAN '90 conference on programming language design and implementation, White Plains, NY, pp. 92–101
3. Kernighan BW, Ritchie DM (1988) The C programming language, 2nd edn. Prentice Hall, Englewood Cliffs
4. Thompson K (1968) Regular expression search algorithm. Commun ACM 11(6):419–422
5. Mogensen TÆ (2011) Introduction to compiler design. Undergraduate topics in computer science. Springer, Berlin
6. Lesk ME (1975) Lex – a lexical analyser generator. AT&T Bell Laboratories, Murray Hill. Computing Science Technical Report 39
7. Levine J (2009) Flex & bison. O'Reilly Media, Sebastopol
8. Grune D, Bal HE, Jacobs CJH, Langendoen KG (2000) Modern compiler design. Wiley, New York
9. Aho AV, Ullman JD (1979) Principles of compiler design. Addison-Wesley Publishing Company, Reading
10. Aho AV, Lam MS, Sethi R, Ullman JD (2007) Compilers – principles, techniques and tools, 2nd edn. Pearson Education, London

Chapter 4
Approaches to Syntax Analysis

The heart of the analysis phase of the compiler is the syntax analyser. It takes a stream of lexical tokens from the lexical analyser and groups them together according to the rules of the language, thus determining the syntactic structure of the compiler's input. The syntax analyser creates data structures reflecting this syntactic structure and then it is up to later phases of compilation to traverse these structures and finally to generate target code.

Section 2.3.3 introduced the idea of *parsing* where the syntax rules of the language guide the grouping of lexical tokens into larger syntactic structures. Parsing requires the repeated matching of the input with the right-hand sides of the production rules, replacing the matched tokens with the corresponding left-hand side of the production. But as we have seen, the order in which this matching is done and also the choice of which productions to use is fundamentally important. We need to develop standard algorithms for this task, and as a first step, examining the reverse process of *derivation* may help with this.

4.1 Derivations

We start with a very simple grammar, shown in Fig. 4.1 defining a rudimentary form of arithmetic expressions using the variables `x`, `y` and `z`, with the operators + and `*`.

Our aim is to show how the string `x+y*z` can be derived from the starting symbol <expr>, thus showing that `x+y*z` is a sentence of this grammar.

Fig. 4.1 BNF for a trivial arithmetic language

<expr> ::= <term> | <expr> + <term>
<term> ::= <factor> | <term> * <factor>
<factor> ::= x | y | z

4.1.1 Leftmost and Rightmost Derivations

The process of generating a derivation starts, naturally, with the starting symbol. Then, at each stage of the derivation a non-terminal symbol in the sentential form is chosen and this symbol is replaced by the corresponding production's right-hand side. For example, `x+y*z` can be derived from <expr> as follows:

```
<expr>
<expr> + <term>
<term> + <term>
<factor> + <term>
x + <term>
x + <term> * <factor>
x + <factor> * <factor>
x + y * <factor>
x + y * z
```

In this example, the leftmost non-terminal in each sentential form is expanded at each step, and when `x+y*z` is generated, the expansion stops because there is nothing left to expand. This is the *leftmost derivation*.

It is possible to derive `x+y*z` in a different way. For example:

```
<expr>
<expr> + <term>
<expr> + <term> * <factor>
<expr> + <term> * z
<expr> + <factor> * z
<expr> + y * z
<term> + y * z
<factor> + y * z
x + y * z
```

Here, the rightmost non-terminal in each sentential form is being expanded at each step. This is the *rightmost derivation*, sometimes called the *canonical derivation*.

It is of course possible to perform the derivation by using other selection methods for non-terminals to be expanded but at each step there has to be an explicit or implicit indication of which non-terminal is being expanded and hence which production rule is being used. In the cases of leftmost and rightmost derivations, the identity of the non-terminal being expanded is implicit.

4.1.1.1 Parse Trees

These two derivations look radically different although they both produce the same sentence. They also share a common parse tree. The generation of the parse tree while performing the derivation is not difficult. The root of the tree is the starting symbol (here, <expr>). If the production $X \rightarrow a_1 a_2 \ldots a_n$ is used in the derivation process, add the children $a_1 a_2 \ldots a_n$ to the node X. This implies that the number of children of a node is equal to the number of non-terminal symbols in that node.

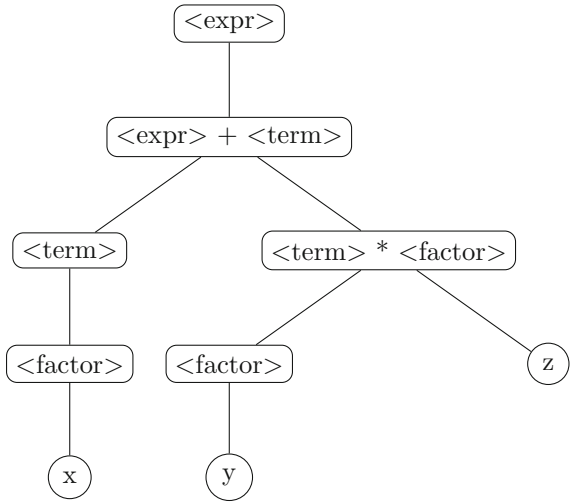


Fig. 4.2 Tree from the derivation of `x+y*z`

It is easy to verify in the example above that the parse tree generated in both the leftmost and the rightmost derivations is the tree shown in Fig. 4.2. It is just the order in which the nodes are generated that is different.

The parse tree has embedded within it all the production rules used in constructing the derivation. The parse tree defines a unique leftmost derivation and a unique rightmost derivation.

The process of parsing is equivalent to performing derivation in reverse. We have already seen in Sect. 2.3.3 that the choice of which substring to reduce at each step is critically important to the parsing process. Specifically, the substrings cannot be chosen at random. But if we guarantee that the substring matching corresponds to the choice and order of either the leftmost or the rightmost derivation then the parsing process will succeed.

4.2 Parsing

Many algorithms have been developed to help solve the parsing problem. Most of these algorithms are designed to work using grammars with particular characteristics and later in this section we will see some of these constraints. Some algorithms work with grammars with few constraints and such parsers are rarely used in real compilers largely because of their significant demands of computation time and/or memory space. For nearly all programming languages comparatively simple parsing techniques can be used and it is those that will be examined in this book. The principles behind parsing will be presented in this chapter. Practicalities of parser implementation are covered in Chap. 5.

Section 2.3.3.2 distinguished between *top–down parsers* and *bottom-up parsers*. The reductions performed by the top–down parser correspond to the order and identity of the substitutions performed by the leftmost derivation. The bottom-up parser

corresponds to the rightmost derivation, performing the substitution steps in the reverse order to that used by the rightmost derivation.

4.2.1 Top–Down Parsing

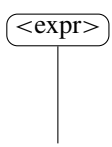
The top–down parser starts by constructing the parse tree with a single node labelled with the start symbol. It can then build up the complete parse tree by creating the subtrees one by one, in a left-to-right order. In building a subtree, the root node of that subtree is created and than all the sub-subtrees of that subtree are generated. This is a recursive algorithm, generating the tree in pre-order (node, then its subtrees in a left-to-right order).

This is hard to visualise. Fortunately, the algorithms required to do this are usually very simple for compliant grammars and the construction of the parse tree integrates easily with the reading and recognition of the lexical tokens.

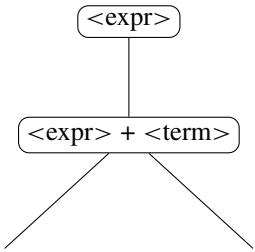
4.2.2 Parse Trees and the Leftmost Derivation

To show how this type of parser works, we can return to the trivial arithmetic language presented above. Our task is to illustrate the parsing of `x+y*z`.

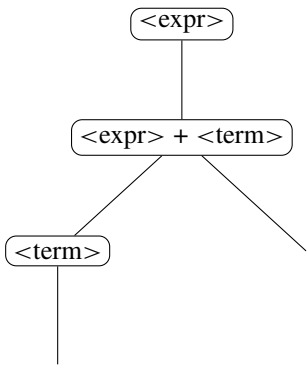
- The parser starts off by constructing a tree containing just the starting symbol as the root node.



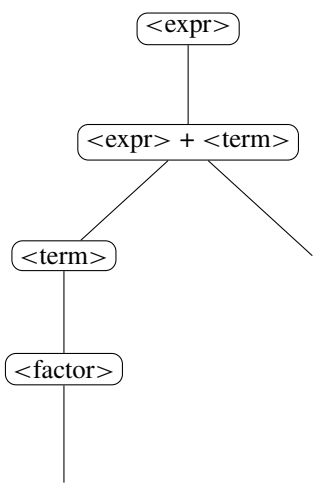
- The next step in the pre-order generation is to set up the leftmost subnode. This is done by looking at the grammar of the language and noting that <expr> is defined as
  <expr> ::= <term> | <expr> + <term>
  Which alternative should be chosen? This turns out to be a difficult issue because of the characteristics of this grammar. We choose the
  <expr> ::= <expr> + <term>
  alternative in this case (because we happen to know that it is the right thing to do). This approach is acceptable for an example, but clearly has no place in a parsing algorithm used in a compiler. This problem will be examined in detail later in this chapter. The tree then looks like this.



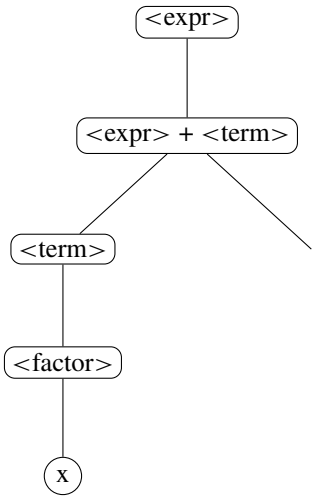
- We then have to deal with the nodes and their subtrees from left to right. This time we use the production
  <expr> ::= <term>
  and the tree becomes:



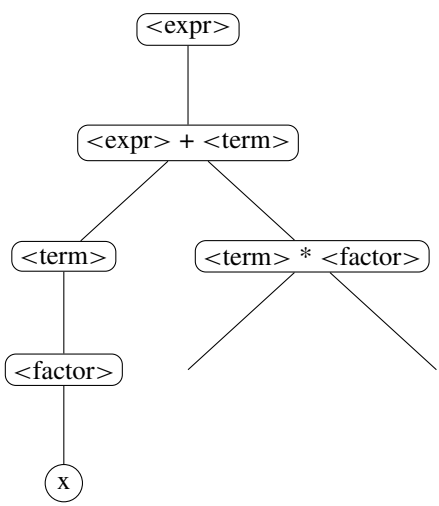
- The lower <term> node has to be tackled next. Use the production
  <term> ::= <factor>
  Again, the reason for this choice has to remain tenuous for now.



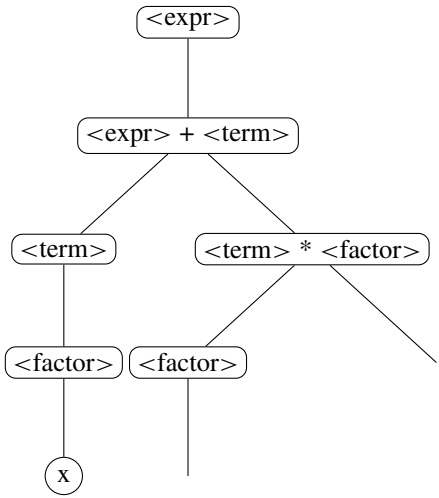
- The <factor> node is given a single child via the production
  <factor> ::= x



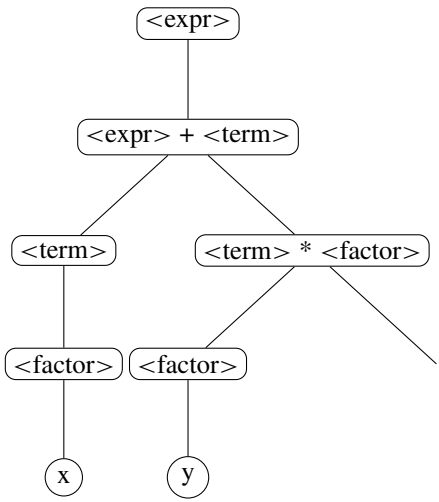
- Since we have ended this branch of the tree with a terminal symbol, this symbol is matched with the input. Our input is x+y*z, the x is matched and the remaining input is +y*z. We then unwind from the recursion all the way back up to deal with the <expr> + <term> node. We have just dealt with the leftmost <expr> node and its children, so we now tackle the node +. This again is matched with the input and so y*z remains. The next step is to deal with the third node <term>. We use the
  <term> ::= <term> * <factor>
  production and the tree becomes:



- The latest <term> is given the child <factor> from the production
  <term> ::= <factor>

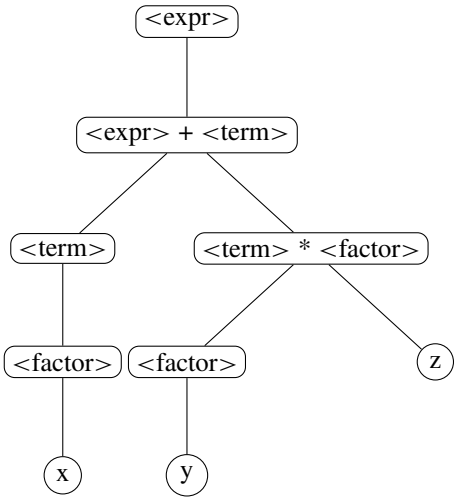


- That <factor> corresponds to `y`. We use the production
  <factor> ::= `y`



- This `y` is matched with the input and the remaining input is `*z`. The recursion returns to the `*` node which again is matched with the input, resulting in just `z` remaining. Finally we use the production
  <factor> ::= `z`

  The resulting tree is then:



- The z is matched with the input, leaving nothing on the input. Matching all tokens of the input leaving nothing is an indication of parsing success.

This detailed example illustrates how the top–down parsing process corresponds to the leftmost derivation. The order in which the productions are used is the same as the order in the leftmost derivation. However, this is not a good example to illustrate how the top–down parser can be programmed. In particular, we have glossed over the problem of how to choose the productions when there are alternatives.

4.2.3 A Top–Down Parsing Algorithm

There are easier ways of thinking about the top–down parsing process. Instead of being driven by the construction of the parse tree, one can think more about the process of the *recognition* of the input. Our starting point here is the formal grammar for the language.

Initially, the task of tree generation can be ignored so that we can concentrate instead on the matching of the terminal symbols in the grammar with the input stream presented to the parser. This formulation of the top–down parser also starts with the starting symbol of the grammar. Suppose the starting symbol S is defined as $S \rightarrow AB$. Our goal of recognising S can then be restated as recognising the subgoal A and then recognising the subgoal B. Then, recognising A and B requires the recognition of their subgoals, and so on.

If instead S is defined as $S \rightarrow A|B$, then S is recognised by recognising an A *or* by recognising a B. When terminal symbols appear in the right-hand sides of production rules, these symbols are matched with the corresponding instances of those tokens in the input string. If the match fails, then the parsing fails. As the parsing process

continues, more and more tokens from the input will be matched and in the case of a successful parse the process will end with all input tokens having been matched.

This approach sounds both attractive and feasible and the obvious way of implementing it is to associate a programming language function with each non-terminal symbol and the role of the function is to recognise an instance of that non-terminal. For example, given the definition of the non-terminal P as $P \rightarrow QR$ a function `P()` could be written as follows.

```
void P() {
    Q(); R();
}
```

Unfortunately dealing with a definition of the form $P \rightarrow Q|R$ is harder. An obvious approach is to allow the recognising functions to return a value indicating success or failure. Then `Q()` could be called, if it returned failure, then `R()` is called. However this is in general wrong because in running `Q()`, tokens from the input will have been read and consumed, and should `R()` then be called it will start reading at the point where `Q()` left off rather than from the beginning.

Again, there seems to be an obvious answer which involves the use of *backtracking*. After the failed call to `Q()` the input could be backspaced to the state it was in just before `Q()` was called. Then `R()` is called and it will read the correct input. And for some grammars this approach will work correctly. But for other grammars, problems remain.

Consider the scenario where `Q()` matches, returning success. Suppose also that `R()` could have matched, matching a *different* substring to that matched by `Q()`. Which match should be chosen? It depends on what follows the matching of P in the grammar. This is beginning to look complicated because it may require the handling of backtracking amongst potentially all of the recognising functions.

In practice, backtracking should be avoided if at all possible. The complexity of the parsing code increases a great deal and the parsing may become considerably less efficient. Fortunately this does not mean that productions of the form $P \rightarrow Q|R$ cannot be handled by a top–down parser. It just means that some constraints have to be placed on the details of such productions.

4.2.3.1 Lookahead

To recognise P defined as $P \rightarrow Q|R$ we really have to know whether to select the Q route or the R route. If we are to avoid backtracking then there has to be some characteristic of the definitions of Q and R to enable the selection to be made. The way in which this is commonly done is to make use of *lookahead*. Suppose the parsing process is at a stage where P has to be recognised. A certain number of tokens have already been consumed from the input. Suppose also that it is possible to examine (without preventing them from being read again) some tokens beyond the

current point of input. It may be that the identity of these lookahead tokens allows us to determine whether to follow the Q path or the R path. For example, if the definition of Q is $Q \rightarrow a \ldots$ and the definition of R is $R \rightarrow b \ldots$ where a and b are terminal symbols, then if the first lookahead token is a we follow the Q path and if the first lookahead token is a b we follow the R path.

This idea of lookahead is essential for practical parsers for non-trivial grammars. But managing many tokens of lookahead adds to parser complexity especially if the tokens are being input as they are required. Perhaps surprisingly just a single token lookahead will suffice for the syntax of most programming languages. Managing a single token lookahead is straightforward and will be discussed in Chap. 5.

Consider this simple example illustrating how lookahead can help.

$S \rightarrow Az|z$
$A \rightarrow xA|B$
$B \rightarrow y$

Here, x, y and z are terminal symbols. In the code to recognise an S, if the current lookahead token is a z, we use the production $S \rightarrow z$, otherwise (if it is an x or a y) use $S \rightarrow Az$. Similarly in the code to recognise an A, if the current lookahead token is a x, we use the production $A \rightarrow xA$ and if it is a y use the production $A \rightarrow B$. If the current lookahead token is a z this indicates a parse error (see Sect. 4.2.6).

If the lookahead can allow the correct alternative to be chosen in a production involving alternation, as in the example above, then a parser can be written without having to make use of any backtracking. The parsing process has been made *deterministic*. This allows the construction of a *predictive parser*. The traditional and practical way of coding such a parser is, as we have seen, to associate a function with each non-terminal symbol whose task is to recognise an instance of that non-terminal. These functions call each other according to the syntax rules of the grammar, matching terminal tokens from the input as they go. This is a *recursive descent parser* and many examples, including the parsing code for this language, will be presented in Chap. 5.

Unfortunately, this approach to parsing will still have difficulties with some particular constructs found in production rules. Fortunately, there are usually uncomplicated solutions.

4.2.3.2 Factoring

Suppose there is a production rule of the general form $A \rightarrow \alpha\beta|\alpha\gamma$. Writing a recognising function for A is problematic because our current lookahead symbol (presumably something that can start an α) is not enough to enable us to determine whether to follow the $\alpha\beta$ or the $\alpha\gamma$ branch. The solution here is not to resort to more lookahead. Instead this production rule can be rewritten as:

$A \rightarrow \alpha A_1$
$A_1 \rightarrow \beta|\gamma$

This technique is known as *factoring*. There is now no alternative in the first production so that can be coded easily, and in the second production the current lookahead token should hopefully be capable of predicting which branch to take, depending on the definitions of β and γ. There are of course grammars that are best dealt with by increasing the parser lookahead, but this comes at a cost of increased complexity.

4.2.3.3 Left Recursion

There is a second issue which occurs often in the definitions of traditional programming languages. Consider a production of the form $E \rightarrow E + T|T$. This style of production is often seen in programming language grammars, defining an expression-like structure. When recognising an E, determining whether to follow the $E + T$ or the T path needs consideration, as described above in the section on factoring. But there is another specific problem in the handling of $E \rightarrow E + T$. The task of recognising an E starts off by recognising an E which in turn starts by recognising an E.... This process will never terminate because it consumes no input at each stage. This is a *left-recursive* production, and left recursion always causes problems for top–down parsers. In order to make the recognition terminate, the alternative production $E \rightarrow T$ has to be used somehow. The grammar has to be modified.

Fortunately there are easy solutions. There is a standard transformation that can be applied to these left-recursive productions. A production of the form $A \rightarrow A\alpha|\beta$ is transformed into the productions

$$A \rightarrow \beta A_1$$
$$A_1 \rightarrow \alpha A_1|\varepsilon$$

where ε represents the empty string. So $E \rightarrow E + T|T$ can be transformed into

$$E \rightarrow T E_1$$
$$E_1 \rightarrow +T E_1|\varepsilon$$

The left recursion has been removed at the cost of an extra non-terminal symbol (E_1) and a little extra complexity. However, in many cases of left recursion occurring in the definitions of programming languages, there is a simpler solution. Left recursion is replaced by iteration. The power of EBNF allows us to rewrite $E \rightarrow E + T|T$ as

$$E \rightarrow T \{ + T \}$$

This turns out to be easy to implement and examples appear in Chap. 5.

Unfortunately, this is not quite the end of the story for this form of left recursion. The replacement of left recursion by iteration may need to be done with care when the parse tree nodes are generated since productions of the form $E \rightarrow E + T|T$ imply that the $+$ operator is left-associative and this has to be carried forward even when the $E \rightarrow T \{ + T \}$ formulation is used. However this is not difficult. Also it should be noted that left recursion is sometimes not so easy to spot. For example

$A \rightarrow \alpha | B\beta$
.
.
$B \rightarrow A\gamma | \delta$

or even

$A \rightarrow BA | \alpha$
.
.
$B \rightarrow \ldots | \varepsilon$

In these examples, multiple productions may be involved in the left recursion and dealing with this neatly and reliably may need some care.

4.2.3.4 Ambiguity

There is another problem area, but not confined to top–down parsing. A grammar is *ambiguous* if there exists more than one parse tree representing a given sentence. For example, returning to the grammar in Fig. 4.1 for the trivial arithmetic language, we can simplify the grammar by just allowing the + operator.

<expr> ::= <factor> | <expr> + <factor>
<factor> ::= x | y | z

There is nothing wrong with this grammar and the + operator is left-associative. An alternative grammar could be written as

<expr> ::= <factor> | <expr> + <expr>
<factor> ::= x | y | z

This grammar is ambiguous because sentences such as `x+y+z` have more than one parse tree as shown in Fig. 4.3. Specifically in this example, the + operator is being defined as being both left- and right-associative. It has to be one or the other.

Detecting ambiguity in a grammar is not always easy. But removing it saves difficulties later.

4.2.4 Classifying Grammars and Parsers

In the discussion above about top–down parsers, it became clear that certain constraints had to be put on the grammar to make it amenable to top–down parsing. The Chomsky hierarchy points to type 2 context-free languages as the class that should be targeted for use as programming languages but the type 2 languages are not subdivided to indicate, for example, which are appropriate for top–down parsing.

A useful terminology for describing grammars exists and this terminology gives some indication of how the grammar should be parsed. An *LL(k)* grammar can be

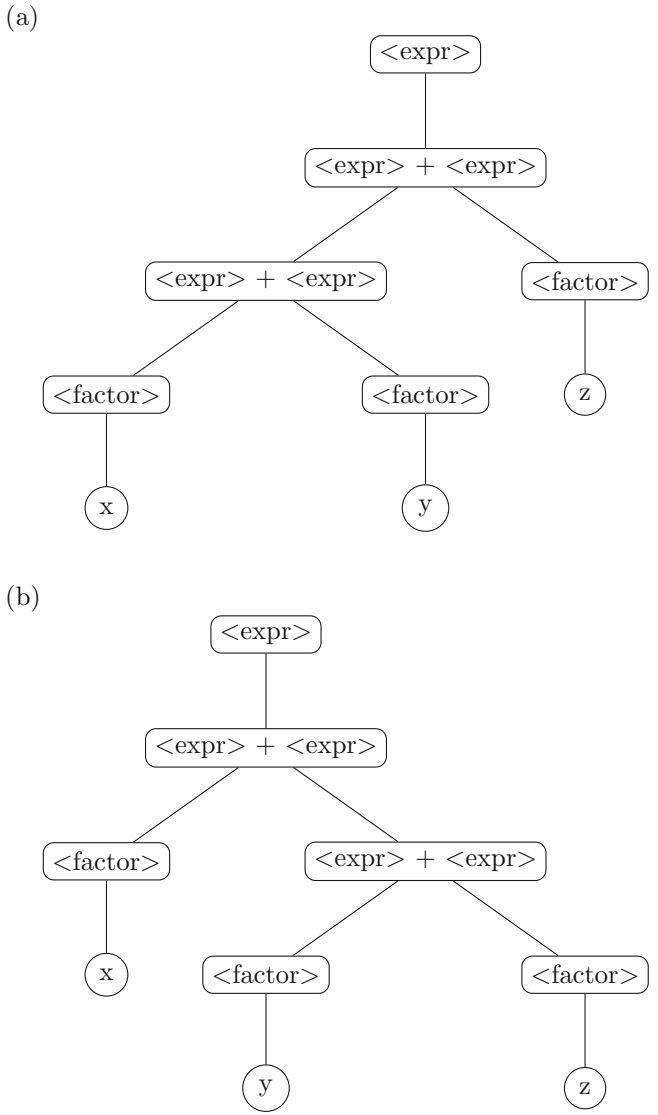


Fig. 4.3 Two parse trees for `x+y+z`

parsed using a top–down parser using a lookahead of at most k symbols. The LL(k) parser reads the input from ***L****eft* to right (i.e. from the beginning to the end) and produces a ***L****eftmost* derivation. It needs up to k symbols of lookahead to determine which production to apply at any stage. The *LL(1)* grammars are widely found in practice because they are sufficiently powerful to cover the needs of most programming languages and they are simple to implement using a top–down parser. As we

have already seen in this chapter these parsers are simple and efficient and do not require any backtracking.

In contrast, an *LR(k)* grammar can be parsed using a bottom-up parser using a lookahead of at most k symbols. The LR(k) parser reads the input from ***L****eft* to right and produces a ***R****ightmost* derivation in reverse. The parser reads the input from left to right and makes a decision on which production to use next on the basis of the symbols already read and the k symbols of lookahead. These parsers will be examined in the next section. They are harder to implement than top–down LL(k) parsers but nevertheless are efficient and require no backtracking. They are important primarily because of their power. It is possible to construct some sort of LR parser for virtually all programming languages for which Chomsky type 2 grammars exist. Furthermore all LL(k) grammars are LR(k). The practical consequences of this are discussed in Chap. 5 where it is argued that both approaches to parsing have a role in the field of high-level language compilers and other software tools.

4.2.5 Bottom-Up Parsing

Top–down parsers start with the starting symbol. Bottom-up parsers, in contrast, start with the first token of the input. Their mode of operation may seem very much more intuitive than top–down parsing. They repeatedly match symbols from the input with the strings on the right-hand sides of production rules, replacing the matched strings with the corresponding left-hand sides. This continues until, hopefully, just the starting symbol remains. This process has already been illustrated in Sect. 2.3.3 where the choice of which substring to reduce at each stage is shown to be critically important in order to achieve a correct parse.

The correct order of matching for the bottom-up parser is defined by the rightmost derivation. The bottom-up parser has to perform the *rightmost derivation in reverse*. This produces the *canonical parse*. For example, if we start with the input `x+y*z` from the example in Sect. 4.1.1 using the grammar in Fig. 4.1 and reverse the rightmost derivation, we can see how the bottom-up parser should operate:

`x+y*z`	(use $<$ factor $>\rightarrow$ `x`)
$<$ factor $>$ `+y*z`	(use $<$ term $>\rightarrow<$ factor $>$)
$<$ term $>$ `+y*z`	(use $<$ expr $>\rightarrow<$ term $>$)
$<$ expr $>$ `+y*z`	(use $<$ factor $>\rightarrow$ `y`)
$<$ expr $>$ `+` $<$ factor $>$ `*z`	(use $<$ term $>\rightarrow<$ factor $>$)
$<$ expr $>$ `+` $<$ term $>$ `*z`	(use $<$ factor $>\rightarrow$ `z`)
$<$ expr $>$ `+` $<$ term $>$ `*` $<$ factor $>$	(use $<$ term $>\rightarrow<$ term $>$ `*` $<$ factor $>$)
$<$ expr $>$ `+` $<$ term $>$	(use $<$ expr $>\rightarrow<$ expr $>$ `+` $<$ term $>$)
$<$ expr $>$	

This looks promising in that it could give a pointer to a particularly simple route to an implementation. However, a major problem remains. This example is easy because we already have the rightmost derivation. But in general we will not have this derivation to hand, so the bottom-up parser has to decide itself *which* substring to reduce at each stage. This substring that is matched with the right-hand side of a production and replaced by the corresponding left-hand side is called the *handle* and the key problem of bottom-up parsing is the identification of the handle at each stage.

The algorithm used to identify the handle obviously has to be based on the identity of the lexical tokens read from the input. Ideally, the number of tokens needed to identify the handle should be limited in order to achieve an efficient parser. Furthermore, backtracking should not be necessary, again for efficiency reasons. So the identification of the handle should be possible by just considering the tokens of the handle itself together with tokens in the immediate locality of the handle. In other words, we may need to examine some *left context*, the handle itself and some *lookahead*. Practical algorithms for handle identification will be presented in Chap. 5.

4.2.6 Handling Errors

The compiler has to be able to deal with syntactically incorrect input. The user of a compiler cannot guarantee that the input presented to the compiler always conforms to the formal syntax of the language and so it is important for the compiler to identify these errors and provide the programmer with appropriate information to make the correction of these errors easy. This turns out to be a surprisingly difficult requirement to fulfil.

We have already looked at some of the issues concerned with errors detectable by the lexical analyser (see Sect. 3.2.2.7). Dealing with errors found in the syntax analyser is rather more complicated. Furthermore the syntax analyser is likely to have to deal with the errors detected and reported by the lexical analyser too. Error handling requires the detection of the error (generally easy), producing an appropriate error message for the compiler user (again, generally easy) and finally *recovering* appropriately from the error (generally difficult).

Detection and reporting of a syntax error in the syntax analyser usually presents no real difficulty. At all stages in the parsing process a top–down or a bottom-up parser will be aware of the set of lexical tokens that can be validly accepted in that particular location (this is indirectly specified by the grammar) and should an unexpected token be read, the syntax analyser can generate an error message of the form "*On source line 123 the token '.' was found, but ';' was expected*". However, there are often slight complications here because precise localisation of the *cause* of the error, as far as the programmer is concerned, may not be possible because the error may be detected some way away from the mistake actually made by the programmer. For example, consider this fragment of a C program:

```
if (a>b) {
  a=0;
  b=10;
  c=100;
}
```

If the `{` token is omitted from the line starting `if`, the error will probably be reported on the line containing the `}` or even beyond. Whether or not this is a major issue is not clear, but getting it "right" may not be easy because it may not be obvious what "right" means in this context.

Real difficulties can occur when recovering from the error. It is rarely acceptable for the syntax analyser to report the error and then stop, relying on the programmer to correct the error and restart the compilation process. In most circumstances, the correct thing to do is to allow the syntax analyser to continue so that it can report any remaining errors in the source file. But the difficulty is that it is usually very awkward to predict how best to restart the syntax analyser. Which syntactic structure should it be expecting after detecting the error? If a strange token was erroneously added to the input and found, for example, when the syntax analyser was parsing an expression, that token could be skipped and the expression parsing resumed. But if that strange token resulted from *omitting* another token, then skipping that strange token may not be the best thing to do. Ideally we would like just a single error message for each distinguishable error in the input and so there is a need for a mechanism to allow the syntax analyser to resynchronise itself with the input. We will look at a practical approach in Chap. 5. In some cases, avoiding over-reporting syntax errors is easy to achieve. For example, one should only report an undeclared variable the first time it is used. But the general case of over-reporting errors is somewhat harder.

4.3 Tree Generation

So far in this chapter, the focus has been on the syntax analyser's role in reading the stream of lexical tokens and ensuring that they form a syntactically correct program. The syntax analyser has another essential task to perform. It has to generate some output for the next phase of compilation. This output is conventionally in the form of a tree and this tree is generated as the parser, top–down or bottom-up, performs reductions, matching right-hand sides of productions. A full parse tree can be generated by creating a new node each time a reduction is made. Later stages of compilation will not need all this data and structure, and so some simplifications to the tree structure can be made. This simplified structure is the *abstract syntax tree (AST)*. An example of the simplification is shown in Fig. 2.4.

Generating this form of tree is not complicated, but requires some planning. The set of node types should correspond closely with the formal syntax of the language, omitting nodes for simple productions of the form $E \rightarrow T$ and for tokens that are now redundant such as parentheses in expressions, where grouping is reflected

by the tree structure. Each node has to contain some indication of the node type (e.g. “if statement”, “identifier”, “string constant”, and so on), together with the corresponding data for that node type. The form of the data depends on the node type. For example, an integer constant may contain a binary or string representation of the value of the constant. A node for a variable may contain a string representation of the variable name or, more likely, a pointer to that variable’s entry in a symbol table. Creating the node and initialising the data should not be difficult and this code can be added easily to the parsing code. But the node has to be linked in with an existing partial tree structure. This can be done by designing the parsing functions to return pointers to these tree nodes to their callers. For example, in a top–down parser a function to parse a simple assignment statement will generate a new “assignment” node with space for two subtrees. One subtree will be set to be a pointer to a node representing a variable, returned by the call to the variable recognition function called in simple assignment, and the other subtree will be set to the value returned by the call to the expression recognition function. The simple assignment function then returns a pointer to the newly-created node. Practical examples are given in Chap. 5.

4.4 Conclusions and Further Reading

This chapter has highlighted the relationship between derivation and parsing, how a top–down parser generates the leftmost derivation and the bottom-up parser generates the rightmost derivation in reverse. A major issue in the development of a compiler is the choice of parsing strategy for the language being compiled. Top–down parsers can in general be written by hand or by parser generator programs, directly from the grammar specification. Bottom-up parsers can offer greater parsing power (dealing with harder grammars) but at the expense of code complexity. They are rarely written by hand, instead requiring the use of a parser generator tool. We will look at practical examples in later chapters.

Practical parsers need to be able to handle errors in their input, reporting and recovering appropriately. They also need to generate an abstract syntax tree for passing on to the semantic analyser—the next phase of compilation. There is considerable flexibility in the design of the AST.

There is an extensive literature on the theory of grammars and parsing. Perhaps one of the most famous textbooks in this area is [1] and a more compiler-oriented coverage of context-free grammars and parsing is found in [2]. Ambiguity in programming language grammars is covered by [3], together with advice on the transformation of grammars to make them unambiguous and suitable for traditional parsing techniques. A classic reference for LR parsing is [4].

Most programming language definitions are based on a context-free grammar written in such a way as to avoid the use of constructs that cause difficulties when writing a parser. But when the grammar is not quite so helpful, there are standard transformations available to turn the grammar into a form that is much easier to implement. For example, a context-free grammar can be transformed into Greibach

Normal Form, yielding something that can be parsed easily, with a bound on the complexity of the parse. This, and the relationship between context-free languages and pushdown automata is covered in [1].

Compiler writers tend to be rather conservative in their choice of parsing techniques for conventional programming languages. There are good, practical reasons for this, but there are times where different and maybe more powerful techniques are required. A comprehensive description of a wide range of parsing techniques is presented in [5], applicable not just to the compiler's syntax analyser. An example of a powerful parser is the Earley parser [6], based on dynamic programming. It is capable of parsing any context-free language. Another powerful parser uses the CYK algorithm and again makes use of techniques of dynamic programming. It is described in [1].

Exercises

4.1 Produce a BNF or EBNF grammar for a simple language designed to be used as a numerical calculator. It should handle the four operators `+`, `-`, `*` and `/` with the conventional precedences, allow parentheses and operate on integer constants. Use this grammar to produce a calculator program, reading a line of input containing an arithmetic expression and outputting the numerical result.

4.2 Extend the grammar for the numerical calculator to include other operators such as `%` to perform percentage calculations, trigonometric functions, hexadecimal arithmetic, ... This may not be trivial.

4.3 Extensive use of productions of the form $E \rightarrow E + T|T$ has been made in this chapter and this form of production fits in well with the generation of a syntax tree. But to avoid the left recursion, $E \rightarrow E + T|T$ is often rewritten as $E \rightarrow T\{+T\}$. How would you generate a correct syntax tree from this form of production?

4.4 Produce an outline design for an abstract syntax tree for a programming language of your choice. Are there any syntactic constructs that cause particular difficulties? Are names/symbols/variables better stored in the tree or in a separate symbol table? By hand, try generating a tree in your format for a simple program.

4.5 Look at the syntax rules of a programming language of your choice and attempt to find rules that might cause difficulty for a predictive top–down parser. Can the rules be rewritten to avoid these difficulties?

4.6 Many high-level languages make use of tokens that appear to be redundant. For example, some programming languages support a `while ... do` statement and the `do` is sometimes, strictly speaking, redundant. Why is this done—after all, it's more typing for the programmer… ?

4.7 A list of items is defined as <list>→<item> <list> | <item> where <item> is defined as any of the lower case letters. The string `abcd` is a list. Generate

its abstract syntax tree. Suppose the list is defined as <list>→<list> <item> | <item>. Generate the new abstract syntax tree. Comment on the differences.

4.8 Do some research on the famous `if...then...else` ambiguity and find out ways in which the syntax of conditional statements changed in order to remove the ambiguity.

4.9 Insert some deliberate mistakes into a high-level language program and investigate how the compiler deals with them. Can you produce an example where the error message is really misleading? What error detection and recovery features would you like to see in a compiler designed for the novice programmer?

References

1. Hopcroft JE, Ullman JD (1979) Introduction to automata theory, languages and computation. Addison-Wesley Publishing Company, Reading
2. Aho AV, Lam MS, Sethi R, Ullman JD (2007) Compilers – principles, techniques and tools, 2nd edn. Pearson Education, Upper Saddle River
3. Mogensen TÆ (2011) Introduction to compiler design. Undergraduate topics in computer science. Springer, Berlin
4. Aho AV, Johnson SC (1974) LR parsing. ACM Comput Surv 6(2):99–124
5. Grune D, Jacobs CJH (2008) Parsing techniques – a practical guide. Monographs in computer science. Springer, New York
6. Earley J (1970) An efficient context-free parsing algorithm. Commun ACM 13(2):94–102

Chapter 5
Practicalities of Syntax Analysis

Chapter 4 provided an introduction to the process of parsing, giving some of the theory and background. In this chapter we adopt a somewhat more practical approach to syntax analysis and we look in detail at the two most popular techniques used for the construction of syntax analysers for programming language compilers and similar tools. After some introduction to these two approaches, together with some simple examples, we move on to the development of two syntax analysers for the DL language, showing how they can be developed and augmented to produce the data structures needed by the next phase of compilation.

The differences between the *top-down* and *bottom-up* approaches have already been described, but the actual choice of a parsing technique for any particular project needs a little more discussion. In practice, the favourite top-down technique is the predictive top-down parser, usually with just one token of lookahead (i.e. LL(1)). Sometimes greater lookahead can be an advantage, but comes at the cost of increased coding complexity. Furthermore, backtracking top-down parsers are not in general needed or appropriate for programming language parsing.

Some of the details of bottom-up parsing will be presented later in this chapter. Although there are many different algorithms that can be used to control a bottom-up parser, most practical implementation make use of a parser generator program to produce an LR(1)-style parser.

Bottom-up parsers can offer the advantage of being able to cope with more complex grammars than those usable with a top-down parser. There are common grammar features that need to be transformed and removed before a conventional top-down predictive parser can be used. The use of factoring and left recursion elimination are important here. These transformations are usually possible for the grammars defining constructs commonly found in today's programming languages, but sometimes the transformations may need to be extensive and may require some care and skill to make them effective.

Therefore, to simplify the choice of parser implementation technique, we can restrict the set of possibilities by examining just four approaches: top-down or bottom-up and hand-written or machine-written using a parser generating tool.

