Eastern Economy Edition

Compiler Design

Santanu Chattopadhyay

PHI

COMPILER DESIGN

SANTANU CHATTOPADHYAY
Professor
Department of Electronics and Electrical Communication Engineering
Indian Institute of Technology Kharagpur

PHI Learning Private Limited
New Delhi-110001
2011

COMPILER DESIGN
Santanu Chattopadhyay

ISBN-978-81-203-2725-2

Sixth Printing . . . . . **October, 2011**

Published by Asoke K. Ghosh, PHI Learning Private Limited, M-97, Connaught Circus, New Delhi-110001 and Printed by Mudrak, 30-A, Patparganj, Delhi-110091.

To

Santana, my wife

My source of inspiration

and

Sayantan, my son

My dream

Contents

List of Figures

List of Tables

Preface

This book originated from my years of study and teaching in the field of *Compilers*. It is also based upon the interactions with the students, their feedback and survey of knowledge in this field. Compiler designing process is an amalgamation of fields like *Automata Theory, Data Structures, Algorithms, Computer Architecture, Operating System* and so on. A compiler designer must have a strong theoretical computer science background, and at the same time, be able to exploit the hardware and software platforms made available. Thus, any text on *compilers* should be able to highlight the contribution of each of these fields and introduce the readers to the challenges and techniques of the field. At the same time, the book should be concise enough to be covered in a single semester. The available tools for compiler designing should also be introduced. This book covers the subject matter keeping in mind all these issues. It has been divided into nine chapters.

Chapter 1 introduces the *compiler* and its applications to the readers. Effort has been made to provide with a large number of application areas, so that the reader can comprehend the breadth of the field. It then introduces the various phases of a compiler enumerating the responsibilities of each of them. Chanllenges faced by a compiler designer, in terms of hardware/software features of the machine and the user's requirements, have been detailed. The chapter ends with an example.

Chapter 2 presents the theory and design of the *Lexical analysis* phase of a compiler. After introducing the formal notations like *regular expressions, nondeterministic*, and *deterministic finite automata*, the book has detailed how starting with a specification of words of a language in terms of *regular expressions*, one can build the recognizers to identify the valid words of the language in an input stream. It also presents the automated tool *lex* to generate the lexical analyzers easily.

Chapter 3 presents *syntax analysis*. It first establishes the role of grammar in specifying a language, the notational conventions for grammars and some associated definitions. The chapter presents a detailed study of various types of parsers starting with the simplest ones like *recursive descent* and *LL* from the class of *top down parsing* to the advanced *bottom up* parsers like *LR, Canonical LR* and *LALR*. Since most of the compilers use *LR* parsers, special emphasis has been placed on them. The tool *yacc* has been introduced to show the automated generation of *LALR* parsers. The chapter concludes by showing how the parsers can be utilized to perform various *syntax directed transformations* of the input stream.

Chapter 4 discusses *type checking*. It introduces the problems associated with type checking and how the theory can be used to check *equivalence* of types. *Type conversion* has been presented to show how the compiler can infer automatically the necessary type conversions.

Chapter 5 introduces the usage of *symbol table* in a compiler. It also discusses various data structures that are needed based on *scope* rules of a language to store the identifiers, so that the compiler can use the information efficiently.

Chapter 6 presents the compiler's responsibilities in maintaining the runtime environment of the program under execution. It introduces the *activation records* and shows how to maintain it using different efficient data structures.

Chapter 7 details the *intermediate code generation*. This is an optional stage of a compiler and helps in retargeting the code to different architectures. It starts with various possible representations of the intermediate code. It then presents syntax directed schemes to translate the commonly available programming language constructs to the intermediate forms.

Chapter 8 enumerates the target code generation techniques. After discussing the factors affecting the code generation, it shows how to represent an intermediate language program in terms of basic blocks and then, generate the target code from the basic blocks. It also discusses the problem of *register allocation*, *cache management* and their solution strategies.

Chapter 9 is on *code optimization*. After discussing the necessity and associated problems in the optimization process, it performs a classification of optimization. It then presents different optimizing transformations like *compile-time evaluation, common sub-expression elimination* and so on. The subject of *global optimization* has been presented next. The *data flow problems* have been introduced. Solution techniques based on setting up and solving *data flow equations* have been enumerated for several data flow problems.

In this book, I have attempted to provide a full coverage of subject matter in the field of *compiler design*. It has been estimated to be covered in a single semester undergraduate course on *compilers*. If the book can be used fruitfully by the students, for whom the book has been written, I will consider my efforts to be successful. Any constructive suggestion for improving the content is most welcome.

SANTANU CHATTOPADHYAY

Acknowledgements

I must acknowledge the contribution of my teachers who have introduced me to the field of *Computer Science* and in particular, *Compilers*. Professor Dipankar Sarkar taught me the difficult course of *Formal Systems*, from which I developed a liking towards the *languages, grammars*, etc. Dr. Probal Sengupta taught us *Compilers*, and many of the discussions, particularly in the chapter on *Intermediate Code Generation* are inspired by his boardworks in the class. I am indebted to him for the crystal-clear discussions in the class which have helped me to present my ideas in a better way. I must also acknowledge the contribution of my students who, through their continuous interactions in the class and laboratory, cleared many doubts in my mind also!

My source of inspiration for writing this book is my wife, *Santana*. It is mainly because of her pushing that the book could come into existence. She has been highly supporting, understanding and encouraging throughout the period of preparing the manuscript, relieving me of many of the pressures of day-to-day life and ensuring that I spend sufficient time on the book. My son *Sayantan* always posed me many small challenges—fulfilling those made the book complete. I also acknowledge our parents (*Sukumar Chattopadhyay—father, Tara Chattopadhyay—mother, Bimalendu Mukherjee—father-in-law, Aruna Mukherjee—mother-in-law*) for bringing us up and making us what we are today.

I also acknowledge the authors of the books and research papers, from which the content of this book has been drawn. A detailed list has been provided in the Bibliography.

Thanks are also to the publishers, Prentice-Hall of India, its editorial and production team for providing necessary support to see my thoughts in the form of a book.

SANTANU CHATTOPADHYAY

Chapter 1

Introduction

Compilers have become part and parcel of today's computer systems. They are responsible for making the user's computing requirements, specified as a piece of program, understandable to the underlying machine. Hence, these tools work as an interface between the entities of two different domains—the human being and the machine. The actual process involved in this transformation is really complex. Its theoretical foundation lies on the formal definition of languages and recognizers, mostly covered in the lectures on *automata theory*. This formalism provides a solid base on which we can design several types of automated tools used in various phases of a compiler design. These tools have been detailed in later chapters. On the other hand, advances in computer architecture, memory management and newer operating systems provide the compiler designer a large number of options to try out to generate optimized code. Continuous research efforts in this direction reflect this observation. The course on compiler design is to address all these issues, starting from the theoretical foundations to the architectural issues to automated tools. In this chapter, we will introduce the job of compilation and will present its application domains, the phases of a compiler and the challenges to be handled by a compiler designer. Finally, we will illustrate the process with an example.

1.1 WHAT IS A COMPILER

A compiler is a system software that converts a source language program into an equivalent target language program. It also validates the input program to be conforming to the source language specification. If this is violated, for example, source language may stipulate that each statement be terminated by a semicolon (;) which might be violated in the program, the compiler will produce an error message to the user. For less critical violations, it may produce warnings. Thus, the overall compilation process may be depicted as in Fig. 1.1.

In the early computer systems, the concept of compilers was not present. The programs were earlier written in assembly languages, which were directly translated into machine code by hand and fed to the computer. Program sizes were also small to be handled. The thrust on designing compilers came in 1950s with the introduction of FORTRAN, the first high level language for computers. At that time, writing a compiler was assumed to be a very difficult task. However, due to the formal background from automata theory, the development

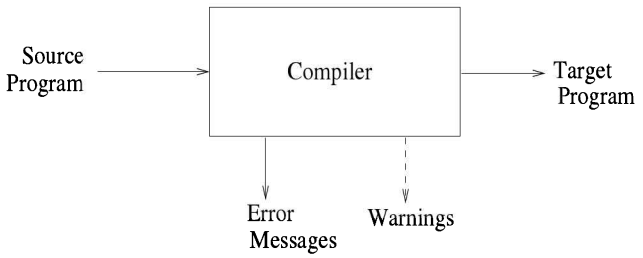

Figure 1.1: Compiler input-output.

accelerated to a great extent, that saw development of a number of automated tools for compiler design. Now, designing a new compiler for a moderately sized language is at most a term-project for students in a semester.

It is almost impossible to quantify the number of compilers designed so far. Thanks to the large number of well known lesser known and possibly unknown computer languages designed so far, and an equally large number of hardware-software platforms, on which they have been tested. However, efficient design of compilers is a must for survival of any language proposed. The case is justified by considering the early efforts to write translators for functional programming languages like *LISP*. The inefficient translators make the programs written in *LISP* to run very slowly as compared to their counterparts in imperative languages like FORTRAN, Pascal, etc. Recent advances in memory management policies particularly garbage collection, has paved the way for faster implementation of such translators and rejuvenating those languages.

1.2 COMPILER APPLICATIONS

The applications of compilers centre around its basic ability of converting a program in one language to the other. Still, we can identify the following major domains in which the application spans.

1. ***Machine code generation.*** As depicted earlier, compilers have most widely been used for converting source language program into machine understandable code. Thus, in one hand, it takes care of the semantics of varied constructs of the source language. At the same time, it also takes into consideration the limitations and specific features of the target machine. It should be noted that though the automata theory helps in the syntactic checks (that is, conformation to the grammar), it does not tell how to generate the code for a syntactically correct program. Thus, the theory of compilation also include efficient generation of machine code.
2. ***Format converters.*** On the same line comes the format converters. These tools are often needed to interface between two or more software packages. For softwares coming from different vendors, many a times the situation arises in which it is required to feed the output of one tool as input to the next one. Compatibility of input-output formats for these type of applications is one of the most essential requirements. In the absence of such compatibility, we have to use format converters which convert the output of the first software to the format acceptable to the next one. Likewise, the code converters have

also been developed to convert heavily used programs, written in some older languages, (like COBOL) to newer languages like C, C++, etc.

3. ***Silicon compilation.*** It is the process of automatically synthesizing a circuit from its behavioural descriptions in languages like *VHDL, Verilog*, etc. As the complexity of the circuit increases with reduced time-to-market, use of such automated tools are increasing by leaps and bounds. As compared to the program code generation, the optimization criteria for silicon compilation are different. The most important issues here are the area requirement, delay, power consumption and so on.
4. ***Query interpretation.*** This belongs to the domain of database query processing. In the case of database, the most important optimization criteria is the reduction in search time. Moreover, often there exists more than one evaluation sequence for such queries, and the cost of these alternatives depends, upon the situation in hand like relative sizes of the tables, availability of indexes, etc. Thus, the compiler here is responsible for generating proper sequence of operations suitable to respond to the query in the shortest possible time.
5. ***Text formatting.*** Text formatting softwares accepting an ordinary text file as input along with the formatting commands embedded within it come into this category. Typical examples of such systems are *troff, nroff, LaTex* and so on.

1.3 PHASES OF A COMPILER

To ease the process of development and understanding, a compiler can be conceptually divided into a number of phases, namely,

- Lexical analysis
- Syntax analysis
- Semantic analysis
- Intermediate code generation
- Target code generation
- Code optimization
- Symbol table management
- Error handling and recovery

It may be noted that there does not exist any hard demarkation between these modules, rather they work hand-in-hand in an intersparsed manner to accomplish the job. Figure 1.2 represents the relationship between the modules. Now, we will briefly enumerate the responsibilities of individual phases.

1. ***Lexical analysis.*** This is the interface of the compiler to the outside world. The task of this phase is to scan the input source program and identify the valid words within it. It also does the additional job of removing *cosmetics* like extra white space, comments added by the user, etc. from the program, expanding the user defined macros like `#include` and `#define` in C and reporting the presence of foreign words, that is, the words not belonging to the language of the source program, if any. It may also perform

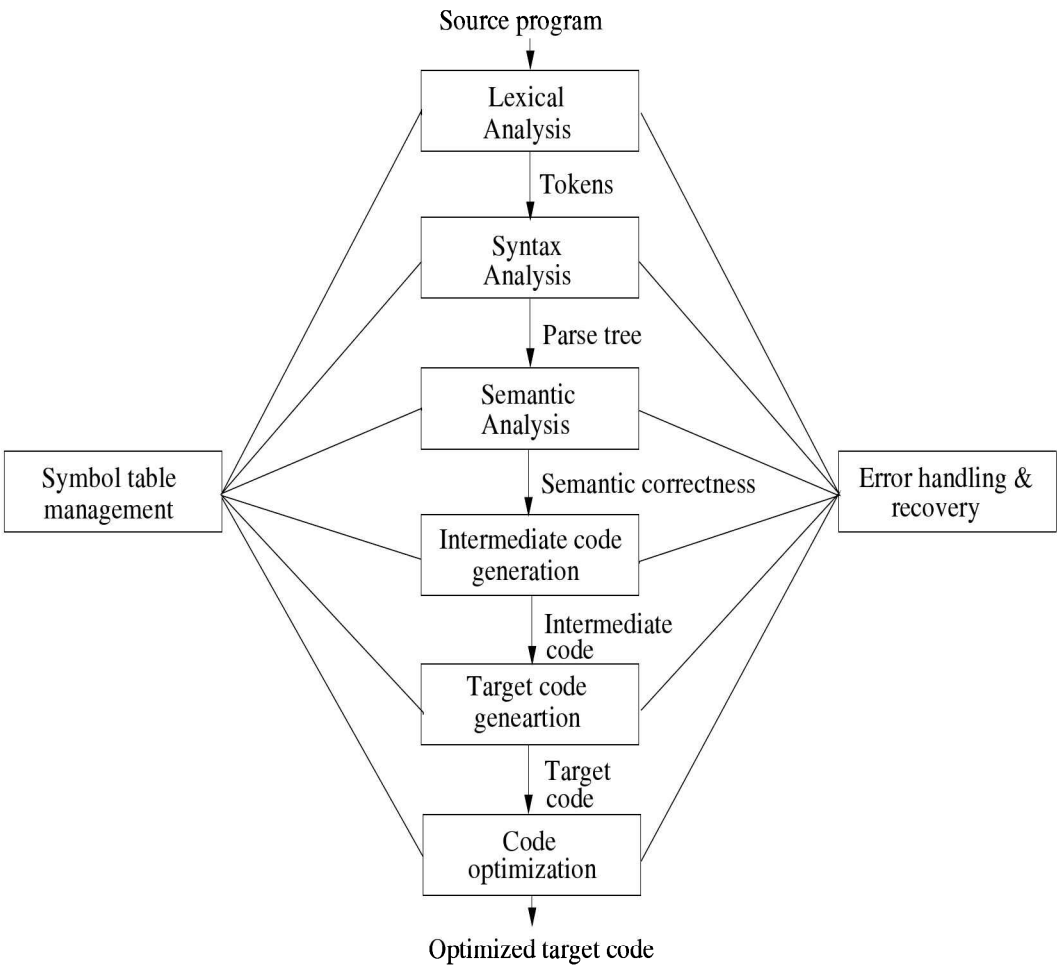

Figure 1.2: Phases of a compiler.

a case-conversion for example, from lower to upper case. The lexical analyzer generally passes a sequence of integers, known as *tokens*, to the syntax analysis phase. Thus, each program keywords, variables, numbers and symbols are converted into integers, so that the later phases of the compiler need not worry about the program text. The lexical analysis phase is generally implemented as a *finite automata.*

2. ***Syntax analysis.*** The syntax analysis phase takes words/tokens from the lexical analyzer and checks the syntactic correctness of the input program. The syntax analysis phase is successful if it can identify a sequence of grammar rules that can be used to derive the input program from a special start symbol of the grammar. In such case, it generates a tree representation, known as *parse tree*, to depict how the sequence of words in program can be combined to establish the program to be grammatically correct. If it is not possible to construct such a parse tree, then the input program is syntactically incorrect, and appropriate error message is flashed to the user. It may be noted that the

lexical and syntax analysis phases work hand-in-hand. Whenever the syntax analyzer needs further tokens, it requests the lexical analyzer which, in turn, scans the program from the current position to identify the immediate next token and returns it to the syntax analyzer. This phase is also known as *parsing*.

3. ***Semantic analysis.*** Once a program is found to be syntactically correct, that is, grammatically okay, the next task is to check for semantic correctness. Since the semantics of a program are dependent on the language, it is not possible to enumerate the types of semantic checks that should be included in a compiler. However, one of the very common check is for the types of variables and expressions occurring in the program. Strongly typed languages require the type of all variables to be declared before using them. On the other hand, loosely typed or untyped languages do not put such restrictions. While deriving the types of expressions, normally two checks are needed. The first one is about the applicability of operators to the operands. For example, integer division is allowed only if the operands are integers, the string concatenation operator can be applied only on string operands and so on. The second check requires determining the definition of the variables to be used. Many programming languages allow the same variable name to be used in more than one places in the program. In such cases, the definition corresponding to the occurrence of a variable name is determined by the *scope rules*. The check is easier to implement for statically scoped languages as compared to dynamically scoped languages, where the correct definition depends on the execution sequence of the program.
4. ***Intermediate code generation.*** This phase is an optional one towards the target code generation. In this phase, codes corresponding to the input source language program are generated in terms of some hypothetical machine instructions. This process helps to retarget the compiler generating code for one processor to another one. The language used for intermediate code generation is simple enough to be supported by most of the contemporary processors, and at the same time is powerful enough to express programming language constructs in terms of statements of this language. Figure 1.3 represents the idea behind using intermediate code to design compilers for different machines. It may again be noted that the lexical analyzer syntax analyzer, semantic analyzer, and intermediate code generator all work in co-operation to generate intermediate language code corresponding to a source program.
5. ***Target code generation.*** The task of target code generation from intermediate code is one of template substitution. For every statement of the intermediate language, a predefined target language template is used to generate the final code. The availability of machine instructions, addressing modes and a number of registers both general and special purpose play vital roles in designing such templates. Efforts are made to pack the temporary variables of the program, some user defined and some compiler generated into the CPU registers. This expedites the program execution.
6. ***Code optimization.*** This is the most vital step in target code generation to make the code efficient. Since the stages of a compiler are often automated, it generates a lot of redundant codes that can possibly be eliminated. For the purpose of optimization, the code is divided into *basic blocks*. A basic block is a sequence of statements with a single entry and single exit. While *local optimizations* restrict their operation within a single basic block, more rigorous *global optimization* that spans across the boundaries of

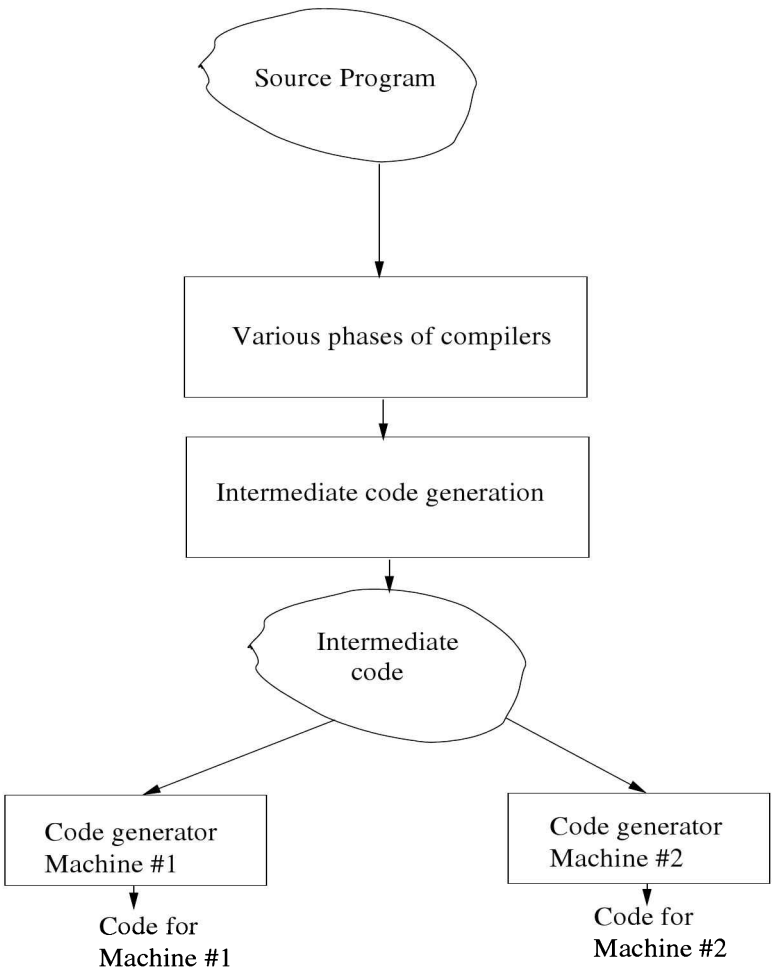

Figure 1.3: Targeting different machines.

basic blocks are needed to generate highly optimized codes. The most important source of optimization are the *loops*. This is because loops are executed repeatedly. A saving of even a single statement in a deeply nested loop will reduce the execution time for the program significantly. Other optimization methods involve *algebraic simplifications, elimination of store-and-load* cases, in which a variable computed in the previous statement is stored in memory, while the subsequent instruction loads it again from memory. Preservence of a copy of the variable in a register removes the reloading requirement. A plenty of other code optimization strategies have been presented in later chapters.

7. ***Symbol table management.*** Symbol table is a data structure holding information about all the symbols defined in the source program. Though it does not form a part of the final code generated, it is used as a reference table by all the phases of a compiler. The typical information stored in the symbol table includes the name of the variables, their types, sizes, relative offset within the program and so on. The generation of this

table is normally carried out by the lexical analyzer and syntax analyzer phases. Since during the compilation process, it is often required to look up this table for definition of variables, good data structures should be used to minimize searching time. Based on the scope rules of the language, it may be necessary to design the symbol table as a set of hierarchical ones rather than a flat table. If the language requires the definition of a variable to precede its use, the information stored in the symbol table can be used to check whether a variable is undefined or not. As and when a new variable definition comes, it is put into the symbol table. The symbol table is searched when a use of the variable is found.

8. ***Error handling and recovery.*** Though not related to the quality of code generated, error handling is one of the criteria of judging quality of a compiler. In case of a semantic error, the parser can still progress. However, in case of a syntax error, the parser reaches an erroneous state. The transition leading to the erroneous state gives an indication regarding the type of error. However, recovery is not an easy job. It needs to undo some processing already carried out by the parser, and perhaps, discarding a few more tokens to reach a descent state from where it can proceed again. Without this, the parser will come out at the first error itself, and the user would need to correct. However, since a number of such errors may be there, lot of user's time is wasted in correcting a single error at a time. Thus, error recovery plays a vital role in the overall operation of the compiler.

1.4 CHALLENGES IN COMPILER DESIGN

Challenges faced by a compiler designer are manifold. It ranges from the language details to the features of underlying hardware to the user-friendliness of the compiler. The following are some of the issues that affect the compiler design process.

1. ***Language semantics.*** The semantics of the source language constructs pose a challenge to the compiler writer because of the presence of constructs with complex semantics. Typical examples of such constructs are *case, loop, break, next*, etc. The semantics of the *case* statement vary in different languages. Some languages support fall-through semantics, whereas the others do not. *Loops* are difficult, because in some languages the value of the index variable after loop-body depends on the execution sequence. For normal exit from the loop, the variable may contain the last assigned value or the value may be undefined depending on the language under consideration. For abnormal exit from the loop, the index value is normally preserved. The statements like *break* and *next* modify the execution sequence of the program, which may depend upon the actual execution. Compiler designer must keep these into consideration, while generating code.

2. ***Hardware platform.*** The underlying machine architecture also affects the compilation process. For example, the code generation strategy for an *accumulator based* machine cannot be similar to a *stack based* machine. This is because, for an accumulator based architecture, for all operations, one of the operands and the destination are the accumulator, while for a stack machine, the operands are popped out of the stack, operation performed on those, and the result is pushed back onto the stack again. The instruction set of the processor also influences the code generation and optimization process. While

for a CISC (Complex Instruction Set Computer) architecture, the machine instructions are quite complex and an individual instruction can do a lot of work, for a RISC (Reduced Instruction Set Computer) machine, the individual instructions are very simple and regular in terms of size and execution time. For a CISC machine, the instructions are of various sizes requiring variable execution times. Thus, for a compiler writer, the RISC machines are preferred because of the regular smaller sized instructions that act as basic building blocks to constitute the target code. Presence of a large number of registers in a RISC CPU also helps in allocating most of the temporary variables to the registers. This often creates better code for a RISC machine than in a CISC machine. The compiler writer thus needs to look into the processor features carefully to fix up the code generation strategy.

3. ***Operating system and system softwares.*** The target code generated by a compiler finally goes to be executed in an environment created by an operating system (OS). The format of the file to be executed is, thus, depicted by the OS or more specifically, by the *loader*. Thus, the compiler's output should be compatible with it. To complete a source language program, it is also necessary to couple a number of library routines with it. The process is known as *linking*. There may be a number of compilers in a system, each for a different language. However, there is only a single linking program. Thus, the output of all these compilers should be compatible with the linker, so that different modules of a program may be written in different languages, converted into object files using respective compilers, and finally, linked by the linker to create the executable version.

4. ***Error handling.*** It has already been pointed out that error handling is one of the most important features of a compiler. A real challenge here for a compiler designer is to show the appropriate error messages to the user—detailed enough to pinpoint the error and at the same time, not too verbose to confuse the user. A typical example may be in the case of a missing semicolon at the end of a statement generating the error message *'<line no.> ; expected'* rather than simple *'syntax error'*. If a variable x is reported to be undefined at some position, it is not necessary to report the same error again and again at subsequent lines having a reference to x, unless and until, the scope of definition of x changes. Several such examples can be crafted for a particular language. The real challenge of a compiler designer is to imagine the probable types of mistakes that the users are expected to make and for each case, to design a suitable detection and recovery mechanism. Some compilers even go to the extent of modifying the source program partially in order to correct it.

5. ***Aid in debugging.*** Debugging is one of the most important techniques to detect logical mistakes in a program. This involves step-by-step executing the program, testing and setting the values of program variables, setting breakpoints within a program and so on. The most important issue, here, is that though the program being executed by the processor is in machine language, the user needs the control at source language level. This requires the compiler to generate extra information regarding the correspondence between source language program statements and the corresponding machine instructions. It also requires the symbol table information to be available to the debugger. Some additional modification to the code generation process may be required to embed extra debugging instructions within the object code. Thus, in order to make the code generated

by a compiler suitable for debugging, a compiler designer must take appropriate steps.

6. ***Optimization.*** Optimization poses a real challenge, because the compiler designer needs to identify the set of transformations that will be beneficial for most of the programs possibly written in that language. This is a nontrivial job, since it depends upon the language constructs, the target machine architecture and even on the type of applications developed in that language to be compiled by the designated compiler. A very important issue, here, is that the transformations should be *safe*, that is, an optimizing transformation should never change the computation done by the unoptimized version of the program. There is often a trade off between the amount of effort spent to optimize a program, vis-a-vis the improvement in execution time. Most of the compilers are designed to have several levels of optimizations that may be tried out by the user. Selecting the debugging option may disable any optimization that changes the order of instructions in the target file, as this will disturb the correspondence between the source program and the object code.

7. ***Runtime environment.*** Runtime environment refers to the handling of subprograms. It necessarily deals with creation of space for parameters passed and the local variables within the subprogram. For earlier languages like FORTRAN, the environment is static. This means that for each of the local variables and the parameters passed, there are fixed memory locations created by the compiler. This is sufficient since the languages did not support recursion. At any point of time, there exists atmost only one activation of a subprogram alive. For languages supporting recursion, the runtime environment management is much more critical. It is normally implemented by pushing stack frames to hold the parameters passed and the local variables. For each call to a subprogram, such a frame is pushed onto the stack and popped out on return. Thus, looking into the language features regarding subprograms, the compiler designer needs to design schemes for efficiently handling the runtime environment.

8. ***Speed of compilation.*** This is one of the most important criteria to judge the acceptability of a compiler to the user community. During the development cycle of a program, initially, there are lot of grammatical and logical bugs. Hence, the logic of the program undergoes frequent changes. The users at this stage are more interested to determine the correctness of their codes, rather than the execution efficiency. The compiler should, thus, be able to compile fast, though, the code produced may be inefficient. Near the final phases of the program development, users pay more concentration to get the highest possible efficiency for their program. Accordingly, the compiler should be able to do a good optimization work, may be at the cost of higher compilation time. Thus, the compiler developed should be able to produce unoptimized code very fast, while the optimized code generation may take some extra time to compile.

1.5 COMPILATION PROCESS—AN EXAMPLE

In this section, we will look into the entire compilation process for a very simple program fragment written in a Pascal-like language. Consider the following program:

```
program
    var X1, X2:  integer;  { integer variable declaration }
    var XR1:  real;        { real variable declaration }
begin
    XR1 := X1 + X2 * 10;   {assignment statement }
end.
```

Lexical analysis. The lexical analyzer scans the program looking for major lexical elements, called *tokens*. For the above example, the lexical analysis produces the following sequence of tokens:
`program, var, X1, ',', X2, ':', integer, ';', var, XR2, ':', real, ';', begin, XR1, ':=', X1, '+', X2, '*', 10, ';', end, '.'`
The whitespaces (blanks, tabs, newlines) and the comments are discarded and removed by the lexical analyzer.

Syntax analysis. Syntax analysis phase analyzes the program for grammatical correctness. For grammatically correct programs, it produces a parse tree. Assuming that the given program is grammatically correct, determined by analyzing the grammar rules and the program, the parse tree produced has been shown in Fig. 1.4.

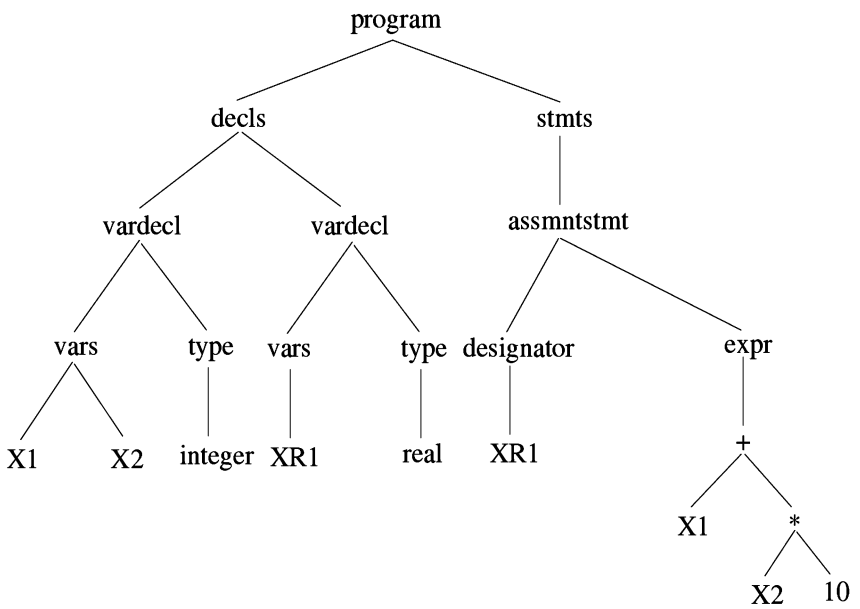

Figure 1.4: An example parse tree.

Code generation. The code generation process traverses the parse tree and generates code for the input program. Assuming that we have a stack-oriented machine with instructions like PUSH, ADD, MULT, and STORE, the generated code will look like the following:

```
PUSH X2
PUSH 10
```

```
MULT
PUSH X1
ADD
PUSH @XR1
STORE
```

The '@' symbol returns the address of the variable following it.

Symbol table. Symbol table created by the lexer and parser is used later by the code generator. It holds information such as type of identifier, etc. For the example program, the symbol table is shown in Table 1.1.

Table 1.1: Symbol table

name	class	type
X1	variable	integer
X2	variable	integer
XR1	variable	real

1.6 CONCLUSION

In this chapter, we have seen an overview of the compiler design process. A compiler is an utility to convert a source language program to a target language. There are several applications of compilers spread over fields from code generation to hardware generation (silicon compilation). We have seen a broad overview of the phases of a compiler. The challenges faced by a compiler designer are many as depicted in the chapter. An example of the compilation process illustrates the ideas. The later chapters present details about the individual phases of the compiler.

EXERCISES

1.1 What is a compiler?

1.2 Distinguish between compiler and interpreter bringing out clearly the situations in which each of them is more suitable than the other.

1.3 Diagram the likely components of a compiler and describe their roles.

1.4 A RISC processor has smaller number of instructions and more number of registers than a CISC machine. Describe the pros and cons of compiler design targeted to a RISC machine as compared to a CISC machine.

1.5 A *cross-compiler* is one that runs on a machine to generate target code for another machine. Identify a few cases where such a cross-compiler will be useful.

1.6 Explain why a system may have several compilers but normally has a single linker.

1.7 Discuss the challenges in compiler design.

1.8 Study the features of any standard compiler available in your computer system.

1.9 Which type of compiler is better—one running slowly but producing optimized code and another one running very fast but producing unoptimized code?

1.10 Can human being optimize a program better than an automated compiler? Justify your answer.

Chapter 2

Lexical Analysis

Lexical Analysis is the first phase of a compiler. The main task accomplished by this phase is to identify the set of valid words of a language that occurs in an input stream. This chapter elaborates the concepts and tools involved in the process of lexical analysis. However, before designing any tool to identify the words of a language, we need to have a concrete notation to express the valid set of words that are possible in the language. Since we are mostly concerned with the programming languages, we can observe that the possible set of words in such languages is finite—they represent either the keywords of the language, a symbol like name of a variable, label, subprogram, etc., numbers—integers, real, etc., some arithmetic and logical operators and so on. To design a recognizer for these words, the set of possible words is represented by a notation known as *Regular Expression.* Once the set of regular expressions for a language is ready, we can construct a *Finite Automata* capable of determining whether or not a given word belongs to the language. This acceptor when accompanied by some attributes related to the word being detected serves the purpose of lexical analysis. There exist automated tools to construct a lexical analyzer from the specification of regular expressions.

2.1 ROLE OF LEXICAL ANALYZER

Lexical analyzer acts as an interface between the input source program to be compiled and the later stages of the compiler. The input program can be considered to be a sequence of characters. Lexical analyzer converts this character stream into a stream of valid words of the language, better known as *tokens.* The parser looks into the sequence of tokens and identify the language constructs occurring in the input program. However, the parser and the lexical analyzer work hand-in-hand, in the sense that whenever the parser needs further tokens to proceed, it requests the lexical analyzer. The lexical analyzer, in turn, scans the remaining input stream and returns the next token occurring there. The idea is explained in Fig. 2.1. Apart from that, the lexical analyzer also participates in the creation and maintenance of *Symbol Table.* This is because, lexical analyzer is the first module to identify the occurrence of a symbol. If the symbol is getting defined for the first time, it needs to be installed into the symbol table. Lexical analyzer is mostly used for doing the same. Some of the other fields of the symbol table like, *type, size,* etc. are generally filled up by the parser. For later

occurrences of the symbol, the symbol table is searched to extract the relevant information about it. The lexical analyzer is also responsible for removing the cosmetics from the program like extra white spaces, comments, etc.

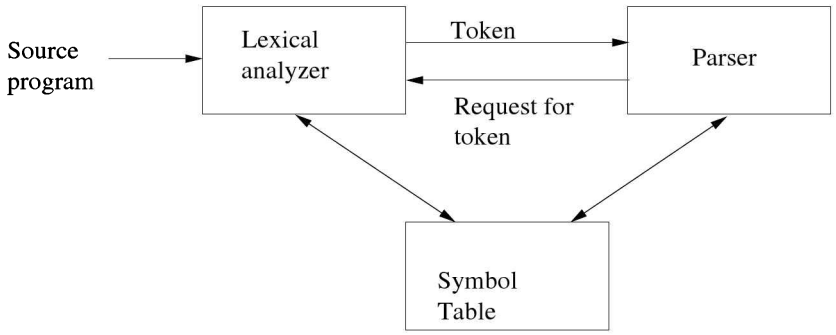

Figure 2.1: Role of lexical analyzer.

Definition 2.1 We will, here, introduce a few definitions to make our further discussion clear. To start with, let us think of the definition of possible identifiers that can occur in some programming language. Suppose, in the language, identifiers are defined as a sequence of characters consisting of letters 'a' to 'z',digits '0' to '9' and the underscore character '_'. Then, a few possible identifiers are "abc", "a_b1", etc. Each of these identifiers occurring in a program is called a **lexeme**. The definition of the identifier is called a **pattern**. For each of these lexemes, the lexical analyzer returns the token "identifier". The tokens are normally represented as integers, so that they can be handled efficiently by the later stages of the compiler.

Attributes of tokens: In many cases, it is not sufficient just to return only the token to the parser. For example, with the token "identifier", we also need information regarding if it is already defined, and if so, the corresponding location in the symbol table. For a constant, it is also necessary for the parser to know the type of the constant, and it's value—apart from returning the token "constant". This extra information attached to the tokens are called *attributes*. The exact set of attributes needed depends upon the language, the tokens and the compilation strategy.

2.2 SPECIFICATION OF TOKENS

The patterns corresponding to a token are generally specified using a compact notation known as *regular expression*. Regular expressions of a language are created by combining its alphabets. For a language L with alphabet set Σ, the following rules define the regular expressions:

1. ϵ is a regular expression denoting the language $\{\epsilon\}$, that is, the set containing only the empty string.
2. If a be a symbol in Σ, then a also denotes a regular expression corresponding to the language $\{a\}$, that is, the set containing only the string a.
3. If r_1 and r_2 be two regular expressions corresponding to the languages L_1 and L_2 respectively, then

(a) $r_1|r_2$ is a regular expression corresponding to the language $L_1 \cup L_2$, i.e., the set containing all the strings of L_1 and L_2.

(b) $r_1 r_2$ is a regular expression corresponding to the language created by concatenating strings of L_2 to the strings of L_1.

(c) r_1^* is a regular expression corresponding to the language L_1^*, that is, the set containing zero or more occurrences of the strings belonging to L_1. Note that, L_1^* also contains the empty string, even if L_1 may not contain it.

(d) (r_1) is a regular expression corresponding to the language L_1 itself.

Precedence and associativity of the operators '|', concatenation and '$*$' may be used to simplify the expression. Unary operator $*$ is of highest precedence and is left associative. Concatenation has the next highest and | has the lowest precedence. All the operators are left associative.

Example 2.1 For $\Sigma = \{0,1\}$, let us consider the following regular expressions:

1. (0|1)* denotes all the binary strings including the empty string.
2. (0|1)(0|1)* denotes all the nonempty binary strings.
3. 0(0|1)*0 denotes all the binary strings of length at least two, both starting and ending with 0s.
4. $((\epsilon|0)1^*)^*$ denotes all the binary strings.
5. (0|1)*0(0|1)(0|1) denotes all the binary strings with at least three characters, in which the third-last character is always zero.
6. 0*10*10*10* denotes all the binary strings possessing exactly three 1s.

Most of the programming languages define variables as *letter (letter | digit | '_')**, that is, any variable name is a sequence of *letters*, *digits* and the underscore character ('_') with the restriction that it must start with a letter. The set of signed integers may be specified as

$$(+|-|\epsilon)\ \textit{digit (digit)}^*$$

Here, the portion '$+|-|\epsilon$' represents the fact that an integer may have an explicit sign (+ or −) or the sign may be absent (ϵ), in which case, it is assumed to be a positive number. The later part *digit (digit)** corresponds to the one or more digits that the number contains. The set of floating point numbers, similarly, may be specified as

$$(+|-|\epsilon)\ \textit{digit (digit)}^*\ (\cdot\ \textit{digit (digit)}^*|\epsilon)((\text{E}\ (+|-|\epsilon)\ \textit{digit (digit)}^*)\ |\epsilon)$$

2.3 TOKEN RECOGNITION

In the previous section, we have learnt to specify tokens of a language using the compact notation called *regular expression*. In this section, we will elaborate how to construct recognizers that can identify the tokens occurring in an input stream. These recognizers are most widely known as *Finite Automata*. As the name suggests, it is a machine with a finite number of states in which the machine can perform. There is a distinctive *start state*, in which the machine

starts. Between the states, there are transitions labeled by alphabets of the language. If the machine is at state s_i, and there is a transition labeled $a_k \in \Sigma$ (the alphabet set) to the state s_j, then the machine can perform that transition, provided the current input symbol is a_k. Starting from the *start state*, depending upon the input stream, the machine, finally, reaches some state after exhausting all input symbols. There are two types of such final states. The set of final states reachable for valid input streams (tokens) is called *acceptor state* or simply *final state*. On the other hand, states reached by input streams not corresponding to any of the tokens are called *failure states* or *reject states*. For example, consider the finite automata shown in Fig. 2.2.

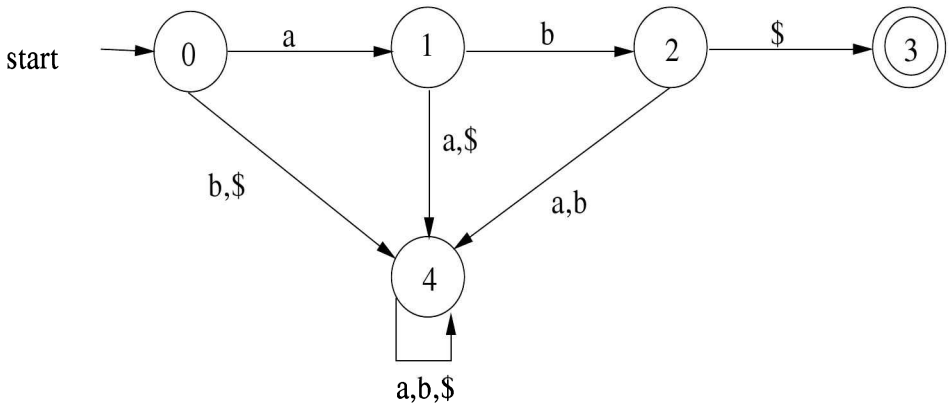

Figure 2.2: Finite automata for *ab*.

It corresponds to the regular expression *ab*. Here, state-0 is the start state, 3 is a final state, and 4 is a reject state. *Start* state is identified by an incoming edge without any predecessor. *Final* states are double-circled. A *reject* state contains a self-loop labeled by all possible input symbols. Input streams are assumed to be terminated by \$. It is easy to see that the machine reaches the acceptor state 3 only for the input *ab*\$. For any other input, it reaches the reject state 4. For the sake of brevity, the *reject* states are, normally, not shown. In that case, if the machine is in state s_i and there does not exist any possible transition from s_i to the next input symbol, then the input is rejected. Transitions on \$ are also not shown. Thus, the simplified machine is as shown in Fig. 2.3.

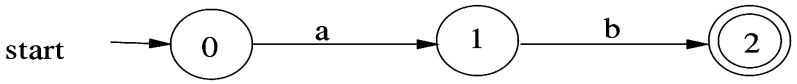

Figure 2.3: Simplified finite automata for *ab*.

There are two types of finite automata—*Non-deterministic finite automata (NFA)* and *Deterministic finite automata (DFA)*. The basic difference between the two lies in the presence of non-determinism in the transitions of NFA, which is made deterministic in DFA. We will now deal with the details of NFA and DFA constructions.

2.3.1 Nondeterministic Finite Automata (NFA)

The basic feature of a nondeterministic finite automata is its ability to have more than one possible transitions from a state on the same input symbol. For example, from state s_i, there may be transitions to two distinct states s_j and s_k labelled by the same input symbol a_m. The actual transition occurring is to be chosen nondeterministically by the machine. Another nondeterminism lies with the transitions labelled by ϵ, known as ϵ-transitions. If there is an ϵ-transition from state s_i to state s_j, then the machine can do such a transition without consuming any further inputs. An input stream is said to be accepted, that is recognized, by the automata, if there exists at least one path (deterministic/nondeterministic) from the start state of the machine to one of its final states whose transitions are governed by the input stream.

Example 2.2 Following are a few examples of NFA.

1. For the regular expression 0(0|1)*0, the NFA is shown in Fig. 2.4. Here, from state 1, there are two transitions labelled 0 – one to the state 1 itself and the other to state 2.

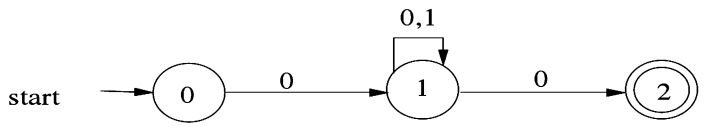

Figure 2.4: NFA for 0(0|1)*0.

2. For the regular expression 01|10, the NFA is shown in Fig. 2.5. Here, from state 0, there are two ϵ-transitions to the states 1 and 4 respectively.

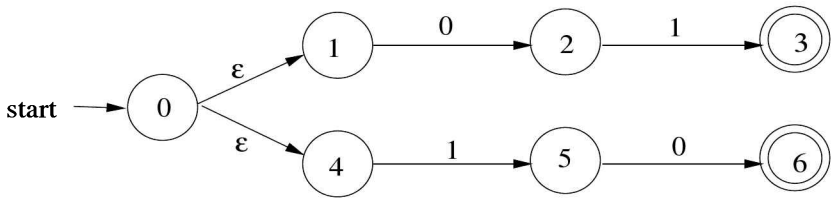

Figure 2.5: NFA for 01|10.

2.3.2 Deterministic Finite Automata (DFA)

As compared to an NFA, the transitions are deterministic in a DFA. Thus, there cannot be more than one transition emanating from the same state labelled by the same input symbol. Secondly, there cannot be any ϵ-transitions.

Example 2.3 The following are a few examples of DFA.

1. For the regular expression 0(0|1)*0, the DFA is shown in Fig. 2.6. As it can be seen, for any state, no two transitions have been labeled the same.
2. For the regular expression 01|10, the DFA is shown in Fig. 2.7.

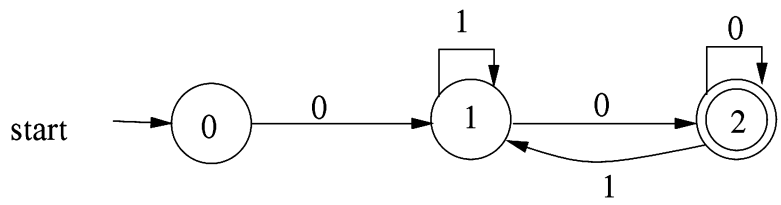

Figure 2.6: DFA for 0(0|1)*0.

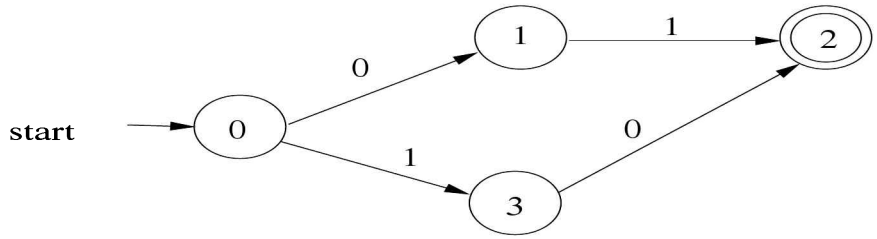

Figure 2.7: DFA for 01|10.

We will, now, see that for every NFA, we can construct an equivalent DFA. Thus, the availability of nondeterminism does not make NFA more powerful than a DFA. However, what matters are the sizes (number of states) of these machines. As it should be understood, the absence of nondeterminism may require a good penalty to be paid by DFA in terms of number of states. Each and every nondeterministic alternative of an NFA gets converted into an explicit path from start state to a final state in the corresponding DFA. This often results into an explosion in the number of states for a DFA. For example, consider the regular expression $(a|b)^*a(a|b)(a|b)\ldots(a|b)$, where there are $(n-1)$ such $(a|b)$ at the end. This represents strings over the alphabet set $\Sigma = \{a, b\}$, in which the n-th character from the end is always a. The NFA for this regular expression is shown in Fig. 2.8. It has $(n+1)$ states, with a nondeterminism in state 0, from which it can either make a transition to state 1 or remain in state 0 for the same input a.

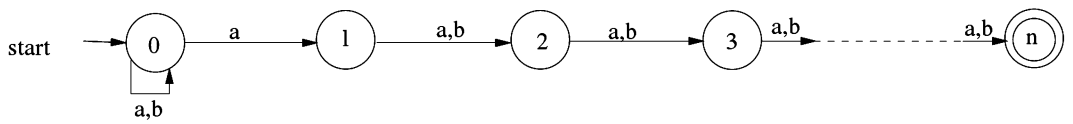

Figure 2.8: NFA for $(a|b)^*a(a|b)(a|b)\ldots(a|b)$.

On the other hand, if we try to construct a DFA for this, then we have to consider all possible strings over $\{a, b\}$ in which the n-th character from the end is a, and the remaining $(n-1)$ characters after it are a's or b's. Thus, for $n = 3$, we have to consider the cases $(a|b)^*a(aa|ab|ba|bb)$. A DFA for this is shown in Fig. 2.9. Here, we essentially construct a machine that keeps track of last three symbols seen. If these symbols are *aaa*, *aab*, *aba*, or *abb*, it declares *accept*, else *failure*. To remember three symbols, we need to have eight distinct states remembering them. Thus, for n such symbols, we will need at least 2^n states in the DFA.

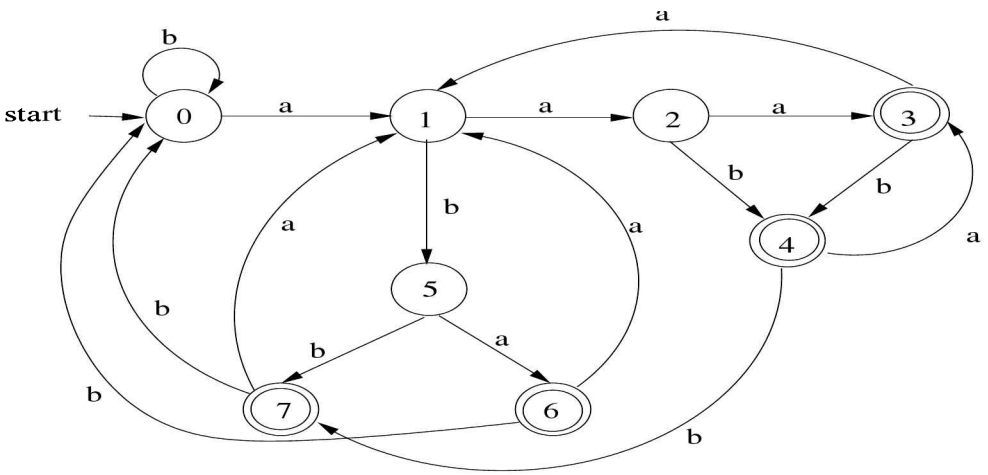

Figure 2.9: DFA for $(a|b)^*a(aa|ab|ba|bb)$.

However, in the ordinary programming languages, we do not encounter many such tokens for which the DFA has got exponential number of states as compared to the corresponding NFA. Moreover, DFA does not have any nondeterminism in its operation, which means that to test whether an input stream forms a valid token or not, we just have to follow the transitions that are possible from the start state, guided by the input symbols. The algorithm to test whether an input stream is acceptable by a DFA or not, is depicted next.

Algorithm DFA-test
Input: A DFA with start state S_0
An input stream
Output: "yes" if accepted, "no" otherwise
Begin
$S = S_0$
While not end-of-input **do**
Let c = next input symbol
If there is a transition on c from S to some state S_1 **then**
$S = S_1$
end while
If S is a *final state* and c= end-of-input **then**
return "yes"
else return "no"

Example 2.4 For the DFA shown in Fig. 2.6, given the input stream "0110", the Algorithm DFA-test makes transitions of states $0 \rightarrow 1 \rightarrow 1 \rightarrow 2$, and returns "yes". On the other hand, for the input stream "011", it traverses $0 \rightarrow 1 \rightarrow 1$. Since 1 is not a final state, the stream is rejected, and the algorithm returns "no".

The existence of nondeterminism makes the task of testing difficult whether an input stream is acceptable by an NFA. Because, in this case, we need to try out all possible alternatives that may occur in nondeterministic transitions of states. For example, if for an input symbol

a there are two transitions from state S to S_1 and S_2 respectively, then both the alternatives need to be explored before declaring the input as rejected. The presence of ϵ-transitions also creates problems, as we may have multiple choices of transitions from a state in that case. Hence, in order to have a lexical analyzer with the ability to detect the tokens fast, DFA is often used. However, in many cases, it is easier to design the NFA than the corresponding DFA. We need to have a tool that can convert an NFA into a DFA. Now, we will discuss a technique for this conversion.

2.3.3 NFA to DFA

To construct the DFA equivalent to a given NFA, we note that a set of states in an NFA corresponds to a state in the DFA. All these states of the NFA are reachable from at least another state of the same set using ϵ-transition only, that is, without consuming any further input. Also, from this set of states, based on some input symbol we can reach another set of states. In the DFA, we take these sets as unique states. To start with, we define two sets:

Definition 2.2 **ϵ-closure(s):** *ϵ-closure of a state s in an NFA is defined to be the set of states (including s) that are reachable from s using ϵ-transitions only.*

Definition 2.3 **ϵ-closure(S):** *ϵ-closure of a set of states S of an NFA is defined to be the set of states reachable from any state in S using ϵ-transitions only.*

The ϵ-closure of a set of states S can thus be computed by the following algorithm:

ϵ-closure(S) = S
Set all states of ϵ-closure(S) to be unmarked
While there are unmarked states in ϵ-closure(S) **do**
  Let s be such a state
  Mark s
  For all state s' having an ϵ-transition from s to it **do**
    If s' not in ϵ-closure(S) **then**
      ϵ-closure(S) = ϵ-closure(S) $\cup$ $\{s'\}$
      Set s' unmarked

Next, we look into the algorithm to convert an NFA to the corresponding DFA.

Algorithm NFA-to-DFA
Input: An NFA with set of states S, start state s_0, set of final states F
Output: Corresponding DFA with start state d_0, set of states S_D, set of final states F_D
Begin
  d_0 = ϵ-closure(s_0)
  $S_D = \{d_0\}$
  If d_0 contains a state from F **then** $F_D = \{d_0\}$ else $F_d = \phi$
  Set d_0 unmarked
  For each unmarked state d in S_D do
    For each input symbol a **do**
      Let T be the set of states in S (of NFA) having transitions on a from any state of the NFA corresponding to the DFA state d

$d' = \epsilon$-closure(T)
If d' is already present in S_D, **add** the transition $d \rightarrow d'$ labelled by a
else
$S_D = S_D \cup \{d'\}$
Add transition from d to d' labelled a
Set d' unmarked
If d' contains a state of F **then** $F_D = F_D \cup \{d'\}$

Example 2.5 Consider the NFA shown in Fig. 2.10 corresponding to the regular expression $a(a|b)^*ab$. We will construct the corresponding DFA using the algorithm *NFA-to-DFA*. To start with, the initial state of the DFA is constructed as,

$$d_0 = \epsilon\text{-closure}(\{1\}) = \{1\}$$

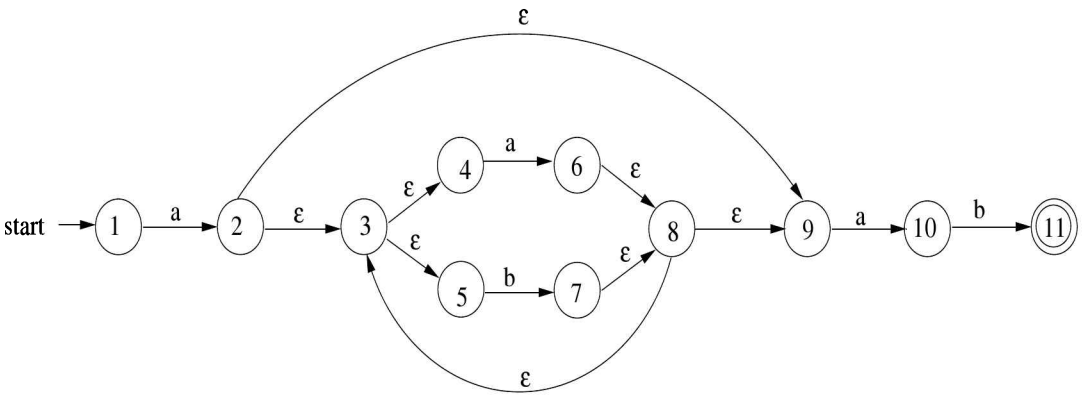

Figure 2.10: NFA for $a(a|b)^*ab$.

This is the only state in D now. From d_0, we consider the transitions on a. The new state constructed is,

$$d' = \epsilon\text{-closure}(\{2\}) = \{2, 3, 4, 5, 8, 9\}$$

In this way, we can construct the new states and transitions in the DFA as shown in Table 2.1. The DFA is shown in Fig. 2.11. Since state 5 of the DFA contains the final state of the NFA (state 11), it is the only *final* state of the DFA. It may be verified that the DFA accepts the same set of strings $a(a|b)^*ab$ as the original NFA.

2.4 REGULAR EXPRESSION TO NFA

In this section, we present the technique to construct an NFA that can be used as a recognizer for the tokens corresponding to a regular expression. The NFA thus obtained can then be converted into a DFA using the approach depicted in the previous section. The conversion of a regular expression to an NFA essentially follows breaking down the regular expression into simplest subexpressions, constructing the corresponding NFAs, and then combining these small NFAs guided by the operators of the regular expression. This construction is known as

Table 2.1: DFA state construction

DFA state	Set of NFA states	Transition on		Is *final*
		a	*b*	
1	{1}	2	-	no
2	{2,3,4,5,9}	3	4	no
3	{3,4,5,6,8,9,10}	3	5	no
4	{3,4,5,7,8,9}	3	4	no
5	{3,4,5,7,8,9,11}	3	4	yes

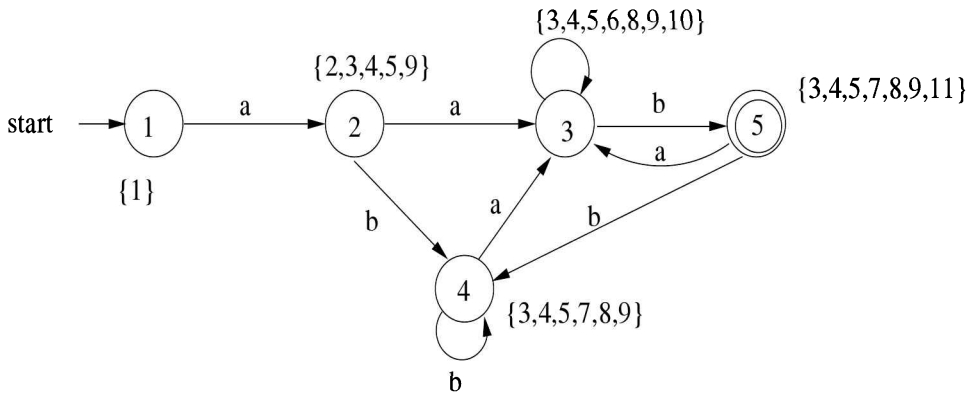

Figure 2.11: DFA for $a(a|b)^*ab$.

Thompson's construction. The set of rules for this construction are as follows. The rules are also shown in Fig. 2.12.

