CHAPTER 1

Introduction to Compilers

The purpose of this book is two-fold. We hope to acquaint the reader with the basic constructs of modern programming languages and to show how they can be efficiently implemented in the machine language of a typical computer. We shall also show how tools can be developed and used to help construct certain translator components. These tools not only facilitate the construction of compilers, but they can also be used in a variety of applications not directly related to compiling.

1.1 Compilers and Translators

A *translator* is a program that takes as input a program written in one programming language (the *source language*) and produces as output a program in another language (the *object* or *target language*). If the source language is a high-level language such as FORTRAN, PL/I, or COBOL, and the object language is a low-level language such as an assembly language or machine language, then such a translator is called a *compiler*.

Executing a program written in a high-level programming language is basically a two-step process, as illustrated in Fig. 1.1. The source program must first be *compiled*, that is, translated into the object program. Then the resulting object program is loaded into memory and executed.

Compilers were once considered almost impossible programs to write. The first FORTRAN compiler, for example, took 18 man-years to implement (Backus et al. [1957]). Today, however, compilers can be built with much less effort. In fact, it is not unreasonable to expect a fairly substantial compiler to be implemented as a student project in a one-semester compiler design course. The principal developments of the past twenty years which led to this improvement are:

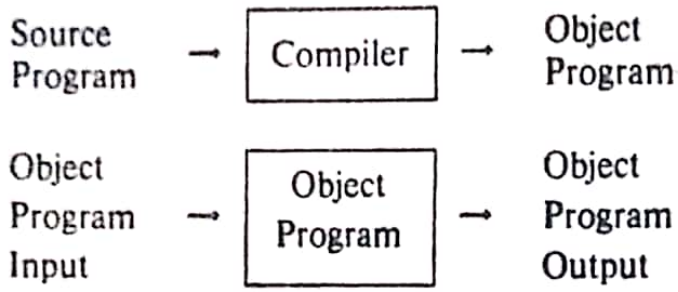


Fig. 1.1. Compilation and execution.

- The understanding of how to organize and modularize the process of compilation,
- The discovery of systematic techniques for handling many of the important tasks that occur during compilation,
- The development of software tools that facilitate the implementation of compilers and compiler components.

These are the developments we shall consider in this book. This chapter provides an overview of the compilation process and introduces the major components of a compiler.

Other Translators

Certain other translators transform a programming language into a simplified language, called *intermediate code*, which can be directly executed using a program called an *interpreter*. We may think of the intermediate code as the machine language of an abstract computer designed to execute the source code. For example, SNOBOL is often interpreted, the intermediate code being a language called Polish postfix notation (see Section 7.4). In some cases, the source language itself can be the intermediate language. For example, most *command languages*, such as JCL, in which one communicates directly with the operating system, are interpreted with no prior translation at all.

Interpreters are often smaller than compilers and facilitate the implementation of complex programming language constructs. However, the main disadvantage of interpreters is that the execution time of an interpreted program is usually slower than that of a corresponding compiled object program.

There are several other important types of translators, besides compilers. If the source language is assembly language and the target language is machine language, then the translator is called an *assembler*. The term *preprocessor* is sometimes used for translators that take programs in one

high-level language into equivalent programs in another high-level language. For example, there are many FORTRAN preprocessors that map "structured" versions of FORTRAN into conventional FORTRAN.

1.2 Why do we Need Translators?

The answer to this question is obvious to anyone who has programmed in machine language. With machine language we must communicate directly with a computer in terms of bits, registers, and very primitive machine operations. Since a machine language program is nothing more than a sequence of 0's and 1's, programming a complex algorithm in such a language is terribly tedious and fraught with opportunities for mistakes. Perhaps the most serious disadvantage of machine-language coding is that all operations and operands must be specified in a numeric code. Not only is a machine language program cryptic, but it also may be impossible to modify in a convenient manner.

Symbolic Assembly Language

Because of the difficulties with machine language programming, a host of "higher-level" languages have been invented to enable the programmer to code in a way that resembles his own thought processes rather than the elementary steps of the computer. The most immediate step away from machine language is symbolic assembly language. In this language, a programmer uses mnemonic names for both operation codes and data addresses. Thus a programmer could write ADD X, Y in assembly language, instead of something like 0110 001110 010101 in machine language (where 0110 is the hypothetical machine operation code for "add" and 001110 and 010101 are the addresses of X and Y).

A computer, however, cannot execute a program written in assembly language. That program has to be first translated to machine language, which the computer can understand. The program that performs this translation is the assembler.

Macros

Many assembly (and programming) languages provide a "macro" facility whereby a macro statement will translate into a sequence of assembly language statements and perhaps other macro statements before being translated into machine code. Thus, a macro facility is a text replacement capability. There are two aspects to macros: definition and use. To illustrate the utility of macros, consider a situation in which a machine does not have a single machine- or assembly-language statement that adds the contents of one memory address to another, as did our hypothetical assembly

instruction ADD X, Y, above. Instead, suppose the machine has an instruction LOAD, which moves a datum from memory to a register, an instruction ADD, which adds the contents of a memory address to that of a register, and an instruction STORE, which moves data from a register to memory. Using these instructions, we can create, with a *macro definition*, a "two-address add" instruction as follows.

```
MACRO       ADD2    X, Y
            LOAD    Y
            ADD     X
            STORE   Y
ENDMACRO
```

The first statement gives the name ADD2 to the macro and defines its dummy arguments, known as *formal parameters*, X and Y. The next three statements define the macro, that is, they give its translation. We assume that the machine has only one register, so the question of what registers LOAD and STORE refer to needs no elaboration.

Having defined ADD2 in this way, we can then use it as an ordinary assembly language operation code. For example, if the statement ADD2 A, B is encountered somewhere after the definition of ADD2, we have a *macro use*. Here, the macro processor substitutes for ADD2 A, B the three statements which form the definition of ADD2, but with the *actual parameters* A and B replacing the formal parameters X and Y, respectively. That is, ADD2 A, B is translated to

```
LOAD    B
ADD     A
STORE   B
```

High-Level Languages

Symbolic assembly programs are easier to write and understand than machine-language programs primarily because numerical codes for addresses and operators are replaced by more meaningful symbolic codes. Nevertheless, even with macros, there are severe drawbacks to writing in assembly language. The programmer must still know the details of how a specific computer operates. He must also mentally translate complex operations and data structures into sequences of low-level operations which use only the primitive data types that machine language provides. The programmer must also be intimately concerned with how and where data is represented within the machine. Although there are a few situations in

which such detailed knowledge is essential for efficiency, most of the programmer's time is unnecessarily wasted on such intricacies.

To avoid these problems, high-level programming languages were developed. Basically, a high-level programming language allows a programmer to express algorithms in a more natural notation that avoids many of the details of how a specific computer functions. For example, it is much more natural to write the expression A+B than a sequence of assembly language instructions to add A and B. COBOL, FORTRAN, PL/I, ALGOL,† SNOBOL, APL, PASCAL, LISP and C are some of the more common high-level languages, and we assume the reader is familiar with at least one of these languages. References for these languages and others are found in the bibliographic notes of Chapter 2.

A high-level programming language makes the programming task simpler, but it also introduces some problems. The most obvious is that we need a program to translate the high-level language into a language the machine can understand. In a sense, this program, the compiler, is completely analogous to the assembler for an assembly language.

A compiler, however, is a substantially more complex program to write than an assembler. Some compilers even make use of an assembler as an appendage, with the compiler producing assembly code, which is then assembled and loaded before being executed in the resulting machine-language form.

Before discussing compilers in detail, however, we should know the types of constructs typically found in high-level programming languages. The form and meaning of the constructs in a programming language have a strong impact on the overall design of a compiler for that language. Chapter 2 of this book reviews the main concepts concerning programming languages.

1.3 The Structure of a Compiler

A compiler takes as input a source program and produces as output an equivalent sequence of machine instructions. This process is so complex that it is not reasonable, either from a logical point of view or from an implementation point of view, to consider the compilation process as occurring in one single step. For this reason, it is customary to partition the compilation process into a series of subprocesses called *phases*, as shown in Fig. 1.2. A phase is a logically cohesive operation that takes as input one representation of the source program and produces as output another representation.

† Throughout this book, ALGOL refers to ALGOL 60 rather than ALGOL 68.

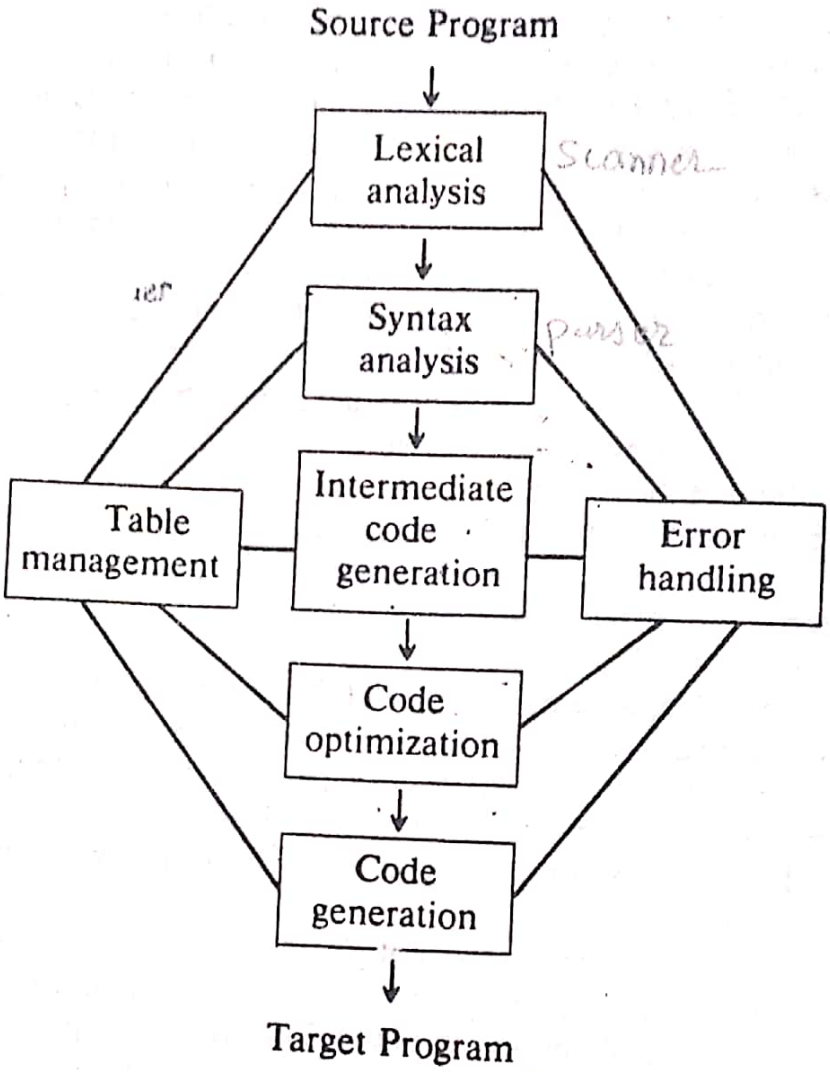


Fig. 1.2. Phases of a compiler.