- Top-down, hand-written: a popular and widely used technique, particularly for simpler programming languages. The grammar should permit the construction of predictive parsers and backtracking should be avoided.
- Top-down, machine-written: there are good software tools available, generating top-down parsers from a formal grammar specification, in a variety of languages. For example, *JavaCC* generates top-down parsers in Java and can generate lexical analysers too. Some of these tools (*JavaCC* included) can deal painlessly with grammars requiring more lookahead than just one token.
- Bottom-up, hand-written: not recommended! The task of generating the controlling rules of a bottom-up parser is complex and error-prone even for simple grammars. For a parser for all but the simplest of grammars, this would be an unwise choice.
- Bottom-up, machine-written: there are many powerful software tools generating bottom-up parsers and they have been used extensively for the construction of real compilers. Tools such as *yacc* [1], *bison* [2] or CUP [3] have been used for a huge variety of parsing problems.

This book concentrates on two of these categories—top-down, hand-written and bottom-up, machine-written. But as far as their use is concerned, the tools for generating top-down parsers appear quite similar to the bottom-up parser generators in that they both require BNF-like grammar specifications. So the experience of using the bottom-up parser tools is valuable when learning how to use a top-down parser-generating tool and vice versa.

There are so many different ways of writing a parser. They vary enormously in complexity, efficiency, power, application area and popularity. We will be concentrating on just a tiny subset of the approaches available, but the parsers chosen cover the most popular techniques appropriate for today's programming languages.

5.1 Top-Down Parsing

Section 4.2.3 presented the basic ideas of practical top-down parsing. Ideally, recognising code can be written directly from the language's BNF or EBNF or equivalent rules, structured so that there is a recognising routine, function or procedure for each of the language's non-terminals. For example, the production rule $P \rightarrow QR$ is handled by the code

```
void P() {
    Q(); R();
}
```

To recognise a P, a Q followed by an R have to be recognised. And once recognising code has been implemented for all the non-terminals in the language, the code recognising the starting symbol can be called, having the effect of recognising complete programs written in that language. Before looking at the practicalities of coding all this recognising code, we should understand exactly what is being achieved here.

These functions simply *recognise* an instance of the corresponding non-terminal construct. The function `P` above just recognises an instance of P. In general these recognising functions produce no output unless an error is found. But programs that produce no form of output are rarely useful. We will see later in this section how these recognising functions can be augmented to produce nodes in a syntax tree and also produce output to help debugging.

The need for *lookahead* has also been highlighted. For example, writing a predictive recognising function for $P \rightarrow Q|R$ requires code to determine in all circumstances whether to follow the Q route or the R route. This is done by having one or more tokens in hand and on the basis of the identity of these tokens, the code can decide which route to follow. Hand-written top-down parsers generally rely on just a single token lookahead throughout the parsing process. Some parsers may use just a single token lookahead except when recognising particular non-terminals where greater lookahead may be temporarily required.

5.1.1 A Simple Top-Down Parsing Example

This is all best illustrated by an example (from Sect. 4.2.3).

$S \rightarrow Az|z$
$A \rightarrow xA|B$
$B \rightarrow y$

Here is a complete C program to recognise strings of this language.

```
#include <stdio.h>
#include <stdlib.h>

int ch;

void error(char *msg) {
  printf("Error - found character %c - %s\n",ch,msg);
  exit(1);
}

void b() {
  if (ch == 'y') ch = getchar();
  else error("y expected");
}

void a() {
  if (ch == 'x') {
    ch = getchar();
    a();
  }
  else b();
}
```

```
void s() {
  if (ch == 'z') ch = getchar();
  else {
    a();
    if (ch != 'z') error("z expected");
    else ch = getchar();
  }
  printf("Success!\n");
}


int main(int argc, char *argv[])
{ ch = getchar();
  s();
  return 0;
}
```

Compiling and running this program gives output of the form:

```
$ ./simpletopdown
xyz
Success!
$ ./simpletopdown
xxxxyz
Success!
$ ./simpletopdown
xxxxz
Error - found character z - y expected
$ ./simpletopdown
z
Success!
```

The program includes a global declaration for a character `ch`. As the parser runs this variable always contains the *next* character from the input—it is the *lookahead*. In this simple language, all the lexical tokens are single characters and so the lexical analyser can be replaced by the call to the standard function `getchar` to read the next character from the input. Before the parsing process starts by a call to the `s` function, the lookahead is initiated by calling `getchar` to read the first character of the input.

The function `b` checks that the current token is a `y`. If so, it reads the next input character, but if not, it issues an error message. Here, a function `error` is called which outputs a message and then halts the execution of the parser.

The function `a` checks that the current token is an `x`. If so, it skips over it and calls `a` recursively, corresponding to the grammar rule $A \rightarrow xA$. Otherwise it calls `b`.

Finally, the function `s` checks for `z` and if found, it reads the next token and the parse succeeds. Otherwise, `a` is called, and `s` checks for a `z`. If found, it reads the next token and the parse succeeds again, otherwise an error is generated. There is a minor issue in this function in that `s` finishes by performing a lookahead. Strictly speaking this is not necessary but in this case, no harm is done and it emphasises the need for consistent lookahead throughout the parsing process.

This code can be written painlessly directly from the BNF rules. The rules needed no modification before the parser could be written. A key issue to notice is the management of lookahead. It works in a very similar way to the one character lookahead already shown in the hand-coding of lexical analysers. The rule to follow is to ensure that on entry to any of the non-terminal recognising functions, the lookahead variable `ch` should already contain the first token of the non-terminal being recognised. The first step in debugging a syntax analyser of this style is to check that the lookahead is being carried out consistently. Making a mistake with the lookahead is a very common coding error.

What is next? This piece of code is just a recogniser. It does indicate parsing success by outputting a comforting message, but we will need to add code to handle tree generation (see Sect. 5.3 to see how this is done). Also, a simplistic approach to error handling has been adopted and, specifically, no attempt has been made to do any error recovery. This issue is examined later in Sect. 5.5. But notice that this parser can easily generate informative error messages indicating what was expected and what was actually found.

5.1.1.1 Practicalities

Faced with a complete BNF or similar grammar the task of coding a predictive top-down parser may seem daunting. Experience suggests that the best way of tackling the task is as follows.

- Firstly, just check again that the lexical analyser works. Trying to debug the syntax analyser when the problem actually lies in the lexical analyser is a miserable task.
- Write a *recogniser* first and do not worry at all about the generation of the tree. Include code to output tracing messages indicating which recognising functions are being called. It may be possible to develop the recogniser in stages by starting off with just a subset of the grammar and gradually adding code to recognise further non-terminals until the recogniser is complete. Do not worry about error recovery at this stage, but error detection should come out semi-automatically in the recognition process. Finally, test extensively. This is somewhat tiresome because the tracing output needs to be checked and associated with the grammar-defined structures of the source program.
- Add code to generate the syntax tree. Write a simple function to display the tree in a human-readable form. Do not remove the tracing code inserted in the last step. It can just be disabled and then re-enabled if necessary to help with debugging. Section 5.3 covers tree generation.
- Consider error recovery. See Sect. 5.5.
- Program *defensively* throughout. For example, check for out-of-range parameters, ensure consistency in data structures and so on. Although the code is *supposed* to get this right, double checking does no real harm and can be of huge benefit

in debugging. Some programming language implementations support an `assert` facility which can be a useful debugging aid.
- Test repeatedly, effectively and intelligently.

5.1.2 Grammar Transformation for Top-Down Parsing

In Sect. 4.2.3 several potential problem areas for top-down parsers were mentioned. While designing a predictive top-down parser from the grammar it is essential to check that the parser stays *predictive*. This means that whenever there is an alternative in the grammar, the lookahead can unambiguously determine which of the alternatives to follow. In some cases, factoring can resolve the problem but in other cases, the grammar may need modification. In practice this seems to be a rare problem.

Left recursion is frequently found in programming language grammars and dealing with it is usually a routine task. For example, a production of the form $E \rightarrow E + T|T$ is best handled by rewriting the production in an EBNF style as $E \rightarrow T\{+T\}$ and coding the recogniser as follows:

```
void E() {
  T();
  while (ch == '+') {
    ch = getchar();       /* or call the lexical analyser
                             for the next token */
    T();
  }
}
```

The recursion in the original BNF production has been replaced by iteration in the EBNF rule and implemented using a `while` statement. It is of course also possible to deal with the left recursion using the standard grammar transformation described in Sect. 4.2.3 but in practice the minor reformulation as shown above will suffice.

The problem of ambiguity has also already been mentioned. The proper solution to this is of course to remove the ambiguity from the grammar. But it may be possible to resolve the ambiguity within the parser in an ad hoc way.

5.2 Bottom-Up Parsing

The price paid for the potential power of bottom-up parsing is a significant increase in software complexity, effectively making it advisable to make use of parser generator tools, rather than coding the parser directly in a conventional programming language. Using these specialised tools certainly simplifies the process, but it is important to have some knowledge of what is going on in the operation of a bottom-up parser.

5.2.1 Shift-Reduce Parsers

There are many different approaches to bottom-up parsing but most of the practical and widely used implementations are based on *shift-reduce parsers*. These parsers perform a single left-to-right pass over the input without any backtracking. The parser itself is simple and easy to code. It is the *control* of the parser that is more challenging.

The shift-reduce parser implements just four basic operations.

- The *shift* operation reads and stores a token from the input.
- The *reduce* operation matches a string of stored tokens with the right-hand side of a production rule, replacing the stored tokens with the left-hand side of the production rule.
- The *accept* operation indicates that the parse has succeeded.
- The *error* operation signals a parse error.

The problem, of course, is knowing when to shift, when to reduce, when to accept and when to error. But before returning to this issue of control it is sensible to consider an implementation.

The natural way of implementing a shift-reduce parser is to use a stack for the storage of input tokens and their replacements from the left-hand sides of productions. Let us go back to the familiar grammar for our trivial arithmetic language in Fig. 4.1, repeated here:

<expr> → <term> | <expr> + <term>
<term> → <factor> | <term> * <factor>
<factor> → x | y | z

and suppose we wish to parse the sentence x+y*z. In this example, we show the contents of the stack (which starts off empty), the remaining input and the parser action (shift, reduce, accept or error) at each stage. The base of the stack is on the left.

Stack	**input**	**action**
	x + y * z	shift x
x	+y * z	reduce using < factor > → x
< factor >	+y * z	reduce using < term > → < factor >
< term >	+y * z	reduce using < expr > → < term >
< expr >	+y * z	shift +
< expr > +	y * z	shift y
< expr > +y	*z	reduce using < factor > → y
< expr > + < factor >	*z	reduce using < term > → < factor >
< expr > + < term >	*z	shift *
< expr > + < term > *	z	shift z
< expr > + < term > *z		reduce using < factor > → z
< expr > + < term > * < factor >		reduce using < term > → < term > * < factor >
< expr > + < term >		reduce using < expr > → < expr > + < term >
< expr >		

Comparing this sequence of parsing steps with the rightmost derivation of `x+y*z` from Sect. 4.1.1, we see that this parser is producing the rightmost derivation in reverse, as expected for a bottom-up parser.

This is all very well, but the choices of parser actions here have been done without any explanation or justification. There has to be a mechanism which, given the contents of the stack and the input remaining, tells the parser whether to shift, reduce, accept or error. We need to be able to identify the handle—the next substring to be reduced. Plainly, relying on the contents of the entire stack and all the input pending will result in a parser which will be powerful but at the cost of excessive complexity. At the other extreme, we can consider what happens if the parser is controlled by the single token at the top of the stack and the next token of the input (the lookahead token). We can envisage a table indexed by the identities of these two tokens and the content of that table tells the parser what to do. This is the basis of *precedence parsing* and provides a simple parsing technique for just a restricted set of grammars. For further details, see [4].

The design of this simple precedence parser gives us the idea for a more general approach for designing shift-reduce parsers. A table, indexed by stack tokens or a function of stack tokens and also indexed by lookahead tokens is used to drive the parser. It does not have to be a two-dimensional table. We can resort, for example, to three dimensions, indexed perhaps by the token at the top of the stack and *two* tokens of lookahead. But for reasons of table size and code complexity, it makes sense to retain just two-dimensional parsing tables. There are many different parsing techniques that can be implemented in this way and they vary in what indexes the tables, the size of the tables, the complexity of the table generation process, the range of grammars the parser can cope with, the difficulty of understanding the parsing process and so on. The general LR (left-to-right, rightmost derivation) parser can be implemented in this way, together with variants such as LALR (look-ahead LR parser) and SLR (simple LR). In these parsers, the table is indexed by the lookahead token and also, in effect, by a value obtained from the identity of the current left context. Many textbooks provide the theoretical background—for example, see [5].

Setting up these tables by hand is in general a painful process, time-consuming and error-prone. However, there are now many practical bottom-up parser-generating tools which generate complete parsers made up of the code to run the shift-reduce parser together with the controlling table. These tools make the construction of a reliable and practical parser very much easier and it is the use of a popular member of this set of tools that will be the basis of examples for bottom-up parsers in this book. Compromising between table size and parsing power means that most of these parser generators produce LALR(1) parsers, certainly adequate to parse today's programming languages.

5.2.2 *Bison—A Parser Generator*

There is a long history of software tools to generate parsers. Parsers were first studied in the very early days of computer science and it became obvious that the complex yet logically intuitive task of generating some types of parsers could be automated. Perhaps the most famous of the early parser generators is *yacc* [1] developed in the early 1970s. It generates table-driven LALR(1) parsers in C. It was incorporated into the Unix operating system distribution and used for the development of other Unix software tools as well as in many other projects. *Yacc* was not the first of such tools—*yacc* is an acronym for "yet another compiler-compiler", suggesting many predecessors. And *yacc* is still in widespread use today, although many other more recent tools have been developed to generate parsers in a range of programming languages, accepting grammar specifications in formats generally upwards compatible with *yacc*. *Yacc* was designed to pair with *lex*, the lexical analyser generating tool. Although these tools can be used independently they are often used together when generating code for a compiler-related application.

Bison [2] was developed as the GNU Project version of *yacc*. It is designed to be upwardly compatible with *yacc* and includes a few minor changes. It has been distributed widely. It too can generate table-driven LALR(1) parsers in C but includes support for other bottom-up parsing methods. Just as *yacc* was transformed into *bison*, *lex* was transformed to *flex* and *flex* and *bison* are often used together, with the syntax analyser generated by *bison* calling the lexical analyser generated by *flex* as a C function.

Why choose *bison* for the examples in this book? It is not a recently developed tool, nor is it one of the most powerful parser generators available. However, it is still very popular and in widespread use in many existing and new projects. It is stable, capable, comparatively easy to use, its syntax and style have been adopted by other packages, it generates C (and we are using C in this book), it has been ported to many architectures and operating systems and there are many good examples of its use freely available on the web. This book attempts to cover the use of *flex* and *bison* primarily by means of examples. There are many good reference manuals and tutorials for *flex* and *bison* available on the web, together with text books such as [6, 7].

Before tackling the principles of developing *bison* grammar specifications it may help to glance back to Sect. 3.4.1 where there is a brief description of the input required by *flex*. At the highest level, the input required by *bison* is identical, consisting of three sections, separated by `%%` lines.

```
definitions
%%
rules
%%
user code
```

Furthermore, the ideas behind the rules are similar. The rules consist of a list of patterns with corresponding C code. In the parser that is generated, when a pattern is matched, the corresponding C code is executed. But here, the patterns are not regular expressions. They are instead *grammar production rules* and these can be taken directly from the formal definition of the language being parsed.

5.2.2.1 A Very Simple *Bison* Example

Let us return to a simple example, used earlier in this chapter (in Sect. 5.1.1) to illustrate the top-down parsing of the grammar from Sect. 4.2.3. Here again is the grammar.

$S \rightarrow Az|z$
$A \rightarrow xA|B$
$B \rightarrow y$

We have already seen that parsing it using a hand-written top-down predictive parser poses no real problems. Generating a bottom-up LALR(1) parser using *bison* is straightforward too. Again, we start with a recogniser rather than a full parser.

```
%{
#include <stdio.h>

void yyerror(char*);
int yylex(void);
%}

%%

S:
  A  'z' { printf("S->Az, done\n"); }
  |   'z' { printf("S->z, done\n"); }

A:
  'x' A { printf("A->xA\n"); }
  | B { printf("A->B\n"); }

B:
  'y' { printf("B->y\n"); }


%%
void yyerror(char *s)
{
  printf("***%s\n",s);
}

int yylex() {
  int ch;
  ch=getchar();
  while (ch=='\n') ch=getchar();
```

```
  return ch;
}

int main()
{
  if (yyparse() == 0) printf("Parsing successful\n");
  else printf("Parsing failure\n");

  return 0;
}
```

This file is passed through the *bison* tool and a C file is generated. This C file is then compiled and when run produces this output:

```
$ ./simplebottomup
xyz
B->y
A->B
A->xA
S->Az, done
Parsing successful
$ ./simplebottomup
xxxxyz
B->y
A->B
A->xA
A->xA
A->xA
A->xA
S->Az, done
Parsing successful
$ ./simplebottomup
xxxxz
xxxxz
***syntax error
Parsing failure
$ ./simpletopdown
z
S->z, done
Parsing successful
```

The *bison* code needs some explanation.

- The three sections are clearly visible—definitions, rules, user code. The definitions section contains declarations required by the *bison* rules, but there are none needed in this example. See below for the use of `%token` declarations. It also contains C code to be copied directly to *bison*'s output, contained between the symbols `%{` and `%}`. This is a good place for including header files and declarations required by the C code in the actions in the rules section and in the user code.

In this example, we include the function prototypes for `yyerror` (called by the *bison* parser when a syntax error is detected) and `yylex` (the lexical analyser function, returning the next lexical token from the input each time it is called).

- The rules section corresponds to the set of BNF rules defining the syntax of the language. The syntax of the rules is reminiscent of BNF, but a colon is used to separate the non-terminal from its definition. In the rule defining the non-terminal `S` the right-hand side of the definition indicates that `S` is *either* an `A` followed by the character `z` *or* a single character `z`. The `|` symbol is used as in BNF to indicate alternatives. Because `A` is not enclosed in quote marks, it is taken as a non-terminal and *bison* expects its definition to appear in due course. In this grammar, `x`, `y` and `z` are all terminal symbols because they are enclosed by single quote marks. And because `S` is the first non-terminal to be defined, it is taken as the *starting symbol*.
- Each rule consists of a *pattern* and a corresponding *action*. The idea is simple. When the *bison*-generated parser runs, it matches these patterns with the input, controlled by the parsing algorithm, and if a pattern matches, the corresponding action is executed. Here, the actions are used for tracing the execution of the parsing process. We will worry about more complex actions later, specifically the generation of the parse tree.
- The final section of user code defines additional functions required by the parser. *Bison* simply copies this section straight to its output. The `yyerror` function is called by the parser when an error has been detected and here we just output a message. We will tackle the problem of error reporting and recovery later in this chapter. The `yylex` function is the *lexical analyser*. Here, it gets the next character from the input, ignoring newline characters. Although the original BNF grammar says nothing about ignoring newlines they are ignored here to result in a slightly cleaner user interface, removing the potential confusion caused by the need for a newline to send an input buffer to the running program when running interactively. Finally the `main` function is defined. This calls the `yyparse` function which is the *bison*-generated parser. If `yyparse` returns the value zero, the parse has been successful.

Does the output from the parser look reasonable? If we consider just the parsing of the xyz example, the rightmost derivation is $S \rightarrow Az \rightarrow xAz \rightarrow xBz \rightarrow xyz$. When reversed, this corresponds to the tracing generated by the parse of xyz, shown above.

This approach is looking promising. The transformation from BNF rules to *bison* rules does not seem too complicated and once the nature of the definitions and user code sections have been understood, the preparation of the complete *bison* input should be manageable. But there are some significant practical issues that have to be considered in moving from this simple example to a parser for a more complex programming language.

- The actions in this example are simplistic. We typically need to do a lot more. Specifically, there has to be a mechanism for passing data from the matching rules and the corresponding action code. For example, if we wanted to generate a node for the parse tree in the first rule of `S`'s definition, the action code has to have

access to a pointer value corresponding to the tree already generated for the `A` non-terminal. Fortunately, *bison* has a simple syntax for specifying this, and it will be shown in the next example.

- We have to think about error reporting and error recovery. Looking at the top-down parser example for this grammar, it is easy to see how to add helpful messages about expected tokens, but that seems more difficult here.
- The function `yylex` here is hand-written. Clearly, for more complex languages, using *flex* to generate the lexical analyser would be more sensible. The integration of *flex* output with *bison* code is fortunately very simple.
- It is important to spend some time checking that the original grammar has been translated accurately into the *bison* format. Errors can easily occur and can cause warnings referring to shift/reduce or reduce/reduce conflicts, or worse, can cause no warnings at all and a parser for the wrong grammar is produced. These errors can be difficult to diagnose.

At this stage it may be helpful to add a little explanation of shift/reduce and reduce/reduce conflicts, reported by tools such as *yacc* and *bison*. The parsers generated by these tools are implementations of shift–reduce parsers. If the grammar being processed results in a parser where it is possible that both a shift and a reduce are valid at a particular point in the parse, then a shift/reduce conflict will be flagged. This indicates a form of ambiguity in the grammar. Ideally it will be mended by modifying the grammar appropriately. However, both *yacc* and *bison* have special directives to permit ad hoc resolutions of this form of conflict to be made. A reduce/reduce conflict also indicates problems with the grammar. This type of conflict occurs when there are two or more rules that can be applied to perform a reduction on the same sequence of input. Again, modification of the grammar is required.

Properly dealing with any such conflicts in a grammar is important. But tracing the real cause of the problem from the *yacc* or *bison* errors and making appropriate modifications can be difficult. An understanding of the operation of a shift–reduce parser can help.

5.2.2.2 A More Useful *Bison* Example

No practical description of the *flex* and *bison* tools is complete without an example of a numerical calculator. A conventional grammar for arithmetic expressions, based on the example in Fig. 2.3 can be constructed as follows:

```
<calculation> :: = <expr> \n
<expr> :: = <term> |<expr> + <term> | <expr> - <term>
<term> :: = <factor> | <term> * <factor> | <term> / <factor>
<factor> :: = CONSTANT | ( <expr> )
```

This grammar accepts a single expression terminated by a newline character ('`\n`'). The expression uses brackets and the standard integer arithmetic operators with the conventional precedences.

```
%{
#include <stdio.h>
#include <ctype.h>

void yyerror(char*);
int yylex(void);

int ival;
%}

%token CONSTANT

%%

calculation:
  expr '\n' { printf("%d\n", $1); }

expr:
  term { $$ = $1; }
  | expr '+' term { $$ = $1 + $3; }
  | expr '-' term { $$ = $1 - $3; }

term:
  factor { $$ = $1; }
  | term '*' factor { $$ = $1 * $3; }
  | term '/' factor { $$ = $1 / $3; }

factor:
  CONSTANT { $$ = ival; }
  | '(' expr ')' { $$ = $2; }

%%
void yyerror(char *s)
{
  printf("***%s\n",s);
}

int yylex() {
  int ch;
  ch=getchar();
  while (ch==' ') ch=getchar();
  if (isdigit(ch)) {
   ival=0;
   while (isdigit(ch)) {
     ival=ival*10+(int)ch-(int)'0';  /* ignore overflow */
     ch=getchar();
   }
   ungetc(ch,stdin);
   return CONSTANT;
 }
 else return ch;
}

int main()
{
  return yyparse();
}
```

This file (called `calc.y`) is processed by *bison* (`bison calc.y` using the Linux command line) and a C file (`calc.tab.c`) is generated. This C file is then compiled (`gcc -o calc calc.tab.c`) and when `calc` is run it produces this output:

```
$ ./calc
(2+3)*(9+100)
545
$ ./calc
22/7 + 10
13
$
```

Again, this *bison* code needs some explanatory notes.

- In the definitions section we define a global variable `ival` which contains the value of the integer `CONSTANT` last read. So when the `yylex` lexical analyser function returns the token type `CONSTANT`, the syntax analyser has the value of the constant available in `ival`.
- The rules section define four non-terminal symbols—`calculation`, `expr`, `term` and `factor`. In the action code for `calculation` the final value of the calculation is printed, and this value was passed back as the value returned by the parsing of `expr`. Here we see a really important feature of *bison*. *Bison* allows the return of a value from the recognition of a symbol. In default the type of this value is an `int` but we will see later that it makes sense to change this when we are dealing with tree construction in the parser.
  In this application we use this mechanism to pass back integer values from the symbols `CONSTANT`, `factor`, `term` and `expr`. In the rule `calculation`, the result is returned through `expr` and this value is referred to in the action using the symbol `$1`—the value from the *first* token (token number `1`) in the rule.
  The next action (for the rule `expr: term`) passes the value obtained from the parsing of `term` (`$1`) as the value to be returned by `expr` (`$$`). Similarly, the action for the rule `expr: expr '+' term` adds the values obtained via `expr` and `term` and returns the sum through `expr`.
  In the actions for `factor`, the value `ival` is returned through `CONSTANT` and note that the value returned for `'(' expr ')'` is `$2`. The pseudo-variable `$1` corresponds here to the symbol `'('`.
- The lexical analyser `yylex` looks for space characters and ignores them. This is not strictly necessary because spaces are not mentioned in the original BNF grammar, but ignoring them seems like a sensible thing to do. If `yylex` encounters a digit, it accumulates the value of the constant, leaving this value in the global variable `ival` and returning the pre-defined terminal `CONSTANT`. Note the use of `ungetc(ch,stdin)` to backspace over the character *following* the constant. This character will be read next time `yylex` is called. The alternative way of managing this situation is using a single character lookahead, but this would add a little to the code complexity in this example. Note also that we do not deal with overflow properly here.

- The `main` program simply calls the parser.

There are some further practical issues here. This example shows an easy way of implementing a numerical calculator program. The rules section is built directly from the BNF specification and there are no worries about left recursion because this is a LALR(1) parser. It would be possible to add other features, operators and functions, floating point arithmetic and so on, but under these circumstances it would be sensible to resort to a lexical analyser generated by *flex*.

One obvious shortcoming of this particular implementation is that the program needs to be restarted for each new calculation. Mending this is easy. We can just make the non-terminal `calculation` a *list* of expressions separated by newlines. So `calculation` can be redefined in BNF as
<calculation> :: = <calculation> <expr> \n | ε
Bison can cope perfectly easily with this form of production, even with the use of an empty alternative. The first *bison* rule is then replaced by:

```
calculation:
  calculation expr '\n' { printf("%d\n", $2); }
  | /* empty */
```

and multiple expressions separated by newlines can be entered. It is especially important to realise that in the modification of the pattern we have introduced a new symbol `calculation` and this means that the `$1` in the action has to be changed to `$2`. Unfortunately losing the correct correspondence between symbols and the `$n` symbols in the action is a common source of error in *bison* specifications and can result in parsers that are difficult to debug. This is always worth double checking, preferably before things start going wrong.

If an action is omitted, the default action of `{ $$ = $1; }` is normally applied automatically. However this may not be what is always needed, and so always inserting an explicit action is sensible.

The power and ease of use of *bison* and similar tools is attractive and this approach to the coding of bottom-up parsers has been used effectively in many compiler projects.

5.3 Tree Generation

In the last two sections, examples of top-down and bottom-up *recognisers* have been presented. But in a compiler, more is required from the syntax analysis phase. There are some compiler-related applications where the task can be completed by simply adding code generation code to the recognising code. For example, it may be possible for very simple programming languages to produce target code, probably for some form of virtual machine, directly from the recognising process. But for the majority of compiled languages we will need a little more, and the code that has to be added will generate the *parse tree*.

Generating a parse tree turns out (at least in theory) to be simple. But before any code can be added to the recogniser, it is vital to do some tree design first. In Sect. 2.3.3.1, parse trees and abstract syntax trees were illustrated. Clearly, the design of the tree produced by the syntax analyser is controlled by the formal syntax of the language being compiled, but the details, specifically of which of the non-terminals and/or productions should have tree nodes associated with them, have to be resolved. In other words, we have to decide for each reduction performed by the syntax analyser whether a tree node is generated and if so, what data must that node include.

This can be made clearer using an example. Return again to the simple arithmetic expression language from Fig. 2.3.

<expr> ::= <term> | <expr> + <term> | <expr> - <term>
<term> ::= <factor> | <term> * <factor> | <term> / <factor>
<factor> ::= <integer> | (<expr>)
<integer> ::= <digit> | <integer> <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The abstract syntax trees for this language can be made very simple, as in the example of Fig. 2.4b. There are five different types of node in the tree, one for each of the four arithmetic operators and the fifth to hold a constant integer value. This implies that only the following reductions from the grammar cause actual tree nodes to be generated:

<expr> ::= <expr> + <term>
<expr> ::= <expr> - <term>
<term> ::= <term> * <factor>
<term> ::= <term> / <factor>
<factor> ::= <integer>

We have made an obvious and reasonable simplification here by regarding an <integer> as a lexical token, thus not requiring the structure of an integer to be encoded into the syntax tree. Furthermore, for this example, parentheses are not required in the tree either. These are used for syntactic grouping and their effect is automatically included because of the way in which the tree is constructed.

The actual code required to construct the tree fortunately turns out to be simple. We have already seen (Sect. 5.2.2.2) how a bottom-up recogniser for these arithmetic expressions can be coded using *bison*. This code can be augmented as follows to generate a tree rather than evaluating the expression on the fly:

```
expr:
  term { $$ = $1; }
  | expr '+' term { $$ = newnode(N_PLUS, 0, $1, $3); }
  | expr '-' term { $$ = newnode(N_MINUS, 0, $1, $3); }

term:
  factor { $$ = $1; }
  | term '*' factor { $$ = newnode(N_MUL, 0, $1, $3); }
  | term '/' factor { $$ = newnode(N_DIV, 0, $1, $3); }
```

```
factor:
  INTEGER { $$ = newnode(N_INTEGER, ival, NULL, NULL); }
  | '(' expr ')' { $$ = $2; }
```

This code assumes the availability of a function `newnode` which generates a new node for the syntax tree. Each of these nodes contains four fields. The first field contains an integer constant (`N_PLUS`, `N_MINUS`, `N_MUL`, `N_DIV` or `N_INTEGER`) indicating the type of the node. If the node has type `N_INTEGER` (i.e. it contains an integer constant) the second field contains the value of the integer constant and zero otherwise. The third and fourth fields contain links to the left and right subtrees, respectively. For example, the syntax rule `expr '+' term` results in a node being created with type `N_PLUS`, a zero second field, a third field (the left subtree) pointing to the tree generated for the `expr` and a fourth field (the right subtree) pointing to the tree generated for the `term`. Once the reduction of `expr '+' term` has been completed, `newnode` is called and the result returned by `newnode` is passed back to the calling rule.

Similarly, when an `INTEGER` is reduced, a new node of type `N_INTEGER` is created, the value of that integer constant (in this case placed in the C variable `ival`) is placed in the node and the link to the node is returned to the calling rule.

As the parse proceeds, the tree is constructed node by node, the linking of nodes being driven by the order in which the reductions are performed. A full example of this technique can be seen in Sect. 5.4.2.

A very similar approach is taken for the top-down approach. The recogniser can be transformed into the complete parser by adding the code to generate tree nodes. Each time a construct requiring a tree node is recognised, a call is made to a function to allocate space for a new node, the fields are filled in appropriately and the recognising function passes back a link to this node to its caller. For example, suppose a grammar contains the production $P \rightarrow QR$. The recogniser would then have the form

```
void P() {
    Q(); R();
}
```

To add tree generation, it would make sense to modify these recognising functions to return a pointer to the subtrees they generate. The code would then look like this:

```
astptr P() {
    astptr qptr, rptr;
    qptr = Q();
    rptr = R();
    return newnode(N_PNODE, qptr, rptr);
}
```

Here, the type `astptr` represents a pointer to a node in the tree, and we assume that the node for this structure `P` has three fields, the first identifying it as representing a structure `P`, the second pointing to the subtree for `Q` and the third pointing to the subtree for `R`.

Dealing with the recognising code generated from EBNF productions of the form $E \rightarrow T\{+T\}$ is not difficult but has to be done after deciding on the structure required for the resulting tree. Examples of tree generating code for top-down parsers is shown in Sect. 5.4.1.

5.4 Syntax Analysis for DL

The BNF of DL is shown in the appendix. The process of transforming this grammar into a syntax analyser, top-down or bottom-up, should be uncomplicated, especially if the code is developed in a structured and step by step way. However, there are inevitably some trickier aspects, and these will be highlighted in the sections below.

The aim here is to develop two syntax analysers for DL, the first using a hand-written top-down approach and the second using *bison* to generate a bottom-up parser. The syntax analysers should generate identical outputs for feeding into the next phase of compilation. The top-down parser here has been combined with a hand-written lexical analyser and the bottom-up parser is used with a *flex*-generated lexical analyser.

5.4.1 A Top-Down Syntax Analyser for DL

DL has been designed to have a simple but not trivial syntax, without any structures which could cause any significant difficulty to a top-down predictive parser. However, there is sufficient complexity to require some careful planning before coding.

Tackling the implementation of the front-end of a DL compiler raises some important issues.

- Lexical analysis—let us assume here that the lexical analyser is complete and well-tested. We should also know in detail how the tokens from the lexical analyser are structured, encoded and passed on to the syntax analyser. This issue has already been covered in Chap. 3.
- What are we expecting the syntax analyser to produce? Traditionally the next phase of compilation is likely to require a syntax tree and this is what will be produced in these examples. Clearly, debugging output is required too. This should include a trace of the execution of the parsing process together with a human-readable version of the tree (to be checked in the testing phase) and the contents of other front-end data structures such as the symbol table.
  What should the tree look like? It is essential to produce a list of node types in the tree and decide what data each node should contain. This should largely be driven by the non-terminal symbols defined in the grammar of DL.
- How reliable would we like error recovery to be? Always getting it right, whatever that means, may not be feasible. But error detection should always work.

- How should names/symbols be handled? A symbol table is essential. What has to be stored for each symbol? What about data structures? Do we need a hash table or some form of binary tree, or will simple linear search be adequate? Are there issues concerning the handling the scopes of names?
  Fortunately DL's symbol management is not too challenging. Names in the symbol table refer to a function or to an array or to a variable. So there are just three distinct types. There are a few other items of information to store for each symbol and these will be covered later but each symbol entry can be represented by a simple C `struct`. As names are declared they are inserted into the symbol table and whenever a name is used, it is looked up in the symbol table. But there is a slight complication here. DL allows the declaration of variables *within* functions so that in code of the form:

```
int v1;

f();
int v1;
{  v1 = 1;  /* this sets the locally defined v1 */
   .
   .
   .
}

{  v1 = 1;  /* this sets the globally defined v1 */
   .
   .
   .
}
```

  the variable `v1` will be placed into the symbol table twice and the correct one has to be found each time it is used. Fortunately, there is an easy way of managing this. If the symbol table is structured as a *stack*, linear search starting at the top of the stack can be used for symbol lookup. Looking up `v1` within function `f` will find the entry nearer the top of the stack. If another variable `v2` is declared globally, searching for `v2` within the compilation of `f` will also succeed. This may well not be the most efficient way of managing a symbol table, but it is certainly simple. Maybe with such a simple language, we do not expect huge programs to be written so the overhead of linear search is not a problem.
  Symbol tables are covered in more detail in Sect. 5.6.

Tackling the top-down syntax analyser for DL should follow the steps outlined in Sect. 5.1.1.1—check the lexical analyser, write a recogniser, add tree generation code, add error recovery, test. During this process there will be a need to include various other utility functions such as symbol table management, syntax tree management (particularly printing the tree) and so on. We have already discussed the testing of the lexical analyser and in this section we examine some of the key aspects of the subsequent steps.

5.4.1.1 Writing a Recogniser

A recognising function should be written for each of the non-terminals in the BNF grammar. Fortunately there is little in the DL grammar that needs special attention in its translation from BNF rules to recognising code and the approach presented earlier in this chapter can be adopted. We can examine here any special issues for the functions recognising individual non-terminal symbols.

- `<program>`

  Looking further on in the grammar, it can be seen that a `<block>` must start with a `{` token. This enables the correct alternative in the definition of `<program>` to be selected using the single token lookahead. The global variable `token` contains the current lexical token returned by the lexical analyser (the function `lex()`). The function `syntrace` simply outputs its argument if the value of a global tracing flag is set TRUE. This allows simple turning on or off of the tracing of the recogniser's execution. Here, the message is output when the recognising function is entered. It is perfectly possible to output the message instead when the recognition has been successfully completed.

```
<program> ::= <block>
            | <declarations> <block>


void program()
{
  syntrace("<program>");
  if (token == opencurlysym) block();
  else {
    declarations();
    block();
  }
}
```

- `<declarations>`

  It is clear that the BNF rule defines `<declarations>` to be a sequence of one or more `<declaration>`. But what terminates this sequence? Looking at the definition of `<declaration>`, a `<declaration>` must start with an `int` token from `<variabledeclaration>` or an `<identifier>` from a `<functiondeclaration>`. But maybe a slightly clearer approach is to stop when the token following `<declarations>` is found. This is just the token `{` (which starts a `<block>`).

```
<declarations> ::= <declaration>
                 | <declaration> <declarations>
```

```
void declarations()
{
  syntrace("<declarations>");
  declaration();

  while (token != opencurlysym)   /* until a <block> */
    declaration();
}
```

- <vardec>

The definition of <vardec> has two alternatives, both starting with the token <identifier>. A factoring transformation can formalise the implementation of a recogniser here, but it amounts to the same thing as coding the recognition of an <identifier> and then looking for a following [token.
In this code the function error is called, outputting the error message and then, for now, causing the program to stop. We will worry about error recovery later. Note the explicit calls (token=lex();) to the lexical analyser to maintain the single token lookahead throughout.

```
<vardec> : := <identifier> | <identifier> [ <constant> ]


void vardec()
{
  syntrace("<vardec>");
  if (token != identifiersym)
    error("identifier expected in vardec\n");
  else {
    token=lex();
    if (token == opensquaresym) {       /* it's an array */
      token=lex();
      if (token != constantsym)
        error("need array size\n");
      else {
        token=lex();
        if (token != closesquaresym)
          error("need close square bracket in array declaration\n");
        else
          token=lex();
      }
    }
  }
}
```

- <statement>

Handling a <statement> may look challenging but it turns out to be perfectly-manageable. Looking elsewhere in the grammar it is clear that the current lookahead lexical token will disambiguate between the alternatives. The <empty>

statement is recognised if none of the other alternatives can match. In this case control just falls through the `statement()` function.

```
<statement> : := <assignment> | <ifstatement> | <whilestatement>
              | <block> | <printstatement> | <readstatement>
              | <returnstatement> | <empty>
```

```
void statement()
{
  syntrace("<statement>");
  if (token==identifiersym) assignment();
  else if (token==ifsym) ifstatement();
  else if (token==whilesym) whilestatement();
  else if (token==opencurlysym) block();
  else if (token==printsym) printstatement();
  else if (token==readsym) readstatement();
  else if (token==returnsym) returnstatement();
   /* else empty */
}
```

- `<expression>`

Finally we should include an example of a left-recursive production. The `<expression>` rule shows that `<addingop>` is both a binary and unary operator. As usual, we deal with the left recursion by replacing it by iteration.

```
<expression> : := <expression> <addingop> <term>
               | <term> | <addingop> <term>
```

```
void expression()
{
  syntrace("<expression>");
  if ((token==plussym) || (token==minussym))
    token=lex();

  term();

  while ((token==plussym) || (token==minussym)) {
    token=lex();
    term();
  }
}
```

Not all the non-terminal symbols in DL's BNF grammar need to have corresponding recognising functions because sometimes the recognition can be done more easily and transparently where they are used in other recognising functions. For example, there is no real need for explicit recognising functions to be written for `<relop>`, `<addingop>` or `<multop>`.

The tracing of the execution of the parsing of the very simple DL program shown in Fig. 5.1 illustrates the order in which the recognising functions are called.

Fig. 5.1 A very simple DL program

```
int a;
{
  a = 2 + 3*a
}
```

The output from the recogniser when parsing this simple program should be something like this:

```
<program>
<declarations>
<declaration>
<variabledeclaration>
<vardeflist>
<vardec>
<block>
<statementlist>
<statement>
<assignment>
<expression>
<term>
<factor>
<term>
<factor>
<factor>
****PARSE ENDED****
```

Checking this output by hand is plainly tiresome for anything but the most trivial of programs. It may be better to leave more extensive testing to the next stage, once the tree has been generated.

5.4.1.2 Tree Generation

We have already seen in Sect. 5.3 that the principles of adding tree generation to the recognising code are easy, but there are inevitable complications. Each recognising function has to be modified so that it returns a pointer to a tree node corresponding to the syntactic structure it has just recognised. The linking of all these nodes takes place as the parsing proceeds, thus ending up with a complete parse tree.