1. For ϵ, the NFA is as shown in Fig. 2.12(a) consisting of two states—a start state and a final state. The transition is labeled by ϵ.
2. For any alphabet symbol a in the alphabet set Σ, the NFA is constructed as in Fig. 2.12(b). It is similar to the ϵ-case, with the labeling done with a instead of ϵ.
3. For the regular expression $r_1|r_2$, if $N(r_1)$ be the NFA for r_1 and $N(r_2)$ be the NFA for r_2, then the NFA $N(r_1|r_2)$ is constructed as in Fig. 2.12(c). In this case, a new start state and final state are added. ϵ-transitions are introduced from the new start state to the start states of $N(r_1)$ and $N(r_2)$. Similarly, ϵ-transitions are added from the final state of $N(r_1)$ and $N(r_2)$ to the newly created final state. The final states of $N(r_1)$ and $N(r_2)$ cease to be final states any more.
4. For the regular expression r_1r_2, the NFA $N(r_1r_2)$ is constructed by merging the NFAs $N(r_1)$ and $N(r_2)$. The final state of $N(r_1)$ is merged with the start state of $N(r_2)$. It is shown in Fig. 2.12(d). The transitions emanating from the start state of $N(r_2)$ now originates at the final state of $N(r_1)$. The start state of $N(r_1)$ becomes the start state of the new NFA and the final state of $N(r_2)$ is the final state of new NFA. Final state of $N(r_1)$ ceases to be the final state any more.

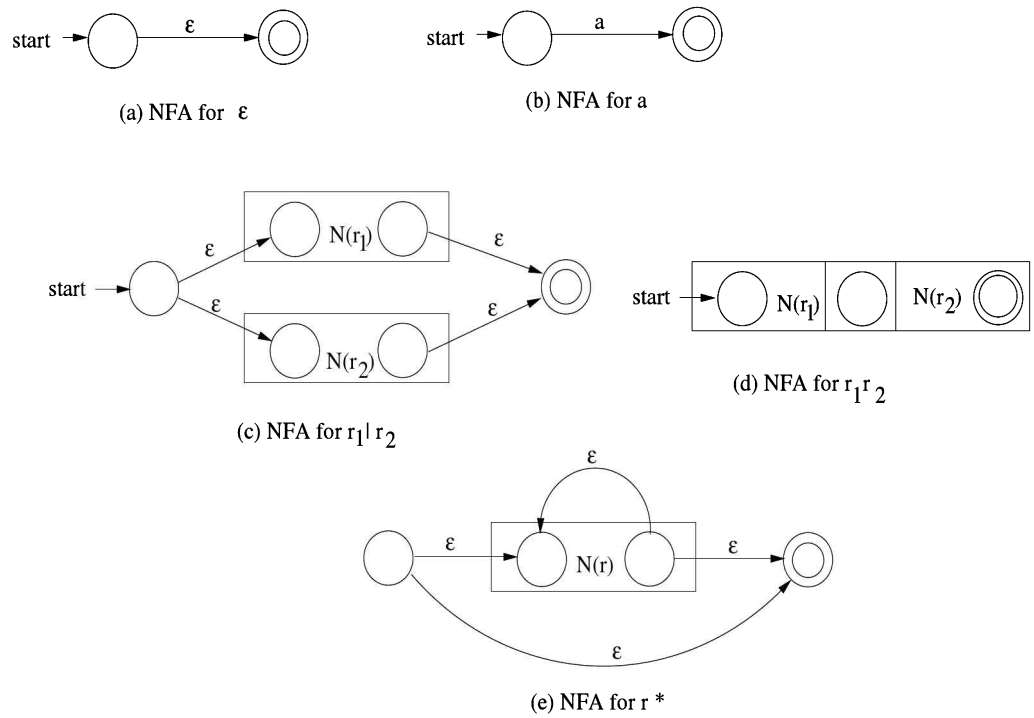

Figure 2.12: Regular expression to NFA construction.

5. For the regular expression r^*, NFA $N(r^*)$ is constructed from the NFA $N(r)$ of r as follows:
   First a new start state and a new final state are introduced. Then, ϵ-transitions are added from the new start state to the start state of $N(r)$, from final state of $N(r)$ to the new final state, from final state of $N(r)$ back to the start state of $N(r)$, and from the new start state to the new final state. Addition of ϵ-transition from new start state to new final state corresponds to the zero-occurrence of r, whereas, the ϵ-transition from the final to initial state of $N(r)$ corresponds to the repeated occurrence of r. The situation has been shown in Fig. 2.12(e).
6. If $N(r)$ be the NFA for a regular expression r, it is also the NFA for the parenthesized expression (r).

Example 2.6 Let us consider the construction of the NFA for the regular expression $a(a|b)^*ab$. Here, the subexpressions are:

1. a
2. b
3. $a|b$
4. $(a|b)^*$
5. $a(a|b)^*$

6. ab
7. $a(a|b)^*ab$

The corresponding NFAs are shown in Fig 2.13. First, the basic NFAs are constructed which are then combined using the rules discussed earlier.

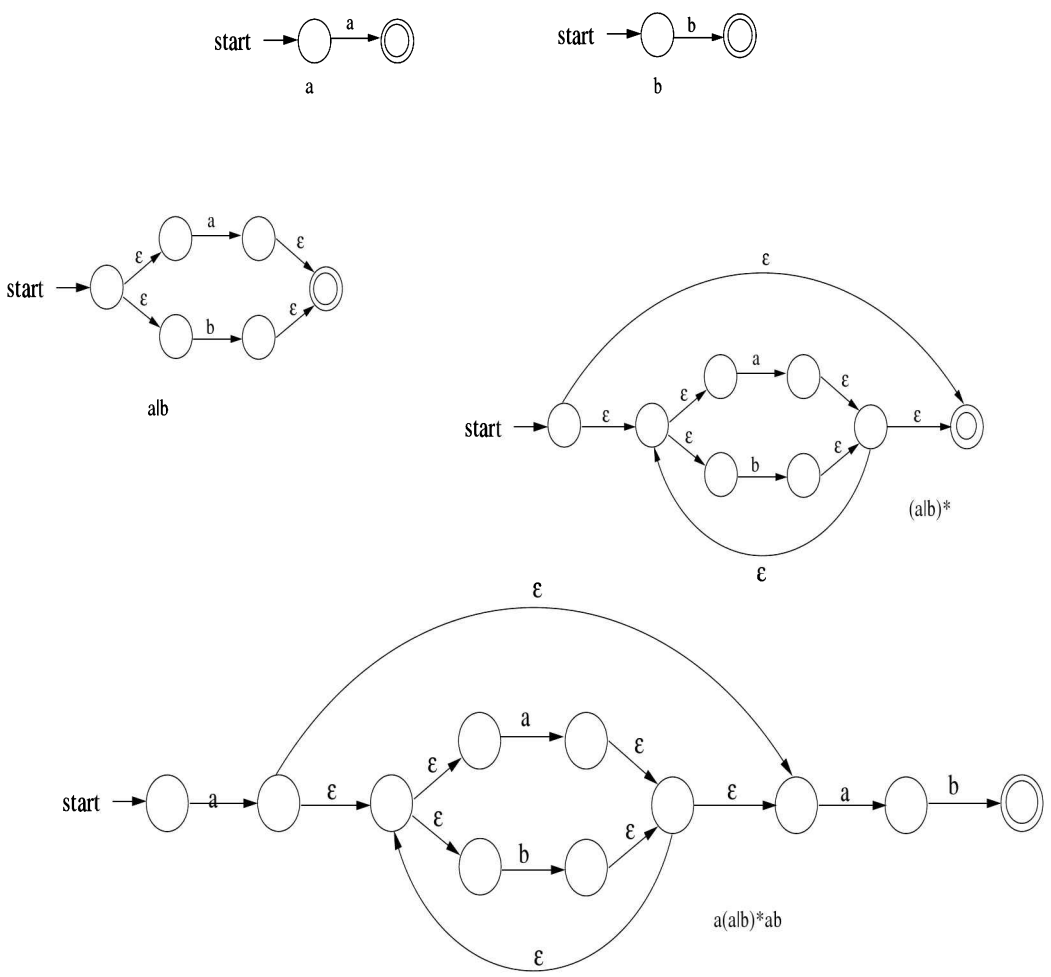

Figure 2.13: NFA construction for a regular expression.

2.5 LEXICAL ANALYSIS TOOL–LEX

In this section, we will present an overview of a tool for automatically constructing a lexical analyzer for a language. The tool is called *Lex compiler* or simply *Lex*. It comes as an integrated utility of the UNIX operating system. It has been ported to other systems also. The tool accepts a specification of the tokens in terms of extended regular expressions along with

supporting routines. It produces a C-program (*lex.yy.c*) as an output, that can be compiled to obtain an executable version of the lexical analyzer. When used in conjunction with a parser, the parsing routine can call the function *yylex* generated automatically. However, Lex can also be used independently for a number of text processing applications that we shall see shortly. The overall strategy for constructing the lexical analyzer is shown in Fig 2.14. The

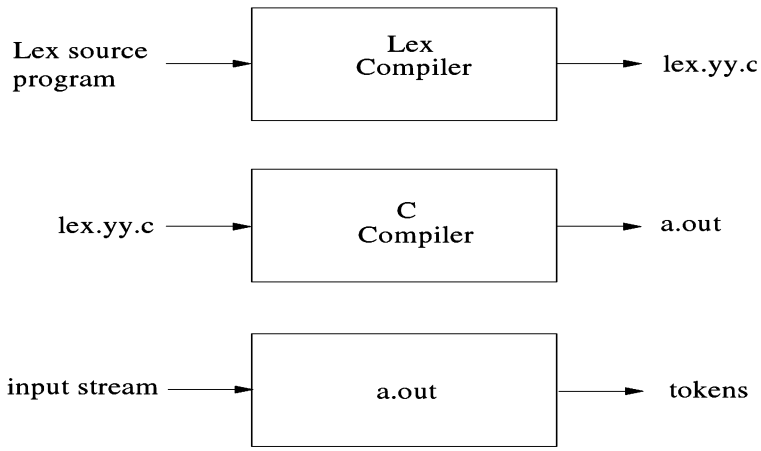

Figure 2.14: Constructing lexical analyzer.

lexical specification file, e.g. *lex.l*, is passed through *lex* to produce the C program file *lex.yy.c*, which is then compiled using a C compiler to produce the lexical analyzer (*a.out*). This lexical analyzer can now be fed with an input stream to determine the tokens in it.

2.5.1 Lex specification

Any Lex program consists of three parts:

declarations
%%
rules
%%
auxiliary routines

The *declarations* section includes variable declarations, constant declarations, and regular definitions. The regular definitions can be used as parts in the *rules* section. The *rules* section contains the specification of tokens and the associated *action*. This section is of the form:

$$\begin{array}{ll} r_1 & \{action_1\} \\ r_2 & \{action_2\} \\ & \vdots \\ r_k & \{action_k\} \end{array}$$

Here, each r_i is a regular expression (extended with the additional flexibilities provided by Lex). Each *action* specifies a set of statements to be executed whenever rule r_i matches the

current input sequence. The *auxiliary routines* are the functions that may be used to write the *action* parts.

When coupled with a parser, as and when the parser needs further tokens, it calls the lexical analyzer. The rules are then checked by the lexical analyzer as per their order of occurrence in the *rules* section. For the rule with maximum matching, it executes the corresponding *action* part. The *action* part is typically coded to fill up data structures and token attributes that may be used by the parser along with the token to analyze the input stream.

Regular expressions in Lex are composed of metacharacters. These metacharacters are shown in Table 2.2. Pattern matching examples using these metacharacters are shown in Table 2.3. Within a character class, normal operators lose their meaning. Two operators allowed in a character class are hyphen (-) and circumflex (^). The hyphen represents a range of characters when used between two characters. The circumflex, when used as the first character, negates the expression. If two patterns match the same string, the longest match wins. In case both the matches are of same length, the first pattern listed is used.

Table 2.2: Lex pattern matching primitives

Metacharacter	Matches
.	any character except newline
\n	newline
*	zero or more copies of the preceding expression
+	one pr more copies of the preceding expression
?	zero or one copy of the preceding expression
^	beginning of line
$	end of line
$a|b$	a or b
$(ab)+$	one or more copies of ab (grouping)
"$a+b$"	literal "$a+b$"
[ ]	character class

The most simple Lex program is the following single-line one:

```
%%
```

Input is copied to output, one character at a time. The first %% is always required, as there must be a rules section. However, if we do not specify any rules, the default action is to match everything and copy it to output. Defaults for input and output are *stdin* (standard input) and *stdout* (standard output) respectively. Here, is the same example with defaults explicitly coded:

```
%%
    /* match everything except newline */
.       ECHO;
    /* match newline */
\n      ECHO;
```

Table 2.3: Pattern matching examples

Expression	Matches
abc	abc
abc*	ab abc abcc abccc ...
abc+	abc abcc abccc ...
a(bc)+	abc abcbc abcbcbc ...
a(bc)?	a abc
[abc]	one of a, b, c
[a-z]	any letter a-z
[a\-z]	one of a, -, z
[-az]	one of -, a, z
[A-Za-z0-9]+	one or more alphanumeric character
[\t\n]+	whitespace
[^ ab]	anything except a b
[a^ b]	one of a, ^, b
[a|b]	one of a, |, b
a|b	one of a, b

```
%%
int yywrap( void ) {
    return 1;
}
int main( void ) {
    yylex();
    return 0;
}
```

Two patterns have been specified in the rules section. Each pattern must begin from column one. This is followed by whitespace (space, tab or newline) and an optional action associated with the pattern. The action may be a single C statement, or multiple C statements enclosed in braces. Anything not started in column one is copied verbatim to the generated C file. We may take advantage of this behaviour to specify comments in our Lex file. In the example mentioned above, there are two patterns, "." and "\n", with an ECHO action associated with each pattern. Several macros and variables are predefined by Lex. ECHO is a macro that writes code matched by the pattern. This is the default action for any unmatched strings. Typically, ECHO is defined as:

```
#define ECHO fwrite(yytext, yyleng, 1, yyout)
```

Variable yytext is a pointer to the matched string (NULL-terminated), and yyleng is the length of the matched string. Variable yyout is the output file, and defaults to stdout. Function yywrap is called by lex when input is exhausted, return if we are done or 0 if more processing is required. Every C program requires a main function. In this case, we simply call yylex, the main entry-point for Lex. Some implementations of Lex include copies of main and yywrap

in a library, eliminating the need to code them explicitly. This is why, our first example, the shortest program, functioned correctly. Table 2.4 presents some of the most widely used predefined variables and functions of Lex.

Table 2.4: Lex predefined variables

Name	Function
int yylex(void)	call to invoke lexer, returns tokens
char *yytext	pointer to matched string
yyleng	length of matched string
yylval	value associated with the token
int yywrap(void)	wrapup, return 1 if done, 0 otherwise
FILE *yyout	output file
FILE *yyin	input file
INITIAL	initial start condition
BEGIN	condition switch start
ECHO	write matched string

Example 2.7 Here is a program that does nothing at all. All input is matched, but no action is associated with any pattern, so there will be no output.

```
%%
.
\n
```

Example 2.8 This example prepends line numbers to each line in a file. For example, consider a file having the following two lines as its content:

This is a book.
This book is on Compilers.

The Lex program given next will produce the output:

1. This is a book.
2. This book is on Compilers.

Some implementations of Lex predefine and calculate yylineno that input file for Lex is yyin that defaults to stdin.

```
    %{
        int yylineno;
    %}
    %%
    ^(.*)\n  printf(``%4d\t%s'', ++yylineno, yytext);
%%
    int main( int argc, char **argv) {
        yyin = fopen( argv[1], ``r'' );
```

```
    yylex();
    fclose(yyin);
}
```

The definitions section is composed of substitutions, code, and start states. Code in the definitions section is simply copied as is to the top of the generated C file and must be bracketed with "%{" and "%}" markers. Substitutions simplify pattern matching rules. For example, we may define digits and letters:

```
digit   [0-9]
letter  [A-Za-z]
%{
    int count;
%}
%%
        /* match identifier */
{letter}({letter}|{digit})*   count ++;
%%
int main( void ) {
    yylex();
    printf( ''number of identifiers = %d\n'', count);
    return 0;
}
```

Whitespace must separate the defining term and the associated expression. References to substitutions in the rules section are surrounded by braces to distinguish them from literals. When we have a match in the rules section, the associated C code is executed.

Example 2.9 Here is a scanner that counts the number of characters, words and lines in a file (similar to Unix wc):

```
%{
    int nchar, nword, nline;
%}
%%
\n          {nline ++; nchar ++;}
[^ \t\n]+   {nword ++; nchar += yyleng; }
.           {nchar ++;}
%%
int main( void ) {
    yylex();
    printf( ''%d\t%d\t%d\n'', nchar, nword, nline );
    return 0;
}
```

Strings. Quoted strings frequently appear in programming languages. Here is one way to match a string in Lex:

```
%{
    char *yylval;
    # include <string.h>
%}
%%
\‘‘[^’’\n]*[’’\n] {
        yylval = strdup(yytext + 1);
        if (yylval[yyleng − 2] != ’ " ’)
            warning(’’Improperly terminated string’’);
        else
            yylval[yyleng − 2] = 0;
        printf(’’found ‘%s’\n’’, yylval);
    }
```

The above example ensures that the strings don't cross line boundaries and remove enclosing quotes. If we wish to add escape sequences, such as \n or \“, start state simplify the matters:

```
%{
    char buf[100];
    char *s;
%}
%x STRING
%%
\‘‘                     {BEGIN STRING; s = buf; }
<STRING> \\n            {*s ++ = ’\n’; }
<STRING> \\t            {*s ++ = ’\t’; }
<STRING> \\\"           {*s ++ = ’\"’; }
<STRING> \"
                        {   *s = 0;
                                    BEGIN 0;
                                    printf("found ’%s’\n",buf);
                        }
<STRING> \n             {printf("Invalid string"); exit(1); }
<STRING>.               { *s++ = *yytext; }
```

Exclusive start state STRING is defined in the definition section. When the scanner detects a quote, the BEGIN macro shifts Lex into the STRING state. Lex stays in the STRING state, recognizing only patterns that begin with <STRING>, until another BEGIN is executed. Thus, we have a mini-environment for scanning strings. When the trailing quote is recognized, we switch back to state 0, the initial state.

Reserved words. If there exists a large collection of reserved words, it is more efficient to let Lex simply match a string, and determine in the code whether it is a variable or reserved word. For example, instead of coding

```
"if"    return IF;
"then"  return THEN;
```

```
"else"  return ELSE;
{letter}({letter}|{digit})* {
        yylval.id = symLookup( yytext );
        return IDENTIFIER;
    }
```

where symLookup returns an index into the symbol table, it is better to detect the reserved words and identifiers simultaneously, as follows:

```
tt{letter}({letter}|{digit})* {
    int i;
    if ((i = resWord( yytext )) != 0)
        return (i);
    yylval.id = symLookup( yytext );
    return (IDENTIFIER);
}
```

This technique significantly reduces the number of states required, and thus results in smaller scanner tables.

Debugging Lex. Lex has facilities that enable debugging. This feature may vary with different versions of Lex, so one should consult documentation for details. The code generated by Lex in the file lex.yy.c includes debugging statements that are enabled by specifying command line option "-d". Debug output in *flex* (a GNU version of Lex) may be toggled on and off by setting yy_flex_debug. Output includes the rule applied and the corresponding matched text. Alternatively, one can write his own debug code by defining functions that display information for the token value, and each variant of the yylval union. This is illustrated in the following example. When DEBUG is defined, the debug functions take effect, and a trace of tokens and associated values is displayed.

```
%union {
    int ivalue;
    ...
};
%{
    #ifdef DEBUG
        int dbgtoken( int tok, char *s ) {
            printf("token %s\n", s);
            return tok;
        }
        int dbgtokenIvalue( int tok, char *s ) {
            printf("token %s (%d)\n", s, yylval.ivalue);
            return tok;
        }
        # define RETURN(x) return dbgtoken(x, &x)
        # define RETURN_ivalue(x) return dbgtokenIvalue(x, &x)
```

```
    #else
        # define RETURN(x) return (x)
        # define RETURN_ivalue(x) return (x)
    #endif
%}
%%
[0-9]+   {
         yylval.ivalue = atoi( yytext );
         RETURN_ivalue( INTEGER );
         }
"if"     RETURN(IF);
"else"   RETURN(ELSE);
```

2.6 CONCLUSION

In this chapter, we have discussed how the words of a language can be specified using regular expressions. Acceptors have been designed for them using the nondeterministic and deterministic finite automata (NFA and DFA). Conversion from regular expression to NFA and from there to DFA has been illustrated. Finally, we have seen the tool lex for automated generation of lexical analyzer for a language.

EXERCISES

2.1 Describe the role of lexical analysis in the design of a compiler.

2.2 Is it possible to design a compiler without a distinct lexical analysis phase?

2.3 What is regular expression? Give its formal definition.

2.4 Distinguish between NFA and DFA. Compare their powers as token recognizer.

2.5 Write down the regular expressions for the following:

(a) identifiers of C language.

(b) binary strings such as a '0' is always followed by a '1'.

(c) to check an IP address for correct syntax, that is four groups of 1-3 digits separated by periods.

(d) to check correct syntax for the E-mail address.

(e) a sentence (something that begins with a capital letter and ends with a full stop).

(f) any decimal number that is a multiple of 5.

(g) any string whose length is a multiple of 5.

2.6 What does the following regular expressions mean?

(a) $a(ab)^*a$

(b) [0-7][0-7]*

(c) $(0|1)^*0(0|1)(0|1)(0|1)$

(d) $(a|b)^*abb$

(e) $(a|b)^*a(a|b)(a|b)(a|b)$

2.7 Construct NFAs for the regular expressions given in Problem 2.6.

2.8 Convert the NFAs created in Problem 2.7 to DFAs.

2.9 C++ has various integer constants. These are names for integer values. A simple sequence of decimal digits is always an integer constant. A sequence of hexadecimal digits preceded by "0x" is an integer constant. A sequence of octal digits preceded by "0" is an integer constant. An integer constant may be followed by "L" or "l" to indicate that it is a "long int". Write a regular expression for C++ integer constants as described here. Construct the corresponding NFA and DFA.

2.10 Write a LEX specification file to identify the tokens of the language C.

2.11 Write a LEX program that accepts the English language words (without bothering for meaning of course!) and replaces each occurrence of the string "abc" in it to "ABC".

2.12 Explain why is it not possible to design a lexical analysis tool that can detect the strings having equal number of two characters, say a and b.

Chapter 3

Syntax Analysis

Syntax analysis or parsing is the most important phase of a compiler. This extracts the syntactic units within the input stream. As noted earlier, syntax analyzer works hand-in-hand with previous lexical analysis phase. The lexical analyzer determines the tokens occurring next in the input stream and returns the same to the parser when asked for. The syntax analyzer considers the sequence of tokens for possible valid constructs of the programming language. In this chapter, we will deal with the syntax analysis phase. We will first discuss on the role of parser and parsing strategies in short. Based upon the complexity of the parsing strategies, there exists a number of parsing algorithms which are broadly classified as *top-down* and *bottom-up* strategies. Within each category, there are several parsers—like predictive parsers parsers (recursive and non-recursive), shift-reduce parsers (operator precedence, LR parsers), etc. We will gradually look into the construction of these parsers starting with simpler ones like predictive parsers to the complex ones like LR parsers.

3.1 ROLE OF PARSER

The parser looks into the sequence of tokens returned by the lexical analyzer and extracts the constructs of the language appearing within the sequence. Thus, the role of parser is two-fold:

1. To identify the language constructs present in a given input program. If the parser determines the input to be a valid one, it outputs a representation of the input in the form of a parse tree.
2. If the input is grammatically incorrect, the parser declares the detection of syntax error in the input. This case, no parse tree can be produced. The role of the parser is explained in Fig. 3.1.

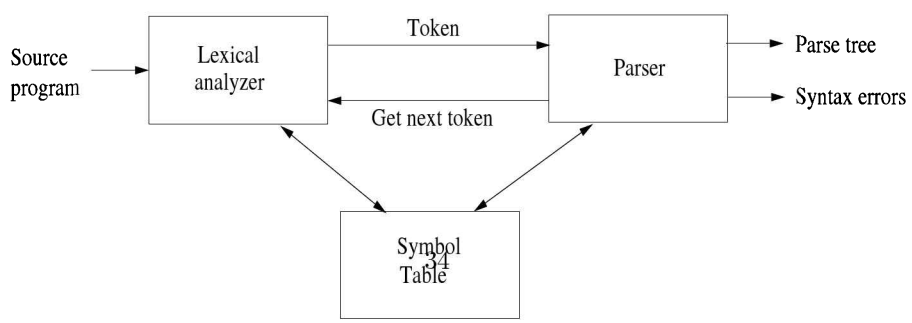

Figure 3.1: Role of parser.

3.2 ERROR HANDLING

Error handling is one of the most important feature of any modern compiler. It is highly unlikely that a compiler without good error handling capacity will be accepted by the users, even if it can produce correct code for the correct programs. The most important challenge, here, is to have a good guess of possible mistakes that a programmer can do and to come up with strategies to point those errors to the user in a very clear and unambiguous manner. It should be noted that the language designers do not provide information regarding errors, it is the compiler designer who has to do it. The common errors occurring in programs can be classified into four categories:

1. *Lexical Errors*—These errors are mainly the spelling mistakes and accidental insertion of foreign characters, for example '$', if the language does not allow it. They are mostly caught by the lexical analyzer.
2. *Syntactic Errors*—These are grammatical mistakes, such as unbalanced parentheses in arithmetic expressions, ill-formed constructs, etc. These errors occur mostly in the program. The parser should be able to catch these errors efficiently.
3. *Semantic Errors*—They involves errors due to undefined variables, incompatible operands to an operator, etc. These errors can be caught by introducing some extra checks during parsing.
4. *Logical Errors*—These are the errors such as infinite loops. There is not any way to catch the logical errors automatically. However, use of debugging tools may help the programmer to identify such errors.

Thus, an important challenge of syntax analysis phase is to detect syntactical errors. However, nobody would like a compiler that stops after detecting the first error because, there may be many more such errors. If all or most of these errors can be reported to the user in a single go, the user may correct all of them and resubmit them for compilation. Reporting a single error at a time necessitates a large number of iterations. But, in most of the cases, the presence of an error in the input stream leads the parser to an erroneous state, from where it cannot proceed further until certain portion of its work is undone. The strategies involved in the process are broadly known as *error-recovery strategies.*

Of the different recovery strategies, the following are worth mentioning:

- Panic mode
- Phrase level
- Error production
- Global correction

Panic mode recovery. In this case, the parser discards enough number of tokens to reach a descent state on the detection of an error. A set of tokens marking the end of language constructs is defined to make a *synchronizing set.* The parser discards tokens till it comes across a token from the synchronizing set. Typical examples of synchronizing tokens are semicolon, closing brace, *end*, etc., which are mostly used to terminate statements or blocks. This technique is effective because once the parser notices a synchronizing token, the effect of previous syntax error is most probably over. Although, it also skips a few more tokens in the process, which may introduce some spurious errors later.

Phrase level recovery. In this strategy, the parser makes some local corrections on the remaining input on detection of an error, so that the resulting input stream gives a valid construct of the language. Typical examples of this category include replacing a comma by a semicolon, inserting a missing character, etc. However, inserting a new character should be done carefully so that extraneous errors are not introduced and also the parsing algorithm can proceed without any problem.

Error production. This involves modifying the grammar of the language to include error situations. In this case, the compiler designer has a very good idea about the possible types of errors so that he can modify the grammar appropriately. The erroneous input program is, thus, valid for the new grammar augmented by error productions. Thus, the parser can always proceed.

Global corrections. This is a rather theoretical approach. The problem, here can be stated as following:

Given an incorrect input stream x for a grammar G, find another stream y acceptable by G, so that the number of tokens to be modified to convert x into y is the minimum.

The approach is definitely very costly, and the program created by the error recovery strategy may not be what the programmer had in his mind initially. Thus, the approach is not very practical.

Before going into discussions on parsers, we need to understand a few basic concepts and notations used for representing grammar.

3.3 GRAMMAR

A grammar G is defined as a 4-tuple $< V_N, V_T, P, S >$, where V_N is a set of nonterminal symbols used to write the grammar, V_T is the set of terminals (that is, the set of words of the language), P is a set of production rules and $S \in V_N$ is a special nonterminal called the *start symbol* of the grammar. The strings of the language are derived from S by applying the production rules from the set P. The rules are applied on intermediary strings consisting of sequence of symbols belonging to $V_N \cup V_T$. If the matching left hand side of a certain rule can be found to occur in the string, it may be replaced by the corresponding right hand side. Thus, the production rules define how a sequence of terminal and nonterminal symbols can be replaced by some other sequence.

Example 3.1 Consider a grammar to generate arithmetic expressions consisting of numbers and operator symbols $+$, $-$, $*$, $/$, and $\uparrow$. The rules of the grammar can be written as,

$$
\begin{aligned}
E &\rightarrow EAE \\
E &\rightarrow (E) \\
E &\rightarrow -E \\
E &\rightarrow \textbf{number} \\
A &\rightarrow + \\
A &\rightarrow - \\
A &\rightarrow *
\end{aligned}
$$

$$A \rightarrow /$$
$$A \rightarrow \uparrow$$

We can apply the rules to derive the expression "2*(3+5*4)" as follows:

$E \rightarrow EAE \rightarrow EA(E) \rightarrow EA(EAE) \rightarrow EA(EAEAE) \rightarrow EA(EAEA4) \rightarrow EA(EAE * 4) \rightarrow EA(EA5 * 4) \rightarrow EA(E + 5 * 4) \rightarrow EA(3 + 5 * 4) \rightarrow E * (3 + 5 * 4) \rightarrow 2 * (3 + 5 * 4)$

In this grammar, E and A are nonterminals, whereas the rest of the symbols, that is, $(,), -,$ **number**, $+, -, *, /, \uparrow$, are terminals.

There exists several classes of languages based on the production rules. A detailed discussion can be found in books on *Formal Languages.* Most of the programming language constructs belong to the class of *Context Free Languages.* The grammar of a context free language consists of production rules in which the left-hand side contains only a single nonterminal and no terminals. The name "context free" originates from the fact that since the left hand side has a single nonterminal, there is no context dependency for applying the rule in the derivation process.

3.3.1 Notational Conventions

In order to make the discussion clear and concise, we will use a few notational conventions throughout the book. The notations used are as follows:

1. The following will be taken as terminals:
   - Lower case letters occurring near the beginning of an alphabet, such as a, b, c etc.
   - All operator symbols.
   - Punctuation symbols including parentheses.
   - The digits.
   - Boldface strings such as **id**, **number**, **while**, etc.
2. The following will represent nonterminals:
   - Upper case letters occurring near the beginning of an alphabet, such as A, B, C etc.
   - The letter S to represent start symbol.
   - Lower case names like *expr, stmt* are also used to represent nonterminals.
3. Upper case letters in the later part of the alphabet, such as X, Y, Z represent grammar symbols (may be terminal or nonterminal).
4. Lower case letters later in the alphabet represent strings of terminals.
5. Lower case Greek letters like α, β etc. represent strings of grammar symbols.
6. A set of productions with same left hand side, like, $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, ..., $A \rightarrow \alpha_k$ is usually written as $A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_k$.
7. If no start symbol is mentioned, then the nonterminal appearing in the left hand side of the first production rule is taken as the start symbol.

3.3.2 Derivation

The sequence of intermediary strings generated to expand the start symbol of the grammar to a desired string of terminals is called a derivation. Each step of derivation modifies an intermediary string to a new one by replacing a substring of it that matches with the left hand side of a production rule by the string on the right hand side of the rule. The derivation can also be represented in the form of a tree, called *parse tree.* For example, the derivation of "2*(3+5*4)" in Example 3.1 can be represented as a parse tree shown in Fig 3.2.

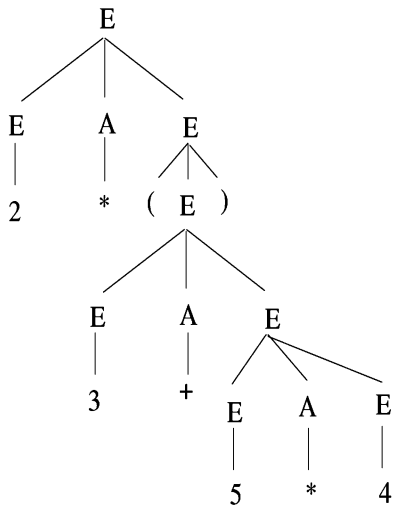

Figure 3.2: Parse tree for "2*(3+5*4)".

Leftmost and Rightmost Derivations. The derivation in which the leftmost nonterminal is always replaced at each step is called *leftmost derivation.* Similarly, the derivation in which the rightmost nonterminal is replaced at each step is called *rightmost derivation.* In a leftmost derivation, the intermediate strings are called *left sentential forms.* It consists of a sequence of grammar symbols, both terminals and nonterminals. On the other hand, the intermediary strings of a rightmost derivation are called *right sentential forms.* The rightmost derivation is also called the *canonical representation.*

3.3.3 Ambiguity

A grammar is said to be ambiguous if there exists more than one parse tree for the same sentence. An ambiguous grammar can have more than one leftmost and rightmost derivations. For example, consider the grammar,

$$S \rightarrow aSbS \mid bSaS \mid \epsilon$$

For the string "*abab*", we have two different parse trees as shown in Fig 3.3.

A classical example of ambiguous grammar is that of *if-then-else* construct of many programming languages. Most of the languages have both *if-then* and *if-then-else* versions of the statement. The grammar rules for it are as follows:

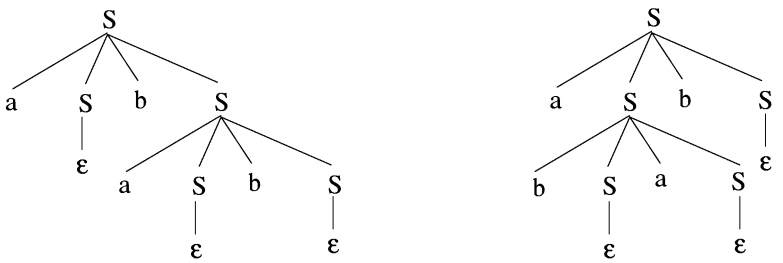

Figure 3.3: Two parse trees for "*abab*".

stmt → **if** *condition* **then** *stmt* **else** *stmt*
| **if** *condition* **then** *stmt*

Now, consider the following code segment,

```
if a > b then
    if c > d then x = y
    else x = z
```

We can have two different parse trees as shown in Fig 3.4(a) and (b). The first figure shows the situation in which the *else* is taken with the outer *if*-statement. In the second case, the *else* is taken with the inner *if*. Thus, outer one is an *if-then* statement while the inner one is *if-then-else* statement. Most of the programming languages accept second one as the correct syntax, that is, *else* if associated with the innermost *if*.

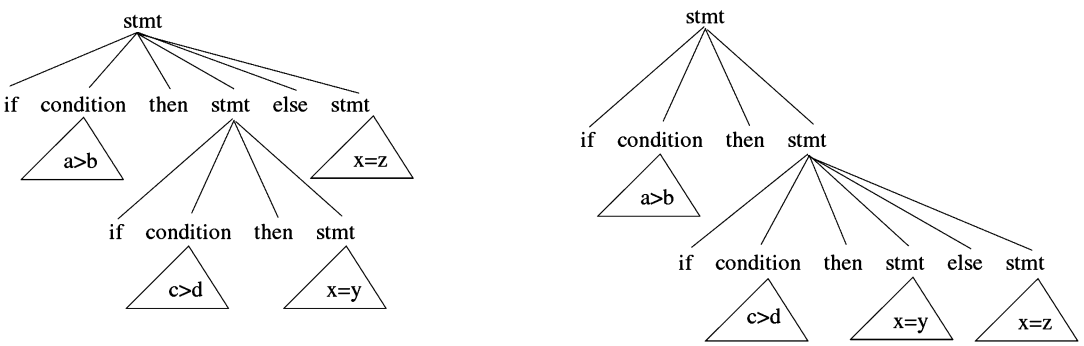

Figure 3.4: Two parse trees for *if-then-else* statement.

Eliminating Ambiguity. Ambiguities may be eliminated by rewriting the grammar. For example, the *if-then-else* grammar may be rewritten as follows.

stmt → *matched_stmt* | *unmatched_stmt*
matched_stmt → **if** *condition* **then** *matched_stmt* **else** *matched_stmt*
| **other_stmt**
unmatched_stmt → **if** *condition* **then** *stmt*
| **if** *condition* **then** *matched_stmt* **else** *unmatched_stmt*

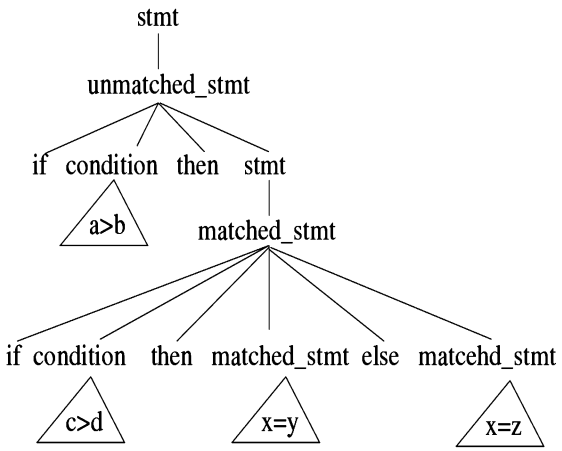

Figure 3.5: Parse tree for *if-then-else*.

Here, **other_stmt** represents all other statements apart from **if**. The only parse tree possible now is shown in Fig 3.5.

Another technique to resolve ambiguity may be to modify the language a bit. Many languages require that an *if* should have a matching *endif*. Thus, the grammar is modified as,

$$\begin{aligned} \textit{stmt} \quad &\rightarrow \textbf{if } \textit{condition } \textbf{then } \textit{stmt } \textbf{else } \textit{stmt } \textbf{endif} \\ &\mid \textbf{if } \textit{condition } \textbf{then } \textit{stmt } \textbf{endif} \end{aligned}$$

Here, due to the presence of *endif*, the ambiguity is not there and only a single parse tree is possible.

3.3.4 Left Recursion

A grammar is *left recursive* if it has a nonterminal, say A, that has a derivation of $A\alpha$ from it. Presence of left recursion creates difficulties while designing the corresponding parsers as we will see later. Left recursion may be of two types:

- immediate left recursion
- general left recursion

An immediate left recursion happens with a nonterminal A having production rule of the form $A \rightarrow A\alpha \mid \beta$. The immediate left recursion can be eliminated by introducing a new nonterminal symbol, say A' thus modifying the grammar. The grammar rule $A \rightarrow A\alpha \mid \beta$ is modified as,

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \epsilon \end{aligned}$$

Thus, the rule $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$ can be modified as,

$$\begin{aligned} A &\rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A' \\ A' &\rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \epsilon \end{aligned}$$

Example 3.2 Consider the following left-recursive grammar for arithmetic expressions,

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \textbf{id}$$

Elimination of immediate left recursion from the rules modifies the grammar as,

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \epsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \epsilon$$
$$F \rightarrow (E) \mid \textbf{id}$$

However, even if there may be no immediate left recursion, a number of production rules may act together to give a general left recursion. For example, consider the grammar,

$$\begin{array}{lcl} S & \rightarrow & Aa \\ A & \rightarrow & Sb \mid c \end{array}$$

Here, S is left recursive, because $S \rightarrow Aa \rightarrow Sba$. This form of general left recursion can be eliminated with the following algorithm. The algorithm is guaranteed to work if the grammar has no cycles. Thus, it is not possible to derive a nonterminal, say A, starting from itself and no ϵ productions.

Algorithm eliminate_left_recursion

1. Arrange nonterminals in some order, say A_1, A_2, ..., A_m.
2. For $i = 1$ to m do
   For $j = 1$ to $i - 1$ do
   For each set of productions $A_i \rightarrow A_j\gamma$ and $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$
   Replace $A_i \rightarrow A_j\gamma$ by $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$
3. Eliminate immediate left recursion from all productions.

For example, consider the grammar,

$$\begin{array}{lcl} S & \rightarrow & Aa \\ A & \rightarrow & Sb \mid c \end{array}$$

Let the order of nonterminals be S, A. For $i = 1$, the rule $S \rightarrow Aa$ is through, since there is no immediate left recursion. For $i = 2$, $A \rightarrow Sb \mid c$ is modified as, $A \rightarrow Aab \mid c$, which has immediate left recursion and hence, is eliminated by modifying the rule as,

$$\begin{array}{lcl} A & \rightarrow & cA' \\ A' & \rightarrow & abA' \mid \epsilon \end{array}$$

Next, we start our discussion on parsing strategies. First, we will look into top-down parsing.

3.4 TOP-DOWN PARSING

Top-down parsers get the name from the fact that they try to find a derivation of the input stream from the start symbol of the grammar. Equivalently, it can be viewed as an attempt to construct the parse tree rooted at the start symbol of the grammar for the input stream. There are two main approaches for top-down parsing:

- Recursive descent parsing
- Predictive parsing

3.4.1 Recursive Descent Parsing

These parsers consist of a set of mutually recursive routines that may require backtracking to create the parse tree. Thus, it may require repeated scanning of the input.

Example 3.3 Consider the grammar

$$S \rightarrow abA$$
$$A \rightarrow cd \mid c \mid \epsilon$$

For the input stream ab, the recursive descent parser starts by constructing a parse tree representing $S \rightarrow abA$ as shown in Fig. 3.6(a). In Fig. 3.6(b), the tree is expanded with the production $A \rightarrow cd$. Since it does not match the string ab, the parser backtracks and then, tries the alternative $A \rightarrow c$ as shown in Fig 3.6(c). However, here also the parse tree does not match the string ab. So, the parser backtracks and tries out the alternative $A \rightarrow \epsilon$. This time, it finds a match. Thus, the parsing is complete and successful.

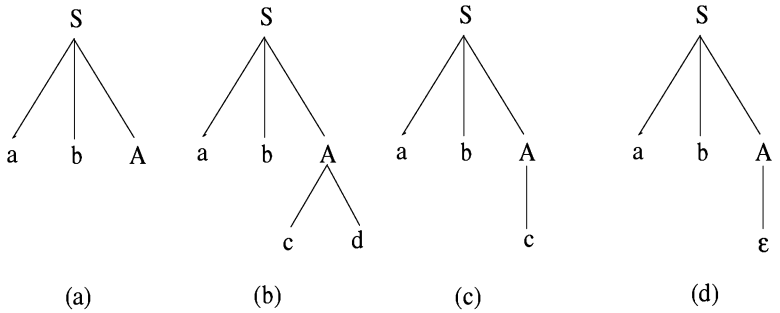

Figure 3.6: Example of parse tree construction.

One way to make the generation of a recursive descent parser is to map BNF forms onto program constructs. In general, we can write a procedure for each non-terminal in the (post lexical analysis) language definition. Within these procedures, the coding is semi-automatic.

1. Where the right hand side contains alternatives, the next symbol to be input provides the selector for a switch statement, each branch of which represents one alternative.
2. If ϵ is an alternative, this is the `default` of the switch.
3. Where a repetition occurs this may be implemented iteratively, that is, by a `while` or `until` statement, with greater efficiency than is possible for recursion.

4. The sequence of actions within each branch, possibly only one, consists of an inspection of the lexical token for each terminal, and a procedure will be assigned to a local or global variable, as required by the semantics of the language.

Example 3.4 A recursive descent calculator
The following is a code for a simple calculator performing integer arithmetic involving '+' and '*', implemented using recursive descent strategy. It consists of a set of mutually recursive routines. The routine *expr* evaluates an expression. It calls the routine *term* to find the value of a term created by multiplying a few factors. An individual factor may be an integer or an expression within parentheses. The routine *factor* makes a call to *expr* to evaluate expressions within parentheses.

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

    void err(const char* s)
    {
        perror(s);
        exit(0);
    }
    int getint(int ch)
    {
        int val;
        if (isdigit(ch) ) val = ch − '0';
        else err("Illegal character in place of integer");
        while ( ((ch = getch() != EOF) &&isdigit(ch) )
            val = val * 10 + ch − '0';
        ungetc(ch, stdin);
        return val;
    }

    int expr();

    int factor()
    {
        int ch, val;
        printf(``f \n'');
        switch (ch = getch())
        {
        case '(':  val = expr();
            if ((ch = getch() != ')') err("Missing closing parenthesis in factor");
            break;
        case '0':
        case '1':
        case '2':
        case '3':
```

```
        case '4':
        case '5':
        case '6':
        case '7':
        case '8':
        case '9':  val = getint(ch);
            break;
        default:  err("Illegal character at start of factor");
        }
        return val;
    }

    int term()
    {
        int val, ch;
        printf("t \n");
        val = factor();
        while ((ch = getch()) == '*') val = val * factor();
        ungetc(ch,stdin);
        return val;
    }

    int expr()
    {
        int ch, val;
        printf("e \n");
        val = term();
        while ((ch = getch()) == '+') val = val + term();
        if (ch == ')') ungetc(Ch, stdin);
        return val;
    }

    void main()
    {
        printf("%d\n", expr());
    }
```

It should be noted that if the grammar is left-recursive, a recursive descent parser (even in the presence of backtracking), may fall into an infinite loop. This happens because of the fact that for a left-recursive rule, the parser has to expand without consuming any further input symbol. Now, we will present a parser construction strategy, known as *predictive parser* to create a recursive descent parser that does not need backtracking. The predictive parser can be constructed in both recursive and non-recursive manner. We will first discuss the recursive construction and then the non-recursive one.

3.4.2 Recursive Predictive Parsing

For a grammar free of left-recursion, we can construct a recursive predictive parser consisting of a set of mutually recursive procedures. For example, if the grammar rules be,

$$\begin{aligned} \textit{statement} \rightarrow\ & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{statement} \\ \mid\ & \textbf{while}\ \textit{expr}\ \textbf{do}\ \textit{statement} \end{aligned}$$

then by looking into the next input token (IF or WHILE), we can identify the rule to be tried out to obtain the parse tree. However, consider the case,

$$\begin{aligned} \textit{statement} \rightarrow\ & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{statement}\ \textbf{else}\ \textit{statement} \\ \mid\ & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{statement} \end{aligned}$$

Here, by looking at the first token (IF for both the cases), it is not possible to identify the rule to be applicable. Only after *then-part*, if token ELSE is found, we know the first rule to be used. This necessitates backtracking if token ELSE is absent in the input stream, that is, it is an *if-then* statement. To get rid of this problem, the grammar is *left-factored* to take out the common portion separately as follows:

$$\begin{aligned} \textit{statement} \rightarrow\ & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{statement}\ \textit{else-clause} \\ \textit{else-clause} \rightarrow\ & \textbf{else}\ \textit{statement} \\ \mid\ & \epsilon \end{aligned}$$

Note that we still have an alternative in a right-hand side, but there is no *significant common prefix* any more. Thus, it is possible to decide which alternative is valid from the first symbol of the *else-clause.*

After left-factoring, the resultant rules can also be represented in the form of a set of transition diagrams. For this, for each nonterminal A, we can create a diagram as follows:

1. Create an initial and a final state.
2. For each rule $A \rightarrow X_1 X_2 \ldots X_k$, insert an additional $(k-1)$ intermediary states with an edge from the initial state to *state-1* labelled by X_1, from *state 1* to *state 2* labelled by X_2 and so on, finally from *state-*$(k-1)$ to the final state labelled by X_k.

For a given input stream, we can now traverse the transition diagrams starting with the start state of the transition diagram corresponding to the start symbol of the grammar. When we traverse an edge labelled by a terminal symbol, the input symbol is consumed. On the other hand, if the edge is labelled by a nonterminal symbol, we jump to the start state of the transition diagram corresponding to it. When we reach the final state of this new diagram (by consuming further inputs and/or calling other transition diagrams), we come back to the caller transition diagram in the next state. Finally, when no more progress can be made, if we have consumed the entire input stream and reached the final state of the transition diagram for the start symbol, the stream is accepted. Now, we will discuss construction and simplification of transition diagrams for an example grammar.

Example 3.5 Consider the grammar for arithmetic expressions

$$\begin{aligned} \textit{exp} &\rightarrow \textit{exp} + \textit{term} \mid \textit{term} \\ \textit{term} &\rightarrow \textit{term} * \textit{factor} \mid \textit{factor} \\ \textit{factor} &\rightarrow (\ \textit{exp}\ ) \mid \textbf{id} \end{aligned}$$

After removal of left-recursion we get,

$$
\begin{array}{lcl}
\textit{exp} & \rightarrow & \textit{term exp\_tail} \\
\textit{exp\_tail} & \rightarrow & + \textit{ term exp\_tail} \mid \epsilon \\
\textit{term} & \rightarrow & \textit{factor term\_tail} \\
\textit{term\_tail} & \rightarrow & * \textit{ factor term\_tail} \mid \epsilon \\
\textit{factor} & \rightarrow & (\textit{ exp }) \mid \textbf{id}
\end{array}
$$

This gives the transition diagrams shown in Fig. 3.7.Simplification of these diagrams may create a more compact and efficient parser.

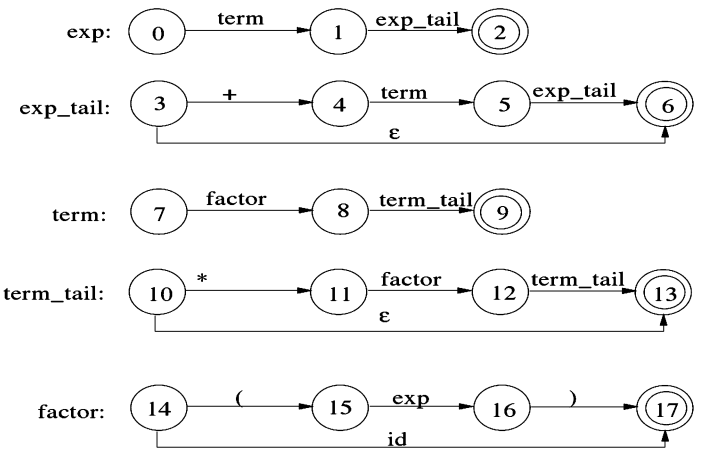

Figure 3.7: Transition diagram for expression grammar.

First, we eliminate self-recursion in the *exp_tail* transition diagram, substituting an iterative model. This is shown in Fig. 3.8.

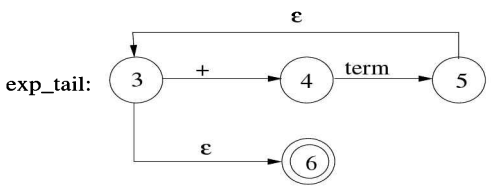

Figure 3.8: Simplified diagram.

Further simplifying, by removing the redundant ϵ-edge from 5 to 3, we get Fig. 3.9.

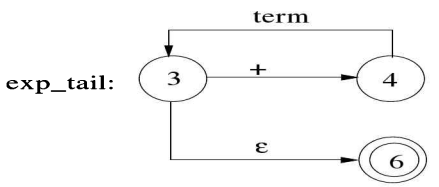

Figure 3.9: Simplified diagram.

If we substitute the *exp_tail* transition diagram by the *exp* one, replacing the *exp_tail* edge from 1 to 2, we get Fig. 3.10.

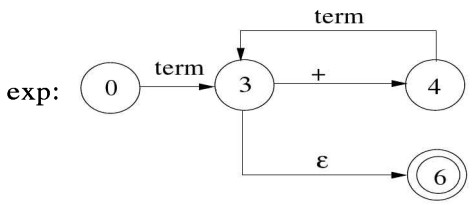

Figure 3.10: Simplified diagram.

This, in turn, can be simplified to Fig. 3.11.

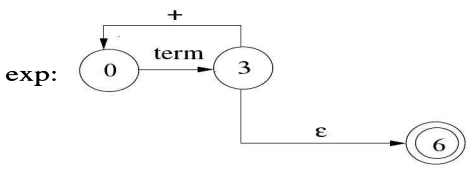

Figure 3.11: Simplified diagram.

Applying the same approach to *term* and *term_tail*, we get a reduced set of diagrams for arithmetic expressions in Fig. 3.12.

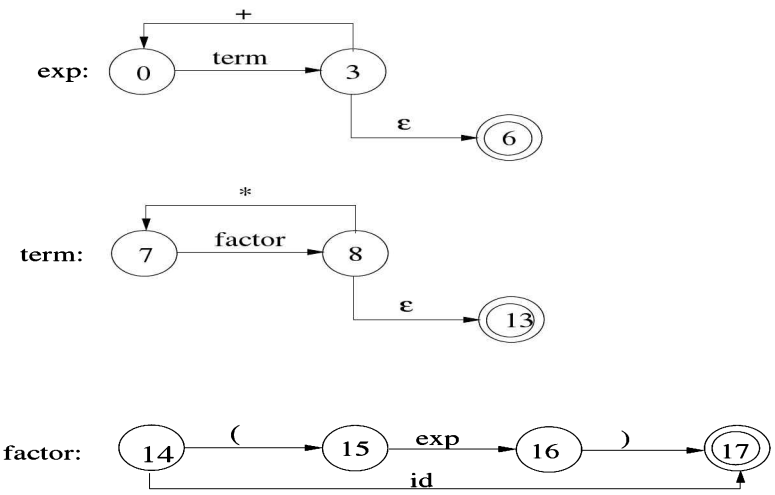

Figure 3.12: Final set of transition diagrams for expression grammar.

3.4.3 Non-recursive Predictive Parsing—LL(k) Parsing

It is possible to automatically generate table-driven top-down parsers. These parsers scan over the input stream using a prefix of tokens to identify the production applied. The language accepted by these parsers is called $LL(k)$, where k is the length of the prefix. "LL" stands

for *left-to-right* scanning of the input stream and *left-most* derivation respectively. The term *left-most* derivation implies that at each step, the left most nonterminal is replaced by a string of terminals and nonterminals as guided by the grammar rules. For the remaining section, we will consider the case of $k = 1$.

The construction of an LL(1) parser for a grammar (V_N, V_T, P, S) requires first computing the following properties:

$$\begin{array}{lcl} \text{First}(A) & = & \{U | U \in V_T \text{ and } A \text{ can derive a string starting with } U\ \} \\ \text{Follow}(A) & = & \{U | U \in V_T \text{ and } S \text{ can derive a string like } \alpha A U \beta\} \\ \text{First}(\alpha) & = & \{U | U \in V_T \text{ and } \alpha \text{ can derive a string like } U\beta\ \} \end{array}$$

Intuitively, First(A) is the set of terminals that can start a string that is derivable from A, Follow(A) is the set of terminals that can follow an occurrence of A and First(α) is the set of terminals that can start a string that is derivable from α. To compute these sets for a grammar, we need to define another property:

$$\text{nullable}(A) = \begin{cases} \text{true} & \text{if } A \text{ can derive } \epsilon \text{ in zero or more steps} \\ \text{false} & \text{otherwise} \end{cases}$$

We say that a nonterminal A is *nullable* if *nullable(A) = true* and a terminal symbol is never nullable. Then, the algorithm for computing First and Follow sets for a grammar is as follows:

```
for all A ∈ N do
    First(A) = φ;
    Follow(A) = φ;
for all U ∈ T do
    First(U) = {U};
do
    for each A → X1 ..... Xn ∈ P do
        for each i ∈ [1...n] do
            if X1, ... Xi−1 are all nullable then
                First(A) = First(A) ∪ First(Xi)
            if Xi+1, ... Xn are all nullable then
                Follow(Xi) = Follow(Xi) ∪ Follow(A)
            for each j ∈ [i + 1, ..., n] do
                if Xi+1, ...,Xj−1 are all nullable then
                    Follow(Xi) = Follow(Xi) ∪ Follow(Xj)
        if all the Xi's are nullable then nullable(A) = true
until First, Follow and nullable sets do not change
```

Note that to simplify the presentation, we extend the notion of First and Follow sets to terminal symbols. We can avoid doing so by adding extra conditional statements to the algorithm. We extend the notion of First sets to include ϵ and define them for strings as follows:

$$\begin{array}{rcl} \text{First}(\epsilon) & = & \{\epsilon\} \\ \text{First}(X\ \ \alpha) & = & \begin{cases} X & \text{if } X \in T \\ \text{First}(X) & \text{if nullable}(X) = \text{false} \\ \text{First}(X) \cup \text{First}(\alpha) & \text{if nullable}(X) = \text{true} \end{cases} \end{array}$$

A grammar is an LL(1) grammar if all productions conform to the following LL(1) conditions:

1. For each production $A \rightarrow \sigma_1 \mid \sigma_2 \mid \ldots \mid \sigma_n$, $\text{First}(\sigma_i) \cap \text{First}(\sigma_j) = \phi, \forall i \neq j$
2. If nonterminal X can derive an empty string, then $\text{First}(X) \cap \text{Follow}(X) = \phi$

The first condition means that we always know which condition to choose. The second condition means that we always know whether something optional is there or not.

Example 3.6 The grammar $S \rightarrow A \mid B$, $A \rightarrow cA + b \mid a$, $B \rightarrow cB + a \mid b$ is not LL(1), since $\text{First}(A) \cap \text{First}(B) = \{c\}$.

Once we have computed the First and Follow sets, we can construct an LL(1) parse table M that maps pairs of nonterminals and terminals to productions using the following algorithm:

```
for each A → α ∈ P do
    if ϵ ∈ First(α) then
        for each U ∈ Follow(A) do
            Add A → α to M[A, U]
    for each U ∈ First(α) do
        Add A → α to M[A, U]
```

If the resulting table has atmost one production per (A, U) pair, then the grammar is LL(1). For example, consider the grammar,

$$\begin{aligned} E &\rightarrow ME' \\ E' &\rightarrow \epsilon \\ E' &\rightarrow +ME' \\ M &\rightarrow AM' \\ M' &\rightarrow \epsilon \\ M' &\rightarrow *AM' \\ A &\rightarrow \textbf{num} \\ A &\rightarrow (E) \end{aligned}$$

The *nullable, first,* and *follow* sets are given in the following table:

Symbol	Nullable	First	Follow
E	false	{**num**, (}	{), \$}
E'	true	{+}	{), \$}
M	false	{**num**, (}	{), +, \$}
M'	true	{*}	{), +, \$}
A	false	{**num**, (}	{), +, *, \$}

Note that we have included the special "end-of-input" symbol (\$) in the Follow set. The parsing table for the grammar is

	num	+	*	(	)	$
E	$E \to ME'$			$E \to ME'$		
E'		$E' \to +ME'$			$E' \to \epsilon$	$E' \to \epsilon$
M	$M \to AM'$			$M \to AM'$		
M'		$M' \to \epsilon$	$M' \to *AM'$		$M' \to \epsilon$	$M' \to \epsilon$
A	$A \to$ **num**			$A \to (E)$		

The table-driven parser for LL(1) grammar works using a parsing table, as constructed above and an auxilliary stack of grammar symbols. The table-driven parser works as follows:

while *stack is not empty* **do**
  Let X be the top symbol on the stack
  Let U be the next input symbol
  if $X \in T$ **then**
    if $X = U$ **then**
      pop X off the stack and advance the input
    else
      parsing error
  elsif $M[X, U] = A \to Y_1 \dots Y_n$ then
    pop X and push $Y_n, \dots, Y_1$ onto the stack
  else
    parsing error

Here is a trace of running this algorithm on the string "1+2*3" using the above parsing table:

Stack	Input	Action
E	1+2*3$	parse $E \to ME'$
$E'M$	1+2*3$	parse $M \to AM'$
$E'M'A$	1+2*3$	parse $A \to$ **num**
$E'M'$**num**	1+2*3$	advance input
$E'M'$	+2*3$	parse $M' \to \epsilon$
E'	+2*3$	parse $E' \to +ME'$
$E'M+$	+2*3$	advance input
$E'M$	2*3$	parse $M \to AM'$
$E'M'A$	2*3$	parse $A \to$ **num**
$E'M'$**num**	2*3$	advance input
$E'M'$	*3$	parse $M' \to *AM'$
$E'M'A*$	*3$	advance input
$E'M'A$	3$	parse $A \to$ **num**
$E'M'$**num**	3$	advance input
$E'M'$	$	parse $M' \to \epsilon$
E'	$	parse $E' \to \epsilon$
	$	done

Example 3.7 Consider the following grammar for **if-then-else** statement.

$$\begin{aligned} S &\to \textbf{if expr then } S\ S' \mid \textbf{assignment} \\ S' &\to \textbf{else } S \mid \epsilon \end{aligned}$$

To simplify the grammar, we have assumed **expr** to be a terminal modelling an expression, and **assignment** another terminal modelling the assignment statement. The parsing table is shown in Table 3.1. The entry $M[S'$, **else**$]$ is multiply defined. This is because the Follow(S')

Table 3.1: An example LL parsing table

	if	**expr**	**then**	**assignment**	**else**	$
S	$S \rightarrow$ **if expr then** S S'			$S \rightarrow$ **assignment**		
S'				$S' \rightarrow \epsilon$ $S' \rightarrow$ **else** S	$S' \rightarrow \epsilon$	

$=$ {**else**, \$}. First(**else** S) $=$ {**else**}, forcing the entry $S' \rightarrow$ **else** S to $M[S'$, **else**$]$. Now, for the rule $S' \rightarrow \epsilon$, First$(\epsilon) = \{\epsilon\}$, and **else** is in Follow(S'), thus, $S' \rightarrow \epsilon$ is added to $M[S'$, **else**$]$. The situation is justified since the dangling else problem occurs after getting the token **else** – the options are either to expand through the else branch or consider the statement seen so far as an **if-then** statement.

3.5 BOTTOM-UP PARSING

Bottom-up parsing is based on the reverse process to top-down. Instead of expanding successive nonterminals according to production rules, a current string or *right-sentential form* is collapsed each time until the start non-terminal is reached to predict the legal next symbol. This can be regarded as a series of reductions. The approach is also known as *shift-reduce* parsing. This is the primary parsing method for many compilers, mainly because of its speed and the tools which automatically generate a parser based on the grammar. The techniques for shift-reduce parsing use certain concepts that we will define now.

Definition 3.1 **Handle:** *A handle is a right sentential sequence which is the rightmost derivation within the string being parsed. The shift-reduce sequence results in the reverse of a series of rightmost derivations to produce the string of terminals forming the total input stream.*

Consider the grammar,

$$S \rightarrow aABe \tag{3.1}$$
$$A \rightarrow Abc|b \tag{3.2}$$
$$B \rightarrow d \tag{3.3}$$

and the sentence **abbcde**. Parsing by bottom up methods, give the following series of reductions:

$$\begin{aligned} abbcde & \\ aAbcde & \quad \text{by (3.2)} \\ aAde & \quad \text{by (3.2)} \\ aABe & \quad \text{by (3.3)} \\ S & \quad \text{by (3.1)} \end{aligned}$$

Reverse of this gives:

$$\begin{aligned} S &\Rightarrow \\ aABe &\Rightarrow \\ aAde &\Rightarrow \\ aAbcde &\Rightarrow \\ abbcde, & \end{aligned}$$

which is clearly a series of rightmost derivations. The question is how do we decide which reduction corresponds to a rightmost derivation. It is not necessarily the rightmost reduction, nor always the leftmost. How, then, do we define the handles at various stages? A second problem arises when the same right sentential form is found in more than one production. How do we choose which production to reduce to? For instance, in some programming languages both

$$\begin{aligned} \textit{proc-call} &\rightarrow \textbf{id}\ ((\textit{exp}\ (,\ \textit{exp})^*)^?), \text{ where } (r)^? \text{ stands for 0 or 1 occurrence of } r \\ \textit{array-var} &\rightarrow \textbf{id}\ (\textit{exp}\ (,\ \textit{exp})^*) \end{aligned}$$

are productions, none as a procedure or function call and the other as a subscripted variable. The sub-string $X(3)$ fits both.

The final problem concerns ambiguous grammars, where more than one rightmost derivation can lead to a particular string. Bottom up parsing has to be able to choose between two or more valid handles. Thus, for the grammar

$$\begin{aligned} \textit{exp} &\rightarrow \textit{exp} + \textit{exp} \\ \textit{exp} &\rightarrow \textit{exp} * \textit{exp} \\ \textit{exp} &\rightarrow (\ \textit{exp}\ ) \\ \textit{exp} &\rightarrow \textbf{id}, \end{aligned}$$

we can have the two following rightmost derivations

$$\begin{aligned} &\textit{exp} \rightarrow \textit{exp} + \textit{exp} \\ &\rightarrow \textit{exp} + \textit{exp} * \textit{exp} \\ &\rightarrow \textit{exp} + \textit{exp} * \mathbf{id_3} \\ &\rightarrow \textit{exp} + \mathbf{id_2} * \mathbf{id_3} \\ &\rightarrow id_1 + \mathbf{id_2} * \mathbf{id_3} \end{aligned} \qquad \begin{aligned} &\textit{exp} \rightarrow \textit{exp} * \textit{exp} \\ &\rightarrow \textit{exp} * \mathbf{id_3} \\ &\rightarrow \textit{exp} + \textit{exp} * \mathbf{id_3} \\ &\rightarrow \textit{exp} + id_2 * \mathbf{id_3} \\ &\rightarrow id_1 + id_2 * \mathbf{id_3} \end{aligned}$$

In the first derivation, * has higher precedence while in the second, + has higher precedence.

Implementing shift reduction. The normal way to view a shift reduce parser is to use the notion of

an input stream, containing
a phrase to be parsed and
a stack holding symbols

The input stream holds terminals, the stack can hold a mixture of terminals and nonterminals, the later generated by earlier reductions. The operation *shift* moves a symbol from the input to the stack. The operation *reduce* combines the sequence ending with the last terminal shifted to form a nonterminal on the stack. When the string is exhausted, the single start symbol should be presented assuming that all reductions have been performed.

The question is how to decide when to shift and when to reduce and, if we choose to reduce, which reduction to apply.

	Sequence of shift/reduce/accept for a string	
Stack	Input	Action
$	**id1** + **id2** * **id3** $	shift
$ **id1**	+ **id2** * **id3** $	reduce using exp → **id**
$ exp	+ **id2** * **id3** $	shift
$ exp +	**id2** * **id3** $	shift
$ exp + **id2**	* **id3** $	reduce using exp → **id**
$ exp + exp	* **id3** $	shift
$ exp + exp *	**id3** $	shift
$ exp + exp * **id3**	$	reduce using exp → **id**
$ exp + exp * exp	$	reduce using exp → exp * exp
$ exp + exp	$	reduce using exp → exp + exp
$ exp	$	accept

Problems in Shift-Reduce Parsing: The two most common problems are the *shift/reduce conflict* and the *reduce/reduce conflict*. Shift/reduce conflicts occur when the parser cannot decide whether to shift a token onto the stack or reduce a handle. Reduce/reduce conflicts are similar as the name implies: the parser cannot decide which reduction to make. The classic *if-then-else* problem demonstrates shift/reduce conflicts.

Stack	Input
$ ... **if** <expr> **then** <stmt>	**else** ...

Here, the parser does not know whether to reduce the **if** <*expr*> **then** <*stmt*> handle or to shift **else** onto the stack. Of course, this can easily be resolved by simply carrying a rule that says, in this case, "always shift".

Reduce/reduce conflicts are less common, but somewhat more insidious. Assume that in our language, we call functions and access arrays with the same syntax: **id** (< *list* >) where <*list*> is a list of parameters or expressions consisting of *id*s.

Stack	Input
$ ... **id** (**id**	, **id**) ...

The question here is, do we reduce the *id* at the top of the stack to an <*expr*> or a <*param*>. Since both of them have the same syntax, it is simply not known. In order to fix it, one must change the context somehow. This can be done by changing the identifier for a procedure call to be **procid**.

Now, we will look into two main classes of bottom-up parsing—*operator precedence* and

LR. Operator precedence is a simpler parser to construct and of course has lesser power than LR parsers.

3.5.1 Operator Precedence Parsing

The operator precedence parsing technique can be applied to *Operator grammars.*

Definition 3.2 Operator grammar: A grammar is said to be operator grammar if there does not exist any production rule with right hand side

1. as ϵ
2. two nonterminals appearing consecutively, that is, without any terminal between them

Operator precedence parsing is not a simple technique to apply to most of the language constructs, but it evolves an easy technique to implement where a suitable grammar may be produced.

We consider its use to parse arithmetic expressions with the following grammar,

$$\begin{aligned} exp &\rightarrow exp\ operator\ exp \mid (exp) \mid -exp \mid \mathbf{id} \\ operator &\rightarrow + \mid - \mid * \mid / \mid \uparrow \end{aligned}$$

Rewriting this as an operator grammar is straightforward:

$$\begin{aligned} exp \rightarrow\ & exp + exp \mid exp \text{ - } exp \mid exp * exp \mid exp / exp \mid exp \uparrow exp \mid (\ exp\ ) \mid \\ & -exp \mid \mathbf{id} \end{aligned}$$

Here, no two consecutive nonterminals are found in any production.

We now define precedence relations between certain pairs of terminals. These relations are defined by disjoint relational symbols $<\cdot$, $\doteq$, $\cdot>$. Roughly speaking, where we wish * to have higher precedence than that of +, we define the relations:

$$\begin{aligned} + &<\cdot * \text{ and} \\ * &\cdot> + \end{aligned}$$

In other cases, we may need to define *term1* $\cdot>$ *term2* and *term1* $<\cdot$ *term2.* This is not a contradiction, since the relations are disjoint. Finally, no precedence may be defined for some pairs. The idea is to build a table of relations, such that a handle will be defined by a sequence with $<\cdot$ marking its start, $\doteq$ possibly the middle and $\cdot>$ at the end. A part of the table for our grammar is given below:

	id	+	*	$
id		$\cdot>$	$\cdot>$	$\cdot>$
+	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$
*	$<\cdot$	$\cdot>$	$\cdot>$	$\cdot>$
$	$<\cdot$	$<\cdot$	$<\cdot$	

$ is a special imaginary terminal marking each end of the string to be parsed. $ is always $<\cdot$ any other terminal and any other terminal is always $\cdot>$$, as shown in the table.

To parse our string, we can now apply these relations, ignoring the non-terminals. Thus,

$$\mathbf{id1} + \mathbf{id2} * \mathbf{id3} \Rightarrow \$ <\cdot \mathbf{id1} \cdot> + <\cdot \mathbf{id2} \cdot> * <\cdot \mathbf{id3} \cdot> \$$$

We scan the resulting sequence left to right until the first $\cdot >$ and then, backtrack to the immediately left $< \cdot$ ignoring any $\doteq$ relations. The $< \cdot$ and $\cdot >$ enclose the handle, in terms of terminals. Any intervening or adjacent nonterminals are also taken to be in the handle. In our example, initially there are no nonterminals. Clearly, **id1** is the first handle and this is reduced to *exp* giving:

$$exp + \textbf{id2} * \textbf{id3}$$

Applying the same process twice more gives us

$$exp + exp * exp$$

From this, we derive

$$\$ < \cdot + < \cdot * \cdot > \$$$

which has $exp * exp$ as its handle. This is then reduced reflecting its greater precedence compared to $+$. Lastly, we will find $exp + exp$ as a handle and reduce this. The reductions generate a parse tree in reverse. In each reduction, a new node is formed with links down to the symbols from which it was reduced.

The operator precedence parsing algorithm works as follows:

```
Initialize stack to $
while () do
    Let U be the topmost terminal on the stack
    Let V be the next input symbol
    if U = $ and V = $ then return
    if U <·V or U ≐ V then
        Shift V onto the stack
        Advance the input pointer
    elseif U·> V then
        do
            Pop the topmost symbol, call it V, from the stack
        until the top of the stack is <·V
    else
        error
    end
end
```

3.5.2 Establishing Precedence Relationships

The intuitive approach may allow us to assign precedences to conventional operators, but clearly, we need to be able to include all terminals in the grammar. Some more general approach is needed. Following *Bornat*, we can use the following method, based on *first operator* and *last operator* lists. It builds a precedence matrix along the lines of our expression table. *Firstop+* is a list of all those terminal symbols ("operators") which can appear first on any right hand side of a production. *Lastop+* is a similar list of those terminals which can appear last. The steps are:

1. For each nonterminal, that is, left hand side, construct a *Firstop* list containing the first terminal in each production for that nonterminal. Where a non-terminal is the first symbol on the right hand side, include both it and the first terminal following, e.g. for

$$X \rightarrow a \ldots | Bc$$

include a, c and B in X's *Firstop* list.

2. Similarly, construct a *Lastop* list for each non-terminal, e.g. for

$$Y \rightarrow \ldots u | \ldots vW$$

include u, v and W in Y's Lastop list.

3. Compute the *Firstop*+ and *Lastop*+ lists using Warshall's Closure Algorithm, as follows:
   (a) Take each nonterminal in turn, in any order and look for it in all the *Firstop* lists. Add its own first symbol list to any other in which it occurs. Similarly process the *Lastop* lists.
   (b) The nonterminals may now be deleted from the lists.

4. Construct the precedence matrix by the following rules
   (a) Whenever terminal a immediately precedes non-terminal B in any production, put $a <\cdot \alpha$, where α is any terminal in the *Firstop*+ list for B.
   (b) Whenever a terminal b immediately follows non-terminal C in any production, put $b \cdot > \beta$, where β is any terminal in the *Lastop*+ list for C.
   (c) Whenever a sequence aBc or ac occurs in any production, put $a \doteq c$.

5. Add the relations $\$ <\cdot a$ and $a \cdot > \$$ for all terminals in the *Firstop*+ and *Lastop*+ lists, respectively, for S.

Any entry left blank indicates that the two symbols should never occur consecutively in a handle. Such a sequence is a syntax error.

Example 3.8 To clarify this, consider the grammar for expressions:

$$\begin{array}{lcl} S & \rightarrow & A \\ A & \rightarrow & T \mid A+T \mid A-T \\ T & \rightarrow & F \mid T*F \mid T/F \\ F & \rightarrow & P \mid P \uparrow F \\ P & \rightarrow & i \mid n \mid (A) \end{array}$$

Applying steps 1 and 2 gives us

Symbol	Firstop	Lastop
S	A	A
A	$T + A-$	$T + -$
T	$F * T/$	$F * /$
F	$P \uparrow$	$P \uparrow F$
P	$i\ n\ ($	$i\ n\ )$

Applying step 3 gives us

Symbol	Firstop+	Lastop+
S	$T + A - F * P \uparrow i\ n\ ($	$AT + -F * /P \uparrow i\ n\ )$
A	$T + A - F * P \uparrow i\ n\ ($	$T + -F * /P \uparrow i\ n\ )$
T	$F * T/P \uparrow i\ n\ ($	$F * /P \uparrow i\ n\ )$
F	$P \uparrow i\ n\ ($	$P \uparrow Fi\ n\ )$
P	$i\ n\ ($	$i\ n\ )$

Removing the nonterminals gives

Symbol	Firstop+	Lastop+
S	$+ - * \uparrow i\ n\ ($	$+ - */ \uparrow i\ n\ )$
A	$+ - * \uparrow i\ n\ ($	$+ - */ \uparrow i\ n\ )$
T	$*/ \uparrow i\ n\ ($	$*/ \uparrow i\ n\ )$
F	$\uparrow i\ n\ ($	$\uparrow i\ n\ )$
P	$i\ n\ ($	$i\ n\ )$

Finally, we use steps 4 and 5 to compute our precedence relation matrix as follows.

	$	(	)	i	n	$\uparrow$	*	/	+	−
$		$<\cdot$		$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$
(		$<\cdot$	$\doteq$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$
)	$\cdot>$		$\cdot>$			$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$
i	$\cdot>$		$\cdot>$			$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$
n	$\cdot>$		$\cdot>$			$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$
$\uparrow$	$\cdot>$	$<\cdot$	$\cdot>$	$<\cdot$	$<\cdot$	$<\cdot$	$\cdot>$		$\cdot>$	$\cdot>$
*	$\cdot>$	$<\cdot$	$\cdot>$	$<\cdot$	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$
/	$\cdot>$	$<\cdot$	$\cdot>$	$<\cdot$	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$
+	$\cdot>$	$<\cdot$	$\cdot>$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$
−	$>$	$<\cdot$	$>$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$

3.5.3 Error Recovery

There are two basic types of error conditions:

- No precedence relation holds between the current top of stack terminal and the next symbol to be input.
- No production has a right sentential form to match the handle found.

To cope with type 1 errors, we must change the stack or the input or both. We may change, insert or delete extra input symbols to try to produce a *sensible* handle. Insertion could lead to an infinite loop. A *safe* approach is to ensure that whatever is done will allow the next input symbol to be shifted. In general, a recovery procedure should be specified for each blank entry in the precedence table.

To cope with type 2 errors, we try to produce a *close approximation* to the handle from

the legal set of right hand sides and then report the differences as the error. For example '( )' is a handle and we would like to approximate it to '(*A*)' and report *Missing expression within parentheses.*

3.6 LR PARSING

One of the best methods for syntactic recognition of programming languages is LR parsing. An LR parser uses the *shift-reduce* technique discussed earlier. The L stands for left-to-right scanning and the R for a rightmost derivation in reverse.

When we speak about LR parsing, we are actually speaking of LR(1) parsing, that is, LR parsing with one symbol lookahead. In general, we can have LR(k) parsing, with k symbols of lookahead.

The advantages of LR parsing are numerous:

1. An LR parser can recognize virtually all programming language constructs written with context-free grammars.
2. It is the most general nonbacktracking technique known.
3. It can be implemented in a very efficient manner.
4. The languages it can recognize is a proper superset of that for predictive parsers.
5. It can detect syntax errors quickly.

The primary disadvantage to LR parsers is that it is far too much work to manually create LR parsing tables. However, tools exist to automatically generate an LR parser from a given grammar. These are called LR parser generators, such as *yacc, bison* etc. These parser generators are not only useful in creating the parser, but also in finding errors in the grammar.

3.6.1 LR Parsing Methods

There are actually three different methods to perform LR parsing.

1. **SLR** stands for simple LR. It is easy to implement, but is less powerful than the other parsing methods.
2. **Canonical LR** is the most general and powerful, but is tedious and costly to implement. For the same grammar, the canonical LR parser has got much more number of states as compared to the simple LR parser.
3. **LALR** stands for lookahead LR. It is a mix of SLR and canonical LR, but can be implemented efficiently. It contains the same number of states as the simple LR parser for the same grammar.

Most parser generators generate LALR parsers, since they are the trade-off between power and efficiency.

3.6.2 LR Parsing Algorithm

An LR parser has nearly the same form as a nonrecursive predictive parser, except the parsing table which consists of two parts—*action* and *goto*. The algorithm used is much the same as the shift-reduce method.

The stack for an LR parser consists of *grammar symbols* and *states*. The states summarize what is below that state on the stack and pairing the state on top of the stack with the next input symbol indices into the parsing table to determine the next action.

An LR parsing configuration has the following form:

$$(s_0 X_1 s_1 \ldots X_m s_m, a_i a_{i+1} \ldots a_n \$),$$

where s_j is a state, X_k is a grammar symbol and a_l is an input symbol. The configuration is just a pair with the current stack and the current unused input.

The LR parsing table, as mentioned, is splitted into the *action* and *goto* sections. The *goto* section actually implements a finite automaton that recognizes the viable prefixes of the grammar. The *action* portion is indexed by *action*[s_m,a_i] to perform the following actions:

1. **shift** pushes the next input symbol and next state (s) onto the stack to enter the configuration

$$(s_0 X_1 \ldots X_m s_m a_i s, a_{i+1} \ldots a_n \$)$$

2. **reduce** matches the handle to some production $A \rightarrow \beta$. It then pops $r = 2 * |\beta|$ elements off the stack and pushes A and $s = goto[s_{m-r}, A]$. The configuration ends up as

$$(s_0 X_1 \ldots X_{m-r} s_{m-r} A s, a_{i+1} \ldots a_n \$)$$

3. **accept** means the parser is finished.
4. **error** causes the parser to call an error handling routine of some sort, as it has found an illegal configuration.

Algorithm LR Parsing
Input: LR parsing table and input string w
Output: true, if parse is successful
Begin
    Push initial state
    repeat forever
        Let s = top of stack, a = next input symbol
        if action[s,a] = shift s' **then**
            Push a and s'
            Advance input pointer to the next input symbol
        else if action[s,a] = reduce $A \rightarrow \beta$ **then**
            Pop $2|\beta|$ symbols
            s' = top of stack
            Push A and goto[s', A]
        else if action[s,a] = accept **then** return true
        else error
End

3.6.3 Constructing LR Parsing Tables

Obviously, the table is an essential part of LR parsing. But, how does one go about making the tables. It is, in fact, a rather mechanical process. However, it is a daunting task if one does not use automated tools for the purpose.

We will look at creating tables for the three different types of LR parsers (SLR, Canonical LR, LALR). The first will be SLR, since it is the simplest.

One important fact to be kept in mind while constructing LR parsing tables is that the state on the top of the stack provides a wealth of information to the parser. Essentially, an LR parser is keeping track of viable prefixes for the handles. It uses an automaton to recognize these prefixes. The *goto* portion of the table simulates this automaton, but it does not need to scan the stack on every input symbol to figure out what state it is in. Instead, it is kept on the top of the stack.

Items

The first concept in LR table construction is that of an *item*. An item is a production rule with a position indicator (dot) at some point on the right hand side. If $A \rightarrow XYZ$ is a production, the possible items of this production are:

$$\begin{aligned} A &\rightarrow \cdot XYZ \\ A &\rightarrow X \cdot YZ \\ A &\rightarrow XY \cdot Z \\ A &\rightarrow XYZ \cdot \end{aligned}$$

Items are also known as *LR(0)* items, since they assume no lookahead. Intuitively, an item denotes how much of a production we have seen so far during the parsing.

SLR Parsing Tables

What we want to do while creating a SLR parsing table is to make a finite automaton that accepts the viable prefixes of the grammar. In order to do this, we use sets of items as our states and grammar symbols to define the transition function. For this, we will step through the process of creating a *canonical LR(0)* collection, that is, collections of items. To do this, we *augment* the grammar and use two functions—*closure* and *goto*.

Definition 3.3 Augmented grammar: *An augmented grammar simply has a new "dummy" start symbol, whose only production is the start symbol of the grammar in question. If G is our grammar with start symbol S, then the augmented grammar $G' = (V_T, V_N \cup \{S'\}, S', F \cup \{S' \rightarrow S\})$.*

Definition 3.4 Closure operation: *Let I be a set of items. closure(I) is defined by the following rules:*

1. *All elements of I are in closure(I).*
2. *If $A \rightarrow \alpha \cdot B\beta \in$ closure(I) and $B \rightarrow \gamma$ is a production, then add $B \rightarrow \cdot\gamma$ to closure(I). Continue until no new items are added to closure(I).*

Definition 3.5 **Goto operation:** *The goto operation is a function goto(I,X), where I is a set of items and X is a grammar symbol. It is defined as the closure of the set of all items* $[A \rightarrow \alpha X \cdot \beta]$ *such that* $[A \rightarrow \alpha \cdot X\beta] \in I$.

We can think of the goto operation as consuming the next input in the production. If we look at the dot as the position of the automata, we want to take the closure of all the items in I that would be advanced on the input X.

Creating the Sets of Items
Algorithm Set of Items for SLR parsing table

Input: Augmented grammar G'
Output: Set of items I_0, ..., I_n
Begin
$C \leftarrow \{closure(\{S' \rightarrow \cdot S\})\}$
repeat
for each $X \in V_n \cup V_T$
for each $I_k \in C$ with $goto(I,X) \neq \phi$
if $goto(I,X) \notin C$ **then**
$C \leftarrow C \cup goto(I,X)$
until no more items can be added to C
End.

The sets of items I_k are, in fact, the results of the *goto* function. The initial set I_0 is the closure of $S' \rightarrow \cdot S$. The next sets are the result of running $goto(I_0,X)$ for each symbol X in the grammar. Each set of items created represents a state in the finite automata to recognize the viable prefixes of the grammar. This is because *goto* is defined on a grammar symbol. It says that all items in the set I_k can be advanced on input X to the set created by the *goto* call.
Creating the SLR Parsing Table: The following are the steps to create the SLR parsing table:

1. Construct the collection of sets of LR(0) items.
2. State i is constructed from I_i. We determine the actions in the table as follows:
   (a) If $[A \rightarrow \alpha \cdot a\beta] \in I_i$ and $goto(I_i,a) = I_j$ with $a \in V_T$, then action$[i,a]$ = shift j.
   (b) If $[A \rightarrow \alpha \cdot] \in I_i$, then action$[i, a]$ = reduce $A \rightarrow \alpha$ for all $a \in$ Follow$(A)$with $A \neq S'$.
   (c) If $[S' \rightarrow S \cdot] \in I_i$, then action$[i,\$]$ = accept.

   If any entry is multiply defined, the grammar is ambiguous and not SLR(1). We can then quit.
3. If $goto(I_i, A) = I_j$ then goto$[i, A] = j$.
4. All entries not defined are marked "errors".
5. The initial state is the set of items containing $[S' \rightarrow \cdot S]$

Example 3.9 Let us consider the following augmented grammar for constructing the SLR parsing table:

$$S' \rightarrow S$$

$$\begin{aligned} S &\rightarrow aABe \\ A &\rightarrow Abc \\ A &\rightarrow b \\ B &\rightarrow d \end{aligned}$$

The sets of items for the augmented grammar computed using the process outlined above is:

$$\begin{aligned} I_0 &= \{[S' \rightarrow \cdot S], [S \rightarrow \cdot aABe]\} \\ I_1 &= \{[S' \rightarrow S\cdot]\} \\ I_2 &= \{[S \rightarrow a \cdot ABe], [A \rightarrow \cdot Abc], [A \rightarrow \cdot b]\} \\ I_3 &= \{[S \rightarrow aA \cdot Be], [A \rightarrow A \cdot bc], [B \rightarrow \cdot d]\} \\ I_4 &= \{[A \rightarrow b\cdot]\} \\ I_5 &= \{[S \rightarrow aAB \cdot e]\} \\ I_6 &= \{[A \rightarrow Ab \cdot c]\} \\ I_7 &= \{[B \rightarrow d\cdot]\} \\ I_8 &= \{[S \rightarrow aABe\cdot]\} \\ I_9 &= \{[A \rightarrow Abc\cdot]\} \end{aligned}$$

The *goto* function for this grammar is:

$$\begin{aligned} goto(I_0, S) &= I_1 \\ goto(I_0, a) &= I_2 \\ goto(I_2, A) &= I_3 \\ goto(I_2, b) &= I_4 \\ goto(I_3, B) &= I_5 \\ goto(I_3, b) &= I_6 \\ goto(I_3, d) &= I_7 \\ goto(I_5, e) &= I_8 \\ goto(I_6, c) &= I_9 \end{aligned}$$

We can use a graph to describe both the canonical item sets and the *goto* function. The graph corresponding to this grammar is shown in Fig. 3.13.

The Follow sets for the grammar are as follows:

$$\begin{aligned} \text{Follow}(S) &= \{\$\} \\ \text{Follow}(A) &= \{b, d\} \\ \text{Follow}(B) &= \{e\} \end{aligned}$$

The resulting SLR parsing table is shown in Table 3.2. Now, consider the string "*abbcbcde*". The operation of the LR parsing algorithm on this string has been show in Table 3.3.

Limitations of SLR Parsers: SLR parsers are sometimes limited in the sense that even for unambiguous grammar they may result into a parse table with shift/reduce conflicts. The

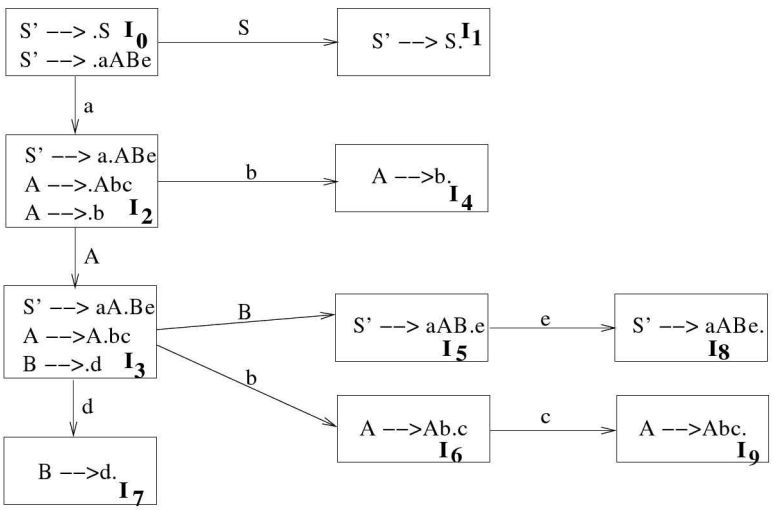

Figure 3.13: Graph representing canonical LR(0) items.

Table 3.2: Parsing table for Example 3.9

	a	b	c	d	e	\$	S	A	B
I_0	s2						1		
I_1						accept			
I_2		s4						3	
I_3		s6		s7					5
I_4		r4		r4					
I_5					s8				
I_6			s9						
I_7					r5				
I_8						r2			
I_9		r3		r3					

main reason behind this is that the follow set is often too rough an estimate as to what can come next. It is elaborated by the following example.

Example 3.10 Consider the following grammar for C-style variable assignments:

$$
\begin{aligned}
S &\rightarrow E\$ \\
E &\rightarrow L = R \\
E &\rightarrow R \\
L &\rightarrow *R \\
L &\rightarrow \mathbf{id} \\
R &\rightarrow L
\end{aligned}
$$

Table 3.3: SLR operation of Example 3.9

Stack	Input	Action
0	$abbcbcde\$$	shift
$0a2$	$bbcbcde\$$	shift
$0a2b4$	$bcbcde\$$	reduce $A \rightarrow b$
$0a2A3$	$bcbcde\$$	shift
$0a2A3b6$	$cbcde\$$	shift
$0a2A3b6c9$	$bcde\$$	reduce $A \rightarrow Abc$
$0a2A3$	$bcde\$$	shift
$0a2A3b6$	$cde\$$	shift
$0a2A3b6c9$	$de\$$	reduce $A \rightarrow Abc$
$0a2A3$	$de\$$	shift
$0a2A3d7$	$e\$$	reduce $B \rightarrow d$
$0a2A3B5$	$e\$$	shift
$0a2A3B5e8$	$\$$	reduce $A \rightarrow aABe$
$0A1$	$\$$	accept

Consider the two states:

I_0: $S \rightarrow \cdot E\$$
$E \rightarrow \cdot L = R$
$E \rightarrow \cdot R$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot$ **id**
$R \rightarrow \cdot L$

I_1: $E \rightarrow L\cdot = R$
$R \rightarrow L\cdot$

We have a shift/reduce conflict in I_1. The Follow of R is the union of the Follow of E and the Follow of L, that is, it is $\{\$,=\}$. In this case, $=$ is a member of the Follow of R, so we cannot decide shift or reduce by using just one lookahead as in the case of SLR parsing.

We will now describe an even more powerful grammar - LR(1). This grammar is not used in practice because of the large number of states it generates. A simplified version of this grammar, called LALR(1), has the same number of states as LR(0). It is far more powerful than LR(0) but less powerful than LR(1). Both LR(1) and LALR(1) check one lookahead token. They read one token ahead from the input stream (in addition to the current token). An item used in LR(1) and LALR(1) is like an LR(0) item but with the addition of a set of expected lookaheads which indicate what lookahead tokens would make us perform a reduction when we are ready to reduce using the production rule. The expected lookahead symbols for a rule $X \rightarrow a$ are always a subset or equal to Follow(X). The idea is that an item in an itemset represents a potential for reduction using the rule associated with the item. But, each itemset, that is, state, can be reached after a number of transitions in the state diagram, which means that each itemset has an implicit context, which, in turn, implies that there are only few terminals permitted to appear in the input stream after reducing by this rule. In SLR, we made the assumption that the followup tokens after reduction by $X \rightarrow a$ are exactly equal to Follow(X). But, this is too conservative and may not help us resolve the conflicts. So, the

idea is to make a more careful analysis by taking into account the context in which the item appears, thus restricting the followup tokens. This will reduce the possibility of overlappings in shift/reduce or reduce/reduce conflict state.

LR(1) Parsing

Definition 3.6 LR(1) grammar: *A grammar is said to be LR(1) if in a single left-to-right scan, we can construct a reverse rightmost derivation, while using atmost a single token lookahead to resolve ambiguities. A more general case appears with LR(k) parsers doing k token lookaheads.*

LR(k) Items: The table construction algorithms use LR(k) items to represent the set of possible states in a parser. An LR(k) item is a pair $[\alpha;\ \beta]$, where

1. α is a production from G with a '$\cdot$' at some position in the right hand side.
2. β is a lookahead string containing k symbols (terminals or \$).

For example, $[A \rightarrow X \cdot YZ;\ a]$ is an LR(1) item. LR(1) items have lookahead strings of length 1. Several LR(1) items may have the same *core*, that is, a production with '$\cdot$' at the same position. For example,

$$[A \rightarrow X \cdot YZ; a] \quad \text{and} \tag{3.4}$$
$$[A \rightarrow X \cdot YZ; b] \tag{3.5}$$

have the same core and can be represented together as

$$[A \rightarrow X \cdot YZ; \{a, b\}]$$

Usage of LR(1) Lookahead: The usage of lookahead symbols are as follows:

1. Carry them along to allow us to choose correct reduction when there is any choice.
2. Lookaheads are bookkeeping, unless item has a '$\cdot$' at right end.
   (a) in $[A \rightarrow X \cdot YZ; a]$, a has no direct use
   (b) in $[A \rightarrow XYZ\cdot; a]$, a is useful
3. Allows use of grammars that are not *uniquely invertible.* A grammar G is *uniquely invertible* if no two productions have the same right hand side. For example, for the two items $[A \rightarrow \alpha\cdot, a]$ and $[B \rightarrow \alpha\cdot, b]$, we can decide between reducing to A or to B by looking at limited right context.

Canonical LR(1) Items
To construct the canonical collection, we need two functions:

- closure(I)
- goto($I; X$)

LR(1) closure
Given an item $[A \rightarrow \alpha \cdot B\beta; a]$, its closure contains the item and any other items that can generate legal substrings to follow α. Thus, if the parser has a viable prefix α on its stack, the input should reduce to $B\beta$ (or γ for some other item $[B \rightarrow \cdot\gamma; b]$ in the closure).

function closure(I)
    repeat
        new_item $\leftarrow$ false
        for each item $[A \rightarrow \alpha \cdot B\beta; a] \in I$,
                each production $B \rightarrow \gamma \in G'$,
                and each terminal $b \in FIRST(\beta a)$ do
            if $[B \rightarrow \cdot\gamma; b] \notin I$ **then**
                add $[B \rightarrow \cdot\gamma; b]$ to I
                new_item $\leftarrow$ true
            end if
    until (new_item = false)
    return I

LR(1) goto
Let I be a set of LR(1) items and X be a grammar symbol. Then, goto$(I; X)$ is the closure of the set of all items $[A \rightarrow \alpha X \cdot \beta; a]$ such that $[A \rightarrow \alpha \cdot X\beta; a] \in I$. If I is the set of valid items for some viable prefix γ, then goto$(I; X)$ is the set of valid items for the viable prefix γX, which represents the state after recognizing X in state I.

function goto(I, X)
    $J \leftarrow$ set of items $[A \rightarrow \alpha X \cdot \beta; a]$
        such that $[A \rightarrow \alpha \cdot X\beta; a] \in I$
    $J' \leftarrow$ closure(J)
    return J'

Collection of sets of LR(1) items
We start the construction of the collection of sets of LR(1) items with the item $[S' \rightarrow \cdot S; \$]$, where

- S' is the start symbol of the augmented grammar G'
- S is the start symbol of G, and
- \$ is the right end string marker

procedure items(G')
    $C \leftarrow \{$closure$(\{[S' \rightarrow \cdot S; \$]\})\}$
    repeat
        new_item $\leftarrow$ false
        for each set of items I in C, each grammar symbol X such that goto$(I;X) \neq \phi$ and
            goto$(I;X) \notin C$ do
                add goto$(I;X)$ to C
                new_item $\leftarrow$ true
        end for
    until (new_item = false)

LR(1) table construction
The algorithm to construct the LR(1) parsing table consists of the following steps:

1. Construct the collection of sets of LR(1) items for G'.
2. State i of the parser is constructed from I_i
   (a) if $[A \rightarrow \alpha \cdot a\gamma; b] \in I_i$ and goto$(I_i; a) = I_j$, then set action$[i, a]$ to "shift j". (a must be a terminal)
   (b) if $[A \rightarrow \alpha\cdot; a] \in I_i$, then set action$[i, a]$ to "reduce $A \rightarrow \alpha$".
   (c) if $[S' \rightarrow S\cdot; \$] \in I_i$, then set action$[i, \$]$ to "accept".
3. If goto$(I_i; A) = I_j$, then set goto$[i, A]$ to j.
4. All other entries in action and goto are set to "error".
5. The initial state of the parser is the state constructed from the set containing the item $[S' \rightarrow \cdot S; \$]$.

Example 3.11 Let us consider the following augmented grammar for construction of LR(1) parsing table.

$$\begin{aligned} goal &\rightarrow expr \\ expr &\rightarrow term + expr \\ expr &\rightarrow term \\ term &\rightarrow factor * term \\ term &\rightarrow factor \\ factor &\rightarrow \mathbf{id} \end{aligned}$$

The set of items for the grammar are as follows:

I_0 : $[goal \rightarrow \cdot expr, \$], [expr \rightarrow \cdot term + expr, \$], [expr \rightarrow \cdot term, \$], [term \rightarrow \cdot factor * term, \{+, \$\}]$, $[term \rightarrow \cdot factor, \{+, \$\}], [factor \rightarrow \mathbf{id}, \{+, *, \$\}]$

I_1 : $[goal \rightarrow expr\cdot, \$]$

I_2 : $[expr \rightarrow term\cdot, \$], [expr \rightarrow term \cdot + expr, \$]$

I_3 : $[term \rightarrow factor\cdot, \{+, \$\}], [term \rightarrow factor \cdot * term, \{+, \$\}]$

I_4 : $[factor \rightarrow \mathbf{id}\cdot, \{+, *, \$\}]$

I_5 : $[expr \rightarrow term + \cdot expr, \$], [expr \rightarrow \cdot term + expr, \$], [expr \rightarrow \cdot term, \$]$, $[term \rightarrow factor * term, \{+, \$\}], [term \rightarrow \cdot factor, \{+, \$\}], [factor \rightarrow \cdot \mathbf{id}, \{+, *, \$\}]$

I_6 : $[term \rightarrow factor * \cdot term, \{+, \$\}], [term \rightarrow \cdot factor * term, \{+, \$\}], [term \rightarrow \cdot factor, \{+, \$\}]$, $[factor \rightarrow \cdot \mathbf{id}, \{+, *, \$\}]$

I_7 : $[expr \rightarrow term + expr\cdot, \$]$

I_8 : $[term \rightarrow factor * term\cdot, \{+, \$\}]$

The LR(1) automaton for the grammar has been shown in Figure 3.14. The corresponding parse table is shown in Table 3.4.

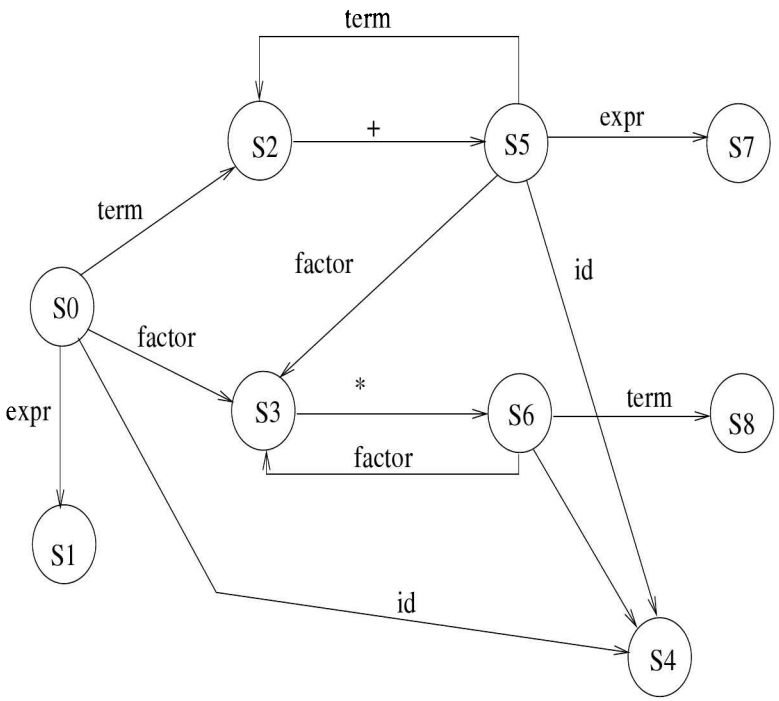

Figure 3.14: LR(1) automaton.

Table 3.4: LR(1) parsing table

	ACTION				GOTO		
	id	+	*	$	expr	term	factor
0	s4				1	2	3
1				accept			
2		s5		s3			
3		s5	r6	r5			
4		r6	r6	r6			
5	s4				7	2	3
6	s4					8	3
7				r2			
8		r4		r4			

LALR(1) Parsing

With LALR parsing, we attempt to reduce the number of states in an LR(1) parser by merging states differing only in their lookahead sets. Typically, SLR and LALR tables for a grammar will always have the same number of states which for a C like language will be several hundred. A LR(1) parser would have several thousand states for the same language. To start with, let

us consider the LR(1) set of items for the following grammar:

$$S' \rightarrow S$$
$$S \rightarrow CC$$
$$C \rightarrow cC|d$$

The set of items are as follows:

$$\begin{array}{lcl} I_0 & : & [S' \rightarrow \cdot S, \$], [S \rightarrow \cdot CC, \$], [C \rightarrow \cdot cC, \{c, d\}], [C \rightarrow \cdot d, \{c, d\}] \\ I_1 & : & [S' \rightarrow S\cdot, \$] \\ I_2 & : & [S \rightarrow C \cdot C, \$], [C \rightarrow .cC, \$], [C \rightarrow \cdot d, \$] \\ I_3 & : & [C \rightarrow c \cdot C, \{c, d\}], [C \rightarrow \cdot cC, \{c, d\}], [C \rightarrow \cdot d, \{c, d\}] \\ I_4 & : & [C \rightarrow d\cdot, \{c, d\}] \\ I_5 & : & [C \rightarrow CC\cdot, \$] \\ I_6 & : & [C \rightarrow c \cdot C, \$], [C \rightarrow .cC, \$], [C \rightarrow .d, \$] \\ I_7 & : & [C \rightarrow d\cdot, \$] \\ I_8 & : & [C \rightarrow cC\cdot, \{c, d\}] \\ I_9 & : & [C \rightarrow cC\cdot, \$] \end{array}$$

In the above example, some of the states can be merged since they differ only in their lookahead sets. Specifically, states I_4 and I_7, I_3 and I_6, I_8 and I_9. This gives rise to the states:

$$\begin{array}{lcl} I_{47} & : & [C \rightarrow d\cdot, \{c, d, \$\}] \\ I_{36} & : & [C \rightarrow c \cdot C, \{c, d, \$\}], [C \rightarrow \cdot c, \{c, d, \$\}], [C \rightarrow \cdot d, \{c, d, \$\}] \\ I_{89} & : & [C \rightarrow cC\cdot, \{c, d, \$\}] \end{array}$$

But, isn't this just SLR all over again? In this case, it is, since when we do the merging we end up with the complete Follow sets. However, this is not always the case. For example, consider the following grammar:

$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & Bbb|aab|bBa \\ B & \rightarrow & a \\ I_0 & : & [S' \rightarrow \cdot S, \$], [S \rightarrow .Bbb, \$], [S \rightarrow \cdot aab, \$], [S \rightarrow \cdot bBa, \$], [B \rightarrow \cdot a, b] \\ I_1 & : & [S' \rightarrow S\cdot, \$] \\ I_2 & : & [S \rightarrow B \cdot bb, \$] \\ I_3 & : & [S \rightarrow a \cdot ab, \$], [B \rightarrow a\cdot, b] \\ & & \ldots\ldots \end{array}$$

In SLR parsing, a shift-reduce conflict would occur in state 3, because on input 'a', we can either shift or reduce with $B \rightarrow b$. In LALR, however, there is a similar shift, but a reduction occurs only if a b comes up next.

Conflicts in LALR mergings

An important question that arises while doing these merging is can merging states ever introduce new conflicts. A shift-reduce conflict can never be introduced by a merging unless the conflict is already present in the LR(1) configuration sets. For example, we have two states:

$$\begin{array}{lll} \text{I245:} & A \to w\cdot, b & \text{I3882:} \quad A \to w\cdot, b \\ & B \to \ldots, d & \qquad\quad B \to ub \cdot z, c \end{array}$$

If we are allowed to merge these two states together, the core productions must be in both. So, the shift-reduce error had to be in the original states somewhere. Reduce-reduce conflicts, however, are another story. Consider the following grammar:

$$\begin{array}{rcl} S' & \to & S \\ S & \to & aBc \mid bCc \mid aCd \mid bBd \\ B & \to & e \\ C & \to & e \end{array}$$

The LR(1) sets of items are:

$$\begin{array}{rcl} I_0 & : & [S' \to \cdot S, \$], [S \to \cdot aBc, \$], [S \to \cdot bCc, \$], [S \to \cdot aCd, \$], [S \to \cdot bBd, \$] \\ I_1 & : & [S' \to S\cdot, \$] \\ I_2 & : & [S \to a \cdot Bc, \$], [S \to a \cdot Cd, \$], [B \to \cdot e, c], [C \to \cdot e, d] \\ I_3 & : & [S \to b \cdot Cc, \$], [S \to b \cdot Bd, \$], [B \to \cdot e, d], [C \to \cdot e, c] \\ I_4 & : & [S \to aB \cdot c, \$] \\ I_5 & : & [S \to aC \cdot d, \$] \\ I_6 & : & [B \to e\cdot, c], [C \to e\cdot, d] \\ I_7 & : & [S \to bC \cdot c, \$] \\ I_8 & : & [S \to bB \cdot d, \$] \\ I_9 & : & [C \to e\cdot, c], [B \to e\cdot, d] \\ I_{10} & : & [S \to aBc\cdot, \$] \\ I_{11} & : & [S \to aCd\cdot, \$] \\ I_{12} & : & [S \to bCc\cdot, \$] \\ I_{13} & : & [S \to bBd\cdot, \$] \end{array}$$

We can merge I_6 and I_9 as $I_{69} : [C \to e\cdot, c/d], [B \to e\cdot, d/c]$, which calls for reduction by two different productions when next token is c or d. So, we have to be careful in doing mergings because reduce-reduce conflicts can be introduced. When such a conflict arises while doing a merging, we say that the grammar is not LALR(1).

LALR Table Construction

There are two ways to construct LALR(1) parsing table. The first and most obvious way is to construct LR(1) table and merge the sets. Consider the grammar,

$$\begin{array}{rcl} S' & \to & S \\ S & \to & CC \\ C & \to & cC \mid d, \end{array}$$

for which we have already shown the LR(1) items earlier. Also, the LR(1) table for it is shown in Table 3.5. From this, we can construct the LALR(1) table by merging various sets as shown in Table 3.6.

Table 3.5: An LR(1) table

	ACTION			GOTO	
State	c	d	\$	S	C
0	s3	s4		1	2
1			accept		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Table 3.6: LALR table

	ACTION			GOTO	
State	c	d	\$	S	C
0	s36	s47		1	2
1			accept		
2	s36	s47			5
36	s36	s47			89
47	r3	r3			
5			r1		
89	r2	r2	r2		

This hardly saves any time but it may save memory. There is another method for LALR(1) that saves both. It is called *Step-by-Step Merging*:

Merge the configurating sets at each step in the derivation of the collection of configurating sets. Sets of states are constructed as in LR(1) method, but at each point where a new set is spawned, it may be merged with an existing set. When a new set S is created, all other states are checked to see if one with the same core exists. If not, S is kept; otherwise it is merged with the existing set T with the same core to form state ST. The merged set is assigned a new set number and the original set T is deleted.

Here is an example of this method in action:

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow V = E \\ E &\rightarrow F|E + F \end{aligned}$$

$$\begin{aligned} F &\rightarrow V|\mathbf{integer}|(E) \\ V &\rightarrow \mathbf{id} \end{aligned}$$

Begin building the LR(1) collection of configurating sets:

$$\begin{aligned}
I_0 &: [S' \rightarrow \cdot S, \$], [S \rightarrow \cdot V = E, \$], [V \rightarrow \cdot \mathbf{id}, =] \\
I_1 &: [S' \rightarrow S\cdot, \$] \\
I_2 &: [S \rightarrow V\cdot = E, \$] \\
I_3 &: [V \rightarrow \mathbf{id}\cdot, =] \\
I_4 &: [S \rightarrow V = \cdot E, \$], [E \rightarrow \cdot F, \$+], [E \rightarrow \cdot E + F, \$+], [F \rightarrow \cdot V, \$+], \\
&\quad [F \rightarrow \cdot \mathbf{integer}, \$+], [F \rightarrow \cdot(E), \$+], [V \rightarrow \cdot \mathbf{id}, \$+] \\
I_5 &: [S \rightarrow V = E\cdot, \$], [E \rightarrow E \cdot + F, \$+] \\
I_6 &: [E \rightarrow F\cdot, \$+] \\
I_7 &: [F \rightarrow V\cdot, \$+] \\
I_8 &: [F \rightarrow \mathbf{integer}\cdot, \$+] \\
I_9 &: [F \rightarrow (\cdot E), \$+], [E \rightarrow \cdot F,)+], [E \rightarrow \cdot E + F,)+], [F \rightarrow \cdot V,)+], [F \rightarrow \cdot \mathbf{integer},)+], \\
&\quad [F \rightarrow \cdot(E),)+], [V \rightarrow \cdot \mathbf{id},)+] \\
I_{10} &: [V \rightarrow \mathbf{id}\cdot, \$+] \text{ same as } I_3 \text{ so merge to get} \\
I_{10}/I_3 &: [V \rightarrow \mathbf{id}\cdot, = \$+] \\
I_{11} \ldots I_{12} \ldots &: \text{ gives new sets} \\
I_{13} &: [E \rightarrow F\cdot,)+] \text{ same as } I_6 \text{ so merge to get} \\
I_{13}/I_6 &: [E \rightarrow F\cdot, \$)+]
\end{aligned}$$

and so on. When we finish creating the sets, we then construct the table as in LR(1).

3.6.4 Handling Ambiguity in LR Parsers

The existence of ambiguity in the grammar creates LR parsing tables with conflicting entries. However, the ambiguous grammars are often smaller in size, since they generally have lesser number of nonterminals. Thus, the size of the parsing table is reduced, and hence, the parsing routine runs faster, thus resulting faster compilation. The conflicts in the table are often resolved by adopting certain rules – known as *disambiguating rules*, which perhaps cannot be specified by the grammar without introducing further nonterminals. For example, consider the following augmented grammar for expressions

$$\begin{aligned} E' &\rightarrow E \\ E &\rightarrow E + E \mid E - E \mid E * E \mid -E \mid (E) \mid \mathbf{id} \end{aligned}$$

The LR(0) sets of items created are as follows:

$$\begin{aligned}
I_0 &: \{[E' \rightarrow \cdot E], [E \rightarrow \cdot E + E], [E \rightarrow \cdot E - E], [E \rightarrow \cdot E * E], [E \rightarrow \cdot E/E], [E \rightarrow \cdot - E], \\
&\quad [E \rightarrow \cdot(E)], [E \rightarrow \cdot \mathbf{id}]\}, \\
I_1 &: \{[E' \rightarrow E\cdot], [E \rightarrow E \cdot + E], [E \rightarrow E \cdot - E], [E \rightarrow E \cdot * E], [E \rightarrow E \cdot /E]\},
\end{aligned}$$

I_2 : $\{[E \rightarrow - \cdot E], [E \rightarrow \cdot E + E], [E \rightarrow \cdot E - E], [E \rightarrow \cdot E * E], [E \rightarrow \cdot E/E], [E \rightarrow \cdot - E],$
$[E \rightarrow \cdot(E)], [E \rightarrow \cdot \mathbf{id}]\},$

I_3 : $\{[E \rightarrow (\cdot E)], [E \rightarrow \cdot E + E], [E \rightarrow \cdot E - E], [E \rightarrow \cdot E * E], [E \rightarrow \cdot E/E], [E \rightarrow \cdot - E],$
$[E \rightarrow \cdot(E)], [E \rightarrow \cdot \mathbf{id}]\},$

I_4 : $\{[E \rightarrow \mathbf{id}\cdot]\},$

I_5 : $\{[E \rightarrow E + \cdot E], [E \rightarrow \cdot E + E], [E \rightarrow \cdot E - E], [E \rightarrow \cdot E * E], [E \rightarrow \cdot E/E], [E \rightarrow \cdot - E],$
$[E \rightarrow \cdot(E)], [E \rightarrow \cdot \mathbf{id}]\},$

I_6 : $\{[E \rightarrow E - \cdot E], [E \rightarrow \cdot E + E], [E \rightarrow \cdot E - E], [E \rightarrow \cdot E * E], [E \rightarrow \cdot E/E], [E \rightarrow \cdot - E],$
$[E \rightarrow \cdot(E)], [E \rightarrow \cdot \mathbf{id}]\},$

I_7 : $\{[E \rightarrow E * \cdot E], [E \rightarrow \cdot E + E], [E \rightarrow \cdot E - E], [E \rightarrow \cdot E * E], [E \rightarrow \cdot E/E], [E \rightarrow \cdot - E],$
$[E \rightarrow \cdot(E)], [E \rightarrow \cdot \mathbf{id}]\},$

I_8 : $\{[E \rightarrow E/ \cdot E], [E \rightarrow \cdot E + E], [E \rightarrow \cdot E - E], [E \rightarrow \cdot E * E], [E \rightarrow \cdot E/E], [E \rightarrow \cdot - E],$
$[E \rightarrow \cdot(E)], [E \rightarrow \cdot \mathbf{id}]\},$

I_9 : $\{[E \rightarrow -E\cdot], [E \rightarrow E \cdot +E], [E \rightarrow E \cdot -E], [E \rightarrow E \cdot *E], [E \rightarrow E \cdot /E]\},$

I_{10} : $\{[E \rightarrow (E\cdot)], [E \rightarrow E \cdot +E], [E \rightarrow E \cdot -E], [E \rightarrow E \cdot *E], [E \rightarrow E \cdot /E]\},$

I_{11} : $\{[E \rightarrow E + E\cdot], [E \rightarrow E \cdot +E], [E \rightarrow E \cdot -E], [E \rightarrow E \cdot *E], [E \rightarrow E \cdot /E]\},$

I_{12} : $\{[E \rightarrow E - E\cdot], [E \rightarrow E \cdot +E], [E \rightarrow E \cdot -E], [E \rightarrow E \cdot *E], [E \rightarrow E \cdot /E]\},$

I_{13} : $\{[E \rightarrow E * E\cdot], [E \rightarrow E \cdot +E], [E \rightarrow E \cdot -E], [E \rightarrow E \cdot *E], [E \rightarrow E \cdot /E]\},$

I_{14} : $\{[E \rightarrow E/E\cdot], [E \rightarrow E \cdot +E], [E \rightarrow E \cdot -E], [E \rightarrow E \cdot *E], [E \rightarrow E \cdot /E]\},$

I_{15} : $\{[E \rightarrow (E)\cdot]\}$

The Follow of E is $\{+, -, *, /,), \$\}$. The corresponding SLR parsing table has been shown in Table 3.7. There are several entries with *shift-reduce* conflicts. For example, from state I_9 on input '+', it suggests either to reduce by $E \rightarrow -E$ or shift. Since '$-$' here is the unary one, the correct action is to *reduce*. Thus, all conflicts of state I_9 are to be resolved as *reduce*. From state I_{11} on '+', it says either to shift or reduce by $E \rightarrow E + E$. The correct action is *reduce* since '+' is *left associative*. Similarly, for '$-$' also, the entry is to *reduce*. However, for '$*$' and '/', since their precedences are more than '+', the correct action is to shift. In this way, all other conflicts can be resolved by noting the associativity and precedence in the following descending order. Here, all operators have been assumed as left associative.

unary $-$,
$*$, /,
$+$, $-$

The parsing table after removing the ambiguities is shown in Table 3.8.

3.6.5 Error Recovery in LR Parser

As noted earlier, the undefined entries in the LR parsing table means *error*. The parser detects an error while consulting the table for a particular state and input symbol combination, for which the table entry has been left blank. However, each such error situation can be arrived

Table 3.7: SLR parsing table with duplicate entries

State	ACTION								GOTO
	+	−	*	/	(	)	**id**	$	*E*
0		s2			s3		s4		1
1	s5	s6	s7	s8				acc	
2		s2			s3		s4		9
3		s2			s3		s4		10
4	r8	r8	r8	r8		r8		r8	
5		s2			s3		s4		11
6		s2			s3		s4		12
7		s2			s3		s4		13
8		s2			s3		s4		14
9	r6/s5	r6/s6	r6/s7	r6/s8		r6		r6	
10	s5	s6	s7	s8		s15			
11	r2/s5	r2/s6	r2/s7	r2/s8		r2		r2	
12	r3/s5	r3/s6	r3/s7	r3/s8		r3		r3	
13	r4/s5	r4/s6	r4/s7	r4/s8		r4		r4	
14	r5/s5	r5/s6	r5/s7	r5/s8		r5		r5	
15	r7	r7	r7	r7		r7		r7	

Table 3.8: SLR parsing table after resolving duplicate entries

State	ACTION								GOTO
	+	−	*	/	(	)	**id**	$	*E*
0		s2			s3		s4		1
1	s5	s6	s7	s8				acc	
2		s2			s3		s4		9
3		s2			s3		s4		10
4	r8	r8	r8	r8		r8		r8	
5		s2			s3		s4		11
6		s2			s3		s4		12
7		s2			s3		s4		13
8		s2			s3		s4		14
9	r6	r6	r6	r6		r6		r6	
10	s5	s6	s7	s8		s15			
11	r2	r2	s7	s8		r2		r2	
12	r3	r3	s7	s8		r3		r3	
13	r4	r4	r4	r4		r4		r4	
14	r5	r5	r5	r5		r5		r5	
15	r7	r7	r7	r7		r7		r7	

at from some particular sequences of moves by the parser, thus proper error messages can be flashed to the user. Moreover, the error handling routines can be made to modify the parser stack by popping out some entries, or pushing some desirable entries into the stack to make the parser arrive at a descent state from which it can proceed further. This is known as *error recovery*. Error recovery is necessary to enable the parser to detect multiple syntax errors and flashing them to the user for correction. We shall illustrate the idea with the help of a small expression grammar. Consider the grammar

$$\begin{aligned} E' &\rightarrow E \\ E &\rightarrow E + E \mid E * E \mid \mathbf{id} \end{aligned}$$

The LR(0) sets of items are as follows:

$$\begin{aligned} I_0 &: \{[E' \rightarrow \cdot E], [E \rightarrow \cdot E + E], [E \rightarrow \cdot E * E], [E \rightarrow \cdot \mathbf{id}]\}, \\ I_1 &: \{[E' \rightarrow E \cdot], [E \rightarrow E \cdot + E], [E \rightarrow E \cdot * E]\}, \\ I_2 &: \{[E \rightarrow \mathbf{id} \cdot]\}, \\ I_3 &: \{[E \rightarrow E + \cdot E], [E \rightarrow \cdot E + E], [E \rightarrow \cdot E * E], [E \rightarrow \cdot \mathbf{id}]\}, \\ I_4 &: \{[E \rightarrow E * \cdot E], [E \rightarrow \cdot E + E], [E \rightarrow \cdot E * E], [E \rightarrow \cdot \mathbf{id}]\}, \\ I_5 &: \{[E \rightarrow E + E \cdot], [E \rightarrow E \cdot + E], [E \rightarrow E \cdot * E]\}, \\ I_6 &: \{[E \rightarrow E * E \cdot], [E \rightarrow E \cdot + E], [E \rightarrow E \cdot * E]\} \end{aligned}$$

The Follow set of E is $\{+, *, \$\}$. The SLR parsing table for the grammar is shown in Table 3.9. The error entries have been marked as $e1$ and $e2$, corresponding to the two error routines to be called in various situations. The routine $e1$ has been called from states 0,3, and 4 on seeing an operator or end of string while an **id** was expected. Thus, the routine may push an **id** and state 2 onto the stack and proceed. On the other hand, routine $e2$ has been called from states 1, 2, 5, and 6 upon getting an **id** while expecting an operator. The error handler will push a '+' and state 3 onto the stack and proceed.

Table 3.9: SLR parsing table with error entries

State	ACTION				GOTO
	id	+	*	$	E
0	s2	e1	e1	e1	1
1	e2	s3	s4	acc	
2	e2	r3	r3	r3	
3	s2	e1	e1	e1	5
4	s2	e1	e1	e1	6
5	e2	r1	s4	r1	
6	e2	r2	r2	r2	

Consider the input string "**id** + *$". The parser proceeds as shown in Table 3.10. Thus, the parsing proceeds to the end recovering from syntax errors, flashing proper error messages and modifying the stack to recover from the error.

Table 3.10: Error recovery

Stack	Input	Error message and action
0	**id** + *$	shift
0**id**2	+ * $	reduce by $E \rightarrow$ **id**
0E1	+ * $	shift
0E1 + 3	*$	"**id** expected", pushed **id** and 2 to stack
0E1 + 3**id**2	*$	reduce by $E \rightarrow$ **id**
0E1 + 3E5	*$	shift
0E1 + 3E5 * 4	$	"**id** expected", pushed **id** and 2 to stack
0E1 + 3E5 * 4**id**2	$	reduce by $E \rightarrow$ **id**
0E1 + 3E5 * 4E6	$	reduce by $E \rightarrow E * E$
0E1 + 3E5	$	reduce by $E \rightarrow E + E$
0E1	$	accept

3.7 LALR PARSER GENERATOR–yacc

The tool *yacc* can be used to generate automatically an LALR parser for a grammar from its specification. The input of yacc is divided into three sections:

```
... definitions ...
%%
... rules ...
%%
... subroutines ...
```

The *definitions* section consists of token declarations and C code bracketed by %{ and %}. The grammar is placed in the *rules* section, and user subroutines are added in *subroutines* section.

This is best illustrated by constructing a small calculator that can add and subtract numbers. We will begin by examining the linkage between *lex* and *yacc*. Here is the definitions section for the yacc input file:

```
%token INTEGER
```

This definition declares an INTEGER token. When we run yacc, it generates a parser in the file *y.tab.c* and also creates an include file *y.tab.h*:

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define INTEGER 258
extern YYSTYPE yylval;
```

Lex includes this file and utilizes the definitions for token values. To obtain tokens, yacc calls *yylex*. Function *yylex* has a return type of *int* and returns the token value. Attributes associated with the token are returned by lex in variable *yylval*. For example,

```
[0-9]+          {
                        yylval = atoi(yytext);
                        return INTEGER;
                }
```

would store the value of the integer in *yylval* and return token INTEGER to yacc. The type of *yylval* is determined by YYSTYPE. Since the default type is integer, this works well in this case. Token values 0-255 are reserved for character values. Generated token values typically start around 258, as lex reserves several values for end-of-file and error processing. The complete lex input for the calculator is as follows:

```
%{
#include < stdlib.h >
void yyerror(char *);
#include "y.tab.h"
%}
%%
        [0-9]+
            yylval = atoi(yytext);
            return INTEGER;
        }
[− + \n]return *yytext;
[ \t]   ; /* skip whitespace */
.       yyerror("Invalid character");
%%
int yywrap(void) {
    return 1;
}
```

Internally, yacc maintains two stacks in memory—a parse stack and a value stack. The parse stack contains terminals and nonterminals and represents the current parsing state. The value stack is an array of YYSTYPE elements, and associates a value with each element in the parse stack. For example, when lex returns an INTEGER token, yacc shifts this token to the parse stack. At the same time, the corresponding yylval is shifted to the value stack. Since the parse and value stacks are always synchronized, finding a value related to a token on the stack is easily accomplished. The yacc input for the calculator is as follows:

```
%{
    int yylex(void);
    void yyerror(char *);
%}
%token INTEGER
```

```
%%
program:
        program expr '\n'  {printf("%d\n", $2); }
        |
        ;
expr:
        INTEGER         {$$ = $1;}
        | expr '+' expr {$$ = $1 + $3;}
        | expr '-' expr {$$ = $1 -- $3;}
        ;
%%
void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
    return 0;
}
int main(void) {
    yyparse();
    return 0;
}
```

Actions associated with a rule are entered within braces. By utilizing left-recursion, we have specified that a program consists of zero or more expressions. Each expression terminates with a newline. When a newline is detected, we print the value of the expression. When we apply the rule

expr: expr '+' expr {$$ = $1 + $3;},

we replace the right-hand side of the production in the parse stack with the left-hand side of the same production. In this case, we pop "expr '+' expr" and push "expr". We may reference to the positions in the value stack in our C code by specifying "$1" for the first term on the right-hand side, "$2" for the second term and so on. "$$" designates the top of the stack after reduction has taken place.