The first phase, called the *lexical analyzer*, or *scanner*, separates characters of the source language into groups that logically belong together; these groups are called *tokens*. The usual tokens are keywords, such as DO or IF, identifiers, such as X or NUM, operator symbols such as <= or +, and punctuation symbols such as parentheses or commas. The output of the lexical analyzer is a stream of tokens, which is passed to the next phase, the *syntax analyzer*, or *parser*. The tokens in this stream can be represented by codes which we may regard as integers. Thus, DO might be represented by 1, + by 2, and "identifier" by 3. In the case of a token like "identifier," a second quantity, telling which of those identifiers used by the program is represented by this instance of token "identifier," is passed along with the integer code for "identifier."

The syntax analyzer groups tokens together into syntactic structures. For example, the three tokens representing A+B might be grouped into a

syntactic structure called an *expression*. Expressions might further be combined to form statements. Often the syntactic structure can be regarded as a tree whose leaves are the tokens. The interior nodes of the tree represent strings of tokens that logically belong together.

The *intermediate code generator* uses the structure produced by the syntax analyzer to create a stream of simple instructions. Many styles of intermediate code are possible. One common style uses instructions with one operator and a small number of operands. These instructions can be viewed as simple macros like the macro ADD2 discussed in Section 1.2. The primary difference between intermediate code and assembly code is that the intermediate code need not specify the registers to be used for each operation.

Code optimization is an optional phase designed to improve the intermediate code so that the ultimate object program runs faster and/or takes less space. Its output is another intermediate code program that does the same job as the original, but perhaps in a way that saves time and/or space.

The final phase, *code generation*, produces the object code by deciding on the memory locations for data, selecting code to access each datum, and selecting the registers in which each computation is to be done. Designing a code generator that produces truly efficient object programs is one of the most difficult parts of compiler design, both practically and theoretically.

The *table-management*, or *bookkeeping*, portion of the compiler keeps track of the names used by the program and records essential information about each, such as its type (integer, real, etc.). The data structure used to record this information is called a *symbol table*.

The *error handler* is invoked when a flaw in the source program is detected. It must warn the programmer by issuing a diagnostic, and adjust the information being passed from phase to phase so that each phase can proceed. It is desirable that compilation be completed on flawed programs, at least through the syntax-analysis phase, so that as many errors as possible can be detected in one compilation. Both the table management and error handling routines interact with all phases of the compiler.

Passes

In an implementation of a compiler, portions of one or more phases are combined into a module called a *pass*. A pass reads the source program or the output of the previous pass, makes the transformations specified by its phases, and writes output into an intermediate file, which may then be read by a subsequent pass. If several phases are grouped into one pass, then the operation of the phases may be interleaved, with control alternating among several phases.

The number of passes, and the grouping of phases into passes, are usually dictated by a variety of considerations germane to a particular language and machine, rather than by any mathematical optimality criteria. Appendix A shows the overall structure of several existing compilers.

The structure of the source language has a strong effect on the number of passes. Certain languages require at least two passes to generate code easily. For example, languages such as PL/I or ALGOL 68 allow the declaration of a name to occur after uses of that name. Code for expressions containing such a name cannot be generated conveniently until the declaration has been seen.

The environment in which the compiler must operate can also affect the number of passes. A multi-pass compiler can be made to use less space than a single-pass compiler, since the space occupied by the compiler program for one pass can be reused by the following pass. A multi-pass compiler is, of course, slower than a single-pass compiler, because each pass reads and writes an intermediate file. Thus, compilers running on computers with small memory would normally use several passes while, on a computer with a large random access memory, a compiler with fewer passes would be possible.

Reducing the Number of Passes

Since each phase is a transformation on a stream of data representing an intermediate form of the source program, the reader may wonder how several phases can be combined into one pass without the reading and writing of intermediate files. In some cases one pass produces its output with little or no memory of prior inputs. Lexical analysis is typical. In this situation, a small buffer serves as the interface between passes. In other cases, we may merge phases into one pass by means of a technique known as "backpatching," which is discussed in Section 7.8. In general terms, if the output of a phase cannot be determined without looking at the remainder of the phase's input, the phase can generate output with "slots" which can be filled in later, after more of the input is read.

While we cannot give an example of backpatching as it pertains to compilers until we have described in some detail what the phases do, an example from assemblers will serve as a paradigm. An assembler might have a statement like

```
GOTO L
```

which precedes a statement with label L. A two-pass assembler uses its first pass to enter into its symbol table a list of all identifiers (statement labels and data names) together with the machine address (relative to the beginning of the program), to which these identifiers correspond. Then a

second pass replaces mnemonic operation codes, such as GOTO, by their machine-language equivalent and replaces uses of identifiers by their machine addresses.

A one-pass assembler, on the other hand, could generate a skeleton of the GOTO machine instruction the first time it saw GOTO L. It could then append the machine address for this instruction to a list of instructions to be backpatched once the machine address for L is determined. For example, when the assembler encounters a statement such as

L: ADD X

it scans the list of statements referring to L and places the machine address for statement L: ADD X in the address field of each such instruction. Subsequent assembly instructions referring to L can have the value for L substituted immediately.

In a compiler, most of the backpatching that needs to be done is done over relatively short distances. For example, labels normally need to be backpatched as above only within one procedure or subroutine. We shall see other examples of backpatching in Chapter 7; and Chapter 10 contains other examples where output can be generated on a procedure-by-procedure basis.

The distance over which backpatching occurs is important since the code to be backpatched must remain accessible until backpatching is complete. Even though the object program may fit in memory when it is produced, intermediate forms of the source program may be too big to fit in memory all at once, especially as a substantial portion of memory may be occupied by the compiler program itself.

It is worth noting that programming languages with the structure of ("standard") ALGOL, where each program is a single procedure, do not lend themselves to one-pass compiling, since, for example, forward jumps may traverse the entire length of the program. In contrast, languages like PL/I or FORTRAN lend themselves to a programming style in which large programs can be created as a sequence of relatively small procedures or subroutines. In these cases, most backpatching can be done on a procedure-by-procedure basis, and the loader can be used to link the procedures together (thus providing a hidden extra pass not thought of as part of the compiler). In fact for this reason, among others, most ALGOL implementations add to the "standard" by allowing programs to consist of a sequence of procedures which may be linked.

In this book we shall not consider how much processing should be done in one pass or how big a given pass should be. The answer to this question is too dependent on the particular environment of a given compiler. Rather, we shall study each phase of the compilation process shown in Fig.

1.2 as a process in itself, investigating algorithms and tradeoffs that are applicable to the phase alone. The reader should bear in mind, however, that, in any real compiler, all phases must act in concert, and that a strategy adopted for one phase can affect the type of processing that must be done in a subsequent phase. We now turn to a more detailed look at each of the phases shown in Fig. 1.2.

1.4 Lexical Analysis

The lexical analyzer is the interface between the source program and the compiler. The lexical analyzer reads the source program one character at a time, carving the source program into a sequence of atomic units called *tokens*. Each token represents a sequence of characters that can be treated as a single logical entity. Identifiers, keywords, constants, operators, and punctuation symbols such as commas and parentheses are typical tokens. For example, in the FORTRAN statement

$$\text{IF (5 .EQ. MAX) GO TO 100} \qquad (1.1)$$

we find the following eight tokens: IF; (; 5; .EQ.; MAX;); GOTO; 100.

What is called a token depends on the language at hand and, to some extent, on the discretion of the compiler designer; but in general each token is a substring of the source program that is to be treated as a single unit. For example, it is not reasonable to treat M or MA (of the identifier MAX above) as an independent entity.

There are two kinds of token: specific strings such as IF or a semicolon, and classes of strings such as identifiers, constants, or labels. To handle both cases, we shall treat a token as a pair consisting of two parts: a token type and a token value. For convenience, a token consisting of a specific string such as a semicolon will be treated as having a type (the string itself) but no value. A token such as the identifier MAX, above, has a type "identifier" and a value consisting of the string MAX. Frequently, we shall refer to the type or value as the token itself. Thus, when we talk about identifier being a token, we are referring to a token type; when we talk about MAX being a token, we are referring to a token whose value is MAX.

The lexical analyzer and the following phase, the syntax analyzer, are often grouped together into the same pass. In that pass, the lexical analyzer operates either under the control of the parser or as a coroutine with the parser. The parser asks the lexical analyzer for the next token, whenever the parser needs one. The lexical analyzer returns to the parser a code for the token that it found. In the case that the token is an identifier or another token with a value, the value is also passed to the parser. The usual method of providing this information is for the lexical analyzer to call

a bookkeeping routine which installs the actual value in the symbol table if it is not already there. The lexical analyzer then passes the two components of the token to the parser. The first is a code for the token type (identifier), and the second is the value, a pointer to the place in the symbol table reserved for the specific value found.†

Finding Tokens

To find the next token, the lexical analyzer examines successive characters in the source program, starting from the first character not yet grouped into a token. The lexical analyzer may be required to search many characters beyond the next token in order to determine what the next token actually is.

Example 1.1. Suppose the lexical analyzer has last isolated the left parenthesis as a token in statement (1.1). We may represent the situation as follows.

IF(5.EQ.MAX)GOTO100

The string to the left of the arrow represents the symbols already broken up into tokens by the lexical analyzer. Note that blanks have been removed, since they are ignored in FORTRAN.

When the parser asks for the next token, the lexical analyzer reads all the characters between 5 and Q, inclusive, to determine that the next token is just the constant 5. The reason it has to scan as far as it does is that until it sees the Q, it is not sure it has seen the complete constant; it could be working on a floating-point constant such as 5.E−10. After determining that the next token is the constant 5, the lexical analyzer repositions its input pointer at the first dot, the character following the token.