The first step here is to design the tree, specifying all the different node types and what each node should contain. Typically, there should be one node type for each of the non-terminals in the grammar, but in practice there is no need for some of these non-terminals to appear in the tree since they are just not needed for the generation of target code. Also it may be possible to simplify the tree and hence its further processing by making minor changes to the way in which the grammar is represented. For example, the node for the BNF rule <expression> ::= <expression><addingop><term> could be represented as an <expression> node with three subtrees, storing the

<expression>, <addingop> and <term>, respectively. Instead, it is probably better to associate the <addingop> with the <expression> node so we can use two different nodes labelled with plus and minus both having <expression> and <term> subtrees. However, remember that there is not just a single correct tree design.

Turning now to the specific task of the design of a tree for DL, we can propose a set of node types based on a subset of the non-terminal symbols. In this design each tree node is made up of a fixed number of fields. This simplifies the implementation a little and is fine for a compiler for DL. Each node contains an integer value indicating the type of the node, an additional two integer values and three pointers to be used to point to subnodes. The integer field astdata2 is used where needed to indicate whether a variable has been declared locally or globally. This will be described in detail later. A list of these node types is shown in Table 5.1.

The tree nodes are declared as follows:

```
typedef struct tnode *astptr;

typedef struct tnode {
  int asttype;
  int astdata1,astdata2;
  astptr p1,p2,p3;
} astnode;
```

To illustrate this tree structure we can return to the simple program of Fig. 5.1. The generated tree is shown in Fig. 5.2. The parser keeps a pointer to the main block and a separate pointer to the chain of function definitions. Here, the chain is empty because there are no functions in the program.

Enhancing the recognising code to add tree generation is illustrated by the code for handling <expression>. The function expression() returns a pointer of type astptr to its caller and this returned result points to the complete tree representing the expression.

```
astptr expression()
{
  astptr pfirst;     /* first pointer in the expr chain */
  astptr term2;
  int startingtoken,nodetype;

  syntrace("expression");
  startingtoken=plussym;
  if ((token==plussym) || (token==minussym)) {
    startingtoken=token;
    token=lex();
  }
  pfirst=term();
  if (startingtoken==minussym)
    pfirst=newnode(N_UMINUS, 0, pfirst, NULL, NULL);
  while ((token==plussym) || (token==minussym)) {
    if (token==plussym) nodetype=N_PLUS; else nodetype=N_MINUS;
    token=lex();
```

Table 5.1 Parse tree nodes

Node type	Data	Pointer 1	Pointer 2	Pointer 3
`N_SLIST` (Statement list)	–	Next statement	–	–
`N_ASSIGN` (Assign to variable)	Variable location	Expression	–	–
`N_PRINT` (Print expression)	–	Expression	–	–
`N_UMINUS` (Unary minus)	–	Term	–	–
`N_PLUS`, `N_MINUS`, `N_MUL`, `N_DIV` (Arithmetic operators)	–	Term or factor	Term or factor	–
`N_EQ`, `N_NE`, `N_LT`, `N_LE`, `N_GT`, `N_GE` (Relational operators)	–	Expression	Expression	–
`N_CONST` (Integer constant)	Constant value	–	–	–
`N_ID` (Variable identifier)	Variable location	–	–	–
`N_ARRAYREF` (Array reference)	Array location	Expression	–	–
`N_ARRAYWRITE` (Assign to array element)	Array location	Index expression	Rhs expression	–
`N_FUNCTION` (Function definition)	Function location	Argument list	Block	–
`N_FNCALL` (Function call)	Function location	argument list	–	–
`N_ARG` (Argument list)	–	Argument expression	Next in argument list	–
`N_RETURN` (Return from function)	–	Expression	–	–
`N_READ` (Read to variable)	Variable location	–	–	–
`N_WHILE` (While statement)	–	Bexpression	Block	–
`N_IF` (If statement)	–	Bexpression	Then block	Else block
`N_FUNCTIONDEC` (Function definition chain)	–	Function	Next in chain	–

```
      term2=term();
      pfirst=newnode(nodetype, 0, pfirst, term2, NULL);
    }
    return pfirst;
}
```

This code needs a little explanation. The BNF rule for `<expression>` has been rewritten as the EBNF equivalent

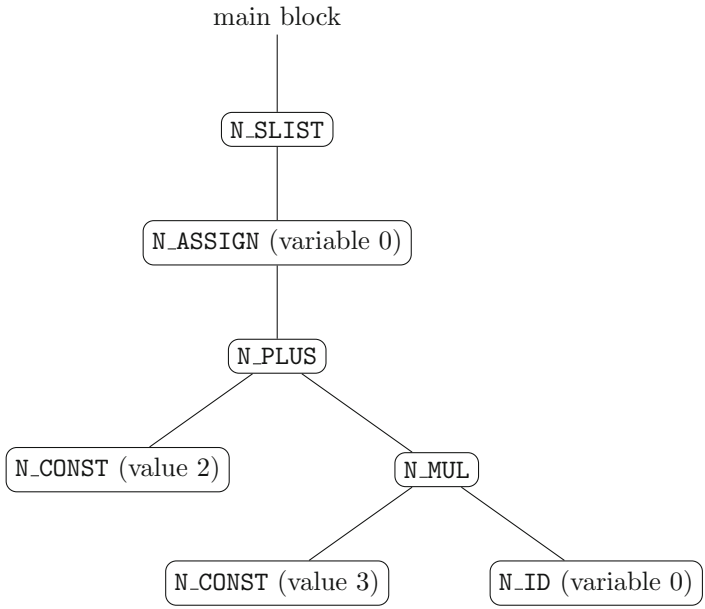


Fig. 5.2 Tree from the program of Fig. 5.1

```
expression = ["+" | "-"] term {("+" | "-") term}.
```

The code stores any leading sign (plus in default) and then calls `term()`. The tree generated by `term()` is saved in `pfirst` and if that term was preceded by a unary minus, a unary minus node is inserted. Then, each time an adding operator followed by a term is found, a new node is generated containing the adding operator found, with the expression so far as the left-hand side and the new term as the right hand side. Note that this ensures the *left-associativity* of the adding operators. The `newnode` function obtains memory space for a new tree node, large enough to contain the node's type, an integer value and three pointers to other nodes. As seen in this code, the `N_UMINUS` node requires one pointer and the `N_PLUS` and `N_MINUS` nodes require two pointers.

A similar example handles the `while` statement.

```
astptr whilestatement()
{
  astptr pbexp=NULL;
  astptr pwhile=NULL;

  syntrace("whilestatement");
  if (token != whilesym) error("while expected\n");
  else {
    token=lex();
    if (token != openbracketsym)
```

```
        error("open bracket after while expected\n");
      else {
        token=lex();
        pbexp=bexpression();
        if (token != closebracketsym)
          error("close bracket in while expected\n");
        else {
          token=lex();
          pwhile=newnode(N_WHILE, 0, pbexp, block(), NULL);
        }
      }
    }
    return pwhile;
  }
```

The `N_WHILE` node combines two subtrees, one from the `<bexpression>` and the other from the target `<block>` of the `while`.

An important issue concerns the parse tree and variable declarations. *In DL there is no need to put variable declarations* (`<variabledeclaration>`) *in the parse tree*. When a variable is declared, the information about that variable is stored in the symbol table. This information includes the text representation of the symbol, its type (integer, array or function), its size if an array and for integer variables and arrays the runtime location of the storage used for that variable. A reference to a variable in the parse tree just needs its runtime storage location. After the parsing process there is no need to know the text name of the variable. We will be looking at runtime storage issues later in Sect. 5.6

Variable name management in the syntax analyser for DL therefore requires that:

- on declaration, insert the variable name into the symbol table, including type and size if necessary.
- on encountering the use of a variable, look up that variable in the symbol table. If it is not there, then an error is reported (variable not declared) and if it is there, then the runtime storage location stored in the symbol table entry is copied to the parse tree. A pointer to the symbol table entry could be stored in the tree instead.

5.4.1.3 Error Detection and Recovery

The error detection code is already present in these examples. It is easy and natural to incorporate it with the recognition code as it is being written. However, error recovery has not been specified, but an `error` function has already been presented above where the action is to output the error message and then exit from the program. We need to do better and this is discussed in Sect. 5.5.

5.4.1.4 Utility Functions

To support the lexical analysis, the syntax analysis and the tree generation, we need a set of utility functions to perform various general tasks such as symbol table management, space allocation and initialisation of tree nodes, tree printing and other debugging operations such as tracing the execution of both the lexical and syntax analysers.

Printing the tree can be a little awkward because something is required during the debugging stages to enable the tree to be verified by eye. A simple approach is to write code to perform a conventional *pre-order* traversal of the tree where the node identity is output, followed, recursively, by the children of that node. For example, the output for the program of Fig. 5.1 presented in this way appears as:

```
N_SLIST - Statement list, this statement:
|  N_ASSIGN - Assign to variable at offset 0, expression is:
|  |  N_PLUS - plus, lhs is:
|  |  |  N_CONST - const, value is: 2
|  |  N_PLUS - plus, rhs is:
|  |  |  N_MUL - mul, lhs is:
|  |  |  |  N_CONST - const, value is: 3
|  |  |  N_MUL - mul, rhs is:
|  |  |  |  N_ID - identifier, runtime offset 0
N_SLIST - Statement list, continued...
```

This is obviously not as clear as the graphical tree of Fig. 5.2 but the traversal code is easy to write and with some practice, this format is not too difficult to comprehend. An alternative approach is to get the compiler to generate control commands for some text processing/graphics program which has facilities for the generation of neatly presented trees.

The code in function `printtree` to generate this human-readable tree is based on a large `switch` statement with a `case` for each of the node types. An example of this code shows how to deal with the node for the `if` statement:

```
case N_IF:
    printf("N_IF - if, condition:\n");
    printtree(xx->p1, depth+1);
    tabout(depth);
    printf("N_IF - if, thenpart:\n");
    printtree(xx->p2, depth+1);
    tabout(depth);
    printf("N_IF - if, elsepart:\n");
    printtree(xx->p3, depth+1);
    break;
```

The second argument to `printtree` is a tree depth, used to control the printing of the leading layout characters on each output line.

Managing the symbol table implemented in the form of a stack poses no real problems. Functions for symbol insertion and symbol lookup need to be coded, and

a function to output the entire contents of the symbol table can be a useful debugging aid.

5.4.1.5 Testing

The front-end code has to be systematically tested in stages. Clearly, validating the output for a large range of test programs, including difficult examples, is essential. It is possible to use a software tool to generate random syntactically correct DL programs from the BNF grammar and these can all be put through the front-end. The trees thus generated could also be verified, perhaps automatically.

Including *tracing* code can also help verify that all is well. And keeping this tracing code available throughout the compiler's development will help debug problems found later in the project.

Testing is not easy and perhaps the most important point is that it is essential when scheduling a compiler project to allow sufficient time for this phase of development. Just checking that a "Hello world!" program works is definitely not adequate.

5.4.2 A Bottom-Up Syntax Analyser for DL

Most of the issues discussed in the last section on top-down syntax analysis apply just as well to a bottom-up approach. Our aim here is to make use of the lexical analyser already discussed in Chap. 3, generated by *flex* from regular expressions defining the syntax of DL's lexical tokens, and then to use *bison* to generate the complete syntax analyser. We hope to be able to generate an identical tree to that produced by the top-down parser.

The task of producing a bottom-up parser follows the same steps as for the top-down parser. Firstly, ensure that the lexical analyser is working, then code a recogniser, add tree generation, add error recovery and test throughout.

The lexical analyser for DL written using *flex* has already been developed in Sect. 3.4.1.2. The code generated by *flex* will be combined with the code from the *bison*-generated syntax analyser to produce the complete front-end. Many of the design decisions have already been taken because there is much commonality between the top-down and bottom-up parsers. For example, all the symbol table functions, the tree manipulation code (creating and adding nodes, printing the tree) and the execution tracing can be used without change. However, moving from a traditional top-down LL(1) parser to a much more powerful LALR(1) bottom-up parser has some consequences, particularly in the way in which the tree is constructed. But before any tree nodes are generated it makes good sense to develop a recogniser and this is where the power of a syntax analyser generating tool is particularly apparent.

5.4.2.1 Writing a Recogniser

Recall that the input to *bison* consists of three sections—definitions, rules and user code, separated by `%%` lines. For the recogniser, the definitions section of the *bison* input has to declare the named tokens returned by the lexical analyser. The complete definitions section therefore looks like this:

```
%{
#include "dlrecog_defs.h"
%}

%token CONSTANT IDENTIFIER LESYM EQSYM GESYM NESYM ELSESYM IFSYM
%token INTSYM PRINTSYM READSYM RETURNSYM WHILESYM

%%
```

The header file `dlrecog_defs.h` defines any function prototypes required and can contain `#include` directives for other (system) header files. The `%token` declarations correspond to the tokens used in the lexical analyser (see Sect. 3.4.1.2). In these examples, we are using a naming convention where terminal tokens are in upper case and non-terminals are in lower case.

The *bison* rules for the recogniser are derived easily from the BNF rules defining DL. The correspondence between the two sets of rules should be very close. For example, the BNF rules for `<program>`, `<functiondeclaration>`, `<ifstatement>` and `<expression>` in DL are:

```
<program> : := <block> | <declarations> <block>

<functiondeclaration> : := <identifier> (); <functionbody>
                         | <identifier> ( <arglist> ); <functionbody>

<ifstatement> : := if ( <bexpression> ) <block> else <block>
                 | if ( <bexpression> ) <block>

<expression> : := <expression> <addingop> <term>
                | <term> | <addingop> <term>
```

These BNF rules are translated into *bison* rules as follows. The `syntrace` function is there to trace the execution of the syntax analysis and can be as simple as just outputting its string argument.

```
program:
  block
    { syntrace("program - without declarations"); }
  | declarations block
    { syntrace("program - with declarations"); }

functiondeclaration:
  IDENTIFIER '(' ')' ';' functionbody
                  { syntrace("functiondeclaration - no args"); }
```

```
  | IDENTIFIER '(' arglist ')' ';' functionbody
                { syntrace("functiondeclaration - with args"); }

ifstatement:
  IFSYM '(' bexpression ')' block ELSESYM block
        { syntrace("ifelsestatement"); }
  | IFSYM '(' bexpression ')' block
        { syntrace("ifstatement"); }

expression:
  expression addingop term { syntrace("expression"); }
  | term { syntrace("expression - just term"); }
  | addingop term { syntrace("expression - unary op"); }
```

Once the full set of *bison* rules has been produced, the section is terminated with a `%%` line. The *bison* input file is then completed by adding the definitions of three functions:

```
void syntrace(char *s)
{
  if (TRACE) fprintf(stdout, "%s\n", s);
}

void yyerror(char *s)
{
  printf("%s on line %d\n",s,yylineno);
}

int main(int argc, char *argv[])
{
  if (yyparse() == 0) syntrace("Recogniser succeeds");
  return 0;
}
```

The function `yyerror` is called if a syntax error is found. The generated parser/recogniser is called by `yyparse()` in the main program.

It is important to understand how the lexical and syntax analysers are combined. In particular, the names declared in the `%token` declarations in the syntax analyser specification must be accessible in the lexical analyser. Taking the specific example of this recogniser for DL, we start off with three files—the lexical analyser specification file (`dlrecog_lex.l`), the syntax analyser specification file (`dlrecog_syn.y`) and the header file (`dlrecog_defs.h`). When the file `dlrecog_syn.y` is processed by *bison*, a new header file (`dlrecog_syn.h`) is generated, containing information about the `%token` declarations and so on. This header file has to be included in the code generated by *flex*. So the `#include` directives that have to be explicitly coded are:

- `dlrecog_lex.l` has to contain `#include dlrecog_defs.h` and also it must contain the line `#include dlrecog_syn.h`
- `dlrecog_syn.y` has to contain `#include dlrecog_defs.h`

This task of generating a recogniser may turn out to be very simple, but it is important to do it before tackling further steps such as tree generation. It forms a good check on the correctness of the grammar and if shift/reduce or reduce/reduce errors are reported these problems should definitely be investigated.

5.4.2.2 Tree Generation

Adding tree generation code to the recogniser can be a little tricky for some constructs. The rule is, after the tracing code, to include a call to a function to generate and initialise a new node for the tree. We have already seen how this can be done in Sect. 5.3.

Our aim in this example is to produce the same tree as generated by the DL top-down parser, already described. This can be done by including the functions already developed for the top-down parser for DL for symbol table and abstract syntax tree management and building on the rules in the *bison* recogniser. The additional code in the rules has to generate and link tree nodes and it also has to access and manipulate data in the symbol table.

One of the more significant changes in moving from a recogniser to the implementation of full tree generation is that the *bison* rules now have to return data to their calling rules. Normally, the data type returned will be a pointer to a tree node, but it is usual that other data types will be needed too. For example, a rule to recognise an identifier may return the identifier as a string. *Bison* allows the user to specify a union type to cover all the possible data types to be returned via the terminal and non-terminal symbols. In our implementation for the parser for DL, the specification is

```
%union {
   int sval;
   astptr psval;
   char *strval;
}
```

and this allows integers, tree pointers and strings to be returned via rules. But to do this, the actual single type returned by each of the terminal and non-terminal symbols has to be specified in the *bison* input. Therefore, in the definitions section, terminal symbols have to be declared (with their type) using `%token` and non-terminals are given a type using the `%type` declaration.

```
%token <sval> CONSTANT
%token <sval> IDENTIFIER
%token <sval> ELSESYM
%token <sval> IFSYM
%token <sval> LESYM
.
.
%type <psval> program
```

```
%type <psval> declarations
%type <psval> declaration
.
.
%type <sval> relop
%type <strval> identifier
%type <sval> constant
```

The design of the tree has already been presented in Table 5.1. For many of the non-terminal symbols, the construction of the corresponding tree nodes is easy. For example, the rule for the conditional statement is:

```
ifstatement:
  IFSYM '(' bexpression ')' block ELSESYM block
        { syntrace("ifelsestatement");
          $$ = newnode(N_IF, 0, $3, $5, $7); }
  | IFSYM '(' bexpression ')' block
        { syntrace("ifstatement");
          $$ = newnode(N_IF, 0, $3, $5, NULL); }
```

Most of the statement types of DL are converted to their tree nodes in this general way. Another example is the handling of expressions:

```
expression:
  expression addingop term { syntrace("expression");
                             $$ = newnode($2, 0, $1, $3, NULL); }
  | term { syntrace("expression"); $$ = $1; }
  | addingop term
        { syntrace("expression");
          if ($1 == N_MINUS)
            $$ = newnode(N_UMINUS, 0, $2, NULL, NULL);
          else $$ = $2; }
```

In this example, the first part of the rule, corresponding to the BNF rule
<expression> ::= <expression><addingop><term>
uses the value returned through `addingop` as the node's type (either `N_PLUS` or `N_MINUS`), the `expression` as the left subtree and the `term` as the right subtree. There is no need to worry about associativity because this is handled automatically by the parser and the way in which the grammar is phrased. For the BNF rule <expression> ::= <term>, there is no need to create a new node. And for the BNF rule <expression> ::= <addingop><term> a unary minus node is created (or no new node if the `addingop` is `N_PLUS`).

Lists often appear in the designs of programming languages. For example, the `statementlist` rule is

```
statementlist:
  statement { syntrace("statementlist");
              $$ = newnode(N_SLIST, 0, $1, NULL, NULL); }
  | statement ';' statementlist {
              syntrace("statementlist");
```

```
$$ = newnode(N_SLIST, 0, $1, $3, NULL); }
```

Care has to be taken with such lists to ensure that the items in the list are placed in the tree in the right order. Drawing pictures helps to get things correct.

There are some constructs where more complex actions are required. For example, consider the handling of the <functiondeclaration>, defined as:

```
<functiondeclaration> : := <identifier> (); <functionbody>
                        | <identifier> ( <arglist> ); <functionbody>
```

To deal with this, the syntax analyser has to parse all the individual components, place the <identifier> in the symbol table (as a function name) and then it has to ensure that the state of the symbol table is saved before any new declarations from the function are inserted, so that at the end of the parsing of the function, the local variables can be removed by reinstating the symbol table. It is immediately clear that this action of adding the function name to the symbol table cannot be done at the end of the parsing of the <functiondeclaration> because at that stage, the <functionbody> has been parsed. Fortunately *bison* has a useful feature that resolves this problem, shown in the code and description below.

```
functiondeclaration:
  identifier  '(' { /* Code placing identifier into symbol table.
                     Save the state of the symbol table.
                     Save and reset a pointer to the storage of
                     local variables in the runtime stack. */ }
   arglist ')' ';' functionbody { syntrace("functiondeclaration");
                      $$ = newnode(N_FUNCTION, functioncount++,
                                   $4, $7, NULL);
                      /* Save storage requirements of this
                         function in the symbol table.
                         Reset local variable pointer. */ }
.
.
.
```

The code following `identifier '('` is executed at that stage in the parsing process, specifically before the `arglist` has been parsed. Since in this version of the parser, the symbol table is organised in the form of a stack, just a single stack pointer has to be saved at the beginning of the compilation of the function and restored at the end. Further discussion of the issues concerning runtime storage of DL variables appear in Sect. 5.6.

Dealing with declarations does not require new nodes to be placed into the tree. Here, we need to be manipulating the symbol table instead:

```
vardeflist:
  vardec { syntrace("vardeflist"); $$=NULL; }
  | vardec ',' vardeflist { syntrace("vardeflist"); $$=NULL; }

vardec:
```

```
  identifier { syntrace("vardec"); insertid($1,S_INT,1); $$=NULL;}
  | identifier '[' constant ']' { syntrace("vardec");
                                  insertid($1,S_ARRAY,$3);
                                  $$=NULL; }
```

Why is `identifier` rather than `IDENTIFIER` used in this example above? According to the naming convention, `IDENTIFIER` is a terminal symbol (and indeed is returned by the lexical analyser) and `identifier` should be a non-terminal. The reason for doing this is shown in the definition of `factor`:

```
factor:
.
.
.
  | identifier '[' expression ']'
        { $$ = newnode(N_ARRAYREF, lookupid($1, S_ARRAY),
                       $3, NULL, NULL); }
.
.
.
```

In this simple rule and action, once the entire factor is recognised, the node for the tree is created and returned. If `IDENTIFIER` had been used instead, the code of the rule's action would have no access to the text string corresponding to `IDENTIFIER` assigned to a string variable in the lexical analyser, especially if the `expression` contained other identifiers. This problem is resolved by defining `identifier` as a non-terminal which returns a *copy* of its text representation as its result. The same applies to the terminal token `CONSTANT`:

```
identifier:
  IDENTIFIER { $$ = strdup(lexident); }

constant:
  CONSTANT { $$ = lexnumval; }
```

Finally, the main program has to initialise global variables such as the top of the symbol table, the first virtual location for the storage of variables, and so on, and then call the *bison*-generated parser `yyparse()`.

5.4.2.3 Error Detection and Recovery

If the *bison*-generated parser encounters a syntax error, the function `yyerror` is called. This function is normally provided by the *bison* user and allows an error message and other information to be output. The automatically generated error message is normally "syntax error", a message not sufficiently helpful for most applications. `yyerror` returns to the parser and it tries to recover from the error if error recovery has been specified, and the parsing continues. Otherwise the parsing terminates.

Ignoring error recovery is adequate for some applications and even for initial testing of a parser. However, the addition of some very simple error recovery may make the parser much more useful.

An easy and possibly acceptable action on detecting an error is to skip input tokens until a semicolon or close curly bracket is found and then regard that as the end of a statement. Then parsing is allowed to continue. This can be done by augmenting the set of rules for `statement` to include the rules

```
| error ';' { $$ = NULL; }
| error '}' { $$ = NULL; }
```

The use of the `error` token attempts to re-synchronise the input with the DL rules so that parsing may continue. Plainly this will not always work perfectly, but it usually does a good job.

5.4.2.4 Utility Functions

The utility functions used in the bottom-up parser are largely identical to those already described for the top-down parser. Specifically, tree generation and manipulation need no modification at all. The use of a *bison* parser needs a few small functions to be added and these have already been described.

5.4.2.5 Testing

Approaches to testing are the same as with the top-down parser. Checking that the language's grammar has been correctly transliterated into *bison* rules is a good first step, ensuring that the pseudo-variables `$1`, `$2`, etc. are being used correctly. *Bison* provides a set of tracing and verbose error reporting facilities. These may help, if desperate.

5.4.3 Top-Down or Bottom-Up?

Choosing between a hand-written top-down predictive parser and a machine-written bottom-up parser is not always easy. Issues to be considered include the following:

- The bottom-up approach requires the availability of an appropriate software tool generating code in a form and in a programming language that can be integrated with the rest of the project.
- Using parser generator tools is sometimes demanding because there may be aspects of the language's grammar requiring implementation skill and maybe even syntax modification to fit in with the constraints of the generator tool. Fortunately this rarely seems to be an issue.

- The use of a parser generator tool *can* save a great deal of time in generating parsing code directly from a version of the language's grammar. Implementation issues for difficult aspects of the grammar are essentially hidden. It is therefore likely that the parser will be somewhat more reliable than the hand-written top-down parser.
- The hand-written top-down approach makes it easier to introduce code hacks to deal with particularly awkward aspects of the translation process, for example dealing with some aspects of context sensitivity. Dealing with such issues in the more formal framework imposed by a parser tool may be much harder.
- The bottom-up approach permits the use of more powerful grammars (for example, LALR(1) versus LL(1)).
- Efficiency, in terms of execution time and runtime storage requirements, is unlikely to be an issue in either approach.
- All parsing methods may require particular modifications to be applied to the language's grammar.
- The hand-written top-down predictive parser should be easy to write, easy to understand and hence easy to modify and maintain. The internals of the bottom-up parser are much more obscure. But there should be no real need to ever examine the code generated by the bottom-up parser generator tool.

In the case of coding a parser for DL, there seems to be little difference in implementation efforts and time for the two approaches. The language is simple and straightforward and causes neither implementation route much difficulty. The numbers of lines of source code are comparable. For larger languages the advantages offered by the use of a parser generator may become more apparent. Furthermore, the choice of an implementation method is made more complex because of the availability of a large number of parser-generating tools, both bottom-up and top-down, with a wide range of different characteristics.

5.5 Error Handling

Errors are detected in all stages of compilation. We have already seen how errors can be detected and reported by the lexical analyser. Errors detected in the syntax analysis phase are examined in this section. And in the next chapter, semantic errors such as type errors will be discussed. There are also, of course, errors detected when the program generated by the compiler is run. The program might fail with a runtime error such as dividing by zero or it could just give wrong answers. These errors suggest problems with the program being compiled rather than with the compiler.

Detecting an error in syntax analysis triggers the production of a message indicating the nature and approximate location of the error. All parsers should be able to produce an informative error message, but providing a *precise* location of the error (rather than where it was detected) is not always possible. For example, consider the C code:

```
a = 0
b = 1;
.
.
```

This will probably cause a compilation error of a missing semicolon to appear when the line `b = 1;` is parsed, despite the fact that the error was actually on the preceding line. In practice, this is not a serious problem for the seasoned compiler user.

But error *recovery* is the key problem. The traditional batch view of compilation—compile, get a list of all the errors, edit the source program accordingly, re-compile—does not apply in all circumstances. In an interactive program development environment it may be much better to flag syntax errors as the source program is typed in. There is no real need for full error recovery here. After the user corrects the error, compilation continues (or even restarts). But in most compilers, error recovery *is* required.

There are several ways in which error recovery can be tackled, but none of them are perfect. The aim is to somehow eliminate or even correct the construct causing the error and then continue with the parsing.

- It may be possible to augment the grammar of the language being compiled using production rules which explicitly recognise specific syntactically incorrect constructs. For example, it *may* be possible to include an extra rule to recognise an `if...then...else` construct with a missing `then`. An appropriate message is output and a tailored error recovery can be achieved. If the grammar of the language contains syntactic redundancy then this approach may be partially successful, but the additional rules have to be designed carefully to avoid introducing ambiguity into the grammar.
- Inserting additional tokens or making minor local modifications to the input once an error has been detected may help, for example by inserting an additional close parenthesis at the point where the parser was expecting a close parenthesis and some other token was found. However this may cause further problems later in the parsing process. This approach is difficult to get right.
- A much more structured approach involves regular checks to make sure that the parser stays *synchronised* with the input at all times. For example, consider the predictive top-down parser with a recognising function for each syntactic construct. Code is introduced at the beginning of each recognising function to ensure that the current lexical token is a member of the *FIRST* set. This is the set of tokens than can validly start the construct being recognised. Similarly, at the end of each recognising function a check is made to ensure that the current lexical token is a member of the *FOLLOW* set. This is the set of tokens than can validly follow the construct being recognised. Determining the *FIRST* and *FOLLOW* sets for the non-terminal symbols in a grammar is not difficult. For example, in DL, a simple scan through the grammar shows that an `<expression>` must start with +, −, (, a constant or an identifier. This is the *FIRST* set for `<expression>`. The recognising function is also passed a set of tokens (*TERMINATE*) that the caller of the function feels should act as additional terminators for the recognition

process. For example, in DL, an `if` token could be considered to terminate an expression. The caller of the function is responsible for recovering if a symbol in *TERMINATE* is found. If the check at the start of the recognising function fails, an error message is output and recovery is achieved by skipping until a symbol in *FIRST* $\cup$ *TERMINATE* is found. Similarly, if the check at the end of the recognising function fails, an error message is output and the function recovers by skipping until a symbol in *FOLLOW* $\cup$ *TERMINATE* is found. This technique is called *panic-mode error recovery* and it can be very effective.

- Parser generator tools such as *bison* have features to help handle errors. *Bison* supports the `error` token and this allows control to be returned to the parser so that error recovery can be achieved. When an error has occurred, the `error` token will match, allowing rules of the form `statement: error';'` to be written. This causes tokens to be skipped (matched by the `error` token) until a semicolon has been read.

  This is a powerful feature, but not easy to get right. Having a large number of rules with the `error` token will result in a fragile parser. Instead a small number of error rules at a high level in the grammar is much more likely to succeed. The *bison* documentation contains much useful practical advice.

Experimentation is plainly involved here, adding recovery actions until something acceptable has been produced. We all have horror stories of inappropriate error messages or poor recovery actions in our favourite compiler. Presenting error information in a form that is useful to the programmer is the aim—a universal and rather unhelpful message saying "syntax error" is produced surprisingly often.

Finally, it is important to avoid the parser making attempts to *correct* the error. In the history of programming language implementation, there have been many cases of the compiler making a correction to the source program as the error recovery action, issuing a warning message and the programmer ignoring the message. It may be that the correction was not what the programmer had originally intended.

5.6 Declarations and Symbol Tables

Before leaving the topic of syntax analysis we must look at the issues of the handling of declarations and the management of names and other symbols. DL has a very simple view of names. In DL a declaration refers to a simple (integer) variable, a single-dimensional integer array (with a size known at compile time) or a function (returning a single integer value).

Designing the data structure for a compiler's symbol table can be a difficult task. The idea that the symbol table is a simple two-dimensional table containing the text representation of the symbol and a small amount of other information is, unfortunately, far from the truth. There are many issues to be considered, including:

- Symbol table lookup normally uses the symbol name as a key. What is an appropriate underlying data structure to support the lookup? Hash tables may be a good

plan, but simple binary trees are unlikely to work well because of the alphabetically clumped nature of names used in real programs.

- Is there a maximum length for names? What precisely is the character set allowed for names? Are upper and lower case letters equivalent?
- How should type information, if any, be stored? For languages supporting a small and fixed number of types such as DL, then a single field can be set aside in the symbol table entry to hold an indication of the type. For languages where, for example, types can be user-defined such as C, a more powerful approach is required, such as using a parse tree of the type specifier. The data structures used for storing type information must allow decisions of type compatibility and so on to be made easily later in the compilation process.
- What are the scoping rules? In languages where variable names can be multiply defined, for example where declarations are local to the block and blocks can be nested, the symbol table has to support the identification of the correct instance of the name. A declaration in an inner block can hide a variable with the same name defined in an outer block. As we have seen, in DL, a stack-based symbol table may help to resolve this requirement.
- Can a name be used for multiple purposes? For example, in DL, is it possible to have declarations in the same scope for symbol `x` of the form `int x`, `int x[3]` and `x` being defined as a function? In DL this results in no ambiguity because when a name is used it is clear that the name refers to a variable, an array or a function. Other languages may be more strict (and sensible!).
- The symbol table entry has to include, directly or indirectly, information on where the storage associated with the symbol is going to be located at runtime. Furthermore the storage size of the object associated with the name will be needed, although this may be implicitly or explicitly available via the type information. Storing some sort of target machine address is plainly inappropriate because at this stage in compilation the code should really be target machine-independent. Storing some form of virtual address is the right approach and this can then be translated at a later stage of compilation into a machine-oriented way of accessing the right symbol. This will be examined later in the context of code generation.

There are a few further DL-specific issues concerned with managing the scoping of names. DL allows the code in a function to have access to the local variables as well as the global variables not having duplicate names that have been declared so far. This can be handled easily using the stack-based symbol table so that on entry to the compilation of a new function, the current index of the top of the symbol table is saved and the new symbols are placed in subsequent entries. When a symbol is referenced in the compilation of the function, the stack is searched down from the top and the correct entry for that symbol is found, be it local or global. On exit from the compilation of that function, the top of the stack pointer is restored, effectively removing the local variables from the symbol table.

The DL symbol table must also include a flag for each symbol to indicate whether it is a local or a global variable. This is needed because we have to distinguish

between the local and global variables in the intermediate code generated by the semantic analysis phase. This is covered in Chap. 6.

For symbols referring to DL functions, we have to store some value to allow the identification of the correct function at runtime. Also a value indicating the total number of local storage locations required by the function has to be stored because this will be important later in code generation. Fortunately, in DL, this value is known at compile time. Dynamic arrays are not allowed. Also the number of arguments in the function's definition should be stored. This will allow a check to be made that the right number of arguments are being supplied on each call.

5.7 What Can Go Wrong?

Hopefully, when implementing the lexical and syntax analysers in a compiler for a real programming language, no severe problems should be encountered. Today's programming languages are well suited to analysis using standard parsers and in most cases a formal grammar is available which can be translated into a specification for a parser-generating tool. It may be that coding a top-down predictive parser by hand is difficult, requiring significant grammar rewriting, so that using a LALR(1) bottom-up parser may be a better plan. However, it is likely that some aspects of the implementation of the analysis phase will go wrong. It is important to be prepared for this.

There may be problems directly attributable to the language's grammar. Has the grammar been designed with a more powerful parsing method in mind? Ambiguities and other errors in the grammar must be investigated. Relying on the parser generator to get it right automatically in all such circumstances is a very bad idea.

It is not unusual for things to start going wrong with the management of names. For example, the grammar of C presented in [8] defines

typedef-name: identifier

and also uses *identifier* in the definition of a conventional variable declaration. In parsing C, the lexical analyser will return an *identifier* but the syntax analyser cannot tell whether this identifier is a type name or a variable name. The lexical analyser really needs context information to determine whether the identifier has already been defined in a `typedef` declaration. This is messy, but something has to be done in all C compilers to resolve this issue. Some parsing techniques can handle this kind of ambiguity in an elegant manner, the correct parse being chosen later when the types of names are all available.

An example of this problem is shown in the complete C program:

```
#include <stdio.h>

typedef char x;

void foo() {
  int a=sizeof(x),x,b=sizeof(x);
```

```
    x=10;
    printf("a=%d, b=%d, x=%d\n",a,b,x);
}

int main() { foo(); return 0; }
```

This code illustrates the context sensitivity problem, and also shows the importance of processing lists (in this case, a list of declarations) in the right order. Here, in the function `foo()`, `a` has to be declared, then `x` and finally `b`. Think carefully about how lists are defined in grammars and how the corresponding actions are executed.

Many programming languages have awkward context-sensitive aspects, often requiring hacky solutions where the syntax analyser sets a global flag that can be read by the lexical analyser. In some cases, grammar modifications can improve the situation. For example C grammars modified to be suitable for *yacc* or *bison* are widely available on the internet. Using parsing techniques specifically for context-sensitive grammars is *not* the way to go. Expressing the context sensitivity in a grammar is difficult and parsing efficiency will suffer too.

5.8 Conclusions and Further Reading

A good way of developing parser-writing skills is to look at the formal grammars of a variety of languages and consider how they can be implemented using both top-down and bottom-up approaches. The grammar for ANSI C contained in [8] is a good starting point, and searching for modifications to this grammar to facilitate its implementation using *bison* and other tools highlights some of the problems in the original grammar.

Many C compiler projects have been written up in detail such as [9, 10]. These books give invaluable practical advice for the construction of high-quality compilers. Software tools such as *lex*, *flex*, *yacc* and *bison* are well-documented with examples on the internet as well as in textbooks such as [6]. Many other tools are available (search for compiler construction tools), specifically *JavaCC* and *CUP* for those who program in Java. *Pyparsing* provides parsing facilities for the Python programmer. There are tools to generate top-down parsers as well as top-down parsers and tools combining lexical analyser and syntax analyser generation. These are all well-worth investigating.

If you have to write a compiler for a real language, it is vital that you start off with a detailed knowledge of that language. If that language is C, then [11] is an excellent starting point.

Exercises

5.1 The rules of DL specify that names have to be declared before they can be used. This effectively disallows mutually recursive functions. How would the design

and implementation of DL have to change to allow mutually recursive functions? Implement these changes.

5.2 Use a standard grammar for arithmetic expressions and write a desk calculator program based on a top-down grammar. It should support integer values, add, subtract, multiply, divide and parentheses and should evaluate expressions typed in a line at a time. Compare this implementation with the version produced in Sect. 5.2.2.2.

5.3 Extend the calculator from Sect. 5.2.2.2 to support unary minus, floating point arithmetic, further scientific functions (such as trigonometric functions), hexadecimal arithmetic, etc.

5.4 It may be tempting to rewrite the example top-down parsing code in Sect. 5.3 as:

```
astptr P() {
    return newnode(N_PNODE, Q(), R());
}
```

Consider whether this would be acceptable.

5.5 At the moment, the syntax of `<bexpression>` in DL is simple because of the lack of boolean operators such as *and*, *or*, *not* and so on. Devise an appropriate syntax and implement the syntax analysis.

5.6 This is a rather open-ended exercise, but useful if you intend to use *bison* seriously. Sometimes things go wrong with *bison* grammars and shift/reduce or reduce/reduce errors are reported. Produce simple grammars exhibiting both of these errors and ensure that you understand why they occur and what is going on.

5.7 Look again at the C example in Sect. 5.7. Try to predict the output when that program is run on your system. Then try compiling and running it.

5.8 Try writing a parser using a top-down parser-generating tool. Compare the experience with writing the same parser using a bottom-up parser generating tool such as *bison*.

References

1. Johnson SC (1975) Yacc – Yet Another Compiler-Compiler. AT&T Bell Laboratories, Murray Hill, New Jersey. Computing Science Technical report 32
2. Free software foundation. GNU bison (2014). https://www.gnu.org/software/bison/. Accessed 31 Jan 2016
3. Technical university of Munich (2015). CUP – LALR parser generator for Java. http://www2.cs.tum.edu/projects/cup/. Accessed 31 Jan 2016
4. Tremblay J-P, Sorenson PG (1985) The theory and practice of compiler writing. McGraw-Hill Book Company, New York
5. Mogensen TÆ (2011) Introduction to compiler design., Undergraduate topics in computer science. Springer, Berlin

6. Levine J (2009) Flex & bison. O'Reilly Media, Sebastopol
7. Aho AV, Lam MS, Sethi R, Ullman JD (2007) Compilers – principles, techniques and tools, 2nd edn. Pearson Education, Upper Saddle River
8. Kernighan BW, Ritchie DM (1988) The C programming language, 2nd edn. Prentice Hall, Englewood Cliffs
9. Holub AI (1994) Compiler design in C, 2nd edn. Prentice Hall International, New York
10. Fraser C, Hanson D (1995) A retargetable C compiler: design and implementation. Addison-Wesley, Reading
11. van der Linden P (1994) Expert C programming: deep C secrets. Prentice Hall, Englewood Cliffs

Chapter 6
Semantic Analysis and Intermediate Code

The semantic analysis phase of a compiler is the last phase directly concerned with the analysis of the source program. The syntax analyser has produced a syntax tree or some equivalent data structure and the next step is to deal with all those remaining analysis tasks that are difficult or impossible to do in a conventional syntax analyser. These tasks are principally concerned with context-sensitive analysis.