In the event of syntax errors, yacc calls the user-defined function yyerror. yacc can also determine the shift/reduce and reduce/reduce conflicts in a specified grammar. The shift/reduce conflice is resolved in favour of "shift", while a reduce/reduce conflict follows the first reduction rule, by default.

3.8 Syntax Directed Translation

At the end of parsing stage, we would know if a sentence is grammatically correct. However, in the process, many other useful things can be done towards code generation. This is done by defining a set of semantic actions for various grammar rules. The process is known as *syntax directed translation.* A set of attributes is associated with each grammar symbol—some of them are *synthesized* and some are *inherited.* The actions written corresponding to a production rule manipulate these attributes effectively to do the desired translation. The

parse tree with attributes is called an *annotated parse tree*, and the act of adding the attributes is called *annotating* the parse tree. The following example illustrates the process.

Example 3.12 Consider the problem of generating postfix equivalents of infix expressions. An infix expression grammar can be written as,

$$\begin{aligned} E &\rightarrow E+T \mid E-T \mid T \\ T &\rightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9 \end{aligned}$$

Let us assume an attribute *val* with E and T that holds the string corresponding to the postfix expression. The semantic actions can be written as,

$$\begin{aligned} E \rightarrow E_1 + T \qquad & \{E.val = E_1.val \mid\mid T.val \mid\mid '+'\} \\ E \rightarrow E_1 - T \qquad & \{E.val = E_1.val \mid\mid T.val \mid\mid '-'\} \\ E \rightarrow T \qquad & \{E.val = T.val\} \\ T \rightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9 \qquad & \{T.val = number\} \end{aligned}$$

The operator '||' is used to represent string concatenation. Consider the expression "3+2−4". The annotated parse tree for the string is given in Fig. 3.15. It generates the postfix string "3 2+4−".

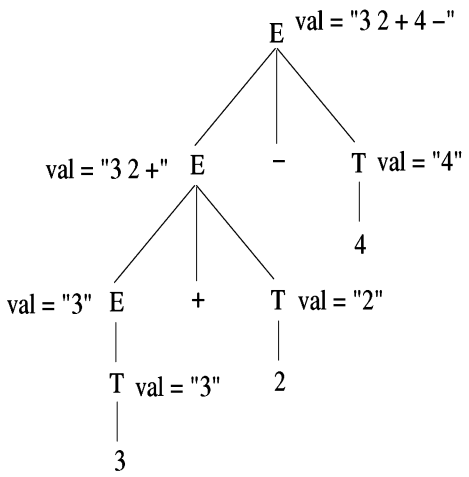

Figure 3.15: Annotated parse tree.

3.9 CONCLUSION

In this Chapter, we have dealt with the design of various types of parsers to be used for syntax analysis of strings of a language. We have also looked into the strategies for error detection and recovery, syntax directed definitions to accomplish various desired operations on the input strings and the automated parser generators. The parse tree produced implicitly or explicitly by the stage will be used in later stages for code generation and other related activities.

EXERCISES

3.1 Write context free grammars to detect the following. Is your grammar ambiguous?

(a) occurrence of balanced parentheses.

(b) strings over the alphabet set $\{a, b\}$, such that every a is immediately followed by a b.

(c) strings over the alphabet set $\{a, b\}$, such that every pair of a (that is aa) is immediately followed by a pair of b's (that is bb).

(d) strings over the alphabet set $\{a, b\}$, having equal number of a's and b's.

(e) strings over the alphabet set $\{a, b\}$, having unequal number of a's and b's.

3.2 For each of the grammars constituted in 3.1, draw parse trees for representative strings of the language. For example, "(()()())" is a representative for (a). Check carefully if it accepts any string not in the language.

3.3 Write down the grammar for Boolean expressions with operators *or and not* and the constants *true, false*, identifier *id* and parentheses (,). Design

(a) a recursive predictive parser.
(b) a nonrecursive predictive parser.

Show the parsing of the string "*id or id and not id*" using the nonrecursive parser.

3.4 Construct the operator precedence parser for the following grammar.

$$\begin{aligned} S &\rightarrow (L) \mid a \\ L &\rightarrow L, S \mid S \end{aligned}$$

Show the parsing of the string "$(a, ((a, a), (a, a)))$" using the parser constructed.

3.5 Consider the grammar

$$E \rightarrow E \ . \ i \ (\ E \ ) \mid (\ E \ ) \mid E \ ? \ E \ : \mid i,$$

where $\{E\}$ is the set of nonterminal symbols, E is the start symbol and

$$\{ . \ ? \ : \ (\ ) \ \} \cup \{i \mid i \text{ is an identifier}\}$$

is the set of terminal symbols. Give a corresponding LL(1) grammar which generates the same language as the one above. Show the First and Follow sets for each nonterminal symbol and the predictive parsing table. Argue that the grammar is LL(1).

3.6 Consider the grammar

$$\begin{aligned} A &\rightarrow B \ C \ x \mid y \\ B &\rightarrow y \ A \mid \epsilon \\ C &\rightarrow A \ y \mid x, \end{aligned}$$

where $\{A, B, C\}$ is the set of nonterminal symbols, A is the start symbol, $\{x, y\}$ is the set of terminal symbols. The grammar is not LL(1). Why? As part of your answer, show the First and Follow sets for each nonterminal symbol.

3.7 Consider the grammar

$$\begin{aligned} E &\rightarrow B\ A \\ A &\rightarrow \&B\ A \mid \epsilon \\ B &\rightarrow true \mid false \end{aligned}$$

where $\{E, A, B\}$ is the set of nonterminal symbols, E is the start symbol and $\{\&, true, false\}$ is the set of terminal symbols. Show that the grammar is LL(1). In the process, construct the predictive parsing table.

3.8 Consider the following grammar for regular expressions.

$$R \rightarrow R\ '|'\ R \mid RR \mid R^* \mid (R) \mid a \mid b$$

Modify the grammar to make it LL(1). Generate the predictive parsing table for the resulting grammar.

3.9 Construct the SLR parsing table for the grammar of 3.1.

3.10 Construct the SLR parsing table for the grammar of 3.3.

3.11 Construct the SLR parsing table for the grammar of 3.4.

3.12 Construct the SLR parsing table for the grammar of 3.5.

3.13 Construct the SLR parsing table for the grammar of 3.6.

3.14 Construct the SLR parsing table for the grammar of 3.7.

3.15 Consider the grammar

$$\begin{aligned} S &\rightarrow A\ S \mid b \\ A &\rightarrow S\ A \mid a \end{aligned}$$

(a) Show that the grammar is ambiguous.

(b) Construct the corresponding SLR parsing table.

(c) Construct the corresponding LR(1) parsing table.

(d) Construct the corresponding LALR parsing table.

3.16 Consider the grammar

$$\begin{aligned} E &\rightarrow E\ +\ T \mid T \\ T &\rightarrow T\ F \mid F \\ F &\rightarrow F^* \mid a \mid b \end{aligned}$$

(a) Construct the SLR parsing table.

(b) Construct the corresponding LALR parsing table.

3.17 Consider the grammar

$$\begin{aligned} S &\rightarrow AaAb \mid BbBa \\ A &\rightarrow \epsilon \\ B &\rightarrow \epsilon \end{aligned}$$

Show that the grammar is LL(1) but not SLR.

3.18 Consider the grammar

$$\begin{aligned} S &\rightarrow Aa \mid bAc \mid dc \mid bda \\ A &\rightarrow d \end{aligned}$$

Show that the grammar is LALR(1) but not SLR.

3.19 Show that the following grammar is LR(1) but not LALR(1).

$$\begin{aligned} S &\rightarrow Aa \mid bAc \mid Bc \mid bBa \\ A &\rightarrow d \\ B &\rightarrow d \end{aligned}$$

3.20 Extend the parsing table for Boolean expression grammar to show error handling.

3.21 Discuss the role of parser in compiler design.

3.22 Arrange as per processing power: SLR, LR(1), LALR. Also, arrange them as per the number of states.

3.23 Write YACC specification files for each of the grammars to identify (a) Boolean expressions and (b) regular expressions.

3.24 Write syntax directed translation scheme for the regular expression grammar to generate the NFA corresponding to a regular expression.

3.25 Write syntax directed translation scheme to evaluate arithmetic expressions.

Chapter 4

Type Checking

Type checking is one of the most important semantic aspects of compilation. Essentially, type checking

1. allows the programmer to limit what types may be used in certain circumstances,
2. assigns types to values, and
3. determines whether these values are used in an appropriate manner.

Apart from verifying the code to be *correct*, like checking if function calls have the correct number and types of parameters, type checking also helps in deciding which code to be generated as in case of arithmetic expressions.

In this chapter, we will discuss how type checking can be carried out in a programming language to verify their usage in programs. There can be many variants of type checking. In the simplest situation, it may check the types of objects and report a *type-error* in case of a violation. On the other hand, an incorrect type may be *corrected* (by *coercing*, which is defined later) to a valid one. The definition of type is very subtle, and as we will see in this chapter, checking type equivalence of two objects is an important problem to be solved.

4.1 STATIC VS. DYNAMIC CHECKING

In many of the modern languages, type checking is done at *compile time.* This is also known as *static checking*, because in this type of checking, properties can be verified before the program is run. On the other hand, *dynamic checking* or *runtime checking* is performed during execution of the program.

The advantages of compile time checking are as follows:

1. It can catch many common errors.
2. Static checking is desired when speed is important, since it can result faster code that does not perform any type checking during execution.

Dynamic checking offers the following advantages:

1. It usually permits the programmer to be less concerned with types. Thus, if frees the programmer.

2. It may be required in some cases like *array bounds check*, which can be performed only during execution.
3. It may give rise to in more robust code by ensuring thorough checking of values for the program identifiers during execution.
4. It can give in clearer code.

4.2 TYPE EXPRESSIONS

Type expressions are used to represent the types of language constructs. A type expression can either be a *basic type* a *type name*, or a *type constructor* applied to a list of type expressions.

In this section, we will deal with different kinds of type expressions.

1. The *basic types* include *integer, real, char, boolean* and other *atomic* types that do not have internal structure. Every programming language will have a set of such basic types. The identifiers used in the language can be of the basic types or may be of types derived from them. A special type called *type-error* will be used to indicate that there is some type violation.
2. *Arrays* are specified as $array(I, T)$, where T is a type expression while I is usually an *integer* or a *range of integers.* The *array* (I, T) denotes an array of type T with I elements, provided I is an integer. In case I is a range, it gives the possible index values to access the array. For example, the C declaration

   int a[100]

   identifies the type of a to be *array (100, integer).*
3. If T_1 and T_2 are two type expressions, then their cartesian *product* $T_1 \times T_2$ is also a type expression. Products are generally used to denote *anonymous records* (in which, field names are absent), and function argument list. For example, an argument list passed to a function *fun* with first argument as integer and second argument as real, has the associated type *integer* $\times$ *real.*
4. *Named records* are products, but with named elements. For example, consider a record structure with two named fields—*length* which is an *integer* and *word* which is type *array(10, char).* The record is of type

   $$record((length \times integer) \times (word \times array(10, character)))$$

5. If T is a type expression, then *pointer(T)* is also a type expression representing objects which are pointers to objects of type T. For example, *pointer(integer)* represents a type which is a pointer to an integer.
6. *Functions* map a collection of types to another. It is represented by the type expression $D \rightarrow R$, where D is the *domain* and R is the *range* of the function. Both D and R should be type expressions. For example, the type expression

   $$integer \times integer \rightarrow character$$

represents a function that takes two integers as arguments and returns a character value. Let us consider the following type expression:

$$integer \rightarrow (real \rightarrow character)$$

It represents a function that takes an integer as an argument and returns another function which maps a real number to a character. Many languages restrict the return value of functions not to include *array* or *function* as the return type. However, a function as a return value is quite common for functional programming languages.

Type Systems *Type system* of a language is a collection of rules depicting the type expression assignments to program objects. This is usually done using a syntax directed definition. An implementation of a type system is called a *type checker.*

Definition 4.1 Strongly typed language: *A strongly typed language is one in which the compiler can verify that the program will execute without any type errors. All the checks are made static in this case. It is also called a sound type system. It completely eliminates the necessity of dynamic type checking.*

Most of the programming languages are *weakly typed*, because strongly typed languages put lot of restrictions on the programmer. Also, there are cases in which a type error can be caught dynamically only. Many languages also allow the user to override the system. For example, in the language C, one can cast types or use void pointers and basically, can send incorrect types to a function.

4.3 TYPE CHECKING

Type checking can most conveniently be carried out using a *syntax directed definition.* This computes the type of the derived object from the types of its syntactical components. We will now look into the computation of derived type for *expressions, statements* and *function calls.*

Type Checking of Expressions. We introduce a synthesized attribute *type* for the non-terminal E representing an expression. Table 4.1 below represents a few representative rules for computing the types of expressions. It may be noted that it is only a representative set.

Table 4.1: Type checking expressions

Expression	Action
$E \rightarrow$ **id**	$E.type \leftarrow lookup($**id**$.entry)$
$E \rightarrow E_1\, op\, E_2$	$E.type \leftarrow$ if $E_1.type = E_2.type$ then $E_1.type$ else *type-error*
$E \rightarrow E_1\, relop\, E_2$	$E.type \leftarrow$ if $E_1.type = E_2.type$ then *boolean* t else *type-error*
$E \rightarrow E_1[E_2]$	$E.type \leftarrow$ if $E_2.type =$ *integer* and $E_1.type = array(s,t)$ then t else *type-error*
$E \rightarrow E_1 \uparrow$	$E.type \leftarrow$ if $E_1.type = pointer(t)$ then t else *type-error*

The actual set of rules may be much more complex for a programming language. For example, the language may support mixed mode expressions allowing real and integer in the same expression. Certain set of operations may be supported for only a few particular combinations of types.

Type Checking of Statements. The *statements* normally do not have any value. Thus, they are of type *void.* However, to propagate the type error occurring in some statement nested deep inside a block, we need a set of rules. Table 4.2 is such a collection of representative rules. It passes on the type errors to higher level constructs. Of course, there exists the scope of pointing more accurately the occurrence and nature of the type error. We assume the attribute *type* with the non-terminal S for statements to hold the type of the statements.

Table 4.2: Type checking statements

$S \rightarrow \textbf{id} = E$	$S.type \leftarrow$ if **id**.*type* $= E.type$ then *void* else *type-error*
$S \rightarrow \textbf{if } E \textbf{ then } S_1$	$S.type \leftarrow$ if $E.type =$ *boolean* then $S_1.type$ t else *type-error*
$S \rightarrow \textbf{while } E \text{ do } S_1$	$S.type \leftarrow$ if $E.type =$ *boolean* then $S_1.type$ else *type-error*
$S \rightarrow S_1; S_2$	$S.type \leftarrow$ if $S_1.type =$ *void* and $S_2.type =$ *void* then *void* else *type-error*

Type Checking of Functions A function call can be considered as the application of one expression to another. The type checking rule for the same is shown in Table 4.3. Here, we

Table 4.3: Type checking functions

$E \rightarrow E_1(E_2)$	$E.type \leftarrow$ if $E_2.type = s$ and $E_1.type = s \rightarrow t$ then t else *type-error*

first check whether E_1 has a type which is a function that has the *domain* same as the type of E_2. If a match is found, the type of the function call is set to the type of the return value of the function. That is, if the function is $s \rightarrow t$, the type of the function call is set to t.

4.4 TYPE EQUIVALENCE

The notion of *type equivalence* is very important during type checking. As we have seen earlier, during type checking we need to answer whether two type expressions s and t are the same or not. This can be answered by deciding the equivalence between the two types. There are two different categories of equivalence—*name equivalence* and *structural equivalence.*

1. **Name equivalence:** Two types are name equivalent if they have the same name or label. For example, consider the following few type and variable declarations.

   ```
   typedef int Value
   typedef int Total
   ...
   Value var1, var2
   Total var3, var4
   ```

   Here, the variables *var1* and *var2* are name equivalent, so are *var3* and *var4.* However, *var1* and *var4* are not name equivalent, because their type names are different.

2. **Structural equivalence:** It checks the structure of the type and determines equivalence based on whether or not the types have had the same constructor applied to

structurally equivalent types. It is checked recursively. For example, the types *array(I_1, T_1)* and *array(I_2, T_2)* are structurally equivalent if I_1 and I_2 are equal and T_1 and T_2 are structurally equivalent.

The structure of a type expression can be represented as a *Directed Acyclic Graph (DAG)* or a *tree.* Figure 4.1 below shows the representation of the type expression *record((length* $\times$ *integer)* $\times$ *(word* $\times$ *array(10, character)))* in the form of a tree.

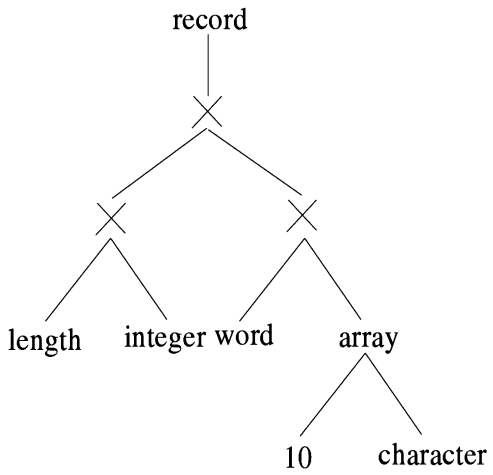

Figure 4.1: Tree for type expression.

Once the type expressions have been represented in the form of a tree or DAG, the structural equivalence can be checked through a simple recursive routine *dag-equivalence* described next.

function *dag-equivalence(s,t: type-DAGs): boolean*
begin
    if s and t represents the same basic type **then** return *true*
    if s represents *array(I_1, T_1)* and t represents *array(I_2, T_2)* **then**
        if $I_1 = I_2$ **then** return *dag-equivalence(T_1, T_2)*
        else return *false*
    if s represents $s_1 \times s_2$ and t represents $t_1 \times t_2$ **then**
        return *dag-equivalence(s_1, t_1) and dag-equivalence(s_2, t_2)*
    if s represents *pointer(s_1)* and t represents *pointer(t_1)* **then**
        return *dag-equivalence(s_1, t_1)*
    if $s = s_1 \rightarrow s_2$ and $t = t_1 \rightarrow t_2$ **then**
        return *dag-equivalence(s_1, t_1) and dag-equivalence(s_2, t_2)*
    return *false*
end.

Cycles in Type Representation

Many of the programming languages allow the types to be defined in a cyclical fashion. For example, consider the *C* declaration for *list* shown below.

```
struct list
{
    int val;
    struct list * next;
}
```

The possible representation of the type can be done either as shown in Fig. 4.2(a) or (b). The representation 4.2(a) shows an acyclic strategy, whereas Fig. 4.2(b) shows a cyclic one. Though there are strategies by which type equivalence can be checked with cyclic representation, most of the programming languages, including *C*, uses the acyclic one. The programming language *C* requires type names to be declared before using it, with the exception of *pointers* which can have undeclared record types. The name of the structure is also made part of the type, thus allowing testing of equivalence to stop when a structure is reached. At this point, the type expressions are equivalent if they point to the same structure name and are, otherwise inequivalent.

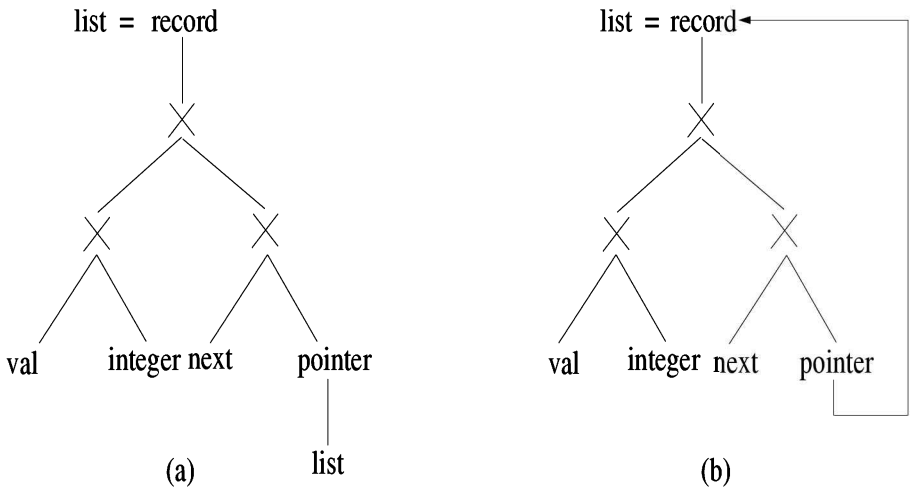

Figure 4.2: (a) Acyclic representation, (b) Cyclic representation.

4.5 TYPE CONVERSION

Type conversion refers to the local modification of type for a variable or subexpression. For example, it may be necessary to add an integer quantity to a real variable. However, the language may require both the operands of addition to be of the same type. Modifying the integer variable to real will require more space, since the reals are normally allocated more space than the integers. Thus, the solution is to treat the integer operand as a real operand locally and perform the operation, whereas, the variable, otherwise, remains to be of type integer. It may be done *explicitly* or *implicitly*. The implicit conversion is called *type-coercion*. In explicit conversion, the programmer writes code to instruct type conversion. For example, in language *C*, a programmer can write either of the following two codes.

```
int x;                          int x;
float y;                        float y;
...                             ...
y = ((float)x)/14.0;            y = x/14.0;
```

4.6 CONCLUSION

In this chapter, we have seen techniques for type checking of language constructs used in a program. Compiler usually performs *Static* type checking. *Dynamic* type checking, on the other hand, is often very costly. Types are normally represented as type expressions. Type checking can be performed using syntax directed techniques as illustrated in this chapter. Type equivalence of two type expressions can be verified by comparing the type graphs. The compiler can also perform type coercion to make an expression type correct.

EXERCISES

4.1 Explain the importance of type checking.

4.2 Compare and contrast static and dynamic type checkings. Give an example of the situation in which dynamic checking is really helpful.

4.3 What is a type expression? How does a weakly typed language may be helpful over a strongly typed one and vice versa?

4.4 Modify the type checking rules for expressions so that the type coercion features of *C* language are incorporated into them.

4.5 How can the statement types be helpful?

4.6 Discuss the importance of type equivalence checking.

4.7 Draw the type trees for the following pairs of type declarations and argue whether they are type equivalent or not.

(a)
```
struct list1 {                  struct list2 {
    char x;                         char abc;
    int y;                          int xy;
    int a[100];                     int ar[100];
}                               }
```

(b)
```
struct list1 {                      struct list2 {
    int val;                            int val;
    struct list1 *next;                 struct list2 *next;
}                                   }
```

Chapter 5

Symbol Tables

Symbol table is an essential data structure used by the compilers to remember information about identifiers appearing in the source language program. A symbol table contains nearly all the information needed by different phases of compilers. Usually, the phases—lexical analyzer and parser fill up the entries in the table, while the later phases like code generator and optimizer make use of the information available in the table. Thus, it acts as a common interface between the phases of a compiler. The types of symbols that are stored in the symbol table include variables, procedures, functions, defined constants, labels, structures, file identifications and computer generated temporaries. However, it should be kept in mind that the symbol table is designed by the compiler writer to facilitate the compilation process. Thus, the identifiers stored in the symbol tables may vary widely from implementation to implementation, even for the same language.

5.1 INFORMATION IN SYMBOL TABLE

For an identifier stored in a symbol table, the following are the associated information stored.

- **name**—the *name* of the identifier. The name may be stored directly in the table or the table entry may point to another character string, possibly stored in an associated *string table.* The second indirect approach is particularly suitable if the names can be arbitrarily long and widely varying in length.
- **type**—the *type* of the identifier. It defines, whether or not the identifier is a variable, a label, a procedure name, etc. For variables, it will further identify the type—the basic types like *integer, real, character*, or derived types like *arrays, records, pointers* and so on.
- **location**—This is generally an offset within the program where the identifier is defined.
- **scope**—The *scope* identifies the region of the program in which the current definition of the symbol is valid.
- **other attributes**—Some other information may be stored depending upon the type of the identifier—like *array limits, fields of records, parameters* and *return values for functions* etc.

5.1.1 Usage of Symbol Table Information

As discussed earlier, information about the identifiers are stored in the symbol table during the lexical and syntactic analysis phases of compilation. These information are used by the later phases of compiler in the following ways.

- **Semantic analysis:** To check the correct semantic usage of the language constructs. This often needs checking the types of the identifiers – the information normally available in the symbol table.
- **Code generation:** All program variables and temporaries need to be allocated some memory locations. A symbol table provides information regarding the size of the memory required for the identifiers through their types.
- **Error detection:** It is a very common error to leave variables undefined in the program. If a particular variable is undefined, every reference to it will result in an error message, informing that it is undefined. A better approach would be to make an entry into the symbol table marking it undefined at the very first reference to it, so that no more redundant error messages are generated in subsequent references to the variable.
- **Optimization:** To reduce the total number of variables used in a program, we need to reuse the temporaries generated by the compiler. We can merge two or more temporaries into one, only if their types are same (to keep the memory allocated same in both cases). Thus, the information stored in the symbol table regarding types of temporaries may be utilized by the optimizer to reuse the same memory.

5.2 FEATURES OF SYMBOL TABLES

The set of operations on a symbol table is as follows:

1. **Lookup:** This is the most frequent operation. As and when we come across an identifier in the program, we need to check whether it is defined or not. If it is defined, then the relevant information may be retrieved. On the other hand, if it is undefined, then need to create an entry into the symbol table.
2. **Insert:** Adding new names into the table. It occurs mostly in the lexical and syntax analysis phases.
3. **Modify:** Sometimes, when a name is defined, all the information about it is not available. Later on, when the information becomes available, appropriate entries of the table may be updated.
4. **Delete:** Though not very frequent, sometimes we need to delete entries from the symbol table. For example, when a procedure body ends, the variables defined inside may not be available any more to the later part of the program. Thus, all such variables may be deleted from the symbol table.

Keeping in view the above set of operations, the primary issues in symbol table design are as follows:

1. *Format of symbol table entries.* The symbol table may have various formats from linear array to list to tree structure and so on. We will deal with various structures in next section. Often the structures depict the procedures for accessing the tables.

2. *Access methodology.* It may be linear search, binary search, tree search, hashing and so on. The access method depends mainly on the structure of the table.
3. *Location of storage.* The symbol table may be located in the primary memory, thus making access fast. However, a large symbol table may necessitate to store it, at least partially in the secondary storage.
4. *Scope issues.* Scope rules of a language also put some constraint in the symbol table structure. For example, in a block-structured language, it may be necessary that a variable defined in upper blocks must also be visible to its nested subblocks. At the same time, if the same variable is defined both in the outer and the inner block, the inner definition should be taken. In other words, the innermost scoped variable must be found first.

Based on the scope rules, the symbol tables can be classified into two different classes:

- Simple symbol table with no nested scope and
- Scoped symbol table with nested scopes

5.3 SIMPLE SYMBOL TABLE

This organization works well for languages having a single scope. That is, all the identifiers are global. They also need fewer attributes per symbol. Since there is a single scope, the entries in the symbol table are never deleted. The fundamental operations on such a table are the following.

1. Enter a new symbol into the table,
2. Lookup for a symbol and
3. Modify information about a symbol stored earlier in the table.

There are several alternative data structures to choose from to create such a simple symbol table. However, the commonly used techniques are given below.

- linear table,
- ordered list,
- tree and
- hash table.

5.3.1 Linear Table

A linear table is a simple array of records with each record corresponding to an identifier of the program. The entries are made in the same order in which they appear in the program.

Example 5.1 Consider the following definitions:

int x, y
real z
...
procedure abc

...
$L1$: ...
...

The linear table will store the information as shown in Table 5.1.

Table 5.1: Symbol Table

name	type	location
x	integer	offset of x
y	integer	offset of y
z	real	offset of z
abc	procedure	offset of abc
$L1$	label	offset of $L1$

If the language puts no restriction on the length of the string representing the name of an identifier, then it may be convenient to store the names in a string table with the name field holding pointers to it. With this type of organization, the *lookup, insert* and *modify* operations are going to take $O(n)$ time, n being the number of identifiers stored. Of course, *insertion* can be made $O(1)$ by remembering the pointer to the next free index of the table where an entry can go. Also, if we can scan the most recent entries first, then it can probably speed up the access. This is because of the program locality—a variable defined just inside a block is expected to be referred to more often than some earlier variable.

5.3.2 Ordered List

This is a variation of linear tables, in which a list organization is used. The list may be sorted in some fashion, and then, binary search can be used to access the table in $O(\log n)$ time. However, an insertion needs to be done at an appropriate place to preserve the sorted form. Thus, the insertion becomes costly.

A modification to the ordered list is *self-organizing* list. Here, the neighbourhood of entries are changed dynamically. The position of a symbol in the table is dictated by the recency of reference.

Example 5.2 Consider the situation shown in Fig. 5.1(a), which shows that the most recently used symbol is *Identifier 4*, followed by *Identifier 2*, *Identifier 3* and so on. Now, if *Identifier 5* is accessed next, it comes to be the first entry in the list, *Identifier 4* is the second entry and so on. The situation has been shown in Fig. 5.1(b). Due to the program locality, it is expected that at any point of time during compilation, the entries near the beginning of the ordered list will be accessed more than those appearing later. This will definitely reduce the *lookup* time for symbols.

5.3.3 Tree

Tree organization of symbol table may be used to speed up the access. Each entry is represented by a node of the tree. Based on string comparison of names, the entries lesser than a reference

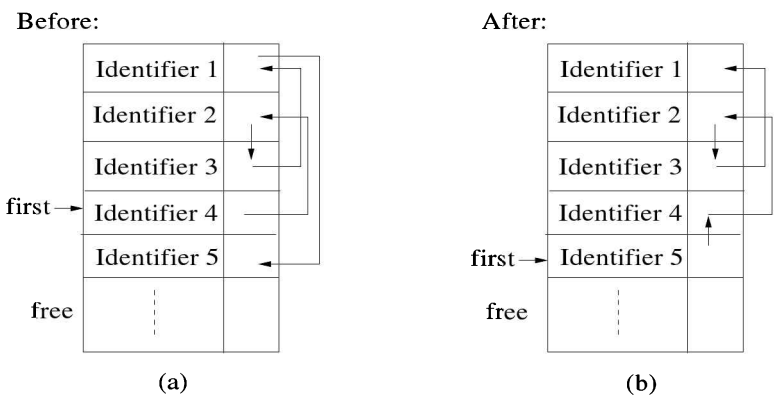

Figure 5.1: Ordered list symbol table

node are kept in its left-subtree, whereas, the entries greater than the reference node are put into the right subtree. The individual nodes also hold two more pointer fields to the left and right subtrees apart from having the regular information related to the symbol. Moreover, each lookup operation, apart from the equality check, needs to be followed by a *less-than/greater-than* check for each node of the tree where a match is not found, and the search must proceed further. A possible tree representing the entries of *Example* 5.1 has been shown in Fig. 5.2.

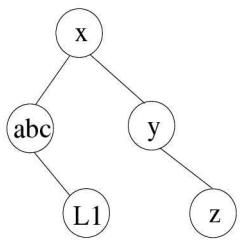

Figure 5.2: Tree symbol table.

The average lookup time for an entry in the tree is $O(\log n)$. However, it could degrade to a single long branching, thus requiring $O(n)$ search time. Proper *height balancing* strategies may be used for example *AVL trees*, to keep the tree balanced.

5.3.4 Hash Table

Hash table organization is desirable in those cases for which we want to minimize the access time. It is probably the most common method of implementing the symbol tables in compilers. Here, a set of names is mapped to a unique hash table position. The mapping is done using a *hash function.* The table, as such, is organized as an array. To store a symbol into the table, the hash function is applied on it which results into a unique location of the table. The symbol, along with its associated information is stored in this location. To access the symbol, again its name is hashed to a location of the table. Thus, the access time is $O(1)$. However, the problem arises due to *imperfectness* of hash functions that maps a number of symbols to the same location. To resolve the conflict, some *collision resolution* strategy needs to be followed

(For a detailed discussion on this refer to books on *Data Structures*). The commonly used strategies for collision resolution are *open addressing* and *chaining*. With both the strategies, the worst case access time is $\theta(n)$, n being the total number of records in the table. To reduce the number of collisions keeping the size of the table reasonable, the hash table is chosen to be of size between n and $2n$ for n keys.

Essentially, the hash function should have the following properties.

1. The hash function should depend on the name of the symbol. Equal emphasis should be given to each part of the name.
2. The function should be quickly computable, otherwise this will put severe limitation on the speed of compilation.
3. The function should be uniform in mapping names to different parts of the table. Similar names should not let to cluster to the same address. For example, the symbols like *data1, data2*, etc. should be mapped to places distributed over the table.
4. The computed value must always be within the range of table index.

5.4 SCOPED SYMBOL TABLE

The scope of a symbol table defines the region of the program in which a particular definition of the symbol is valid. This definition is said to be *visible* within the scope region. Most of the modern programming languages, particularly, the block structured languages, permit different types of scopes for the identifiers. The corresponding set of rules is called the *scope rules* for the language. In general an identifier may have different scopes depending upon the language under consideration these scopes have been dealt with below:

1. **Global scope:** An identifier with a global scope has visibility throughout the program. The *global variable* declarations of a program, generally, come with this scope.
2. **File-wide scope:** For a program with modules distributed in more than one file, an identifier defined to have file-wide scope is visible only within the file. In other words, it is global within the file.
3. **Local scope within a procedure:** For programs having multiple procedures, an identifier defined within a procedure may be visible only to the points inside the procedure. This is commonly referred to as *local variables* of functions or procedures.
4. **Local scope within a block:** A program block is a code chunk having single entry and single exit. Thus, a procedure may be further divided into a number of blocks and the identifier defined inside a block will be visible only within that block.

The scoping rules can be broadly classified into two categories depending upon the time at which the scope gets defined.

- **Static or Lexical Scoping:** This depends purely on the organization of the program. Here the scope is defined by syntactic nesting. The lexical scoping can be used efficiently by the compilers to generate correct references.
- **Dynamic or Runtime Scoping:** Here, scoping depends on the execution sequence of the program. For example, let us suppose that a procedure P_1 callable from two different procedures P_2 and P_3 has reference to a non-local variable x. Let us also assume that

there exists two different definitions of x, each in P_2 and P_3. Under dynamic scope rule, when P_1 is called from P_2, x will refer to the definition in P_2, whereas, in the other case, it will refer to x occurring in P_3. A compiler supporting this type of scoping needs to put a lot of extra code to decide dynamically the definition to use. In the remaining part of the chapter, we will concentrate on lexical scoping only.

5.4.1 Nested Lexical Scoping

Lexical scope is nested in most of the cases, particularly for block structured languages, which means that in order to reach the definition of a symbol, apart from the current block of code, we also have to consider the blocks that contain this innermost one. The *current scope* is the innermost one. However, there exists a number of *open scopes*, one corresponding to the current scope and others to each of the blocks surrounding it. For example, in the following piece of code, current scope of the variable x is the procedure P_4 while its open scopes include P_1, P_3, and P_4.

```
Procedure P1
    ...
    Procedure P2
        ...
    end procedure
    Procedure P3
        ...
        end Procedure P4
            ...
            x = ...
            ...
        end procedure
    end procedure
end procedure
```

A set of visibility rules is used to resolve conflicts which arise out of the case in which the same variable is defined more than once – may be once in the current scope, and some more in the higher blocks nesting it. In most of the cases, the conflict is resolved as follows:

If a name is defined in more than one scope, the innermost declaration closest to the reference is to be used to interpret the reference to that name.

When a scope is exited, all declared variables in that scope are deleted from the symbol table. The scope that is exited is, thus, *closed.*

There can be two methods for implementing symbol tables with nested lexical scoping.

- One table for each scope and
- A single global table

Before going into the details of these table organizations, we need to understand the operations to be performed on such tables:

1. **Lookup:** The lookup operation to search for a symbol should start at current scope and proceed to other open scopes in a most recent scope first order.

2. **Insert:** To insert a new symbol into the current scope.
3. **Modify:** To modify information about any symbol which is visible at that point.
4. **Create:** To create a new scope whenever a new block is encountered.
5. **Delete:** To delete the most recent scope, once the scope is closed (for example, a block/procedure being over).

5.4.2 One Table Per Scope

In this case, we maintain a different table for each scope. Since the order of the symbol tables created is not explicit, a mechanism is needed to search these tables in the most recent scope first order. This essentially points to using a stack to remember the scopes of symbol tables. The approach has got a few associated problems also.

1. For a single-pass compiler, when a scope is closed, it can be popped out from the stack and destroyed. However, for a multipass compiler, we need to somehow maintain the scope information to be used in code generation during later passes.
2. The serach may be very expensive if the variable is defined much above in the hierarchy of blocks nesting the innermost one where the variable has been referred to.
3. Table size to be allotted to individual blocks is another issue. It may be too small to accommodate a large number of local symbols, or it may be too big for the scope defining only a few symbols.

Just like in case of a single scoped symbol table, here also, we can use lists, trees, hash tables, etc. to organize the symbol tables. However, some extra care needs to be taken to remember the nested scopes.

Lists. Here, each symbol table is organized as a list. A stack of scopes is maintained, and each entry in this stack points to the corresponding symbol table. For example, for the symbol tables shown in Fig. 5.3, the current scope defines the variables a, c and m. However, it also has another open scope with definitions h, a, and l.

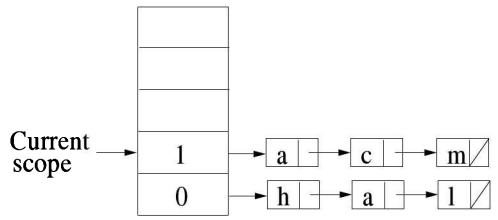

Figure 5.3: Scoped symbol table—lists.

Trees. Similar to lists, here also we have to maintain the scope stack. However the individual tables are created as trees. Each entry of the scope stack contains a pointer to the root of the corresponding symbol tree. The situation is shown in Fig. 5.4.

Hash Tables. Similar to the single scope case, a hash table is built for each scope. Each entry in the scope stack points to the hash table corresponding to the scope.

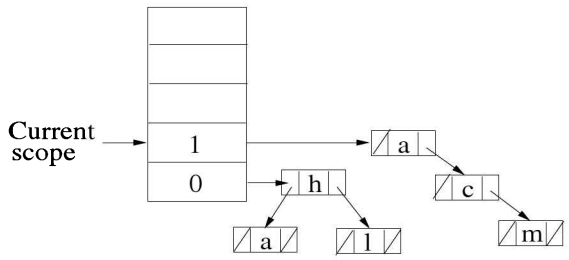

Figure 5.4: Scoped symbol table—trees.

5.4.3 One Table for All Scopes

In this case, there exists a single table into which all variables will be stored. In order to remember the scope of the variables, each entry in the symbol table has an extra field identifying the scope of the symbol. Single table is more efficient in the sense that it does not waste space as compared to the case of one table per scope. However, some space saved in the process is eaten up by the scope number field. As we go into more and more nesting, the scope number increases. Thus, in order to search for a variable, we have to start with the highest scope number and then, try out the entries having the next lesser scope number and so on. As and when a scope gets closed, all the variables with that scope number are removed from the table. Thus, this scheme is suitable particularly for single-pass compilers. The data structure to be used here are again list, tree or hash table.

Lists. It is similar to the one-table-per-scope case, except the fact that all the tables are now concatenated into a single list. The scope number is maintained and when a scope gets closed, that portion of the list is deleted. A typical such list has been shown in Fig. 5.5. It corresponds to the case with three nested scopes—3 being the innermost, 2 next and 1 the outermost. To search for X, the first entry matches, Y matches with the definition in scope 2 and so on.

Figure 5.5: Scoped symbol table with single list.

Trees. It builds a single tree alongwith the scope information stored at each node. For example Fig. 5.6 shows a tree structured table for the nested blocks with H, A, and L appearing at scope 1; and A and C appearing at scope 3. The major difficulty encountered here is that the insertion of a new symbol always occurs at the leaf level. Thus, to search for the definition of a variable, we have to search for matching entry all the way to the leaves till the last matching entry found. Deletion of entries also needs traversing the whole tree to remove symbols from the current scope.

Hash Tables. Single table version of hash table organization is again similar to the hash table with chaining. All collided symbols are contained in the chain pointed to by the hash table entry. Each entry also stores the scope identification. To make sure that the innermost definition is obtained the definitions with higher scope identifier are kept at the beginning of the chains. A typical organization is shown in Fig. 5.7. To close a scope, all entries with the corresponding scope number are removed from the chain.

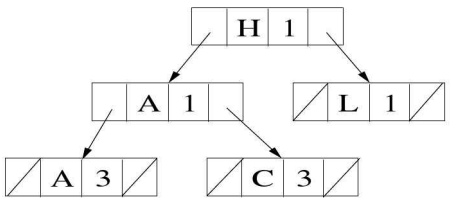

Figure 5.6: Scoped symbol table with single tree.

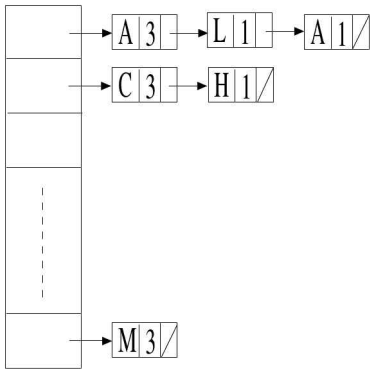

Figure 5.7: Scoped symbols with single hash table.

Many programming languages support record as a data type. A record contains a number of fields, each having a unique name within the record. The fields of records are accessible from the places where the record is visible. As an example, let us consider a record R with two fields–A of type integer and B of type real. The field A may be accessed as $R.A$ and B as $R.B$. To facilitate this access, the symbol table entry for R may point to another symbol table that contains the definitions of A and B. Another technique may be to qualify A and B with R as $R.A$ and $R.B$ and store them in the symbol table as identifiers with names $R.A$ and $R.B$ respectively.

5.5 CONCLUSION

In this chapter, we have seen various structures for symbol table. Symbol table, though not part of the code generated by the compiler, helps in the process of compiling. While the phases like *lexical analysis* and *syntax analysis* produce the symbol table, the other phases use its content. Depending upon the *scope rules* of the language, symbol table needs to be organized in various different manners. The data structures commonly used for symbol table are *linear table, ordered list, tree, hash table* etc.

EXERCISES

5.1 Study the scope rules for the languages Pascal and C.

5.2 Discuss the importance of symbol table in compiler design. How is the symbol table manipulated at various phases of compilation?

5.3 What should be the typical entries in the symbol table for the *C* language?

5.4 Compare various symbol table structures with respect to the operations on the tables.

5.5 Justify or contradict the statement, "Dynamic scoping results in slower execution".

Chapter 6

Runtime Environment Management

6.1 INTRODUCTION

Runtime environment refers to the program snap-shot during execution. A program consists of three main segments—

- code for the program,
- static and global variables,
- local variables and arguments.

Each of the above three segments need proper allocation of memory to hold their values. Thus, three kinds of entities, discussed below, are to be managed at the runtime:

1. *Generated code*: For various procedures and programs that form text or code segment. The size of this segment is known at compile time. Thus, the space can be allocated statically before the execution commences.
2. *Data objects*: They can be of different types.
    - *Global variables/constants:* The size of the total space required by global variables or constants is known at compile time.
    - *Local variables:* for local variables also, the size is known at compile time.
    - *Variables created dynamically:* these variables correspond to the space created in response to the memory allocation requests from the program during execution. Since it depends on the execution sequence of the program, the size is unknown at compile time. The dynamic allocation is done in *heap*.
3. *Stack*: To keep track of procedure activations.

Thus, the logical address space of a program can be viewed as in Fig. 6.1. The *code* occupies the lowest portion. The global variables are allocated in the *static* portion. In the remaining chunk of the address space, stack and heap are allocated from the opposite ends to have maximum flexibility. A program may not use much of the dynamic space, so its heap usage remains less. However, it may be using a lot of stack space due to a large number of nested procedure calls. The reverse may also become true for some other program. Thus, allocating stack and heap from the opposite ends is justified.

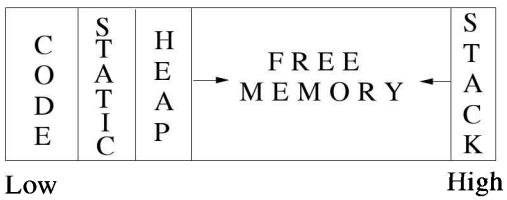

Figure 6.1: Logical address space of program.

6.2 ACTIVATION RECORD

Associated with each activation of a procedure, some storage space is needed for the variables. This storage is called *activation record* or *frame.* A typical activation record contains the following things:

- Parameters passed to the procedure.
- Bookkeeping information, including the return address.
- Space for local variables.
- Space for local temporaries, generated by the compiler to hold subexpression values.

An organization is shown in Fig. 6.2. The size of the bookkeeping information is fixed and is same for all the procedures. The compiler can determine the sizes of other segments.

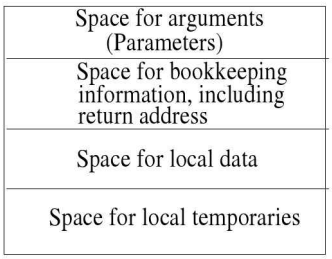

Figure 6.2: A typical activation record.

Depending upon the language, an activation record can be created in any of the *static, stack* or *heap* area. In early languages like FORTRAN, activation records are created in the static area. Thus, the addresses to be used for all the arguments to different procedures and local variables defined within them are preset at compile time itself. Compiler can generate code using these static locations. To pass parameters, the values are copied into these locations at the time of invoking the procedure and copied back on return. The major difficulty with such allocation is that the same space is used for each invocation of a procedure. Thus, at any point of time, there can only be a single activation of a procedure. This makes the implementation of recursive procedures impossible. The next choice for creating activation records is in the *stack.* It is used for languages like *C, Pascal, Java*, etc. As and when a procedure is invoked, the corresponding activation record is pushed onto the stack. At the return, the entry is popped out. This works well if the local variables are not needed beyond the procedure body. For languages like *LISP*, in which a full function may be returned, the allocation of activation record is done in the heap.

It may be noted that processor registers are also a part of runtime environment. They are used to store temporaries, local variables, global variables and some other special information. For example, the *program counter* points to the statement to be executed next while the *stack pointer* points to the top of the stack. A few registers are used to keep track of procedure activations:

- *Frame Pointer (fp):* It points to the current activation record.
- *Argument pointer (ap):* It points to the area of activation record reserved for arguments.

The structure of the activation record, that is, fields contained in it, may vary depending upon the language. If a language supports nested definition of procedures, the access to the variables is defined by certain scope rules.

We will now discuss two different classes of runtime environments and their associated activation record structures.

1. Stack based environment without local procedures and
2. Stack based environment with local procedures.

The first type is common for languages such as *C*, while the second one is followed for block structured languages like *Pascal.*

6.2.1 Environment without Local Procedures

In a language where all procedures are global, a stack based environment needs two things about the activation records.

1. Maintenance of a pointer to the current activation record to allow access to local variables and parameters. This is, normally, the *frame pointer (fp).*
2. A record of position of the immediately preceding activation record, so that it can be retrieved when the current call finishes. This information is kept in the current activation record by means of a pointer to the previous activation record. The pointer is called *control link* or *dynamic link.*

Example 6.1 Consider the following *C*-code consisting of two mutually recursive routines *f* and *g* apart from the function *main.*

```
int x = 2;
void f (int n) {
    static int x = 1;
    g(n);
    x − −;
}
void g (int m) {
    int y = m − 1;
    if (y > 0) {
        f(y);
        x − −;
}}
main() { g(x); return 0;}
```

The *main* calls the function g which, in turn, calls f conditionally. The function f again calls g. The static integer variable x in f can be treated as a global variable, as its value is remains there even if the execution is outside f. A snapshot of the program execution is shown in Fig. 6.3. It is the snapshot after *main* has called g, g has called f, and f has, in turn, called g. The execution is within the second invocation of g.

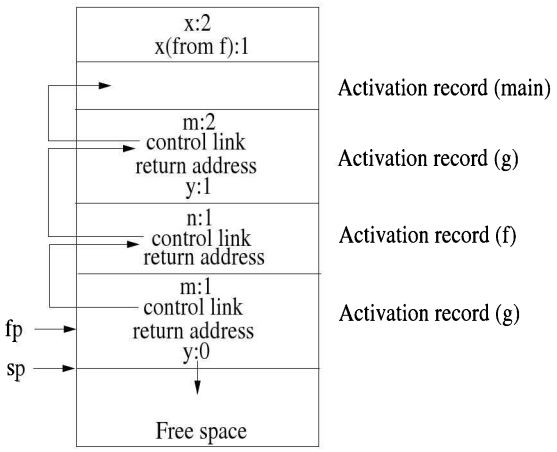

Figure 6.3: An example snapshot.

Accessing Variables

The parameters and local variables are found by offset from the current frame pointer. Offsets can be calculated statically by the compiler as shown in the following example.

Example 6.2 Consider a procedure g with a parameter m and local variable y. For the procedure g its each invocation has the same structure for the activation record. The parameters and local variables are located at fixed offsets from the frame pointer. The structure is shown in Fig. 6.4. Further, assume that the control link field and the return address field are each of four bytes. Then, the offset of m from the frame pointer is given by

$mOffset$ = size of control link = +4 bytes
Similarly, the offset of y from the frame pointer is given by
$yOffset$ = − (size of y + size of return address) = −6
Thus, m and y can be accessed by $4(fp)$ and $-6(fp)$ respectively.

Creation of Activation Record

The activation record is created when a procedure is called. As the components in an activation record include both the parameters passed and the local variables, it cannot be created completely by the caller routine or the callee. Both caller and callee have got specific roles to play both at the time of call, known as calling sequence and at the time of return, called the return sequence. The operations involved are illustrated in Table 6.1.

6.2.2 Environment with Local Procedures

The runtime support provided in earlier section is insufficient for the environment with local procedures. This is because, now, the variables defined have got various scopes. In order to

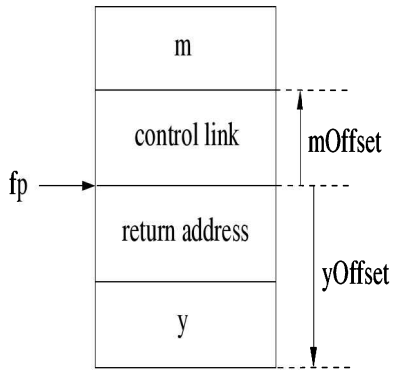

Figure 6.4: Offset calculation for variables.

Table 6.1: Creation of activation record

At a call	
Caller	Callee
1. Allocate basic frame 2. Store parameters 3. Store return address 4. Save caller-saved registers 5. Store self frame pointer 6. Set frame pointer for child 7. Jump to child	1. Save callee-saved registers, state 2. Extend frame (for locals) 3. Initialize locals 4. Fall through to code
At a return	
Caller	Callee
1. Copy return value 2. Deallocate basic frame 3. Restore caller-saved registers	1. Store return value 2. Restore callee-saved registers, state 3. Unextend frame 4. Restore parent's frame pointer 5. Jump to return address

determine the definition to be used for a reference to a variable, we need to access non-local, non-global variables. These definitions are local to one of the procedures nesting the current one. Thus, we need to look into the activation records of these nesting procedures to determine the location.

The solution is to add an extra bookkeeping information, which is called *access link*. The access link points to the activation record that represents defining environment for a procedure.

Example 6.3 Consider the program fragment shown below.

```
program chaining;
procedure p;
var x:  integer;
```

```
    procedure q;
        procedure r
        begin
            x := 2;
            ...
            if ... then p;
        end {of r}
    begin
        r;
    end {of q }
begin
    q;
end {of p}
begin {of main}
    p;
end.
```

The corresponding activation records are shown in Fig. 6.5. The current procedure is r. To locate the definition of x, it has to traverse through the activation records using the access links. Finally, when the required procedure containing the definition of x is reached, it is accessed via offset from the corresponding frame pointer.

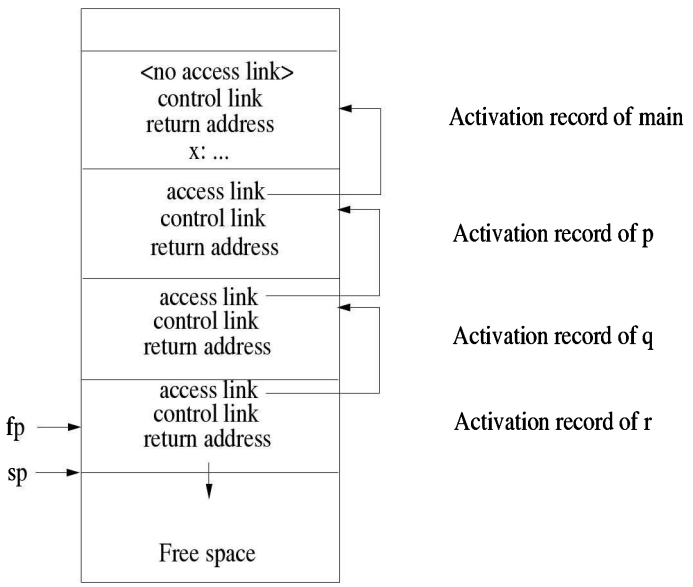

Figure 6.5: Access link chain.

Compiler's responsibility

The compiler is responsible for generating proper code to access the correct definitions. It consists of the following steps:

1. Find the difference d between the lexical nesting level of declaration of the name and the lexical nesting level of the procedure referring to it.
2. Generate code for following d access links to reach the right activation record.
3. Generate code to access the variable through offset mechanism.

Example 6.4 Consider the program shown below having a set of nested procedures.

```
program M;
    procedure P;
        var x, y, z;
        procedure Q;
            procedure R;
            begin
                ...
                z = P;
                ...
            end R;
        begin
            ...
            y = R;
            ...
        end Q;
    begin
        ...
        x = Q;
        ...
    end P;
begin
    ...
    P;
    ...
end M;
```

The procedure nesting graph alongwith lexical levels is shown in Fig. 6.6. Figure 6.7 shows a few snapshots of the activation records of the program. The solid links are the control links, whereas the dotted links are the access links. Figure 6.7(a) shows the situation when M has called P, P has called Q and Q has called R. Fig 6.7(b) shows the situation after R makes a recursive call to P. Here, the control link of P points to R, while the access link points to M, the next higher lexical nesting level. Fig 6.7(c) shows the situation after P has given another call to Q. The control and access links both point to the second invocation of P, since it is the lexical and logical predecessor of the second invocation of Q.