IF(5.EQ.MAX)GOTO100

The lexical analyzer may return token type "constant" to the parser. The value associated with this "constant" could be the numerical value 5 or a pointer to the string 5. When statement (1.1) is completely processed by the lexical analyzer, the token stream might look like

if ([**const**, 341] **eq** [**id**, 729]) **goto** [**label**, 554] (1.2)

Here we use boldface codes to represent the token types. Parentheses represent their own codes. The tokens having an associated value are

† We shall see in Section 10.1 that in some cases it is impossible for the lexical analyzer to install the identifier in the symbol table, and the identifier itself must be passed to the parser.

represented by pairs in square brackets. The second component of the pair can be interpreted as an index into the symbol table where information about constants, variables, and labels is kept. The relevant entries of the symbol table are suggested in Fig. 1.3. □

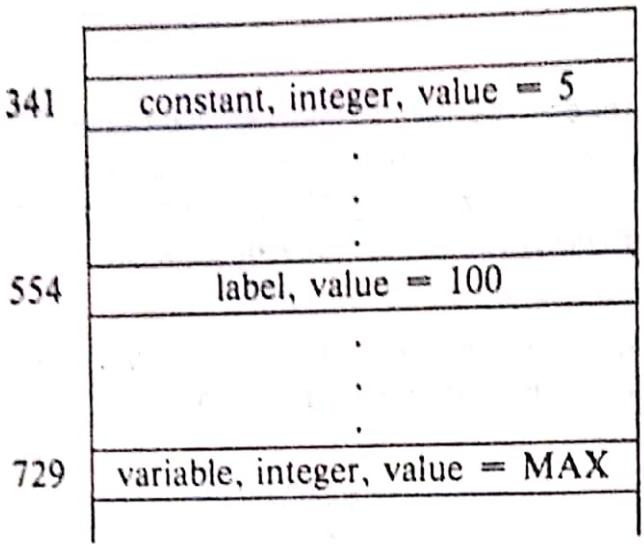


Fig. 1.3. Symbol table.

1.5 Syntax Analysis

The parser has two functions. It checks that the tokens appearing in its input, which is the output of the lexical analyzer, occur in patterns that are permitted by the specification for the source language. It also imposes on the tokens a tree-like structure that is used by the subsequent phases of the compiler.

For example, if a PL/I program contains the expression

$$A + / B$$

then after lexical analysis this expression might appear to the syntax analyzer as the token sequence

$$\mathbf{id} + / \mathbf{id}$$

On seeing the /, the syntax analyzer should detect an error situation, because the presence of these two adjacent binary operators violates the formation rules of a PL/I expression.

The second aspect of syntax analysis is to make explicit the hierarchical structure of the incoming token stream by identifying which parts of the token stream should be grouped together. For example, the expression

A / B * C.

has two possible interpretations:

a) divide A by B and then multiply by C (as in FORTRAN); or

b) multiply B by C and then use the result to divide A (as in APL).

Each of these two interpretations can be represented in terms of a *parse tree*, a diagram which exhibits the syntactic structure of the expression. Parse trees that reflect orders (a) and (b) are shown in Fig. 1.4(a) and (b), respectively. Note how in each case the operands of the first operation to be performed meet each other at a lower level than that at which they meet the remaining operand.

The language specification must tell us which of interpretations (a) and (b) is to be used, and in general, what hierarchical structure each source program has. These rules form the syntactic specification of a programming language. We shall see in Chapter 4 that *context-free grammars* are particularly helpful in specifying the syntactic structure of a language. Moreover, efficient syntactic analyzers can be constructed automatically from certain types of context-free grammars. This matter is pursued in further detail in Chapters 5 and 6.

Example 1.2. While the exact parsing of a token stream depends on the grammar chosen, a plausible grammar for FORTRAN might impose the tree structure of Fig. 1.5 on the token stream (1.2) discussed in Example 1.1. □

1.6 Intermediate Code Generation

On a logical level the output of the syntax analyzer is some representation of a parse tree. The intermediate code generation phase transforms this parse tree into an intermediate-language representation of the source program.

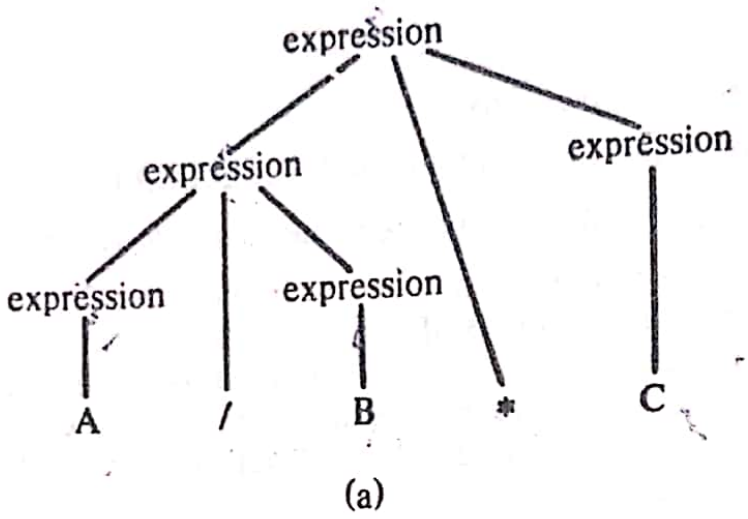
Three-Address Code

One popular type of intermediate language is what is called "three-address code." A typical three-address code statement is

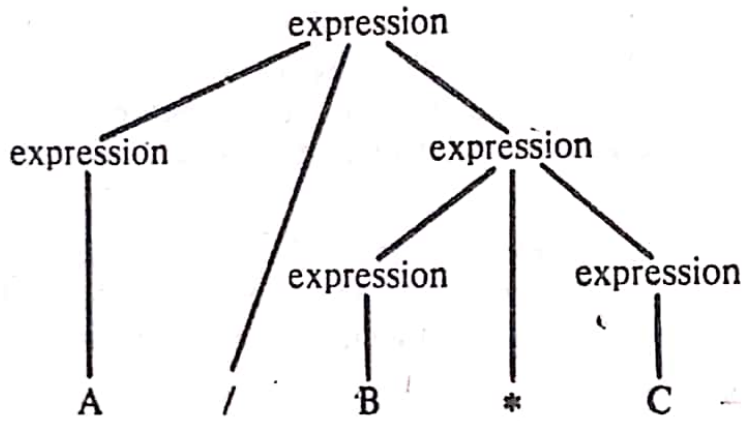
A := B **op** C

where A, B, and C are operands and **op** is a binary operator.

The parse tree in Fig. 1.4(a) might be converted into the three-address code sequence



(a)



(b)

Fig. 1.4. Parse trees.

$T_1 := A / B$

$T_2 := T_1 * C$

where T_1 and T_2 are names of temporary variables.

In addition to statements that use arithmetic operators, an intermediate language needs unconditional and simple conditional branching statements, in which at most one relation is tested to determine whether or not a branch is to be made. Higher-level flow of control statements such as

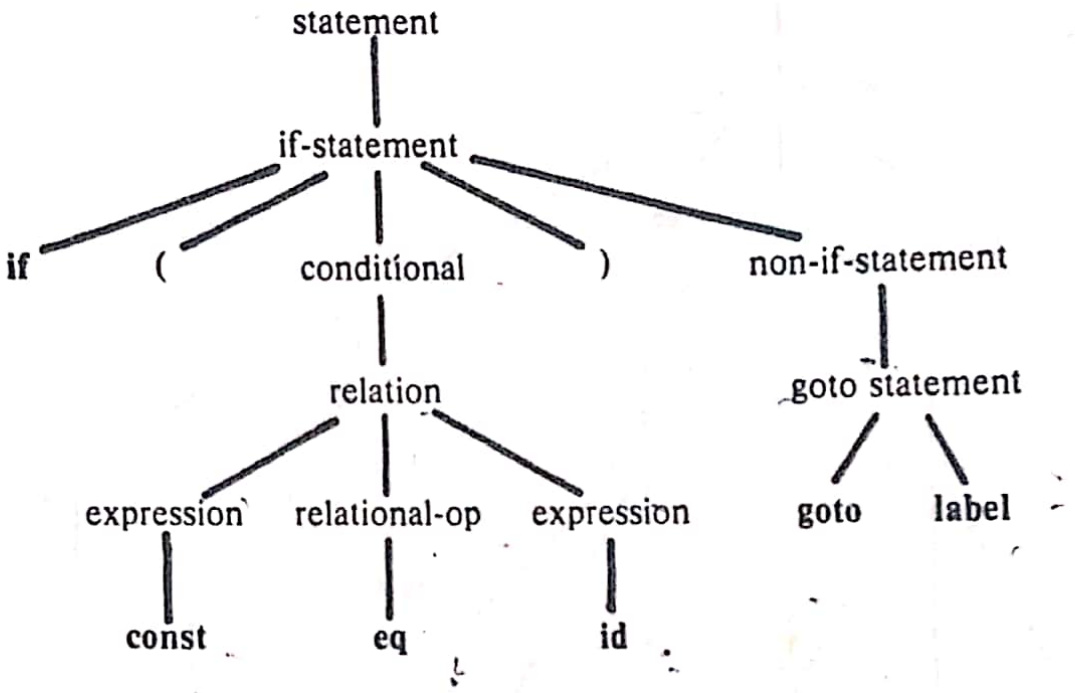


Fig. 1.5. Parse of if-statement (1.2).

while-do statements, or if-then-else statements, are translated into these lower-level conditional three-address statements.

Example 1.3. Consider the following while-statement

while A>B & A<=2*B−5 **do**
A := A + B;

which has the corresponding token stream

while [**id**, n_1] > [**id**, n_2] & [**id**, n_1] ⩽ [**const**, n_3] * [**id**, n_2]
− [**const**, n_4] **do** [**id**, n_1] ← [**id**, n_1] + [**id**, n_2];

Here n_1, n_2, n_3, and n_4 stand for pointers to the symbol table entries for A, B, 2 and 5, respectively. The parse tree for this statement might plausibly be the one shown in Fig. 1.6. We use "exp" for "expression," "relop" for "relational operator," and we indicate parenthetically the particular name or constant to which each instance of token **id** and **const** refer.