Much of the work of the semantic analyser is based around the management of names and types. It has to deal with declarations and scopes, checking that each use of a name corresponds to an appropriate declaration (assuming that we are dealing with a programming language where names have to be declared). This means that this phase has to apply the language's scope rules and it has to check that each use of a name is allowed according to the type rules of the language. Whenever an operator is used to generate a new value, the type rules for that operator have to be checked to ensure that the operands have appropriate types, the correct form of the operator has to be selected, type transfer operations have to be inserted if necessary, and so on. These are not processes specified by the BNF/EBNF syntax of the language. Instead these rules are usually specified by a narrative associated with the programming language specification. These tasks are usually more easily done once syntax analysis is complete.

The semantic analyser may also have to change the program's representation from a tree to some form of intermediate language. There are good reasons for this change of format. Flattening the tree to produce a simple, linear intermediate form should not be difficult, but there may be good reasons for generating an intermediate form of greater complexity, supporting code optimisation. Designing or choosing such an intermediate representation may be more challenging.

The symbol table is the central data structure in this phase. Data is inserted when symbols are declared and data is read whenever symbols are used. Chapter 5 has already shown how the symbol table can be managed largely during the syntax analysis phase but it is also possible to delay at least some of the symbol table operations until semantic analysis is being carried out.

Exactly what has to be done in the semantic analysis phase depends on the nature of the language being compiled. It may be possible to do much of the type checking, if not all, during syntax analysis. This may make sense for simple languages with a limited set of type rules. The DL language falls into this category. Furthermore, it may even be possible to generate target code directly from the abstract syntax tree. Obviously this leaves little for the semantic analyser to do. But there are usually strong reasons in anything but the simplest of compilers to have a separate semantic analysis phase. Separating out the type management makes good software engineering sense and the choice of an appropriate intermediate form rather than a tree simplifies the job of generating high-quality target code.

6.1 Types and Type Checking

The concept of a data type is central to the design of programming languages. A type system should be able to help describe the structure and behaviour of valid programs. Support for typing can improve the programmer's productivity, speed up debugging and offer opportunities for target code optimisation. Strong typing can help avoid some runtime errors. It is always preferable to move as much error detection as possible from runtime to compile time, finding the errors at the earliest stage possible. It is true that strong typing can sometimes get in the way, but it is now generally agreed that the benefits outweigh the disadvantages.

Our task as compiler writers is to develop techniques for the management of type data within the compiler, to make sure that the operations being attempted in the source program are compatible with the language's type rules and to make the most of the available type information in order to generate correct and efficient target code.

6.1.1 Storing Type Information

Sometimes handling types is really simple. For example, DL allows integer variables and integer arrays to be declared and it also supports functions. So a name used in a DL program refers to one of these three data types. This can be represented in the symbol table by storing an integer tag taking, for example, the values 1, 2 or 3. DL is designed so that the syntax analyser can determine the correct type of a name according to its syntactic context, and the data in the symbol table is checked by the syntax analyser to ensure that the use of a name is compatible with its definition. A language rule stating that all variables have to be declared before they are used (i.e. earlier in the source of the program) makes the handling of variables and their type information very much easier.

Adding further types may be managed simply by increasing the number of distinct tag values, but as soon as user-defined types are introduced, a new approach is needed. There is no standard way of representing this information. The design of the data

structures used will be influenced by the range of type specifications available and how they have to be used when checking for the correct use of names in the program. For example, the `typedef` feature in C allows arbitrarily complex types to be defined and named. Here, the type information can be represented as an abstract syntax tree of the type specifier linked to by the type name's entry in the symbol table.

Do not underestimate the potential complexity of representing type information. Dealing with C structs and unions, function declarations, enumerations, arrays and so on requires a careful plan, and tying that plan to the formal syntactic definition of the language is doubtless a good approach.

6.1.2 Type Rules

Having dealt with type declarations, storing the type information into data structures accessible via the symbol table or perhaps directly from nodes in the abstract syntax tree, this information can then be used to validate the use of variables and other structures in the program being compiled. Different programming languages have different type models. The traditional model, found in many imperative programming languages such as C, is to implement *static typing* where a variable is declared with a type and that variable keeps that type throughout its existence. So, in C, if `i` is declared as `int i;` the compiler can cause an integer-sized location in memory to be set aside when the program runs, for the time during which `i` exists. If an attempt is made to store a floating point value into `i`, then the compiler can issue an error message. Note that this checking is done during compilation during syntax analysis, or more likely during semantic analysis.

This approach is contrasted with languages supporting *dynamic typing* where the type of the data stored in a variable can change during that variable's existence. So statement sequences of the form `i=1 ... i=3.14 ... i="Hello"` are valid. Note that this does not imply that no type checking is required — it is just that it is now unlikely that all type checking can take place during compilation. Instead, some or all of the type checking has to be delayed until runtime and this implies that the implementation of the language has to support the association of a type with each variable *at runtime* so that checking code, generated by the compiler or included in the interpreter, can ensure that the type rules are being adhered to. Clearly there is a runtime overhead in doing this. Languages such as Perl, MATLAB and Python are in this category.

Clearly, hybrid approaches are possible and may be necessary for some languages. To maintain strong typing, not all checking is possible at compile time and so at least some runtime checking is required too. However, in this chapter we will be concentrating on compile-time static type checking.

Most modern programming languages are designed to support strong or fairly strong type checking. But some programming languages are untyped and then it is up to the programmer to keep track of the nature of the data stored in each variable. In some languages type checking is present but not very strict. For example, C's union

construct provides a loophole for the type checking system. If the language specifies type checking rules, then they have, of course, to be supported by the implementation. Separating this potentially complex type checking into a separate phase of semantic analysis is sensible.

Knowing the type of a variable at compile time or at runtime allows simplifications to be made to the way in which the programmer manipulates data. If variables `i` and `j` are declared with type `int` and `a` and `b` are declared with type `float` (and note that we are *not* assuming that the programming language is C in these examples), then it is likely that we can write statements such as `i = i + j` and `a = a + b`. In the first assignment, the addition operator is assumed to take two integer operands and produce an integer result. In the second assignment, the operands and result are all real numbers. The addition operator is performing two distinct tasks, implemented using different instructions on the target machine. This has important consequences for the compiler and it is during the semantic analysis phase in statically typed languages that a type for the operator can be selected. In dynamically typed languages, the operator choice may well have to be delayed until runtime.

The use of the single addition operator for both integer and real addition is an example of *operator overloading*. It could get worse. Maybe the programming language allows the addition operator to be used for further purposes such as the concatenation of strings and forming the union of sets. But as long as the type rules of the language are clear, the implementation of this should be feasible. There will also be rules forbidding certain type combinations. For example, addition of a string and an integer may not be allowed. Again, the semantic analysis phase is a good place to generate error messages about such disallowed operations. Another possibility is shown by the example `a = i + j`. Here, the addition operator is an integer add, but the assignment operator has to transform an integer value into a real value and the semantic analyser has to tag the assignment operator accordingly. Similarly in the case `a = b + i`, a type transfer has to be included to convert `i` into a real value before using it as an argument to the real addition operator. These examples obviously assume that we are working with an appropriately flexible programming language. Similar considerations apply to the use of constant values in expressions. For example, the assignment `a = b + 2` may require a compile time (rather than at runtime) conversion of the integer value 2 to the real value 2.0, assuming that the language allows such mixed-mode arithmetic expressions.

Associated with the typing of operators is the notion of *type equivalence*. When are two types considered to be the same? A simple assignment (`p = q`) requires *compatibility* between the types of `p` and `q`, and we can safely assume that type equivalence implies compatibility. The language's rules for type equivalence are relevant to the coding of the semantic analyser. We can distinguish between *name equivalence* and *structural equivalence*. Types are said to have name equivalence if they have the same name. Types have structural equivalence if they have the same structure. For example, in Fig. 6.1 the two C structure types `person1` and `person2` are structurally equivalent but do not have name equivalence.

The correct algorithm for the programming language has to be used to determine whether two types are equivalent.

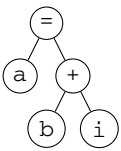
```
struct person1 {
   char *name;
   int yearofbirth;
}
```

```
struct person2 {
   char *surname;
   int birthyear;
}
```

Fig. 6.1 Structural equivalence

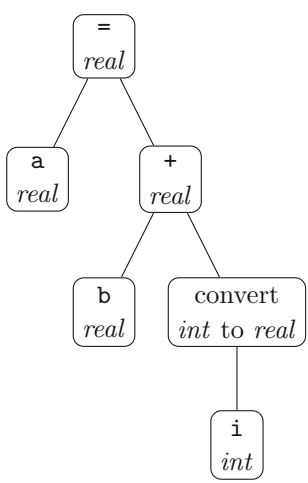
6.1.2.1 Type Checking

An intuitive way of managing type checking in the semantic analysis phase is to traverse the abstract syntax tree, tagging nodes in the process. Using the variable types from the section above, consider the source statement `a = b + i`. The syntax analyser will represent this as a tree.



The semantic analyser will start at the root node containing the assignment operator and perform a recursive post-order traversal of the tree, tagging nodes as it goes. The first node to be tagged contains `a` and this is tagged with the type *real*, then the `b` node with *real* and then the `i` node with *int*. The `+` node is then examined and a decision has to be made whether this operator applied to a *real* and an *int* is valid and if so, how it should be interpreted. Assuming that conventional mixed-mode arithmetic is permitted, we have to introduce a new node to convert the *int* `i` to a *real* value and tag the `+` node with *real* (it is now an addition of two *real* values). Finally, the top assignment node is tagged with *real*. The annotated tree is shown in Fig. 6.2.

Fig. 6.2 Annotated tree



This tree traversal/annotation process is driven by the type rules of the language being compiled. Each node requiring a type will be tagged with a type derived from the types of its children according to the type rules. And tree modifications will be made where necessary. Separating this task from the syntax analyser makes the compiler's code much simpler and more maintainable.

6.1.2.2 Type Rules

Let us consider an example of another language where type rules are needed. Suppose a programming language for the manipulation of dates has been devised. This supports an *int* data type to represent whole numbers of days and a *date* data type to represent a calendar date. Literal integers and dates are supported together with a set of operators to manipulate these values. So, for example, a date variable could be initialised to contain a certain date and the add operator could be used to generate the date 100 days later. To manipulate days and dates using the operators add and subtract, the following combinations are possible.

Expression	Result
days + days	days
days + date	date
date + days	date
date + date	*error*
days – days	days
days – date	*error*
date – days	date
date – date	days

These rules obviously depend on the details defined in the language specification. The introduction of other operators may well benefit the language, but the semantics of operators such as multiply and divide may be more difficult to envisage when manipulating dates.

6.2 Storage Management

Within syntax and semantic analysis, decisions have to be made concerning the run-time representations of data and language-supported data structures. Target machine-dependent details are better left until the code generation phase but the intermediate code to be generated by the semantic analyser really needs some sort of data storage model which can be mapped later to physical storage in the target machine. Provision has to be made for the storage of data in variables, function (or procedure or method) parameters and results, temporary variables, static data and so on according to the design of the programming language being compiled.

The semantic analysis phase has no particular concern with C's dynamic storage allocation using system-defined functions such as `malloc`. These are facilities

supported by the runtime library and space is usually allocated from a memory area provided by the operating system. These system calls are treated as conventional function calls. They just refer to code defined in a system library which can be linked to the code generated from the user program by the compiler to form a complete executable program.

6.2.1 Access to Simple Variables

At some stage in the language implementation process, variable names have to be mapped to physical storage locations. There are clearly many ways in which this can be done. Probably the most practical and likely approach is to think of one or more runtime storage areas, each being a contiguous area of memory, pointed to and hence identified by one or more target machine registers. Then, as far as the compiler is concerned, variables can be represented by identifying a storage area and an offset in that storage area. For particularly simple languages where all variable storage requirements are known at compile time, a *static allocation* scheme can be used. The semantic analysis phase can associate a single offset value for each variable and these offsets can be saved in the abstract syntax tree and hence in the intermediate representation each time a variable is used. Dealing with different memory-sized data representations is not a problem because the type information is stored in the symbol table and the tree and this can be translated into a memory size indication which can then be translated into a target machine offset during code generation. Instead, it also may be possible to perform the translation of text variable names into offsets in the syntax analyser, allocating space as variables are declared.

6.2.2 Dealing with Scope

The simplistic view of variable declaration and use presented in the last section is inadequate for any reasonable programming language. At the very least the concept of *local* variables is required.

Suppose our programming language allows declarations of this form:

```
{  int i, j;
   .
   .
   {  int j, k;
      .
      .
   }
   .
   .
```

```
        {   int l, m;
            .
            .
        }
        .
        .
    }
```

Here, variables `i` and `j` are globally declared, `j` (another, different one) and `k` are locally declared in the first inner block and `l` and `m` are locally declared in the second inner block. The compiler has to map these six distinct variables to six storage locations. A reasonable approach is to treat the runtime storage area as a *stack*. Let us assume that we are using a storage model where each integer variable requires a runtime location of size one unit (this can be scaled appropriately later if necessary) and the compiler maintains a stack pointer indicating how many stack locations have already been allocated to variables. This is initialised to zero. Global variable `i` is then assigned to the location at offset 0 and `j` to offset 1. When the start of the first inner block is encountered, the current value of the stack pointer is saved (it has the value 2). Variable `j` (the second one) is set to have offset 2 and `k` set to offset 3. At the end of this inner block, the space used for variables `j` and `k` can be relinquished by restoring the stack pointer to its saved value from the start of the block (2). On entry to the second inner block, the stack pointer is again saved and variable `l` is set to have offset 2 and `m` set to offset 3. This reusing of storage locations is possible because the variables of the first inner block and the variables of the second inner block cannot exist at the same time. The handling of the scoping of variables is managed by the symbol table for example, by structuring it as a stack as well. The use of a stack-based symbol table was mentioned in Sect. 5.6.

There may, of course, be complications here, depending on how the language is defined. The assumption that variable space can be reused for the inner blocks fails if the language demands that space is allocated statically. The language implementer may also decide that the effort required to manage this form of dynamic variable allocation is not worth it for the potential space savings. For example, it would be reasonable for a C compiler to allocate all the space for variables in a function, even those declared in inner blocks, in a single contiguous area.

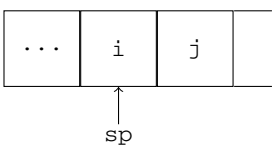
6.2.3 Functions

The storage allocation problem is made more complicated when functions are involved. When a function is invoked, then memory space has to be available for the storage of local variables declared in that function and also for the storage of the function's arguments. In particular, when recursive function calls are permitted, static space allocation will not be adequate since new space has to be allocated for local variables on each recursive call and it is in general impossible to predict in

advance the actual maximum depth of the recursion. A *dynamic allocation* scheme is required. The usual way of implementing this is via a *stack*. Details of this implementation approach are left to Chap. 8 and only the basic principles are covered here to highlight how both local and global variables can be accessed.

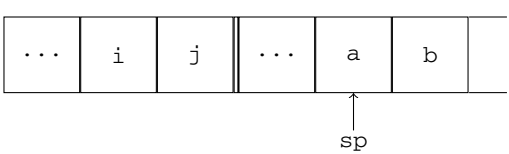
Consider a simple program, written in a language following the conventional scoping rules, with a main program declaring integer variables `i` and `j` and a function `r` with local variables `a` and `b`. The function `r` calls itself recursively. Each time `r` is entered, space for a new set of variables `a` and `b` has to be found and the addressing environment changed so that each time the code in `r` references `a` or `b` it is the "new" `a` or `b` being used. If storage for these variables is managed in a piece of contiguous memory used as a stack then the space allocation problem is easily solved.

When the main program starts, the only variables accessible (in scope) are `i` and `j`. The stack in which variables are stored looks like this:

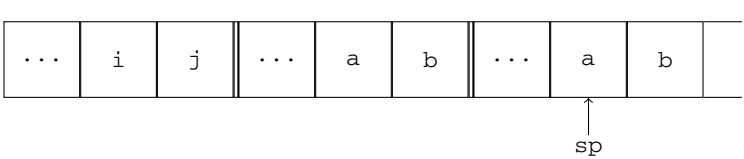


Here, `sp` is the stack pointer, pointing to the first variable of the set of local variables in scope. At runtime, the stack pointer will be held in a machine register and indexed addressing will be used to access the variables stored on the stack. The stack locations lower in the stack than the location for `i` are used for miscellaneous linkage information. They will be described in Chap. 8.

After the first call to `r`, space is set aside for variables `a` and `b` declared in `r`. A new *stack frame* is created. In these diagrams, a double line is used to separate stack frames.



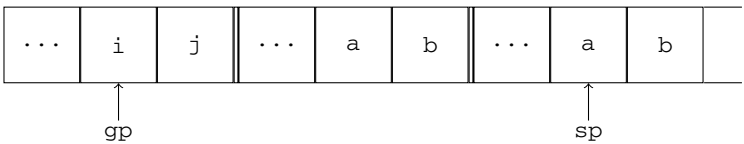
Variables `a` and `b` can now be accessed by direct indexing off the stack pointer (offsets 0 and 1 respectively). Then `r` calls itself recursively and the stack now looks like this:



The crucial point to note here is that variables `a` and `b` are *still* accessible via offsets 0 and 1 from the stack pointer as before, but it is the *new* `a` and `b` that are being accessed.

So as far as the compiler is concerned, all it needs to do in generating intermediate code to access local variables like `a` and `b` is to record their stack offsets, and these should have already been stored in the symbol table.

There is, however, a slight complication. Most languages of this type (such as C) allow the global variables `i` and `j` to be accessed in the code of `r`. How can access to these variables be specified in the intermediate representation? All that needs to be done is to explicitly record that these are *global* variables 0 and 1 and code can ultimately be generated to access `i` and `j` via some global variable pointer (`gp`):



Further complications arise in languages such as Pascal where function definitions can be nested. For example, if function `f1` is defined in the main program and function `f2` is defined within `f1`, the language's scope rules allow code in `f2` to have access to `f2`'s local variables, `f1`'s local variables and the global variables. Managing this requires multiple pointers into the stack (gathered together in a data structure called a *display*) or a chain of pointers running down the stack identifying the currently active stack frames. This chain of pointers is called the *static chain*.

6.2.4 Arrays and Other Structures

So far we have concentrated on the storage and handling of simple variables. But most programming languages offer data structuring facilities. The most commonly offered structure is the array, and because arrays are central to the coding of so many important computer applications, efficient implementations of arrays are essential. Also, many programming languages provide an aggregate data type. This is a single structure containing multiple elements. A C `struct` is an example. Unions are often available, allowing data of different types to be stored in a single unit. *Object-oriented programming languages* need implementation techniques for the storage of objects made up of a collection of elements or fields, together with code (as methods) to manipulate this data.

There is an issue here concerned with target machine independence. Ideally the semantic analysis phase should be independent of the design of the target machine but if this phase has the responsibility for designing the layout of these data structures, complete target machine independence may prove to be difficult.

When an array is declared, an entry is made in the symbol table to store the array's name. Each element in the array is of the same type and this type has to be stored in the symbol table too, together with a specification of the array bounds. Also, if multi-dimensional arrays are supported by the language, the number of dimensions

has to be stored. Some languages allow the subscript ranges to be specified while other languages assume that each subscript starts at 0 or 1 and a maximum range is recorded. Some languages permit non-integer types to act as subscripts. All this information has to be stored in the symbol table too.

What should happen when the semantic analyser encounters a statement containing an array reference? Clearly, the type checking process will make use of the information stored in the symbol table to partially validate the array reference but more may be required. An array access has to be translated during the compilation process into a form appropriate for target machine implementation, requiring the address of the array element being referenced. For example, in C, if an array is declared as `int x[5]` and a reference is made to element `x[i]`, then at *runtime* (unless the compiler can predict the actual value of `i`) the computation

address of the first element of `x` + `i` * size of a C `int`

has to be performed to find the address of the indexed element. Assuming that the abstract syntax tree makes use of nodes directly representing array access, the semantic analyser could expand these nodes to represent this full address calculation process. This expansion may seem unnecessary, but there are good reasons for carrying it out before the intermediate representation of the program is passed on to the next phase of compilation. This is because expressing array access in this low level form provides more machine-independent optimisation opportunities.

6.2.4.1 Multi-dimensional Arrays

When multi-dimensional arrays are used, the address calculations become a little more complicated. For example, if an array is declared in C as `int y[10][5]` then 50 `int` locations are set aside and these locations can be individually accessed by references of the form `y[i][j]`. Here, `i` should be in the range 0 to 9 and `j` in the range 0 to 4. The address computation for this element then becomes

address of the first element of `y` + (`i` * 5 + `j`) * size of a C `int`

with the 5 in the computation coming from the second bound of the array. Similar computations can be done for arrays of three or more dimensions. Note the significant computational cost for accessing array elements, particularly in multi-dimensional arrays. As will be seen, optimisations in various circumstances may be able to reduce this cost. Also note that there are choices to be made concerning the order in which array elements are mapped to linear storage. Here, for two-dimensional arrays, the C rules are followed so that the second subscript varies most rapidly. This is not the same in all programming languages. Some languages specify that the mapping is done so that the *first* subscript varies most rapidly and some others say nothing and the storage order is implementation-dependent.

6.2.4.2 Array Bound Checking

Should array bound checking be implemented? Should code be generated so that each time an array is accessed, the subscript values are checked to ensure that they lie in the permitted ranges? Some array bound checking may be done at compile time if static analysis (see Chap. 7) can determine the actual value of an index. But runtime array bound checking comes with a computational overhead. Also it is possible but unusual for the rules of the language to make array bound checking inappropriate. The addition of this checking code can be done by modifying the tree, during the generation of the intermediate representation or later during target code generation.

6.2.4.3 Structures and Unions

The ability to group together data items into a single named structure is common in today's programming languages using a struct, a record, a part of an object, and so on. The structure can be referenced as a whole while retaining access to the individual fields. Fortunately, implementation poses no real difficulties with the structure information, field names and types and so on all being placed in the symbol table. For example, the node type in the abstract syntax tree in Chap. 5 was defined as

```
typedef struct tnode {
  int asttype;
  int astdata1,astdata2;
  astptr p1,p2,p3;
} astnode;
```

where a node contains three integers and three pointers. The structure can be seen as a block of contiguous memory locations, large enough to store all the fields. All the field names and their offsets in the storage associated with the structure are stored in the symbol table. The sizes of the fields are either assumed at this stage or they can be decided upon later in the code generator. Target machine constraints may influence the *order* in which the fields are stored. For example, consider a structure defined as containing two characters, an integer and then two more characters, to be implemented on a machine where characters occupy one byte each, an integer requires four bytes, and integers have to be stored on word boundaries in memory. In this case, it may make sense to pack the four characters together and follow them with the integer rather than retaining the original ordering, where two bytes of padding would be required for each instance of the structure.

Unions are handled in a similar way. For example

```
union utype {
  char xch;
  int xint;
  float xfloat;
} uval;
```

causes all the names to be stored in the symbol table, the size of the union structure being set to the size of the largest member of the union, and it can be regarded as a struct with all the fields having a zero offset.

6.3 Syntax-Directed Translation

We have already seen how a formal specification of a context-free grammar can be used to produce a syntax analyser. The natural way of progressing from the recognition of syntactic constructs to code for a complete syntax analyser is to make use of the ideas of *syntax-directed translation*, where grammar productions are associated with program fragments or some more formal rules and these program fragments perform the translation of each syntactic construct to some new program representation. This new program representation can of course be an abstract syntax tree and in this case the translation process has already been discussed. But it may be possible to go further and cause the actions associated with the productions to do more work, generating intermediate code, target machine code or even interpreting the code directly. The translation of the source language is being driven by the syntax analysis process.

6.3.1 Attribute Grammars

To what extent can this translation process be formalised? An attribute grammar can extend a context-free grammar by allowing the manipulation of *attributes*. Each symbol in the grammar is associated with a set of attributes and each attribute has a value (such as an integer or a type specification). Attributes are computed at each node in the parse tree or, equivalently, at each production rule used in a reduction. An attribute is said to be *synthesised* if its value is computed from data in the node itself and from data from that node's children. *Inherited* attributes use data from or via the node's parents as well as from the node itself.

This approach allows context-sensitive analysis to be formalised. For example a type attribute can be associated with symbols and type rules can be checked at each node. The use of attributes can be taken further. Consider a language (defining expressions and so on) for a numerical calculator. Here, an attribute for each grammar symbol could be a numerical value and the attribute grammar rules actually perform the computation specified. So, for example, the rule for an `<expression> + <term>` node could synthesise the attribute for the node by adding the attributes from its two children.

This approach should look familiar. With tools such as *bison*, the actions specified by each grammar rule correspond closely to the management and calculation of attribute values. For example:

```
expression: expression '+' term     { $$ = $1 + $3; }
```

gives a clear example of the use of synthesised attributes, with the attribute associated with `expression` and `term` being the numerical values of the expression and term.

Software tools are available to at least partially generate parsers or translators via the use of attribute grammars, thus helping to automate the handling of semantic information. There is a parallel here with functional programming, with the returning of attribute values via children being similar to a function call. This approach is good for simpler applications but implementation practicalities, particularly concerned with the handling of inherited attributes, makes it difficult to use for more complex grammars. Tools such as *bison* allow ad hoc solutions to be developed by associating an action expressed as code with each production and the code of the action can perform the required checking and manipulation of the attributes.

Do attribute grammars provide us with a general-purpose tool for the construction of practical compilers? In particular, can they be used to manage the process of target code generation by merging it into the parsing process? The answer is probably no. The key problem is that processing a parse tree node effectively is not just a local operation because information is required from children, from ancestors as well as from the node itself, and managing all this non-local information via attribute grammars is not easy. Translating each node in isolation is appropriate for simple cases only. Instead, avoiding complexity by developing the compiler as a series of communicating phases still seems very attractive. The tree generated by the syntax analyser can then be annotated by the semantic analyser and intermediate code generated, and there is no reason why a formalisation based on the attribute grammars cannot be used here. A separate process can optimise the intermediate code and the independent code generation process can then produce target code.

6.4 Intermediate Code

The abstract syntax tree generated by the syntax analyser and annotated by the semantic analyser provides a mid-way representation of a program in the process of compilation. But is this a good data structure from which to generate target machine code? Is it a good intermediate representation?

The intermediate representation (IR) forms an interface between specific compiler phases, typically between the front-end and the back-end of the compiler. The last phase of the front-end is usually the semantic analyser and the back-end is responsible for code generation. The compiler could make use of the same IR between multiple back-end code optimisation phases. And there is no reason why a compiler cannot be designed to use more than one IR.

The IR should be designed to facilitate the interface between the front-end and the back-end. It should not be regarded as an interface with which the user of the compiler is particularly concerned, but ensuring that there is a way for the compiler writer or interested user to inspect a human-readable representation of the IR is important. The design should be such that the generation of the IR in the front-end should not be overly difficult and the IR should support effective code generation and

optimisation. It may be helpful to think of the IR as the machine code for a high-level and structured target machine. The IR should be sufficiently expressive so that all the constructs of the source language can be encoded cleanly into IR statements or structures. Assuming that we are thinking about a traditional compiler generating code for a target machine, the IR should be easily translatable into target machine code. Another important characteristic of an IR is that it should be able to support aggressive code optimisation, thus permitting the generation of high-quality code. In many modern compilers this is the IR's key design aim.

The IR can play an important part in adapting the compiler to compile a different source language or generate code for a different target machine. Here, new design aims are appearing for the IR and issues of source language and target machine independence become significant. Furthermore, the IR may be easily interpretable, providing a rapid route to programming language implementation. These issues of compiler implementation and porting are discussed in Chap. 9.

Is the IR file or data structure generated by the front-end completely self-contained? Does the back-end of the compiler require access to other data structures such as the symbol table? This is not a critical issue, but there is some attraction in making the IR self-contained in order to simplify the interface to the compiler's back-end. It may be necessary for the front-end to export all the source program symbol information so that it can be held in the executable program generated by the back-end, making it possible for a runtime symbolic debugger to relate storage locations to source program variable names.

This book concentrates on the use of a linear IR but some graph-based IRs are also described. There are advantages for the use of a linear IR. For example, it should be easy to generate target code (not necessarily optimised) from linear code by translating statement by statement. There are many forms of linear IR in use today and there is no international standardisation of IRs as there often is for programming languages. But graph-based IRs are popular too, again with a variety of designs. Some of these IR designs are allied to particular programming languages or language types, some designs are good for particular types of optimisations and a few designs try to be general purpose, appropriate for many source languages and target machines. In this section various IR types are outlined. For DL's implementation, a simple linear IR will be used and this IR can illustrate a range of optimisations.

6.4.1 Linear IRs

A linear IR can be regarded as the machine code for a virtual machine. This virtual machine should be regular and structurally simple, yet have enough expressive power to cope with all aspects of the language being compiled. A wide range of such IRs are used today, but it is possible to adopt existing high-level languages as the IR. For example, C has been used in many compilers as an IR. The generation of C from the syntax tree should be easy, and this C can be compiled by an existing C

compiler to produce an executable program, without having to write a conventional code generator at all.

6.4.1.1 Traditional IRs

Generating the IR from the abstract syntax tree using just a simple prefix or postfix traversal will yield a form of linear IR. Such IRs were sometimes used in the early days of compilers, particularly for stack-based target hardware. These representations have advantages of simplicity but they are not so popular now because they do not work quite so well with the powerful optimisation algorithms used in today's compilers. Although stack-based IRs, such as the Java Virtual Machine, are used, being interpreted or translated into a target machine code, non-stack based linear IRs based on instructions containing an operator, up to two arguments and a result have become more popular. These instructions can be structured in a variety of ways [1]. A natural way of representing these instructions is by using a *three-address code*, the three addresses being used for the two operands and the result. This results in a readable IR and one that can be generated from the tree fairly easily. For example, the statement `x = y*3 + z;` could be translated as:

$$\begin{aligned} t_1 &= y * 3 \\ x &= t_1 + z \end{aligned}$$

t_1 is a compiler-generated temporary variable. The instructions are written here using symbolic references to variables (it is assumed that a symbol table is available if necessary) and this makes the code much more readable. Some instructions (such as $x = -y$ or a simple copy $x = y$) only require one operand and so the second operand is just omitted. A range of operators is required to match the functionality of the source language (and hence the nodes in the abstract syntax tree) including arithmetic operators, conditional and unconditional jumps, array access and a function call mechanism, including arguments. There are similarities in style between three-address code instructions and the instructions of a RISC processor and those similarities might help in the code generation process. In Sect. 6.5 a complete three-address code IR suitable for DL is defined.

6.4.1.2 Static Single Assignment Form

Three-address code, as described above, explicitly defines the control flow of the program. Paths through the program follow the statement order and are changed by jump instructions and labels. But there are many potential optimisations that are based on analysis of *data* values. For example, is it possible to predict using static analysis (i.e. without executing the program) the values of particular variables after the execution of a piece of code? This type of analysis can be facilitated by adding data flow information to the IR.

Static single assignment (SSA) form provides this information [2]. SSA is *not* an IR. It just defines some structuring rules that can be applied to an IR such as three-address code so that the data flow information is made explicit. In SSA form, each variable can only be assigned to *once*. So if a piece of code makes an assignment to a particular variable and then that variable is assigned to again, the second assignment must be modified to use a new variable. For example, the code

```
x = a + b
y = x + 1
x = b + c
z = x + y
```

is transformed to

```
x1 = a + b
y1 = x1 + 1
x2 = b + c
z1 = x2 + y1
```

In the original code, `x` is assigned to twice and hence the introduction of x_1 *and* x_2. This representation makes it clear that, for example, the use of x_2 in line 4 corresponds to its definition in line 3. That use on line 4 has nothing to do with the definition on line 1.

But this protocol causes a problem. Consider the code

```
x=1;
if (a<0) x=0;
b=x;
```

which can be translated into a three-address code version:

```
x = 1
if a >= 0 goto l1
x = 0
l1 :
b = x
```

But when translated into SSA form

```
x1 = 1
if a >= 0 goto l1
x2 = 0
l1 :
b1 =???
```

the value to be assigned to b_1 is going to be either x_1 or x_2 and it is only at runtime when the decision can be made. To resolve this issue, SSA allows us to write

```
x1 = 1
if a >= 0 goto l1
x2 = 0
l1 :
b1 = φ(x1, x2)
```

where the ϕ-function is a function to indicate the meeting of control flow paths and the result is chosen according to which variable definition (the most *dominating*) was made in the control flow path most recently executed. At first sight implementing the ϕ-function appears difficult. But it can be implemented during code generation by ensuring that all variable parameters of the ϕ-function share the same register or storage location or by ensuring that appropriate register copies are made.

All this additional complexity is justified by the fact that the data dependence information embedded in SSA is necessary for the implementation of a wide range of optimisations. For example, constant propagation is made very easy. If, say, variable t_{19} is found to have the value 3, then replacing all uses of t_{19} by 3 and deleting the definition of t_{19} is easily done.

The generation of SSA is not trivial but good algorithms are now available. For further information see [2–4].

6.4.2 Graph-Based IRs

Although it is often convenient to have a human-readable linear IR, using more powerful data structures may offer advantages. The syntax tree data structure produced by the syntax analyser can be regarded as an IR in itself.

6.4.2.1 Trees and Directed Acyclic Graphs

The abstract syntax tree is easy to construct and can be used directly for the generation of target code. A tree traversal based on post-order can be used for code generation, with code being generated for the children of a node before dealing with the operation specified in the node. This appears to be a very attractive approach to code generation, but there is a clear disadvantage in that it is extremely difficult to generate high-quality code from this representation. As soon as efforts are made to optimise the generated code, the simple tree traversal has to be abandoned. The code generation process ceases to be local and when generating code for a particular node in the tree, neighbours of the node (children, parents and beyond) have to be examined too.

A good example of this issue is when code-generating expressions with common subexpressions. For example, consider the expression $a * a/(a * a + b * b)$. The corresponding tree is shown in Fig. 6.3.

There is a common subexpression here $(a * a)$ and this can be detected by traversing the tree looking for common sub-trees. The links in the tree can easily be changed to reflect the fact that only one instance of this subexpression is necessary. This is shown in Fig. 6.4.

Note that this structure is no longer a tree because the subtrees of the root node are no longer disjoint. It has become a directed acyclic graph (DAG). This means that traditional tree traversal algorithms will no longer be appropriate and more complex algorithms are required to traverse and extract data from the graph.

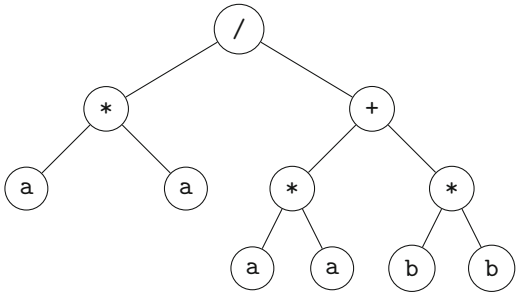


Fig. 6.3 Tree for a * a/(a * a + b * b)

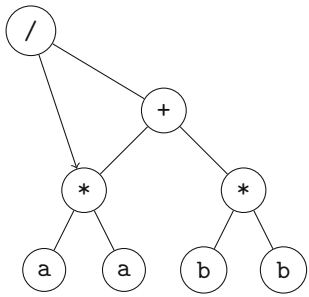


Fig. 6.4 Common subexpression

Finding common subexpressions is a significant optimisation but using a DAG does not automatically solve the problem. Common subexpressions need to be detected even when they are not within the same statement or expression. Some sort of control flow analysis is needed too, ensuring that if common subexpressions are found there are no intervening operations that could change the computed value of the subexpression. This issue will be examined in detail when code optimisation is covered in Chap. 7.

6.4.2.2 Control Flow Graphs

The control flow graph (CFG) [5] is a directed graph made up of nodes and edges with the edges representing possible paths of execution. Each node in the graph represents either a single instruction or a *basic block*. A basic block is an instruction sequence with a single entry point at the first instruction and an exit point at the last instruction. Any jump (conditional or unconditional) instruction must be the last instruction in a block. If a block ends with a conditional jump, there are two edges coming out of the node representing that block.

For example, consider the code fragment:

```
t = 0;
i = 0;
while (i <= 10) {
```

```
    t = t + i;
    i = i + 1;
  }
  x = t;
```

This code can be translated into three-address code:

```
t = 0
i = 0
l2:
if i > 10 goto l1
t = t + i
i = i + 1
goto l2
l1:
x = t
```

These instructions can be grouped into basic blocks and control flow between the basic blocks added, as shown in Fig. 6.5.

This is the control flow graph and it is used as an IR in many compilers. It is a good basis for code optimisation and it is not difficult to construct. There is a need for some additional structure to represent functions and the usual approach is to have a CFG for each function with a separate call graph to show the possible paths for function calls.

In this example, the code in the basic blocks has been shown in three-address code, but other representations are possible. For example, the three-address code can

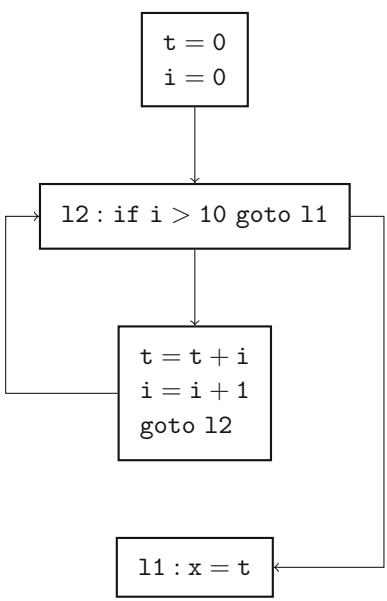


Fig. 6.5 Basic blocks with control flow

be converted into SSA form and then the CFG can be constructed. This approach offers further opportunities for optimisation.

6.4.2.3 Data Dependence Graphs

Graphs can also be used to show how *data* flows in a program. For example, in the code:

```
x = a + b;
y = a + 2;
z = x * b;
```

there is a data dependence from the first statement to the third. The variable `x` is set in the first statement and used in the third. The data dependence graph reflects this data flow. The nodes in this graph represent *operations* and the edges connects *definitions* with *uses*.

The data dependence graph for this trivial example would contain three nodes, one for each assignment, with an edge from node 1 to node 3. This implies that the operation in node 1 has to be completed before performing the operation in node 3. Nothing is said about when the operation in node 2 should be performed. So this representation is in general incomplete in that the program's control flow is not uniquely specified by the data dependence graph. The data dependence graph only imposes a partial ordering on the statements or instructions. More is needed for a complete IR. But this structure can be useful when performing various specific optimising transforms such as parallelism detection and instruction scheduling. Whether this data dependence information is included in the IR or whether it is computed when needed later in the compilation process is largely a matter of opinion. Furthermore, data dependence information is implicitly encoded in SSA-style representations and so if an SSA-based IR is used, adding extra data dependence structures may not be quite so useful.

To allow the IR to provide more comprehensive information, the *program dependence graph* was proposed [6]. This structure contains both control and data flow information, with nodes representing statements, predicate expressions or regions (nodes with the same control dependence) and edges representing control or data dependencies. Many further IRs have been developed in recent years, some being designed for general-purpose optimisation and others being focused on specific types of optimisation [7].

6.5 Practical Considerations

Syntax analysers for DL generating abstract syntax trees were developed in Chap. 5. Transforming an abstract syntax tree to a linear sequence of intermediate code instructions is based on a post-order traversal of the tree, outputting code as nodes are visited.

However, this is not a wholly routine task and this section examines some of the issues involved in generating a three-address code intermediate representation.

6.5.1 A Three-Address Code IR

The IR chosen for this example implementation of DL is based on the three-address code described in [1]. The instructions manipulate data held in *virtual registers* and in the virtual machine defined by this IR, there is no limit on the number of registers available. These registers are called `r0`, `r1`, `r2`, …. Variables declared in the DL programs are either *global* variables (declared at the top level) or *local* variables (defined within a function). The global variables are given the names `vg0`, `vg1`, `vg2`, … and these are considered to be mapped on an appropriately sized block of contiguous storage so that the address of register `vgn` is the starting address of the block of storage + n. This allows arrays to be implemented easily. Similarly, the local variables are given the names `vl0`, `vl1`, `vl2`, … and they are placed in a similar contiguous storage area.

The instructions can also use constant values. These are expressed as conventional decimal integers. None of the instructions can use more than three addresses (it being a three-address code).