6.3 DISPLAY

A major difficulty faced in accessing non-local definitions is to search by following access links. Due to the virtual paging environment, certain portion of the stack containing activation

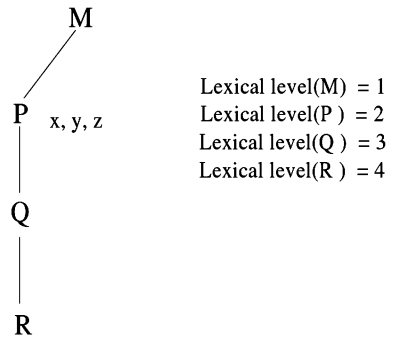

Figure 6.6: Nested procedures.

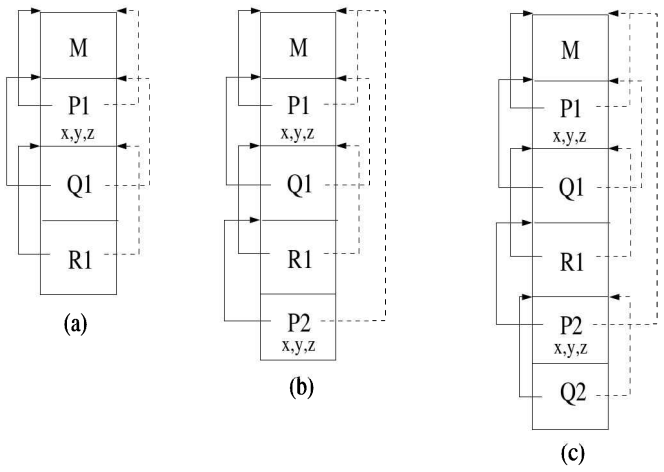

Figure 6.7: Activation records.

records may be swapped out. Thus, the access may be very slow. In order to solve this problem and access the variables without search, *display* is used.

Display d is a global array of pointers to activation records, indexed by the lexical nesting depth. The number of elements in the display is computed at compile time by considering the maximum nesting depth. Array element $d[i]$ points to the most recent activation of the block at nesting depth i. A non-local X is found in the following manner:

1. Use one array access to find the activation record containing X. If the most closely nested declaration of X is at nesting depth i, then $d[i]$ points to the activation record containing the location for X.
2. Use relative address within the activation record to access X.

Example 6.5 For the program code of Example 6.4, a snapshot of activations records and display is shown in Fig. 6.8. Since the maximum nesting depth is four, there are only 4 entries in the display. Figure 6.8(a) corresponds to the situation when the program M has called the procedure P, that, in turn, has called Q which, in turn, has called R. The compiler knows

that the definition of x is available in procedure P at lexical level 2. So, it will generate code to access the second entry of the display to reach the activation record of P directly. Now, suppose R makes a call to P. Then, another activation record is created and pushed onto the stack. However the variables referred to inside P should be either local to P or available in M. Thus, only two entries of the display are now valid – 1 to M and 2 to P, as shown in Fig. 6.8(b).

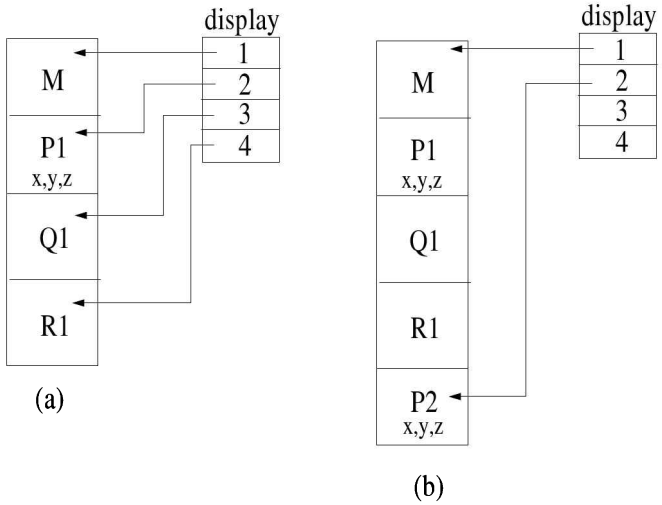

Figure 6.8: Activation records with displays.

Maintaining Display

When a procedure P at nesting depth i is called, the following actions are taken.

1. Save value of $d[i]$ in the activation record for P.
2. Set $d[i]$ to point to new activation record.

Similarly, when P finishes, $d[i]$ is reset to the value stored in the activation record of P. This makes sure that the display entries are set and restored correctly at procedure invocation and return.

6.4 CONCLUSION

In this chapter, we have seen strategies to efficiently manage the storage during program execution. One particular data structure, called *activation record* has been shown to contain the necessary information to control program execution. Compiler writer must generate appropriate code to maintain these activation records and ensure correct access of the variable through these acivation records during program execution. A small array, called *display*, helps in the process. However, display management becomes a part of the compiler's responsibilities.

EXERCISES

6.1 What do you mean by runtime storage allocation? Explain the difference between static and dynamic allocations.

6.2 Why is it the case that statically allocated languages cannot support recursion?

6.3 What is an activation record? Explain clearly the components of an activation record.

6.4 For a language implementation in which the activation records are created in stack, why is it not recommended to have large arrays as local variables and parameters?

6.5 The *C* language uses *Call-by-value* type of parameter passing, that is, the value of the parameters are copied to the arguments at the time of function call. This implies that the modifications done on arguments do not affect the actuals. However, this is not true for an array. Explain why this happens in the light of runtime allocation. Do you recommend the scheme?

6.6 Show the snapshots of the stack of activation records at the time of executing statements at line numbers 6, 11, 14, 17 and 18 of the following program:

```
(1)   Program X
(2)   var x,y,z;
(3)   Procedure P
(4)       var a;
(5)   begin (of P)
(6)       a = Q
(7)   end (of P)
(8)   Procedure Q
(9)       Procedure R
(10)      begin (of R)
(11)          P
(12)      end (of R)
(13)  begin (of Q)
(14)      R
(15)  end (of Q)
(16)  begin (of X)
(17)      P
(18)      Q
(19)  end (of X)
```

6.7 What is a display? How does it help in accelerating the program execution?

6.8 For the problem 6.6, show the corresponding displays.

Chapter 7

Intermediate Code Generation

After looking into the design of parsers, symbol table and storage organization strategies, in this chapter we will proceed with the code generation techniques. When a compiler compiles a program in high-level language, it produces the machine code targeted to some processor. However, many a times, the compilers are designed to produce an intermediate output, which represents the input program in some hypothetical language or data structure. Such representations are known as *intermediate code* to signify that they are representations between the source language and the target machine language programs. The intermediate codes offer a host of advantages as compared to direct code generations, which are given below.

1. Intermediate code is closer to the target machine than the source langauge, and hence, is easier to generate code from.
2. Unlike machine languages, intermediate code is, more or less, machine independent. This makes it easier to retarget the compiler to various different target processors.
3. It allows a variety of optimizations to be performed in a machine-independent fashion.
4. Typically, intermediate code generation can be implemented via *syntax-directed translation*, that is, alongwith the parser while doing various reductions, and thus, can be folded into parsing by augmenting the parser.

7.1 INTERMEDIATE LANGUAGES

The representation techniques in intermediate languages can be classified into two categories:

1. High-level intermediate representation and
2. Low-level intermediate representation.

The *high-level representation* expresses the high-level structure of a program. The example of this category is the syntax tree as produced by the parser.The following are the essential features of this representation:

1. It is closer to the source language program. Thus, it retains the program structure.
2. It is easy to generate from the input program.

3. However, code optimization may not be straight-forward. This is because the input program is not broken down to extract the finer levels of code sharings and optimizations.

The other category, *low-level representation*, on the other hand, expresses the low-level structure of a program. The essential features of this kind of representation are the following:

1. It is closer to target machine. For example, the RTL used in GCC and three-addresses code, which is discussed later.
2. It is easy to generate the final code from this representation.
3. However, the generation of this representation from the input program requires a good amount of effort, as will be discussed in later parts of this chapter.

7.2 INTERMEDIATE LANGUAGE DESIGN ISSUES

Every language design has some specific goals corresponding to the features needed to be represented. For example, the high-level languages attempt to include expression powers so that the user's computation requirements can be represented concisely and efficiently. On the other hand, the assembly language designers concentrate on the available hardware and the instructions map directly to the operations in the underlying hardware. Similarly, intermediate languages have got a set of design issues that emphasize on the bridge between the source language and the target language. The following are some salient points regarding the intermediate language.

1. The set of operators in the intermediate language must be rich enough to allow the source language to be implemented.
2. A small set of operations in the intermediate language makes it easy to retarget.
3. Intermediate code operations that are closely tied to a particular machine or architecture can make it harder to port.
4. A small set of intermediate code operations may lead to long instruction sequences for some source language constructs. This implies more work during optimization.

7.3 INTERMEDIATE REPRESENTATION TECHNIQUES

In this section, we will look briefly into the various intermediate representations. Some of them are high level representations, while others are low level.

7.3.1 High Level Representation

The following are a few typical examples of high level intermediate representations:

- Abstract Syntax Trees
- Directed Acyclic Graphs
- P-code

Abstract Syntax Trees: An *Abstract Syntax Tree (AST)* or simply *Syntax Tree* is a compacted form of a parse tree that represents the hierarchical structure of the program. The nodes represent operators, while the children of a node represent the operands on which the operator operates.

Example 7.1 Consider the following piece of code in the source language:

$$\textbf{if } x > 0 \textbf{ then } x = 3 * (y + 1) \textbf{ else } y = y + 1$$

The syntax tree for this is shown in Fig. 7.1.

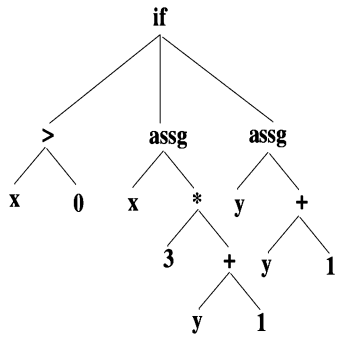

Figure 7.1: Syntax Tree.

Directed Acyclic Graphs (DAGs)

Directed Acyclic Graphs are similar a to syntax trees, except the fact that the common subexpressions are represented here by a single node. The DAG corresponding to the syntax tree of Fig. 7.1 has been shown in Fig. 7.2.

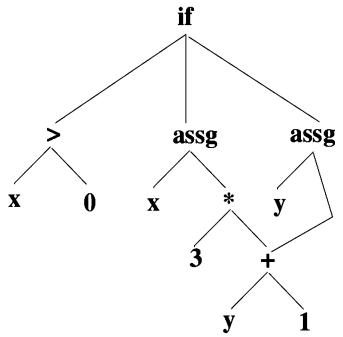

Figure 7.2: Directed Acyclic Graph.

P-code

This representation is used for stack-based virtual machines. In these machines, the operands for the operators are always found on the top of the stack. This may necessitate pushing the operands first into the stack. Apart from expression, other language constructs are translated as usual.

The intermediate representations can be converted easily from one format to another. For example, Fig. 7.3(a) shows a complex expression syntax tree. The corresponding possible P-codes are shown in Fig. 7.3(b).

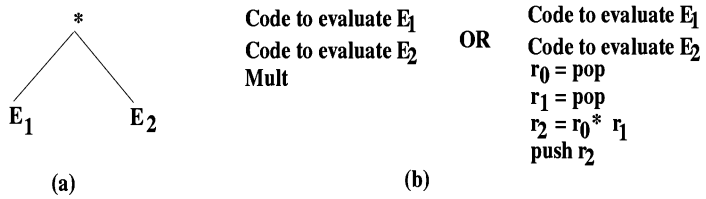

Figure 7.3: Syntax tree to P-code.

7.3.2 Low Level Representation

A typical example of low level intermediate representation is *three address code.* The salient features of this representation are as follows:

1. This is a sequence of instructions of the form

$$x = y \ op \ z,$$

   where x, y and z are variable names, constants, or temporary variables generated by the compiler to compute subexpressions.
2. Only one operator is permitted in the right hand side, ensuring that the operations and expressions are simple. Thus, a source language statement like

$$x = y * z + w * a$$

   may translate to

$$t_1 = y * z$$
$$t_2 = w * a$$
$$x = t_1 + t_2$$

Because of its simplicity, the three address code offers better flexibility in terms of target code generation and code optimization. In remaining part of the chapter, we will concentrate only on the three address code based representations.

7.4 STATEMENTS IN THREE–ADDRESS CODE

As discussed earlier, the statements in three–address code are such that the source language statements can be efficiently represented. Keeping this in view, the intermediate languages, usually, have the following types of statements:

Assignment. The three types of assignment statements, which are required, are

$x = y$ op z, op being a binary operator
$x =$ op y, op being a unary operator
$x = y$

It should be noted that for all operators in the source language, there should be a counterpart in the intermediate language. Any other translation may lead to unoptimized code.

Jumps. Both unconditional and conditional jumps are required.
Unconditional jump is of the form,

goto L, L being a label

The conditional jump has the form

if x relop y goto L

Indexed Assignment.

$x = y[i]$
$x[i] = y$

Only one dimensional arrays need to be supported. Arrays of higher dimensions are converted to one-dimensional arrays by techniques discussed later.

Address and Pointer Assignments. Languages like Pascal and C allow pointer assignments. The three address codes representing these languages also need statements handling pointers. The following are the types of statements required:

$x = \&y$, address of y assigned to x
$x = *y$, content of location pointed to by y is assigned to x
$x = y$, simple pointer assignment, where x and y are pointer variables

Procedure Call/Return. A call to the procedure $P(x_1, x_2, \ldots, x_n)$ with the parameters $x_1, x_2, \ldots, x_n$ is converted as

```
param x1
param x2
  ⋮
param xn
call P, n
```

A procedure is implemented using the following statements (over and above the other three-address statements):

```
enter f, setup and initialization
leave f, cleanup actions (if any)
return
return x
retrieve x, save returned value in x
```

Miscellaneous. There may be some other statements needed depending upon the source language. However, one such statement to define the jump target is

label L

7.5 IMPLEMENTATION OF THREE–ADDRESS INSTRUCTIONS

There can be several possible implementation for the three-address instructions. Amongst them, the *quadruple* representation is the most widely used because of its ease of optimization. In this representation, every instruction is represented by structures having at most four fields:

- *Operation* – identifying the operation to be carried out.
- Upto two *Operands.*
- *Destination*

For the *Operands* fields, a bit is used to indicate whether it is a constant or a pointer for variables into the symbol table. Two examples of such quadruples are shown in Fig 7.4, corresponding to the assignment $x = y + z$ and conditional statements *if* $t_1 \geq t_2$ *goto* L.

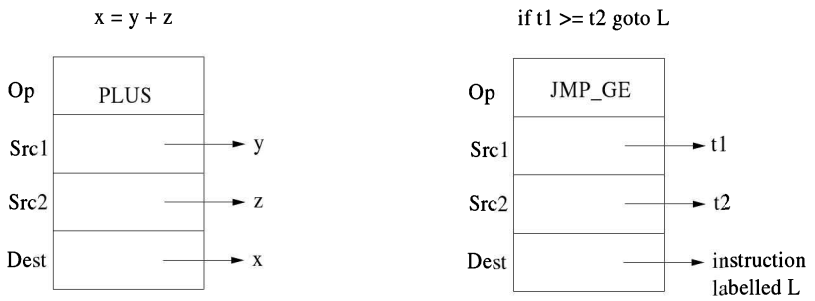

Figure 7.4: Three address code structure.

Example 7.2 Let us consider the source language statement,

$$\textbf{if } x + 2 > 3 * (y - 1) + 4 \textbf{ then } z = 0.$$

The three-address code corresponding to this is as shown below:

$t_1 = x + 2$
$t_2 = y - 1$
$t_3 = 3 * t_2$
$t_4 = t_3 + 4$
—ttif $t_1 \leq t_4$ goto L
$z = 0$
Label L

7.6 THREE–ADDRESS CODE GENERATION

Three address code for a source language program can be generated efficiently by associating some attributes to the terminals and nonterminals of the grammar and computing the values of these attributes as parsing proceeds following *syntax directed translation.* The set of attributes

to be associated depends upon the source language constructs. To start with, let us consider the grammar for assigning an expression to an identifier as per the following grammar:

$$\begin{aligned} S &\rightarrow \mathbf{id} := E \\ E &\rightarrow E + E \mid E * E \mid -E \mid (E) \mid \mathbf{id} \end{aligned}$$

With the nonterminal E, two attributes $E.place$ and $E.code$ are assigned. $E.place$ is the name that will hold the value of E and $E.code$ is a sequence of three-address statements corresponding to the evaluation of E. For S, we have only one attribute $S.code$, which is a sequence of three-address statements. For the terminal symbol **id**, the attribute **id**.*place* contains the name of the variable to be assigned. We use two more functions to write the semantic rules. The function *newtemp()* without any argument returns a unique new temporary variable, while the function *gen* accepts a string and produces it as a three-address quadruple. The operator '||' corresponds to the concatenation and is used to join two three-address code segments. The semantic actions are shown in Table 7.1.

Table 7.1: Three-address code generation for assignment statements

Grammar Rule	Semantic Actions
$S \rightarrow \mathbf{id} := E$	$S.code := E.code \|\|$gen(**id**.place ':=' $E.place$)
$E \rightarrow E_1 + E_2$	$E.place :=$ newtemp();
	$E.code := E_1.code \|\| E_2.code \|\|$gen($E.place$ ':=' $E_1.place$ '+' $E_2.place$)
$E \rightarrow E_1 * E_2$	$E.place :=$ newtemp();
	$E.code := E_1.code \|\| E_2.code \|\|$gen($E.place$ ':=' $E_1.place$ '*' $E_2.place$)
$E \rightarrow -E_1$	$E.place :=$ newtemp();
	$E.code := E_1.code \|\|$gen($E.place$ ':=' 'uminus' $E_1.place$)
$E \rightarrow (E_1)$	$E.place := E_1.place$;
	$E.code := E_1.code$
$E \rightarrow \mathbf{id}$	$E.place :=$ **id**.*place*;
	$E.code :=$ ' '

Example 7.3 Consider the statement:

$$x := (y + z) * (-w + v)$$

The annotated parse tree for the statement is shown in Fig. 7.5. The reduction sequence for a bottom-up parser is marked with the encircled numbers in the figure. The actions at various stages of reductions encircled in the figure are as shown in Table 7.2.

7.6.1 Code Generation for Arrays

Elements of a one-dimensional array A are represented as $A[i]$ in the source language. Assuming that the lowest and the highest indices of A are *low* and *high* respectively, and each element has a width w, the element $A[i]$ starts at location

$$base + (i - low) * w$$

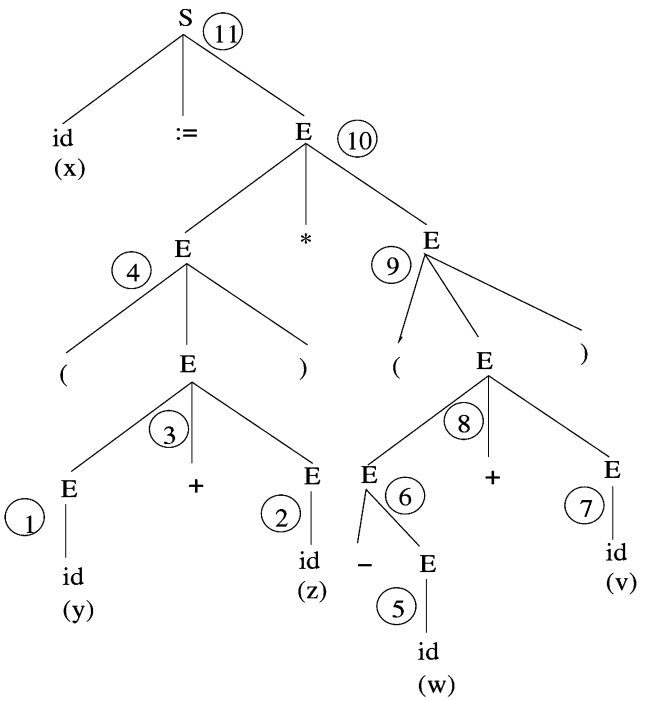

Figure 7.5: Parse tree.

Table 7.2: Code generation

Reduction No.	Action
1	$E.place = y$
2	$E.place = z$
3	$E.place = t_1$
	$E.code = \{t_1 := y + z\}$
4	$E.place = t_1$
	$E.code = \{t_1 := y + z\}$
5	$E.place = w$
6	$E.place = t_2$
	$E.code = \{t_2 := uminus \;\; w\}$
7	$E.place = v$
8	$E.place = t_3$
	$E.code = \{t_2 := uminus \;\; w, t_3 := t_2 + v\}$
9	$E.place = t_3$
	$E.code = \{t_2 := uminus \;\; w, t_3 := t_2 + v\}$
10	$E.place = t_4$
	$E.code = \{t_1 := y + z, t_2 := uminus \;\; w, t_3 := t_2 + v, t_4 := t_1 * t_3\}$
11	$S.code = \{t_1 := y + z, t_2 := uminus \;\; w, t_3 := t_2 + v, t_4 := t_1 * t_3, x := t_4\}$

base being the start address of the storage allotted to A. The expression can be rewritten as

$$(base - low * w) + i * w$$

For the array, the values of *base, low, high* and w are known beforehand. Thus, the compiler can generate code only for the offset $i * w$, whereas the first part of the expression can be precomputed into a constant and added to the offset.

Similarly, for a two-dimensional array with row-major storage, an element $A[i_1, i_2]$ starts at location

$$base + ((i_1 - low_1) * n_2 + i_2 - low_2) * w$$

n_2 being the size of the second dimension. This can again be rewritten as

$$base - ((low_1 * n_2) + low_2) * w) + ((i_1 * n_2) + i_2) * w$$

that can easily be extended to higher dimensions.

Translation scheme. The following is the grammar for assignment statement, where the identifiers may be array elements instead of simple variables. The indices to the arrays may also be expressions.

$$\begin{array}{rcl} S & \rightarrow & L := E \\ E & \rightarrow & E + E \mid (E) \mid L \\ L & \rightarrow & Elist] \mid \textbf{id} \\ Elist & \rightarrow & Elist, E \mid \textbf{id}[E \end{array}$$

The attributes associated with the nonterminals are as follows:

- $L.place$: holds the name of the variable (may be array name also).
- $L.offset$: is null if it refers to a simple variable, otherwise for arraying offset of the element within the array.
- $E.place$: name of the variable that holds the value of the expression E.
- $Elist.array$: holds the name of the array referred to.
- $Elist.place$: name of the variable holding value for the index expression. For example, for the reference $A[i_1, i_2]$, it holds the value of the expression $(i_1 * n_2 + i_2)$.
- $Elist.dim$: holds the current dimension under consideration for the array.

The semantic actions are as follows:

1. $S \rightarrow L := E$

```
{ if L.offset = null then
      emit(L.place ':=' E.place);
  else
      emit(L.place'['L.offset']' ':=' E.place)
}
```

2. $E \rightarrow E_1 + E_2$

   { $E.place$:= newtemp();
   emit($E.place$ ':=' $E_1.place$ '+' $E_2.place$)
   }

3. $E \rightarrow (E_1)$

   { $E.place := E_1.place$};

4. $E \rightarrow L$

   { **if** $L.offset$ = null **then**
   $E.place := L.place$
   else
   $E.place$:= newtemp();
   emit($L.place$ ':=' $L.place$ '[' $L.offset$ ']')
   }

5. $S \rightarrow L := Elist$

   { $L.place$ = newtemp();
   $L.offset$ = newtemp();
   emit($L.place$ ':=' $c(Elist.array)$); /* c returns constant part of address */
   emit($L.offset$ ':=' $Elist.place * width(Elist.array)$)
   }

6. $L \rightarrow$ **id**

   { $L.place$ = **id**.$place$;
   $L.offset$ ':=' null;
   }

7. $Elist \rightarrow Elist_1, E$

   { t := newtemp();
   $m := Elist_1.dim + 1$;
   emit(t ':=' $Elist_1.place$ '*' $limit(Elist_1.array, m)$);
   emit(t ':=' t '+' $E.place$);
   $Elist.array = Elist_1.array$;
   $Elist.place = t$;
   $Elist.dim = m$;
   }

8. $Elist \rightarrow$ **id**[E

   { $Elist.array$ = **id**.$place$;
   $Elist.place = E.place$;
   $Elist.dim = 1$;
   }

Example 7.4 Consider the following assignment statement:

$$X[i, j] := Y[i + j, k] + Z[k, l]$$

Assume the arrays to be of dimensions $X[d_1, d_2]$, $Y[d_3, d_4]$ and $Z[d_5, d_6]$ and each element of all the arrays to be of width w. The annotated parse tree for the statement is shown on Fig. 7.6. The code generation has been explained in Table 7.3.

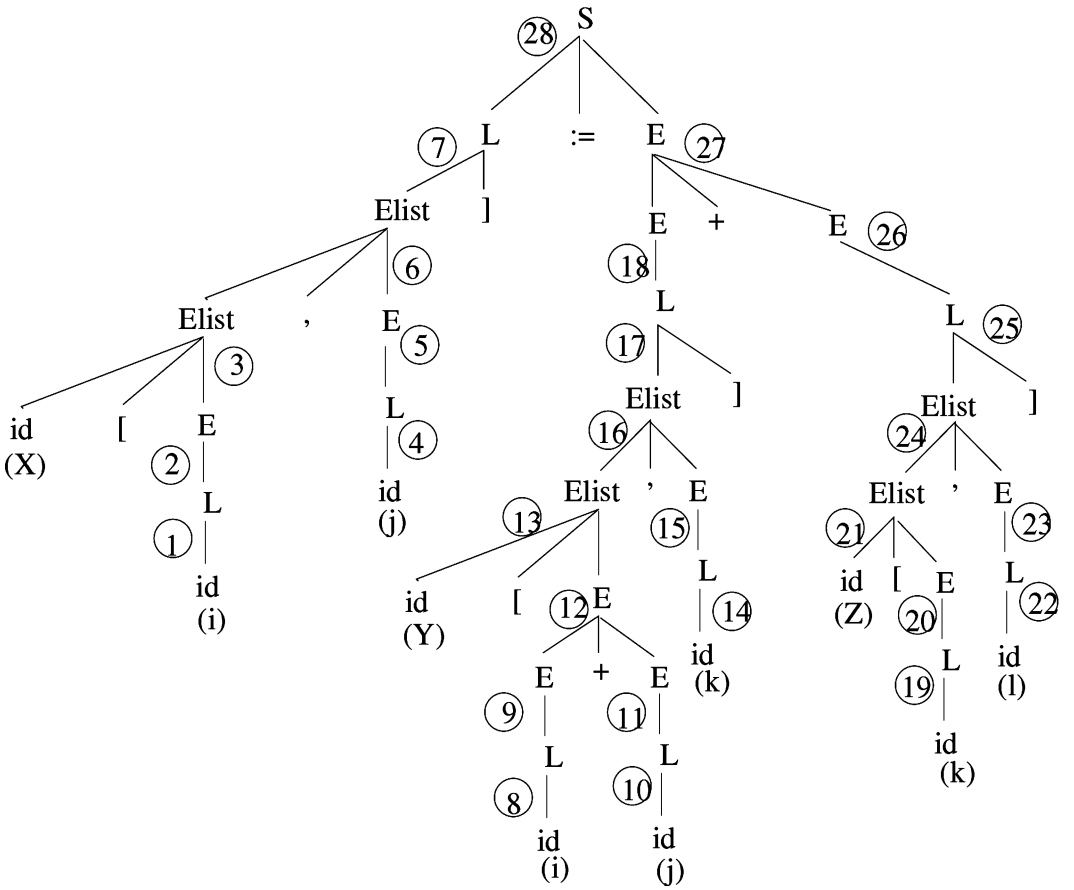

Figure 7.6: Parse tree.

7.6.2 Translation of Boolean Expressions

In most of the programming languages, Boolean expressions are used in the conditional statements to express the condition. The expression evaluates to either *true* or *false*, and accordingly, program control jumps to one statement or the other. Thus, we can associate two attributes with a Boolean expression B:

1. $B.true$: defining the place control should reach if B is true
2. $B.false$: defining the place control should reach if B is false

Table 7.3: Code generation process

Step No.	Attribute assignment	Code generated
1	$L.place = i, L.offset = null$	
2	$E.place = i$	
3	$Elist.array = X, Elist.place = i, Elist.dim = 1$	
4	$L.place = j, L.offset = null$	
5	$E.place = j$	
6	$Elist.array = X, Elist.place = t_1, Elist.dim = 2$	$t_1 = i * d_2, t_1 := t_1 + j$
7	$L.place = t_2, L.offset = t_3$	$t_2 := C(X), t_3 := t_1 * w$
8	$L.place = i, L.offset = null$	
9	$E.place = i$	
10	$L.place = j, L.offset = null$	
11	$E.place = j$	
12	$E.place = t_4$	$t_4 := i + j$
13	$Elist.array = Y, Elist.place = t_4, Elist.dim = 1$	
14	$L.place = k, L.offset = null$	
15	$E.place = k$	
16	$Elist.array = Y, Elist.place = t_5, Elist.dim = 2$	$t_5 := t_4 * d_4, t_5 := t_5 + k$
17	$L.place = t_6, L.offset = t_7$	$t_6 := C(Y), t_7 := t_5 * w$
18	$E.place = t_8$	$t_8 := t_6[t_7]$
19	$L.place = k, L.offset = null$	
20	$E.place = k$	
21	$Elist.array = Z, Elist.place = k, Elist.dim = 1$	
22	$L.place = l, L.offset = null$	
23	$E.place = l$	
24	$Elist.array = Z, Elist.place = t_9, Elist.dim = 2$	$t_9 := k * d_6, t_9 := t_9 + l$
25	$L.place = t_{10}, L.offset = t_{11}$	$t_{10} := C(Z), t_{11} := t_9 * w$
26	$E.place = t_{12}$	$t_{12} := t_{10}[t_{11}]$
27	$E.place = t_{13}$	$t_{13} := t_8 + t_{12}$
28		$t_2[t_3] := t_{13}$

Let the grammar for Boolean expressions be

$$B \rightarrow B \textbf{ or } B \mid B \textbf{ and } B \mid \textbf{not } B \mid (B) \mid \textbf{id relop id} \mid \textbf{true} \mid \textbf{false}$$

A three-address code generation scheme for $B \rightarrow B_1$ or B_2 can be like

```
{ B1.true = B.true
  B1.false = newlabel()
  B2.true = B.true
  B2.false = B.false
  B.code = B1.code || gen(B1.false ':') || B2.code
}
```

Here we assume that the true and false transfer points for the entire expression B are known beforehand. If B_1 is true, then the entire expression B is true. We need not to evaluate B_2. This shortening of evaluation for Boolean expression is also known as *short-circuit evaluation.* Thus, if B_1 is true, then the control can be transferred to the point corresponding to B.true. However, if B_1 is false, then B_2 needs to be evaluated. Thus, B_1.false is assigned a new label marking the beginning of evaluation of B_2. This Table is generated by a call to the function *newlabel* (). The full set of rules for translation is shown next.

1. $B \rightarrow B_1$ **or** B_2

   { B_1.true = B.true
   B_1.false = newlabel()
   B_2.true = B.true
   B_2.false = B.false
   B.code = B_1.code || gen(B_1.false ':') || B_2.code
   }

2. $B \rightarrow B_1$ **and** B_2

   { B_1.true = newlabel()
   B_1.false = B.false
   B_2.true = B.true
   B_2.false = B.false
   B.code = B_1.code || gen(B_1.true ':') || B_2.code
   }

3. $B \rightarrow$ **not** B_1

   { B_1.true = B.false
   B_1.false = B.true
   B.code = B_1.code
   }

4. $B \rightarrow (B_1)$

   { B_1.true = B.true
   B_1.false = B.false
   B.code = B_1.code
   }

5. $B \rightarrow$ **id$_1$ relop id$_2$**

   {
   B.code = gen('if' **id$_1$**.place **relop id$_2$**.place 'goto' $B.true$) || gen('goto' $B.false$)
   }

6. $B \rightarrow$ **true**

   {
   B.code = gen('goto' $B.true$);
   }

7. $B \rightarrow$ **false**

   {
   B.code = gen('goto' $B.false$);
   }

However, this style of translation makes the scheme inherently a two-pass procedure, where we compute all the jump targets in the first pass, i.e., $B.true$ and $B.false$, for all Boolean expressions appear within conditional statements. However, the actual code generation occur in the second pass. The process can be clubbed into a single phase by modifying the grammar a little bit, alongwith the introduction of a few more attributes for B and a few new procedures. The new attributes of B are as follows:

1. $B.truelist$: It contains the list of locations within the generated three-address code for B, at which B is definitely true. Once defined all these points should transfer the control to $B.true$.
2. $B.falselist$: similar to $B.truelist$. However, it contains all the locations where B is definitely false and once defined, should be filled up with the value $B.false$.

The generated three-address code is visualized as an array of quadruples. The following functions are defined to carry out various tasks during code generation:

1. *Createlist(i):* It creates a new list with a single entry i – an index into the array of quadruples.
2. *mergelist(*$list_1$*,* $list_2$*):* It returns a new list containing $list_1$ followed by $list_2$.
3. *backpatch(list, target):* It inserts the *target* as the target label into each quadruple pointed to by the entries in *list*.

To ease the code generation process, we introduce a dummy nonterminal M with an attribute $M.quad$, which can hold the address of a quadruple. The modified grammar is

$$B \rightarrow B \textbf{ or } MB \mid B \textbf{ and } MB \mid \textbf{not } B \mid (B) \mid \textbf{id relop id} \mid \textbf{true} \mid \textbf{false}$$
$$M \rightarrow \epsilon$$

The rule $M \rightarrow \epsilon$ has got only one action during reduction:

$$\{M.quad = nextquad()\}$$

Here, the function *nextquad()* returns the address of the next quadruple to be generated. To understand its role in the code generation process, let us consider the rule

$$B \rightarrow B_1 \textbf{ or } MB_2$$

Since before the reduction for B_2 starts, the reduction of $M \rightarrow \epsilon$ has already taken place, $M.quad$ points to the address of the first quadruple of B_2. Thus, the semantic actions for this rule can be written as

$$
\begin{array}{l}
\{\ \text{backpatch}(B_1.falselist,\ M.quad) \\
\quad B.truelist = \text{mergelist}(B_1.truelist, B_2.truelist) \\
\quad B.falselist = B_2.falselist \\
\}
\end{array}
$$

Since, we have to evaluate B_2 if B_1 is false, $B_1.falselist$ has been filled up with address of the starting quadruple of B_2. The full truelist for B is obtained by merging the truelists of B_1 and B_2. The falselist of B is same as that of B_2. The complete set of rules is shown next.

1. $B \rightarrow B_1$ **or** MB_2

$$
\begin{array}{l}
\{\ \text{backpatch}(B_1.falselist,\ M.quad) \\
\quad B.truelist = \text{mergelist}(B_1.truelist, B_2.truelist) \\
\quad B.falselist = B_2.falselist \\
\}
\end{array}
$$

2. $B \rightarrow B_1$ **and** MB_2

$$
\begin{array}{l}
\{\ \text{backpatch}(B_1.truelist,\ M.quad) \\
\quad B.truelist = B_2.truelist \\
\quad B.falselist = \text{mergelist}(B_1.falselist, B_2.falselist) \\
\}
\end{array}
$$

3. $B \rightarrow$ **not** B_1

$$
\begin{array}{l}
\{\ B.truelist = B_1.falselist \\
\quad B.falselist = B_1.truelist \\
\}
\end{array}
$$

4. $B \rightarrow (B_1)$

$$
\begin{array}{l}
\{\ B.truelist = B_1.truelist \\
\quad B.falselist = B_1.falselist \\
\}
\end{array}
$$

5. $B \rightarrow$ **true**

$$
\begin{array}{l}
\{\ B.truelist = makelist(nextquad) \\
\quad \text{emit('goto ...')} \\
\}
\end{array}
$$

6. $B \rightarrow$ **false**

$$
\begin{array}{l}
\{\ B.falselist = makelist(nextquad) \\
\quad \text{emit('goto ...')} \\
\}
\end{array}
$$

7. $M \rightarrow \epsilon$

$$\{M.quad = nextquad\}$$

Example 7.5 Consider the Boolean expression

$$x > y \text{ or } z > k \text{ and not } r < s$$

The corresponding parse tree has been shown in Fig. 7.7. Assuming that code generation starts from quadruple number 1, that is, *nextquad* is initialized to 1, actions at various reductions are as follows:

Red. 1: `B.truelist` $= \{1\}$
`B.falselist` $= \{2\}$

Following lines of codes are generated:

1: `if x > y goto ...`
2: `goto ...`

Red. 2: `M.quad` $= 3$

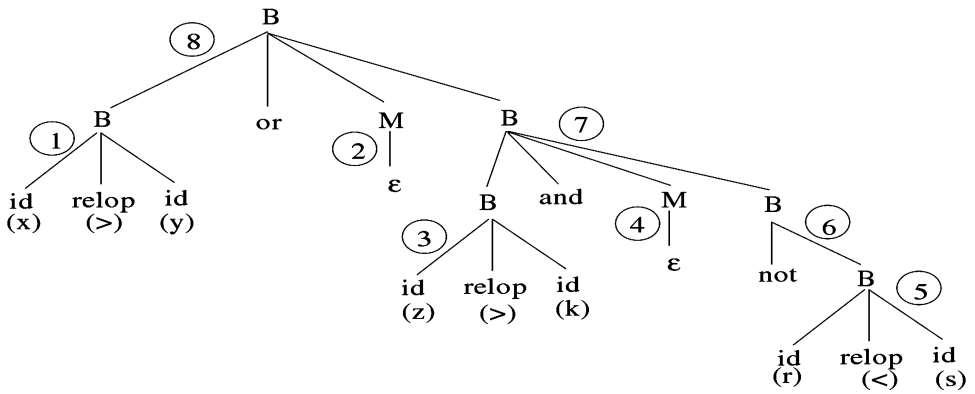

Figure 7.7: Parse tree for Boolean expression.

Red. 3: `B.truelist` $= \{3\}$
`B.falselist` $= \{4\}$

Codes generated are as follows:

3: `if z > k goto ...`
4: `goto ...`

Red. 4: `M.quad` $= 5$

Red. 5: `B.truelist` $= \{5\}$
`B.falselist` $= \{6\}$

Codes generated are as follows:

5: `if r < s goto ...`
6: `goto ...`

Red. 6: `B.truelist` = {6}
`B.falselist` = {5}

Red. 7: It first backpatches the list {3} with 5. Thus, quadruple 3 is modified as:

3: `if` $z > k$ `goto 5`

Red. 8: It backpatches the list {2} with 3. Thus, quadruple 2 is modified as:

2: `goto 3`

`B.truelist` = {1, 6}
`B.falselist` = {4, 5}

Thus, the full code generated is as follows:

1: `if x > y goto ...`
2: `goto 3`
3: `if z > k goto 5`
4: `goto ...`
5: `if r < s goto ...`
6: `goto ...`

The branch target of 1 and 6 is the true exit point and that of 4 and 5 is the false exit point of the entire Boolean expression.

7.6.3 Translation of Control Flow Statements

Most of the programming languages have a common set of statements that define the control flow of a program written in that language. These statements are:

1. *Assignment statement:* It has a single statement assigning some expression to a variable. After executing this statement, the control flows to the immediate next statement in the sequence.
2. *if-then-else statement:* It has a condition associated with it. The control flows either to the *then-part* or to the *else-part.*
3. *while-do loop:* The control remains within the loop until a specified condition becomes false.
4. *Block of statements:* It is group of statements put within a *begin-end* block marker.

The grammar for these statements is given by,

$$\begin{array}{rcl} S & \rightarrow & \textbf{if } B \text{ then } S \\ & | & \textbf{if } B \textbf{ then } S \textbf{ else } S \\ & | & \textbf{while } B \textbf{ do } S \\ & | & \textbf{begin } L \textbf{ end} \\ & | & A \text{ /* } \textbf{for assignment} \text{ */} \\ L & \rightarrow & L \ S \\ & | & S \end{array}$$

To aid the code generation process, the grammar is slightly modified by introducing two new marker nonterminals M and N. The modified grammar is as follows:

$$\begin{array}{rcl} S & \rightarrow & \textbf{if } B \textbf{ then } M \ S \\ & | & \textbf{if } B \textbf{ then } M \ S \ N \textbf{ else } M \ S \\ & | & \textbf{while } M \ B \textbf{ do } M \ S \\ & | & \textbf{begin } L \textbf{ end} \\ & | & A \text{ /* } \textbf{for assignment} \text{ */} \\ L & \rightarrow & L \ M \ S \\ & | & S \\ M & \rightarrow & \epsilon \\ N & \rightarrow & \epsilon \end{array}$$

Apart from the set of attributes for B, as in Boolean expressions, a few more attributes are defined as follows:
S.nextlist: It is a list of quadruples containing jumps to the quadruple following S.
L.nextlist: It is same as S.nextlist, except the fact that L stands for a group of statements.

The nonterminal N has been introduced to enable us to generate a jump after the *then* part of *if-then-else* statement. The attribute N.nextlist holds the quadruple number for this statement. The entire set of actions are as follows:

1. $S \rightarrow$ **if** B **then** $M \ S_1$

   { backpatch(B.truelist, M.quad)
   S.nextlist := mergelist(B.falselist, S_1.nextlist) }

2. $S \rightarrow$ **if** B **then** $M_1 S_1 N$ **else** $M_2 S_2$

   { backpatch(B.truelist, M_1.quad)
   backpatch(B.falselist, M_2.quad)
   S.nextlist := mergelist(S_1.nextlist, mergelist(N.nextlist, S_2.nextlist)) }

3. $S \rightarrow$ **while** $M_1 B$ **do** $M_2 S_1$

{ backpatch(S_1.nextlist, M_1.quad)
backpatch(B.truelist, M_2.quad)
S.nextlist := B.falselist
emit('goto' M_1.quad) }

4. $S \rightarrow$ begin L end

{ S.nextlist := L.nextlist }

5. $S \rightarrow A$

{ S.nextlist := nil }

6. $L \rightarrow L_1 \quad M \quad S$

{ backpatch(L_1.nextlist, M.quad)
L.nextlist := S.nextlist }

7. $L \rightarrow S$

{ L.nextlist := S.nextlist }

Example 7.6 Consider the following piece of code:

```
begin
  while a > b do
  begin
    x = y + z
    a = a - b
  end
  x = y - z
end
```

The parse tree for the code segment has been shown in Fig. 7.8. The activities at various stages of reduction are as shown in Table 7.4.

The final code generated is as follows:

```
1: if a > b goto 3
2: goto 8
3: t1 = y + z
4: x = t1
5: t2 = a - b
6: a = t2
7: goto 1
8: t3 = y - z
9: x = t3
```

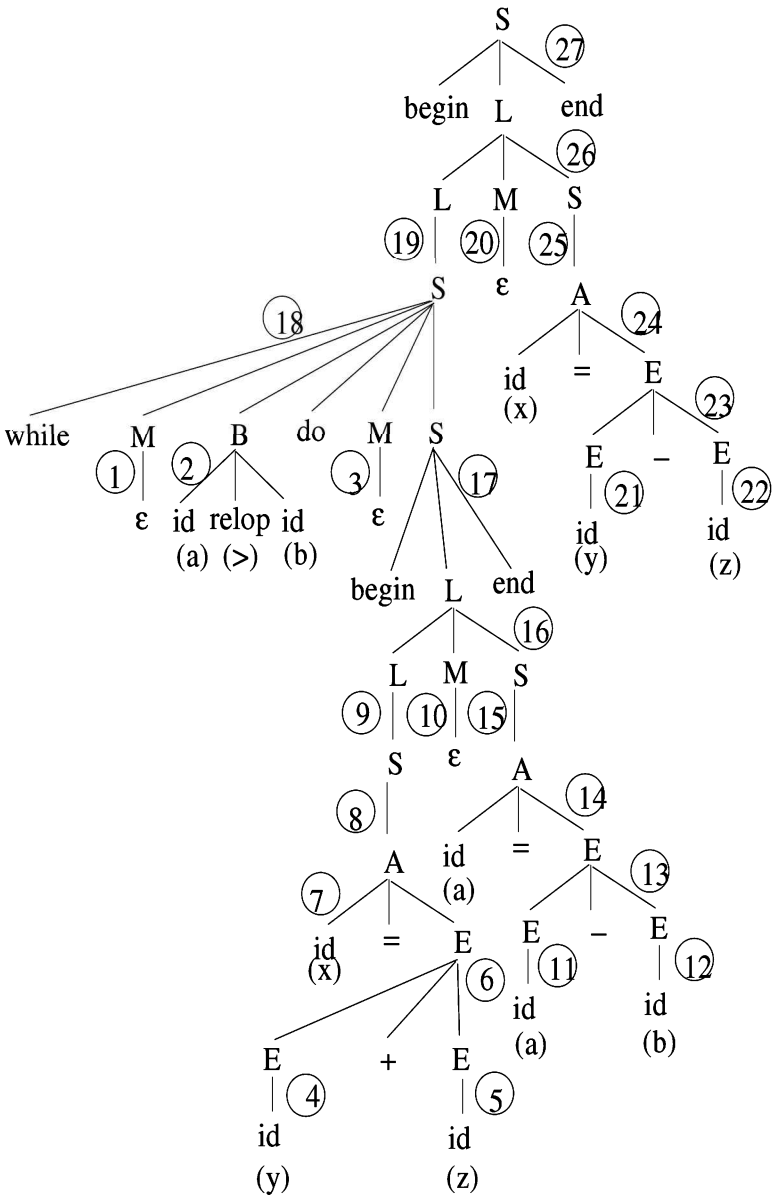

Figure 7.8: Parse tree.

7.6.4 Translation of Case Statements

Case statements are a bit unique in the sense that the structure contains an expression, and depending upon the value of the expression, control jumps to one of the many alternatives provided. Though there are slight variations in various languages, the statement normally looks like

Table 7.4: Code generation actions

Red. no.	Action
1	$M.quad = 1$
2	$B.truelist = \{1\}$, $B.falselist = \{2\}$
	Code generated:
	1: if $a > b$ goto ...
	2: goto ...
3	$M.quad = 3$
4	$E.place = y$
5	$E.place = z$
6	$E.place = t_1$
	Code generated:
	3: $t_1 = y + z$
7	Code generated:
	4: $x = t_1$
8	$S.nextlist = \{\}$
9	$L.nextlist = \{\}$
10	$M.quad = 5$
11	$E.place = a$
12	$E.place = b$
13	$E.place = t_2$
	Code generated:
	5: $t_2 = a - b$
14	Code generated:
	6: $a = t_2$
15	$S.nextlist = \{\}$
16	Backpatch({}, 5)
	$L.nextlist = \{\}$
17	$S.nextlist = \{\}$
18	backpatch({},1)
	backpatch({1},3) ⇒ Code modified as:
	1: if $a > b$ goto 3
	$S.nextlist = \{2\}$
	Code generated:
	7: goto ...
19	$L.nextlist = \{2\}$
20	$M.quad = 8$
21	$E.place = y$
22	$E.place = z$
23	$E.place = t_3$
	Code generated:
	8: $t_3 = y - z$
24	Code generated:
	9: $x = t_3$
25	$S.nextlist = \{\}$
26	Backpatch ({2},8) ⇒ Code modified as:
	2: goto 8
	$L.nextlist = \{\}$
27	$S.nextlist = \{\}$

```
switch (E) {
    case c1: ...
    ⋮
```

```
    case c_n: ...
    default: ...
}
```

The procedure to generate code so that we can efficiently choose one of a set of different alternatives, can use one of the following alternative implementations, depending on the value of the expression:

1. Linear search for the matching option.
2. Binary search for the matching case.
3. A jump table.

While the linear or binary search may be cheaper if the number of cases is small, for larger number of cases, a jump table may be cheaper. On the other hand, if the case values are not clustered closely together, a jump table may be too costly as far as the space requirement is concerned. For example, the following is a bad situation for jump table implementation:

```
switch (E) {
    case 1: ...
    case 1000: ...
    case 1000000: ...
}
```

Implementing Jump Tables. Let the maximum and the minimum case values be c_{max} and c_{min} respectively. The structure of the generated code is:

```
Code to evaluate E into t
if t < c_min goto Default_Case
if t > c_max goto Default_Case
goto JumpTable[t]
Default_case: ...
```

JumpTable is an array of addresses: *JumpTable*[i] is the address of the code to execute if E is evaluated into i. However, special treatment is needed for holes in the range of values. One possible strategy may be to fill up the holes with *Default_case*.

7.6.5 Function Calls

The processing of the function calls can be divided into two subsequences:

1. Calling Sequence–the set of actions executed at the time of calling a function.
2. Return Sequence–the set of actions at the time of returning from the function call.

For both of the above-mentioned sequences, some actions are performed by the *Caller* of the function and the other functions by the *Callee*. Now, we will discuss the set of responsibilities of caller and callee for both calling and returning from a function.

1. *Calling Sequence: Caller:*

   - Evaluate actual parameters; place actuals where the callee wants them. The corresponding three address instruction used is:

     param t

   - Save machine state (current stack and/or frame pointers, return address) and transfer control to the callee. The corresponding instruction is:

     call p, n (n = number of actuals)

2. *Calling Sequence: Callee:*

   - Save registers, if necessary; update stack and frame pointers to accommodate m bytes of local storage. The instruction to be used is:

     enter m

3. *Return Sequence: Callee:*

   - Place return value x, if any, where the caller wants it; adjust stack/frame pointers (may be); jump to return address. Instruction to be used is:

     return x or return

4. *Return Sequence: Caller:*

   - Save the value returned by the callee, if any, into x. The instruction to be used is:

     retrieve x

Example 7.7 Consider the function call:

$$x = f(0, y + 1) - 1$$

The corresponding intermediate code is given by

```
t1 = y + 1
param t1
param 0
call f, 2
retrieve t2
t3 = t2 − 1
x = t3
```

Storage Allocation for Functions. Allocating the storage to the function creates problem because the first instruction in a function is:

enter n /* n = space for locals, temporaries */

However, the value of n is not known until the whole function has been processed. There can be two possible solutions to this problem, which are given next:

1. Generate final code into a list. *Backpatch* the appropriate instructions after processing the function body. The approach is similar to single-phase code generation for Boolean expressions and control flow statements as discussed earlier. The advantage of this scheme is the ability to do machine-dependent optimizations also. However, it may be slower and may require more memory during code generation.
2. The second strategy is to use a pair of *goto* statements to enable us to put the instruction *enter n* at the end of the body of the function. It has been shown in Fig. 7.9.

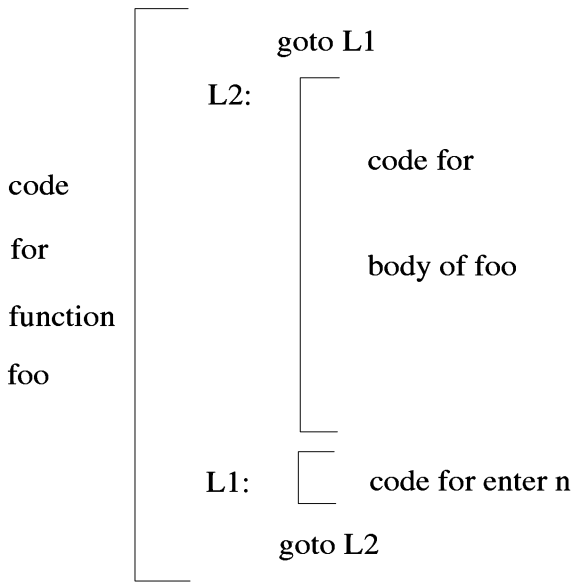

Figure 7.9: Code generation structure for functions.

7.7 CONCLUSION

In this chapter, we have seen translation strategies to generate intermediate code for various programming language constructs. Intermediate code generation, though not mandatory, helps in retargeting the compiler towards various different architectures. Selecting a good intermediate language itself is a formidable task, since it has to be closer to the target architecture. At the same time it should be able to represent the programming language constructs efficiently. We have seen that *three-address code* is one such representation. Syntax directed schemes can be utilised to generate the three-address code from the parse tree of the input program. The chapter discusses the conversion of almost all major programming language constructs to three-address codes.

EXERCISES

7.1 What is the role of intermediate code generation in overall compiler design?

7.2 Show the annotated parse tree and code generation process for the following arithmetic expressions

(a) $a + (b - c) * d$

(b) $-(a + b) * (c + d) + (a * b + c)$

7.3 Show the annotated parse tree and code generation process for the following Boolean expressions.

(a) *a and (b or c and not d)*

(b) *not (not (c and d) or a)*

7.4 Show the annotated parse tree and code generation process for the following expression involving arrays.

$$A[i, j] = B[C[i, j]] + C[i, j] + D[i * j]$$

7.5 Consider the following code fragment:

```
while a > b do
begin
     if x = y then c = a + b
     else d = a - b
     p = q + r
end
x = y + z
```

Show the annotated parse tree and the code generation process.

7.6 Write semantic actions for the following:

(a) *for-loop*

(b) *repeat-until loop*

7.7 Justify or contradict the statement, "backpatching is essential for constructs with forward branch targets".

7.8 Modify the semantic actions for array references to introduce bounds check dynamically.

7.9 Write the semantic actions to generate the three-address code for *case* statement of any language you are familiar with.

7.10 Write the semantic actions to generate the three-address code for *function call and return* statements of any language you are familiar with.

Chapter 8

TARGET CODE GENERATION

The final phase of the compiler is the *target code generation.* The intermediate code generator produces an intermediary code from which a simple template substitution could give the target code. However, generating an efficient code is always a challenge due to the heterogeneous structures of target machines. The major responsibility of this phase is to ensure the generation of a correct code. Secondly, the code generated should be optimal in terms of space and time required for its execution. However, the problem of generating an optimal code is undecidable. Thus, only heuristic algorithms can be applied to generate semi-optimal codes. The stage of target code generation may optionally be followed by optimization, which will be discussed later.

8.1 FACTORS AFFECTING CODE GENERATION

There are several factors affecting the code generation process. Some of them are discussed below. It may be noted that the peculiarity of target code requirement can also affect the process significantly.

1. ***The input.*** As noted earlier, the input comes in the form of an intermediate code. The proximity of the intermediate language to the target language controls the complexity of code generation. If for all operators, datatypes and addressing modes in the intermediate language straight-forward mappings to the target language exist, the code generation is simple template substitution, otherwise a good amount of translation effort may be needed for each of the non-supported features.
2. ***Target code structure.*** The structure of the target code controls the complexity of code generation. The final code may be an *absolute code* in which all the addresses are fixed, *relocatable*—allowing runtime relocation or *assembly language code*—requiring assembler to generate the executable version. *Absolute code generation* may be easy because of the fixed location of program variables and code. *Relocatable code* may or may not be supported by the underlying hardware. For example, many processors support *based addressing, relative addressing*, etc. to aid the relocation process. If relocation is not supported directly, then the compiler must insert code to ensure smooth relocation. *Assembly language code* can be generated to be converted into executable version by

an assembler. Features of the assembler may be utilized to make the code generation simpler. Code may also be generated targeted to virtual machine environment like *Java*. Thus, the challenges of code generation may be depicted to a great extent by the target architecture and language.

3. ***Instruction selection.*** Instruction selection plays a critical role. This is because the target machine normally has a large variety of instructions, and there may be several alternative approaches possible for performing a particular operation of the intermediate language. This is particularly true for *Complex Instruction Set Computers (CISC)*. On the other hand, the *Reduced Instruction Set Computers (RISC)* offer comparatively fewer alternatives to the compiler designer. Though it is restrictive in some sense, the code generation algorithm can exercise the options in a better fashion when the number of alternatives is less.
4. ***Register allocation.*** Assigning variables to CPU registers is a difficult task because of the fact that the allocation problem is *NP*-complete. We will later discuss some heuristic approaches based on *graph colouring* to solve the problem. The issue is further complicated by the underlying hardware differentiating registers as *data* or *address* register. For operations involving some data operand, it should be kept in one of the data registers, while for address operands, like pointers, the address registers should be used. Some operations need the source and destination operands to be kept in special pair of registers. All these architecture specific issues make the register allocation for target code generation a non-trivial task.
5. ***Evaluation order.*** Finding a good evaluation order may affect the target code performance. Some order may require fewer registers, some may ensure higher degree of accurracy for floating point operations. Determining the best order is again an *NP*-complete problem.

Next, we discuss the code generation strategies. To make the problem manageable, the entire input (intermediate code) to the module is first divided into a set of *basic blocks*. Code generation is carried out on a per block basis. This enables the optimizer to look more thoroughly into the individual blocks of code for local optimization. The final code optimizer may make a detailed analysis of information flow between these basic blocks to perform global optimization.

8.2 BASIC BLOCK

A *basic block* is a sequence of statements through which the control flows in a straight line fashion. Possibility of branching exists only at the end of the block. A program will, thus, consist of a set of basic blocks with control flow between them. Now, we look into the strategy to construct the basic blocks of a program.

Definition 8.1 *Leader: The following statements in a program are defined as leaders:*

1. *The first statement of a program.*
2. *The branch targets.*
3. *The statement immediately following a branch statement.*

Once we have determined the *leaders*, the basic blocks can very easily be identified. Each leader corresponds to a unique basic block starting with the leader and containing all statements upto the next leader, excluding the next leader. Thus, statements within a basic block constitutes a continuous flow of control. The control flow through the entire program can be captured by constructing the *flow graph* with basic blocks as nodes and edges defining the possible control transfers.

Example 8.1 Consider the following piece of code for searching an element x in an array $A[100]$.

```
begin
   location = -1
   i = 0
   while (i < 100) do
   begin
      if A[i] = x then location = i
      i = i + 1
   end
end
```

The corresponding three-address code is given by

1. location = -1
2. i = 0
3. if i < 100 goto 5
4. goto 13
5. $t_1 = 4 * i$
6. $t_2 = A[t_1]$
7. if t_2 = x goto 9
8. goto 10
9. location = i
10. $t_3 = i + 1$
11. $i = t_3$
12. goto 3
13. ...

The leaders are 1 (by rule 1); 3, 5, 9, 10 (by rule 2) and 4, 8, 13 (by rule 3). The basic blocks are as shown in Table 8.1. The flow graph is shown in Fig. 8.1.

8.2.1 Representation of Basic Blocks

Basic blocks are most conveniently represented using *Directed Acyclic Graph (DAG)*. The inherent advantage of such representation is to identify the common computational parts that may be present at multiple points within the block. The nodes of DAG correspond to the operations in the block. Each node has an associated *label*. The labels are assigned as per the following rules:

Table 8.1: Basic blocks

Block No.	Statements
B_1	1,2
B_2	3
B_3	4
B_4	5,6,7
B_5	8
B_6	9
B_7	10,11,12
B_8	13

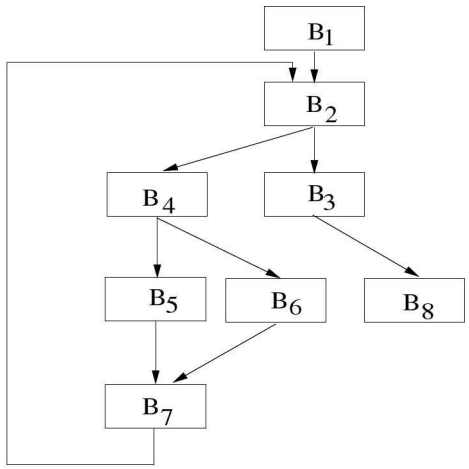

Figure 8.1: Flow graph for Example. 8.1

1. A leaf node represents an identifier. Hence, this identifier is used to label the node.
2. Interior nodes are labeled by operator symbol corresponding to the operation carried out by the node.