The actual algorithms by which parse trees such as Fig. 1.6 can be translated to intermediate code will not be discussed until Chapter 7. However, we can now show what the intermediate code should look like. A straightforward algorithm for translation would produce intermediate code like that shown in Fig. 1.7. The jumps over jumps, such as in the first two

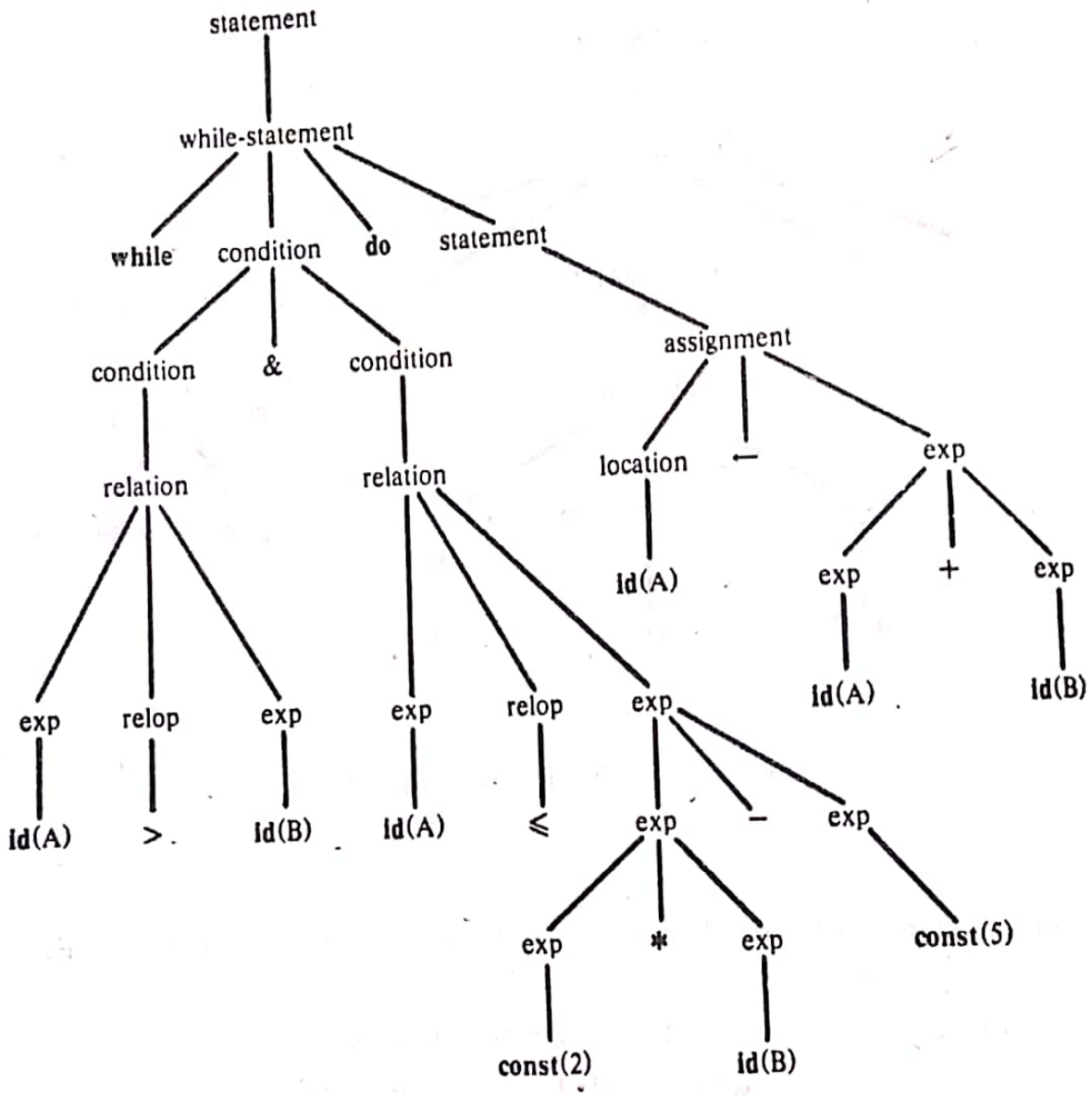


Fig. 1.6. Parse tree for while-statement.

statements, can be cleaned up during the code-optimization or code-generation phase. □

Most compilers do not generate a parse tree explicitly but rather go to intermediate code directly as syntax analysis takes place. We have chosen to talk of syntax analysis as producing a parse tree explicitly because we want to be able to evaluate several parsing algorithms in Chapters 5 and 6, without cluttering the analysis with details of intermediate-code generation.

Chapters 7 and 8 talk about intermediate-code generation by *syntax-directed translation*, a technique in which the actions of the syntax analysis

```
L1:   if A > B goto L2
      goto L3
L2:   T1 := 2 * B
      T2 := T1 - 5
      if A ≤ T2 goto L4
      goto L3
L4:   A := A + B
      goto L1
L3:
```

Fig. 1.7. Intermediate code for while-statement.

phase guide the translation. These two chapters show how to define intermediate-language constructs in terms of the syntactic constructs found in a programming language. They also show how the intermediate code can be generated as syntax analysis takes place.

1.7 Optimization

Object programs that are frequently executed should be fast and small. Certain compilers have within them a phase that tries to apply transformations to the output of the intermediate code generator, in an attempt to produce an intermediate-language version of the source program from which a faster or smaller object-language program can ultimately be produced. This phase is popularly called the *optimization phase*.

The term "optimization" in this context is a complete misnomer, since there is no algorithmic way of producing a target language program that is the best possible under any reasonable definition of "best." Optimizing compilers merely attempt to produce a better target program than would be produced with no "optimization." A good optimizing compiler can improve the target program by perhaps a factor of two in overall speed, in comparison with a compiler that generates code carefully but without using the specialized techniques generally referred to as code optimization.

Local Optimization

There are "local" transformations that can be applied to a program to attempt an improvement. For example, in Fig. 1.7 we saw two instances of jumps over jumps in the intermediate code, such as

```
        if A > B goto L2
        goto L3                    (1.3)
L2:
```

This sequence could be replaced by the single statement

$$\text{if } A \leqslant B \text{ goto } L3 \qquad (1.4)$$

Sequence (1.3) would typically be replaced in the object program b machine statements which:

1. compare A and B to set the condition codes,
2. jump to L2 if the code for $>$ is set, and
3. jump to L3.

Sequence (1.4), on the other hand, would be translated to machine instructions which:

4. compare A and B to set the condition codes, and
5. jump to L3 if the code for $<$ or $=$ is set.

If we assume $A > B$ is true half the time, then for (1.3) we execute (1) and (2) all the time and (3) half the time, for an average of 2.5 instructions. For (1.4) we always execute two instructions, a 20% savings. Also, (1.4) provides a 33% space saving if we crudely assume that all instructions require the same space.

Another important local optimization is the elimination of common subexpressions. Provided A is not an alias for B or C, the assignments

$$A := B + C + D$$

$$E := B + C + F$$

might be evaluated as

$$T_1 := B + C$$

$$A := T_1 + D$$

$$E := T_1 + F$$

taking advantage of the common subexpression B+C. Common subexpressions written explicitly by the programmer are relatively rare, however. A more productive source of common subexpressions arises from computations generated by the compiler itself. Chief among these is subscript calculation. For example, the assignment

$$A[I] := B[I] + C[I]$$

will, if the machine memory is addressed by bytes and there are, say, four bytes per word, require 4*I to be computed three times. An optimizing compiler can modify the intermediate program so that the calculation of 4*I is done only once. Note that it is impossible for the programmer to specify that this calculation of 4*I be done only once in the source program, since these address calculations are not explicit at the source level.

Loop Optimization

Another important source of optimization concerns speedups of loops. Loops are especially good targets for optimization because programs spend most of their time in inner loops. A typical loop improvement is to move a computation that produces the same result each time around the loop to a point in the program just before the loop is entered. Then this computation is done only once each time the loop is entered, rather than once for each iteration of the loop. Such a computation is called *loop invariant.*

Chapter 12 introduces the various aspects of code optimization – local optimization, loop optimization, and the analysis of the flow of control and data. Chapter 13 considers control-flow analysis, loops, and loop optimization in detail; Chapter 14 covers data-flow analysis, the gathering and utilization of information about the use of data throughout the program.

1.8 Code Generation

The code-generation phase converts the intermediate code into a sequence of machine instructions. A simple-minded code generator might map the statement A:=B+C into the machine code sequence

```
LOAD  B
ADD   C
STORE A
```

However, such a straightforward macro-like expansion of intermediate code into machine code usually produces a target program that contains many redundant loads and stores and that utilizes the resources of the target machine inefficiently.

To avoid these redundant loads and stores, a code generator might keep track of the run-time contents of registers. Knowing what quantities reside in registers, the code generator can generate loads and stores only when necessary.

Many computers have only a few high-speed registers in which computations can be performed particularly quickly. A good code generator would therefore attempt to utilize these registers as efficiently as possible. This

aspect of code generation, called *register allocation*, is particularly difficult to do optimally, but some heuristic approaches can give reasonably good results. Chapter 15 discusses some basic strategies for code generation and register allocation.

1.9 Bookkeeping

A compiler needs to collect information about all the data objects that appear in the source program. For example, a compiler needs to know whether a variable represents an integer or a real number, what size an array has, how many arguments a function expects, and so forth. The information about data objects may be explicit, as in declarations, or implicit, as in the first letter of an identifier or in the context in which an identifier is used. For example, in FORTRAN, A(I) is a function call if A has not been declared to be an array.

The information about data objects is collected by the early phases of the compiler — lexical and syntactic analysis — and entered into the symbol table. For example, when a lexical analyzer sees an identifier MAX, say, it may enter the name MAX into the symbol table if it is not already there, and produce as output a token whose value component is an index to this entry of the symbol table. If the syntax analyzer recognizes a declaration **integer** MAX, the action of the syntax analyzer will be to note in the symbol table that MAX has type "integer." No intermediate code is generated for this statement.

The information collected about the data objects has a number of uses. For example, if we have the expression A+B, where A is of type integer and B of type real, and if the language permits an integer to be added to a real, then on most computers code must be generated to convert A from type integer to type real before the addition can take place. The addition must be done in floating point, and the result is real. If mixed-mode expressions of this nature are forbidden by the language, then the compiler must issue an error message when it attempts to generate code for this construct.

The term *semantic analysis* is applied to the determination of the type of intermediate results, the check that arguments are of types that are legal for an application of an operator, and the determination of the operation denoted by the operator (e.g., + could denote fixed or floating add, perhaps logical "or," and possibly other operations as well). Semantic analysis can be done during the syntax analysis phase, the intermediate code generation phase, or the final code generation phase.