The instructions used are:

- Assignments with two arguments and a destination (a = b *op* c), with a single argument for a unary operator and a destination (a = *op* b) and simple copy assignments (a = b).
- Unconditional jumps (`goto` *label*).
- Conditional jumps (`if` a *relop* b `goto` *label*).
- Array access (a = x[i] and x[i] = a).
- Function call (`call` *function*, and arguments are passed by preceding the call with an appropriate number of `param` instructions of the form `param` *arg*).
- Function return (`return`).
- Input and output (`read` a and `write` a).

A three-address code for a more comprehensive source language would need further instructions including a set for pointer/reference management. The instructions available must cover all the operations required for all the data types supported by the source language.

The very simple example shown in Fig. 6.6 illustrates the style of this IR. The source program is on the left and the IR generated for the four assignment operations is shown on the right.

The IR that this front-end generates uses the convention that the program variables are called `vgn` or `vln` where n is an integer value. Here, `a` is in `vg0`, `b` in `vg1`, `c` in `vg2` and `d` in `vg3`. The registers named `rn` are used for temporary values. It is easy to see the correspondence between the two pieces of code. Some of the characteristics of the generated code are examined in the next section.

Fig. 6.6 Translation of DL to IR

```
int a,b,c,d;
{
  a=1;
  b=2;
  c=3;
  d=a*b*c + b*3
}
```

```
vg0 = 1
vg1 = 2
vg2 = 3
r3 = vg0 * vg1
r2 = r3 * vg2
r4 = vg1 * 3
r1 = r2 + r4
vg3 = r1
```

6.5.2 Translation to the IR

There are some important design decisions to be made before tackling the coding of a translator from a tree-based representation to a three-address code. It becomes clear very quickly that this phase of compilation is different from earlier phases because there is no predefined "correct" output. For example, looking at the IR in Fig. 6.6, there are infinitely many semantically correct translations that could be produced from the abstract syntax tree. These different translations will have different costs in terms of numbers of IR instructions. We can make some overall design proposals and then examine the consequences.

- There is no pressing need to optimise in this phase. It is better to keep the translation simple even though that may mean the generation of verbose code. But it does no harm to implement some simple optimisations.
- The generated code can be profligate with its use of temporary variables. Whenever a new temporary variable is needed, it is perfectly acceptable to use a new variable name rather than keeping track of variable names already used and determining when they are no longer required. The code generator can assign these temporary variables to target machine registers or main storage locations, as appropriate.
- It is not wrong to deal with each tree node independently. It is not essential to examine the parent/child context of each node to generate better code. Optimisations based on context can be dealt with later. This reduces the complexity of this phase enormously.
- There is nothing wrong in making the IR readable, making use of symbolic names for labels and functions rather than absolute or relative addresses, aiming for an assembler-like language rather than a machine code-like language. However, one should ideally avoid references to the symbol table so that access to the symbol table does not need to be carried forward into the compiler's back-end.
- It is sensible to try to maintain target machine independence. In the case of this compiler for DL, this is probably not too much of a problem because there is essentially only one data type being used.

It is clear that generating IR directly from the tree will result in code that is not optimal. For example, the code produced in the example in Fig. 6.6 contains some redundancy both in terms of instructions as well as in terms of registers used. Generating highly optimised code directly from a tree is difficult. It is much better to

generate code that can be optimised later. These optimisation algorithms are described in Chap. 7. In this example, code is being generated for each statement independently and when statements are juxtaposed, the code starts looking rather poor.

The formatting of the IR is of some minor significance. Ideally, to save writing a parser for the IR as the front-end of the next phase of compilation, it makes sense to generate the IR in a low-level binary format which can be read directly by the optimiser or target code generator. But producing a human-readable version as well may be very helpful, particularly when debugging the compiler.

The next step is to look at some of the details of the code to translate from tree nodes to the IR. A recursive tree-traversal function `cg` is defined, having an overall structure very similar to that of `printtree`, the tree printing function. It is based on a large `switch` statement, each `case` dealing with a particular node type. For many of the abstract syntax tree node types the actions are straightforward, but there are some constructs needing additional explanation.

6.5.2.1 The IR Generating Function `cg`

Consider the problem of generating code for an arbitrary node (Fig. 6.7) containing an operator and left and right subtrees providing two operands for the operator.

The code generator function is called with an argument (p) pointing to the operator node. The function calls itself recursively with the left pointer as argument and then with the right pointer as argument (or vice versa as the subtrees are independent), and *then* code can be generated for the operator, referring to registers used to hold the results from the left and right subtrees. Specifying this a little more carefully, `cg` should return either

- a temporary register (e.g. `r5`), or
- a register associated with a variable (e.g. `vg5` or `vl2`), or
- a constant value (e.g. `5`), or
- nothing (because some constructs return no value at all).

Therefore `cg` should return a `struct a3token` defined as:

```
typedef enum {
  R_REG, R_GLOBAL, R_LOCAL, R_CONST, R_NONE} regtags;
              /* r5, vg5, vl5, 5, nothing returned */

/* Structure returned by the cg function */
typedef struct regstruct {
```

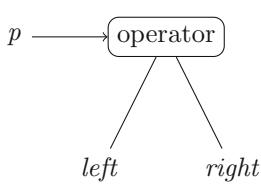


Fig. 6.7 A generalised tree node

```
    regtags regtype;            /*  R_REG, R_GLOBAL, R_LOCAL, R_CONST,
                                    R_NONE */
    int regvalue;               /* which register or integer value of
                                   constant */
} a3token;
```

Examples of code to handle particular DL constructs using this approach are presented below.

There are of course other ways in which the code generator can be structured but this general approach is simple and effective. It also generates fairly good intermediate code. Although it has already been said that there is no need to optimise at this stage, compiler writers inevitably like to show off their optimisation skills, even in this phase!

Considering the code for `cg` itself, a `switch` statement can be used with a `case` for each node type. This is very similar to the code used for outputting the tree.

```
a3token cg(astptr p)
{
  int nodetype;
  a3token noresult = {R_NONE, -1};
  astptr left,right;

  if (p==NULL) return noresult;

  nodetype = p->asttype;

  switch (nodetype) {
    .
    .
    .
  }
}
```

6.5.2.2 Variables

Before looking at the details of the translation of specific statements, the handling of variables should be examined. The use of a stack for variable storage at runtime, as described in Sect. 6.2.3, is appropriate for DL. The global variables are those declared at the top level and they are all stored in the lowest frame on the stack. At runtime, a pointer (called `gp` — the global pointer) is set up to point to the start of this stack frame. The IR variables `vg0`, `vg1`, `vg2`, … are stored here so that at runtime, the address of, for example, `vg23` is found by adding 23 to the contents of register `gp`.

When functions are involved, the situation becomes a little more complex. The code within a function can access both the *local* variables declared within that function, together with the argument variables. The code can also access all the *global*

variables already defined so far in the program. Accessing the global variables is as already described via the `gp` register. Local variables are stored further up the stack. These variables are called `vl0`, `vl1`, `vl2`, … and they are accessed via offsets 0, 1, 2, … from the contents of register `sp`. Because `sp` always points to the base of the *current* stack frame this plan works even if the function calls itself recursively.

This is why there are two distinct forms of variable in the IR — the global and the local variables. The `vg`*n* variables are accessed via `gp` and the `vl`*n* variables are accessed via `sp`.

6.5.2.3 Expressions

The handling of arithmetic expressions provides a good example of how three-address code can be generated. A tree of the form shown in Fig. 6.7 can be used to represent an expression with the node containing an arithmetic operator. The general approach to code generation presented in the last section can then be adapted to generate code for arithmetic expressions. Given the expression `1 * 2 + 3 + 4` this algorithm will produce three-address code of the form:

```
r3 = 1 * 2
r2 = r3 + 3
r1 = r2 + 4
```

leaving the result in `r1`.

The code in `cg` generating this output for nodes containing an operator is as follows:

```
case N_PLUS:
case N_MINUS:
case N_MUL:
case N_DIV:

  left=p->p1;
  right=p->p2;
  t=genreg();

  t2=cg(left);
  t3=cg(right);

  opreg(t); printf("="); opreg(t2); outop(nodetype); opreg(t3);
  printf("\n");

  return t;
```

The `genreg()` function returns the identity (as an `a3token`) of the next unused temporary register. Code is generated for the left and right subtrees of the node and the arithmetic instruction for the node is output using the `outop` function, outputting `+`, `-`, `*` or `/` according to its argument. The `opreg` function outputs an argument for the machine instruction, either a temporary register, a register storing a variable or

an integer constant. Returning `t` at the end of the code indicates to the caller that the result has been placed in register `t`. Further optimisations are of course possible. For example, if both subtrees refer to constant values, the computation specified by the node can be done immediately and the computed value returned as a constant. However, optimisations like this may be better performed later.

This code is simple and it works well. It produces fairly high-quality code because the `cg` function returns a result that can be placed directly into an instruction being generated for the current node. An alternative approach is for `cg` to return its results in temporary registers so, for example, the code for `1 + 2` would take the form

```
r2 = 1
r3 = 2
r1 = r2 + r3
```

But it makes sense to code `cg` to make the generated code more compact. The `cg` function just has to return a little more information to its caller.

The handling of boolean expressions is not an issue for DL, but clearly similar techniques can be used to those for arithmetic expressions. As ever, the language definition is the guide for the evaluation rules. Consider the simple example of the evaluation of the expression `a & b`. Here the & operator is the boolean `and`. Evaluating this expression can be done by evaluating `a`, evaluating `b` and then computing `a & b` by code generating the appropriate target machine instruction. There is a potential alternative. Suppose `a` is evaluated. If it is found to be `false`, then `b` need not be evaluated because the value of the entire expression must be `false`. This *short-circuited evaluation* looks like a good optimisation. But a problem arises if the evaluation of `b` has side effects. The language rules have to be consulted to ensure that the correct form of evaluation is being used.

6.5.2.4 Conditional Statements

DL's conditional statements (`if` and `while`) are both based on a test using a `<bexpression>`, a relational operator separating two expressions. Consider first the `if` statement. There are two possible forms since the `else`-part is optional and they are defined in BNF as follows:

```
<ifstatement> ::= if ( <bexpression> ) <block> else <block>
                | if ( <bexpression> ) <block>
```

The tree node generated by the parser contains the tag `N_IF`, a pointer to the condition (`<bexpression>`), a pointer to the *then*-block and a pointer to the *else*-block (`NULL` if not present). An obvious way of generating code for this construct is to first generate code for the `<bexpression>` with the assumption that this code leaves a value in a register indicating whether the value of the condition evaluated as true or false (say, 1 or 0 respectively). Then a conditional jump can be compiled

to jump to the *else*-block or the end of the statement if the condition evaluated as false. The *then*-block is then code generated, followed by a jump, if necessary, to the end of the code for the *else*-block. This will work but relies on a messy double comparison, once in the code generated for `<bexpression>` and once in the code checking the condition.

A better approach is for the code generation of the `if` statement to deal also with the code generation of the `<bexpression>`. This is shown in the code example below where only one test is needed in each `if` statement.

Consider then the code generation of a statement starting `if (a>0)` with a *then*-block but no *else*-block. The code should have the form:

```
if (a <= 0) goto l1
code for then block
l1:
```

and if there is an *else*-block the code should have the form:

```
if (a <= 0) goto l1
code for then block
goto l2
l1:
code for else block
l2:
```

Note that the sense of the relational operator has to be *inverted* in this code so that, for example the operator > in the source code is translated to the operator <= in the IR. This is the role of the function `invcondition` in the code below. It examines the relational operator in the node passed as an argument and prints the inverse operator.

Generating intermediate code of this form is not difficult. For example:

```
case N_IF:
  l1=genlab();
  cnode=p->p1;                  /* point to the cond node */
  t2=cg(cnode->p1); t3=cg(cnode->p2);
  printf("if ("); opreg(t2); invcondition(cnode); opreg(t3);
  printf(") goto l%d\n",l1);    /* jump to else part or
                                   end of statement*/
  t=cg(p->p2);                  /* cg then part */
  if (p->p3 != NULL) {          /* else present */
    l2=genlab();
    printf("goto l%d\n",l2);
    printf("l%d:\n",l1);
    t=cg(p->p3);                /* cg else part */
    printf("l%d:\n",l2);
  }
  else                          /* no else present */
    printf("l%d:\n",l1);

  return noresult;
```

An example of the IR generated by this code from the DL source code `if (a+b > a*b) c = 1 else c = 2` is:

```
r1 = vg0 + vg1
r2 = vg0 * vg1
if (r1<=r2) goto l1
vg2 = 1
goto l2
l1:
vg2 = 2
l2:
```

DL's `while` statement is treated in a similar manner. For example, the IR generated from a DL statement starting `while (a>0)` would have the form

```
l1:
if (a <= 0) goto l2
code for while block
goto l1
l2:
```

The code to generate this IR takes the form:

```
case N_WHILE:
  l1=genlab();
  l2=genlab();
  cnode=p->p1;                          /* point to the cond node */
  t2=cg(cnode->p1); t3=cg(cnode->p2);
  printf("l%d:\nif (",l1); opreg(t2);
  invcondition(cnode); opreg(t3);
  printf(") goto l%d\n",l2);            /* jump out of while */
  t=cg(p->p2);                          /* cg do part */
  printf("goto l%d\n",l1);
  printf("l%d:\n",l2);

  return noresult;
```

and the IR this generates from the DL source statement `while (i<=10) i = i + 1` is:

```
l1:
if (vg0>10) goto l2
r1 = vg0 + 1
vg0 = r1
goto l1
l2:
```

Although not relevant to DL, the implementation of the `switch` construct should be mentioned. There are many ways in which this can be tackled. A simple approach is to regard the switch statement as an `if` … `then` … `else if` … construct. For switch statements with many cases, this may not result in high performance code, but the logic is simple. The range and number of case values are important in deciding

on a more efficient implementation strategy. It may be that a jump table can be used, or even some sort of binary search algorithm.

6.5.2.5 Functions

The parser constructs a linked list of pointers to all the functions declared in the DL program. This is a distinct structure, independent of the abstract syntax tree generated for the main program. There is a node for each function in this separate abstract syntax tree and in this node is an identification of the symbol table entry for the function together with a pointer to the tree for the block defining the code for the function. The number of arguments is placed in the symbol table so that when the function is called, the number of arguments provided in the call can be compared with the number of arguments in the definition. They should of course be equal.

Generating code for each function is not difficult. The code needs to start with an identification of the function name, and this is followed by the code for the function itself, generated in the normal way by the `cg` function. However there are two further issues, one for the beginning of the function and one for the end.

Associated with every function and the main program too is the size of its stack frame. This is the total size of all the variables declared within the function (or the main program — the global variables). If the function or the main program calls another function, knowledge of this size value is essential because the stack pointer (`sp`) has to be incremented by this value to point to the space available for the local variables of the function being called. This size value is normally stored in the symbol table, but it would be convenient to have the value available at the start of the IR code for the function. In the case of the DL compiler, the code for the function is preceded by a line starting with a colon character followed by the name of the function being defined, followed by the size of the local variables for that function enclosed in parentheses. Examples of this can be seen in Sect. 6.5.3.

The other issue is concerned with the return of results to the caller of the function. The syntax of DL shows the `return <expression>` construct and this is translated into a `return` IR instruction. But how can a value be returned to the caller? A convenient way of managing this is to reserve a temporary register for the entire compilation and this register is always assumed to contain the result from the last function call. For example, the function `return1` and its corresponding translation is:

```
return1();                  :return1(0)
{                           r0 = 99
   return 99                return
}
```

Here it can be seen that the preallocated register for return values is `r0`. The caller of `return1` can then save the value of `r0` if necessary.

But what happens with a DL function that finishes without having executed a `return` instruction? Is this an error? The specification of DL should of course say

what should happen (but it doesn't!). So we can take an easy way out at this stage by assuming that all functions should end with the code

```
r0 = 0
return
```

so that a value of zero is returned if control falls through the end of a function without having executed an explicit `return` instruction.

Function call is easily managed. The code to generate the arguments, if any, for the call is generated, with the arguments going into `param` instructions. The call is made, a new temporary register is allocated and the value in the preallocated result register is placed in this new register. The code in `cg` to do this is:

```
case N_FNCALL:
  t=cg(p->p1);       /* deal with arguments */
  printf("call %s\n",symtable[p->astdata1].symname);
  t=genreg();         /* may need a new register for the result */
  opreg(t); printf("="); opreg(returnreg); printf("\n");
  return t;
```

In this code, the `p1` field of the structure pointed to by `p` is the list of arguments to the function and the `astdata1` field is the index of the symbol table entry defining this function. To help show how this code works, consider this example and the corresponding IR code:

```
int x;                          param 3
{                               call f
   x = f(3) + f(4);             r5 = r0
}                               param 4
                                call f
                                r6 = r0
                                r4 = r5 + r6
                                vg0 = r4
```

6.5.3 An Example

Here is some DL code showing a few of the features that have been discussed in this chapter. It shows how functions are represented and called, how results are returned from functions (in `r0`) and how the size of the stack required for local data storage in each function and in the main program is stated in brackets after the function name. Note that the main program is given the name `!MAIN!` — an arbitrary name that is distinct from any DL function (a function name cannot include exclamation marks).

```
int x;


f1();
```

```
int x1,x[10],y;
{
   x1=100;
   return x1
}

int zzz;
int zzz1[100];

f2(n);
int x1,a,b,c;
{
   x1=200*n + f1();
   return x1
}



{
   x = f1() * f2();
   print(x)
}
```

The IR generated by the compiler is:

```
:f1(12)
vl0 = 100
r0 = vl0
return
r0 = 0
return

:f2(5)
r2 = 200 * vl0
call f1
r3 = r0
r1 = r2 + r3
vl1 = r1
r0 = vl1
return
r0 = 0
return

:!MAIN!(102)
call f1
r5 = r0
```

```
call f2
r6 = r0
r4 = r5 * r6
vg0 = r4
print vg0
r0 = 0
return
```

Another example appears in the appendix.

6.6 Conclusions and Further Reading

DL has extremely limited support for making use of type information. It is certainly worth looking at how type information can be stored and managed in compilers for more powerful languages. For example [8, 9] show code for C compilers and therefore cover the issues of user-defined types and type equivalence. There is also a large published literature on type systems and a relevant example is [10]. Type checking is covered well in [11]. It is instructive to find out how typing is used in a variety of programming languages.

A great deal has also been written in the field of comparative programming languages and reading in this area gives a good insight into the huge range of programming language features needing implementation. For example, Java is an important milestone in the development of general purpose (and object-oriented) languages and in the context of this chapter, the Java Virtual Machine (JVM) [12] has particular significance.

This book concentrates on traditional imperative programming languages and the worlds of object-oriented, logic programming, functional and other languages are largely ignored. Many of the techniques described in this book have general applicability, but more specialised approaches are required to deal with specific language features. There is a good introduction in [13].

The design of intermediate representations for imperative languages is covered in [7]. Intermediate representations for functional languages are discussed in [14, 15]. The LLVM compiler infrastructure project is documented in detail at http://llvm.org and there is much there on the design of intermediate languages.

The syntax-directed translation approach to the generation of code is covered in [1] and much more detail about attribute grammars is in [3]. Most programming languages need to implement sophisticated storage management techniques not just to provide storage space for variables but to provide flexible storage for objects, dynamic heap-based data structures and so on. Storage management using garbage collection, reference counting, etc. may well be necessary. A good summary appears in [1] and these issues are dealt with more generally in many of the data structures and algorithms textbooks.

DL provides only a very simple form of boolean expression support. When evaluating more complex boolean expressions, it may be possible to implement some significant optimisations. These techniques of short-circuit evaluation are covered in [3].

The generation of high-quality code for the switch statement has been studied, for example in [16, 17].

Exercises

6.1. The date manipulation language introduced in Sect. 6.1.2.2 is worth developing. Produce a specification for a simple but full programming language for the manipulation of dates and code an implementation. What other features could reasonably be added to a language like this?

6.2. Some programming languages (for example, Pascal) allow the nesting of function definitions. How valuable is this feature? Can it be added easily to DL? If so, plan an implementation by starting with the changes necessary for the symbol table.

6.3. What are your views on strong type checking? Does it get in the way?

6.4. Investigate some IRs used in production compilers. For example, look at LLVM's IR and the Java Virtual Machine.

6.5. It should be straightforward to implement a constant folding optimisation during the traversal of the abstract syntax tree to generate the IR. Or is this easier to do on the three-address code once it has been generated?

6.6. Many programming languages (C included) support a `switch` statement. Devise an abstract syntax tree structure to support this statement and consider its implementation by showing how intermediate code can be generated. Are there extensions to the three-address code proposed in this chapter which would be appropriate? Where would you check for the existence of duplicate case labels? It may be helpful to consult the discussion of `switch` statements in [1].

6.7. Design a three-address code for C or for any other language you know. How general purpose can you make a three-address code?

6.8. Write an interpreter for a three-address code. Parsing the statements could be done easily with *flex*/*bison*-style tools.

6.9. Try using C (or any other high-level language) as an intermediate representation for DL.

6.10. Is it easy to determine whether a value is being returned by a DL function? How would you implement an acceptable checking algorithm?

6.11. The material in this chapter has shown that the syntax specification of DL is telling just part of the language definition story. Some of the issues have already been highlighted. There are more.

6.12. What would be the semantic analyser consequences of introducing a new data type (say, `float`) to DL?

References

1. Aho AV, Lam MS, Sethi R, Ullman JD (2007) Compilers – principles, techniques and tools, 2nd edn. Pearson Education, Upper Saddle River
2. Cytron R, Ferrante J, Rosen BK, Wegman MN, Kenneth Zadek F (1991) Efficiently computing static single assignment form and the control dependence graph. ACM Trans Program Lang Syst 13(4):451–490
3. Cooper KD, Torczon L (2011) Engineering a compiler, 2nd edn. Morgan Kaufmann, San Francisco
4. Bilardi G, Pingali K (2003) Algorithms for computing the static single assignment form. J ACM 50(3):375–425
5. Allen FE (1970) Control flow analysis. ACM SIGPLAN Notices 5(7):1–19
6. Ferrante J, Ottenstein KJ, Warren JD (1987) The program dependence graph and its use in optimization. ACM Trans Program Lang Syst 9(3):319–349
7. Stanier J, Watson D (2013) Intermediate representations in imperative compilers: a survey. ACM Comput Surv 45(3):26:1–26:27
8. Holub AI (1994) Compiler design in C, 2nd edn. Prentice Hall International, New York
9. Fraser C, Hanson D (1995) A retargetable C compiler: design and implementation. Addison-Wesley, Reading
10. Pierce BC (2002) Types and programming languages. The MIT Press, Cambridge
11. Mogensen TÆ (2011) Introduction to compiler design. Undergraduate topics in computer science. Springer, Berlin
12. Lindholm T, Yellin F (1997) The Java virtual machine specification. The Java series. Addison-Wesley, Reading
13. Grune D, Bal HE, Jacobs CJH, Langendoen KG (2000) Modern compiler design. Wiley, New York
14. Appel AW (1992) Compiling with continuations. Cambridge University Press, Cambridge
15. Peyton Jones SL (1987) The implementation of functional programming languages. Prentice Hall international series in computer science. Prentice Hall, Englewood Cliffs
16. Bernstein RL (1985) Producing good code for the case statement. Softw Pract Exp 15(10):1021–1024
17. Hennessy JL, Mendelsohn N (1982) Compilation of the Pascal case statement. Softw Pract Exp 12(9):879–882

Chapter 7
Optimisation

The generation of high-quality code is a key objective in the development of compilers. It is of course true that a usable compiler can be written with little or no provision for optimisation, but the performance of the generated code may be disappointing. Today's production compilers can generate code of outstanding quality, normally much better than even hand-written target assembly code produced by an expert. However, producing this high-quality code using a compiler is not at all easy. The optimisation algorithms used are often complex and costly to run, they interact with each other in unpredictable ways and the choice of which techniques to use for best results will be influenced by the precise structure of the code being optimised. There is no easy solution and an approach to optimisation has to be adopted which works well with the "average" program.

Is optimisation important? How hard should a compiler work to produce highly optimised code? The answer depends on the nature of the program being compiled and on the needs of the programmer. For many programs, optimisation is an irrelevance but performing it does no real harm. For other programs resource constraints may mean that optimisation is essential. An argument often heard against the need for optimisation, both by the programmer and by the compiler, is that by waiting for processors to get fast enough the problem will disappear. As the growth in processor speeds seems to be slowing, the wait for sufficiently fast processors may be longer than expected. This makes it increasingly important for both the programmer and the compiler writer to be aware of the importance of optimisation.

Before examining practical techniques, it is important to remember that the term *optimisation* when used in the context of compiler design does not refer to the search for the "best" code. Instead it is concerned with the generation of "better" code. We will return to this issue later in Chap. 8 when covering the topic of superoptimisation. Another related issue is the choice of criteria for code improvement. Are we looking for fast code, small code (in terms of total memory footprint, code size, data usage or some other criterion), code that generates the least i/o traffic, code that minimises power consumption, or what? Some optimisations trade speed improvements against code size and so it is essential to know the real optimisation goal. In most of the techniques described in this chapter, the optimisation goal is to reduce the

instruction count. This is a reasonable target for intermediate code because it has the effect of reducing code size and almost always reducing execution time. If specific aims for optimisation are required in a particular compiler then each optimisation applied should be examined in detail, both in isolation and when combined with other optimisations, to determine its effect on the optimisation goal.

7.1 Approaches to Optimisation

Section 2.4 showed where code optimisation can be placed amongst the modules of a compiler. This chapter concentrates on target machine-independent optimisation, operating on the intermediate representation generated by the semantic analyser. Chapter 8 will cover the optimisation during and after the code generation phase. This is machine-dependent optimisation. Machine-independent optimisation can be very effective and there are many ways in which code improvements may be made at this stage of compilation. It is helpful to think about this optimisation as a collection of techniques applied in turn to the intermediate representation, in effect a set of IR filters. There is no necessity for the same intermediate representation to be used for both the input and the output of this optimisation phase, but in this chapter it is assumed that the IRs are the same. This raises an important issue of how these optimisation filters may be combined. It would be really convenient if the optimisation process could be achieved by combining as many filters as are available or necessary, in an arbitrary order. But these optimisation processes interact with each other in complex ways and they may even need repeating. Therefore, optimising the ordering of optimisation filters can be an surprisingly complex task.

7.1.1 Design Principles

In today's production compilers, a large proportion of the compiler's code is concerned directly with optimisation and most of the research currently being undertaken in the field of compiler design is now concentrated on optimisation-related issues. But there are still limits to how far optimisation can go. For example, making significant changes to the algorithms in the source program is not generally appropriate. For example, it is probably not the role of the compiler to recognise a source code sequence that has been written to invert a matrix and replace it by code that uses a faster algorithm. But it probably is appropriate for the compiler to evaluate certain source code sequences at compile time and use the results in the generated code. This, of course, can only be done when there is no doubt that the evaluation process will terminate. Completely running the whole program at compile time is rarely possible and indeed impossible if the program is expecting input during execution.

The optimisation process should not alter the program's semantics. Optimisations must be *safe* in that the results from running a program with and without optimisation

should be identical. When developing a compiler it is easy to make mistakes concerned with safety. For example it is possible to make the false assumption that the code of a loop is always executed at least once. It should be mentioned here that the issues concerned with the preservation of program semantics may turn out to be complex and unclear. Many compilers have been developed that openly make changes to the semantics of a program on optimisation. These changes are documented so hopefully the programmer is aware of them and they usually occur in the handling of numerical calculation errors—maybe overflow is not detected when optimisation is turned on. Whether or not this type of optimisation is acceptable depends on the nature and behaviour of the program being compiled.

The optimisations should also be *worthwhile* in that they must produce a noticeable benefit to whatever it is that is being optimised. For example, a loop unrolling optimisation might enlarge the executable code so much that the loop no longer fits in the processor cache, resulting in the code becoming slower. Some optimisations can be very expensive to perform and therefore may not be worthwhile if, for example the target program is only going to be run once.

These goals to ensure that optimisations are both safe and worthwhile may sound obvious, but achieving them is far from automatic. Giving the user of the compiler the ability to choose whether to include such optimisations is wise.

The compiler tries to predict what is going to happen when the target machine program actually runs and the compiler has to use this information in its optimisation processes. For example, it is sensible for the optimisation to be concentrated on code that is expected to run many times. By ensuring that code within loops is as efficient as possible, a significant improvement can be made to overall program performance. An interesting approach is to compile the source program and then start running the generated code, allowing feedback about performance to pass from the running program back to the compiler. The compiler then makes corresponding modifications to the code (for example, by performing further optimisation on blocks of code that seem to be executing frequently), the executing code is updated and continues in execution. This is performed repeatedly. Here, the compiler can make optimisation decisions based on real dynamic data rather on static speculation based on the source program.

While developing optimising transformations within the compiler, it is important to remember how the intermediate code is transformed during code generation, machine-dependent optimisation and while actually running on the target machine. Do not implement optimisations in the machine-independent stage that can be better done during code generation. Furthermore, modern processors usually include powerful features to perform runtime optimisation. For example, some processors support *out-of-order execution* where if an executing instruction stalls because results generated by earlier instructions are not yet available, the processor can automatically start executing other instructions that are ready. Any necessary adjustment to generated results because of the changed order of instruction execution is also performed automatically by the hardware. Not everything has to be done in the compiler.

What sort of optimisations can be done on the intermediate representation at this stage of compilation? Optimisations that concentrate on the removal of code that is

unreachable, performs no useful function or is somehow redundant are particularly important. Common subexpression elimination can be done here. It may be possible to move code to parts of the program that are executed less frequently. For example, loop invariant code can be moved out of a loop. Operations on variables whose values are known at compile time can be performed at this stage. Constants can be folded and values propagated. If the target machine offers any form of parallel processing it may also be possible to use dependence analysis to determine which code fragments are independent and therefore candidates for execution in parallel.

Code optimisation is *difficult*. Coding an optimising compiler should not be undertaken lightly. But today's compilers can produce impressive code because the complexity of the interactions of operations specified by the source program and the complexity of the characteristics of the target machine can be managed effectively by clever algorithms and the computational power used by the compiler.

A few of the many available optimisation techniques are presented below. Techniques of local optimisation concentrate on basic blocks. These are comparatively small blocks of code where the analysis is undemanding. With global optimisation the issues of potentially complex flow of control have to be taken into account and the complexity of analysis for optimisation greatly increases.

7.2 Local Optimisation and Basic Blocks

The stream of intermediate code generated by the front-end of the compiler may well have a complex control structure with conditional and unconditional branches, function calls and returns scattered through the code. However, there will be sections of code not containing any of these transfers of control—"straight-line code"—where control starts at the first instruction and passes through all the instructions in the section. These sections of code are called *basic blocks*.

Partitioning the intermediate code into a set of basic blocks is not difficult. Control enters a basic block via the first instruction of the block. Therefore this first instruction must either be the destination of some sort of branch instruction, the instruction following a conditional branch or the first instruction of a function or the main program. The last instruction of the basic block must be a conditional or unconditional branch instruction, a `return` instruction or the last instruction of the main program. Section 6.4.2.2 shows how the basic block can be used as an integral part of an intermediate representation and Fig. 6.5 gives an example of a basic block representation of a simple program.

Another example comes from the main program of the factorial program in the appendix and is shown in Fig. 7.1.

Producing this control flow graph is often the first step in the intermediate representation optimisation process. The next step is to examine each basic block individually and perform local optimisation. This local optimisation is completely independent of the flow of control between basic blocks.

7.2.1 Constant Folding and Constant Propagation

There are many simple optimisations that can be done when the values of operands in the IR instructions are known or can be calculated. Consider the DL code:

```
a = 2;
b = 3 * 4;
c = a + b;
```

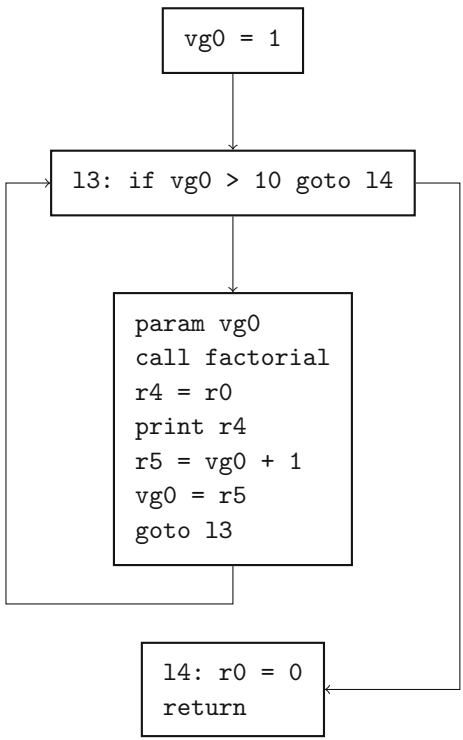


Fig. 7.1 Basic blocks of factorial main program (see appendix)

producing the code (taken as a basic block):

```
vg0 = 2
r1 = 3 * 4
vg1 = r1
r2 = vg0 + vg1
vg2 = r2
```

A good first step is to replace all operators with explicit constant arguments by the result of the computation. Here, `3 * 4` can be replaced by `12`. The computation is

being done at compile time rather than at runtime. But there is more that can be done by propagating known values. Stepping through the basic block code instruction by instruction, right-hand sides can be replaced by constant values where they are known. We then obtain the code:

```
vg0 = 2
r1 = 12
vg1 = 12
r2 = 14
vg2 = 14
```

The two arithmetic operators have been optimised away. Clearly there is more that can be done, and this code will be improved a little more in the examples below.

This type of analysis can be taken further. By replacing expressions in the code having constant operands with their corresponding values the program is essentially being run (or partially run) at compile time. Is it feasible to actually run the program at compile time, assuming it needs no user input, and then generate a program that outputs the results? For some simple programs this will happen as a consequence of conventional static analysis, but in a more general case this extreme attempt at optimisation is not wise. With more complex control flow the analysis becomes really difficult and it has to be known in advance whether the program will terminate. This is not possible in general.

7.2.2 Common Subexpressions

Common subexpression elimination is a well-known optimisation and it is a technique that has been used by programmers as well as by compilers for many years. Some programmers, when faced with code of the form

`x = (a*a-1)/(a*a+1)`

will automatically rewrite it as

`t = a*a; x = (t-1)/(t+1)`

even when subconsciously aware that the compiler will be doing that optimisation automatically.

Performing this type of optimisation within a basic block can be done by sequentially passing through the basic block, keeping a record of the values calculated so far. For example, suppose the source code contains two assignments to `x` and `y`.

```
x = a/(b*b + c*c);
y = b/(b*b + c*c + a)
```

The corresponding IR together with the IR rewritten so that the vg variables are replaced by their source program names to make the example a little clearer is shown here.

```
r3 = vg3 * vg3          r3 = b * b
r4 = vg4 * vg4          r4 = c * c
r2 = r3 + r4            r2 = r3 + r4
r1 = vg2 / r2           r1 = a / r2
vg0 = r1                x = r1
r8 = vg3 * vg3          r8 = b * b
r9 = vg4 * vg4          r9 = c * c
r7 = r8 + r9            r7 = r8 + r9
r6 = r7 + vg2           r6 = r7 + a
r5 = vg3 / r6           r5 = b / r6
vg1 = r5                y = r5
```

The algorithm to find common subexpressions scans through the IR, maintaining a list of computed expressions and where their results are stored. When an instruction is found with a computation which has already been performed on its right-hand side, it is replaced by an assignment to the variable/register already containing the value. Entries in the list can be invalidated by encountering an instruction that updates the value in a variable used in the expression. For example, the IR instructions above can be scanned one after the other:

```
r3 = b * b      b * b is now stored in r3
r4 = c * c      c * c is now stored in r4
r2 = r3 + r4    r3 + r4 is now stored in r2
r1 = a / r2     a / r2 is now stored in r1
x = r1
r8 = b * b      b * b is already stored in r3 and so this instruction
                can be replaced by r8 = r3
r9 = c * c      c * c is already stored in r4 and so this instruction
                can be replaced by r9 = r4
r7 = r8 + r9    r8 + r9 is now stored in r7
r6 = r7 + a     r7 + a is now stored in r6
r5 = b / r6     b / r6 is now stored in r5
y = r5
```

Making these two replacements and scanning through once again we obtain:

```
r3 = b * b      b * b is now stored in r3
r4 = c * c      c * c is now stored in r4
r2 = r3 + r4    r3 + r4 is now stored in r2
r1 = a / r2     a / r2 is now stored in r1
x = r1
r8 = r3
r9 = r4
r7 = r8 + r9    r8 + r9 is now stored in r7
r6 = r7 + a     r7 + a is now stored in r6
r5 = b / r6     b / r6 is now stored in r5
y = r5
```

Redundant registers can now be removed. If an instruction of the form `r1 = r2` is found then subsequent uses of `r1` can be replaced by `r2` and the assignment `r1 = r2` can be removed. We therefore obtain:

```
r3 = b * b
r4 = c * c
r2 = r3 + r4
r1 = a / r2
x = r1
r7 = r3 + r4      r8 replaced by r3 because of r8 = r3 and r9 replaced
                  by r4 because of r9 = r4
r6 = r7 + a
r5 = b / r6
y = r5
```

Then again we can scan for common expressions on the right-hand sides:

```
r3 = b * b        b * b is now stored in r3
r4 = c * c        c * c is now stored in r4
r2 = r3 + r4      r3 + r4 is now stored in r2
r1 = a / r2       a / r2 is now stored in r1
x = r1
r7 = r3 + r4      r3 + r4 is already stored in r2 and so this instruc
                  tion can be replaced by r7 = r2
r6 = r7 + a       r7 + a is now stored in r6
r5 = b / r6       b / r6 is now stored in r5
y = r5
```

Making the indicated replacement, removing the redundant `r7` (replacing it by `r2`) and scanning again, we obtain:

```
r3 = b * b        b * b is now stored in r3
r4 = c * c        c * c is now stored in r4
r2 = r3 + r4      r3 + r4 is now stored in r2
r1 = a / r2       a / r2 is now stored in r1
x = r1
r6 = r2 + a       r2 + a is now stored in r6
r5 = b / r6       b / r6 is now stored in r5
y = r5
```

There are no remaining common subexpressions. This marks the end of this optimisation. The eleven original instructions have been optimised to eight.

There is a little more to add to this process. The search for common subexpressions should take into account that the value of an expression of the form `a + b` is the same as `b + a`. Also, the expressions `a + c` and `a + b + c` *may* have a common subexpression but it depends on what the language definition says. If expressions of the form `a + b + c` are supposed to be evaluated from left to right, then there is no common subexpression.

Also, care has to be taken when deleting registers/variables. In the example above, the first register to be deleted was `r8`, being replaced by `r3`. Subsequent uses of `r8` will have already been replaced by `r3` in the basic block. But a check may have to be

made to see whether r8 is used in any of the basic blocks that can follow the current block. If so, the assignment to r8 has to stay or the code in the other basic blocks has to be altered to use r3 instead. This issue will be examined again in Sect. 7.3.1.

Common subexpressions can be killed by assigning to a variable used in the expression. For example, consider:

```
r1 = a * a
x = r1
a = 3
r2 = a * a
y = r2
```

After the first instruction, the list of already computed expressions will store the fact that the value of a * a is stored in r1. But encountering the third instruction will kill the a * a entry in the list (unless it can be guaranteed that the value of a is already 3 when this block of code starts). The value of a has changed and so the value of a * a is no longer stored in r1. There are no common subexpressions here, but the second a * a can be replaced by 9.

In languages that allow the access to variables via pointers, this problem of common subexpression analysis becomes more awkward. Fortunately DL does not suffer from this issue, but languages such as C do. Consider the code:

```
r1 = a * a
x = r1
*p = 3
r2 = a * a
y = r2
```

At first sight, it looks as though a * a is a common subexpression, but there is an aliasing problem. What is p pointing to? Could it possibly be pointing to a? Or can the compiler guarantee that it is not pointing to a? In general the analysis of pointers is notoriously difficult and it is likely that the default action here is to play safe and recalculate a * a. Similar considerations apply to function calls. A function call can update values in variables so that relying on variables keeping their values over a function call is not possible unless a full analysis of the effects of the function has been done.