Example 8.2 Consider the following piece of code.

```
repeat {
    p = p + x[i]/y[i]
    i = i + 1
}
until i > 100
```

The block corresponding to the code consists of the following three-address code statements:

1: $t_1 = 4 * i$
2: $t_2 = x[t_1]$
3: $t_3 = 4 * i$

4: $t_4 = y[t_3]$
5: $t_5 = t_2/t_4$
6: $t_6 = p + t_5$
7: $p = t_6$
8: $t_7 = i + 1$
9: $i = t_7$
10: if $i > 100$ goto (3)

The DAG corresponding to this basic block is shown in Fig. 8.2.

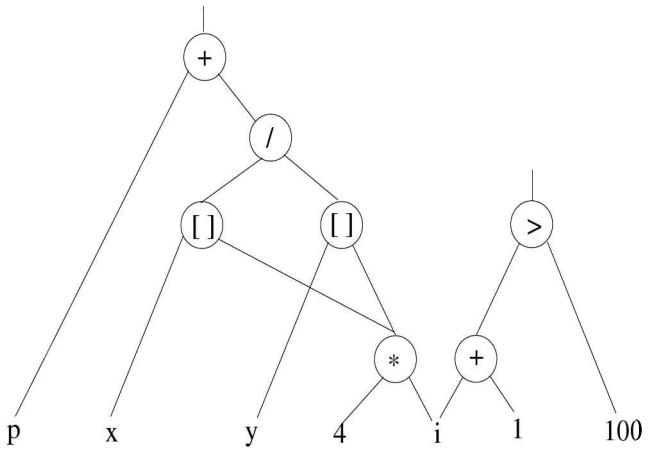

Figure 8.2: The DAG.

The DAG helps in the target code generation. In next section we will discuss the strategies for code generation. First, we will present a simplistic approach that can be used if the DAG is a tree, i.e., there is no common subexpression. We will then present a dynamic programming based approach for the case of a general DAG.

8.3 CODE GENERATION FOR TREES

Sethi & Ullman proposed an algorithm to generate target code for a tree-DAG using the least number of registers. The algorithm can be divided into two phases—

1. *labeling phase* that assigns a number to each tree node indicating the number of registers needed to evaluate the subtree rooted at this node and
2. *code generation phase* that performs the actual code generation.

To understand the process, we will study an example. Let us consider the expression $(A-B)+((C+D)+(E*F))$. The parse tree for it is shown in Fig. 8.3. Let the machine instructions are of the form

$op\ R_i,\ T$

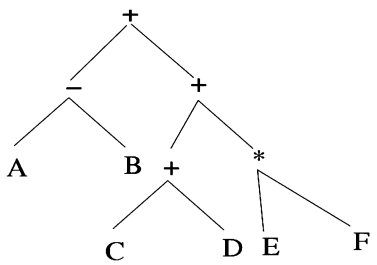

Figure 8.3: A tree DAG.

that has *op* as an operation (LOAD, ADD, SUB, MULT, etc.), R_i as a register, and T as a memory reference or a register. For a tree node T, let l and r be the number of registers required to evaluate its left and right subtrees respectively. The following cases may arise:

Case 1: $l > r$

In this case, we can evaluate the left subtree first and store it into one of the registers, say R_k. Now, we can evaluate the right subtree using the same registers except R_k. This is always possible, since $l > r$.

Case 2: $r > l$

This is similar to case 1, except the fact that now, the right subtree should be evaluated first.

Case 3: $l = r$

In this case, we need an extra register R_{l+1} to remember the result of the left subtree.

Case 4: T is a leaf node

If T is a tree leaf, then the number of registers to evaluate T is either 1 or 0, depending upon whether or not T is a left or a right subtree of its parent. This is because, the parent node "*op* R_i, T" can handle T directly from the memory, but R_i must be in a register. Thus, for a left subtree, the label is 1, whereas for a right subtree, it is 0.

Thus, the numbering algorithm starts from the tree leaves and assigns 1 or 0 as in Case 4. Then, for each node with the children labeled l and r, if $l = r$, then the node is labeled as $l + 1$, else the node is labeled as *max(l,r)*. The labels for the example are shown in Fig 8.4.

Next, we will elaborate the code generation phase. We use a stack of available registers, which initially contains all the registers in some order. The recursive code generation algorithm is as follows:

Algorithm Generate
Input: The tree rooted at T
Output: Target code for T
{
    if T is a leaf node and label(T) = 1 **then**
        output("load top(), T")
    if T is an internal node with children *left* and *right* **then** {
        if label(*right*) = 0 **then** {Generate(*left*); output("*op* top(), *right*")}

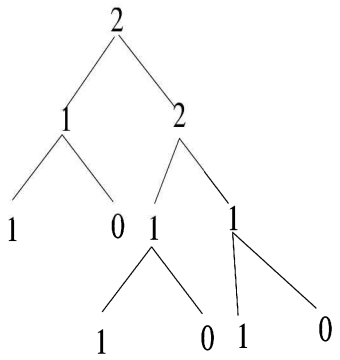

Figure 8.4: The labeled tree

```
        else if (label(left) ≥ label(right)) then {
            Generate(left)
            R = pop()
            Generate(right)
            output("op R, top()")
            push(R)
        }
        else        /* label(l) < label(r) */
        {
            swap the top two elements of the stack
            Generate(right)
            R = pop()
            Generate(left)
            output("op top(), R")
            push(R)
            swap the top two elements of the stack
        }
    }
}
```

For example, let us assume that only two registers R_1 and R_2 are available to evaluate the expression in the example under consider. Then, the code produced by the algorithm is as follows:

```
LOAD R2, C
ADD R2, D
LOAD R1, E
MULT R1, F
ADD R2, R1
LOAD R1, A
SUB R1, B
ADD R1, R2
```

The algorithm has several limitations. Some of them are noted below.

1. The number of registers needed is no greater than the number of available registers. Otherwise, we need to spill the intermediate result of a node to memory. This has been discussed in next section.
2. The algorithm does not use the commutativity and associativity properties of operators in an expression.

8.4 REGISTER ALLOCATION

Register allocation plays a vital role in determining the execution speed of a program. In a typical memory hierarchy, the registers constitute the storage closest to the processor. This is because the registers are on the same chip as the processor. It can be accessed directly by the processor within a single clock cycle. The next possible storage is the cache memory, typically requiring about three clock cycles to access. On the other hand, the main memory needs 20-100 cycles. However, the storage capacity is very small for registers, at most 256-8000 bytes. Cache is of size 256 KB to 1 MB, and main memory 32 MB to 1 GB.

It is a difficult task to manage this memory hierarchy. It needs co-ordination between various elements. Typically, a program is written considering only two kinds of storage–main memory and disk. The main memory holds the variables and constants of the program, whereas, the disk is used whenever a program performs a *read* or *write* operation. To reduce the main memory access time, the memory management unit decides on a policy to keep the relevant portions of the memory that are frequently accessed into cache. The portion of memory to be kept in cache is determined by studying the program behaviour. The current trend shows that the processor speed improves faster than memory or disk speed. Another degree of speed-up can be achieved by properly selecting the program variables and trying to put them into CPU registers, as this will minimize the access time to the least possible. Apart from program variables, a number of temporaries are generated by the compiler. Thus, the compiler can perform the task of allocating all these variables judiciously to the registers. To simplify the translation scheme, the intermediate code generator uses too many temporaries. The register allocation problem is to rewrite the intermediate code to use fewer temporaries than the machine registers. If there are not enough registers in the machine, we have to choose registers to "spill" to memory, so that these are made free to allot more variables to them. However, the program behaviour should not be changed.

The problem of register allocation is as old as intermediate code generation. Register allocation was first used in the original FORTRAN compiler in the 1950s. It used very crude algorithms for the same. In 1980, *Chaitin* invented a register allocation scheme based on *graph colouring*. The scheme consists of first constructing a *register-interference graph* and then, using a colouring approach to do the allocation. The following example shows how proper register allocation can reduce the number of registers needed.

Example 8.3 Consider the following piece of program:

$$\begin{aligned} \texttt{a} &= \texttt{c} + \texttt{d} \\ \texttt{e} &= \texttt{a} + \texttt{b} \\ \texttt{f} &= \texttt{e} - 1 \end{aligned}$$

with the assumption that a and e are not required in the later part of the program. Thus, the temporary a can be reused after the statement $e = a + b$. Similarly, the temporary e can be

reused after $f = e - 1$. Hence, all the three variables a, e and f can be allotted to a single register, say, r_1 giving the code:

$$\begin{aligned} \mathtt{r_1} &= \mathtt{c} + \mathtt{d} \\ \mathtt{r_1} &= \mathtt{r_1} + \mathtt{b} \\ \mathtt{f} &= \mathtt{r_1} - \mathtt{1} \end{aligned}$$

In the above example, we could reuse r_1 to store e because the variable a *died* after the computation of $a + b$. As a general rule, a dead temporary can always be reused. To state more formally, we can say that

Two variables v_1 and v_2 can share the same register if at any point in the program atmost one of v_1 or v_2 is alive.

8.4.1 Register Interference Graph Construction

This part of register allocation algorithm constructs the register interference graph after computing the set of *live variables* at each point of the program. Here, by *point* we refer to the positions just after each statement. A technique for the same will be discussed in the chapter on *Optimization.* For the time being, we follow the definition that a variable x is live at a point if it has been assigned some value by the statement or it was assigned some value earlier and being used by some other statement in some execution sequence of the program. Fig. 8.5 shows a flowgraph with live variables computed. We assume that the live variable at both E_1 and E_2 is $\{b\}$. After the first statement $a = b + c$ has been executed, the set of live variables is $\{a, c, f\}$. The statement computes a, the value of which is used in the statement $d = -a$ before any further assignment to a. Similarly, the values of c and f which reached the point through the backedge, are also live as they have been used subsequently in the statements $e = d + f$ and $b = f + c$. Similarly, the live variables are computed for each point of the program. Two

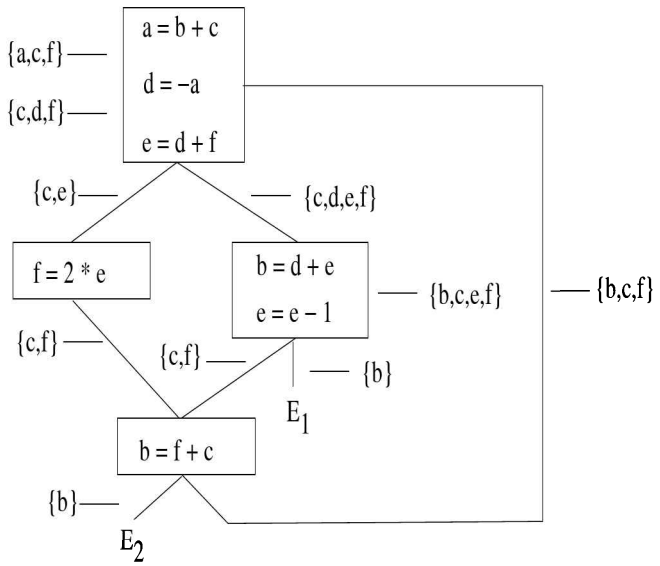

Figure 8.5: Flow graph with live variables.

variables that are live simultaneously cannot be allocated the same register. From the *liveness* information for each variable, we construct the undirected *register interference graph* as follows:

- Each node corresponds to a unique variable.
- An edge exists between two nodes if and only if the corresponding variables are *live* simultaneously at some point in the program.

Thus, two variables can be assigned the same register, only if there is no edge between the nodes corresponding to them.

Example 8.4 The register interference graph for the flow graph in Fig. 8.5 has been shown in Fig. 8.6. Here, the variables b and c cannot be allotted the same register since there is an edge between the nodes corresponding to them. Similarly, the variables f and d cannot be allotted the same register. However the variables a and b can be allotted the same register.

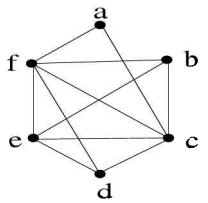

Figure 8.6: Register interference graph.

Once the register interference graph has been formulated, the register allocation is done using the *graph colouring* approach. To start with, we present the statement of the graph colouring problem.

Definition 8.2 Graph Colouring: *A colouring of a graph is an assignment of colours to its nodes, such that the nodes connected by an edge have different colours. A graph is k-colourable if it has a colouring with k colours.*

The register allocation problem can be formulated as graph colouring by the following analogies:

1. The colours correspond to registers. We need to assign colours (registers) to graph nodes (variables).
2. Let k = number of machine registers.
3. If the interference graph is k-colourable, then there is a register assignment to the variables that use no more than k registers.

Example 8.5 The interference graph shown in Fig. 8.6 is 4-colourable, and there does not exist any colouring with less than four colours. The assignment is shown in Fig. 8.7(a). The modified code after register allocation is given in Fig. 8.7(b). However, the computation of colouring for a given interference graph is an *NP*-hard problem. Moreover, since the maximum number of possible colours is fixed—equal to the number of *user registers* available for the processor, colouring might not exist.

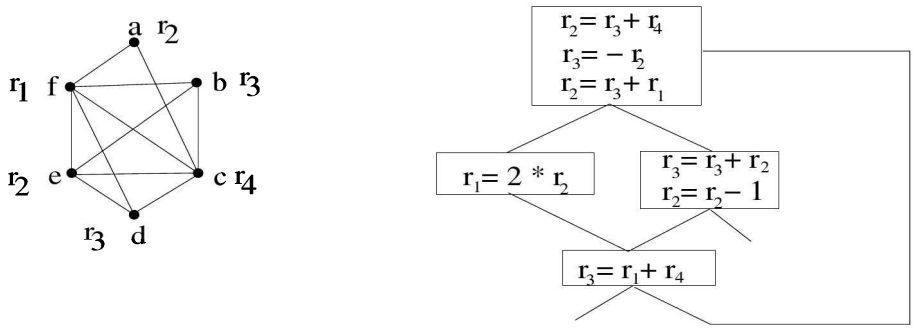

Figure 8.7: (a) Register allocation, (b) Modified code.

Now, we will see a heuristic approach to perform the colouring. The case of non-existence of colouring is discussed later using a special technique called *spilling.*

Graph Colouring Heuristic. The following is a heuristic that usually works well and has been used in many compilers.

Algorithm graph_colour
Input: An interference graph
Number of available colours k (say)
Output: A colouring for the graph nodes
Assumption: The graph is k-colourable
(1) Pick a node t with fewer than k neighbours
(2) Push t onto stack and remove it from the graph along with all adjoining edges
(3) Repeat steps 1 and 2 until no node is left
(4) While stack is not empty do
Pop a node (say x)
Colour x with a colour different from those assigned to already coloured neighbours of x
End.

The algorithm works since removing a node in step 1 does not modify the colouring property of the graph. If a node t with $n(< k)$ neighbours is selected to be removed and the resulting graph is k-colourable, then the original graph is also k-colourable. This is because after assigning n different colours to the neighbours of t, still $(k-n)$ colours are left for t. Moreover, $k - n > 0$ since $k > n$.

Example 8.6 Consider the interference graph shown in Fig. 8.8(a) and let the number of colours available be $k = 4$. As the first step of the algorithm, the nodes a and d have less than four neighbours. Thus, these nodes are removed from the graph and pushed onto the stack. The new graph along with stack content is shown in Fig. 8.8(b). The stack contains d at the top followed by a. In Fig. 8.8(b), all nodes have less than four neighbours. Thus, all of them are removed and pushed onto the stack giving the stack content as f, e, b, c, d, a. Now, step 4 of the algorithm starts. Node f is popped out and is assigned colour r_1. Next, e is popped out. Since its neighbour f is already coloured with r_1, e is assigned r_2. Similarly, b and c

are assigned the colours r_3 and r_4 respectively. Then, d is popped out. Since its neighbours have been allotted colours r_1, r_2 and r_4, the colour r_3 is free and is thus, allotted to d also. Similarly, r_2 is assigned to a. The final allotment is shown in Fig. 8.8(c).

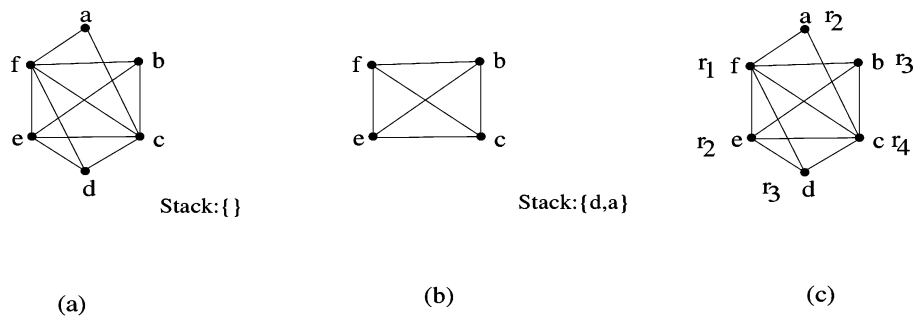

Figure 8.8: (a) Interference graph, (b) After removing a, d, (c) Final allocation.

Now, we consider the case where the graph is not k-colourable for a given value of k. Let us take the example of the graph shown in Fig. 8.8(a). If we try to find a 3-colour here the algorithm removes a and then, gets stuck, since all the nodes now have three or more neighbours. The resulting graph has been shown in Fig. 8.9.

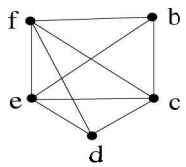

Figure 8.9: Failure in 3-colouring.

In this situation, we have to choose a node for *spilling* into the memory. In other words, the corresponding variable will always reside in a memory location allotted. For example, let us assume that the node f has been chosen for spilling. Now, f alongwith the edges incident on it is removed from the interference graph. The resulting graph has been shown in Fig. 8.10. The simplification now proceeds to remove the nodes b, d, e and c, since each of them has less

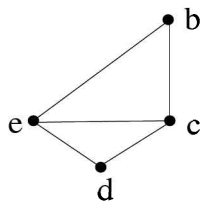

Figure 8.10: Graph after removing node f.

than three neighbours. On the assignment phase (Step 4 of the algorithm), at the time of assigning a colour to the node f, we check whether or not the neighbours of f (that is, b, c, d, and e) use less than three colours. This *Optimistic Colouring* may then allot the remaining

colour to node f. However, is not the case in this example. Hence, a memory location, say f_m is allotted to f.

Before each operation that uses f, the statement

$$f = \text{load } f_m$$

is inserted.

Similarly, after each operation that defines f, the following statement is inserted:

$$\text{store } f, f_m$$

The liveness information also needs to be recomputed. The spilled variable f is live only

1. between a $f =$ load f_m and the next instruction and
2. between a store f, f_m and the preceding instruction.

This spilling reduces the live range of f. This, in turn, simplifies the interference graph. The modified code along with liveness values is shown in Fig. 8.11(a). The new interference graph is also shown in Fig. 8.11(b). The graph is now three colourable.

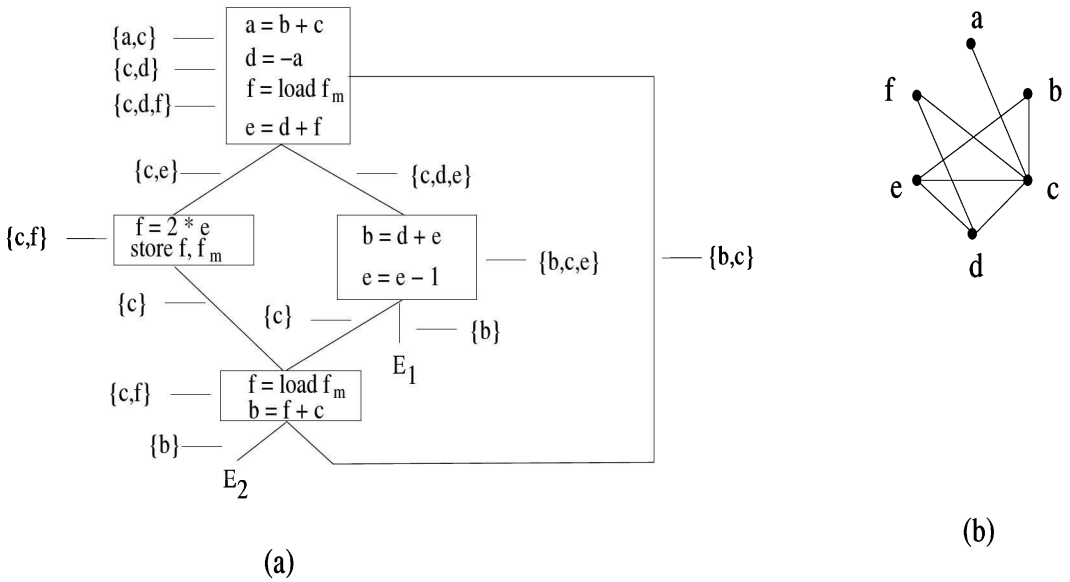

Figure 8.11: (a) Modified code after spilling f, (b) New interference graph.

It may be noted that additional spilling might be required before a colouring is found. Thus, selection of variable to spill may play a crucial role in the final allocation. The following are few possible heuristics to choose the variable to spill:

1. Spill the variable with maximum conflicts.
2. Spill the variable with few definitions and uses.
3. Avoid spilling in inner loops, since the associated load and store statements will get executed for many times.

8.5 CACHE MANAGEMENT

Cache is normally managed better by the programmers. However, some compilers do some amount of cache management. Let us consider the following program:

```
for j = 1 to 10 do
      for i = 1 to 1000 do
            a[i] = a[i] * b[i]
```

Here each element of a is multiplied ten times by the corresponding elements of b. If the cache size be such that it can hold 100 elements of a and b, then $a[1]$ to $a[100]$ and $b[1]$ to $b[100]$ will get loaded into the cache at the execution of $a[1] = a[1] * b[1]$. However, when this computation is done in the next iteration of j, the elements will not be available in the cache. A simple loop interchange by the compiler will improve the execution significantly:

```
for i = 1 to 1000 do
      for j = 1 to 10 do
            a[i] = a[i] * b[i]
```

Here, once the iterations of j are over for a particular value of i, this $a[i]$ and $b[i]$ values will no more be needed, thus improving the performance significantly.

8.6 CODE GENERATION USING DYNAMIC PROGRAMMING

In this section, we will illustrate another method of optimal code generation. This approach uses dynamic programming strategy for the same. The technique designed by *Aho* and *Johnson* in 1976 has been applied in a number of compilers. It is applicable to a broader class of register machines with complex instruction sets. In general, the scheme works for any machine with r interchangeable registers $R_0, R_1, \ldots, R_{r-1}$ and instructions of the form $R_i = E$, where E is any expression involving register and/or memory. In case of register operands in E, one of them must be R_i.

The dynamic programming approach divides a problem into subproblems. The optimal solution of a subproblem is found by dividing it further, until it becomes trivial. Then, the original solution of the problem is obtained from the optimal solution of the subproblems. The dynamic programming approach for code generation for an expression tree T works in three different phases, which are discussed below.

Phase I: The first phase computes a cost array C for each node of the tree in a bottom-up fashion. For a particular node, the i-th component of the cost array, $C[i]$, contains the optimal cost of computing the subtree rooted at this node into a register assuming the i registers are available for the computation.

Phase II: In the second phase, the tree T is traversed using the cost vectors in order to determine the subtrees of T which must be computed into memory. It may be noted that for each node, $C[0]$ holds the cost of computing the subexpression into memory.

Phase III: In the third phase, the tree is traversed using cost vectors and the corresponding instructions. The final target code is generated in the process.

Example 8.7 Let us consider the following expression for which we need to generate the code for a machine with two user registers R_0 and R_1.

$$(a * b) + (c - (d + e))$$

Let the instructions available be

$$\begin{aligned} R_i &= M_j \\ R_i &= R_i op\ R_j \\ R_i &= R_i op\ M_j \\ R_i &= R_j \\ M_i &= R_j \end{aligned}$$

The DAG for the expression alongwith the cost vector C for individual nodes is shown in Fig. 8.12.

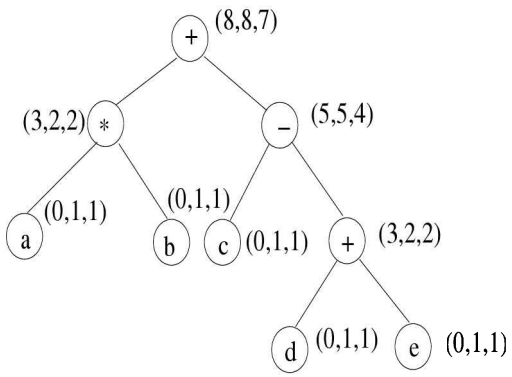

Figure 8.12: DAG with cost vectors.

Phase I: This phase computes cost for each node in a bottom-up fashion. For a leaf node, say a, the cost array $C = (0, 1, 1)$. Let us elaborate how the cost array is computed. Since a is already in memory, the cost of evaluating a into memory is zero, thus giving $C[0] = 0$. Further since here we need to load a from memory to the register $C[1] = 1$. Similarly, $C[2] = 1$. Hence, all the leaf nodes have cost array $C = (0, 1, 1)$.

In order to compute $C[0]$ for node '*', we have to evaluate a into a register, b into memory, compute $a * b$ using the instruction $R_i = R_i op\ M_j$, and finally, store the result into memory. Thus, requires 1 (to evaluate a), + 0 (to evaluate b) + 1 (to do operation) + 1 (to store result) = 3 memory/register operations. On the other hand, if the final result can reside in a register, we need not store it back. Thus, $C[1] = 2$ and $C[2] = 2$. Same is the case for '+' performing $d + e$.

For node '-', cost $C[2]$ is computed by evaluating left subtree into a register (cost 1), evaluating right subtree into a register (cost 2) and then, doing operation (cost 1) = 4. Similarly, computation of $C[1]$ involves evaluating left subtree into a register (cost 1), evaluating right subtree into memory (cost 3) and then doing operation (cost 1) = 5. The cost $C[0]$ is evaluated by considering the case of evaluating the expression with two registers available and then, storing the result into memory = (4 + 1) = 5.

For the root node, $C[1]$ = cost of evaluating left subtree into register + cost of evaluating right subtree into memory + 1 = 2 + 5 + 1 = 8. However, for $C[2]$, the following cases are available.

1. Compute left subtree with 2 registers available into R_0, compute right subtree with 1 register available in R_1 and then, perform $R_0 = R_0 + R_1$. Thus, total cost = 2 + 5 + 1 = 8.
2. Compute right subtree with 2 registers available into R_1, compute left subtree with 1 register available in R_0 and then, perform $R_0 = R_0 + R_1$. Thus, total cost = 4 + 2 + 1 = 7.
3. Compute right subtree into memory, compute left subtree with 2 registers available in R_0, and then, perform $R_0 = R_0 + M$. Thus, total cost = 2 + 5 + 1 = 8.

Minimum cost is 7, thus $C[2] = 7$. $C[0]$ is obtained by evaluating root with two registers available and storing the result into memory = 7 + 1 = 8.
Phase II: This phase identifies the memory operations. We see that the root node '+' must be evaluated with 2 registers that come when left subtree is evaluated with 1 register while right subtree with 2 registers. For '*' to be evaluated with 1 register available, a must be loaded into the register while b resides in memory. Similarly, e must reside in memory.
Phase III: In this phase, the code is generated using the traversal as follows:

$R_0 = c$
$R_1 = d$
$R_1 = R_1 + e$
$R_0 = R_0 - R_1$
$R_1 = a$
$R_1 = R_1 * b$
$R_1 = R_1 + R_0$

8.7 CONCLUSION

In this chapter, we have seen a few techniques to generate code targeted to different architectures. There are various factors affecting the code generation process, namely, *input, target machine, registers* available and so on. Normally, the intermediate language program is broken down into a set of straight line segments having a single entry and exit, called *basic blocks*. Codes are generated for each of these basic blocks separately. The code generation process is much simpler if the basic block needs a graph to represent it, the optimum code generation is a difficult task. Register allocation is another important issue, which is solved using heuristic approaches like *graph colouring* with *spill code* to manage large number of variables. Another good approach for code generation is to use *dynamic programming*.

EXERCISES

8.1 Discuss the factors affecting target code generation.

8.2 Why is it necessary to break the intermediate code into basic blocks? Write down the intermediate code for the following code fragment, and show the basic blocks and the control flow within them.

```
begin
     sum = 0
     term = 1
     while term ≤ 100 do
          sum = sum + term
          term = term + 1
     end
end
```

8.3 Consider the following program:

```
a = 1
b = 10
c = 20
d = a + b
e = c + d
f = c + e
b = c + e
e = b + f
d = 5 + e
return d+f
```

(a) What is the fewest number of registers that are needed for the program without *spilling*? Justify your answer by showing the interference graph and a colouring of the graph. Also, show the program after register allocation.

(b) Assuming that only two registers are available, show the code after doing the necessary *spilling*.

8.4 Consider the following sequence of statements:

```
x = y * z
w = p + y
y = y * z
p = w − x
```

(a) Construct the corresponding DAG.

(b) Perform code generation assuming two registers are available.

(c) Perform code generation assuming only one register is available.

8.5 Consider the following statement.

$$x = a/(b + c) - d * (e + f)$$

Assume that two registers are available. Perform code generation using

(a) tree and

(b) dynamic programming.

8.6 For the statement in above problem, perform register allocation assuming that

(a) only one register is available

(b) two registers are available

(c) three registers are available

Chapter 9

Code Optimization

The final stage of a compiler is code optimization. Though this is an optional stage, most of the compilers perform some amount of optimization. With the memory chips becoming cheaper and cheaper, the major motivation behind code optimization has been shifted from compact code generation to production of target programs with high execution efficiency. However, any optimization attempted by the compiler must satisfy the following conditions.

1. Semantic equivalence with the source program be maintained.
2. The algorithm should not be modified in any sense.

As we go through the chapter, we will notice that some optimizations are simple and can be carried out without much difficulty, whereas, some other optimizations may need a thorough analysis of the program. Naturally, for the second case, compilation time may be much higher. Generally, the compilers come up with various optimization levels, and the user may choose various degrees of optimizations.

9.1 NEED OF OPTIMIZATION

With the availability of more and more source language options and taking into view the volume of code for newer applications, the following are the major issues establishing the need of optimization.

1. Programming languages permit many flexibilities, often leading to inefficient coding by the programmer. For example, the statement $a = a + 1$, where a is of real type, requires a type conversion of 1 to 1.0 at run-time. An optimization of the statement may modify it to $a = a + 1.0$ thus avoiding the run-time conversion.
2. Optimization allows programmers to write code in straight forward manner, expressing their intention clearly and allowing the compiler to make choices about implementation details that lead to efficient code.

It may be noted that code optimization by a compiler rarely results in code that is perfectly optimal by any measure, only there may be lot of improvement as compared to the original program written by the programmer. However, code optimization may be very effective. For

example, a program executes three times faster and occupies 25% less storage when optimized by the *IBM/360 Fortran H* compiler of mid-sixties. The improvement may be more pronounced in modern contemporary compilers because of the following factors:

1. More effective optimization techniques are available.
2. At the time of software development, less emphasis is being put on execution efficiency than on other attributes of a program like structure, maintainability, reusability, etc. Code sharing and re-use lead to *black box* view of program, which further emphasizes execution efficiency.
3. Exploitation of advanced architectural features like instructional pipelining requires *smart* code generation, which is only possible in an optimizing compiler.

Can a programmer out-perform an optimizer. Answer to this question is both *Yes* and *No*. The reasons behind the answer *Yes* are as follows:

1. The programmer may choose a better algorithm for the problem being solved by the program.
2. Knowing more about the definitions and uses of data items in the program, for example, the programmer may know the relative probabilities of branches being taken.
3. An optimizer has to ensure correctness of the optimized program under all conditions. Thus, it has to be conservative. For example, if the variable *age* stores the age of a human being, then it is most probably restricted to 150. If a program defines the variable as a full integer, then the compiler has to allocate enough space for it and use instructions for integer manipulation. However, since *age* requires much lesser number of bits, a possible optimization is to reduce the size of *age* and use instructions that can manipulate shorter sized data efficiently.

The logic behind the *No* answer is as follows:

1. It takes too much time to perform some optimizations by hand.
2. Certain machine level details are beyond the control of a programmer, namely, instructions and addressing modes supported by the target architecture. For example, in the RISC CPU architecture, compiler optimization is the key for obtaining an optimal code. The RISC instruction set is so compact that it is hard to schedule or combine small instructions manually to get efficient results.

9.2 PROBLEMS IN OPTIMIZING COMPILER DESIGN

Designing an optimizing compiler is further complicated by the following issues:

1. Usually, an optimizing compiler takes the intermediate representation of a program code and replaces it with a better version. In the process, the high-level redundancy in the source program, such as an inefficient algorithm, remains unchanged.
2. Modern third-party compilers usually have to support several objectives, for example, different architectures, making it difficult to optimize well for all the cases.

3. A compiler typically deals with only a small part of an application at a time, at most a module and usually a procedure, the result being, it is unable to consider important contextual information.

Another set of tools known as *post pass optimizers* tries to overcome this problem by working on the *assembly language* level of the program, rather than the intermediate form. However, for this tool also, much of the information found in the original source is lost.

9.3 CLASSIFICATION OF OPTIMIZATION

The commonly used optimization techniques can be classified into various classes by considering the dimensions of classification.

Scope of optimization. The effect of a particular compiler optimization can be anywhere from a single statement to an entire program. In this direction, the optimization techniques can be classified as follows:

- **Peephole optimization:** Usually performed late in the compilation process, peephole optimizations examine atmost a few instructions transforming them into other less expensive ones. For example, turning a multiplication of variable x by two into an addition of x to itself.
- **Local optimizations:** These optimizations operate on a single *basic block.*
- **Loop optimizations:** These optimization act on a number of basic blocks which make up a loop. Loop optimizations are important because many programs spend a large percent of their time inside loops.
- **Global or intraprocedural optimizations:** This type of optimization acts on the complete control flow graph of a single procedure.
- **Interprocedural or whole-program optimization:** The most powerful of all in some ways, they optimize interactions between procedures. A simple example is taking a call to a function with constant parameters, copying the code of that function and substituting the constant parameters throughout. This type of analysis is often very limited in order to deal with the complexity of an object as large as a whole program.

Language independent vs. language dependent. Most of the high level languages share common programming constructions and abstractions – *decision* (if, switch, case), *looping* (for, while, repeat until, do while), *encapsulation* (structures, objects). Thus, similar optimization techniques can be used across languages. However, certain language features make some kinds of optimization easy or difficult. For instance, the existence of pointers in C, C++ makes certain optimization of array accesses difficult. Conversely, in some languages, functions are not permitted to have *side effects.* Therefore, if repeated calls to the same function with the same arguments are made, then the compiler can immediately infer that results need to be computed only once and the result referred to repeatedly.

Machine independent vs. machine dependent. A lot of optimizations that operate on abstract programming concepts (loops, objects, structures) are independent of the machine targetted by the compiler. But, many of the most effective optimizations are those that best exploit the features of target platform.

Example 9.1 Often, the machine dependent optimizations are local. As an example, consider the operation: to set a register to 0. The obvious way is to use the constant 0 with the instruction that sets a register value to a constant. A less obvious way is to XOR a register with itself. The compiler should know which instruction variant is suitable to use. On many RISC machines, both of the instructions would be equally appropriate, since both would be of the same length and would take the same time. On many microprocessors, such as Intel X86 family, it turns out that the XOR variant is shorter and may be faster (no need to decode an immediate operand nor use the internal *immediate operand register*).

In fact, machine independent optimizations can be both *local* or *global*. The cost-effectiveness of such transformations depends on their scope.

1. **Local optimization:** As noted earlier, the local optimizations are restricted to essentially sequential sections of the program code (*basic blocks*). This restricts the amount of analysis required. However, it also restricts the kinds of optimization possible (for example, loop optimization cannot be performed locally) and the associated gains.
2. **Global optimization:** It is applied on a larger section of a program, typically on a loop or procedure/function.

   Knuth (1971) reported speed up factors of

   - ≥ 1.4 due to local optimization
   - ≥ 2.7 due to global optimization

However, both local and global optimizations have their own importance because of the following factors:

1. Local optimization simplifies global optimization. Consider the elimination of redundant computations from a basic block shown below.

   ```
   a * b
   ...
   a * b      ⇐ assumed eliminated by local optimization
   ```

   Thus, global optimization needs to be considered only on the first occurrence of $a * b$ within a basic block.
2. Local optimization can be merged with the preparatory phase of global optimization.

9.4 FACTORS INFLUENCING OPTIMIZATION

There are several factors that influence the optimization process. In this section, we discuss some of these issues.

1. **The machine:** Many of the choices about which optimizations can and should be done depend on the characteristics of the target machine. Sometimes, it is possible to parameterize some of these machine dependent factors, so that a single piece of compiler code can be used to optimize different machines just by altering the machine description parameters. For example, the source code for *gcc* compiler can be supplied with the target machine type to generate code optimized for that machine.

2. **Architecture of the target CPU:** Following are the important properties of the CPU which influence code optimization.

   - **Number of CPU registers:** To a certain extent, more the registers, easier it is to optimize for performance. Local variables can be allocated the registers and not the stack. Temporary/immediate results can be left in registers without writing to and reading back from memory.
   - **RISC vs. CISC:** CISC instruction sets, often, have variable instruction lengths, and a larger number of possible instructions that can be used. Also, each instruction can take different amounts of time. RISC instruction sets attempt to limit the validity in each of these: instruction sets are of usually constant length, with few exceptions, there are usually fewer combinations of registers and memory operations, and the instruction issue rate (the number of instructions completed per time period, usually an integer multiple of the clock cycle) is usually constant in cases where memory latency is not a factor. There may be several ways of carrying out a certain task, with CISC usually offering more alternatives than RISC. Compilers have to know the relative costs among various instructions and choose the best instruction sequence.
   - **Pipelines:** A pipeline is essentially an *ALU* broken up into an assembly line. It allows use of parts of the ALU for different instructions by breaking up the execution of instructions into various stages: instruction decode, address decode, memory fetch, register fetch, compute, register store, etc. One instruction could be in the register store phase, while another could be in the register fetch stage. Pipeline conflict occurs when an instruction in one stage of pipeline depends on the result of another instruction ahead of it in the pipeline but not yet completed. A pipeline conflict can lead to pipeline stalls where the CPU wastes cycles waiting for the conflict to resolve. Compilers can *schedule* or reorder instructions so that the pipeline stalls occur less frequently.
   - **Number of functional units:** Some CPUs have several ALUs and FPUs. This allows them to execute multiple instructions simultaneously. There may be restrictions on pairing, i.e., the simultaneous execution of two or more instructions, and the functional unit executing an instruction. There are also issues similar to pipeline conflicts. Thus, the instructions have to be scheduled so that the various functional units are fed fully with instructions to execute.

3. **Machine architecure:** There are various issues regarding machine architecture that affects optimization.

   - **Cache size and type:** The cache size may vary from 256KB to 1MB, and also, the type may be fully-associative or k-way associative. Optimizations like *inlining* (that is, substituting calls to small procedure by the body of the procedure) and loop unrolling may increase the size of the generated code and reduce the code locality. The program may slow down drastically if an oft-run piece of code like inner loops in various algorithms, suddenly, cannot fit in the cache. Also, caches which are not fully associative have higher chances of cache collision even in an unfilled cache.

- **Cache/Memory transfer rates:** This gives the compiler an indication of the penalty for cache misses. This is used mainly in specialized applications.

9.5 THEMES BEHIND OPTIMIZATION TECHNIQUES

To a large extent, optimization techniques have the following themes, which sometimes conflict:

1. **Avoid redundancy:** If something has already been computed, it is generally better to store it and reuse it later, instead of recomputing it.
2. **Less code:** There is less work for the CPU, cache, and memory. So, it is likely to be faster.
3. **Straight line code, fewer jumps:** It is a less complicated code. The jumps interfere with prefetching of the instructions, thus slowing down the code.
4. **Code locality:** Pieces of code executed close together in time should be placed close together in memory. This increases locality of reference.
5. **Extract more information from code:** More the information the compiler has, the better it can optimize.

9.6 OPTIMIZING TRANSFORMATIONS

In this section, we elaborate various techniques for optimization which may be applied on the code to transform it, probably leading to better implementations. The approaches may be tried out for both local and global optimizations with the scope as found earlier. The major transformations of such kinds are:

1. Compile-time Evaluation
2. Common Sub-expression Elimination
3. Variable Propagation
4. Code Movement Optimization
5. Strength Reduction
6. Dead Code Elimination
7. Loop Optimization

Now, we will discuss each of these transformations.

9.6.1 Compile-time Evaluation

It deals with modifying the code such that the expressions whose values can be precomputed at the compilation time and used, perhaps repeatedly, in the execution of the program are identified. Then, the computations of such expressions are moved to compile-time, rather than during execution. There are two possible avenues through which it can be carried out, namely, *folding* and *constant propagation.*

Folding. Folding corresponds to the evaluation of an expression with constant operands at compilation time. As a result, the expression is replaced by a single value, hence, the term *folding.*

Example 9.2 Consider the expression to compute the area of a circle that needs the value of the constant π, which is approximated by the expression 22/7. The programmers often write the corresponding code as

$$\texttt{area} = (22.0/7.0) * \texttt{r} * *2$$

Here, the evaluation of 22.0/7.0 can be performed during compilation to the machine accurracy and used in the execution.

Typical application of folding occurs in address calculation for array references, where the products of many constants are *folded* into single constant value.

Constant Propagation. Constant propagation corresponds to replacing a variable by the constant which has been assigned to it earlier. Though constant propagation alone does not lead to any optimization, it may help in other optimizations. For example, the constant propagation may help in folding an expression.

Example 9.3 Consider the following piece of code:

```
a = 3.1
...
x = a * 2.5
```

The constant propagation can propagate the value 3.1 of a assigned at the first statement to the right-hand side of the statement $x = a * 2.5$ to modify it to the statement $x = 3.1 * 2.5$, which may now be folded and evaluated at compile-time.

However, the constant propagation is possible only if there are no intervening statements that may modify the value of the variable directly or indirectly. A direct modification is through an assignment statement whose left-hand side has the variable under consideration. Indirect assignment may result from usage of pointers and function calls with the variable as a modifiable parameter. Moreover, if the intervening code between the assignment of the variable and its usage is not linear, that is, it contains branchings, a variable should be assigned the same constant value along all of the paths reaching its use.

Example 9.4 Consider the program flow graph shown in Fig. 9.1. Here, constant propagation and folding is possible for $x + 17$ of block 9. However, it is not possible for $a + 17$ of block 10 because of the presence of another assignment $a = b$ at block 4.

9.6.2 Common Sub-expression Elimination

A major source of optimization is the identification of common subexpressions that are present in various expressions within the program. Once identified, the common subexpression may be evaluated just once to a new temporary variable, if needed, and may replace all of the occurrences of the subexpression by the variable. For example, the code

```
a = b * c
...
x = b * c + 5
```

has the subexpression $b * c$ common between the two statements and thus, can be reduced to single computation as follows.

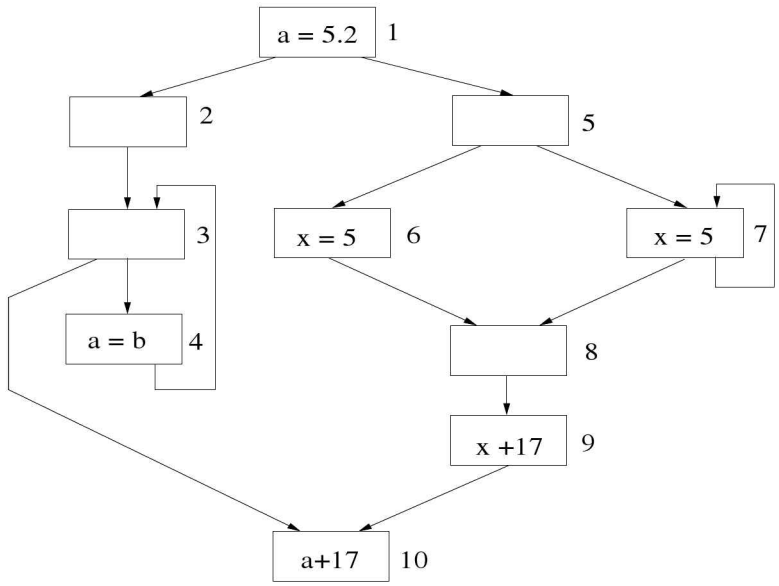

Figure 9.1: Constant propagation.

```
temp = b * c
a = temp
...
x = temp + 5
```

Identification of common subexpressions is not a very simple task as explained in the following example. Typically, the search is made only for *lexically equivalent* expressions, that is, the expressions consisting of the same identifiers and operators, and appearing in the same order. The expressions must also be evaluated to *identical* values during any course of execution of the program. This ensures the preservation of *semantic equivalence* between the original and transformed code.

Example 9.5 Consider the program flow graph shown in Fig. 9.2. The following observations can be made:

1. $a * b$ of block 10 is a common subexpression.
2. $x + y$ of block 8 is not a common subexpression to $x + y$ in block 5. This is because in an execution sequence that passes the control from block 5 to block 8 via block 6, the value of x is modified. Thus, the two expressions will not be evaluated to identical values.
3. $c * b$ in block 4 is also a common subexpression to $a * b$ in blocks 1 and 10. The assignment $c = a$ in block 2 ensures this. However, it is very difficult to catch such situations. Here, we have a *nonlexical equivalence* between the expressions.

9.6.3 Variable Propagation

Similar to constant propagation, variable propagation also suggests replacing a variable by another variable holding identical value.

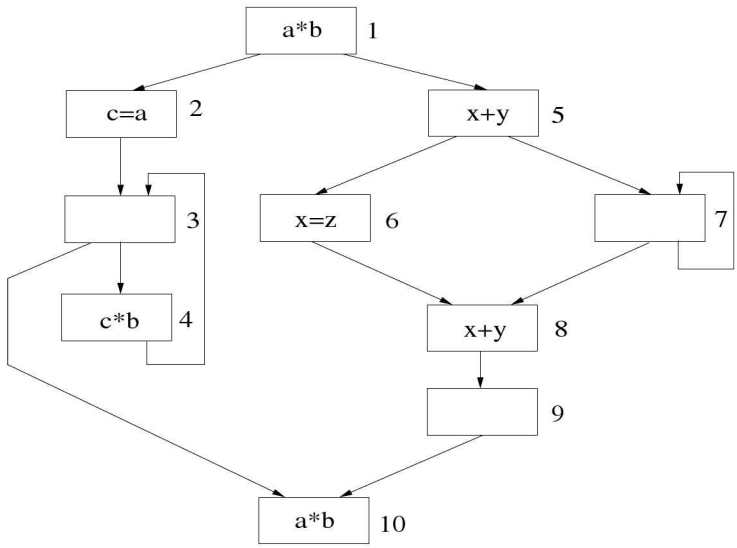

Figure 9.2: Common subexpression.

For example, consider the following piece of code:

```
c = d
...
...
z = c + e
x = d + e − 10.5
```

Here, the first statement ensures that the values of c and d are same. In the subsequent code, if none of them is modified, the statement $z = c + e$ can be modified to $z = d + e$ by propagating the variable d to it. This creates the possibility of identifying $d + e$ as a common subexpression in the last two statements, and possibly evaluated only once.

Variable propagation should be carried out with the restriction that along all the paths reaching to its use, a variable should be assigned the value of the same right-hand side variable, and neither variable should be modified following such assignment. The following example illustrates the case further.

Example 9.6 Consider the flow-graph shown in Fig. 9.3. Here, the variable a in block 10 cannot be replaced by c following the assignment $a = c$ in block 1. This is because, block 4 modifies the value of c, and thus, the modified expression in block 10 will not be equivalent to the original expression. However, assuming that neither x nor z has been modified in the blocks 6 and 7, the expression $x + y$ in block 8 can be modified to $z + y$ by propagation of z.

9.6.4 Code Movement Optimization

Code movement refers to moving the code from one part of the program to another, ensuring that the resulting program is equivalent to the original one. Code movement is performed so as to

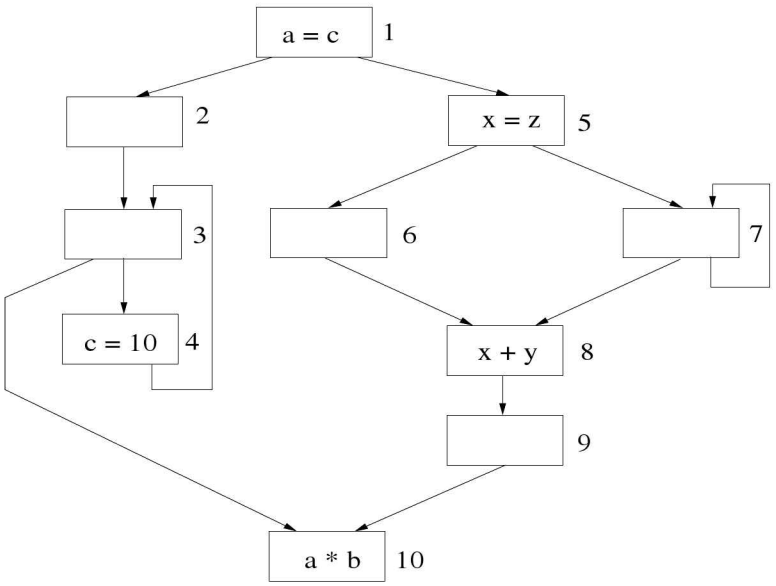

Figure 9.3: Variable propagation.

- **Reduce size of the program,** also known as *code space reduction* and
- **Reduce execution frequency of the code subjected to movement** – *Execution frequency reduction*

1. **Code space reduction:** It is similar to common subexpression elimination. However, here, the objective is not to reduce the execution frequency of the common subexpression. Rather, it reduces the code size by generating the code for common subexpression only once. This is, often, performed by a technique called *code hoisting*, which brings the code nearer to the start point. For example, consider the following piece of code:

```
if a < b then
    z = x**2
    ...
else
    y = x**2 + 19
```

Here, the code to compute $x**2$ is repeated in both the *then* and *else*-part of the program. It can be modified to compute $x**2$ only once before the *if*-statement. Thus, though the execution frequency for the subexpression is not modified, the code is generated for once only.

```
temp = x**2
if a < b then
    z = temp
    ...
else
    y = temp + 19
```

2. **Execution frequency reduction:** This technique identifies the common codes to be evaluated at various places in the program and tries to move them to the place, so as to reduce the frequency of their execution. There are two avenues of doing such transformations, which are given below:

   (a) *Code hoisting:* As noted earlier, code hoisting refers to promoting the code to some earlier part of the program. It can be used to reduce execution frequency of an expression if the expression is *partially available*. That is, it is available along at least one path reaching the evaluation of the expression.

   Example 9.7 Consider the code segment

```
if a < b then
    z = x * 2
    ...
else
    y = 19
g = x * 2
```

   In this case, during execution of the code segment, the expression $x * 2$ is evaluated twice for the condition $a < b$. The computation of $x * 2$ can be hoisted to the *else*-part also to make the evaluation of the expression always one.

```
if a < b then
    temp = x * 2
    z = temp
    ...
else
    y = 19
    temp = x * 2
g = temp
```

   Safety of code movement. Movement of an expression e from a block b_i to another block b_j is said to be *safe* if it does not introduce a new occurrence of e along any path in the program. *Unsafe* code movement may change the meaning of the program altogether. It may lead to surprising execution conditions, for example, overflow.

   Example 9.8 Consider the code segment

```
...
if a < b then
    z = x * 2
else
    y = 19
```

   The following is an unsafe hoisting of the code as the expression $x*2$ gets introduced into the *else* part also.

```
    temp = x * 2
if a < b then
```

```
        z = temp
    else
        y = 19
```

Unsafe movement of code should be avoided.

(b) *Loop optimization:* It involves identifying such statements within the body of the loop whose value do not depend on the iteration. Such computations can be taken out of the loop to reduce their frequency of execution. We will discuss it in detail while handling techniques for *loop optimization.*

9.6.5 Strength Reduction

Strength reduction corresponds to the replacement of an operator requiring high execution time by an equivalent operator requiring less processor time. For example, it may be possible to replace '*' by '+'. Multiplication/division by the powers of 2 can be modified to be left/right shift of integer data.

Example 9.9 Consider the code segment

```
for i = 1 to 10 do
    ...
    x = i * 5
    ...
end
```

The code segment can be modified as follows to use addition instead of multiplication.

```
temp = 5
for i = 1 to 10 do
    ...
    x = temp
    ...
    temp = temp + 5
end
```

A typical case of strength reduction occurs in address calculation for array references. For example, the code segment,

```
for i=1 to 50 do
    a[i] = ...
end
```

If each element of the array requires 4 bytes, then address of $a[i]$ is equal to ((address of $a[0]$) $+i*4$). Now, strength reduction can be applied to optimize the code as shown in the previous example.

Strength reduction is typically applied to integer expressions involving an *induction variable*, which is defined later during loop optimization, and a high strength operator. The strength reduction should not be attempted on floating point expressions because the strength reduced

program may produce results different from the original program. This may happen due to the finite precision of computer arithmetic. A repeated floating point operation may accummulate a good amount of error value as compared to the result of a multiplication.

9.6.6 Dead Code Elimination

Dead codes are the portions of the program that can never get executed for any control flow through the program. Thus, these codes can be eliminated from the program without affecting the program behaviour. For example

1. A variable is said to be *dead* at a place in a program if the value contained in the variable at that place is not used anywhere in the program.
2. If an assignment is made to a variable v at a place where v is dead, then the assignment is a dead assignment.

Dead codes can get introduced in various manners. A typical example is the debugging information printed in the early phase of the program development. It is normally done as shown below:

```
#define DEBUG 1
...
#ifdef DEBUG
    print debugging information
...
```

Once the programmer is satisfied that the program is executing correctly, the first line is just changed to '#define DEBUG 0'. However, the statements to print debugging information remain there, which now become *dead code.*

Example 9.10 Consider the flow graph shown in Fig. 9.4. The following observations can be made:

1. The assignment $a = c$ of block 1 is not dead, since a is used in block 4.
2. The assignment $x = y - 5$ of block 5 is dead. The expression $y - 5$ can also be eliminated since it is no longer meaningful. However, an expression capable of producing *side effects* like function calls cannot be eliminated so simply.

9.6.7 Loop Optimization

Loops are very important candidates for optimization. Since the body of the loop is executed again and again, a small improvement in the execution of the loop body can result in a remarkable overall improvement in the performance of the program. Moreover, the loops may be nested. Thus, an improvement in the inner-most loop body is always targetted by the optimizing compilers. We, now, discuss the commonly used loop optimization techniques:

Induction Variable Analysis. An induction variable v is an integer scalar variable which is subjected only to the following kinds of assignments in a loop:

$$v = v \ op \ constant,$$

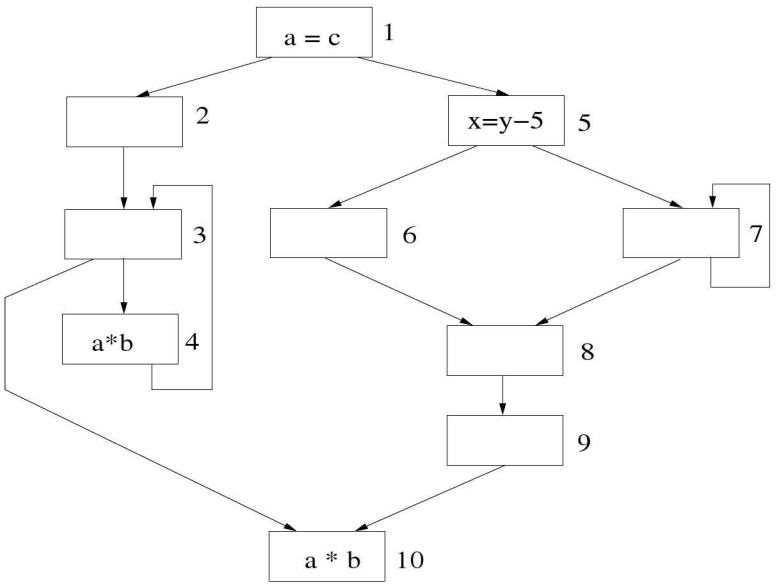

Figure 9.4: Dead code elimination.

where the operator *op* can only be addition or subtraction. For example, the code $j = 4 * i + 1$ can be updated appropriately each time the loop variable is changed, as shown in the Example 9.9, resulting into a *strength reduction* transformation. This may even result in the elimination of certain portion of the code as *dead code.*

Loop Fission. Loop fission attempts to break a loop into multiple loops over the same index range, each taking only a part of the loop's body. It can improve the *locality of reference.* For example, consider the code segment,

```
for i = 1 to 100 do
     a[i] = ...
     b[i] = ...
end loop
```

Here, for each value of i, the program first refers to an element of the array a and then, to another element from the array b. If there is no data dependency between the two statements in the loop body, then it can be broken down into two loops as shown below:

```
for i = 1 to 100 do
     a[i] = ...
end loop
for i = 1 to 100 do
     b[i] = ...
end loop
```

This version definitely shows better locality in the reference pattern. This is because in the first loop, it accesses the memory locations corresponding to a only, while in the second loop, it accesses the area for b only.

Loop Fusion or Loop Combining. The transformation opposite to *loop fission* is *loop fusion.* When two adjacent loops would iterate the same number of times, whether or not that number is known at compile time, their bodies can be combined as long as they make no reference to each other. This approach attempts to reduce the overhead due to loop setup, loop condition check and loop termination. Naturally, the decision about loop fusion depends upon the relative execution speed of the two loops versus single combined loop.

Loop Interchange. These optimizations exchange inner loops with outer loops. When the loop variables index into an array, such transformation can improve locality of reference, depending upon the layout of the array.

Example 9.11 Consider a two dimensional array $a[100][100]$ arranged in a *row-major* order, that is the elements are stored as $a[0][0], a[0][1], \ldots, a[0][99], a[1][0], \ldots, a[99][99]$. The program segment noted below makes nonlocal references as it accesses the array as $a[0][0], a[1][0], \ldots$ and so on.

```
for j = 0 to 99 do
     for i = 0 to 99 do
          a[i][j] = ...
```

If the loops on i and j are interchanged, the resulting program segment shows much better locality of reference as follows:

```
for i = 0 to 99 do
     for j = 0 to 99 do
          a[i][j] = ...
```

Loop Reversal. It reverses the order in which values are assigned to the index variable. This is a subtle optimization which can help eliminate dependencies and thus, enable other optimizations.

Example 9.12 Consider the problem of finding last occurrence of x in an integer array $a[100]$. The following is a code doing the same:

```
for i = 0 to 99 do
     k = 99 - i
     if a[k] = x then break
     ...
end loop
```

Here, the expression $99 - i$ is computed for each iteration of i. Changing the order of index variable assignment, computation of k is not necessary, making it a dead code to be eliminated. The modified code is,

```
for i = 99 downto 0 do
     if a[i] = x then break
     ...
end loop
```

Loop Unrolling. It duplicates the body of the loop multiple times, in order to decrease the number of times the loop condition needs to be tested. It also reduces the number of jumps from end of the loop to the beginning. It may be noted that jump statements affect performance by impairing the instruction pipeline. Completely unrolling a loop eliminates all overhead, but requires the number of iterations to be known at compile time. Moreover, if the loop body is big, then unrolling may increase code size significantly affecting performance.