1.10 Error Handling

One of the most important functions of a compiler is the detection and reporting of errors in the source program. The error messages should allow the programmer to determine exactly where the errors have occurred. Errors can be encountered by virtually all of the phases of a compiler. For example,

1. The lexical analyzer may be unable to proceed because the next token in the source program is misspelled.
2. The syntax analyzer may be unable to infer a structure for its input because a syntactic error such as a missing parenthesis has occurred.
3. The intermediate code generator may detect an operator whose operands have incompatible types.
4. The code optimizer, doing control flow analysis, may detect that certain statements can never be reached.
5. The code generator may find a compiler-created constant that is too large to fit in a word of the target machine.
6. While entering information into the symbol table, the bookkeeping routine may discover an identifier that has been multiply declared with contradictory attributes.

Whenever a phase of the compiler discovers an error, it must report the error to the error handler, which issues an appropriate diagnostic message. Once the error has been noted, the compiler must modify the input to the phase detecting the error, so that the latter can continue processing its input, looking for subsequent errors.

Good error handling is difficult because certain errors can mask subsequent errors. Other errors, if not properly handled, can spawn an avalanche of spurious errors. Techniques for error recovery are discussed in Chapter 11.

1.11 Compiler-Writing Tools

A number of tools have been developed specifically to help construct compilers. These tools range from scanner and parser generators to complex systems, variously called *compiler-compilers, compiler-generators,* or *translator-writing systems,* which produce a compiler from some form of specification of a source language and target machine. The input specification for these systems may contain:

Bootstrapping

A compiler is characterized by three languages: its source language, its object language, and the language in which it is written. These languages may all be quite different. For example, a compiler may run on one machine and produce object code for another machine. Such a compiler is often called a *cross-compiler*. Many minicomputer and microprocessor compilers are implemented this way; they run on a bigger machine and produce object code for the smaller machine.

Sometimes we hear of a compiler being implemented in its own language. This naturally raises the question, "How was the first compiler compiled?" This question may sound like "Who was the first parent?" but it is not nearly as hard.

Suppose we have a new language L, which we want to make available on several machines, say A and B. As a first step we might write for machine A a small compiler C_A^{SA}† that translates a subset S of language L into the machine or assembly code of A. This compiler can first be written in a language that is already available on A (the assembly language of A if need be).

We then write a compiler C_S^{LA} in the simple language S. This program, when run through C_A^{SA}, becomes C_A^{LA}, the compiler for the complete language L, running on machine A, and producing object code for A. The process is shown in Fig. 1.8.

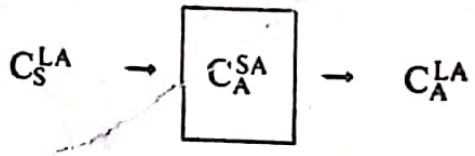


Fig. 1.8. Bootstrapping a compiler.

Now suppose we want to produce another compiler for L to run on machine B and to produce code for B. If C_S^{LA} has been designed carefully and machine B is not that different from machine A, it should be far less work to convert C_S^{LA} into a compiler C_L^{LB} which produces object code for B than it is to write a new compiler from scratch. Note that we can now use the full language L to implement C_L^{LB}.

† We use the notation C_Z^{XY} to stand for a compiler for language X, written in language Z, and producing object code in language Y. We use A and B to stand for the machine codes of computers A and B.

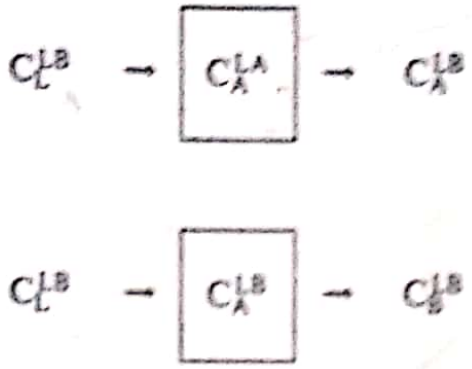


Fig. 1.9. Bootstrapping a compiler to a second machine.

Using C_L^{LB} to produce C_B^{LB}, a compiler for L on B, is now a two-step process, as shown in Fig. 1.9. We first run C_L^{LB} through C_A^{LA} to produce C_A^{LB}, a cross-compiler for L which runs on machine A but produces code for machine B. Then we run C_L^{LB} through this cross-compiler to produce the desired compiler for L that runs on machine B and produces object code for B.

Bibliographic Notes

As the division line between assemblers and compilers is not a sharp one, pronouncing a particular piece of software "the first compiler" is somewhat akin to determining "the first human species." The first compilers were developed during the mid-to-late 1940's and the early 1950's, and the reader interested in the history of the subject is referred to Sammet [1968, 1972], Bauer [1974], and Knuth and Trabb Pardo [1976].

There are a number of widely-circulated books covering various aspects of compiler design. These include Randell and Russell [1964], Hopgood [1969], Cocke and Schwartz [1970], Gries [1971], Bauer and Eickel [1974], and Lee [1974].

Some of the mathematics underlying compiler design is covered in Aho and Ullman [1972b, 1973a] and Lewis, Rosenkrantz, and Stearns [1976]. Rosen [1967] and Pollack [1972] are compendia of fundamental papers in the field.

Feldman and Gries [1968] survey the early compiler-compilers. Some of the more popular systems described in the literature are BMCC (Brooker *et al.* [1963], Rosen [1964]), META (Schorre [1964]), TGS (Cheatham [1965]), TMG (McClure [1965]), COGENT (Reynolds [1965]), XPL (McKeeman, Horning, and Wortman [1970]), and CDL (Koster [1974]).

CHAPTER 5

Basic Parsing Techniques

Chapter 4 showed how a context-free grammar can be used to define the syntax of a programming language. This chapter shows how to check whether an input string is a sentence of a given grammar and how to construct, if desired, a parse tree for the string. As every compiler performs some type of syntax analysis, usually after lexical analysis, the input to a parser is typically a sequence of tokens. The output of the parser can be of many different forms. This chapter assumes for simplicity that the output is some representation of the parse tree. Chapters 7 and 8 show how to attach "semantic" rules to the productions of a grammar to produce other kinds of outputs.

This chapter discusses the two most common forms of parsers — operator precedence and recursive descent. Operator precedence is especially suitable for parsing expressions, since it can use information about the precedence and associativity of operators to guide the parse. Recursive descent uses a collection of mutually recursive routines to perform the syntax analysis. The great bulk of compilers in existence in the early 1970's use one or both of these methods (McClure, [1972]). A common situation is for operator precedence to be used for expressions and recursive descent for the rest of the language.

The primary advantage of these methods is that they are easy to implement by hand. But there are some drawbacks as well. Operator precedence has the curious property that if one is not careful, one can recognize inputs that are not in the language of the underlying grammar. Likewise, recursive descent, particularly when augmented with backtracking, can produce rather unexpected results.

Fortunately, there are two newer methods gaining popularity that are both more general than the older methods and more firmly grounded in grammar theory. Moreover, with the proper tools (parser generators) the newer methods are easier to use than the more classical approaches. The first of these methods, LL parsing, will be mentioned in this chapter, as it

is really a table-based variant of recursive descent. The second method, LR parsing, is the subject of Chapter 6, where its most important variants will be discussed.

5.1 Parsers

A *parser* for grammar G is a program that takes as input a string w and produces as output either a parse tree for w, if w is a sentence of G, or an error message indicating that w is not a sentence of G. Often the parse tree is produced in only a figurative sense; in reality, the parse tree exists only as a sequence of actions made by stepping through the tree construction process. This chapter discusses the operation of two basic types of parsers for context-free grammars — bottom-up and top-down. As indicated by their names, bottom-up parsers build parse trees from the bottom (leaves) to the top (root), while top-down parsers start with the root and work down to the leaves. In both cases the input to the parser is being scanned from left to right, one symbol at a time.

The bottom-up parsing method we discuss is called "shift-reduce" parsing because it consists of shifting input symbols onto a stack until the right side of a production appears on top of the stack. The right side may then be replaced by (*reduced to*) the symbol on the left side of the production, and the process repeated.

Unfortunately, if $A \rightarrow XYZ$ is a production, then not every time that XYZ is on top of the stack is it correct to reduce XYZ to A; there may be occasions where it is necessary to continue to shift input symbols on top of XYZ. Designing an algorithm from a grammar so that shift-reduce decisions are made properly is the fundamental problem of bottom-up parser construction. In Section 5.3 we show how to construct one kind of shift-reduce parser called an operator-precedence parser. Chapter 6 discusses LR parsers, a more general type of shift-reduce parser.

The top-down parsing method we discuss is called recursive descent parsing. Section 5.5 shows how to construct a tabularized form of recursive descent parser called a predictive parser. Section 5.5 concludes with a discussion of LL parsers, a special kind of predictive parser.

Representation of a Parse Tree

Chapter 7 discusses the type of output typically produced by a parser in a compiler. In this chapter we shall treat the output of a parser as a representation of a parse tree for the input, if the input is syntactically well formed. There are two basic types of representations we shall consider — implicit and explicit. The sequence of productions used in some derivation is an example of an implicit representation. A linked list structure for the parse

tree is an explicit representation.

Recall that a derivation in which the leftmost nonterminal is replaced at every step is said to be leftmost. If $\alpha \Rightarrow \beta$ by a step in which the leftmost nonterminal in α is replaced, we write $\alpha \underset{lm}{\Rightarrow} \beta$. Every leftmost step, using our notational conventions, has the form $wA\gamma \underset{lm}{\Rightarrow} w\delta\gamma$ in which w consists of terminals only. If α derives β by a leftmost derivation, we write $\alpha \overset{*}{\underset{lm}{\Rightarrow}} \beta$. If $S \overset{*}{\underset{lm}{\Rightarrow}} \alpha$, then we say α is a *left-sentential form* of the grammar at hand. An analogous definition holds for rightmost derivation, where the rightmost nonterminal is replaced at every step. Rightmost derivations are sometimes called *canonical* derivations.

Every sentence of a language has both a leftmost and a rightmost derivation, as well as many others. To find one leftmost derivation for a sentence w, we can take any derivation for w and construct from it the corresponding parse tree T. From T we can then construct a leftmost derivation by traversing the tree top-down. We begin with the start symbol S, which corresponds to the root of T. We then construct the leftmost derivation

$$S = \alpha_1 \underset{lm}{\Rightarrow} \alpha_2 \underset{lm}{\Rightarrow} \cdots \underset{lm}{\Rightarrow} \alpha_n = w$$

corresponding to T, one step at a time, using the following procedure.