7.2.2.1 Arrays

The intermediate code generated for the DL source statement a[i] = a[i]+ b[i] could have the form:

```
r2 = vg1[vg0]
r3 = vg11[vg0]
r1 = r2 + r3
vg1[vg0] = r1
```

Here, `vg0` is `i`, `vg1` is `a` and `vg11` is `b`. This code looks perfectly reasonable, but consider the target machine code generation of the array access operations. Suppose the target machine is byte-addressed and each element of `a` and `b` uses 4 bytes, making the assumption that integers in DL are represented using 4 bytes. A simple array reference then requires the computation of the index value, multiplication by 4 and adding that value to the start address of the array to obtain the address of the selected element. In the example above, the value `vg0` has to be multiplied by 4 on three separate occasions. This is a clear example of a common subexpression. But this common subexpression is hidden within the IR being used and a lower level IR would allow the optimisation to be achieved with ease. Such an optimisation could only be performed at this stage if the IR became target machine-dependent because the use of the factor 4 depends on the architecture of the target machine. There is no correct answer here and it is just important to ensure that this form of common subexpression elimination is performed somewhere, because considerable gains can be achieved by optimising code manipulating arrays.

7.2.3 Elimination of Redundant Code

Looking through examples of intermediate code generated by the DL front-end, it is obvious that some improvements can be made even after constant analysis and common subexpression elimination have been performed. For example, looking again at the code before constant analysis:

```
vg0 = 2
r1 = 3 * 4
vg1 = r1
r2 = vg0 + vg1
vg2 = r2
```

it is clear that `r1` and `r2` are redundant so the code could be improved to read:

```
vg0 = 2
vg1 = 3 * 4
vg2 = vg0 + vg1
```

and then after constant values are calculated and propagated:

```
vg0 = 2
vg1 = 12
vg2 = 14
```

Can the removal of redundant variables/registers be done after the constant analysis? Going back to the code we have already generated:

```
vg0 = 2
r1 = 12
vg1 = 12
r2 = 14
vg2 = 14
```

it is then obvious that if `r1` and `r2` are not read in subsequent instructions we can dispense with those two assignment instructions to `r1` and `r2`. Finding these *dead registers* or variables is important and gives the opportunity to remove redundant code.

Removing an unnecessary assignment to register `r` requires us to check whether `r` is ever read in another instruction reachable from the assignment. If there are no further reads from `r` or if `r` is written to before any further reads, then the assignment to `r` can be removed. Doing this wholly within the basic block is really easy because the flow of control is from the first instruction to the last instruction. But if `r` is referred to in *another* basic block, we have to determine whether the execution of this other reference can follow the original assignment. This is more complicated and the process is examined in Sect. 7.3.2.

There are other forms of redundant code which can be removed at this stage. For example, code of the form

```
x = 0;
if (x > 0) { ... }
```

can be simplified by removing the `if` statement completely, assuming that there are no jumps into the code following the `if`. In the case of DL there is no difficulty here, because DL does not support labels or goto statements.

If the target of a jump instruction is another jump instruction then the first jump can be changed to point to the destination of the second jump.

Any unlabelled instruction following an IR `goto` or `return` instruction can be removed because they can never be executed. The example IR in the appendix shows an example of this in the form of a `return` followed by a `goto`.

Various algebraic simplifications can also be made at this stage. For example, the expression `x + 0` can be replaced by `x`, `x * 1` by `x`, `x * 0` by `0` and so on.

7.3 Control and Data Flow

Section 7.2 showed how to take the intermediate representation of a program and divide it into basic blocks. A control flow graph can then be constructed (see Fig. 7.1) to show how control can pass from one basic block to another. This structure forms the basis for performing various non-local optimisations.

7.3.1 Non-local Optimisation

Once each basic block has been optimised, as described above, it is time to consider the flow of control between basic blocks. What does a basic block carry forward to its successor?

In performing local common subexpression elimination a list of expressions and where they are stored is maintained as the basic block is scanned. That list can be saved when the end of the basic block is reached and tagged with the identity of that basic block. The list can then be passed on to the next basic block in the execution path. This is part of the technique of data flow analysis. There are two possibilities, as shown in Fig. 7.2.

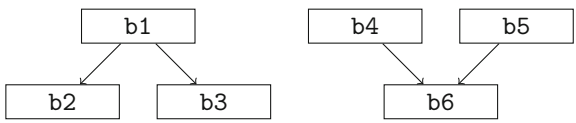
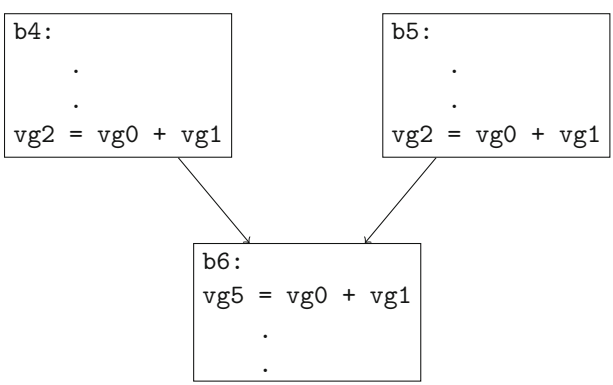


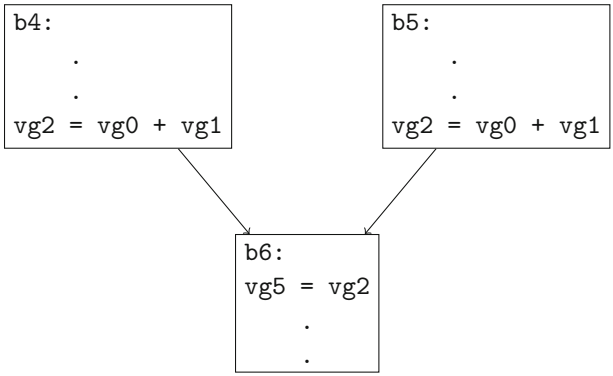
Fig. 7.2 Flow between basic blocks

The list of expressions and corresponding storage locations at the end of basic block `b1` is passed directly into the analysis of basic block `b2` so that `b2` can make use of any subexpressions generated by `b1`. The same applies to `b3`. In the case of blocks `b4` and `b5` the situation is a little more complicated because when analysing `b6` there is no knowledge of whether the entry to `b6` was via `b4` or `b5`. Therefore, the list of expressions and corresponding storage locations at the start of the analysis of `b6` has to be the *intersection* of the two sets, from `b4` and `b5`.

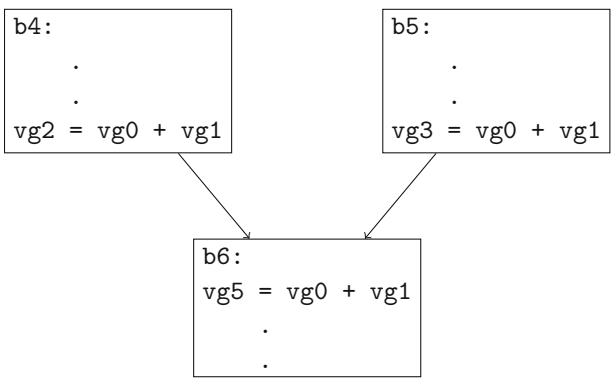
Suppose, for example, that the code in `b4`, `b5` and `b6` contains these instructions:



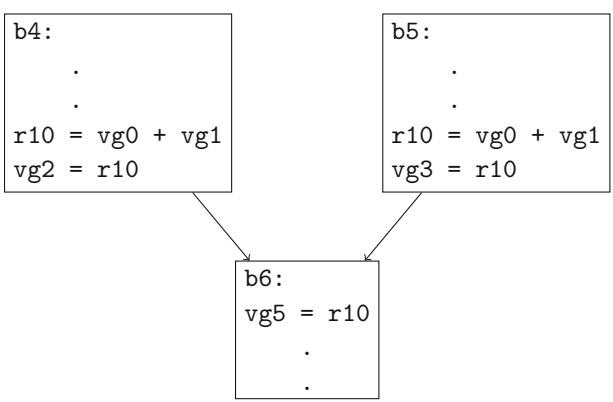
Then the common subexpression elimination can be performed.



But if the code is slightly different so that there is a mismatch of storage locations for the expression, as shown here:



Then `vg5` has to be set to *either* `vg2` *or* `vg3`. This effect can be achieved by introducing a new temporary register as follows:



7.3.2 Removing Redundant Variables

In these program transformations, the number of variables or registers being used tends to increase. It is important to make sure in this stage of optimisation that efforts are made to remove redundant variables. These are variables that are assigned a value and then are not used. In the last example in the previous section, block `b6` starts with an assignment (`vg5 = r10`). Whenever a simple assignment like this is encountered, the code following the assignment should be checked and all uses of the value of (in this case) `vg5` should be replaced by references to `r10`. If all such uses of `vg5` can be removed, then the original assignment is redundant and therefore it can be removed.

Inevitably there are some complications. In DL's IR, there are three different types of variable. The `vg` variables are tied to globally defined variables in the source language and the `r` variables are used for temporary values and other compiler-related purposes. These two sets of variables have global existence. They are present throughout the running of the program and there is no logical difference between the two sets. They can be treated identically. However, the `vl` variables are *local* to functions. They come into existence when the function is entered and disappear when the function returns. Therefore, at this stage of optimisation at the IR level, the `vl` variables can be handled just like all the others except that all analysis of these has to be restricted to the current function. Global analysis is required for `vg` and `r` variables.

This leads on to the concept of *live variable analysis*. Suppose the variable `x` is assigned the value 1 (`x` is "defined"), and then `x` is used later in the code:

```
x = 1;
...
a = x + 1;
...
```

The variable `x` is said to be live between its definition and its use. In the code:

```
x = 1;
...
a = x + 1;
...
x = 2;
...
```

the assignment `x = 2;` *kills* `x` and the live range ends here. An analysis of live ranges helps with the optimisation in that variable assignments that have no corresponding uses can be removed. This analysis is complicated by the fact that the definition and use do not have to be in the same basic block and so control flow between basic blocks has to be examined. If there is a path of execution between the definition and the use, then the definition has to stay. But in general, it is not possible to predict actual execution paths at compile time. For example:

```
x = 1;
...
if (p > q) a = x + 1;
...
```

It may be that `p` is never greater than `q` at runtime so that the assignment `a = x + 1` is never performed. But the compiler is unlikely to be able to make this assertion and has to play safe, keeping the definition of `x` and the conditional statement in the compiled code.

Data flow analysis is not easy and algorithms are influenced by the choice of intermediate representation. The use of static single assignment form (see Sect. 6.4.1.2) can simplify this analysis as can other data flow-oriented IRs. Various approaches to many forms of flow analysis are included in the references mentioned in the further reading section in this chapter.

7.3.3 Loop Optimisation

All the general-purpose optimisation techniques described so far can of course be applied to the code within loops, and if the optimisations are for execution speed, this is where to apply optimisation for most benefit. But there are some techniques particularly related to loop optimisation that can have significant effects on execution speed.

The first step is of course to identify the loops. This can be done using the control flow graph and it involves finding a basic block (the *loop header*) through which all paths to basic blocks in the loop pass and also finding a path from one of the loop's blocks back to the loop header (the *back edge*). The header block is said to *dominate* all the nodes in the loop. It should be possible to follow a control path from any of the nodes in the loop to any of the other nodes in the loop without passing through a node not in the loop.

Once the loops have been identified, each can be optimised in turn.

7.3.3.1 Removing Loop-Invariant Code

If a computation is performed in a loop such that the values used in the computation remain unchanged for each iteration of the loop, then the computation should be performed just once before the loop starts. The loop invariant code is hoisted out of the loop. For example in

```
i = 0;
while (i<n) {
   a[i] = a[i] + n*n;
   i = i + 1
}
```

the `n*n` computation is done each time the loop executes but it only needs to be done once because `n` is a loop invariant—it does not change as the loop is executed. The code then becomes:

```
i = 0;
t = n*n;
while (i<n) {
   a[i] = a[i] + t;
   i = i + 1
}
```

This proves to be a fairly simple optimisation to apply. Finding loop invariants is based on finding and then tagging loop invariant variables/registers and then the corresponding loop-invariant computations.

Note that the compiler is not performing the movement of loop-invariant code at the source code level. This transformation is being done on the IR. This means that it is possible to apply this type of optimisation even when it could not be done by modifying the source code. For example, there may be good opportunities for loop-invariant code motion when using multi-dimensional arrays in loops. Consider the C program fragment:

```
for (k=0; k<n; k++)
   a[i][j][k] = a[i][j][k] + 1;
```

Finding the offset from the start of the array of element `a[i][j][k]` does not need to be done from scratch on each iteration of the loop. The offset calculations involving the subscripts `i` and `j` can be done outside the loop. Just as in Sect. 7.2.2.1, the IR has to be designed so that this type of optimisation can be achieved.

Finally, although it is unlikely to be a problem with this type of optimisation, any movement of loop-invariant code must take into account the possibility that the loop is not executed at all.

7.3.3.2 Induction Variables

Loops often contain one or more variables that are incremented or decremented by a constant value each time control passes through the loop. In other words, these variables take values which are linear functions of the loop counter, containing the number of times the loop has been executed since it was last entered. Consequently a variable whose value is a linear function of induction variables must also be an induction variable. They keep in step with each other as the loop executes. It may then be possible to reduce the number of induction variables in a loop by replacing induction variables by linear functions of other induction variables.

7.3.3.3 Strength Reduction

It may be possible to save some processor cycles by replacing potentially costly operations on induction variables by cheaper operations. For example:

```
i = 0;
t = 0;
while (i<n) {
   t = i*3;
   a[i] = a[i] + t;
   i = i + 1
}
```

Here, `t` is an induction variable. The code can be rewritten:

```
i = 0;
t = 0;
while (i<n) {
   a[i] = a[i] + t;
   t = t + 3;
   i = i + 1
}
```

The multiplication operator has been replaced by a presumably cheaper addition operator. Clearly this optimisation is machine-dependent, but it is probably safe to assume that most common target architectures perform faster additions than multiplications.

7.3.3.4 Loop Unrolling

Consider the two pieces of code:

```
i = 0;                          a[0] = 0;
while (i<4) {                   a[1] = 0;
   a[i] = 0;                    a[2] = 0;
   i = i + 1                    a[3] = 0;
}
```

Many programmers would produce the code on the left (or something like it, probably using a `for` loop), but the code on the right could be more efficient, in speed and also possibly in memory usage. The code on the right is produced by *unrolling* the loop on the left and it obviously avoids the loop control variable `i` and the code that has to be generated at the start and at the end of the loop.

This form of loop unrolling is only applicable when the number of iterations is known at compile time. Loop unrolling should always reduce the total number of

operations performed by the target machine in executing the code. But if the number of copies of the body of the loop is large because the total number of iterations of the loop is large, then there may well be code space issues. There is a clear tradeoff here of code space against execution time and an appropriate compromise has to be reached.

As well as removing the loop control code, loop unrolling may have further benefits by creating opportunities for other optimisations. The expanded code can be optimised using the IR optimisation techniques already described in this chapter and also there may be more opportunities for pipeline optimisations on the target machine. However, depending on target machine code sizes, it could be that there are performance penalties in unrolling because the code no longer fits in the target machine's instruction cache. These are complex interrelated issues requiring careful analysis.

7.4 Parallelism

The analysis that takes place on the intermediate representation as it is being optimised examines control and data flow through the program. This type of analysis is also relevant for the search for sections of the program that can be executed in parallel.

Parallelism has become essential for high-performance computing—high performance now implies parallelism. Parallelism can exist at several different levels in a modern computer system. Instruction-level parallelism can be supported by the processor's hardware where several instructions can be in execution at any one time. This is the technique of *pipelining*. In some processors this parallelism is automatic, performed directly by the hardware. In other processors, clever sequencing of machine instructions generated by the compiler is required to make significant improvements to performance.

Some machines support *vector* instructions allowing operations to be performed on all elements of a vector (a single-dimensional array) simultaneously. Operations such as:

```
for (i=0; i<N; i++)
   a[i] = b[i] + c[i];
```

can be implemented using vector operations, and if the processor allows vectors of size `N` or greater, the implementation is particularly simple, presumably resulting in a very fast execution time.

Multicore processors are now widespread. These are processors that support multiple processor units (*cores*) sharing main memory and other hardware and these cores can execute machine instructions simultaneously. To make the best use of such systems the compiler has to produce multiple instruction streams, one for each core, that can be executed simultaneously. But for this to work reliably, the instruction

streams generated for each core must be independent. They must not interfere with each other, for example by writing to the same memory location. There also has to be synchronisation between the cores. Note that the cores need not have the same architectures. It could be that one of the cores has a special purpose (for rendering graphics, for example) and in such non-homogeneous systems, the different cores should be given workloads appropriate to their architectures.

Other models of parallelism are possible. For example, computer systems can be interconnected via the internet. Such *loosely coupled systems* can clearly execute programs in parallel, but the cost of communication between the executing streams is much greater than if they were executing on separate cores in the same processor. Furthermore, the sharing of data between these processors is more complex.

These approaches to parallelism all have very different characteristics and in order to make use of this parallelism to achieve high-performance computation, different software development and compilation techniques have to be used. The hope that with N processors the program will run N times faster is rarely realised. Furthermore, compiler technology of today is far from being able to take an arbitrary program written in a high-level language and compile it so that it runs at maximum efficiency on a particular multiprocessor system. In some limited circumstances we are close to this goal, but there is still a great deal of work to be done. This is a research issue that has been alive for *many* years and although good progress has been made, there are still many difficult questions to solve.

It is clear that parallelisation by the compiler is a hard problem. But it certainly is a problem that is worth tackling because there are so many computer applications that require huge amounts of computation necessitating the best performance to be squeezed from the hardware. Many of these applications have the potential for parallelisation, using large arrays and extensive numerical computation. There is of course an important underlying issue in the implementation of these applications which is whether they should be coded or recoded by the programmer to make the parallelism explicit. There are now many programming languages supporting the specification of parallel computation. Is the programmer, with a good knowledge of the system design and the algorithms being used, in the best position to decide how the parallelism should be configured? The programmer can also choose algorithms wisely because not all algorithms to perform a particular task adapt well to parallel execution. The alternative approach is to allow the compiler and other software tools to partition the application appropriately. So what can be done by the compiler?

Instruction-level parallelisation has been studied extensively and effective techniques are available. This area is covered in Chap. 8. This is a *local* optimisation in the sense that comparatively small sets of machine instructions have to be examined and rearranged at each step. Vectorisation too has received a great deal of attention and compiler technology can now generate code making good use of the vector instructions available in many of today's architectures. But solving the general problem of taking existing code, detecting sections that can be executed in parallel and determining whether communication costs negate the performance benefits of particular instances of parallel execution is hard and generating any solutions with a degree of

optimality is even harder. Fortunately, there are many aspects of this general problem that are well understood and some of these issues are examined here.

7.4.1 Parallel Execution

The design of the target system is central to the choice of approach to generate parallel code. Although it retains many machine-independent characteristics, parallelisation is clearly a machine-dependent optimisation, and therefore becomes another process that may sit awkwardly in the traditionally machine-independent optimisation phase. In designing parallelisation strategies the key considerations are the number of processors available and whether the processors have shared memory. The style of the decomposition of the problem into sub-problems may depend critically on the number of processors available—2, 10, 1000, a million,…? Systems without shared memory require explicit communication between the processor nodes and this communication may be very slow in comparison with the processor execution speed. In the examples in this section, the assumption is that the processors share a common architecture. These are homogeneous systems.

Consider a simple loop:

```
for (i=0; i<N; i++) {
   a[i] = b[i] + 1;      /* statement 1 */
   c[i] = b[i] * d[i]    /* statement 2 */
}
```

There is clearly scope for vectorisation here, but suppose that we are working on a computer system with multiple processors. There are several ways in which parallelism can be used to speed up the execution of this code, including:

- If there are N or more processors, each of the loop iterations can be done in parallel on N processors;
- If there are 2 or more processors, the first loop (the loop formed by the execution of the first assignment instruction) can be performed on one processor while the second loop is performed on the another processor.
- If $2 \leq$ number of processors $\leq N$ then the N individual iterations could be distributed fairly amongst the processors available.

It may not be possible for the compiler to guarantee that it has made the *best* choice because the choice depends on information that may not all be available at compile time. For example, the actual value of N is clearly important. But a choice can be made.

There may be constraints on the decomposition into parallel processes. Consider modifying the code above:

```
for (i=0; i<N; i++) {
   a[i] = b[i] + 1;      /* statement 1 */
   c[i] = a[i] * d[i]    /* statement 2 */
}
```

Just statement 2 has been changed, replacing `b` by `a`. This immediately raises a problem with the *sequencing* of statements. In any individual iteration of this loop, statement 1 *must* be executed before statement 2, because the value generated by statement 1 is used by statement 2. In this case the opportunities for parallelisation become somewhat restricted.

7.4.2 Detecting Opportunities for Parallelism

To understand how parallelism is affected by the detail of individual statements, a much simpler scenario should be considered where the parallel execution of individual statements is allowed. Suppose the three statements

```
s1: a = b + c;
s2: d = b + 1;
s3: e = b * 2;
```

appear in a program. When coded in a traditional programming language there is an implicit implication that these statements are executed sequentially, `s1` followed by `s2` followed by `s3`. Are other orderings of statements possible? In particular, would it be possible to allocate these three statements to three processors, executing the statements concurrently? It is easy to see here that these statements can be executed in any order. There is no direct interaction between them. This means that the three statements can be executed in parallel. But if small changes are made to this code, constraints are introduced:

```
s1: a = b + c;
s2: d = a + 1;
s3: e = a * 2;
```

Here, `s1` has to be executed first (and its execution has to complete so that the variable `a` is set) and *then* `s2` or `s3` can execute. `s2` and `s3` can execute in either order (or, indeed, simultaneously). There is said to be a *true dependence* from `s1` to `s2` and from `s1` to `s3`. This dependence information can be used to schedule the execution of these statements.

If the code is changed to:

```
s1: a = b + c;
s2: b = c + 1;
```

an ordering is enforced of `s1` followed by `s2`. There is *antidependence* from `s1` to `s2`. This dependence can be avoided by using a new variable name:

```
s1: a = b + c;
s2: b1 = c + 1;
```

where there is now no dependence.

There is a third form of dependence. Consider:

```
s1: a = b + c;
...
s2: a = d * 2;
```

Here, there is a forced ordering on the statements because the value in `a` after both of these statements have been executed depends on the order of their execution. There is an *output dependence* from `s1` to `s2`. Again, this dependence can be avoided by using a new variable name.

A similar form of analysis can be used to determine the existence of dependence between groups of statements and this type of analysis can determine whether blocks of code can be executed safely in parallel. Dependence analysis is also central to the task of instruction scheduling as will be shown in Chap. 8.

7.4.3 Arrays and Parallelism

Loops are particularly important in parallelism detection. Consider the vector version of the first piece of code in the last section.

```
for (i=0; i<N; i++) {
   s1: a[i] = b[i] + c[i];
   s2: d[i] = b[i] + 1;
   s3: e[i] = b[i] * 2;
}
```

Where is the parallelism here? If there are N or more processors available, then each iteration of the loop could be assigned to a different processor and `s1`, `s2` and `s3` executed sequentially on each processor. Or with three processors the loops could be separated into three independent loops so that the loop whose target is `s1` is executed on one processor, `s2` on another and `s3` on a third. It is this second form of parallelism that we should be more interested in here. The aim is to distribute the statements or maybe groups of statements in a loop to different processors so that the loops are executed in parallel. It is a form of vectorisation.

It is easy to see how problems can arise. Consider:

```
for (i=0; i<N; i++) {
   s1: a[i] = b[i] + c[i];
```

```
    s2: d[i] = a[i] + 1;
    s3: e[i] = a[i] * 2;
}
```

Here again there is a true dependence from `s1` to `s2` and from `s1` to `s3`. The implication is that the `s1` loop has to be executed and then the `s2` and `s3` loops can be executed in parallel. Note that we are assuming here that the `s1`, `s2` and `s3` loops are indivisible—for example, we cannot execute a bit of `s1` and then a bit of `s2` and so on.

Consider another example:

```
for (i=0; i<(N-1); i++) {
    s1: a[i] = b[i] + c[i];
    s2: d[i] = a[i+1];
}
```

Parallelising `s1` and `s2` here is not possible because there is an antidependence from `s2` to `s1`. This is easier to see if the loop is unrolled:

```
s1: a[0] = b[0] + c[0];
s2: d[0] = a[1];
s1: a[1] = b[1] + c[1];
s2: d[1] = a[2];
...
```

and it is clear that there is an antidependence from `s2` in one iteration to `s1` in the next iteration. Hence, there is no easy parallelism here.

Obviously the subscripts matter in these array examples. Consider:

```
for (i=0; i<N; i++) {
    s1: a[i*2] = b[i] + c[i];
    s2: d[i] = a[i*2 + 1];
}
```

This is vaguely similar to the last example, but here there is *no* dependence between `s1` and `s2` and so these two loops may be executed in parallel. There is no dependence because the sets of values generated by `i*2` and `i*2 + 1` have no common values.

The more general case has the form:

```
for (i=L; i<=U; i++) {
    s1: a[F(i)] = ...;
    s2: ... = a[G(i)]];
}
```

`F` and `G` are functions returning integer results in the appropriate range. To determine whether there is any dependence between `s1` and `s2`, the set of all values of `F(i)` for $\texttt{L} \leq \texttt{i} \leq \texttt{U}$ and the set of all values of `G(i)` for $\texttt{L} \leq \texttt{i} \leq \texttt{U}$ are compared and

if there are any common values, then there is a dependence. In general, this is not a sensible or, indeed, feasible computation. It can be very expensive (if `U` is very much greater than `L`) and not possible if `L` and `U` are not known at compile time. So, in the general case, a dependence has to be assumed. But if the problem can be simplified so that `F` and `G` are *linear* functions of `i`, then there is a better chance of a solution. Consider:

```
for (i=L; i<=U; i++) {
   s1: a[p*i + q] = ...;
   s2: ... = a[r*i + s];
}
```

In this code `p`, `q`, `r` and `s` are all integer constants (i.e. their values are known at compile time). In this case, the *GCD test* can be applied and this states:

> The greatest common divisor (GCD) of `p` and `r` must divide `(q - s)` with no remainder for there to be dependence.

Consider the code:

```
for (i=0; i<N; i++) {
   s1: a[i*4] = ...;
   s2: ... = a[i*6 + 1];
}
```

Here, `p` has the value 4, `q` has the value 0, `r` has the value 6 and `s` has the value 1. GCD(`p`,`r`) is 2, `q - s` has the value –1, the integer division –1/2 has a remainder and so there is no dependence here.

This is a conservative algorithm in that it can claim that there is dependence when in fact there is none. This is because the algorithm makes no use of the values `L` and `U`. Fortunately this is the safe thing to do. If the algorithm states that there is no dependence then the statements can be parallelised safely. If the algorithm states that there *is* dependence, there is a chance that there is in fact *no* dependence, and the optimisation has been lost, but the right answer from the compiled code will still be produced.

This algorithm can be extended easily to handle multi-dimensional arrays. Other more powerful tools have been developed to obtain better solutions of this problem, but this algorithm is still widely used.

Once the dependences between statements have been determined, it is possible to consider where parallelism can be used to improve performance of the loops. A directed graph is drawn for all the statements in the loop, one node for each statements and directed paths between the nodes to identify the dependences. There are then standard ways of traversing the directed graph to identify which statements can be parallelised and which have to be left as they are.

7.5 Conclusions and Further Reading

Machine-independent optimisation is a well-studied area of compiler design. It is a challenging area of research with the optimisation algorithms being influenced by the designs of the source language, the intermediate representation, the target machine and the user's requirements. Furthermore the interactions between the optimisations are really complicated and it is not possible to model the entire optimisation process with any degree of precision. Extensive empirical studies have to be performed to achieve reliable results which can then influence compiler design.

Because of this complexity, there is little code in this chapter illustrating optimisation algorithms. Instead, the chapter has concentrated on some of the principles of optimisation and the general techniques in widespread use in compilers of today. Fortunately there is a large literature covering this area. Much more detailed information about optimisation and practical algorithms is found in textbooks such as [1–5]. In particular a standard reference for control and data flow is [3] and references [5–7] cover parallelisation in detail. A classic paper on the program dependence graph is [8]. There are survey papers too, for example [9] giving a broad view of optimisation and [10] covering the specific issue of generating compact code. Some background information on programming language constructs on which optimisation can be performed is found in [11, 12].

There are many other interesting approaches to intermediate code optimisation. For example the technique of peephole optimisation (covered in Chap. 8) can be applied to the IR [13]. This proliferation of techniques makes the problem of combining and ordering very difficult and an important reference for this issue is [14].

Exercises

7.1 Have a look at the documentation of your favourite compiler and try and find out which optimisations are supported. If they can be controlled independently, try to assess their effect on a simple benchmark program.

7.2 Propose some guidelines for the implementation of the loop unrolling optimisation. By how much is it reasonable to increase code size? Try to assess the effectiveness of this optimisation.

7.3 Try producing some examples of IR optimisation by hand. Generate some code, preferably involving array manipulation, and carry out local constant propagation and common subexpression elimination. By how much is the code improved?
This task can of course be partially automated by writing an emulator for DL's IR.

7.4 What should a compiler do when it tries to evaluate an expression such as $1/0$ at compile time?

7.5 The obvious exercise is to try implementing some of the optimisations described in this chapter. Produce a front-end to read DL's IR instructions, generating some appropriate internal representation. Then write a back-end to output this

format in the same readable DL IR. Implement some simple optimisations first such as constant folding, then local common subexpression elimination, and then progress to the more complex optimisations.

7.6 Consider extending the idea of constant propagation to propagating *ranges* of values. For example, after a particular statement, it may be possible to show that variable `i` has a value which lies in the range 1 to 10. Under what circumstances could this form of optimisation help? What about maintaining *sets* of possible values?

7.7 How much faster is addition than multiplication on your favourite machine architecture?

7.8 An important machine-dependent optimisation concerns locality of reference. The target machine may run code faster if the data being manipulated can all fit in the processor's cache. Consider the problem of multiplying two two-dimensional arrays. How should the optimisation process try to maximise the locality of reference? Is it a task for the machine-independent optimisation phase?

7.9 Suggest a way in which the evaluation of a complex arithmetic expression could be parallelised.

7.10 Suppose that a shared memory computer system was available which used one million processor cores. How could the compiler make use of such a machine to the benefit of the programmer so that the generated code made good use of the hardware? What sort of software applications would benefit the most?

References

1. Muchnick SS (1997) Advanced compiler design and implementation. Morgan Kaufmann Publishers, Burlington
2. Cooper KD, Torczon L (2011) Engineering a compiler, 2nd edn. Morgan Kaufmann, Burlington
3. Aho, AV, Lam MS, Sethi R, Ullman JD (2007) Compilers – principles, techniques and tools, 2nd edn. Pearson Education, Upper Saddle River
4. Appel AW (2004) Modern compiler implementation in C. Cambridge University Press, Cambridge
5. Allen R, Kennedy K (2002) Optimizing compilers for modern architectures – a dependence-based approach. Morgan Kaufmann, Burlington
6. Zima H, Chapman B (1990) Supercompilers for parallel and vector computers. ACM Press/Addison-Wesley, Reading
7. Wolfe M (1996) High performance compilers for parallel computing. Addison-Wesley Publishing Company, Reading
8. Ferrante J, Ottenstein KJ, Warren JD (1987) The program dependence graph and its use in optimization. ACM Trans Program Lang Syst 9(3):319–349
9. Bacon DF, Graham SL, Sharp OJ (1994) Compiler transformations for high-performance computing. ACM Comput Surv 26(4):345–420
10. Debray S, Evans W, Muth R, de Sutter B (2000) Compiler techniques for code compaction. ACM Trans Program Lang Syst 22(2):378–415
11. Knuth DE (1971) An empirical study of Fortran programs. Softw Pract Exper 1(1):105–133

12. Stanier J, Watson D (2012) A study of irreducibility in C programs. Softw Pract Exper 42(1):117–130. doi:10.1002/spe.1059
13. Tanenbaum AS, van Staveren H, Stevenson JW (1982) Using peephole optimization on intermediate code. ACM Trans Program Lang Syst 4(1):21–36
14. Click C, Cooper KD (1995) Combining analyses, combining optimizations. ACM Trans Program Lang Syst 17(2):181–196

Chapter 8
Code Generation

The process of code generation takes the intermediate representation generated by the front-end of the compiler, with or without any optimisation performed by the machine-independent optimisation phase, and generates code for the target machine. Writing a good code generator is not easy. There is too much choice. There will be infinitely many ways of translating a piece of intermediate code to target code and there is no standard way of performing the translation. The translation process can involve many different algorithms, all interacting with each other in seemingly unpredictable ways, and ensuring that the code produced is of sufficiently high quality is something of a challenge.

However, if optimisation is not a central concern, the generation of target code is not especially difficult. But a good knowledge of the target architecture is needed to design, implement and test the code generation phase. And it is important to plan carefully in advance so that the code generator can be designed to make good use of the features available on the target machine.

8.1 Target Machines

The overwhelming concern while developing a code generator is the nature of the target machine. What are the instructions it supports? What sort of data can it manipulate? What are the memory addressing constraints? Are there caches and if so, how big are they? Does it have registers? If so, how many? Are there registers with special functions? Is it a parallel machine? What is the nature of the parallelism? Does it support multiple cores or are the processors more loosely coupled? Does it support instruction-level parallelism? Are there any other features that could be useful for the generated code? But not all target machines consist of real hardware. A compiler can of course generate code for a virtual machine, implemented in software. Similar

questions can be asked about the nature of the machine. But there may be additional flexibility. Can the design of the virtual machine be changed? Dynamically?

There is a great deal of detail to worry about here. And there is not yet any manageable way of using formal techniques to represent this information and use it to specify a code generator. Before worrying about these complications, it is worth outlining some much simpler techniques for rapid implementation.

A simple way to implement a code generator is to translate each intermediate representation instruction, one at a time, into target code. Context can largely be ignored and this simplifies the translation process greatly. Code generation then becomes an operation based on pattern matching. There is a pattern for each IR instruction and the translation into target code is modified according to the actual arguments of the IR instruction. This results in non-optimised code because only a limited account is taken of the interaction of individual instructions.

It may also be possible to implement the compiler by targeting a high-level language already implemented on the target machine. For example in the early days of C++ compilers, several C++ implementations were produced by generating C code which was subsequently compiled using an existing C compiler. Compilation of C++ then became a two-step process, using C as an intermediate representation. This again is an approach that makes the implementation process somewhat simpler.

Yet another approach completely avoids the writing of a code generator. Instead, an interpreter for the intermediate representation can be coded in any language already available on the target machine. For example, DL's IR is very simple and developing an interpreter for this language is a manageable software project. Many languages have been implemented in this way. This approach will be re-visited in Chap. 9.

However, there will always be a need for compilers that directly and efficiently target specific real or virtual machines and so it is important to examine some of the issues in producing a code generator design.

8.1.1 Real Machines

In the early days of processor design, there was an assumption that most of the programming would be done at the machine/assembly code level. But as the use of high-level languages increased rapidly, the design of processors became more appropriate for compiler-generated code. In particular, the instruction set became more regular so that, for example, the instructions for all the arithmetic and logical instructions shared a common format, resulting in less need for code in the code generator to handle special cases. The compiler writer is also helped by two other processor features. The first is the widespread use of simpler instruction sets and the second is that modern processors often incorporate hardware features to support the dynamic optimisation of executing code.

The processors available today vary hugely in terms of functionality, speed, instruction set, memory, architecture, data width and so on. Writing a compiler for some of these processors could be astonishingly difficult, whereas other processors

are much kinder to the compiler writer. Compiling software for a tiny embedded processor used in a doorbell-style applications is radically different to compilation for a powerful multicore processor used in a general-purpose enterprise server. Compiling where there are real-time constraints poses some interesting problems too. There is no universal solution.

Plainly, processor complexity has increased steadily since the days of the first computers. Moore's law (the number of transistors in an integrated circuit approximately doubles every two years) implies processor speed increases too. But recent trends have suggested that the increases in processor complexity have primarily gone towards the support for increased parallelism in processors, specifically towards the increase in numbers of processor cores available. This poses particular problems for the compiler writers. The application programmers need code to exploit the new highly parallel machines.

8.1.1.1 RISC and CISC

There is a useful classification of processors into either *reduced instruction set computers* (*RISC*) or *complex instruction set computers* (*CISC*). Both types of processor are used today.

The RISC processors have a simple instruction set and each instruction runs fast because it can be completed in a small number of processor cycles. These are the *load/store architectures*, where main memory is accessed by specific load and store instructions (reading from memory to a register and writing to memory from a register) and other instructions do not access main memory. Instead, the arguments for these instructions are held in machine registers. Instruction lengths tend to be uniform.

The CISC processors are usually based on a larger instruction set made up of more complex instructions requiring more processor cycles for their execution. For example, a single instruction may load two arguments from main memory, perform some arithmetic operation on these values and store the result in main memory. Complex addressing modes are often supported.

The dividing line between these two computer types is fuzzy. For example, a RISC may incorporate CISC-like addressing modes. And one type is not inherently better than the other. Performance depends on the application and the compiler far more than on whether the code will run on a RISC or a CISC.

As far as the compiler writer is concerned, the RISC is *probably* the architecture of choice because the instruction set is inherently simpler. However, CISC instruction sets have been designed to include instructions specifically for the support of high-level language constructs. For example, there is often support for function call and return where the instructions automatically perform the housekeeping operations of manipulating the stack pointer, saving registers and so on. This can simplify code generation, but it makes it hard to implement an unconventional form of call and return. Some instructions supported by CISCs are difficult for the compiler to generate despite their obvious application. For example, the VAX architecture

supported polynomial evaluation instructions using a pointer to a table containing the polynomial coefficients and a register containing the value of the independent variable [1]. Compiling an assignment statement evaluating a polynomial into the compiler's IR and then recognising in the code generator that this particular sequence of IR instructions can be implemented by one of these machine instructions is not easy. These polynomial instructions were doubtless directed towards the assembly language programmer writing small library routines.

8.1.1.2 Data Types

For each of the data types supported by the language being compiled, a target machine representation must be found. Hopefully, for most of these data types, the processor provides native support. For example a C `int` could be mapped to a 32-bit integer on a processor providing a full set of arithmetic instructions operating on these integers. In many high-level languages there can be flexibility in the implementation but other languages such as Java specify, for example that integers are 32-bit 2's complement quantities. Sometimes the target machine may not have direct support for the data type. For example, consider a C implementation targeted to a small 8-bit processor. Is it reasonable to limit a C `int` to 8 bits, resulting in integers in the range -128 to $+127$? Or would it be better to make use of library routines to perform the arithmetic so that 16- or 32-bit integers could be emulated? Both signed and unsigned integer arithmetic may be required.

Similar considerations apply to floating point arithmetic. Emulation of floating point arithmetic in software will result in slow calculations but if the hardware lacks floating point support and the language being implemented requires floating point support, the use of emulation is essential.

Implementation of characters and strings should be simple. Is the use of an 8-bit character set such as ASCII sufficient or does the programming language have to support Unicode characters? There are standard ways of representing Unicode characters. Strings are usually stored in a contiguous area of storage with an associated length value or as a sequence of characters terminated by a null or other terminating value. The language may define how strings are to be represented or it may be an implementation decision.

Some languages allow the manipulation of individual bits explicitly (for example, the *bit-fields* of C) or implicitly (for example, if the language supports sets). Boolean values may be implemented as single bits. Mapping these bit operations onto the target hardware can be done by using logical operators, but some processors have machine instructions allowing designated bits in a word to be manipulated.

Machine memory address manipulation is almost inevitably an aspect of the generated code. Do integers and address values require the same amount of storage? How are variables requiring extended storage areas (arrays, structures, objects, etc.) accessed both as a whole and also via their individual elements?

8.1.1.3 Addressing

Processors support a variety of *addressing modes*. For a straightforward implementation of most conventional programming languages only basic addressing modes such as direct (specify an address directly), indirect (the address is held in a register) and indexed (the address is obtained by adding the contents of a register to an explicit offset value) are required. Processors over the years have supported a huge variety of addressing modes, sometimes included on the false assumption that they might be of help to the compiler writer. Many of these fancy addressing modes are really difficult to generate directly, and it may be better to make use of them through optimisation phases late in the code generation process such as peephole optimisation described in Sect. 8.5.3.