Loop Splitting. Loop splitting attempts to simplify a loop or to eliminate dependencies by breaking it into multiple loops which have the same bodies but iterate over different contiguous portions of the index range. For example, consider the code segment

```
for i = 1 to 100 do
     if i < 30 then B1
     if 30 ≤ i < 60 then B2 else B3
end loop
```

The loop can be splitted into three loops as follows.

```
for i = 1 to 29 do
     B1
end loop
for i = 30 to 59 do
     B2
end loop
for i = 60 to 100 do
     B3
end loop
```

A useful special case is *loop peeling*, which can simplify a loop with a problematic first iteration by performing that iteration separately before entering the loop. For example, the code segment

```
for i = 1 to 100 do
     if i = 1 then B1 else B2
end loop
```

can be modified as follows

```
i = 1
B1
for i = 2 to 100 do
     B2
end loop
```

Unswitching. Unswitching makes a conditional to move from inside a loop to out of it by duplicating the loop's body and placing its version of it inside each of the *then* and *else* clauses of the conditional. For example, consider the code segment

```
for i = 1 to 100 do
     if x > y then
```

```
                a[i] = b[i] + c
            else
                a[i] = b[i] − c
        end loop
```

can be unswitched as,

```
        if x > y then
            for i = 1 to 100 do
                a[i] = b[i] + c
            end loop
        else
            for i = 1 to 100 do
                a[i] = b[i] − c
            end loop
```

Loop unswitching may increase the locality significantly.

Loop Test Replacement. It replaces a loop termination test phrased in terms of one variable, by a test phrased in terms of another variable. This may open up the possibility of dead code elimination. It is typically useful in strength reduction.

Example 9.13 Consider the following strength reduced program.

```
            temp = 5
            i = 1
    loop :  x = temp
            i = i + 1
            temp = temp + 5
            if i ≤ 10 then goto loop
```

The program can be modified as

```
            temp = 5
            i = 1
    loop :  x = temp
            i = i + 1
            temp = temp + 5
            if temp ≤ 50 then goto loop
```

Here, the loop induction variable i is no longer used meaningfully in the program. Hence, it can be eliminated.

9.7 LOCAL OPTIMIZATION

Local optimizations are restricted to essentially sequential code. As noted earlier, these optimizations consider the statements within a *basic block* only. Thus, local optimizations have limited scope for optimization, for example, *loop optimization*, *strength reduction,* etc. are not possible here. The most commonly used optimizing transformations are *variable propagation*

and *common subexpression elimination.* The biggest advantage of local optimization is that it does not need any sophisticated analysis of the program structure. The transformations can be carried out as we are building the DAG corresponding to a basic block. We will explain the concept with the help of an example.

Example 9.14 Consider the following piece of code:

stmt no.	**statement**
1.	$a = x * y$
2.	$z = x$
3.	$b = z * y$
4.	$x = b$
5.	$g = x * y$

After processing statement 1, we get the DAG as shown in Fig. 9.5. Here, x_0, y_0 and z_0 represent the initial values of the variables x, y and z respectively.

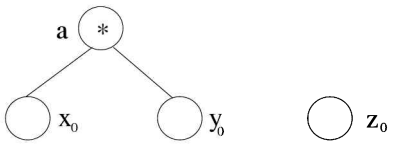

Figure 9.5: DAG after statement 1.

After processing the statements 1 to 3, the intermediate DAG is shown in Fig. 9.6. Here, *variable propagation* has occurred for variable z, while *common subexpression elimination* has taken place for $x * y$.

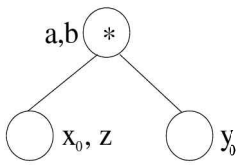

Figure 9.6: DAG after statement 1-3.

After processing the entire block, the resulting DAG is shown in Fig. 9.7. Here, *variable propagation* has taken place for b also.

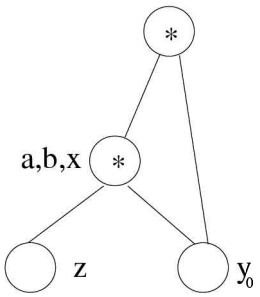

Figure 9.7: Complete DAG.

9.8 GLOBAL OPTIMIZATION

The scope of *global optimization* is, generally, a program unit, that is a procedure or a function body, usually spanning over a number of basic blocks. Since global optimization has better view of the program segment than the local optimization, which is restricted over a single basic block, it can achieve better transformations leading to more optimized code. However, it is difficult to identify the optimizing transformations applicable on individual statements of the bigger sized code chunk. Thus, it needs specialized techniques to identify the candidate statements for optimization and the kind of transformation applicable on them.

As noted in the previous chapter, a program is represented in the form of a *program flow graph*, with the basic blocks forming the nodes. Directed edges are created between the nodes having information transmission between them. For the sake of global optimization, this flow graph normally undergoes two types of analysis, namely *control flow analysis* and *data flow analysis*. *Control flow analysis* determines information concerning the arrangement of the graph nodes, that is, the *structure* of the program. For example, the presence of loops, nesting of loops, nodes visited before the control of execution reaches a specific node, etc. On the other hand, *data flow analysis* determines useful information for the purpose of optimization, for example, how data items are assigned and given reference in a program, values available when program execution reaches a specific statement of the program, etc. The control flow analysis and data flow analysis are now discussed along with their applications.

9.8.1 Control Flow Analysis

To start with, we will present a few definitions. Though some of these definitions have already been defined in the chapter on *Target Code Generation*, we discuss them here for the sake of clarity.

Definition 9.1 Program Point: *A program point w_j is the instant between the end of execution of instruction i_j, and the beginning of the execution of the instruction i_{j+1}. The effect of execution of instruction i_j is said to be completely realized at program point w_j.*

Definition 9.2 Program Flow Graph: *A program flow graph is a directed graph $G = (N, E, n_0)$, where N is the set of nodes. Each basic block of the program has a corresponding node in N. E is the set of control flow edges. An edge exists between the node i and j, if there is a control flow between the blocks b_i and b_j.*

Definition 9.3 Path: *A path is a sequence of edges $(e_1, e_2, \ldots, e_l)$, such that the terminal node of e_i is the initial node of e_{i+1}.*

Definition 9.4 Predecessor, Successor, Ancestor, Descendant: *A node b_i is a predecessor of b_j if there is an edge from b_i to b_j. All the nodes having a path to node b_i are called ancestors of it. The successor and descendant are defined similarly.*

Definition 9.5 Dominators: *A block b_i is said to be a dominator of block b_j if every path from node n_0 (initial node) to b_j passes through b_i. Similarly, b_i is post-dominator of b_j if every part starting at b_j passes through b_i before reaching an exit node of G.*

Definition 9.6 Regions: *A region $r = (V, E', V')$ is a connected subgraph of G, where $V \subseteq N$, $E' \subseteq E$ and $V' \subseteq N$ represent the set of nodes, edges and entry nodes respectively. There can be several kinds of regions, like loops, single entry regions, strongly connected regions, and intervals.*

Definition 9.7 Articulation Block: *b is an articulation block for a region R if every path from an entry node to an exit node of R necessarily passes through b.*

Significance of Control Flow Concepts

The above definitions give us an insight of the optimization techniques. The following observations can be made.

1. It is correct to move some code out of a node i if
   - it is inserted into a dominator node j of i, and
   - no assignment to any operands of the code occurs along any path $j \ldots i$.
2. Meaning of a program may be changed unless some code c, which is moved out of a loop, occurs in an articulation block of the loop.
3. It is incorrect to move any code out of a loop unless its occurrence(s) dominate(s) all exit nodes of the loop.
4. An expression e can be eliminated from a program point w if and only if
   - there exists at least one evaluation of e along every path reaching the point w, and
   - no operand of e is assigned a value after last such evaluation.

9.8.2 Data Flow Analysis

The optimization carried out by a compiler can be very effective if it is done using the global methods. This is because the scope of optimization becomes much bigger extending over a number of basic blocks. However, to identify the points of optimization, we need to understand the program structure in more detail. This can be carried out by analyzing the flow graph of the program. Data flow analysis determines the useful information for the purpose of optimization. For identifying the points of optimization, we need to analyze the data flow graph with different objectives. That is, we have to look for certain properties in the graph. For example, while trying out *common subexpression elimination*, we need detailed information regarding the set of those expressions whose definitions are available at various points in the flow graph. On the other hand, the *global register allocation* needs the set of *live variables* to be known at each and every point in the program. The particular information we are looking for is called a *data flow property.*

Definition 9.8 Data Flow Property: *A data flow property represents an item or a set of items of data flow information. These properties are associated typically with the entities of the flow graph and are defined by the compiler writer to use in a specific optimization.*

Data flow analysis is the process of computing the values of data flow properties. We, now, present some very important data flow properties commonly used by the optimizers.

Available Expressions. An expression e is *available* at the program point w if and only if along all the paths reaching w

- there exists an evaluation point for e. An evaluation point is a program point containing an evaluation of the expression.
- no definition of any operand of e follows its last evaluation along the path.

An expression e is said to be killed by a definition of any of its operands. Thus, an expression is not available following such definition. Determination of *available expressions* can help us to identify the common subexpressions and eliminate them.

Example 9.15 Consider the flow graph shown in Fig. 9.8. The only expression available at the entry point of node 10 is $\{c + d\}$. This is because all paths reaching block 10 have an evaluation of the expression which is not killed on the path. However, the expression $a * b$ is not available. Because, though its evaluation is present on all the paths reaching node 10, it is getting *killed* in block 4 by the definition of its operand a.

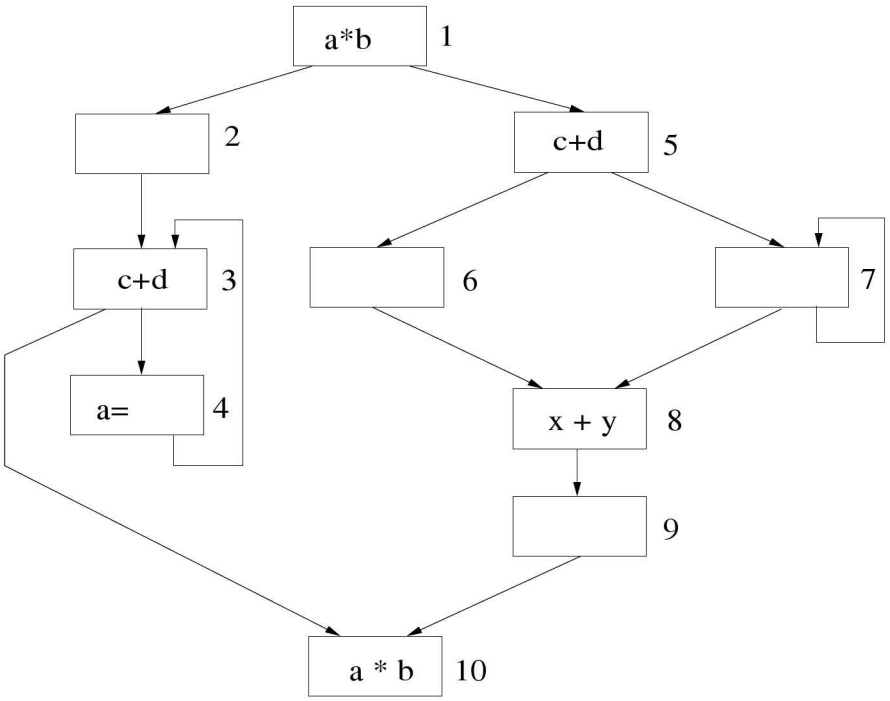

Figure 9.8: Available expression.

Reaching Definitions. A definition d of a variable v situated at a program point w_i is said to *reach* a program point w_j if and only if there exists a path in which the variable is not redefined. The concept of reaching definitions can be used for certain types of optimization. For example, we can try out a constant propagation to a point if it is the only definition that reaches there. On the other hand, a variable propagation can be carried out if we have a corresponding definition on each path reaching the point.

Example 9.16 Consider the flow graph shown in Fig. 9.9. Here the definitions reaching the entry of block 10 are $\{a = b, a = 7, x = 3\}$.

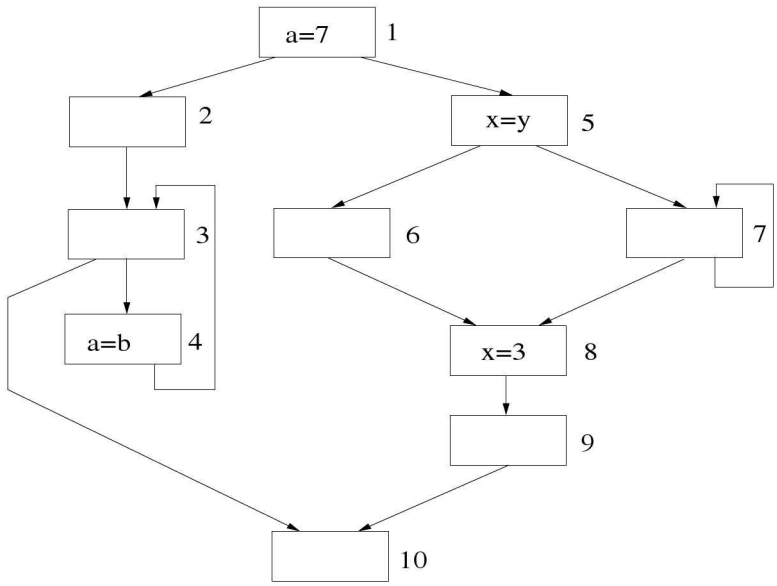

Figure 9.9: Reaching definition.

Live Variables. A variable v is live at a program point w_i if and only if

- v is given reference along some path $w_i \ldots w_j$ staring on program point w_i, and
- no assignment to v occurs before its reference along the path.

Thus, a variable v is live at some point if the value held in v is going to be utilized in some path following that point. Otherwise, the variable is considered *dead.* The *live variable* analysis has got several applications in optimization. For example, any assignment to a dead variable can be eliminated, as it is never going to be used. The live variable analysis is mostly applied in global register allocation. Two or more variables with disjoint life times can be merged and allocated a single register.

Example 9.17 Consider the flow graph shown in Fig. 9.10. Here, the variable a is live in nodes 1,5 and 6 because the only path in which a has been used is 1-5-6. Beyond 6, the variable a is no more needed. The variable x is live in nodes 8 and 9 from the point of its definition to its use. The variable b is never live, since, even if it has been assigned values in the nodes 2 and 10, the values are never utilized.

Busy Expressions. An expression e is *busy* at a program point if and only if

- an evaluation of e exists along some path $w_i \ldots w_j$ starting at program point w_i, and
- no definition of any operand of e exists before its evaluation along the path.

This essentially means that if an expression is found to be busy at some point, it is definitely going to be used in some path following that point. An expression is said to be *very busy* at w_i if it is busy along all the paths starting at w_i. It may be noted that if an expression is found to be very busy at a node, we can always move the evaluation to that node. This is a

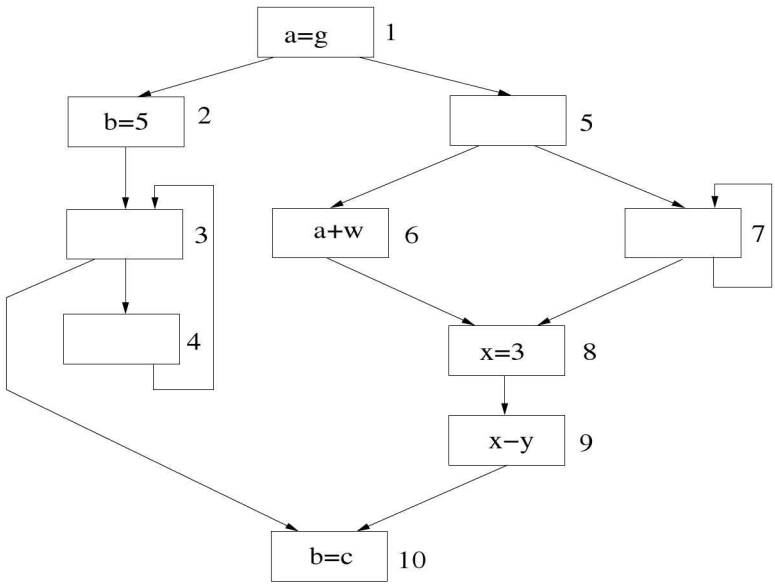

Figure 9.10: Live variables.

safe transformation. On the other hand, movement to a not very busy node is unsafe and may change the meaning of the program.

Example 9.18 Consider the flow graph shown in Fig. 9.11. Here, the expression $a * b$ is busy at node 5, since there exists a path 5-7-8-9-10 in which it is evaluated. However, it is not very busy because of the presence of the path 5-6-8-9-10, in which the expression is not evaluated. Thus, it is unsafe to move the expression to node 5. On the other hand, the expression $x + y$ is very busy at node 2, since it is evaluated on all the paths starting at 2. Thus, the movement of the expression to block 2 is safe.

Data Flow Information Representation. To compute the global data flow information, it is often necessary to calculate the information in a hierarchical fashion. In order to make the computation simpler, we need to have a very good representation of the data flow information. There are two natural representations of data flow information: *set representation* and *bit vector representation.* The applicability of both the representations depend on the type of operation intended on them. While the set representation is very general, the bit vector one is very compact.

Example 9.19 Consider the set of available expressions at some point in the program. It can be represented as a set, for example, $\{a * b, c + d, x - y, \ldots\}$, with the entries corresponding to individual expressions available there. For a bit vector representation, if the total number of expressions in the program be n, then we need an n-bit vector. Each expression is assigned a unique bit of this vector. At any point, the set of available expressions is represented as follows. If the expression k is available there, then the k-th bit is set to '1'. On the other hand, if an expression is not available at a certain point, then the corresponding bit is set to '0'.

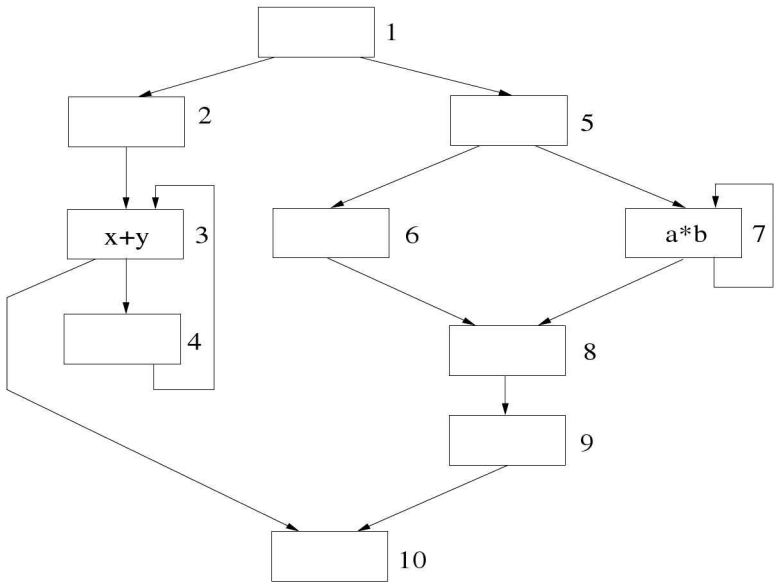

Figure 9.11: Busy expression.

9.8.3 Obtaining Data Flow Information

The data flow information gets accumulated or destroyed as we pass through the various program points. For example, at a point after the execution of the statement $x = y + z$, the variable x gets defined. Thus, we can consider the data flow information of a node corresponding to a basic block to have the following components:

1. ***Generated:*** It corresponds to the data flow information generated at a node. For example, an expression becomes available following its computation in a node.
2. ***Killed:*** Data flow information may be killed at a node. For example, a definition of a variable v kills all the expressions involving v.
3. ***Obtained:*** This is the information obtained from the neighbouring nodes. For example, a definition reaching the exit of a predecessor node also reaches the entry of the next node.

It may be noted that while the *generated* and *killed* components are *local* properties, the *obtained* component is nonlocal in nature. This is in the sense that the statements within a block are solely responsible for the generation or killing of data flow information. On the other hand, the obtained information are passed from its neighbours. Thus,

- data flow information at a node depends on the data flow information at other nodes in the program flow graph and
- data flow information at the entry and exit of a node is likely to be different.

Example 9.20 Let us consider the local data flow information for various points for the flow graph shown in Fig. 9.12. Let the information that we want to compute be the set of *available*

expressions. Let the set of expressions in the flow graph be $a * b$, $a + b$, $c + d$ and $x - y$. It may be coded into a 4-bit coding with each bit corresponding to a particular expression. The

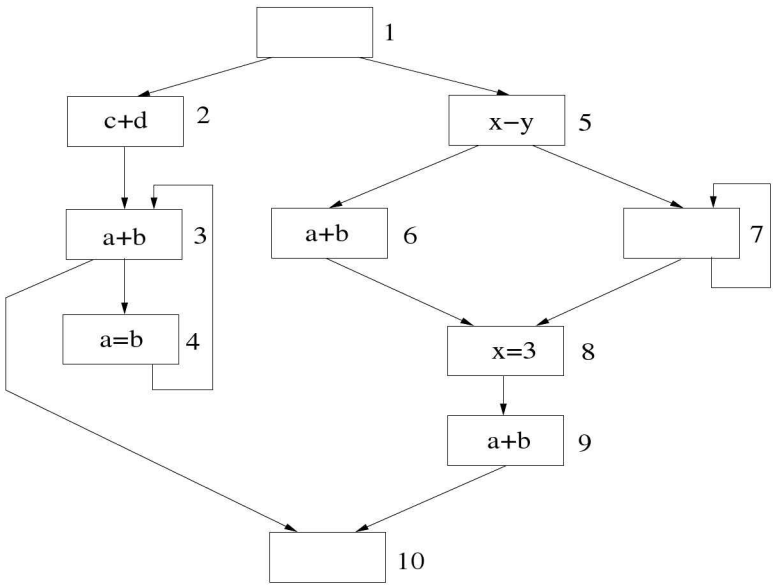

Figure 9.12: Local data flow information.

information generated and killed at various nodes are as follows:

node	*kill*	*gen*
1	ϕ, 0000	ϕ, 0000
2	ϕ, 0000	$\{c + d\}$, 0010
3	ϕ, 0000	$\{a + b\}$, 0100
4	$\{a * b, a + b\}$, 1100	ϕ, 0000
5	ϕ, 0000	$\{x - y\}$, 0001
6	ϕ, 0000	$\{a + b\}$, 0100
8	$\{x - y\}$, 0001	ϕ, 0000
9	ϕ, 0000	$\{a + b\}$, 0100

9.9 COMPUTING GLOBAL DATA FLOW INFORMATION

Once we have identified the *local* data flow information at individual nodes, we will next look into the strategy to combine them so that we can obtain the full data flow information for all the nodes of the flow graph. This will include the local data flow information alongwith the information passed from the predecessors of the nodes. The information are to be combined in different fashion depending upon the data flow property we are looking into. There are two general strategies for computation of global data flow information:

- Meet over paths and
- Constructing and solving data flow equations.

9.9.1 Meet Over Paths

This strategy defines a *merge* operator to combine the data flow information reaching a particular node through different paths. The strategy can be summerized as follows:

1. Consider all paths reaching (or starting on) a node.
2. Determine information flow along each path.
3. Take the meet over all the paths, that is, *merge* the information from various paths in proper fashion to determine the overall data flow information from neighbours. It may be noted that the *merge* or *meet* or *confluence* operator will depend on the type of information we are looking into. For example, while computing *available expressions*, the expressions should be available on all the paths. Thus, the overall data flow is the intersection of individual data flows from different paths. On the other hand, for *live variable* analysis, it should be the union operator.

Example 9.21 Consider the flow graph shown in Fig. 9.12. The set of available expressions for the nodes can be computed by first finding out the *kill* and *gen* for individual nodes and then, combining the information using intersection, which is the meet operator here.

node	available expressions at entry	available expressions at exit
2	ϕ, *0000*	$\{c+d\}$, *0010*
3	$\{c+d\}$, *0010*	$\{a+b, c+d\}$, *0110*
4	$\{a+b, c+d\}$, *0110*	$\{c+d\}$, *0010*
8	$\{x-y\}$, *0001*	ϕ, *0000*
9	ϕ, *0000*	$\{a+b\}$, *0100*
10	$\{a+b\}$, *0100*	$\{a+b\}$, *0100*

We can make the following observations concerning the problem of available expressions:

1. *Generated information:* An expression is generated in a node if the corresponding basic block contains a *downward exposed* occurrence of the expression, that is, an expression evaluation which is not followed by a definition of any of its operands till the end of the block. For example, consider the block:

   ```
   a * b
   c + d
   a = ...
   ```

   Here, the generated set is $\{c+d\}$, because though the expression $a*b$ occurs in the block, it has been followed by a definition of a.
2. *Killed information:* A definition of a variable, that is, an assignment to it, kills all the expressions involving the variable.
3. *Merging of information:* Merging is done using the *set intersection* operator $\cap$ or the bitwise 'and' operation since this is an *all paths* problem.

Definition 9.9 Forward data flow problem: *A data flow problem is said to be forward, if the data flow information for a program point p_i depends on the computations placed along one or more paths reaching p_i. The information flows in the direction of the flow of control in the program.*

Available expressions is a forward dataflow problem, because here the data flow information reaches a node i along all the paths from the start node n_0 to i.

Definition 9.10 Backward data flow problem: *A data flow problem is said to be backward, if the data flow information for a program point p_i depends on the computations placed along one or more paths starting on p_i. The information flows opposite to the direction of flow of control in the program.*

Live variables and *Busy expressions* are backward data flow problems.

Example 9.22 In this example, we illustrate how the identification of *live variables* can be formulated as a backward data flow problem. It may be noted that a variable v is referenced along some path $p_i \ldots p_j$ staring on p_i. Next, we present computation of *generate, kill* and *merge* functions.

1. *Generated information:* A variable v becomes *live* when it is referenced in an expression. Since the data flow is backward, the liveness due to a use in an expression extends *backwards* within the basic block, till a definition of v. Hence, a live variable is generated in a basic block due to an *upwards exposed* reference of a variable v, that is, a use of v not preceded by a definition within the basic block. For example, in the following block, the generated information is $\{v, b\}$. The reference a is not upwards exposed.

$$\begin{aligned} \texttt{a} &= \ldots \\ \ldots &= \texttt{a} * \texttt{b} \\ \ldots &= \texttt{v} * 3 \end{aligned}$$

2. *Killed information:* A definition of a variable, that is an assignment to it kills all of its liveness.
3. *Merging of information:* Merging is done using the *set union* operation $\cup$ or the bitwise 'or' operation.

The problem of *busy expressions* is also a backward data flow problem. A summary of data flow problems is shown in the Table 9.1.

9.9.2 Data Flow Equations

Data flow equations provide a better means to compute the data flow information. It uses the following procedure to obtain the global data flow information:

1. Take the meet of the information available at the entry (or exit) of each node.
2. Consider the information being generated or killed within the node.
3. Obtain the information available at the exit(entry) of the node.

Table 9.1: Summary of Data Flow Problems

Data flow problem	Generated information	Killed information	Merge operator
Available Expressions	Downwards exposed occurrence of an expression	Occurrence of $a = \ldots$ kills expressions using a	$\cap$
Reaching Definitions	Downwards exposed occurrence of $v = \ldots$	Occurrence of $v = \ldots$ kills previous definitions of v	$\cup$
Live Variables	Upwards exposed reference of x	Occurrence of $v = \ldots$ kills variable v	$\cup$
Very Busy Expressions	Upwards exposed occurrence of an expression	Occurrence of Occurrence of $v = \ldots$ kills expressions using v	$\cap$

It may be noted that for certain type of data flow information, we need to compute in a downward manner, while in some other cases we need to proceed in upward manner. To compute the data flow information, we have to construct the data flow equations at each node of the flow graph and then, solve them simultaneously.

Example 9.23 Let us consider the computation of *available expressions* using data flow equation. The data flow equations can be constructed as follows:

$$\begin{aligned} AVIN_i &= \cap_{\forall_p} AVOUT_p \\ AVOUT_i &= AVIN_i - AVKILL_i \cup AVGEN_i \end{aligned}$$

where

$AVIN_i/AVOUT_i$ is availability at entry/exit of node i,
$AVKILL_i$ represents information killed in node i,
$AVGEN_i$ represents information generated in node i and
$\cap_{\forall_p}$ represents $\cap$ over all predecessors.

Forward Data Flow Problems. As noted earlier, the forward data flow problems are those problems in which the data flow information for the nodes can be computed along the direction of flow of control. The corresponding data flow equations are formulated as follows, where p is a predecessor of node i:

$$\begin{aligned} IN_i &= \theta_{\forall_p} OUT_p \\ OUT_i &= IN_i - KILL_i \cup GEN_i, \end{aligned}$$

where

θ is the confluence operator $\cup$ or $\cap$,
IN and OUT represent information at entry and exit respectively,
$KILL$ represents information killed in the node and
GEN represents information generated in the node.

Available expressions and *reaching definitions* are examples of forward data flow problems. For the *available expressions*, we have seen the data flow equations earlier. For *reaching definitions*, the data flow equations are defined as

$$\begin{aligned} DEFIN_i &= \cup_{\forall_p} DEFOUT_p \\ DEFOUT_i &= DEFIN_i - DEFKILL_i \cup DEFGEN_i \end{aligned}$$

where

$DEFIN/DEFOUT$ represent reaching definitions at entry/exit,
$DEFKILL$ represents presence of another definition(s),
$DEFGEN$ represents definitions generated in node and
$\cup_{\forall_p}$ represents $\cup$ over all predecessors.

Backward Data Flow Problems. Backward data flow problems are those problems for which the data flow computation should be done in the direction opposite to the program control flow. The data flow equations corresponding to it can be formulated as follows, where node s is the successor of node i:

$$\begin{aligned} OUT_i &= \theta_{\forall_s} IN_s \\ IN_i &= OUT_i - KILL_i \cup GEN_i, \end{aligned}$$

where

θ is the confluence operator $\cup$ or $\cap$,
IN and OUT represent information at entry and exit respectively,
$KILL$ represents information killed in the node and
GEN represents information generated in the node.

The data flow properties *busy expressions* and *live variables* are the examples of this category.

Example 9.24 The *busy expressions* can be computed by the following data flow equations:

$$\begin{aligned} BUSYOUT_i &= \cup_{\forall_s} BUSYIN_s \\ BUSYIN_i &= BUSYOUT_i - BUSYKILL_i \cup BUSYGEN_i \end{aligned}$$

where

$BUSYIN$ and $BUSYOUT$ represent busy expressions at entry and exit respectively,
$BUSYKILL$ indicates the presence of operand definition(s) for the expression in the node and
$BUSYGEN$ indicates busy expressions generated in the node.

Example 9.25 The *live variables* computation can be performed by the following data flow equations:

$$\begin{aligned} LIVEOUT_i &= \cup_{\forall_s} LIVEIN_s \\ LIVEIN_i &= LIVEOUT_i - LIVEKILL_i \cup LIVEGEN_i \end{aligned}$$

where

$LIVEIN$ and $LIVEOUT$ represent variables live at entry and exit respectively,
$LIVEKILL$ indicates the presence of another definition(s) for the variable in the node and
$LIVEGEN$ indicates live variables generated in the node.

9.10 SETTING UP DATA FLOW EQUATIONS

In the previous section, we have studied the data flow equations as a powerful tool to collect information about the data flow properties. Now, in this section, we will look into how the data flow equations can be set up to collect the information to perform a specific operation. The approach consists of three steps:

1. Step 1: Decide on *what* information is adequate to perform the desired operation.
2. Step 2: Analyze the nature of the information and decide *how* it may be collected.
3. Step 3: Design a data flow problem to collect the information. Develop data flow equations.

Example 9.26 Let us consider the optimization *Copy Propagation*. It is defined as follows:

At program point w, we can replace x by y if x is a copy of y at that point, that is, if x has the same value as y.

To perform this optimization, we have to first decide upon the type of information needed. Obviously, we need to know the pairs of variables which are copies of each other. Let *COPIES* be a set of pairs $\{(x, y) |$ assignment $x = y$ exists along some path$\}$. Thus, if we know the set *COPIES*, at a program point the variable substitution can be carried out based on that. Now, *COPIES* can be computed for every program point by using *C_IN* and *C_OUT* to be the set of copies associated with the entry and exit points.

The *C_IN* of a node can be computed from the *C_OUT* of its predecessors either by *union* or by *intersection* operation. If we use the union operator, we may get a set *COPIES* to be $\{(a, b), (a, c) \ldots\}$. It can happen because the pairs (a, b) and (a, c) may be coming from two different paths. Thus, a variable a occurring at this program point cannot be substituted by either b or c. On the other hand, if we use *intersection* operation, atmost one pair (a, v) may exist in *COPIES* for any a. Existence of such a pair ensures that an occurrence of a at this program point may be substituted correctly by v. Thus, *intersection* is the correct confluence operator in this case.

The data flow equations can now be set as follows:

$$\begin{aligned} C\_IN_i &= \cap_{\forall_p} C\_OUT_p \\ C\_OUT_i &= C\_IN_i - C\_KILL_i \cup C\_GEN_i, \end{aligned}$$

where

C_IN/C_OUT represent information at entry/exit,
C_KILL represents information killed in the node,
C_GEN represents information generated in node and
$\cap_{\forall_p}$ is the confluence operator $\cap$.

Next, we have to define the sets C_GEN and C_KILL. For the assignment statement $x = y$, the sets are defined as follows:

- $C\_GEN = \{(x, y)\}$, because from this point onwards, x is a copy of y till the information gets killed by an assignment to either of x and y.
- $C\_KILL = \{(x, h), (h, x) \forall h\}$. This is because the assignment makes all other copies of x invalid.

9.10.1 Data Flow Analysis

A solution of the data flow equations consists of an assignment of values to the IN and OUT sets of each node in the flow graph. However, to be useful, the information contained in a solution must satisfy the following conditions:

1. *Self-consistency:* The values assigned to the different nodes must be mutually consistent, else the solution is incorrect. Such a consistent solution is known as a *fixed point* of the data flow equations.
2. *Conservative information:* The data flow information must be conservative in that optimization. Using this information should not change the meaning of a program under any circumstances.
3. *Meaningfulness:* The computed values of the properties should provide meaningful (in fact, maximum) opportunities for optimization. This is important, since not performing any optimization is conservative but hardly meaningful in an optimizing compiler.

9.10.2 Conservative Solution of Data Flow Equations

As we have mentioned in the beginning of this chapter, the optimization should always be *safe*, that is, it should not transform the program in a manner which leaves a scope of the behaviour of the program getting changed. Since the data flow equations are used to gather information about the source of optimizations, we have to be careful in choosing the information to be included in the solution set. *Conservative data flow* analysis assists the process. The situation may better be understood by considering a situation in which a procedure (P) is called from two different blocks, say, B_1 and B_2 of the program. Also, assume that depending upon the point from which it was called, the procedure makes call to one of the two possible functions F_1 and F_2. In this case, the flow graph will have paths from both B_1 and B_2 to F_1 and F_2 through P. This path is the *graph theoretic path.* However, actual *execution paths* are different from the graph theoretic paths. Thus, the *computed* data flow information may be different from the *actual* data flow information. Naturally, the optimization based on *computed* data flow information would be correct if the differences between the computed and actual values lie on the safer side. This may be *conservative* in the sense that it tends to disallow certain feasible optimizations, but never enable an erroneous one.

Example 9.27 Let $a * b$ be an available expression at the point just before an *if* statement. Suppose an assignment to the variable a in the *then*-part of the program kills the expression, whereas the *else*-part is not having any such assignment to either a or b. Then, it is conservative to assume that the expression $a*b$ is not available after the *if* statement, even though the *then*-branch may never be visited during the execution of the program. Use of $\cap$ as the confluence operator ensures a conservative solution. Optimization based on this solution will never be wrong.

Conservative Assumptions. It is necessary to make conservative assumptions when complete and precise data flow information is not available. The following are a few instances of such cases:

1. *Array assignments:* If the value of the subscript variable i cannot be determined at compile time, we have to assume that definitions of all the elements of array a get killed by an assignment to $a[i]$.

2. *Pointer based assignments:* Pointers pose a major difficulty to the optimization since, an integer pointer can point to any other integer defined in the program. Thus, in absence of a thorough knowledge about the variable currently pointed to by a pointer, a conservative assumption is to assume that it may point to any one, and thus, an operation changing the content of a location through a pointer can be assumed to affect assignments to all the variables of that type. For example, in the following code segment, though $a * b$ appears to be a common subexpression, a conservative estimate will indicate that the expression is not available at the second point due to the presence of assignment via $*p$.

$$\begin{aligned} &\ldots = \texttt{a} * \texttt{b} \\ &*\texttt{p} = \ldots \\ &\ldots = \texttt{a} * \texttt{b} \end{aligned}$$

3. *Procedure calls:* Procedure calls, particularly with modifiable parameters force us to use conservative assumptions. For example, on the basis of arguments similar as *pointer based assignments*, the following code segment cannot be transformed to use $a * b$ as a common subexpression:

$$\begin{aligned} &\ldots = \texttt{a} * \texttt{b} \\ &\texttt{p(a, x)} \\ &\ldots = \texttt{a} * \texttt{b} \end{aligned}$$

Conservative estimates make the computed values of data flow properties less precise. This can only be corrected through more analysis like, interprocedural analysis, alias analysis etc.

9.11 ITERATIVE DATA FLOW ANALYSIS

Once the data flow equations have been set up, the next task is to obtain solutions from them. Since the equations for all the blocks can be solved simultaneously, an iterative approach needs to be followed. It consists of the following steps:

1. Set the IN and OUT properties of all the nodes in the program flow graph except for the program entry/exit nodes to some initial value.
2. Visit all the nodes in the program flow graph and recompute their IN and OUT properties.
3. If any changes occur in any IN or OUT properties, then repeat steps 2 and 3.

The solution is a *fixed point* one, because the iteration stops only after all the IN and OUT sets have stabilized. However, just like any iterative scheme, the solution depends on the initial assignments chosen for the IN and OUT sets. For example, consider the flow graph shown in Fig. 9.13. Here, in the block 1, the expression e_1 is generated and is assumed not to be killed by any of the remaining blocks. Thus, intuitively, it should be available at the input of each block. That is, when the iteration terminates, the IN of all the blocks should have e_1 as a member. However, as explained below, we may or may not reach this solution.

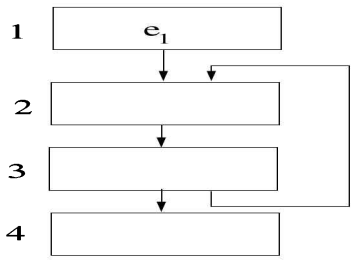

Figure 9.13: Flow graph for available subexpression.

1. Set IN_1 to $\{\}$. Set $IN = OUT = \{e_1\}$ for all other nodes. Using the data flow equations formulated earlier, the algorithm terminates with the solution,

$$IN_2 = IN_3 = IN_4 = \{e_1\}$$

2. Set IN and OUT of all blocks to $\{\}$. Here, the iteration converges to the solution,

$$IN_2 = IN_3 = IN_4 = \{\}$$

Thus, initializing IN and OUT of all the nodes to $\{\}$ leads to a trivial solution for the available expressions. To obtain the most meaningful solution, that is, the largest possible solution to the equations, we have to use the following initializations for different problems:

- For $\cap$ problems, initialize all the nodes to the universal set.
- For $\cup$ problems, initialize all the nodes to $\{\}$.

9.11.1 Available Expressions

In this section, we present a detailed algorithm to determine the set of *available expressions* at all the nodes of a flow graph. The node n_0 is the start node of the flow graph. Since no expression is available initially, the set $AVIN_{n_0}$ is initialized to $\{\}$. On the other hand, based on the principle discussed above, $AVOUT$ for all nodes are initialized to contain all the expressions of the program, represented by the universal set U except those killed by the block and not generated thereafter within the block. The iteration, then, continues till there are modifications in the $AVOUT$ sets.

```
/* Initializations */
    AVIN_n0 = {}
    AVOUT_n0 = AVGEN_n0
    ∀i ∈ N − n0 set AVOUT_i = U − AVKILL_i ∪ AVGen_i
    /* Iteration */
    change = true
    while change do
    begin
        change = false
        ∀i ∈ N − n0 do
        begin
```

```
            AVIN_i = ∩_{∀p} AVOUT_p
            oldout_i = AVOUT_i
            AVOUT_i = AVGEN_i ∪ (AVIN_i − AVKILL_i)
            if AVOUT_i ≠ oldout_i then change = true
        end
    end.
```

It may be shown that an upper bound on the number of iterations around the *while* loop is the number of nodes in the flow graph. Intuitively, the reason is that if an expression is available at a point, it can be coming to the point only through cycle-free path, and the number of nodes in a flow graph is an upper bound on the nodes in a cycle-free path. Similar to the algorithm for *available expressions*, we can formulate strategies for computing other data flow properties like reaching definitions busy expressions and so on.

9.11.2 Live Range Identification

We will, now, discuss a strategy to identify the live range of a variable over the total program flow graph. The information gathered may, then, be utilized to perform *global register allocation.* This method is known as *Chow-Hennessy* approach.

A basic block of the program belongs to the *live range* of a value if the value is *live* within the basic block and a reference or a definition of the value *reaches* it. The live range is the set of such basic blocks. The issue is elaborated below:

1. *Value is live:* This implies that the value is used along some path through this block, and hence, it is meaningful to hold it in a register.
2. *A reference is reaching and the value is live:* This implies that the value is currently in a register (it would have been loaded at the reference that is reaching) and is required along some path through this basic block.
3. *A definition is reaching and the value is live:* The value is currently in a register (it would have been put there by the definition that is reaching) and is required along some path through this block.

Example 9.28 Consider the flow graph shown in Fig. 9.14. The live ranges for the variables a and b are computed as

Live range of a: {1,2,3,4}
Live range of b: {3,4}

9.11.3 Reducing Complexity of Iterative Data Flow Analysis

The complexity of the data flow analysis using data flow equation solution strategy has been shown to be of the order of nodes in the flow graph. However, in many cases we do not need to perform so many iterations. In this direction, we will define a quantity called *depth* of the flow graph to specify the maximum number of iterations needed. Of course, this needs considering the basic blocks in certain order and not in a random arbitrary sequence.

To start with, the nodes of the flow graph are assigned a number, called *depth first number.* The *depth first numbering (dfn)* of the nodes of the graph is the reverse of the order in which

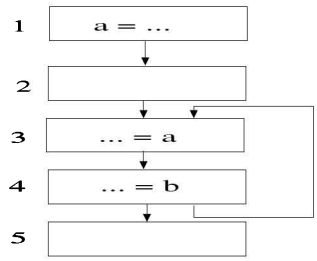

Figure 9.14: Live range identification.

we last visit each node in a pre-order traversal of the graph. The depth first numbering has the following properties:

1. $\forall i \in dominators(j), dfn(i) < dfn(j)$,
2. $\forall forward\ edges\ (i,j), dfn(i) < dfn(j)$ and
3. $\forall backward\ edges\ (i,j), dfn(j) < dfn(i)$.

The *depth d* of a program flow graph is the maximum number of back edges in any acyclic path in it. It may be noted that it is not the same as the nesting depth. For example, in the flow graph shown in Fig. 9.15, the nesting depth is 2, but the *depth d* is equal to 1.

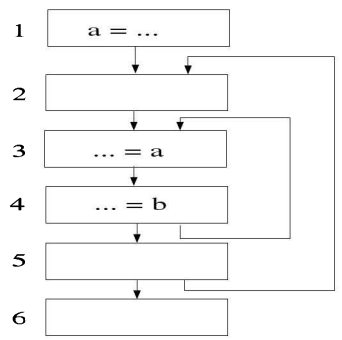

Figure 9.15: Example flowgraph.

The most important result for iterative analysis is as follows:

For a forward data flow problem, if the nodes of a graph are visited in a depth first order, $d + 1$ iterations are sufficient to reach a fixed point. For a backward data flow problem, the nodes should be visited in the reverse depth first order.

The point can be justified by the following observations.

1. For a forward data flow problem, we visit the nodes in the increasing order by depth first numbers. If there are no loops in the program, that is $d = 0$, then the data flow information computed in the first iteration would be a fixed point of the data flow equations.
2. If a single loop exists in the program, then $d = 1$. Now, the new value of the loop exit node computed in the first iteration can influence the property of the loop entry node. This can be achieved only in the next iteration. Hence, two iterations are necessary.

3. If another back-edge starts on some node of the loop such that the loop formed by it is not contained in the outer loop, then yet another iteration is required. For example, for the flow graph shown in Fig. 9.16, the effect of assignment $a = \ldots$ is felt on the availability of $a * b$ at the entry of block 2 only in the third iteration.

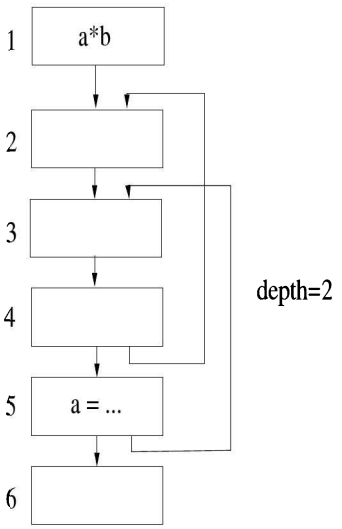

Figure 9.16: Example flow graph.

9.12 CONCLUSION

Code optimization forms a very important part of any good compiler design. Whereas the other portions of compiler design (like lexical analysis, syntax analysis, etc.) are more of a routine matter, *code optimization* leaves a lot of scope for research and development. Optimization is necessary to make the life of a programmer easier, however, identifying the optimizing transformations for a program is a nontrivial task. There are several factors affecting optimization—the major one being target architecture. We have seen various transformations that are expected to improve the performance of the generated code *elimination of common subexpression, loop optimization* and so on. Global optimization of program needs collecting global information about a program. This is done using the technique called *data flow analysis.* We have enumerated several such data flow problems, formulation of data flow equations and their solutions. Finally, iterative data flow analysis has been presented as a good tool to solve the data flow equations.

EXERCISES

9.1 Discuss the necessity of optimization in compilation.

9.2 What are the problems in optimizing compiler design?

9.3 Discuss the factors influencing optimization.

9.4 Study the program flow graph shown in Fig. 9.17 and indicate with justification whether or not any of the following optimizing transformations can be applied to it.

(a) common subexpression elimination:
(b) elimination of dead code
(c) constant propagation
(d) frequency reduction

9.5 Specify the necessary and sufficient conditions for performing:

(a) constant propagation
(b) dead code elimination
(c) loop optimization.

9.6 Develop complete algorithms for the following optimizations:

(a) common subexpression elimination
(b) elimination of dead code
(c) constant propagation
(d) frequency reduction

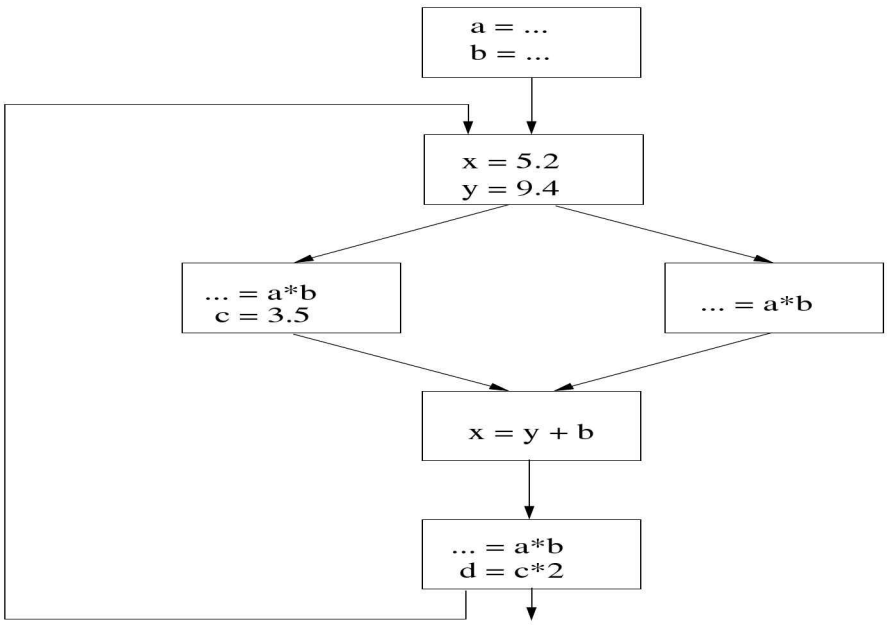

Figure 9.17: Program flow graph for 9.4

Apply these algorithms to the program flow graph of Fig. 9.17.

9.7 Write a note justifying the need for $d+1$ iterations, where d is the depth of a graph, for the iterative solution of a data flow problem.

9.8 Given an assignment statement of the form

$$a = b$$

copy propagation implies substituting b for a at every usage point of a reached by the definition $a = b$.

(a) Develop a complete algorithm for copy propagation.
(b) Can copy propagation be performed transitively? For example, in

$$a = b$$
$$c = a$$

can c be replaced by b? If so, explain how will you modify your algorithm to perform this enhanced copy propagation.

Bibliography

[1] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley, 1986.

[2] R. Bornat, *Understanding and Writing Compilers.* London, England: Macmillian Education, 1979.

[3] P.B. Hanson, *Brinch Hanson on Pascal Compilers.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1985.

[4] P.C. Capon and P.J. Jinks, *Compiler Engineering Using Pascal.* London, England: Macmillian Education, 1988.

[5] A.J.T. Davie and R. Morrison, *Recursive Descent Compiling.* Chichester, England: Ellis Horwood, 1982.

[6] C.N. Fischer and R.J. LeBlanc, *Crafting a Compiler.* Menlo Park, CA: Benjamin/ Cummings, 1988.

[7] K.J. Gough, *Syntax Analysis and Software Tools.* Reading, MA: Addison-Wesley, 1988.

[8] D. Gries, *Compiler Construction for Digital Computers.* New York, NY: John Wiley & Sons, 1971.

[9] D. Grune and C.J.H. Jacobs, *Parsing Techniques a Practical Guide.* Chichester, England: Ellis Horwood, 1990.

[10] R. Hunter, *Compilers Their Design and Construction Using Pascal.* Chichester, England: John Wiley & Sons, 1985.

[11] B.W. Leverett, *Register Allocation in Optimizing Compilers.* Ann Arbor, MI: UMI Research Press, 1983.

[12] S.S. Muchnick and N.D. Jones, *Program Flow Analysis: Theory and Applications.* Englewood Cliffs, NJ: Prentice-Hall, 1981.

[13] S. Pemberton and M. Daniels, *Pascal Implementation: The P4 Compiler.* Chichester, England: Ellis Horwood, 1982.

[14] S. Pemberton and M. Daniels, *Pascal Implementation: Compiler and Assembler/ Interpreter.* Chichester, England: Ellis Horwood, 1982.

[15] T. Pittman and J. Peters, *The Art of Compiler Design Theory and Practice.* Englewood Cliffs, NJ: Prentice-Hall, 1992.

[16] B.W. Pollock, *Compiler Techniques.* Princeton, NJ: Auerbach Publishers, 1972.

[17] P. Rechenberg and H. Moessenboeck, *A Compiler Generator for Microcomputers.* Englewood Cliffs, NJ: Prentice-Hall, 1989.

[18] M. Rees and D. Robson, *Practical Compiling with Pascal-S.* Reading, MA: Addison-Wesley, 1988.

[19] J.S. Rohl, *An Introduction to Compiler Writing.* New York, NY: American Elsevier, 1975.

[20] R. Rustin, *Design and Optimization of Compilers.* Englewood Cliffs, NJ: Prentice-Hall, 1972.

[21] S. Sippu and E.S.-S. (Ed.), *Parsing Theory: Languages and Parsing*, vol. 1, Berlin, Germany: Springer-Verlag, 1988.

[22] J.-P. Tremblay and P.G. Sorenson, *An Implementation Guide to Compiler Writing.* New York, NY: McGraw-Hill, 1982.

[23] W.M. Waite and G. Goos, *Compiler Construction.* New York, NY: Springer-Verlag, 1984.

[24] W.A. Wulf, R.K. Johnsson, B. Weinstock, S.O. Hobbs, and C.M. Geschke, *The Design of an Optimizing Compiler.* New York, NY: American Elsevier, 1975.

[25] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin, "Dynamic typing in a statically typed language," *ACM Transactions on Progmmming Languages and Systems,* vol. 13, pp. 237–268, April 1991.

[26] S.G. Abraham and D.E. Hudak, "Compile-time partitioning of iterative parallel loops to reduce cache coherency traffic," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, pp. 318–328, July 91.

[27] P. Abrahams, "The CIMS PL/1 compiler," in *Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction*, vol. 14, pp. 107–116, August 1979.

[28] A.V. Aho, R. Sethi, and J.D. Ullman, "A formal approach to code optimization," in *Proceedings of a Symposium on Compiler Optimization*, vol. 5, pp. 86–100, July 1970.

[29] A.V. Aho, S.C. Johnson, and J.D. Ullman, "Deterministic parsing of ambiguous grammars," in *Conference Record of the ACM Symposium on Principles of Programming Languages*, pp. 1–21, October 1973.

[30] A.V. Aho and S.C. Johnson, "LR parsing," *ACM Computing Surveys*, vol. 6, no. 2, pp. 99–124, 1974.

[31] A.V. Aho and J.D. Ullman, "Listings for reducible flow graphs," in *7th ACM Symposium on the Theory of Computation*, pp. 177–185, 1975.

[32] A.V. Aho, S.C. Johnson, and J.D. Ullman, "Deterministic parsing of ambiguous grammars," *Communications of the ACM*, vol. 18, no. 8, pp. 441–452, 1975.

[33] A.V. Aho and S.C. Johnson, "Optimal code generation for expression trees," *Journal of the ACM,* vol. 23, pp. 488–501, July 1976.

[34] A.V. Aho, S.C. Johnson, and J.D. Ullman, "Code generation for expressions with common subexpressions," *Journal of the ACM*, vol. 24, pp. 146–160, January 1977.

[35] A.V. Aho, S.C. Johnson, and J.D. Ullman, "Code generation for machines with multiregister operations," in *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, (Los Angeles, CA), pp. 21–28, January 1977.

[36] A.V. Aho and M. Ganapathi, "Efficient tree pattern matching: an aid to code generation," in *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages*, pp. 334–340, January 1985.

[37] A.V. Aho, M. Ganapathi, and S.W.K. Tjiang, "Code generation using tree matching and dynamic programming," *ACM Transactions on Programming Languages and Systems*, vol. 11, pp. 491–516, October 1989.

[38] P. Aigrain, S.L. Graham, R.R. Henry, M.K. McKusick, and E. Pelegri-Llopart, "Experience with a Graham-Glanville style code generator," in *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, vol. 19, pp. 13–24, June 1984.

[39] M. Albinus and W. Aßmann, "The INDIA lexic generator," in *Compiler Compilers and High Speed Compilation 2nd CCHSC Workshop Proceedings* (Berlin, GDR), pp. 115–127, October 1988.

[40] F.E. Allen, *Program Optimization*, vol. 13, pp. 239–307, 1969.

[41] F.E. Allen, "Control flow analysis," in *Proceedings of a Symposium on Compiler Optimization,* vol. 5, pp. 1–19, July 1970.

[42] F.E. Allen and J. Cocke, *A Catalogue of Optimizing Transformations*, pp. 1–30, Prentice-Hall, Englewood Cliffs, NJ, 1972.

[43] F.E. Allen, *Interprocedural Analysis and the Information Derived from it,* pp. 291–321, Springer-Verlag, 1974.

[44] F.E. Allen and J. Cocke, "A program data flow analysis procedure," *Journal of the ACM,* vol. 19, pp. 137–147, March 1976.

[45] F.E. Allen, J. Cocke, and K. Kennedy, *Reduction of Operator Strength*, pp. 79–101, Englewood Cliffs, NJ: Prentice-Hall, 1981.

[46] R. Allen and K. Kennedy, "Automatic loop interchange," in *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, vol. 19, pp. 233–246, June 1984.

[47] B. Alpern, M.N. Wegman, and F.K. Zadeck, "Detecting equality of variables in programs," in *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages* (San Diego, CA), pp. 296–306, January 1988.

[48] R.M. Amadio and L. Cardelli, "Subtyping recursive types," in *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages* (Orlando, FL), pp. 104–118, January 1991.

[49] U. Ammann, "On code generation in a Pascal compiler," vol. 7, pp. 391–423, June 1977.

[50] Z. Ammarguellat and W.L.H. III, "Automatic recognition of induction variables and recurrence relations by abstract interpretation," in *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation,* vol. 25 (White Plains, NY), pp. 283–295, June 1990.

[51] M. Ancona, G. Dodero, V. Gianuzzi, and M. Morgavi, "Efficient construction of LR(k) states and tables," *ACM Transactions on Programming Languages and Systems,* vol. 13, pp. 150–178, January 1991.

[52] T. Anderson, J. Eve, and J.J. Horning, "Efficient LR(1) parsers," *Acta Informatica,* vol. 2, no. 1, pp. 12–39, 1973.

[53] K. Andrews, R.R. Henry, and W.K. Yamamoto, "Design and implementation of the UW illustrated compiler," in *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation,* vol. 23 (Atlanta, GA), pp. 105–114, June 1988.

[54] K. Andrews and D. Sand, "Migrating a CISC computer family onto RlSC via object code translation," in *5th International Conference on Architectural Support for Programming Languages and Operating Systems,* vol. 27 (Boston, MA), pp. 213–222, October 1992.

[55] A.W. Appel, "Semantics-directed code generation," in *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages,* pp. 315–324, January 1985.

[56] A.W. Appel and T. Jim, "Continuation-passing, closure-passing style," in *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages* (Austin, TX), pp. 293–302, January 1989.

[57] A.W. Appel, "Unrolling recursion saves space," Technical Report CS-TR-363–92, Princeton University, March 1992.

[58] B.W. Arden, B.A. Galler, and R.M. Graham, "An algorithm for translating Boolean expressions," *Communications of the ACM,* vol. 5, pp. 222–239, April 1962.

[59] M. Ben-Ari, "Algorithms for on-the-fly garbage collection," *ACM Transactions on Programming Languages and Systems,* vol. 6, pp. 333–344, July 1984.

[60] W. Assmann, "A short review of high speed compilation," in *Compiler Compilers and High Speed Compilation 2nd CCHSC Workshop Proceedings* (Berlin, GDR), pp. 1–10, October 1988.

[61] L.V. Atkinson, "Optimizing two-state case statements in Pascal," vol. 12, pp. 571–581, 1982.

[62] M. Auslander and M. Hopkins, "An overview of the PL.8 compiler," in *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction,* vol. 17 (Boston, MA), pp. 22–31, June 1982.

[63] R.J.R. Back, H. Mannila, and K.-J. Räihä, "Derivation of efficient DAG marking algorithms," in *Conference Record of the 10th Annual ACM Symposium on Principles of Programming Languages,* pp. 20–27, January 1983.

[64] R.C. Backhouse, "An alternative approach to the implementation of LR parsers," *Acta Informatica*, vol. 6, no. 3, pp. 277–296, 1976.

[65] R.C. Backhouse, "Global data flow analysis problems arising in locally least-cost error recovery," *ACM Transactions on Programming Languages and Systems*, vol. 6, no. 2, pp. 192–214, 1984.

[66] J.T. Bagwell, Jr., "Local optimizations," in *Proceedings of a Symposium on Compiler Optimization,* vol. 5, pp. 52–66, July 1970.

[67] R. Bahlke, B. Moritz, and G. Snelting, "A generator for language-specific debugging systems," in *Proceedings of the ACM SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques,* vol. 22 (St. Paul, MN), pp. 92–101, June 1987.

[68] T.P. Baker, "A single-pass syntax-directed front end for Ada," in *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction,* vol. 17, pp. 318–326, June 1982.

[69] T.P. Baker, "A one-pass algorithm for overload resolution in Ada," ACM *Transactions on Programming Languages and Systems,* vol. 4, no. 4, pp. 601–614, 1982.

[70] H.E. Bal and A.S. Tanenbaum, "Language- and machine-independent global optimization on intermediate code," *Computer Language,* vol. 11, no. 2, pp. 105–121, 1986.