If the root labeled S has children labeled A, B, and C, we create the first step of the leftmost derivation by replacing S by the labels of its children; i.e., $S \underset{lm}{\Rightarrow} ABC$. Here S is α_1 and ABC is α_2.

If the node for A has children labeled XYZ in T, we create the next step of the derivation by replacing A by the labels of its children; i.e., $ABC \underset{lm}{\Rightarrow} XYZBC$. Here $XYZBC$ is α_3. We continue in this fashion by finding the node corresponding to the leftmost nonterminal D of α_i and replacing D by its children in T to obtain α_{i+1}, for each $i = 1, 2, \ldots, n-1$.

If we were to construct a parse tree in preorder†, then the order in which the nodes are created corresponds to the order in which the productions are applied in a leftmost derivation.

Example 5.1. Consider the grammar

† A *preorder* traversal of a tree is defined recursively as follows. (1) If a tree consists of one leaf, visit the leaf. (2) If a tree has root n with children $n_1, n_2, \ldots, n_k$ in order from the left, visit n, then visit the subtree rooted at n_1 in preorder, then the subtree rooted at n_2 in preorder, and so on.

$$
\begin{aligned}
&(1)\quad S \to iCtS \\
&(2)\quad S \to iCtSeS \\
&(3)\quad S \to a \\
&(4)\quad C \to b
\end{aligned}
\tag{5.1}
$$

Here i, t, and e stand for **if**, **then**, and **else**, C and S for "conditional" and "statement."

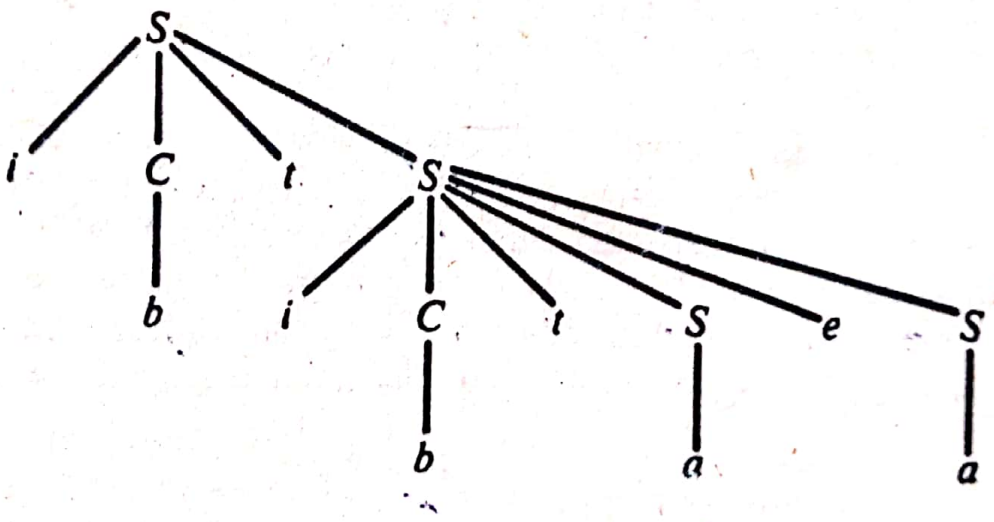
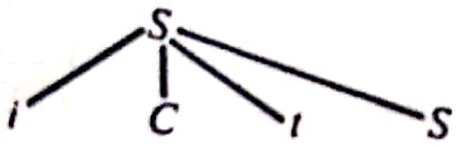


Fig. 5.1. Parse tree T.

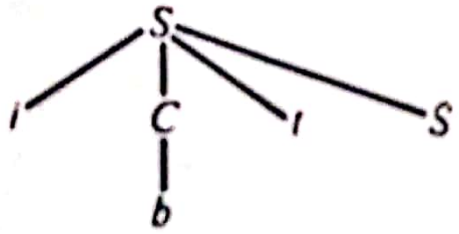
We shall construct a leftmost derivation for the sentence $w = ibtibtaea$. A parse tree T for w is shown in Fig. 5.1. The leftmost derivation corresponding to this parse tree is found to be

$$
\begin{aligned}
S &\underset{lm}{\Rightarrow} iCtS \\
&\underset{lm}{\Rightarrow} ibtS \\
&\underset{lm}{\Rightarrow} ibtiCtSeS \\
&\underset{lm}{\Rightarrow} ibtibtSeS \\
&\underset{lm}{\Rightarrow} ibtibtaeS \\
&\underset{lm}{\Rightarrow} ibtibtaea
\end{aligned}
$$

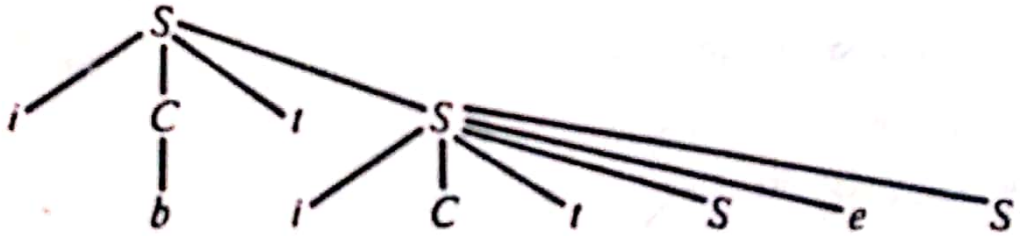
The portions of the parse tree corresponding to the first four steps of this derivation are shown in Fig. 5.2.



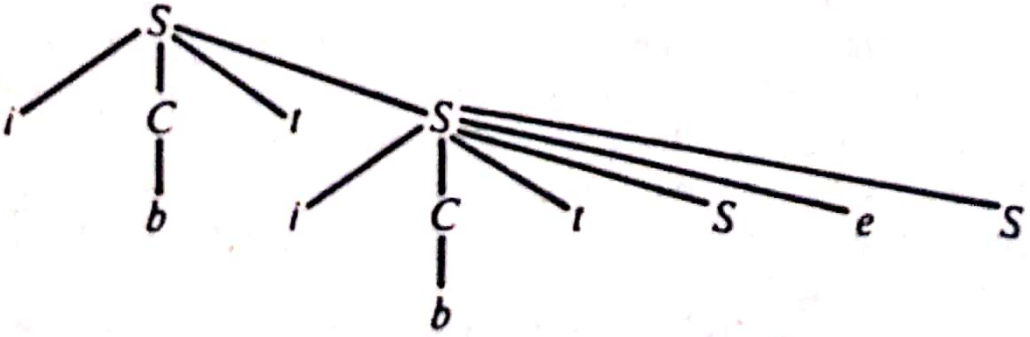
(a) $S \rightarrow iCtS$



(b) $S \rightarrow iCtS \rightarrow ibtS$



(c) $S \rightarrow iCtS \rightarrow ibtS \rightarrow ibtiCtSeS$



(d) $S \rightarrow iCtS \rightarrow ibtS \rightarrow ibtiCtSeS \rightarrow ibtibtSeS$

Fig. 5.2. Constructing a leftmost derivation.

A rightmost derivation can be constructed from a parse tree analogously. At each step we replace the rightmost nonterminal by the labels of its children. For example, the first two steps of a rightmost derivation constructed from the tree in Fig. 5.1 would be $S \Rightarrow iCtS \Rightarrow iCtiCtSeS$. □

It should now be clear that a leftmost (or rightmost) derivation is equivalent to a parse tree, in that we can easily convert a leftmost (or rightmost) derivation to a parse tree, or a parse tree to a leftmost (or rightmost) derivation.

5.2 Shift-Reduce Parsing

In this section we discuss a bottom-up style of parsing called shift-reduce parsing. This parsing method is bottom-up because it attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top). We can think of this process as one of "reducing" a string w to the start symbol of a grammar. At each step a string matching the right side of a production is replaced by the symbol on the left.

For example, consider the grammar

$$S \rightarrow aAcBe$$

$$A \rightarrow Ab \mid b$$

$$B \rightarrow d$$

and the string *abbcde*. We want to reduce this string to S. We scan *abbcde* looking for substrings that match the right side of some production. The substrings b and d qualify. Let us choose the leftmost b and replace it by A, the left side of the production $A \rightarrow b$. We obtain the string *aAbcde*. We now find that Ab, b, and d each match the right side of some production. Suppose this time we choose to replace the substring Ab by A, the left side of the production $A \rightarrow Ab$. We now obtain *aAcde*. Then replacing d by B, the left side of the production $B \rightarrow d$, we obtain *aAcBe*. We can now replace this entire string by S.

Each replacement of the right side of a production by the left side in the process above is called a *reduction*. Thus, by a sequence of four reductions we were able to reduce *abbcde* to S. These reductions, in fact, traced out a rightmost derivation in reverse.

Informally, a substring which is the right side of a production such that replacement of that substring by the production left side leads eventually to a reduction to the start symbol, by the reverse of a rightmost derivation, is called a "handle." The process of bottom-up parsing may be viewed as one of finding and reducing handles.

We must not be misled by the simplicity of this example. In many cases the leftmost substring β which matches the right side of some production $A \rightarrow \beta$ is not a handle because a reduction by the production $A \rightarrow \beta$ may yield a string which cannot be reduced to the start symbol. For example, if we replaced b by A in the second string $aAbcde$ we would obtain a string $aAAcde$ which cannot be subsequently reduced to S. For this reason, we must give a more precise definition of a handle. We shall see that if we write a rightmost derivation in reverse, then the sequence of replacements made in that derivation naturally defines a sequence of correct replacements that reduce the sentence to the start symbol.

Defn.

Handles

A *handle* of a right-sentential form γ is a production $A \rightarrow \beta$ and a position of γ where the string β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ. That is, if $S \underset{rm}{\overset{*}{\Longrightarrow}} \alpha A w \underset{rm}{\Longrightarrow} \alpha\beta w$, then $A \rightarrow \beta$ in the position following α is a handle of $\alpha\beta w$. The string w to the right of the handle contains only terminal symbols.

In the example above, *abbcde* is a right-sentential form whose handle is $A \rightarrow b$ at position 2. Likewise, *aAbcde* is a right-sentential form whose handle is $A \rightarrow Ab$ at position 2.

Sometimes we shall say "the substring β is a handle of $\alpha\beta w$" if the position of β and the production $A \rightarrow \beta$ we have in mind are clear. If a grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.

Figure 5.3 portrays the handle β in the parse tree of a right-sentential form $\alpha\beta w$. The handle represents the leftmost complete subtree consisting of a node and all its children. In Fig. 5.3, A is the leftmost node with all its children in the tree.