Some addressing modes are designed to support particular high-level language operations. For example, *autoincrement* and *autodecrement* addressing are directed towards array operations. The loop:

```
for (i=0; i<N; i++) a[i] = 0;
```

could make use of an autoincrement addressing mode by generating code before the loop to load the address of the element `a[0]` into a machine register (say `r1`) and then within the loop including the machine instruction `clr (r1)+` which sets the location pointed to by `r1` to zero and automatically increments the value in `r1` to point to the next element of `a`. This is an instruction with side-effects and it may be difficult to generate directly during code generation. Instead it is often handled if necessary by the peephole optimiser.

CISCs usually support a large range of addressing modes, some really useful for array access, for example, by automatically multiplying index values by the size of each array element. Stack-based addressing is sometimes supported. This may be of limited utility for a conventional programming language implementation.

8.1.2 Virtual Machines

Compilers do not have to generate code for real machines. Target languages can be designed without the constraints faced by the hardware designer. Using a virtual machine as a target can often simplify the implementation process. An interpreter for the virtual machine may be written without great effort. Furthermore, the target language can be another high-level language. Several compiler projects have used C as a target language and the task is completed by using an existing C compiler to translate the generated C to target machine code. Here, the compiler is translating from one high-level language to another.

A well-known virtual machine associated with programming language implementation is the *Java Virtual Machine* [2]. This is not just an intermediate representation between front-end and back-end—it is a representation that can be interpreted by

fairly simple emulator software, allowing Java programs to be run. This stack-based virtual machine is independent of real hardware and it provides a route towards target machine-independent compilation.

The use of virtual machines is an important aspect of programming language implementation, providing a mechanism to investigate novel processor designs, the possibility of dynamic modification of the characteristics of the virtual machine, support for multi-language systems, supporting language portability and so on.

8.2 Instruction Selection

The process of instruction selection forms the core of code generation. The aim is to take the IR generated by the front-end and replace the IR instructions by functionally equivalent target machine instructions. This phase of compilation can be thought of as another compiler, translating from a source language to a target language. But this translation is inherently ambiguous. There are infinitely many ways of generating target code. Obviously we want the best code (whatever that may mean) but guaranteeing optimality is essentially impossible. Good code may have to do.

Instruction selection can be regarded as a process of pattern matching. As already mentioned, a simple approach to instruction selection is to replace each IR instruction by corresponding target machine instructions. This works but generates poor target code because contextual information is ignored.

A widely used approach to instruction selection is to first transform the IR into a *tree* representation. At first sight, this seems like a retrograde step because the IR has only recently been generated from a tree. But this tree is at a lower level than the tree generated by the syntax analyser. The operations in the tree correspond to the types of operations performed by target machine instructions. Also, for each instruction of the target machine, there is a tree rewriting rule. Each rule contains a tree pattern to be matched (essentially describing the actions of the instruction) and a replacement (essentially specifying where the result of the instruction, if any, is placed). Given the low-level tree generated from the IR and a set of tree rewriting rules, one for each machine instruction, the aim is to *cover* or *tile* the IR tree with these instruction tree patterns. Each time a pattern is matched, the pattern in the tree is tagged with the corresponding replacement in the rewriting rule, the corresponding machine instruction is noted and this process is repeated until all nodes in the IR tree have been matched—it has been completely covered. Then a bottom-up tree walk can be performed, emitting the matched machine instructions as the tree traversal is done.

To illustrate the style of these trees, the tree rewriting rules for some simple machine instructions can be seen in Fig. 8.1. These three examples show tree rewriting rules, indicating how the IR tree should be modified if the tree on the right-hand side of each diagram is matched with a subtree in the IR tree. In the first instruction, a constant value c_i in the tree can be replaced by a machine register r_n by emitting the instruction li r_n, c_i (load immediate). Similarly, in the second instruction, the tree

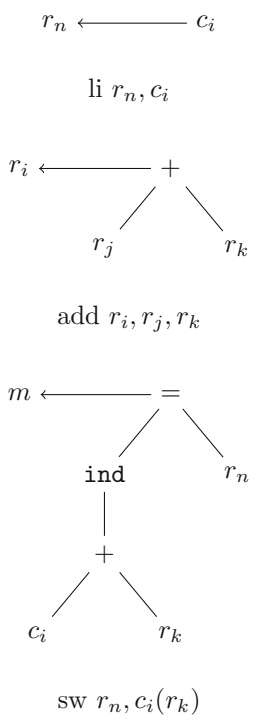


Fig. 8.1 Trees representing machine instructions

can be replaced by r_i, emitting the instruction add r_i, r_j, r_k ($r_i = r_j + r_k$). And in the third instruction, the matched tree is replaced by a memory location m, emitting the instruction sw $r_n, c_i(r_k)$ (store register r_n in the location whose address is found by adding the contents of register r_k to the offset value c_i). The `ind` node (*indirection*) causes the value returned by the subtree below it to be used as an address and the contents of that address are returned.

Clearly an algorithm is needed to control which instruction trees are used and the order in which they should be matched in the IR tree. This process of *tiling* the tree specifies the target machine instructions that will be generated and the quality of the generated code will depend on exactly how the tree is tiled. Furthermore, the tiling algorithm *has to* terminate with all the tree being consumed in the process. The instruction patterns have to be chosen so that this is possible. If there are patterns covering all possible single nodes then the tiling is guaranteed to terminate. An appropriate algorithm is the *maximal munch* algorithm (see [3] for the details). Other algorithms are used too [4]. These algorithms are not perfect in that they are not guaranteed to result in the best tiling. One obvious issue is that no account is taken at this stage of register allocation. It is assumed that there is an unlimited number of registers (if that is what was assumed when the IR was generated) and register allocation is assumed to be performed after instruction selection. But there is a two-

way interaction here between register allocation and instruction selection and so getting the code perfect is not really feasible.

The tree patterns for RISC instructions are mostly small, but the patterns tend to be much larger for CISC instructions. For example, defining patterns for the complex instructions supported by graphics processors may be challenging. Therefore instruction selection for RISCs turns out to be somewhat simpler. But in all cases the search space can be huge. Trying to find a good tiling of the IR tree can be a time-consuming task and so the matching and optimisation algorithms should be chosen and implemented carefully.

A closely related approach to instruction selection is to achieve the pattern matching by running a parser on a flattened version of the IR tree produced by running a pre-order traversal. A LR parser is used to match the patterns with the flattened tree input and target machine code can be emitted as the matching succeeds. This looks very much like the syntax analyser of a compiler, but the key difference is that this matching is inherently ambiguous. The parsing/pattern matching has to be supported by algorithms removing the ambiguity by attempting to minimise the cost of the generated code. This form of instruction selection has been used in various compilers with some success.

8.3 Register Allocation

It is likely that the target machine for which code is being generated has *registers*. Details are processor-dependent, but registers are fundamentally important to the process of code generation for two reasons. The first is that machine operations using values held in registers are almost certainly much faster than if the values were held in main memory. Second, the machine architecture design may require that the arguments for particular instructions, such as arithmetic or logical instructions, must be specified as registers. So the arguments must be explicitly loaded into registers before the particular instruction is executed.

Consider the compilation of the assignment statement `a = b + c` (where the variables `a`, `b` and `c` are held in storage locations named `a`, `b` and `c`) for various styles of target machine. If the machine supports arithmetic instructions taking three address arguments, one for the result and two for the two operands, then this assignment could result in an instruction of the form:

```
add a,b,c
```

to be generated. Few machines support this form of instruction. Here, registers are irrelevant or not very important. One is much more likely to find arithmetic instructions requiring at most one of the arguments to be in main memory and the other or others in registers (*register-memory machines*).

```
load r1,b
add r1,c
```

```
store r1,a
```

To do the computation, `r1` has to be used to hold the value of `b` temporarily. Another machine architecture may require that the `add` instruction has two arguments, *both* being in registers (*register–register machines*).

```
load r1,b
load r2,c
add r1,r2
store r1,a
```

Here, two registers (`r1` and `r2`) are required for the computation. These machines may use three-operand instructions:

```
load r1,b
load r2,c
add r3,r1,r2
store r3,a
```

Here, `r1` is not overwritten with the result of the addition.

Registers are a precious resource and have to be used carefully to enable concise and efficient code to be produced. The process of *register allocation* is responsible for taking the references to values in variables, temporaries, arguments and so on in the intermediate representation and making sure that they are held in registers wherever possible in a way that attempts to maximise the efficiency of the generated code. The aim is to keep all the frequently accessed data in registers.

The intermediate representation being used as the input to the register allocator could be designed so that it does not use any registers. Instead it refers to variables and other values by using symbolic names and it is a task for the code generator to map these variables and values to hardware registers and memory locations. It is more likely that the program's intermediate representation has been designed such that it already makes use of registers called *virtual registers* and no limit has been applied to the number of registers being used. This may be a low-level intermediate representation, produced by the instruction selection phase, and the instructions will be close in structure to the actual target machine instructions. The IR registers have to be mapped onto the set of real hardware registers available. The set of registers available is likely to be a subset of the actual hardware registers on the processor because some of the registers may be dedicated to other tasks such as stack pointers, storage of particular constant values and so on. Some registers may be dedicated for use with specific instructions. Furthermore, it may be that the processor may implement register classes, for example a set of registers for floating-point operations and another set for general integer purposes.

Ideally, we would like to be able to map all virtual registers to distinct physical registers. However it is certain, except for the simplest of source programs, that there will be many more virtual registers than physical registers and so the use of the physical registers will have to be shared. This can be achieved by *register spilling*.

The contents of a register are saved to main memory and the register can then be used for another purpose. The saved register contents can be returned to a register at a later point in the execution of the program. Clearly, spilling should be avoided if at all possible because it can be a costly operation especially if it is executed many times in the code of a loop.

Register allocation is a complicated task, ideally needing information about the runtime behaviour of the program so that execution counts for each instruction can be predicted to determine the optimal allocation. This is not in general possible so heuristic approaches have to be used.

A useful starting point is to consider register allocation within a basic block. Consider a self-contained basic block requiring no registers for passing values into or out of the block. There is a predefined number of hardware registers available for use in the code in the block. One or two of these registers will be needed for the spilling code but the remainder can be allocated to the most frequently referenced virtual registers. If the allocation process concentrates first on those basic blocks within nested loops (this is the code likely to be executed most often) then there is a chance of a reasonably effective register allocation.

However, the register allocation problem for real programs is made more awkward by the need for dealing with programs made up of many basic blocks, with values in registers being passed between them. To help towards a solution to this problem the concept of *liveness* is introduced.

8.3.1 Live Ranges

The live range of a variable starts at its definition (i.e. when it is first initialised) and ends with its last use. If a variable is redefined (i.e. it has a new value assigned to it) a new live range is started.

Consider this code. There are no other references to `x` or `y` outside the live ranges shown.

```
.
x = 1;  ┐
.       │
.       │ live range 1
... = x;┘
.
.
y = 2;  ┐
.       │
.       │ live range 2
... = y;┘
.
```

In this code example, the two live ranges do not overlap. Therefore the generated code can use the same register for both `x` and `y`. Suppose that the references to `y` in this example were changed to references to `x`, there would still be two live

ranges which are non-overlapping so that again just a single register is needed. Static single assignment (SSA) form makes this analysis a little more explicit. In SSA form, variables can only be assigned to once, marking the start of a live range, and the subscripting of variable names makes an explicit connection between a use of a variable and its definition.

It is live ranges rather than variables that are central to the register allocation problem. Finding live ranges in straight-line code is trivial, but when the control flow is more complex the analysis becomes very much more difficult. Further information about appropriate algorithms is contained in the references in Sect. 8.7.

8.3.2 Graph Colouring

The analysis of live ranges can be used to form the basis of register allocation. Each virtual register in the code being analysed has to be allocated to a physical register but there are likely to be many more virtual registers than physical registers. The register allocation analysis requires the construction of a *register interference graph*, where the nodes are the virtual registers and an edge between two nodes indicates that the two virtual registers have overlapping live ranges where one of the virtual registers is live at the point at which the other is defined. If live ranges overlap, the virtual registers cannot share the same physical register.

After the interference graph has been constructed, the allocation problem becomes one of *colouring* the nodes of the graph so that no two connected nodes share the same colour. If the graph can be coloured using n different colours and there are n or more physical registers available, the allocation problem succeeds. A colour represents a particular physical register. This is the graph colouring problem, a well-known problem of graph theory, arising originally from colouring maps so that countries sharing a boundary have different colours.

A very simple example is shown in Fig. 8.2. The live ranges for the three variables `a`, `b` and `c` are marked on the code on the left. Live range `a` overlaps with both live ranges `b` and `c`, and there is no overlap between live ranges `b` and `c`. This information allows the register interference graph on the right to be drawn. This graph can be coloured using just two colours (white for node `a` and grey for nodes `b` and `c`). So two physical registers are required, `b` and `c` can share a register.

Unfortunately, graph colouring is an *NP-complete* problem (no polynomial-time solutions are known) and exhaustive search for minimum register solutions is not practically feasible as the size of the interference graph increases to that found from even average programs. Heuristic approaches have to be adopted.

8.3.2.1 A Graph Colouring Algorithm

A popular algorithm that usually does a good job to colour a register allocation graph uses a recursive process of repeated graph simplification. This is an $O(n)$ algorithm

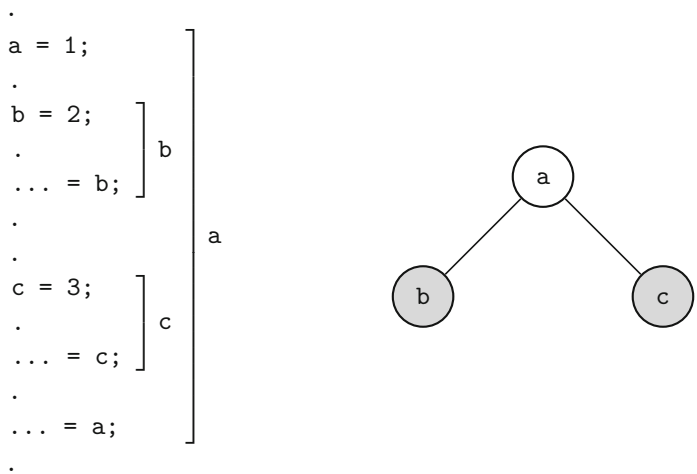


Fig. 8.2 Live ranges and register interference graph

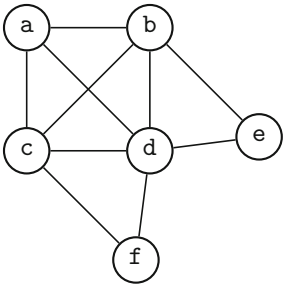
(n is the number of nodes), a major improvement on the exhaustive search implied by the NP-complete problem. Suppose that there are k physical registers available.

Once the register allocation graph has been constructed, the next step is to attempt to simplify it by repeatedly removing nodes. Suppose that there is a node X in the graph which has fewer than k neighbour nodes. Remove X from the graph. If it is then found that the resultant graph is colourable using k colours, then the whole graph with X replaced must also be colourable using k colours. This is because in the graph with X missing, X's neighbours must have been coloured with $\leq k - 1$ colours and so there is one or more colours available to colour node X.

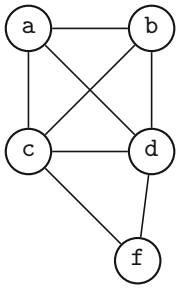
The algorithm therefore repeatedly removes nodes with fewer than k neighbours, adding the identity of the node removed to a stack. At each step the graph is simplified by the removal of a node and its connection links. If this process continues until the last node is removed from the graph, then the graph is colourable using k colours and the register allocation succeeds. The graph can then be reconstructed by returning nodes one by one from the stack, each time colouring the node with a colour distinct from any of its neighbours already replaced in the graph.

To show how this part of the algorithm works consider the register interference graph shown in the first diagram in Fig. 8.3. Suppose that there are four physical registers available. Is it possible to allocate these six virtual registers `a`, `b`, `c`, `d`, `e` and `f` to `r1`, `r2`, `r3` and `r4` without the need for any register spilling? The graph colouring algorithm proceeds as shown in Fig. 8.3.

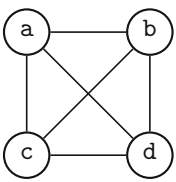
At each step a node with fewer than k neighbours (k is 4 in this example) is chosen and temporarily removed from the graph. The example here arbitrarily chooses node `e` as the first to go (it has two neighbours) and `e` is put on the stack and the new graph is shown in the second picture. Then, nodes `f`, `a`, `b` and `c` are removed in turn, leaving the last node `d` to go at the final step, emptying the graph.



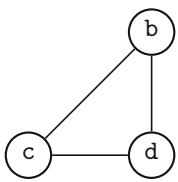
Start — stack: (empty)



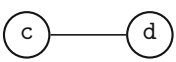
Step 1 — stack: e



Step 2 — stack: e, f



Step 3 — stack: e, f, a



Step 4 — stack: e, f, a, b

Step 5 — stack: e, f, a, b, c

Step 6 — stack: e, f, a, b, c, d

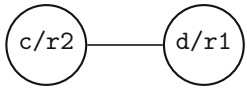
Fig. 8.3 Graph colouring algorithm

The graph can then be reconstructed in reverse order according to the data held on the stack and physical registers allocated as shown in Fig. 8.4.

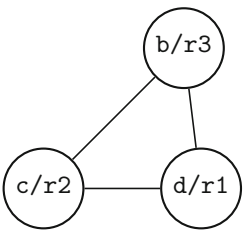
Node `d` is first replaced. At this stage `d` has no connections and so there are no constraints on which physical register (colour) it can be allocated to. Allocate it to `r1`. When node `c` is replaced it has a connection to node `d` and so has to be allocated to a physical register (colour) that is not `r1`. Choose `r2`. Node `b` and `a` are returned, allocated to `r3` and `r4` respectively. When node `f` is returned it has to be allocated to a physical register that is not `r1` or `r2`. Choose `r3`. Finally `e` is returned, allocated to `r2`. All nodes have been coloured. The allocation process is complete.

This algorithm seems to work very well. However it must be enhanced to allow it to deal with the situation where there are no nodes in the graph which have fewer than k neighbour nodes. This implies that the algorithm has failed to find an allocation of virtual to physical registers. For example, consider the graph in Fig. 8.3 but with just 3 physical registers instead of 4. Both steps 1 and 2 can be performed as before, removing nodes `e` and `f`, both having just two connections, but then nodes `a`, `b`, `c` and `d` all have three connections and the allocation process cannot continue. In order

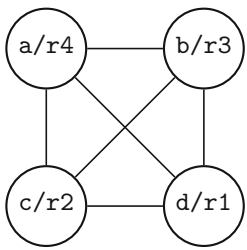
Step 1 — return node `d` — stack: `e`, `f`, `a`, `b`, `c`



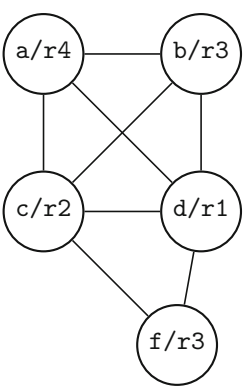
Step 2 — return node `c` — stack: `e`, `f`, `a`, `b`



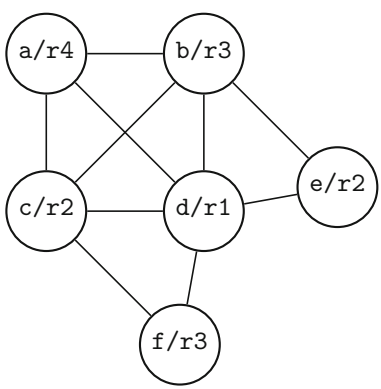
Step 3 — return node `b` — stack: `e`, `f`, `a`



Step 4 — return node `a` — stack: `e`, `f`



Step 5 — return node `f` — stack: `e`



Finish — return node `e` — stack: (empty)

Fig. 8.4 Graph colouring algorithm—register allocation

to resolve this problem, one or more physical registers have to be *spilled*. In other words the virtual register has to be stored in main memory instead of being placed in a physical register throughout its live range. This is modelled in the graph colouring process by removing the node corresponding to the spilled value. This should free up other nodes, reducing their connection count, allowing the allocation process to continue. Choosing a virtual register to be spilled really needs information about estimated execution frequency. Ideally only the infrequently-used values should be considered for spilling.

8.3.3 Complications

Achieving a high-quality register allocation is essential if well-optimised code is to be produced by the compiler. The underlying algorithms may be clear, but there are many irritating complications, usually introduced by particular language statements or statement types.

For example, after a simple assignment of the form `x = y` both `x` and `y` are live. But they do not necessarily need to have separate registers. It may be possible for them to share a register because they contain the same value. This is good news because registers are precious commodities.

Another issue concerns *aliasing*, illustrated well by C's indirection operator `*`. If `p` is declared as `int *p;`, then a statement of the form `*p = 3;` can play havoc with register allocation. Where does `p` point to? If a static analysis of the code can determine that it is definitely pointing to variable `i` then the storage location associated with `i` can be updated to contain the value 3. If `i` is currently held in a register, then that register has to be updated to contain the value 3. But it is more likely that the compiler cannot determine where `p` is pointing and if so, it may then be necessary to generate code to store 3 in the location pointed to by `p` and somehow invalidate or reload all other registers containing variables. This may cause havoc with the register allocation process and may have significant runtime costs.

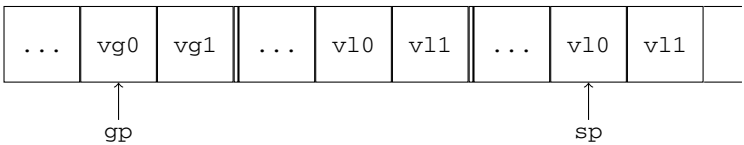
8.3.4 Application to DL's Intermediate Representation

DL's intermediate representation (see Sect. 6.5) uses three register types. Globally declared variables are called `vg0`, `vg1`, …, locally declared variables are called `vl0`, `vl1`, … and temporary values are placed in virtual registers `r0`, `r1`, …. As far as register allocation is concerned, these three types can be treated identically. But there is one small difference in that the virtual registers `r0`, `r1`, … have no associated main memory storage. They have no *home location*. So if one of these virtual registers is stored in a physical register and that register needs to be spilled, there is no automatically available main memory location for that register's storage. During code generation some runtime storage has to be set aside for the purpose. Storage for all the `vg0`, `vg1`, …, `vl0`, `vl1`, … registers is easily allocated on a runtime stack as described in Sect. 8.4. Space for spilling any of `r0`, `r1`, … should it be required can be allocated there too.

8.4 Function Call and Stack Management

The approach to DL's runtime storage allocation described in Sect. 6.2.3 can be carried forward with little change to suit most target machine implementations. A stack is used for variable storage, local variables being accessed via the register `sp`

and global variables via gp. Addressing variables is simple. Consider a possible state for the runtime stack for the program outlined in Sect. 6.2.3:



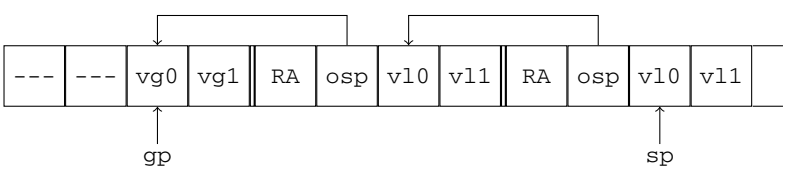
This diagram clearly shows how different types of variables can be accessed in DL.

8.4.1 DL Implementation

Generating target machine code to access the variables should be easy. For example, if an IR instruction references vg1, the code generator produces code to access the storage location whose address is the contents of the gp register plus 1, expressed in target code using indexed addressing as 1(gp) or equivalent. Similarly, vg34 would be accessed as 34(gp), and so on. A reference to vl1 would be expressed as 1(sp).

Function call and return also have to be implemented. This is where the design of the contents of the linkage information stored on the stack become relevant. The key purpose of this information is to enable the execution environment to be restored to what it was at the point of the function call. This means that the linkage information has to contain a *return address* (the address of the instruction following the call to the function) and also the value of the *old stack pointer* at the point of the call. These two values are stored by code generated with the function call and on return the old stack pointer is reinstated and control is passed to the stored return address. These stack pointers form a *chain* pointing down the stack. This is the *dynamic chain*, dynamic because the chain reflects the pattern of function calls and returns as the program executes. Recursive functions need no special treatment in this scheme.

The runtime stack, in greater detail, showing the dynamic chain, has the form:



The old stack pointer (osp) and the return address (RA) are accessed via negative offsets from the stack pointer. The return address and old stack pointer fields in the global stack frame are not needed in this implementation.

DL also requires storage for the temporary variables/registers r0, r1, ... Depending on how the register allocation process performs, some of these registers will almost certainly require storage space. These variables are all local to the function (or the main program) and hence they can be stored on the stack beyond the other

named local variables. The code generator has to keep track of the offsets of these temporary variables so that their "r" names can be mapped to offsets off the current stack pointer.

DL is a simple language where variables are either local or global and so just two stack pointers are required to locate all variables. There are languages, Pascal is a good example, where a more complex implementation is required. In these languages, procedures (or functions) can be defined *within* procedures. For example, if a procedure p2 is defined within a procedure p1 which in turn is defined in the main program, p1 has access to its local variables as well as the global variables defined in the main program. And when p2 is active, it too has access to its local variables, p1's local variables as well as the global variables. In this example, p1 has no access to p2's local variables. Complexity is somewhat increased when these procedures can call themselves recursively. At any time in the program's execution there is a list of procedure activations whose local variables are accessible by the running code. This is *not* the dynamic chain. A separate chain is required reflecting the static nesting of the procedure definitions. This *static chain* has to be updated as calls and returns are performed to ensure that the stack frames linked together by this chain identify the sets of accessible local variables. The static chain stored in the stack can be considered as a chain of pointers to the stack frames containing active variables. In other words it is a set of stack pointers. As the static depth of procedure nesting increases the number of active stack pointer registers has to increase too. Implementation is not difficult, but the management of the static chain introduces a small overhead to the call and return code.

8.4.2 Call and Return Implementation

Having decided on how the stack frame mechanism should work and how variables should be accessed, the next step is to design the details of the code to be generated to perform the function call, to initialise the function and to perform the return to the caller. It may be easy to decide what should happen and hence what code should be executed on a function call, but it may be harder to decide how those operations should be split between the site of the call and the start of the function. Some operations can be performed at either location and for these it is better to perform them at the start of the function. This is to minimise code size because repeating the code at each call results in larger code.

Consider first what happens at a function call. What from the caller's environment has to be saved? Clearly a *return address* has to be saved on the stack so that once the called function has completed its work, control can be transferred to the instruction immediately following the call. The *current stack pointer* has to be stored too so that it can be reinstated on return. The call also has to create a new stack frame for the called environment. In other words, the stack pointer has to be moved up the stack to the first free location following the current local variables. In DL the code generator

will know the size of the current stack frame enabling this value to be added to the stack pointer, ready for the local variables of the called function.

Function return is a little simpler. All that has to be done is to restore the stack pointer by loading the stack pointer register from the old stack pointer value stored on the stack and to jump to the return address, also stored on the stack.

How can all this be implemented on the target machine? Which machine instructions should be generated? This naturally depends on what's available. Some machines support special instructions to perform some aspects of these call and return actions automatically, maybe a special `call` instruction to save the stack pointer and return address in particular registers and then branch to the specified destination address. The use of dedicated registers may help speed up and simplify the call code but dedicating registers for this purpose may result in costs elsewhere. At the very least there should be machine support for a `call` instruction that saves the return address in a register. This can then be saved on the stack with the current stack pointer using instructions with indexed addressing. There should also be some form of `return` instruction to pick up a destination address stored in the stack, or in a register loaded by another instruction, and jump to that address. Selecting these instructions for best performance is important. Space and time efficient calls and returns can make a large difference to overall code efficiency.

8.4.2.1 Argument Passing

On a function call, in addition to the actions described above, any arguments to the function have to be dealt with. In the case of DL, the principle is easy.

```
.
.
myfunc(a,b);
{ ... }
.
.
{
    .
    i = myfunc(3,4);
    .
}
```

In this example, the local variables `a` and `b` in `myfunc` have to be initialised during the call with the *actual parameters* 3 and 4. So code has to be generated to evaluate the actual parameters immediately before the call and store the values at offsets 0 and 1 in the *new* stack frame being created for `myfunc`. The parameters could, of course, be passed in machine registers as well, or instead.

This form of argument passing is known as *call by value*. It is the *values* of the arguments to which the called function has access. The called function has local copies of

the arguments. If `myfunc` had been called using the statement `i = myfunc(p,q);` then `myfunc` cannot directly update the variables `p` and `q` in the caller's environment. Many programming languages, including C, use this form of argument passing.

Some languages support *call by reference* where it is the *address* of the actual argument that is passed so that in the example above, the addresses of `p` and `q` are passed into `myfunc`, allowing the code in `myfunc` full (read and write) access to the variables `p` and `q` in the caller's environment. This makes no sense for calls of the form `myfunc(1,2)` or `myfunc(p+q,p-q)` and appropriate error messages should be generated in these cases.

Other argument passing styles are occasionally used such as call by value-result or call by name. The implementation should obviously support whatever it is that is required.

8.5 Optimisation

Chapter 7 covered techniques for code optimisation at the intermediate representation stage of compilation, using techniques that are independent of the design of the target machine. Optimisation within and after the code generation phase is based on target machine-dependent techniques. This is where a detailed knowledge of the target machine is especially important.

There is considerable scope for optimisation at this stage. Modern processor hardware is often extremely complex and optimisation is no longer centred around the minimisation of the number of instructions. In order to optimise for speed, other considerations such as instruction scheduling, the use of parallel execution, the effective use of caches and so on may be much more important. Rules for the effective use of these machine characteristics can be coded into the code generation process. However, the complexity of this task should not be underestimated.

This section provides a brief overview of some of the issues involved in machine-dependent optimisation. As hardware is developed, new challenges appear for the compiler writer.

8.5.1 Instruction-Level Parallelism

Over many years of processor development, the primary source of performance improvement has been the steady increase in clock speeds. But as physical constraints are preventing unlimited clock speed increases, the introduction of parallelism is becoming increasingly important. In Sect. 7.4 the high level aspects of parallelism were introduced. But performance improvements can also come from parallelism at a very much lower level, within the execution of individual instructions. For the compiler writer this means that instruction order can affect execution speed. The

compiler should schedule instructions to make best use of the instruction parallelism offered by the hardware.

This task is handled by the *instruction scheduler*. It takes target machine code as input and it produces code where the instructions are reordered to make best use of the processor's instruction-level parallelism. It must, of course, maintain the semantics of the code. Instruction scheduling is not relevant to all processors. Also *superscalar* processors can perform out-of-order instruction execution automatically. Here, the processor looks ahead in the instruction stream, dynamically as the program runs, selecting multiple instructions that can be initiated simultaneously in the same processor cycle, allowing instructions or operations within instructions to execute in parallel. However the code for many processors can benefit from the use of an instruction scheduling pass, performed statically by the compiler.

By duplicating hardware, processors can support simultaneous execution of multiple instruction streams. Another parallelism model makes use of an *instruction pipeline*. Here, the execution of an instruction is broken down into several small steps (e.g. instruction fetch, instruction decode, perform operation, store result) and these steps taken from multiple instructions can be interleaved so that multiple instructions seem to be executing in parallel. This pipeline may not be able to run smoothly and may stall because, for example, an instruction may want to use the result from an immediately preceding instruction and therefore has to wait until the first instruction has completed. Jump instructions also cause problems. Therefore, instruction scheduling is important to pipelined architectures to minimise the proportion of instructions causing pipeline stalls.

The instruction scheduler has to examine the dependencies between instructions and determine how many clock cycles are required to resolve each of the dependencies. There are clear parallels here with the ideas of data dependence introduced in the last chapter. For example, consider a simple pipelined machine with a two-stage pipeline so that each instruction executes in two clock cycles. Suppose that the first pipeline slot fetches and executes the instruction and the second slot stores the result. Consider the simple instruction sequence:

```
(1) r1 = x
(2) r2 = y
(3) r3 = r1 + r2
```

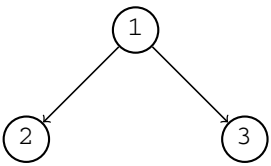
This would be executed as follows.

- Time slot 1—instruction 1 starts.
- Time slot 2—instruction 2 starts, instruction 1 completes and `r1` loaded.
- Time slot 3—instruction 3 cannot start because `r2` is not yet loaded. Introduce a delay slot and do nothing.
- Time slot 4—instruction 2 completes and `r2` loaded, instruction 3 starts.
- Time slot 5—instruction 3 completes and `r3` loaded.

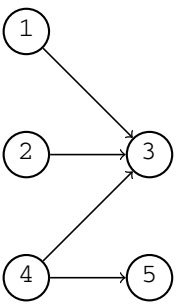
There are dependencies from instruction 1 to instruction 3 and from instruction 2 to instruction 3. In other words, instruction 3 cannot be scheduled for execution

until both instructions 1 and 2 have completed. There is not much that the instruction scheduler can do in this case—changing the order of instructions 1 and 2 is possible but would offer no benefit. The one delay slot is inevitable.

A directed acyclic graph (DAG) can be used to illustrate these dependencies. The graph from the example above is:



The graph from a more interesting example is:



Here, an optimal schedule is 4, 1, 2, 5, 3. There are no additional delay slots needed.

The DAG can be generalised to deal with more complex delay patterns by labelling each arc with the number of time slots before the result becomes available. In the situation described above, all arcs would be labelled with a "1".

Performing the scheduling is another NP-complete problem, so a heuristic approach has to be used. Many different algorithms capable of producing good results exist and are often based on *list scheduling* [4–6].

Note that there is a complex interaction between register allocation and instruction scheduling. A simple example is shown in the compilation of the code `a=1; b=2; c=3;`. The code after register allocation may have the form:

```
r1 = 1
a = r1
r1 = 2
b = r1
r1 = 3
c = r1
```

and scheduling these instructions might well involve the introduction of several delay slots. There is a dependence from instruction 1 to instruction 2 and then an

antidependence from instruction 2 to instruction 3. Removing this antidependence can be done by making use of more registers:

```
r1 = 1
a = r1
r2 = 2
b = r2
r3 = 3
c = r3
```

Here, instruction scheduling will be more successful. But the use of more registers may not always be possible. This inevitably raises difficult questions about the order in which register allocation and instruction scheduling is performed. Unfortunately there is no universal and guaranteed answer.

8.5.2 Other Hardware Features

There comes a time in the development of a compiler when it is sensible to examine carefully the hardware characteristics of the processor and to ask how these features could be used effectively by the compiler to produce high-quality code. Some of these features will be common to a wide range of processors while others will be very processor-specific. And these features will have to be used in such a way as to support the aims of the compiler. Is the compiler optimising for target code speed, or size, or power consumption, or something else, or a combination?

It may be helpful to look at a few machine features and consider how they might be used.

8.5.2.1 Special-Purpose Instructions

The processor may support some special instructions, maybe designed for particular applications or high-level language constructs. For example, function or subroutine call and return instructions may be available and it is likely that the use of these instructions by the compiler may result in better code. However there may be a trade-off. Perhaps the machine instructions do not quite match the function call mechanism already designed for the language and its compiler. Is it worth redesigning to suit the hardware or is it better to make the use of the fancy instructions suit the design by adding further code? Or is it better to avoid the use of the fancy instructions completely?

At some point it is sensible to look through the whole instruction set of the target machine to ensure that all instructions have been considered for use. Are there instructions designed to perform some special-purpose computation that could be

generated by the compiler? This is more likely to occur with CISC architectures. For example, should the polynomial evaluation instruction be generated by a compiler (see Sect. 8.1.1.1)? There is obviously no need to be able to generate all instructions but there may be cases where the code can be improved by the sensible use of these special-purpose instructions.

These special-purpose instructions can be generated by the code generator explicitly detecting the special cases. They can also be generated semi-automatically by some of the techniques used for automated code generator construction (see Sect. 8.6). Also a peephole optimiser (see Sect. 8.5.3) can include rules for their generation.

Almost all modern computer systems, apart from the smallest of embedded systems, include some form of parallel execution. The ideas of parallelisation have already been discussed as optimisation at the IR level, independent of the target machine. Finding independent processes to schedule on homogeneous or heterogeneous sets of processors is something best done before code generation. But there are code generation issues involved here too. Are there special instructions or system calls for managing the multiple processes and processors. How is the operating system involved?

This issue is further complicated by the fact that code may need to be generated for more than one processor architecture. For example, it may be possible for the compiler to generate code for the graphics processing unit (GPU) of a computer system. The architecture of the GPU will offer highly parallel computation, maybe with a huge number of individual parallel processing elements and it is well suited to array manipulation [7]. Furthermore, the GPU computation may be able to proceed in parallel with the conventional processor.

8.5.2.2 Types of Memory

Computer systems provide memory with a wide range of speeds and characteristics. There is a *memory hierarchy*. Processors have a small number of very fast registers, maybe cache memory, then main memory and finally disc, external or network storage. The average programmer will have little direct control of how the registers, cache and main memory are used during the execution of a program but the compiler writer has much greater control. Good register allocation can make a great deal of difference to program efficiency. Similarly, effective use of the cache (or separate instruction and data caches) can improve performance a great deal. Caches effectively act as high-speed memories interfacing between the very fast processor and the much slower main memory. It makes sense to attempt to keep as much of the frequently accessed instructions and data in the cache and the compiler may have some control over this.

A good knowledge of how the cache works on the target machine is of course essential. Consider a short program loop. The first time this loop is executed the code is loaded from main memory into the cache and then executed from there. The next and subsequent times the loop is executed there is no need for the cache to be

reloaded. Because the loop is small it fits completely in the cache and execution is fast. Similar considerations apply to data. A block of data from contiguous locations in main memory is loaded into the cache, making subsequent access to *all* that data very fast. The compiler can have some control over program and data locality of reference. For example, it may be possible to split a loop into two separate loops, one executed after the other, in order to get the loop to fit in the cache. Data cache operation is particularly relevant to array manipulation. Consider the two fragments of C code:

```
total = 0;
for (i=0; i<N; i++)
   for (j=0; j<N; j++)
      total += a[i][j];
```

```
total = 0;
for (j=0; j<N; j++)
   for (i=0; i<N; i++)
      total += a[i][j];
```

Because C multi-dimensional arrays are stored so that the rightmost subscript varies most rapidly as successive storage locations are accessed, the code on the left has greater data locality in accessing array `a`. More data will be found in the cache and so the code on the left will probably run more quickly than the code on the right. The compiler should be aware of this issue and automatically modify this sort of code.

Similar considerations may apply at a different point in the memory hierarchy, in the interface between main memory and the virtual memory system. The performance of the virtual memory system may be affected by the locality of data access. For example, large arrays too big to fit in main memory so are partially held in virtual memory on external storage, may be accessed very much more slowly if data access occurs at widely scattered locations. The compiler may have a role to play here too.

8.5.3 Peephole Optimisation

Code optimisation in a compiler can take place at the intermediate representation stage and throughout the code generation. Additionally, there are some optimisations that are best applied at a late stage of compilation, once target code has been generated. One such optimisation is called *peephole optimisation*, it is simple to implement and it can be very effective [8].

This optimisation scans through the generated code looking for instances of machine instructions that can be replaced by more efficient instructions (*machine idioms*). For example, a machine instruction of the form `add 1,r2` could be replaced with the instruction `inc r2` possibly saving execution time and program memory. The patterns can cover multiple instructions too. So for example instruction pairs of the form `store r1,address` immediately followed by `load r1,address` can be transformed to simply `store r1,address`.

The action here is simple. The generated code is scanned using a set of patterns and corresponding replacements. This optimiser examines short sequences of target machine instructions, through a *peephole*. It replaces single instructions or groups of

instructions by sequences that are shorter or execute faster. The process of passing the peephole over the machine code is repeated until no further improvement to the code is made.

The peephole optimiser is capable of performing a wide range of optimisations. Redundant instructions can be removed. For example, adding zero to a register can probably be removed, assuming that subsequent instructions are not relying on the setting of the condition code. Expensive instructions can be replaced by cheaper instructions, for example replacing multiplication by a power of two by a shift, replacing jump instructions having another jump instruction as their target by a single jump, generating a powerful CISC instruction from a sequence of simpler instructions and so on.

Generating the set of patterns to be matched is central to the success of this technique. One can adopt an obvious hand-generation approach of examining examples of generated code to find substitutions, add these to the table of patterns, and repeat until satisfied. Automated approaches have been used too. Exhaustively searching all possible single instructions, pairs of instructions and so on and determining whether there is a better replacement is an attractive approach [9]. Separating the code for performing the matching and replacement from the data defining the target architecture patterns is a good approach [10].