[71] J.E. Ball, "Predicting the effects of optimization on a procedure body," in *Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction,* vol. 14, pp. 214–220, August 1979.

[72] T. Ball and J.R. Larus, "Optimally profiling and tracing programs," in *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages* (Albuquerque, NM), pp. 59–70, January 1992.

[73] R.A. Ballance, J. Butcher, and S.L. Graham, "Grammatical abstraction and incremental syntax analysis in a language-based editor," in *Proceedings of the ACM SIGPLAN '88 Conference on Progmmming Language Design and Implementation,* vol. 23 (Atlanta, GA), pp. 185–198, June 1988.

[74] J.P. Banning, "An efficient way to find the side effects of procedure calls and aliases of variables," in *Conference Record of the 6th Annual ACM Symposium on Principles of Programming Languages,* pp. 29–41, 1979.

[75] J.M. Barth, "An interprocedural data flow analysis algorithm," in *Conference Record of the 4th ACM Symposium on Principles of Programming Languages* (Los Angeles, CA), pp. 119–131, January 1977.

[76] J.M. Barth, "Shifting garbage collection overhead to compile time," *Communications of the ACM*, vol. 20, pp. 513–518, July 1977.

[77] J.M. Barth, "A practical interprocedural data flow analysis algorithm," *Journal of the ACM*, vol. 21, pp. 724–736, September 1978.

[78] D.H. Bartley, "Optimizing stack frame accesses for processors with restricted addressing modes," vol. 22, pp. 101–110, February 1992.

[79] F. Baskett, "The best simple code generation technique for WHILE, FOR, and DO loops," *SIGPLAN Notices,* vol. 13, pp. 31–32, April 1978.

[80] J. Bates and A. Lavie, "Recognizing substrings of LR(k) languages in linear time," in *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages* (Albuquerque, NM), pp. 235–245, January 1992.

[81] A. Batson, "The organization of symbol tables," *Communications of the ACM,* vol. 8, no. 2, pp. 111–112, 1965.

[82] W. Baxter and H.R.B. III, "The program dependence graph and vectorization," in *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages* (Austin, TX), pp. 1–11, January 1989.

[83] F. Bazzichi and I. Spadafora, "An automatic generator for compiler testing," *IEEE Transactions on Software Engineering*, vol. 8, pp. 343–353, July 1982.

[84] S.J. Beaty, M.R. Duda, R.A. Mueller, P.H. Sweany, and J. Varghese, "Optimization issues for a retargetable microcode compiler," *MicroArchitecture,* vol. 3, pp. 5–15, December 1988.

[85] J. Beney and J.F. Boulicaut, "STARLET: An affix-based compiler compiler designed as a logic programming system," in *Compiler Compilers 3rd International Workshop, CC '90* (Schwerin, FRG), pp. 71–85, October 1990.

[86] M.E. Benitez and J.W. Davidson, "A portable global optimizer and linker," in *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation,* vol. 23 (Atlanta, GA), pp. 329–338, June 1988.

[87] J.-F. Bergeretti and B.A. Carré, "Information-flow and data-flow analysis of while-programs," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 37–61, January 1985.

[88] M.E. Bermudez and K.M. Schimpf, "A practical arbitrary look-ahead LR parsing technique," in *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction,* vol. 21 (Palo Alto, CA), pp. 136–144, June 1986.

[89] M.E. Bermudez and K.M. Schimpf, "On the (non-) relationship between SLR(l) and NQLALR(l) grammars," *ACM Transactions on Programming Languages and Systems,* vol. 10, pp. 338–342, April 1988.

[90] R.L. Bernstein, "Producing good code for the case statement," vol. 15, pp. 1021–1024, October 1985.

[91] D. Bernstein, M.C. Golumbic, Y. Mansour, R.Y. Pinter, D.Q. Goldin, H. Krawczyk, and I. Nahshon, "Spill code minimization techniques for optimizing compilers," in *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation,* vol. 24 (Portland, OR), pp. 258–263, June 1989.

[92] G. Berry and J.J. Levy, "Minimal and optimal computations of recursive programs," in *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pp. 215–226, January 1977.

[93] R.E. Berry, "Experience with the Pascal P-compiler," vol. 8, pp. 617–627, 1978.

[94] P.L. Bird, "An implementation of a code generator specification language for table driven code generators," in *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction,* vol. 17, pp. 44–55, June 1982.

[95] M.P. Bivens and M.L. Soffa, "Incremental register reallocation," vol. 20, pp. 1015–1047, October 1990.

[96] J. Bodwin, L. Bradley, K. Kanda, D. Litle, and U. Pleban, "Experience with an experimental compiler generator based on denotational semantics," in *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction,* vol. 17, pp. 216–229, June 1982.

[97] H.-J. Boehm, "A logic for expressions with side-effects," in *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages,* pp. 268–280, January 1982.

[98] H.-J. Boehm, "Type inference in the presence of type abstraction," in *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation,* vol. 24 (Portland, OR), pp. 192–206, June 1989.

[99] H.J. Boom, "A weaker precondition for loops," *ACM Transactions on Programming Languages and Systems,* vol. 4, pp. 668–677, October 1982.

[100] A.H. Borning and D.H.H. Ingalls, "A type declaration and inference system for Smalltalk," in *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages,* pp. 133–141, January 1982.

[101] D.G. Bradlee, S.J. Eggers, and R.R. Henry, "Integrating register allocation and instruction scheduling for RISCs," in *4th International Conference on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, CA), pp. 122–131, April 1991.

[102] P. Briggs, K.D. Cooper, K. Kennedy, and L. Torczon, "Coloring heuristics for register allocation," in *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, vol. 24 (Portland, OR), pp. 275–284, June 1989.

[103] P. Briggs, K.D. Cooper, M.W. Hall, and L. Torczon, "Goal-directed interprocedural optimization," Technical Report Rice COMP TR90-147, Department of Computer Science, Rice University, 1990.

[104] P. Briggs, "Register allocation via graph coloring," Ph.D. Thesis Rice COMP TR92-183, Department of Computer Science, Rice University, 1992.

[105] G. Brooks, G.J. Hansen, and S. Simmons, "A new approach to debugging optimized code," in *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation,* vol. 27 (San Francisco, CA), pp. 1–11, June 1992.

[106] J. Bruno and T. Lassagne, "The generation of optimal code for stack machines," *Journal of the ACM*, vol. 22, no. 3, pp. 382–396, 1975.

[107] J. Bruno and R. Sethi, "Code generation for a one-register machine," *Journal of the ACM,* vol. 23, no. 3, pp. 502–510, 1976.

[108] T.A. Budd, "An APL compiler for a vector processor," *ACM Transactions on Programming Languages and Systems,* vol. 6, pp. 297–313, July 1984.

[109] M.G. Burke and G.A. Fisher, "A practical method for syntactic error diagnosis and recovery," in *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction,* vol. 17, pp. 67–78, June 1982.

[110] M. Burke and R. Cytron, "Interprocedural dependence analysis and parallelization," in *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction,* vol. 21 (Palo Alto, CA), pp. 162–175, June 1986.

[111] M.G. Burke and G.A. Fisher, "A practical method for LR and LL syntactic error diagnosis," *ACM Transactions on Programming Languages and Systems,* vol. 9, pp. 164–197, April 1987.

[112] M. Burke, "An interval-based approach to exhaustive and incremental interprocedural data-flow analysis," *ACM Transactions on Programming Languages and Systems,* vol. 12, pp. 341–395, July 1990.

[113] M. Burke and J.-D. Choi, "Precise and efficient integration of interprocedural alias information into data flow analysis," *ACM Letters on Programming Languages and Systems,* vol. 1, pp. 14–21, March 1992.

[114] F.W. Burton, "Type extension through polymorphism," *ACM Transactions on Programming Languages and Systems,* vol. 12, pp. 135–138, January 1990.

[115] V.A. Busam and D.E. Englund, "Optimization of expressions in Fortran," *Communications of the ACM,* vol. 12, pp. 666–674, December 1969.

[116] D. Callahan, K.D. Cooper, K. Kennedy, and L. Torczon, "Interprocedural constant propagation," in *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction,* vol. 21 (Palo Alto, CA), pp. 152–161, June 1986.

[117] D. Callahan, "The program summary graph and flow-sensitive interprocedural data flow analysis," in *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation,* vol. 23 (Atlanta, GA), pp. 47–56, June 1988.

[118] D. Callahan, S. Carr, and K. Kennedy, "Improving register allocation for subscripted variables," in *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation,* vol. 25 (White Plains, NY), pp. 53–65, June 1990.

[119] D. Callahan and B. Koblenz, "Register allocation via hierarchical graph coloring," in *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation,* vol. 26 (Toronto, Ontario, Canada), pp. 192–203, June 1991.

[120] L. Cardelli, "Structural subtyping and the notion of power type," in *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages* (San Diego, CA), pp. 70–79, January 1988.

[121] M.D. Carroll and B.G. Ryder, "Incremental data flow analysis via dominator and attribute updates," in *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages* (San Diego, CA), pp. 274–284, January 1988.

[122] R. Cartwright and M. Felleisen, "The semantics of program dependence," in *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation,* vol. 24 (Portland, OR), pp. 13–27, June 1989.

[123] R.G.G. Cattell, J.M. Newcomer, and B.W. Leverett, "Code generation in a machine-independent compiler," in *Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction,* pp. 173–190, August 1979.

[124] R.G.G. Cattell, "Automatic derivation of code generators from machine descriptions," *ACM Transactions on Programming Languages and Systems,* vol. 2, pp. 173–190, April 1980.

[125] A. Celentano, S.C. Reghizzi, P.D. Vigna, and C. Ghezzi, "Compiler testing using a sentence generator," vol. 10, pp. 897–918, 1980.

[126] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markstein, "Register allocation via coloring," *Computer Languages,* vol. 6, pp. 47–57, 1981.

[127] G.J. Chaitin, "Register allocation and spilling via graph coloring," in *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction,* vol. 17, pp. 201–207, June 1982.

[128] C. Chambers and D. Ungar, "Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language," in *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation,* vol. 24 (Portland, OR), pp. 146–160, June 1989.

[129] P.P. Chang, S.A. Mahlke, and W.W. Hwu, "Using profile information to assist classic code optimizations," vol. 21, pp. 1301–1321, December 1991.

[130] D.R. Chase, "An improvement to bottom-up tree pattern matching," in *Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages* (Munich, West Germany), pp. 168–177, January 1987.

[131] D.R. Chase, "Garbage collection and other optimizations," Technical Report, Department of Computer Science, Rice University, August 1987.

[132] D.R. Chase, "Safety considerations for storage allocation optimizations," in *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation,* vol. 23 (Atlanta, GA), pp. 1–10, June 1988.

[133] W.Y. Chen, P.P. Chang, T.M. Conte, and W.W. Hwu, "The effect of code expanding optimizations on instruction cache design," Technical Report CRHC-91-17, Center for Reliable and High-Performance Computing, University of Illinois, Urbana-Champaign, May 1991.

[134] C.-H. Chi and H. Dietz, "Unified management of registers and cache using liveness and cache bypass," in *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation,* vol. 24 (Portland, OR), pp. 344–355, June 1989.

[135] J.-D. Choi, R. Cytron, and J. Ferrante, "Automatic construction of sparse data flow evaluation graphs," in *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages* (Orlando, FL), pp. 55–66, January 1991.

[136] A.L. Chow and A. Rudmik, "The design of a data flow analyzer," in *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction,* vol. 17 (Boston, MA), pp. 106–113, June 1982.

[137] F.C. Chow, "A portable machine-independent global optimizer—design and measurements," Technical Report TR 83–254, Stanford University, December 1983.

[138] F.C. Chow and J.L. Hennessy, "The priority-based coloring approach to register allocation," *ACM Transactions on Programming Languages and Systems,* vol. 12, pp. 501–536, October 1990.

[139] F.C. Chow, "Minimizing register usage penalty at procedure calls," in *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation,* vol. 23 (Atlanta, GA), pp. 85–94, June 1988.

[140] T.W. Christopher, P.J. Hatcher, and R.C. Kukuk, "Using dynamic programming to generate optimized code in a Graham-Glanville style code generator," in *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction,* vol. 19, pp. 25–36, June, 1984.

[141] J. Cocke, "Global common subexpression elimination," in *Proceedings of a Symposium on Compiler Optimization,* vol. 5, pp. 20–24, July 1970.

[142] J. Cocke and K. Kennedy, "An algorithm for reduction of operator strength," *Communications of the ACM*, vol. 20, pp. 850–856, November 1977.

[143] J. Cocke and P. Markstein, "Measurement of code improvement algorithms," *Information Processing 80,* pp. 221–228, 1980.

[144] J. Cohen and T.J. Hickey, "Parsing and compiling using Prolog," *ACM Transactions on Programming Languages and Systems,* vol. 9, pp. 125–163, April 1987.

[145] R. Cohn, T. Gross, M. Lam, and P.S. Tseng, "Architecture and compiler trade-offs for a long instruction word microprocessor," in *3rd International Conference on Architectural Support for Programming Languages and Operating Systems,* vol. 24, (Boston, MA), pp. 2–14, April 1989.

[146] C. Consel and O. Danvy, "Static and dynamic semantics processing," in *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages* (Orlando, FL), pp. 14–24, January 1991.

[147] C. Consel and S.C. Khoo, "Parameterized partial evaluation," in *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation,* vol. 26 (Toronto, Ontario, Canada), pp. 92–106, June 1991.

[148] K.D. Cooper, "Analyzing aliases of reference formal parameters," in *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages* (New Orleans, LA), pp. 281–290, January 1985.

[149] K.D. Cooper, K. Kennedy, and L. Torczon, "Interprocedural optimization: eliminating unnecessary recompilation," in *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction,* vol. 21 (Palo Alto, CA), pp. 58–67, June 1986.

[150] K.D. Cooper and K. Kennedy, "Interprocedural side-effect analysis in linear time," in *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation,* vol. 23 (Atlanta, GA), pp. 57–66, June 1988.

[151] B.G. Cooper, "Ambitious data flow analysis of procedural programs," Master's Thesis, University of Minnesota, May 1989.

[152] J.R. Cordy and R.C. Holt, "Code generation using an orthogonal model," vol. 20, pp. 301–320, March 1990.

[153] G.V. Cormack, "An LR substring parser for noncorrecting syntax error recovery," in *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation,* vol. 24, (Portland, OR), pp. 161–169, June 1989.

[154] G.V. Cormack and A.K. Wright, "Type-dependent parameter inference," in *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation,* vol. 25, (White Plains, NY), pp. 127–136, June 1990.

[155] D.D. Cowan and J.W. Graham, "Design characteristics of the WATFOR compiler," in *Proceedings of a Symposium on Compiler Optimization,* vol. 5, pp. 25–36, July 1970.

[156] E. Crank and M. Felleisen, "Parameter-passing and the lambda calculus," in *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages,* (Orlando, FL), pp. 233–244, January 1991.

[157] A. Critcher, "The functional power of parameter passage mechanisms," in *Conference Record of the 6th Annual ACM Symposium on Principles of Programming Languages,* pp. 158–168, January 1979.

[158] K. Culik, "Towards a theory of control-flow and data-flow algorithms," in *Proceedings of the ICPP,* pp. 341–348, 1985.

[159] R. Cytron, A. Lowry, and F.K. Zadeck, "Code motion of control structures in high-level languages," in *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages* (St. Petersburg, FL), pp. 70–85, January 1986.

[160] R. Cytron, M. Hind, and W. Hsieh, "Automatic generation of DAG parallelism," in *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation,* vol. 24 (Portland, OR), pp. 54–68, June 1989.

[161] J.W. Davidson and C.W. Fraser, "The design and application of a retargetable peephole optimizer," *ACM Transactions on Programming Languages and Systems,* vol. 2, pp. 191–202, April 1980.

[162] J.W. Davidson and C.W. Fraser, "Register allocation and exhaustive peephole optimization," vol. 14, pp. 857–865, September 1984.

[163] J.W. Davidson and C.W. Fraser, "Eliminating redundant object code," in *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages,* (Albuquerque, NM), pp. 129–132, January 1982.

[164] J.W. Davidson and C.W. Fraser, "Automatic generation of peephole optimizations," in *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction,* vol. 19, pp. 111–116, June 1984.

[165] J.W. Davidson and C.W. Fraser, "Code selection through object code optimization," *ACM Transactions on Programming Languages and Systems,* vol. 6, pp. 505–526, October 1984.

[166] J.W. Davidson, "A retargetable instruction reorganizer," in *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction,* vol. 21 (Palo Alto, CA), pp. 234–241, June 1986.

[167] J.W. Davidson and C.W. Fraser, "Automatic Inference and Fast Interpretation of Peephole Optimization Rules," vol. 17, pp. 801–812, November 1987.

[168] J.W. Davidson and D.B. Whalley, "Quick compilers using peephole optimization," vol. 19, pp. 79–97, January 1989.

[169] J.W. Davidson and D.B. Whalley, "Reducing the cost of branches by using registers," in *Proceedings of the 17th Annual International Symposium on Computer Architecture,* (Los Alamitos, CA), pp. 182–191, May 1990.

[170] J.W. Davidson and D.B. Whalley, "Methods for saving and restoring register values across function calls," vol. 21, pp. 149–165, February 1991.

[171] A.L. Davis and R.M. Keller, "Data flow program graphs," *IEEE Computer,* vol. 15, February 1982.

[172] S.K. Debray, "Unfold/fold transformations and loop optimization of logic programs," in *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation,* vol. 23, (Atlanta, GA), pp. 297–307, June 1988.

[173] P. Degano, S. Mannucci, and B. Mojana, "Efficient incremental LR parsing for syntax-directed editors," *ACM Transactions on Programming Languages and Systems,* vol. 10, pp. 345–373, July 1988.

[174] A.J. Demers, "Generalized left corner parsing," in *Conference Record of the 4th ACM Symposium on Principles of Programming Languages,* pp. 170–182, January 1977.

[175] A.J. Demers, J.E. Donahue, and G. Skinner, "Data types as values: Polymorphism, type-checking, encapsulation," in *Conference Record of the 5th Annual ACM Symposium on Principles of Programming Languages,* pp. 23–30, January 1978.

[176] A.J. Demers and J.E. Donahue, "Data types, parameters, and type checking," in *Conference Record of the 7th Annual ACM Symposium on Principles of Programming Languages,* pp. 12–23, January 1980.

[177] A.J. Demers, T. Reps, and T. Teitelbaum, "Incremental evaluation for attribute grammars with application to syntax-directed editors," in *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages,* pp. 105–116, January 1981.

[178] P. Dencker, K. Durre, and J. Heuft, "Optimization of parser tables for portable compilers," *ACM Transactions on Programming Languages and Systems,* vol. 6, no. 4, pp. 546–572, 1984.

[179] F. DeRemer and T.J. Pennello, "Efficient computation of LALR(1) look-ahead sets," in *Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction,* vol. 14, pp. 176–187, August 1979.

[180] F. DeRemer and T. Pennello, "Efficient computation of LALR (1) look-ahead sets," *ACM Transactions on Programming Languages and Systems,* vol. 4, no. 4, pp. 615–649, 1982.

[181] D.M. Dhamdhere and J.S. Keith, "Characterization of program loops in code optimization," *Computer Languages,* vol. 8, pp. 69–76, 1983.

[182] D.M. Dhamdhere, "A fast algorithm for code movement optimization," *SIGPLAN Notices,* vol. 23, no. 10, pp. 172–180, 1988.

[183] D.M. Dhamdhere, "Register assignment using code placement techniques," *Computer Languages,* vol. 13, no. 2, pp. 75–93, 1988.

[184] D.M. Dhamdhere, "A usually linear algorithm for register assignment using edge placement of load and store instructions," *Computer Languages,* vol. 15, no. 2, pp. 83–94, 1990.

[185] D.M. Dhamdhere, "Practical adaptation of the global optimization algorithm of Morel and Renvoise," *ACM Transactions on P~mming Languages and Systems,* vol. 13, pp. 291–294, April 1991.

[186] A. Diwan, E. Moss, and R. Hudson, "Compiler support for garbage collection in a statically typed language," in *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation,* vol. 27 (San Francisco, CA), pp. 273–282, June 1992.

[187] H. Dobler and K. Pirkbauer, "Coco-2 a new compiler compiler," *SIGPLAN Notices*, vol. 25, pp. 82–90, May 1990.

[188] M.K. Donegan, R.E. Noonan, and S. Feyock, "A code generator language," *SIGPLAN Notices,* vol. 14, pp. 58–64, August 1979.

[189] P.J. Downey, R. Sethi, and R.E. Tarjan, "Variations on the common subexpression problem," *Journal of the ACM,* vol. 27 no. 4, pp. 758–771, 1980.

[190] P.K. Dubey and MJ. Flynn, "Branch strategies: modeling and optimization," *IEEE Transaction on Computers,* vol. 40, pp. 1159–1167, October 1991.

[191] K. Ebcioglu, "A compilation technique for software pipelining of loops with conditional jumps," in *Proceedings of the 20th Annual Workshop on Microprogramming* (Colorado Springs, CO), pp. 69–79, December 1987.

[192] E.F. Elsworth and M.A.B. Parkes, "Automated compiler construction based on top-down syntax analysis and attribute evaluation," *SIGPLAN Notices,* vol. 25, pp. 37–42, August 1900.

[193] J. Fabri, "Automatic storage optimization," in *Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction,* vol. 14, pp. 83–91, August 1979.

[194] R. Farrow, K. Kennedy, and L. Zucconi, "Graph grammars and global program flow analysis," in *Proceedings of the 17th Annual IEEE Symposium on Foundations of Computer Science,* November 1975.

[195] R. Farrow, "Linguist-86: yet another translator writing system based on attribute grammars," in *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction,* vol. 17, pp. 160–171, June 1982.

[196] R. Farrow, "Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars," in *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction,* vol. 21 (Palo Alto, CA), pp. 85–98, June 1986.

[197] S.I. Feldman, "Implementation of a portable Fortan 77 compiler using modern tools," in *Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction,* vol. 14, pp. 98–106, August 1979.

[198] J. Ferrante, K.J. Ottenstein, and J.D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems,* vol. 9, pp. 319–349, July 1987.

[199] C.N. Fischer, D.R. Milton, and S.B. Quiring, "An efficient insertion-only error-corrector for LL(1) parsers," in *Conference Record of the 4th ACM Symposium on Principles of Programming Languages,* pp. 97–103, January 1977.

[200] C.N. Fischer, D.R. Milton, and J. Mauney, "A locally least-cost LL(1) error-corrector," Technical Report Tech. Report 371, University of Wisconsin-Madison, 1979.

[201] C.N. Fischer, D.R. Milton, and S.B. Quiring, "Efficient LL(1) error correction and recovery using only insertions," *Acta Informatica,* vol. 13, no. 2, pp. 141–154, 1980.

[202] C.N. Fischer and J. Mauney, "A simple, fast, and effective LL(1) error repair algorithm," Technical Report CSTR 901, Computer Sciences Department, University of Wisconsin–Madison, 1989.

[203] R.W. Floyd, "An algorithm for coding efficient arithmetic operations," *Communications of the ACM,* vol. 4, pp. 42–51, January 1961.

[204] A.C. Fong, J. Kam, and J.D. Ullman, "Application of lattice algebra to loop optimization," in *Conference Record of the 2nd ACM Symposium on Principles of Programming Languages,* pp. 1–9, January 1975.

[205] A.C. Fong and J.D. Ullman, "Induction variables in very high level languages," in *Conference Record of the 3rd ACM Symposium on Principles of Programming Languages,* pp. 104–112, January 1976.

[206] A.C. Fong, "Generalized common subexpressions in very high level languages," in *Conference Record of the 4th ACM Symposium on Principles of Programming Languages,* pp. 48–57, January 1977.

[207] A.C. Fong, "Automatic improvement of programs in very high level languages," in *Conference Record of the 6th Annual ACM Symposium on Principles of Programming Languages,* pp. 21–28, January 1979.

[208] R. Ford and D. Sawamiphakdi, "A greedy concurrent approach to incremental code generation," in *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages,* pp. 165–178, January 1985.

[209] D.J. Frailey, "Expression optimization using unary complement operators," in *Proceedings of a Symposium on Compiler Optimization*, vol. 5, pp. 67–85, July 1970.

[210] D.J. Frailey, "An intermediate language for source and target independent code optimizations," in *Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction,* vol. 14, pp. 188–200, August 1979.

[211] Y. Frantsuzov, "Code scheduling with delayed register allocation," *Programmirovanie,* pp. 58–66, 1991.

[212] C.W. Fraser, "A compact, machine-independent peephole optimizer," in *Conference Record of the 6th Annual ACM Symposium on Principles of Programming Languages,* (San Antonio, TX), pp. 1–6, January 1979.

[213] C.W. Fraser and A.L. Wendt, "Integrating code generation and optimization," in *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction,* vol. 21, (Palo Alto, CA), pp. 242–248, June 1986.

[214] C.W. Fraser and A.L. Wendt, "Automatic generation of fast optimizing code generators," in *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation,* vol. 23 (Atlanta, GA), pp. 79–84, June 1988.

[215] C.W. Fraser, "A language for writing code generators," in *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation,* vol. 24 (Portland, OR), pp. 238–245, June 1989.

[216] C.W. Fraser and R.R. Henry, "Hard-coding bottom-up code generation tables to save time and space," vol. 21, pp. 1–12, January 1991.

[217] C.W. Fraser and D.R. Hanson, "A retargetable compiler for ANSI C," *SIGPLAN Notices,* vol. 26, pp. 29–43, October 1991.

[218] R.A. Freiburghouse, "Register allocation via usage counts," *Communications of the ACM,* vol. 17, pp. 638–642, November 1974.

[219] C.K. Gomard and N.D. Jones, "Compiler generation by partial evaluation: a case study," *Structured Programming,* vol. 12, no. 3, pp. 123–144, 1991.

[220] M. Ganapathi and C.N. Fischer, "Description-driven code generation using attribute grammars," in *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages* (Albuquerque, NM), pp. 108–119, January 1982.

[221] M. Ganapathi, C.N. Fischer, and J.L. Hennessy, "Retargetable compiler code generation," *ACM Computing Surveys,* vol. 14, pp. 573–592, December 1982.

[222] M. Ganapathi and C.N. Fischer, "Affix grammar driven code generation," *ACM Transactions on Programming Languages and Systems,* vol. 7, pp. 560–599, October 1985.

[223] H. Ganzinger, R. Giegerich, U. Möncke, and R. Wilhelm, "A truly generative semantics directed compiler generator," in *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction,* vol. 17, pp. 172–184, June 1982.

[224] M.R. Garey and D.S. Johnson, "The complexity of near-optimal graph coloring," *Journal of the ACM,* vol. 4, pp. 397–411, 1976.

[225] C.W. Gear, "High speed compilation of efficient object code," *Communications of the ACM,* vol. 8, pp. 483–488, August 1965.

[226] M.M. Geller and M.A. Harrison, "Strict deterministic versus LR(0) parsing," in *Conference Record of the ACM Symposium on Principles of Programming Languages,* pp. 22–32, October 1973.

[227] C. Ghezzi and D. Mandrioli, "Incremental parsing," *ACM Transactions on Programming Languages and Systems,* vol. 1, pp. 48–70, July 1979.

[228] R. Giegerich, "A formal framework for the derivation of machine-specific optimizers," *ACM Transactions on Programming Languages and Systems,* vol. 5, no. 3, pp. 478–498, 1983.

[229] R.S. Glanville and S.L. Graham, "A new method for compiler code generation," in *Conference Record of the 5th Annual ACM Symposium on Principles of Programming Languages,* (Tucson, AZ), pp. 231–240, January 1978.

[230] M.C. Golumbic and V. Rainish, "Instruction scheduling beyond basic blocks," *IBM Journal of Research and Development,* vol. 34, pp. 93–97, January 1990.

[231] C.K. Gomard and P. Sestoft, "Globalization and live variables," in *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation,* vol. 26 (New Haven, CN), pp. 166–177, June 1991.

[232] J.R. Goodman and W.-C. Hsu, "Code scheduling and register allocation in large basic blocks," in *Conference Proceedings 1988 International Conference on Supercomputing* (St. Malo, France), pp. 442–452, July 1988.

[233] S.L. Graham and S.P. Rhodes, "Practical syntactic error recovery," in *Conference Record of the ACM Symposium on Principles of Programming Languages,* pp. 52–58, October 1973.

[234] S.L. Graham and M. Wegman, "A fast and usually linear algorithm for global flow analysis," in *Conference Record of the 2nd ACM Symposium on Principles of Programming Languages,* pp. 22–34, January 1975.

[235] S.L. Graham, C.B. Haley, and W.N. Joy, "Practical LR error recovery," in *Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction,* vol. 14, pp. 168–175, August 1979.

[236] S.L. Graham, M.A. Harrison, and W.L. Ruzzo, "An improved context-free recognizer," *ACM Transactions on Programming Languages and Systems,* vol. 2, pp. 415–462, July 1980.

[237] S.L. Graham, "Table-driven code generation," *IEEE Computer,* pp. 25–34, August 1980.

[238] J.O. Graver, "The evolution of an object-oriented compiler framework," vol. 22, pp. 519–535, July 1992.

[239] P. Grogono and A. Bennett, "Polymorphism and type checking in object-oriented languages," *SIGPLAN Notices,* vol. 24, pp. 109–115, November 1989.

[240] J. Grosch and H. Emmelmann, "A tool box for compiler construction," in *Compiler Compilers 3rd International Workshop, CC '90,* (Schwerin, FRG), pp. 106–116, October 1990.

[241] J. Grosch, "Efficient and comfortable error recovery in recursive descent parsers," *Structured Programming,* vol. 11, no. 3, pp. 129–140, 1990.

[242] T. Gross and P.A. Steenkiste, "Structured data-flow analysis for arrays and its use in an optimizing compiler," vol. 20, pp. 133–155, February 1990.

[243] R. Gupta, M.L. Soffa, and T. Steele, "Register allocation via clique separators," in *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation,* vol. 24 (Portland, OR), pp. 264–274, June 1989.

[244] R. Gupta, "A fresh look at optimizing array bound checking," in *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation,* vol. 25 (White Plains, NY), pp. 272–282, June 1990.

[245] R. Gupta, "Generalized dominators and post-dominators," in *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages* (Albuquerque, NM), pp. 246–257, January 1992.

[246] B.K. Haddon and W.M. Waite, "Experience with the universal intermediate language Janus," vol. 8, pp. 601–616, 1978.

[247] M.W. Hall, "Managing interprocedural optimization," Technical Report Rice COMP TR91–157, Department of Computer Science, Rice University, 1991.

[248] R.C. Hansen, "New optimizations for PA–RISC compilers," *Hewlett-Packard Journal,* pp. 15–23, June 1992.

[249] D.R. Hanson, "Simple code optimizations," vol. 13, pp. 745–763, 1983.

[250] D. Harel, "A linear time algorithm for finding dominators in flow graphs and related problems," in *Proceedings of the 17th ACM Symposium on Theory of Computing,* pp. 185–194, May 1985.

[251] W. Harrison, "A new strategy for code generation—the general purpose optimizing compiler," in *Conference Record of the 4th ACM Symposium on Principles of Programming, Languages* (Los Angeles, CA), pp. 29–37, January 1977.

[252] P.J. Hatcher and T.W. Christopher, "High-quality code generation via bottom-up tree pattern matching," in *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages* (St. Petersburg, FL), pp. 119–130, January 1986.

[253] M.S. Hecht and J.D. Ullman, "Flow graph reducibility," *SIAM J. Computing,* vol. 1, pp. 188–202, June 1972.

[254] M.S. Hecht and J.D. Ullman, "Characteristics of reducible flow graphs," *Journal of the ACM,* vol. 21, pp. 367–375, July 1974.

[255] M.S. Hecht and J.D. Ullman, "A simple algorithm for global data flow analysis problems," *SIAM J. Computing,* vol. 4, pp. 519–532, December 1975.

[256] J. Heering, P. Klint, and J. Rekers, "Incremental generation of parsers," in *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation,* vol. 24 (Portland, OR), pp. 179–191, June 1989.

[257] J.L. Hennessy, "Program optimization and exception handling," in *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages,* pp. 200–206, January 1981.

[258] J.L. Hennessy, "Symbolic debugging of optimized code," *ACM Transactions on Programming Languages and Systems,* vol. 4, pp. 323–344, July 1982.

[259] R.R. Henry and P.C. Damron, "Algorithms for table-driven code generators using tree-pattern matching," Technical Report 89-02-03, University of Washington, Department of Computer Science, February 1989.

[260] R.R. Henry, "Encoding optimal pattern selection in a table-driven bottum-up tree-pattern matcher," Technical Report 89-02-04, University of Washington, Department of Computer Science, February 1989.

[261] L.H. Holley and B.K. Rosen, "Qualified data flow problems," in *Conference Record of the 7th Annual ACM Symposium on Principles of Programming Languages,* pp. 68–82, January 1980.

[262] R.N. Horspool, "ILALR: An incremental generator of LALR(1) parsers," in *Compiler Compilers and High Speed Compilation 2nd CCHSC Workshop Proceedings* (Berlin, GDR), pp. 128–136, October 1988.

[263] R.N. Horspool and M. Whitney, "Even faster LR parsing," vol. 20, pp. 515–535, June 1990.

[264] R.N. Horspool, "Recursive ascent-descent parsers," in *Compiler Compilers 3rd International Workshop, CC '90* (Schwerin, FRG), pp. 1–10, October 1990.

[265] W.-C. Hsu, C.N. Fischer, and J.R. Goodman, "On the minimization of loads and stores in local register allocation," *IEEE Transactions on Software Engineering,* vol. 15, pp. 1252–1260, October 1989.

[266] H.B.H. III, T.G. Szymanski, and J.D. Ullman, "On the complexity of LR(k) testing," in *Conference Record of the 2nd ACM Symposium on Principles of Programming Languages,* pp. 130–136, January 1975.

[267] F. Ives, "Unifying view of recent LALR(1) lookahead set algorithms," in *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction,* vol. 21 (Palo Alto, CA), pp. 131–135, June 1986.

[268] S. Jain and C. Thompson, "An efficient approach to data flow analysis in a multiple pass global optimizer," in *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation,* vol. 23 (Atlanta, GA), pp. 154–163, June 1988.

[269] F. Jalili and J.H. Gallier, "Building friendly parsers," in *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages,* pp. 196–206, January 1981.

[270] U. Joerring and W.L. Scherlis, "Compilers and staging transformations," in *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages* (St. Petersburg, FL), pp. 86–96, January 1986.

[271] R.K. Johnson, "An approach to global register allocation," Ph.D. thesis, Carnegie-Mellon University, December 1975.

[272] S.C. Johnson, "A portable compiler: theory and practice," in *Conference Record of the 5th Annual ACM Symposium on Principles of Programming Languages* (Tuscon, AZ), pp. 97–104, January 1978.

[273] C.W. Johnson and C. Runciman, "Semantic errors—diagnosis and repair," in *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction,* vol. 17, pp. 88–97, June 1982.

[274] M.S. Johnson and T.C. Miller, "Effectiveness of a machine-level global optimizer," in *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction,* vol. 21 (Palo Alto, CA), pp. 99–108, June 1986.

[275] S.M. Joshi and D.M. Dhamdhere, "A composite hoisting-strength reduction transformation for global program optimization part I," *International Journal of Computer Mathematics,* vol. 11, pp. 21–41, 1982.

[276] S.M. Joshi and D.M. Dhamdhere, "A composite hoisting-strength reduction transformation for global program optimization part II," *International Journal of Computer Mathematics,* vol. 11, pp. 111–126, 1982.

[277] J.B. Kam and J.D. Ullman, "Global data flow analysis and iterative algorithms," *Journal of the ACM,* vol. 23, no. 1, pp. 158–171, 1976.

[278] J.B. Kam and J.D. Ullman, "Monotone data flow analysis frameworks," *Acta Informatica,* vol. 7, pp. 305–317, 1977.

[279] M.A. Kaplan and J.D. Ullman, "A general scheme for the automatic inference of variable types," in *Conference Record of the 5th Annual ACM Symposium on Principles of Programming Languages,* pp. 60–75, January 1978.

[280] M. Karr, "Code generation by coagulation," in *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction,* vol. 19, pp. 1–12, June 1984.

[281] W. Keller, "Automated generation of code using backtracking parsers for attribute grammars," *SIGPLAN Notices,* vol. 26, pp. 109–117, February 1991.

[282] R. Kelsey and P. Hudak, "Realistic compilation by program transformation," in *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages* (Austin, TX), pp. 281–292, January 1989.

[283] K. Kennedy and P. Owens, "An algorithm for use-definition chaining," Technical Report SETL Newsletter #37, Courant Inst. of Math. Sciences, New York University, New York, July 1971.

[284] K. Kennedy, "Safety of code motion," *International Journal of Computer Mathematics,* vol. 3, pp. 117–130, 1972.

[285] K. Kennedy, "Reduction in strength using hashed temporaries," Technical Report SETL Newsletter #102, Courant Inst. of Math. Sciences, New York University, New York, March 1973.

[286] K. Kennedy, "Global dead computation elimination," Technical Report SETL Newsletter #111, Courant Inst. of Math. Sciences, New York University, New York, August 1973.

[287] K. Kennedy, "Node listings applied to data flow analysis," in *Conference Record of the 2nd ACM Symposium on Principles of Programming Languages,* pp. 10–21, January 1975.

[288] K. Kennedy, *A Survey of Data Flow Analysis Techniques.* Englewood Cliffs, NJ: Prentice-Hall, 1981.

[289] R.R. Kessler, "COG: An architectural-description-driven compiler generator," Ph.D thesis, Dept. of Computer Science, University of Utah, Salt Lake City, January 1981.

[290] R.R. Kessler, "Peep—an architectural description driven peephole optimizer," in *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction,* vol. 19, pp. 106–110, June 1984.

[291] P.B. Kessler, "Discovering machine specific code improvements," in *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction,* vol. 21, pp. 249–254, July 1986.

[292] G.A. Kildall, "A unified approach to global program optimization," in *Conference Record of the ACM Symposium on Principles of Programming Languages,* pp. 194–206, October 1973.

[293] P. Kornerup, B.B. Kristen, and O.L. Madsen, "Interpretation and code generation based on intermediate languages," vol. 10, pp. 635–658, August 1980.

[294] K. Koskimies, K.-J. Räihä, and M. Sarjakoski, "Compiler construction using attribute grammars," in *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction,* vol. 17, pp. 153–159, June 1982.

[295] B.B. Kristensen and O.L. Madsen, "Methods for computing LALR(k) lookahead," *ACM Transactions on Programming Languages and Systems,* vol. 3, pp. 60–82, January 1981.

[296] P. Kroha, "Code generation for a RISC machine," in *Compiler Compilers and High Speed Compilation 2nd CCHSC Workshop Proceedings* (Berlin, GDR), pp. 205–214, October 1988.

[297] D.W. Krumme and D.H. Ackley, "A practical method for code generation based on exhaustive search," in *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction,* vol. 17, pp. 185–196, June 1982.

[298] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe, "Dependence graphs, and compiler optimizations," in *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages,* pp. 207–218, January 1981.

[299] W.R. Lalonde, "On directly constructing LR(k) parsers without chain reductions," in *Conference Record of the 3rd ACM Symposium on Principles of Programming Languages,* pp. 127–133, January 1976.

[300] W.R. LaLonde and J. des Rivieres, "Handling operator precedence in arithmetic expressions with tree transformations," *ACM Transactions on Programming Languages and Systems,* vol. 3, pp. 83–103, January 1981.

[301] W.R. LaLonde, "The construction of stack-controlling LR parsers for regular right part grammars," *ACM Transactions on Programming Languages and Systems,* vol. 3, pp. 168–206, April 1981.

[302] D.A. Lamb, "Construction of a peephole optimizer," vol. 11, pp. 639–647, 1981.

[303] D.A. Lamb, "IDL: sharing intermediate representations," *ACM Transactions on Programming Languages and Systems,* vol. 9, pp. 297–318, July 1987.

[304] R. Landwehr, H.S. Jansohn, and G. Goos, "Experience with an automatic code generator generator," in *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction,* vol. 17, pp. 56–66, June 1982.

[305] J.M. Larcheveque, "Using an LALR compiler compiler to generate incremental parsers," in *Compiler Compilers 3rd International Workshop, CC '90* (Schwerin, FRG), pp. 147–164, October 1990.

[306] J.R. Larus and P.N. Hilfinger, "Register allocation in the SPUR Lisp compiler," in *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction,* vol. 21, (Palo Alto, CA), pp. 255–263, June 1986.

[307] P. Lee and U. Pleban, "A realistic compiler generator based on high-level semantics," in *Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages* (Munich, West Germany), pp. 284–295, January 1987.

[308] T. Lengauer and R.E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Transactions on Programming Languages and Systems,* vol. 1, pp. 121–141, July 1979.

[309] B.W. Leverett, "Register allocation in optimizing compilers," Ph.D. Thesis CMU-CS-81-103, Carnegie-Mellon University, Pittsburgh, PA, 1981.

[310] M.R. Levy, "Type checking in the large," in *Compiler Compilers and High Speed Compilation 2nd CCHSC Workshop Proceedings* (Berlin, GDR), pp. 137–145, October 1988.

[311] E. Pelegri-Llopart and S.L. Graham, "Optimal code generation for expression trees: An application of BURS theory," in *Conference Record of the 15th Annual ACM Symosium on Principles of Programming Languages* (San Diego, CA), pp. 294–308, January 1988.

[312] G. Logothetis and P. Mishra, "Compiling short-circuit boolean expressions in one pass," vol. 11, pp. 1197–1214, 1981.

[313] D. Lomet, "Data flow analysis in the presence of procedure calls," *IBM Journal of Research and Development,* vol. 21, pp. 559–571, November 1977.

[314] D.B. Loveman, "Program improvement by source to source transformation," in *Conference Record of the 3rd ACM Symposium on Principles of Programming Languages,* pp. 140–152, January 1976.

[315] E.S. Lowry and C.W. Medlock, "Object code optimization," *Communications of the ACM,* vol. 12, pp. 13–22, January 1969.

[316] L.-C. Lu, "A unified framework for systematic loop transformations," in *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles & Prentice of Parallel Programming,* vol. 26 (Williamsburg, VA), pp. 28–38, April 1991.

[317] B. Maher and D.H. Sleeman, "Automatic program improvement: variable usage transformations," *ACM Transactions on Programming Languages and Systems,* vol. 5, pp. 236–264, April 1983.

[318] V. Markstein, J. Cocke, and P. Markstein, "Optimization of range checking," in *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction,* vol. 17, pp. 114–119, June 1982.

[319] A. Martelli and U. Montari, "An efficient unification algorithm," *ACM Transactions on Programming Languages and Systems,* vol. 4, pp. 258–282, Apr. 1982.

[320] J. Mauney and C.N. Fischer, "A forward move algorithm for LL and LR parsers," in *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction,* vol. 17, pp. 79–87, June 1982.

[321] J. Mauney, "Least-cost error repair using extended right context," Ph.D. thesis, University of Wisconsin-Madison, 1983.

[322] J. Mauney and C.N. Fischer, "Determining the extent of lookahead in syntactic error repair," *ACM Transactions on Programming Languages and Systems,* vol. 10, pp. 456–469, July 1988.

[323] S. McFarling, "Program optimization for instruction caches," in *3rd International Conference on Architectural Support for Programming Languages and Operating Systems,* vol. 24 (Boston, MA), pp. 183–191, April 1989.

[324] W.M. McKeeman, "Peephole optimization," *Communications of the ACM,* vol. 8, pp. 443–444, July 1965.

[325] D.R. Milton, L.W. Kirchhoff, and B.R. Rowland, "An ALL(1) compiler generator," in *Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction,* vol. 14, pp. 152–157, August 1979.

[326] R.J. Mintz, G.A. Fischer, Jr., and M. Sharir, "The design of a global optimizer," in *Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction,* vol. 14, pp. 226–234, August 1979.

[327] P. Mishra and U.S. Reddy, "Declaration-free type checking," in *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages,* pp. 7–21, January 1985.

[328] J.C. Mitchell, "Coercion and type inference," in *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages,* pp. 175–185, January 1984.

[329] E. Morel and C. Renvoise, "Global optimization by suppression of partial redundancies," *Communications of the ACM,* vol. 22, pp. 96–103, February 1979.

[330] T.M. Morgan and L.A. Rowe, "Analyzing exotic instructions for a retargetable code generator," in *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction,* vol. 17 (Boston, MA), pp. 197–204, June 1982.

[331] W.G. Morris, "CCG: a prototype coagulating code generator," in *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation* vol. 26, (Toronto, Ontario, Canada), pp. 45–58, June 1991.

[332] R.A. Mueller and J. Varghese, "Retargetable microcode synthesis," *ACM Transactions on Programming Languages and Systems,* vol. 9, pp. 257–276, April 1987.

[333] F. Mueller and D.B. Whalley, "Avoiding unconditional jumps by code replication," in *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation,* vol. 27 (San Francisco, CA), pp. 322–330, June 1992.

[334] H. Mulder, "Data buffering: Run-time versus compile-time support," in *3rd International Conference on Architectural Support for Programming Languages and Operating Systems,* vol. 24 (Boston, MA), pp. 144–151, April 1989.

[335] E.W. Myers, "A precise interprocedural data flow algorithm," in *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages,* pp. 219–230, January 1981.

[336] R.E. Nather, "On the compilation of subscripted variables," *Communications of the ACM,* vol. 4, pp. 169–171, April 1961.

[337] B.R. Nickerson, "Graph coloring register allocation for processors with multi-register operands," in *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation,* vol. 25 (White Plains, NY), pp. 40–52, June 1990.

[338] A. Nijholt, "On the covering of left recursive grammars," in *Conference Record of the 4th ACM Symposium on Principles of Programming Languages,* pp. 86–96, January 1977.

[339] K.J. Ottenstein, "A simplified framework for reduction in strength," *IEEE Trans. Software Engineering,* vol. 15, no. 1, pp. 86–92, 1989.

[340] A.B. Pai and R.B. Kieburtz, "Global context recovery: a new strategy for syntactic error recovery by table-driven parsers," *ACM Transactions on Programming Languages and Systems,* vol. 2, pp. 18–41, January 1980.

[341] R. Paige and J.T. Schwartz, "Reduction in strength of high level operations," in *Conference Record of the 4th ACM Symposium on Principles of Programming Languages,* pp. 58–71, January 1977.

[342] J.C.H. Park, K.M. Choe, and C.H. Chang, "A new analysis of LALR formalisms," *ACM Transactions on Programming Languages and Systems,* vol. 7, pp. 159–175, January 1985.

[343] L. Paulson, "A semantics-directed compiler generator," in *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages,* pp. 224–233, January 1982.

[344] T.J. Pennello and F. DeRemer, "A forward move algorithm for LR error recovery," in *Conference Record of the 5th Annual ACM Symposium on Principles of Programming Languages,* pp. 241–254, January 1978.

[345] T.J. Pennello, "Very fast LR parsing," in *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction,* vol. 21 (Palo Alto, CA), pp. 145–151, June 1986.

[346] D.R. Perkins and R.L. Sites, "Machine-independent Pascal code optimization," in *Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction,* vol. 14, pp. 201–207, August 1979.

[347] P. Pfahler, "Optimizing directly executable LR parsers," in *Compiler Compilers 3rd International Workshop, CC '90* (Schwerin, FRG), pp. 179–192, October 1990.

[348] S.S. Pinter and R.Y. Pinter, "Program optimization and parallelization using idioms," in *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages* (Orlando, FL), pp. 79–92, January 1991.

[349] U.F. Pleban, "The use of transition matrices in a recursive-descent compiler," in *Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction,* vol. 14, pp. 144–151, August 1979.

[350] L.L. Pollock and M.L. Soffa, "Incremental compilation of locally optimized code," in *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages* (New Orleans, LA), pp. 152–164, January 1985.

[351] L.L. Pollock and M.L. Soffa, "An incremental version of iterative data flow analysis," *IEEE Trans. Software Engineering,* vol. 15, no. 12, pp. 1537–1549, 1989.

[352] L.L. Pollock and M.L. Soffa, "Incremental global optimization for faster recompilation," in *Proceedings of 1990 IEEE International Conference on Computer Languages,* March 1990.

[353] L.L. Pollock and M.L. Soffa, "Incremental global reoptimization of programs," *ACM Transactions on Programming Languages and Systems,* vol. 14, pp. 173–200, April 1992.

[354] B. Prabhala and R. Sethi, "Efficient computation of expressions with common subexpressions," *Journal of the ACM,* vol. 27, pp. 146–163, January 1980.

[355] V.R. Pratt, "Top down operator precedence," in *Conference Record of the ACM Symposium on Principles of Programming Languages,* pp. 41–51, October 1973.

[356] T. Proebsting and C.N. Fischer, "Probabilistic register allocation," in *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation,* vol. 27 (San Francisco, CA), pp. 300–310, June 1992.

[357] M.V.S. Ramanath and M. Solomon, "Optimal code for control structures," in *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages,* pp. 82–94, January 1982.

[358] M.V.S. Ramanath and M. Solomon, "Jump minimization in linear time," *ACM Transactions on Programming Languages and Systems,* vol. 6, pp. 527–545, October 1984.

[359] S.P. Reiss, "Generation of compiler symbol processing mechanisms," *ACM Transactions on Programming Languages and Systems,* vol. 5, pp. 127–163, April 1983.

[360] S.P. Reiss, "An approach to incremental compilation," in *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction,* vol. 19, pp. 144–156, June 1984.

[361] S. Richardson and M. Ganapathi, "Code optimization across procedures," *IEEE Computer,* vol. 18, pp. 42–49, February 1989.

[362] H. Richter, "Noncorrecting syntax error recovery," *ACM Transactions on Programming Languages and Systems,* vol. 7, pp. 478–489, July 1985.

[363] B.K. Rosen, Applications of high-level control flow," in *Conference Record of the 4th ACM Symposium on Principles of Programming Languages,* pp. 38–47, January 1977.

[364] B.K. Rosen, "High-level data flow analysis," *Communications of the ACM,* vol. 20, pp. 712–724, 1977.

[365] D.J. Rosenkrantz and H.B. Hunt, "Efficient algorithms for automatic construction and compactification of parsing grammars," *ACM Transactions on Programming Languages and Systems,* vol. 9, pp. 543–566, October 1987.

[366] A. Rudmik and E.S. Lee, "Compiler design for efficient code generation and program 'optimization," in *Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction,* vol. 14, pp. 127–138, August 1979.

[367] B.G. Ryder, "Incremental data flow analysis," in *Conference Record of the 10th Annual ACM Symposium on Principles of Programming Languages* (Austin, TX), pp. 167–176, January 1983.

[368] B.G. Ryder and M.C. Paull, "Elimination algorithms for data flow analysis," *ACM Computing Surveys,* vol. 18, pp. 277–316, September 1986.

[369] B.G. Ryder and M.C. Paull, "Incremental data-flow analysis algorithms," *ACM Transactions on Programming Languages and Systems,* vol. 10, pp. 1–50, January 1988.

[370] D.J. Salomom and G.V. Cormack, "Scannerless NSLR(1) parsing of programming languages," in *Proceedings of the ACM SIGPLAN' 89 Conference on Programming Language Design and Implementation,* vol. 24, (Portland, OR), pp. 170–178, June 1989.

[371] H. Samet, "A coroutine approach to parsing," *ACM Transactions on Programming Languages and Systems,* vol. 2, pp. 290–306, July 1980.

[372] V. Santhanam and D. Odnert, "Register allocation across procedure and module boundaries," in *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation,* vol. 25 (White Plains, NY), pp. 28–39, June 1990.

[373] U. Schmidt and R. Voller, "A multi-language compiler system with automatically generated code generators," in *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction,* vol. 19, pp. 202–212, June 1984.

[374] P. Schnorf, "Design of a reusable symbol table abstraction," *Structured Programming,* vol. 12, no. 2, pp. 63–74, 1991.

[375] M.D. Schwartz, N.M. Delisle, and V.S. Begwani, "Incremental compilation in Magpie," in *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction,* vol. 19, pp. 122–131, June 1984.

[376] R. Sethi and J.D. Ullman, "The generation of optimal code for arithmetic expressions," *Journal of the ACM,* vol. 17, pp. 715–728, October 1970.

[377] R. Sethi, "Complete register allocation problems," *SIAM J. Computing,* vol. 4, no. 3, pp. 226–248, 1975.

[378] R. Sethi, "Control flow aspects of semantics-directed compiling," *ACM Transactions on Programming Languages and Systems,* vol. 5, pp. 554–595, October 1983.

[379] M. Sharir and A. Pnueli, *Two Approaches to Interprocedural Data Flow Analysis.* Englewood Cliffs, NJ: Prentice-Hall, 1981.

[380] S. Sippi and E. Soisalon Soininen, "Practical error recovery in LR parsing." in *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages,* pp. 177–184, January 1982.

[381] S. Sippu and E. Soisalon-Soininen, "A syntax-error-handling technique and its experimental analysis," *ACM Transactions on Programming Languages and Systems,* vol. 5, pp. 656–679, October 1983.

[382] R.L. Sites, "The compilation of loop induction expressions," *ACM Transactions on Programming Languages and Systems,* vol. 1, pp. 50–57, July 1979.

[383] R.L. Sites, "Machine-independent register allocation," in *Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction,* vol. 14, pp. 221–225, August 1979.

[384] E. Soisalon-Soininen, "Elimination of single productions from LR parsers in conjunction with the use of default reductions," in *Conference Record of the 4th ACM Symposium on Principles of Programming Languages,* pp. 183–193, January 1977.

[385] E. Soisalon-Soininen, "Inessential error entries and their use in LR parser optimization," *ACM Transactions on Programming Languages and Systems,* vol. 4, pp. 179–195, April 1982.

[386] M. Solomon, "Type definitions with parameters," in *Conference Record of the 5th Annual ACM Symposium on Principles of Programming Languages,* pp. 31–38, January 1978.

[387] N. Suzuki and K. Ishihata, "Implementation of an array bound checker," in *Conference Record of the 4th ACM Symposium on Principles of Programming Languages,* pp. 132–143, 1977.

[388] K.-C. Tai, "Noncanonical SLR(1) grammars," *ACM Transactions on Programming Languages and Systems,* vol. 1, pp. 295–320, October 1979.

[389] Y. Tumir and C.H. Sequin, "Strategies for managing the register file in RISC," *IEEE Transactions on Computers,* vol. C-32, pp. 977–989, November 1983.

[390] A.S. Tanenbaum, H. van Staveren, and J.W. Stevenson, "Using peephole optimization on intermediate code," *ACM Transactions on Programming Languages and Systems,* vol. 4, pp. 21–36, January 1982.

[391] A.S. Tanenbaum, H. van Staveren, E.G. Keizer, and J.W. Stevenson, "A Unix toolkit for making portable compilers," in *Proc. USENIX Conf.,* pp. 255–261, July 1983.

[392] A.S. Tanenbaum, H. van Staveren, E.G. Keizer, and J.W. Stevenson, "A practical tool kit for making portable compilers," *Communications of the ACM,* vol. 26, pp. 654–660, September 1983.

[393] A.S. Tanenbaum, M.F. Kaashoek, K.G. Langendoen, and C.J.H. Jacobs, "The design of very fast portable compilers," *SIGPLAN Notices,* vol. 24, pp. 125–131, November 1989.

[394] J. Tarhio, "A compiler generator for attribute evaluation during LR parsing," in *Compiler Compilers and High Speed Compilation 2nd CCHSC Workshop Proceedings* (Berlin, GDR), pp. 146–159, October 1988.

[395] R.E. Tarjan, "Finding dominators in directed graphs," *SIAM J. Computing,* vol. 3, no. 1, pp. 62–89, 1974.

[396] R.E. Tarjan, "Testing flow graph reducibility," *Journal of Computer and System Sciences,* vol. 9, pp. 355–365, December 1974.

[397] W.F. Tichy, "Smart recompilation," *ACM Transactions on Programming Languages and Systems,* vol. 8, pp. 273–291, July 1986.

[398] J.D. Ullman, "Fast algorithms for the elimination of common subexpressions," *Acta Informatica,* vol. 2, pp. 191–213, 1979.

[399] G.A. Vankatesh, "A framework for construction and evaluation of high-level specifications for program analysis techniques," in *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation,* vol. 24 (Portland, OR), pp. 1–12, June 1989.

[400] J.W. Warfield and H.R.B. III, "An expert system for a retargetable peephole optimizer," *SIGPLAN Notices,* vol. 23, pp. 123–130, October 1988.

[401] J. Warren, "A hierarchical basis for reordering transformations," in *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages,* pp. 272–282, January 1984.

[402] M.N. Wegman and F.K. Zadeck, "Constant propagation with conditional branches," in *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages* (New Orleans, LA), pp. 291–299, January 1985.

[403] M.N. Wegman and F.K. Zadeck, "Constant propagation with conditional branches," *ACM Transactions on Programming Languages and Systems,* vol. 13, pp. 181–210, April 1991.

[404] W.E. Weihl, "Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables," in *Conference Record of the 7th Annual ACM Symposium on Principles of Programming Languages,* pp. 83–94, January 1980.

[405] A.L. Wendt, "An optimizing code generator generator," Ph.D. Thesis, Department of Computer Science, University of Arizona, 1989.

[406] D. Whitfield and M.L. Soffa, "Automatic generation of global optimizers," in *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation,* vol. 26 (Toronto, Ontario, Canada), pp. 120–129, June 1991.

[407] J.H. Williams, A. Aiken, and E.L. Wimmers, "Program transformation in the presence of errors," in *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages* (San Francisco, CA), pp. 210–217, January 1990.

Index

Compiler Design

Santanu Chattopadhyay

This well-designed text, which is the outcome of the author's many years of study, teaching and research in the field of Compilers, and his constant interaction with students, presents both the theory and design techniques used in Compiler Designing. The book introduces the readers to compilers and their design challenges and describes in detail the different phases of a compiler.

The book acquaints the students with the tools available in compiler designing. As the process of compiler designing essentially involves a number of subjects like Automata Theory, Data Structures, Algorithms, Computer Architecture, and Operating System, the contributions of these fields are also emphasized. Various types of parsers are elaborated starting with the simplest ones like recursive descent and LL to the most intricate ones like LR, canonical LR, and LALR, with special emphasis on LR parsers.

Designed primarily to serve as a text for a one-semester course in Compiler Designing for undergraduate and postgraduate students of Computer Science, this book would also be of considerable benefit to the professionals.

KEY FEATURES

- This book is comprehensive yet compact and can be covered in one semester.
- Plenty of examples and diagrams are provided to help the readers assimilate the concepts with ease.
- The exercises given in each chapter provide ample scope for practice.
- Offers insight into different optimization transformations.
- Summary at end of each chapter enables the students to recapitulate the topics easily.

THE AUTHOR

SANTANU CHATTOPADHYAY (Ph.D., Computer Science and Engineering) is Professor, Department of Electronics and Electrical Communication Engineering, IIT Kharagpur. Earlier, he was Associate Professor, Department of Computer Science and Engineering, IIT Guwahati.

A member of IEEE (USA), Dr. Chattopadhyay has to his credit more than 60 research papers published in the international journals such as *IEEE Transactions on Computers*, *Transactions on CAD*, *Transactions on VLSI* and *IEE Proceedings—Computers and Digital Techniques*. He has been associated with the program committee of many international conferences. His areas of interests are Compilers, VLSI Circuit Design, and Testing. He has co-authored a book published by the IEEE Computer Society Press (USA).

ISBN: 978-81-203-2725-2

www.phindia.com