Example 5.2. Consider the following grammar

$$\begin{aligned} &(1)\quad E \rightarrow E + E \\ &(2)\quad E \rightarrow E * E \\ &(3)\quad E \rightarrow (E) \\ &(4)\quad E \rightarrow \mathbf{id} \end{aligned} \tag{5.2}$$

and consider the rightmost derivation

$$\begin{aligned} E &\underset{rm}{\Longrightarrow} \underline{E + E} \\ &\underset{rm}{\Longrightarrow} E + \underline{E * E} \end{aligned}$$

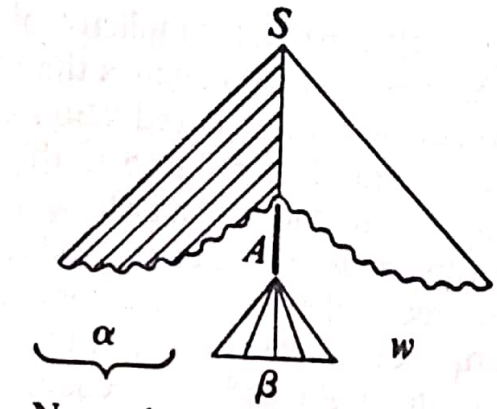


Fig. 5.3. The handle in a parse tree.

$$\underset{\text{rm}}{\Longrightarrow} E + E * \underline{\mathbf{id}_3}$$

$$\underset{\text{rm}}{\Longrightarrow} E + \underline{\mathbf{id}_2} * \mathbf{id}_3$$

$$\underset{\text{rm}}{\Longrightarrow} \underline{\mathbf{id}_1} + \mathbf{id}_2 * \mathbf{id}_3$$

We have subscripted the **id**'s for notational convenience and underlined a handle of each right-sentential form. For example, $\mathbf{id}_1$ is a handle of the right-sentential form $\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$ because **id** is the right side of the production $E \rightarrow \mathbf{id}$, and replacing $\mathbf{id}_1$ by E produces the previous right-sentential form $E + \mathbf{id}_2 * \mathbf{id}_3$. Note that the string appearing to the right of a handle contains only terminal symbols.

Because grammar (5.2) is ambiguous, there is another rightmost derivation of the same string. This derivation begins $E \Longrightarrow E * E$ and produces another set of handles. In particular, $E + E$ is a handle of $E + E * \mathbf{id}_3$ according to this derivation, just as $\underline{\mathbf{id}_3}$ by itself is a handle of this same right sentential form according to the derivation above. □

Handle Pruning

A rightmost derivation in reverse, often called a *canonical reduction sequence*, is obtained by "handle pruning." That is, we start with a string of terminals w which we wish to parse. If w is a sentence of the grammar at hand, then $w = \gamma_n$, where γ_n is the nth right-sentential form of some as yet unknown rightmost derivation

$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w.$$

To reconstruct this derivation in reverse order, we locate the handle β_n in γ_n and replace β_n by the left side of some production $A_n \rightarrow \beta_n$ to obtain the $(n-1)$st right-sentential form γ_{n-1}. Note that we have not yet told how handles are to be found, but we shall give methods of doing so in this chapter and the next.

We then repeat this process. That is, we locate the handle β_{n-1} in γ_{n-1} and reduce this handle to obtain the right-sentential form γ_{n-2}. If by continuing this process we produce a right-sentential form consisting only of the start symbol S, then we halt and announce successful completion of parsing. The reverse of the sequence of productions used in the reductions is a rightmost derivation for the input string.

Example 5.3. Consider the grammar (5.2) of Example 5.2 and the input string $\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$. The following sequence of reductions reduces $\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$ to the start symbol E:

Right-sentential form	Handle	Reducing production
$\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$	$\mathbf{id}_1$	$E \rightarrow \mathbf{id}$
$E + \mathbf{id}_2 * \mathbf{id}_3$	$\mathbf{id}_2$	$E \rightarrow \mathbf{id}$
$E + E * \mathbf{id}_3$	$\mathbf{id}_3$	$E \rightarrow \mathbf{id}$
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
E		

The reader should observe that the sequence of right-sentential forms in this example is just the reverse of the sequence in the rightmost derivation in Example 5.2. Again, recall that there is another rightmost derivation in which $\mathbf{id}_1 + \mathbf{id}_2$ comes from a single E, and this rightmost derivation implies a different sequence of handles. □

Stack Implementation of Shift-Reduce Parsing

There are two problems that must be solved if we are to automate parsing by handle pruning. The first is how to locate a handle in a right-sentential form, and the second is what production to choose in case there is more than one production with the same right side. Before we get to these questions, let us first consider the type of data structures to use in a handle-pruning parser.

A convenient way to implement a shift-reduce parser is to use a stack and an input buffer. We shall use \$ to mark the bottom of the stack and the right end of the input.

Stack	Input
\$	w \$

The parser operates by shifting zero or more input symbols onto the stack until a handle β is on top of the stack. The parser then reduces β to the left side of the appropriate production. The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty:

Stack	Input
\$$S$	\$

In this configuration the parser halts and announces successful completion of parsing.

Example 5.4. Let us step through the actions a shift-reduce parser might make in parsing the input string $\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$ according to grammar (5.2), using the derivation of Example 5.2. The sequence is shown in Fig. 5.4. Note that because grammar (5.2) has two rightmost derivations for this input there is another sequence of steps a shift-reduce parser could take. □

	Stack	Input	Action
(1)	\$	$\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$\$	shift
(2)	\$$\mathbf{id}_1$	$+ \mathbf{id}_2 * \mathbf{id}_3$\$	reduce by $E \rightarrow \mathbf{id}$
(3)	\$$E$	$+ \mathbf{id}_2 * \mathbf{id}_3$\$	shift
(4)	\$$E +$	$\mathbf{id}_2 * \mathbf{id}_3$\$	shift
(5)	\$$E + \mathbf{id}_2$	$* \mathbf{id}_3$\$	reduce by $E \rightarrow \mathbf{id}$
(6)	\$$E + E$	$* \mathbf{id}_3$\$	shift
(7)	\$$E + E *$	$\mathbf{id}_3$\$	shift
(8)	\$$E + E * \mathbf{id}_3$	\$	reduce by $E \rightarrow \mathbf{id}$
(9)	\$$E + E * E$	\$	reduce by $E \rightarrow E * E$
(10)	\$$E + E$	\$	reduce by $E \rightarrow E + E$
(11)	\$$E$	\$	accept

Fig. 5.4. Shift-reduce parsing actions.

While the primary operations of the parser are shift and reduce, there are actually four possible actions a shift-reduce parser can make: (1) shift, (2) reduce, (3) accept, and (4) error.

1. In a *shift* action, the next input symbol is shifted to the top of the stack.
2. In a *reduce* action, the parser knows the right end of the handle is at the top of the stack. It must then locate the left end of the handle within the stack and decide with what nonterminal to replace the handle.

3. In an *accept* action, the parser announces successful completion of parsing.
4. In an *error* action, the parser discovers that a syntax error has occurred and calls an error recovery routine.

There is an important fact that justifies the use of a stack in shift-reduce parsing: the handle will always eventually appear on top of the stack, never inside. This fact becomes obvious when we consider the possible forms of two successive steps in any rightmost derivation. These two steps can be of the form

$$(1) \quad S \underset{rm}{\overset{*}{\Rightarrow}} \alpha Az \underset{rm}{\overset{*}{\Rightarrow}} \alpha\beta Byz \underset{rm}{\overset{*}{\Rightarrow}} \alpha\beta\gamma yz$$

or

$$(2) \quad S \underset{rm}{\overset{*}{\Rightarrow}} \alpha BxAz \underset{rm}{\overset{*}{\Rightarrow}} \alpha Bxyz \underset{rm}{\overset{*}{\Rightarrow}} \alpha\gamma xyz$$

In case (1), A is replaced by βBy, and then the rightmost nonterminal B in that right side is replaced by γ. In case (2), A is again replaced first, but this time the right side is a string y of terminals only. The next rightmost nonterminal, B, will be somewhere to the left of y.

Let us consider case (1) in reverse, where a shift-reduce parser has just reached the configuration

Stack	Input
$\$\alpha\beta\gamma$	$yz\$$

The parser now reduces the handle γ to B to reach the configuration

Stack	Input
$\$\alpha\beta B$	$yz\$$

Since B is the rightmost nonterminal in $\alpha\beta Byz$, the right end of the handle of $\alpha\beta Byz$ cannot occur inside the stack. The parser can therefore shift the string y onto the stack to reach the configuration

Stack	Input
$\$\alpha\beta By$	$z\$$

in which βBy is the handle, and it gets reduced to A.

In case (2), in configuration

Stack	Input
$\$\alpha\gamma$	$xyz\$$

the handle γ is on top of the stack. After reducing the handle γ to B, the parser can shift the string xy to get the next handle y on top of the stack:

Stack	Input
$\$\alpha Bxy$	$z\$$

Now the parser reduces y to A.

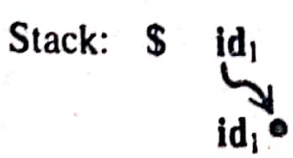
In both cases, after making a reduction the parser had to shift zero or more symbols to get the next handle onto the stack. It never had to go into the stack to find the handle. It is this aspect of handle pruning that makes a stack a particularly convenient data structure for implementing a shift-reduce parser. The next section of this chapter and all of Chapter 6 are devoted to methods whereby shift-reduce-accept-error decisions can be made.

Constructing a Parse Tree

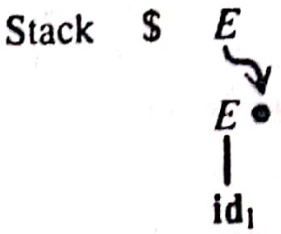
It is often useful to construct a parse tree explicitly. This can be done quite simply as we perform shift-reduce parsing. The strategy is to maintain a *forest* of partially-completed derivation trees as we parse bottom-up. With each symbol on the stack we associate a pointer to a tree whose root is that symbol and whose yield is the string of terminals which have been reduced to that symbol, perhaps by a long series of reductions. At the end of the shift-reduce parse, the start symbol remaining on the stack will have the entire parse tree associated with it. The bottom-up tree construction process has two aspects.

1. When we shift an input symbol a onto the stack we create a one-node tree labeled a. Both the root and the yield of this tree are a, and the yield truly represents the string of terminals "reduced" (by zero reductions) to the symbol a.