More systematic approaches are possible too. It is possible to convert the input machine code into some very low-level intermediate representation, match that with the same form of low-level intermediate representations of the target machine instructions and produce an output stream of machine code. And this can form the basis of a complete instruction selection process where the compiler's IR is translated into this low-level intermediate representation.

Peephole optimisation has also been used successfully on intermediate representations [11].

8.5.4 Superoptimisation

An important and interesting question is whether it is possible to determine whether a piece of code can be optimised more. Is it the best possible according to some pre-specified criteria? Is it possible to find the best code for a particular algorithm? The answer is a tentative and qualified "yes".

A tool called the *superoptimizer* [12] was developed to at least partially answer these questions. A simple C function was defined and an exhaustive search for short machine code sequences semantically equivalent to the original C code was performed. Semantic equivalence was assessed by automatically testing with various inputs and finally using hand-verification. This approach produced the "best" code for the chosen C function and, interestingly, it is shorter than that produced by various C compilers or by experienced assembly language programmers.

Unfortunately, because of the use of exhaustive search, this approach is not feasible for code sequences of more than a handful of machine instructions. Many

instructions and addressing modes together with other hardware complexities make an enormous number of possibilities to try. Inevitably, there are also issues concerning the definition of "best"—how are these code fragments evaluated? Measuring execution speed is difficult, especially on architectures supporting caches, pipelines and so on. The original work was done using the instruction count as the optimisation metric. Despite its limitations, this approach is useful in circumstances where short code sequences have to be designed for subsequent generation by a compiler for specific tasks such as the function call, performing 64-bit arithmetic on a 32-bit machine, and so on.

8.6 Automating Code Generator Construction

Good code generators are large and complex pieces of software. Writing them is hard work. Writing lexical and syntax analysers has been made much easier by the availability of generator tools working from a formal syntax specification. Can similar tools be developed for the construction of code generators using machine descriptions?

Several useful code generator generator tools have been developed generally using the same approach of performing pattern matching on intermediate representation trees or data structures derived from trees. Use of these tools tends not to be straightforward mainly because of the complexity of constructing a large number of patterns defined by the target hardware architecture. Furthermore, the inevitable ambiguity in these patterns has to be handled appropriately so that a pattern match resulting in near-optimal code is generated. One-off code generators are usually written by hand, but code generator generators become particularly useful when a front-end has to be retargeted to multiple target machines.

One way to think about the operation of a code generator generator is to consider using a peephole optimiser on the intermediate representation, transforming sequences of IR instructions into target machine instructions. Using a simple "search and replace" pattern matcher is feasible, but the generated code will be disappointing and there is a danger that all the input will not be matched. The extreme case is to have a set of patterns, one for each IR instruction, so that each IR instruction is translated in isolation into a sequence of target machine instructions. This will work, but will generate very poor code.

There are several ways in which this process of translating a sequence of intermediate instructions into target machine instructions can be formalised. An early approach, called the *Graham-Glanville approach*, was based on the idea of parsing using a context-free grammar [13]. A set of context-free rules is used to perform matching on the intermediate representation and on each match appropriate target machine code can be output. The code generator is being implemented as an LR(1) parser. But there are complications. There is inherent ambiguity in this process so steps have to be taken for the parser to deal with this ambiguity.

Other approaches use an attribute grammar approach to the specification of parsing and code generation rules. Attributes add semantics to the rules [14–16]. Dynamic programming can be used to resolve the ambiguity and a system called *twig* was developed [17]. This system performs pattern matching on the parse tree. A similar system called *BURS* (Bottom-Up Rewrite Systems) was also based on dynamic programming, improving performance by generating table-driven code generation algorithms when the code generator is constructed rather than when the code generator runs [18]. Further developments produced the *BURG* tool [19].

8.7 Conclusions and Further Reading

If quality code is not required, writing a code generator is fairly easy. But writing a good code generator can be very, very hard. The coding certainly requires an in-depth knowledge of the target architecture so that detailed processor manuals should be at hand. Careful planning is definitely required.

The instruction selection phase of code generation has been well studied and effective algorithms are widely available. Appel [3] presents detailed algorithms for tree pattern matching and introduces techniques for the use of dynamic programming. Instruction selection is covered in depth in [20].

Register allocation is also a well-researched area. Chaitin [21] is the first work describing register allocation by graph colouring and Chow [22] is another important reference. Mogensen [23] provides a good overview of the process and Muchnick [5] presents an in-depth study.

Managing function (or procedure, subprogram, subroutine or method) call needs careful reading of the language definition. Argument passing can be troublesome. Call by value is a traditional approach where the argument variable defined in the called function is initialised with the *value* of the corresponding argument used in the call. This is the approach adopted by C and many other languages and it is usually easy to implement. Because addresses can be manipulated by C programs, functions can update variables in the caller's environment by forcing the passing of the address of a variable rather than its value. Sometimes this passing of addresses is done in default (call by reference) and the compiler may have to check that the actual parameter used is a variable name rather than an expression, single numerical value, etc.—it has to be something that "has an address". Some languages offer more complex argument passing mechanisms such as ALGOL 60's call by name [24]. It is worth looking at these argument passing mechanisms in books on programming language design or comparative programming languages. For example, see [25]. Function call and return should be implemented, if at all possible, to make the overhead really small. If the user of the language discovers that call and return are expensive, then functions may not be used nearly as much as they should.

Looking at the documentation of real compilers can provide an excellent insight into what compilers and specifically code generators can do. Examining the nature of the different code optimisation features available shows how far this aspect of com-

piler design has advanced. Many of these compilers allow the selective activation of different optimisations, and it is interesting to examine the effect that these individual optimisations have on the target code. For example, the GCC (GNU Compiler Collection) project has extensive documentation, the compiler source code is available, and it is used widely. It incorporates front-ends for several high-level languages and has been targeted to a very wide range of machines [26].

Instruction scheduling is another well-studied topic and much has been published. Detailed information is available in [5]. Many studies have also been made concerning the combination of code generation tasks. For example, [27] examines the effect of register allocation combined with peephole optimisation and [6] covers instruction selection and peephole optimisation.

Superoptimisation is not a mainstream optimisation technique, but is useful for some specific smaller scale tasks. The *GNU Superoptimizer* (used for the project described in [28]) provides a good framework for experimentation with the technique.

Exercises

8.1 Which target machine addressing modes would you find useful when designing a code generator for DL? Produce a short list of generic machine instructions that would be essential for DL. Are there additional instructions that would be useful?

8.2 Design a "dream" processor to be the target of a C (or any other language) compiler. Don't feel constrained too much by today's hardware limitations. Write an emulator for this processor.

8.3 Design a low-level tree for DL and generate a set of tree patterns for a simple target machine. Try generating code using tree pattern matching.

8.4 Write a register allocator, operating on DL's intermediate code.

8.5 Generate C code from DL's intermediate representation. This should give you a fast way of producing a complete DL compiler.

8.6 Try writing a peephole optimiser for DL's intermediate code. Might it make sense to use some form of SSA-based IR?

8.7 Sometimes the target machine will not provide native support for some data types required by the source language. For example, a small target processor may have no floating point instructions. Consider how to provide floating point operations in the target code. The key issue here is not the code to do the operations—instead it is the code needed to *call* or somehow include the floating point code.

8.8 Design function call and return instructions for a simple machine, appropriate for use by a DL compiler. Do you need to dedicate some registers for specific purposes?

8.9 Write a non-optimising code generator for DL targeting a real or virtual machine of your choice. Now make it optimise using a variety of machine-dependent techniques.

References

1. Payne M, Bhandarkar D (1980) VAX floating point: a solid foundation for numerical computation. SIGARCH Comput Archit News 8(4):22–33
2. Lindholm T, Yellin F (1997) The Java virtual machine specification. The Java series, Addison-Wesley, Reading
3. Appel AW (2004) Modern compiler implementation in C. Cambridge University Press, Cambridge
4. Aho AV, Lam MS, Sethi R, Ullman JD (2007) Compilers – principles, techniques and tools, 2nd edn. Pearson Education, Upper Saddle River
5. Muchnick SS (1997) Advanced compiler design and implementation. Morgan Kaufmann Publishers, San Francisco
6. Cooper KD, Torczon L (2011) Engineering a compiler, 2nd edn. Morgan Kaufmann, San Francisco
7. Mittal S, Vetter JS (2015) A survey of CPU-GPU heterogeneous computing techniques. ACM Comput Surv 47(4):69:1–69:35
8. McKeeman WM (1965) Peephole optimization. Commun ACM 8(7):443–444
9. Davidson JW, Fraser CW (1984) Automatic generation of peephole optimizations. In: Proceedings of the ACM SIGPLAN '84 symposium on compiler construction, Montreal, Canada, pp 111–116. Published as ACM SIGPLAN Notices 19:6
10. Davidson JW, Fraser CW (1980) The design and application of a retargetable peephole optimizer. ACM Trans Program Lang Syst 2(2):191–202
11. Tanenbaum AS, van Staveren H, Stevenson JW (1982) Using peephole optimization on intermediate code. ACM Trans Program Lang Syst 4(1):21–36
12. Massalin H (1987) Superoptimizer – A look at the smallest program. In: Proceedings of the second international conference on architectural support for programming languages and operating systems (ASPLOS-II), Palo Alto, California, pp 122–126. Published as ACM SIGPLAN Notices 22:10
13. Glanville RS, Graham SL (1978) A new method for compiler code generation. In: Proceedings of the 5th annual symposium on principles of programming languages, pp 231–240
14. Ganapathi M, Fischer CN (1982) Description-driven code generation using attribute grammars. In: Proceedings of the 9th annual symposium on principles of programming languages, Albuquerque, NM, pp 108–119
15. Ganapathi M, Fischer CN (1984) Attributed linear intermediate representations for retargetable code generators. Softw Pract Exp 14(4):347–364
16. Ganapathi M, Fischer CN (1985) Affix grammar driven code generation. ACM Trans Program Lang Syst 7(4):560–599
17. Aho AV, Ganapathi M, Tjiang SWK (1989) Code generation using tree matching and dynamic programming. ACM Trans Program Lang Syst 11(4):491–516
18. Pelegri-Llopart E, Graham SL (1988) Optimal code generation for expression trees: an application of BURS theory. In: Proceedings of the 15th annual symposium on principles of programming languages, pp 294–308
19. Fraser CW, Henry RR, Proebsting TA (1992) BURG – fast optimal instruction selection and tree parsing. ACM SIGPLAN Not 27(4):68–76
20. Blindell GH (2016) Instruction selection: principles, methods, and applications. Springer, Switzerland
21. Chaitin GJ (1982) Register allocation and spilling via graph coloring. In: Proceedings of the ACM SIGPLAN '82 symposium on compiler construction, Boston, Mass, pp 98–105. Published as ACM SIGPLAN Notices 17:6
22. Chow FC, Hennessy JL (1990) The priority-based coloring approach to register allocation. ACM Trans Program Lang Syst 12(4):501–536
23. Mogensen TÆ (2011) Introduction to compiler design. Undergraduate topics in computer science. Springer, Berlin

24. Naur P (1963) Revised report on the algorithmic language ALGOL 60. Commun ACM 6(1):1–17
25. Sethi R (1989) Programming languages – concepts and constructs. Addison-Wesley Publishing Company, Reading
26. Free Software Foundation (2016) GCC, the GNU compiler collection. https://gcc.gnu.org/
27. Davidson JW, Fraser CW (1984) Register allocation and exhaustive peephole optimization. Softw Pract Exp 14(9):857–865
28. Granlund T, Kenner R (1992) Eliminating branches using a superoptimizer and the GNU C compiler. In: Proceedings of the ACM SIGPLAN '92 conference on programming language design and implementation. San Francisco, California, pp 341–352, June 1992

Chapter 9
Implementation Issues

This book has concentrated on a traditional and intuitive view of a compiler as a program to translate from a high-level source language to a low-level target machine language, with a potentially visible intermediate representation between a front-end and a back-end. This form of compiler is in essence specified by the source and target languages and also by the language in which the compiler should be coded. But this book has also stressed that the view of a compiler as a single, monolithic piece of code is not helpful. Instead, regarding it as a collection of phases, at least by separating a front-end from a back-end, is very helpful. These issues become particularly important when considering a strategy for a programming language implementation project.

9.1 Implementation Strategies

Careful choice of a strategy, maybe by making use of software already available, can greatly reduce the effort required to produce a programming language implementation.

Suppose that an implementation of a high-level language called L is required for target machine M. The obvious way of approaching this task is to code a complete compiler reading in programs written in L and producing M's machine code as output. We will worry about implementation languages and also on which machine the compiler is to be implemented later in this chapter. We have already seen that it makes sense to divide the task into two with a front-end and a back-end, and the choice or design of an intermediate representation is up to the implementer.

What else does the implementer need to know? Does the generated code have to be highly optimised? Does the compiler have to be particularly fast? Are there other hardware or software constraints on the project? How long is the project supposed to take? How many people are available? For complex source languages and target

architectures, and to generate highly optimised code, the project inevitably becomes huge, undoubtedly requiring a sizeable programming team.

Fortunately, there are many ways in which the compiler for L to M can be written with considerably less work. It may be possible to make use of existing software. There are many high-quality open-source compiler projects easily accessible on the internet and these may be able to offer code or complete programs or packages to help in the project. Also various existing software tools may help in the development process. Consider some examples of ways in which the development process can be simplified:

- It is obviously worth checking first whether a suitable L to M compiler already exists. If not, is a front-end for L available, generating some form of IR? Then the project just involves the coding of a back-end for M. Similarly, does an appropriate back-end for M exist whose input is a suitable IR for L? Writing half a compiler is considerably easier than writing a whole compiler.
- If highly optimised code for M is *not* required, then it may be possible to make considerable simplifications to the coding of the back-end. A code generator can translate the IR, statement-by-statement, into target machine code, with little regard for context. Similarly, code may also be generated directly from the tree generated by the syntax analyser, node by node. Writing this type of code generator is not too difficult. The process can be further simplified by generating assembly language rather than binary object code modules (or equivalent) and then using the system's assembler on M to generate binary modules.
- It may be possible to avoid completely the generation of M's machine or assembly code by generating target code in a high-level language already implemented on the target machine. For example, it may be possible for the compiler for L to generate C code and that code passed through an existing C compiler to produce the target machine code for M. There are significant potential advantages here. Not only is less work involved (generating C from the IR is probably much easier than generating target machine code) but also if the C compiler optimises well, significant optimisation efforts in generating the C may not be necessary.
- A related approach is to avoid the need of a code generator completely. The IR can be *interpreted*, instruction by instruction. Writing an interpreter is likely to be very much simpler than writing a code generator. The obvious disadvantage is of course the difference in execution efficiency, but there are many applications where this would not be an issue. We re-examine this issue in Sect. 9.1.3.
- It always makes sense to use the right tools. Use a sensible implementation language (see Sect. 9.1.2) and make use of compiler-generating tools where appropriate (see [1] for a comprehensive summary).

An important question here is whether the development of the compiler has to be carried out on machine M. Does M have a good range of software development tools already available? Would it be better to develop the compiler on a completely different machine, and then somehow transfer it to M?

9.1.1 Cross-Compilation

The machine M may already provide a good software development environment with appropriate compilers, debuggers, editors and so on, and the new language L is not central to this development process (it has not yet been implemented). In this case, developing the compiler on M is perfectly sensible. However, it may be that M has limited existing software infrastructure. It could have a newly designed processor or its system software could lack the features of a versatile operating system. Here, developing the new compiler software on M would not be appropriate and another existing machine, say M_1, with full software support and perhaps with a different architecture to M could be used instead.

Making this work is not too difficult. The compiler for L is developed on M_1. It runs on M_1 but when this compiler is given a program written in L as input, it generates machine code for machine M. This generated code is transferred to machine M where it can run natively on M's hardware. This process is called *cross-compilation* and it can have a particularly important role to play in the implementation of a language on a new machine.

This approach is used widely. For example, M could be a small, embedded processor, lacking in software or hardware support (no external secondary storage, restricted main memory and so on), not sufficiently powerful to host a compiler. Note that in this scheme two machines are being used to run programs written in L. In Sect. 9.1.2 we will see how this step of cross-compilation can be used to produce a complete implementation of L on M but this of course may not always be the aim, particularly when M is a machine with limited capabilities.

9.1.2 Implementation Languages

The choice of a programming language in which to implement a compiler is very important. The implementation language should have several characteristics:

- Obviously, the language should already be available on the machine on which the compiler is to be developed. Implementing a compiler for one's favourite implementation language in order to write the compiler actually required is rarely sensible.
- Writing a compiler is a big project and so the implementation language must support the manageable development of big software projects, probably using a team of programmers. This rules out low-level languages.
- The language should offer good support for the types of computation performed by the compiler such as character handling, data structure management and so on. Pattern matching is a potentially useful feature.
- If compiler generation tools are being used they may influence or indeed force the choice of a particular language.

- Ideally, the language should have an efficient implementation. This will help with the production of an efficient compiler.
- The implementer should of course be happy programming in the chosen language.

In addition, there are good reasons for coding a compiler for a language L in L itself. At first sight this may seem odd or even impossible, but the advantages are significant.

Consider the scenario introduced in the previous section. An implementation of L is required on machine M, but the development of the compiler takes place on machine M_1 because M has an inadequate or inappropriate development environment. But suppose also that the aim of the project is to transfer the compiler to M in due course so that the development of software in L can continue on M with no further need for machine M_1. A good way of achieving this aim is to follow these steps resulting in the *bootstrapping* of the compiler:

1. On machine M_1 write a compiler translating L into M's machine code. Use language L for the implementation.
2. This compiler is just a program written in L. Use the compiler just written on machine M_1 to compile the source code of the compiler into M's machine code. All being well, this will be the last time machine M_1 will be needed in this process.
3. Transfer this machine code to machine M and when this code runs on M it will translate from L to M's machine code. It is, after all, a compiler for language L for machine M.
4. A sensible test is to use this compiler on M to translate the source code (in L) of the compiler to M's machine code and check that the generated code is the same as that already running as the compiler on M.

This approach to compiler bootstrapping has been used many times. There is considerable flexibility derived from dividing the compiler into two or more parts and having a simple and well-defined intermediate representation interfacing the parts.

Variants of this scheme are possible. If an implementation of L is required rapidly (maybe machine M_1 is only available for a short time) then this implementation plan can be modified. Assuming that the compiler has been structured in the traditional way with a front-end and a back-end communicating via a sensible intermediate representation, it may be possible to follow these steps:

1. On machine M_1 develop a front-end for L, written in L, generating an IR version of L programs. Maybe this is already available as part of another implementation of L.
2. On machine M develop an *interpreter* for the IR. This can be written in any language available on M, even assembly language if the IR is simple. This interpreter allows programs coded in the IR to be run interpretively on machine M.
3. On machine M_1 produce the IR version of the front-end of the L compiler. This can be done by passing the source code of the front-end "into itself" and it will generate the IR version.

4. Transfer this IR version to machine M and it can be "run" on the interpreter, yielding an interpretive L front-end on machine M. This allows L programs to be run interpretively on machine M. Machine M_1 is no longer required.
5. If necessary, a code generator can then be written on machine M, presumably in the language L, generating M's machine code from the IR. By running the compiler interpretively, a M machine code version of the complete compiler can be produced, finally yielding a complete natively running compiler on machine M.

In this approach to implementation the IR has a special role. In past chapters, we have seen how the IR is used as a convenient interface between the front-end and the back-end of the compiler and how it supports machine-independent optimisation. Here, the IR plays a central role in the implementation and specifically in the porting of a compiler. In compiler projects where portability is a key concern, the use of a small and simple IR makes the development of an interpreter for the new target machine straightforward.

9.1.3 Portability

A compiler can be made portable by writing it in a portable fashion in a portable language. So, for example, a compiler written in C for the language L running on machine M_1 producing M_1 code can probably be ported to machine M_2 without too much difficulty but it would, of course, still be generating code for M_1. Porting a compiler to a new machine is usually taken to mean the *retargeting* of the compiler to the new machine (i.e. making it generate code for the new machine) as well as getting it to run on the new machine. Making this process easy helps make the language L portable too.

We have already seen how the use of the intermediate representation can facilitate portability and also how interpreting rather than code generating the IR can help. It is also possible to use the same IR for multiple source languages and multiple target machines. A range of compiler front-ends can be developed for different source languages, all generating the same IR. Various compiler back-ends can be developed, each taking as input this common IR but generating code for different machines. If this kit of front-ends and back-ends handles n source languages and m target machines, in effect $n * m$ complete compilers have been developed at the cost of coding just $n + m$ "half compilers". This is clearly a very attractive proposition and forms the basis of many portable compiler systems. One of the earliest practical examples was the *Amsterdam Compiler Kit*, using a stack-based intermediate code called EM, generated by the front-ends, which was optimised and then passed on to one of a range of target code-generating back-ends [2]. The *GNU Compiler Collection* is another example where multiple front-ends generate RTL (Register Transfer Language) and multiple back-ends generate target code from the RTL [3]. The *LLVM Compiler Infrastructure* project also has multiple front-ends and back-ends interfaced via a common intermediate representation [4].

The idea of a common intermediate language for compilers was proposed in the early days of computing [5]. The design of the IR has always been a central concern in the development of compilers. The need for a simple IR to support compiler portability sometimes conflicts with the requirements of an IR to support optimisation and effective code generation. Therefore, there may be a case for two distinct intermediate representations. For example, the BCPL compiler described in [6] uses INTCODE for bootstrapping and OCODE as the interface between front-end and back-end.

The details of the process of porting a compiler can become quite complex because there are so many ways in which it can be done. Many distinct steps may be involved and systematic testing is particularly important. Ensuring that "when the compiler compiles itself it produces itself" is a good test, but it is certainly not a guarantee of correctness.

9.2 Additional Software

The compiler is only a part of the programming language implementation story. It is very likely that more software is required to complete the implementation project.

Depending on the nature of the language being implemented, as well as the target machine, some form of *runtime library* will almost certainly be required so that the running code from the compiler can communicate with the operating system or in some systems, directly with the hardware. For example, the C statement `printf("Count = %d\n",i)` is translated by the C compiler into a function call with two arguments, but the compiler does not need to be concerned with the code of the `printf` function. The code for all these support functions is contained in a library or libraries, provided with the compiler implementation. It may be possible to write at least part of this runtime library in a high-level language, thus making it portable and independent of the target machine. But it may be necessary to implement some aspects in assembly language, making those parts target machine-dependent.

Particularly when the target machine is a small, embedded system, maybe running without a conventional operating system, the executing program generated by the compiler will probably need access to aspects of the bare machine such as i/o ports, particular memory locations or special machine instructions. Depending on the language being used, these tasks may be done most easily via function calls. A small and specialised runtime library will be required here.

Code may also be required to perform tasks concerned with the setting up of the execution environment, and this can be placed conveniently in the runtime library. Memory may need to be allocated (for example, for the runtime stack), default i/o streams opened, error handling set up. The modules making up the code of the target program will probably be set up so that the entry point is the start of the initialisation code in the runtime library, and once that has executed, control is transferred to the main program in the generated code.

The need for a runtime library implies that there has to be some mechanism for linking separately compiled modules together to produce a single executable binary file. A program (the *linker*) has this role. It is usually part of the "systems software" on the target machine and so it may be of little direct concern to the compiler writer. The modules produced by the compiler just have to be in a format acceptable to the linker.

Debuggers are often provided to help the software developer remove software errors. A symbolic debugger allows the programmer to access information about a running program using the names that were used in the source code. Variables can be examined and set, breakpoints can be specified, the function call stack can be examined and so on. Symbolic information has to be output by the compiler and placed in the object files created by the code generator. Standardised object file formats exist and are widely used. Debuggers may have difficulty when applied to optimised code. After the optimisation processes have been performed by the compiler, there is no longer a direct statement-by-statement or variable-by-variable match between the source and object codes. Statements can be reordered, modified or even removed and making the debugger somehow reverse these transformations in order to display useful information is a difficult problem [7–9].

Code should be left in the compiler to enable intermediate code representations, compiler data structures and so on capable of being output if required. These may help with the debugging of some difficult source program issues and will certainly help with the debugging of the compiler itself. The compiler can also include facilities for the optional generation of runtime checking code such as array bound checking. Features like this can save a great deal of time in the software development process.

Program development environments combine various utility programs such as language-aware editors, debuggers, code formatters, compilers, code cross-referencers, testing frameworks and so on to form an easy to use, integrated tool for constructing software.

Finally, the need for *testing* of the compiler cannot be understated. This issue has been mentioned throughout the text, referring particularly to the testing of individual compiler components but it is vital that the *whole* compiler is tested thoroughly too. The field of software engineering gives good guidance on the development of a testing strategy. A large, well-organised suite of test programs is required, with and without errors, each having their expected corresponding output and they are all used each time a new version of the compiler is produced. The test programs should include programs designed to test specific aspects of the compiler and cover difficult aspects of the language. These test suites are challenging and time consuming to produce but it may be possible to make use of existing test suites which have been made publicly available. Making the compiler compile itself is a useful but not a conclusive test for correctness. Testing is a complex area but it cannot be avoided.

9.3 Particular Requirements

This book covers just the basic techniques of compiler construction. These essential techniques are generally applicable to most high-level languages but there will be special approaches required for some implementations. There are so many issues that could be considered here, but only a few have been selected.

The development of code for *embedded systems* is an important area, one where code optimisation may be vital. The target machine may be very limited in functionality (e.g. a small, simple processor with limited memory, particularly in a consumer electronics device where cost is a key factor). Optimisation may be concentrated on code and/or data size (limited memory), on execution speed (particularly to deal with real time constraints) or on power consumption (battery-powered devices). For these applications, it may not matter that the compiler takes significantly longer to run in order to implement these optimisations. The superoptimiser may have a role to play here. It may be possible to compress the code and uncompress it as it runs in order to save program memory (at the cost of execution speed). Note that with most of these embedded applications, cross-compilation is used. Hosting the compiler on the embedded system itself makes little sense.

Some high-level languages expect a particular implementation plan. For example, some languages were designed with an interpreted implementation in mind. Java is a good case. The traditional implementation compiles the Java source code into code for a virtual machine (the *Java Virtual Machine* [10], where programs are written in *Java bytecodes*) and a separate interpreter program reads this virtual machine code, simulating its execution, hence running the Java code. One of the important advantages offered by this approach is that it allows the implementation of *dynamic class loading* where a running program can call for the execution of a different class which may reside on the local machine or a remote machine somewhere on the local network or on the internet. The remote class is loaded in the form of a file containing code for the Java Virtual Machine and its execution can start using the same interpreter.

The disadvantage of this implementation strategy is of course the execution efficiency overhead introduced by the interpretation rather than the native running of the code. Many techniques have been developed to improve the efficiency. In particular, *just-in-time compilation* is now often used to translate the Java bytecodes automatically into target machine code, just before the code's execution. If the Java system encounters bytecodes that are likely to be executed repeatedly, then it is important that the translation to target machine code is done just once in advance. Interpreting some of the code and running the rest natively is possible. Such systems can offer considerable performance gains.

9.4 The Future

In some respects programming language implementation has been a remarkably stable field of computer science. The theoretical foundations of grammars and parsing informed the development of lexical and syntax analysers and today's implementations of these phases are very similar in structure to those written decades ago. But target machines and compiler back-ends as well as programming languages have all changed a great deal. Developments in instruction selection, control and data dependence representations, optimisation, register allocation and so on have had a significant effect and the huge changes in processor and memory design, influenced by the needs of the compiler writer, have all resulted in extremely efficient implementations of powerful high-level languages.

Compiler development is driven primarily by advances in programming languages and in computer hardware. Changes in compilers tend to be incremental. Processor speeds and capabilities will continue to increase and compilers will have to adapt to the new functionality. In order to continue following the spirit of Moore's Law there has to be an increased reliance on multiprocessor architectures, and despite many years of research, we are still not very good at the automatic parallelisation of programs written in conventional programming languages. Whether this situation is resolved by better parallelisation in compilers or the development and adoption of easy to use parallel programming languages is an interesting question. Today's programmers have been immersed in traditional sequential programming for so long that it is difficult to see how to move to thinking about the solution of problems in terms of parallel algorithms.

There is no doubt that better programming languages are needed to make the process of programming simpler and more reliable. An important driver for compiler development will undoubtedly be these developments in programming languages. Software reliability is a crucial goal. As more and more life-critical devices and computer applications are being developed there has to be technology for the production of reliable software. And reliable compilers are needed too, of course, and this is an active research area. Techniques of formal verification can be applied to compilers and such compilers have special relevance when developing code for high-integrity embedded systems [11].

Radically different computer architectures will evolve. Recent advances in biologically inspired computing and quantum computing suggest that major changes are on the horizon. What about compilers for these machines? That sounds like an interesting project….

9.5 Conclusions and Further Reading

The division of a compiler into a front-end and a back-end has major effects on the ease of implementation of the compiler on a new machine. The free availability today on the internet of many compiler components has had a significant influence

on the ease with which new compilers and other tools can be implemented. There are now extremely reliable, powerful and highly optimising compilers available in source code form, compiling a large variety of source languages to diverse target architectures.

Many of these compilers form part of a compiler kit. They are designed in such a way as to facilitate the addition of new front-ends for new source languages and include support for the porting to new target machines. They usually include detailed documentation, specifically including guidance on the addition of new front-end and back-end components. The GCC [3] and LLVM [4] projects are good examples and their porting documentation is helpful reading. If the machine to which the compiler is to be ported is similar to a machine that already has an implementation, then the task may be comparatively easy.

There are many program development environments in use throughout the software industry today. It is easy to find information about them on the web and to do some sort of feature comparison. They can make a big difference to the ease of managing software projects.

Also, information about other compiler-related software tools is easy to find. A valuable skill is an understanding of object code formats and linkers [12]. Java also is a key aspect of programming language implementation. It is itself a good language for the implementation of compilers [13, 14]. Just- in-time compilers are not confined to Java implementations. They have a long history [15].

Implementation of functional and object-oriented languages has not been covered in this book. These are huge subjects in themselves [16, 17]. And quantum computing is a fast-moving subject. A web search will reveal a great deal.

Exercises

9.1. Write interpreters in various languages for the intermediate representation for DL.

9.2. A good choice of a compiler implementation language is important. Try and find out which languages are and have been popular for the coding of compilers and make some conclusions.

9.3. Look at the LLVM and GCC documentation, particularly the information about porting to a new machine. Estimate how long it would take to produce a "quick and dirty" code generator and how long it would take for a high-quality optimising code generator.

9.4. Does the inclusion of pattern matching features make a language better-suited for use in compiler implementation? Discuss where in a compiler such features would be useful.

9.5. What makes a C program non-portable? What makes a Java program non-portable?

9.6. How well is Java suited to the implementation of applications on a small embedded processor?

9.7. The inevitable programming exercise: write a complete compiler for DL targeted to any real or virtual machine you like.

9.8. Make some predictions about what programming languages and compiler technology will look like in ten years time. Keep them somewhere safe.

References

1. German National Research Center for Information Technology (2016). The catalog of compiler construction tools. http://catalog.compilertools.net/. Accessed 20 Oct 2016
2. Tanenbaum AS, van Staveren H, Keizer EG, Stevenson JW (1983) A practical tool kit for making portable compilers. Commun ACM 26(9):654–660
3. Free Software Foundation (2016). GCC, the GNU compiler collection. https://gcc.gnu.org/
4. Computer Science Department at the University of Illinois at Urbana-Champaign (2016). The LLVM compiler infrastructure. http://llvm.org/. Accessed 26 Oct 2016
5. Conway ME (1958) A proposal for an UNCOL. Commun ACM 1(10):5–8
6. Richards M, Whitby-Strevens C (1980) BCPL – the language and its compiler. Cambridge University Press, Cambridge
7. Brooks G, Hansen GJ, Simmons S (1992) A new approach to debugging optimized code. Proceedings of the ACM SIGPLAN '92 conference on programming language design and implementation. California, San Francisco, pp 1–11
8. Hennessy J (1982) Symbolic debugging of optimized code. ACM Trans Program Lang Syst 4(3):323–344
9. Hölzle U, Chambers C, Ungar D (1992) Debugging optimized code with dynamic deoptimization. Proceedings of the ACM SIGPLAN '92 conference on programming language design and implementation. California, San Francisco, pp 32–43
10. Lindholm T, Yellin F (1997) The Java virtual machine specification. The Java series. Addison-Wesley, Reading
11. Leroy X (2016) INRIA. CompCert. http://compcert.inria.fr/index.html. Accessed 01 Dec 2016
12. Levine JR (2000) Linkers & loaders. Morgan Kaufmann, Massachusetts
13. Appel AW, Palsberg J (2002) Modern compiler implementation in Java, 2nd edn. Cambridge University Press, Cambridge
14. Watt DA, Brown DF (2000) Programming language processors in Java. Prentice Hall, Englewood Cliffs
15. Aycock J (2003) A brief history of just-in-time. ACM Comput Surv 35(2):97–113
16. Craig I (2000) The interpretation of object-oriented programming languages. Springer, Heidelberg
17. Peyton Jones SL (1987) The implementation of functional programming languages. Prentice Hall International Series in Computer Science. Prentice Hall, Englewood Cliffs

The DL Language

This appendix gives a description of the DL language used for many of the compiler examples in the book.

DL is a simple high-level language, only operating on integer data, with a syntax looking vaguely like a simple C. The syntax of DL is defined by the following BNF grammar:

```
<program> ::= <block>
            | <declarations> <block>
<declarations> ::= <declaration>
                 | <declaration> <declarations>
<declaration> ::= <variabledeclaration> | <functiondeclaration>
<variabledeclaration> ::= int <vardeflist> ;
<vardeflist> ::= <vardec> | <vardec> , <vardeflist>
<vardec> ::= <identifier> | <identifier> [ <constant> ]
<functiondeclaration> ::= <identifier> (); <functionbody>
                        | <identifier> ( <arglist> ); <functionbody>
<functionbody> ::= <variabledeclaration> <block> | <block>
<arglist> ::= <identifier> | <identifier> , <arglist>
<block> ::= { <statementlist> }
<statementlist> ::= <statement> | <statement> ; <statementlist>
<statement> ::= <assignment> | <ifstatement> | <whilestatement>
              | <block> | <printstatement> | <readstatement>
              | <returnstatement> | <empty>
<assignment> ::= <identifier> = <expression>
               | <identifier> [ <expression> ] = <expression>
<ifstatement> ::= if ( <bexpression> ) <block> else <block>
                | if ( <bexpression> ) <block>
<whilestatement> ::= while ( <bexpression> ) <block>
<printstatement> ::= print ( <expression> )
<readstatement> ::= read ( <identifier> )
<returnstatement> ::= return <expression>
<expression> ::= <expression> <addingop> <term>
               | <term> | <addingop> <term>
<bexpression> ::= <expression> <relop> <expression>
<relop> ::= < | <= | == | >= | > | !=
<addingop> ::= + | -
<term> ::= <term> <multop> <factor> | <factor>
<multop> ::= * | /
<arguments> ::= <expression> | <expression> , <arguments>
```

```
<factor> ::= <constant> | <identifier>
      | <identifier> [ <expression>]
      | ( <expression> ) | <identifier> ( <arguments> )
      | <identifier> ( )
<constant> ::= <digit> | <digit> <constant>
<identifier> ::= <identifier> <letterordigit> | <letter>
<letterordigit> ::= <letter> | <digit>
<letter> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<digit> ::= 0|1|2|3|4|5|6|7|8|9
<empty> has the obvious meaning
```

Names have to be declared before they are used. This implies that mutually recursive functions are not allowed. All functions return a single integer result.

Comments (zero or more characters enclosed between the standard C comment brackets `/*...*/`) can be inserted. DL has rudimentary support for one-dimensional arrays. The declaration `int a[3]` declares an array of three elements, referenced as `a[0]`, `a[1]` and `a[2]`. Names are scoped in the traditional way.

A simple program written in this language is as follows:

```
factorial(n);
{
   if (n==0) { return 1 }
   else { return n*factorial(n-1) }
}

int x;
{
   x=1;
   while (x<=10) {
      print(factorial(x));
      x = x + 1
   }
}
```

The tree generated for this program using the code outlined in this book has two parts. The first is for the `factorial` function.

```
N_FUNCTIONDEC - function definition chain, this:
|  N_FUNCTION - function def, number 0
|  N_FUNCTION - function def, arglist:
|  |  N_PARMS - parameter, runtime offset 0
|  |  N_PARMS - parameter, next, index is:
|  N_FUNCTION - function def, block:
|  |  N_SLIST - Statement list, this statement:
|  |  |  N_IF - if, condition:
|  |  |  |  N_EQ - eq, lhs is:
|  |  |  |  |  N_ID - identifier, runtime offset 0
|  |  |  |  N_EQ - eq, rhs is:
|  |  |  |  |  N_CONST - const, value is: 0
|  |  |  N_IF - if, thenpart:
```

```
|  |  |  |  N_SLIST - Statement list, this statement:
|  |  |  |  |  N_RETURN - return, value is:
|  |  |  |  |  |  N_CONST - const, value is: 1
|  |  |  |  N_SLIST - Statement list, continued...
|  |  |  N_IF - if, elsepart:
|  |  |  |  N_SLIST - Statement list, this statement:
|  |  |  |  |  N_RETURN - return, value is:
|  |  |  |  |  |  N_MUL - mul, lhs is:
|  |  |  |  |  |  |  N_ID - identifier, runtime offset 0
|  |  |  |  |  |  N_MUL - mul, rhs is:
|  |  |  |  |  |  |  N_FNCALL - function call, function 0 (factorial)
|  |  |  |  |  |  |  N_FNCALL - function call, arguments:
|  |  |  |  |  |  |  |  N_ARG - argument list, expression:
|  |  |  |  |  |  |  |  |  N_MINUS - minus, lhs is:
|  |  |  |  |  |  |  |  |  |  N_ID - identifier, runtime offset 0
|  |  |  |  |  |  |  |  |  N_MINUS - minus, rhs is:
|  |  |  |  |  |  |  |  |  |  N_CONST - const, value is: 1
|  |  |  |  |  |  |  |  N_ARG - argument list, next argument:
|  |  |  |  N_SLIST - Statement list, continued...
|  |  N_SLIST - Statement list, continued...
N_FUNCTIONDEC - function definition chain, next:
```

The second is for the main program.

```
N_SLIST - Statement list, this statement:
|  N_ASSIGN - Assign to variable at offset 0, expression is:
|  |  N_CONST - const, value is: 1
N_SLIST - Statement list, continued...
|  N_SLIST - Statement list, this statement:
|  |  N_WHILE - while, condition:
|  |  |  N_LE - le, lhs is:
|  |  |  |  N_ID - identifier, runtime offset 0
|  |  |  N_LE - le, rhs is:
|  |  |  |  N_CONST - const, value is: 10
|  |  N_WHILE - while, block:
|  |  |  N_SLIST - Statement list, this statement:
|  |  |  |  N_PRINT - print, expression is:
|  |  |  |  |  N_FNCALL - function call, function 0 (factorial)
|  |  |  |  |  N_FNCALL - function call, arguments:
|  |  |  |  |  |  N_ARG - argument list, expression:
|  |  |  |  |  |  |  N_ID - identifier, runtime offset 0
|  |  |  |  |  |  N_ARG - argument list, next argument:
|  |  |  N_SLIST - Statement list, continued...
|  |  |  |  N_SLIST - Statement list, this statement:
|  |  |  |  |  N_ASSIGN - Assign to variable at offset 0, expression is:
|  |  |  |  |  |  N_PLUS - plus, lhs is:
|  |  |  |  |  |  |  N_ID - identifier, runtime offset 0
|  |  |  |  |  |  N_PLUS - plus, rhs is:
|  |  |  |  |  |  |  N_CONST - const, value is: 1
|  |  |  |  N_SLIST - Statement list, continued...
|  N_SLIST - Statement list, continued...
```

The intermediate representation generated for this program is as follows.

```
factorial(1)
if (vl0!=0) goto l1
r0 = 1
return
goto l2
l1:
r2 = vl0 - 1
param r2
call factorial
r3 = r0
r1 = vl0 * r3
r0 = r1
return
l2:
r0 = 0
return

:!MAIN!(1)
vg0 = 1
l3:
if (vg0>10) goto l4
param vg0
call factorial
r4 = r0
print r4
r5 = vg0 + 1
vg0 = r5
goto l3
l4:
r0 = 0
return
```

Index