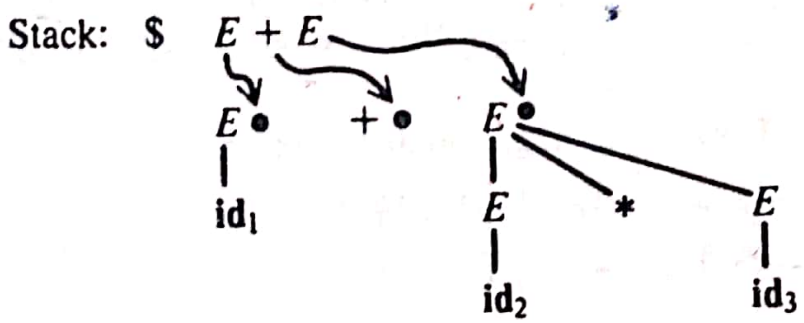
2. When we reduce $X_1X_2 \cdots X_n$ to A, we create a new node labeled A. Its children, from left to right, are the roots of the trees for $X_1, X_2, \ldots, X_n$. If for all i the tree for X_i has yield x_i, then the yield for the new tree is $x_1x_2 \cdots x_n$. This string has in fact been reduced to A by a series of reductions culminating in the present one. As a special case, if we reduce ϵ to A we create a node labeled A with one child labeled ϵ.



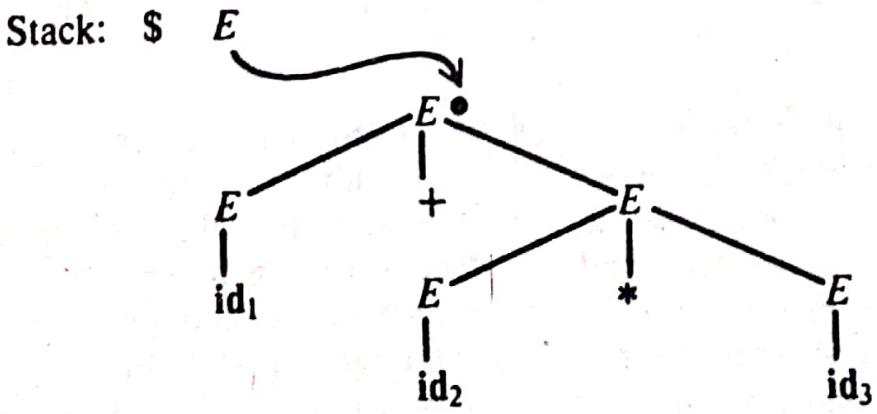
(a) After shifting $\mathbf{id}_1$.



(b) After reducing $\mathbf{id}_1$ to E.



(c) After reducing $\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$ to $E + E$.



(d) At completion.

Fig. 5.5. Parse tree construction.

Example 5.5. Consider the sequence of steps depicted in Fig. 5.4. At line (2) the stack and sequence of trees — one tree consisting of one node — is shown in Fig. 5.5(a). At line (3), the **id** at the top is reduced to E. A node labeled E is created, and its one child is the node pointed to by the **id** removed from the stack. The result is shown in Fig. 5.5(b). After a sequence of shifts and reductions we arrive at the situation shown in Fig. 5.5(c), which corresponds to line (10) of Fig. 5.4. At that point, $E + E$ is

reduced to E. We create a new node labeled E, and this node receives three children, the nodes pointed to by the three top positions on the stack. The result is shown in Fig. 5.5(d) and corresponds to line (11) of Fig. 5.4. □

5.3 Operator-Precedence Parsing

For a certain small class of grammars we can easily construct efficient shift-reduce parsers by hand. These grammars have the property (among other essential requirements) that no production right side is ϵ or has two adjacent nonterminals. A grammar with the latter property is called an *operator grammar*.

Example 5.6. The following grammar for expressions

$$E \rightarrow E\,A\,E \mid (E) \mid -E \mid \mathbf{id}$$
$$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$$

is not an operator grammar, because the right side EAE has two (in fact three) consecutive nonterminals. However, if we substitute for A each of its alternates, we obtain the following operator grammar:

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid E \uparrow E \mid (E) \mid -E \mid \mathbf{id} \quad (5.3)$$

□

We shall now describe an easy-to-implement parsing technique called operator-precedence parsing. Historically, the operator-precedence technique was first described as a manipulation on tokens without any reference to an underlying grammar. In fact, once we finish building an operator-precedence parser from a grammar, we may effectively ignore the grammar, using the nonterminals on the stack only as placeholders for the nodes of a parse tree being constructed in the bottom-up fashion just described.

As a general parsing technique, operator-precedence parsing has a number of disadvantages. For example, it is hard to handle tokens like the minus sign which has two different precedences (depending on whether it is unary or binary). Worse, since the relationship between a grammar for the language being parsed and the operator-precedence parser itself is tenuous, one cannot always be sure the parser accepts exactly the desired language. Finally, only a small class of grammars can be parsed using operator-precedence techniques.

Nevertheless, because of its simplicity, numerous compilers using operator-precedence parsing techniques for expressions have been successfully built. Often these parsers use recursive descent, described in the next section, for statements and higher-level constructs. Operator-precedence

parsers have even been built for entire languages. SNOBOL, being virtually "all operators," is an example of a language for which operator precedence works well.

In operator-precedence parsing, we use three disjoint *precedence relations*, $<\cdot$, $\doteq$ and $\cdot>$, between certain pairs of terminals. These precedence relations guide the selection of handles. If $a <\cdot b$, we say a "yields precedence to" b; if $a \doteq b$, a "has the same precedence as" b; if $a \cdot> b$, a "takes precedence over" b. We should caution the reader that while these relations may appear similar to the arithmetic relations "less than," "equal to" and "greater than," the precedence relations have quite different properties. For example, we could have $a <\cdot b$ and $a \cdot> b$ for the same language, or we might have none of $a <\cdot b$, $a \doteq b$, and $a \cdot> b$ holding for some terminals a and b.

There are two common ways of determining what precedence relation should hold between a pair of terminals. The first method we discuss is intuitive and is based on the traditional notions of associativity and precedence of operators. For example, if $*$ is to have higher precedence than $+$, we make $+ <\cdot *$ and $* \cdot> +$. This approach will be seen to resolve the ambiguities of grammar (5.3) and to enable us to write an operator-precedence parser for it (although the unary minus sign causes problems).

The second method of selecting operator-precedence relations is to first construct an unambiguous grammar for the language, a grammar which reflects the correct associativity and precedence in its parse trees. This job is not difficult for expressions; Example 4.6 provided the paradigm. For the other common source of ambiguity, the dangling **else**, the grammar in Appendix B, which uses the nonterminal restricted-statement to generate statements other than the **if** $\cdots$ **then** statement, is a useful model. Having obtained an unambiguous grammar, there is a mechanical method for constructing operator-precedence relations from it. These relations may not be disjoint, and they may parse a language other than that generated by the grammar, but with the standard sorts of arithmetic expressions, few problems are encountered in practice.

Using Operator-Precedence Relations

The intention of the precedence relations is to delimit the handle of a right-sentential form, with $<\cdot$ marking the left end, $\doteq$ appearing in the interior of the handle, if any, and $\cdot>$ marking the right end. To be more precise, suppose we have a right-sentential form of an operator grammar. The fact that no adjacent nonterminals appear on the right sides of productions implies that no right-sentential form will have two adjacent nonterminals either. Thus, we may write the right-sentential form as $\beta_0 a_1 \beta_1 \cdots a_n \beta_n$, where each β_i is either ϵ (the empty string) or a single

nonterminal, and each a_i is a single terminal. Suppose that between a_i and a_{i+1} exactly one of the relations $\lessdot$, $\doteq$, and $\gtrdot$ holds. Further, \$ marks each end of the string, and we define \$ $\lessdot$ b and b $\gtrdot$ \$ for all terminals b.

Now suppose we remove the nonterminals from the string and place the correct relation, $\lessdot$, $\doteq$, or $\gtrdot$, between each pair of terminals and between the endmost terminals and the \$'s marking the ends of the string. For example, suppose we initially have the right-sentential form **id** + **id** * **id** and the precedence relations are those given in Fig. 5.6.

	id	+	*	\$
id		$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
\$	$\lessdot$	$\lessdot$	$\lessdot$	

Fig. 5.6. Operator precedence relations.

Then the string with the precedence relations inserted is:

$$\$\lessdot \mathbf{id} \gtrdot + \lessdot \mathbf{id} \gtrdot * \lessdot \mathbf{id} \gtrdot \$ \tag{5.4}$$

For example, $\lessdot$ is inserted between \$ and **id** since $\lessdot$ is the entry in row \$ and column **id**. Now the handle can be found by the following process.

1. Scan the string from the left end until the leftmost $\gtrdot$ is encountered. In (5.4) above, this occurs between the first **id** and +.
2. Then scan backwards (to the left) over any $\doteq$'s until a $\lessdot$ is encountered. In (5.4), we scan backwards to \$.
3. The handle contains everything to the left of the first $\gtrdot$ and to the right of the $\lessdot$ encountered in step (2), including any intervening or surrounding nonterminals. (The inclusion of surrounding nonterminals is necessary so that two adjacent nonterminals do not appear in a right-sentential form.) In (5.4) the handle is the first **id**.

If we are dealing with grammar (5.3), we then reduce **id** to E. At this point we have the right-sentential form E+**id***id. After reducing the two remaining **id**'s to E by the same steps, we obtain the right-sentential form $E+E*E$. Consider now the string \$+*\$ obtained by deleting the nonterminals. Inserting the precedence relations, we get

$$\$ \lessdot + \lessdot * \gtrdot \$$$

indicating that the left end of the handle lies between + and * and the right end between * and \$. These precedence relations indicate that, in the

right-sentential form $E+E*E$, the handle is $E*E$. Note how the E's surrounding the * become part of the handle.

Since the nonterminals do not influence the parse, we need not worry about distinguishing among them. A single marker "nonterminal" can be kept on the stack of a shift-reduce parser to indicate placeholders for nodes of the parse tree being constructed or to indicate a variety of possible translations, as discussed in Chapter 7.

It may appear from the discussion above that the entire right-sentential form must be scanned at each step to find the handle. Such is not the case if the precedence relations are used to guide the actions of a shift-reduce parser. If the precedence relation $<\cdot$ or $\doteq$ holds between the topmost terminal symbol on the stack and the next input symbol, the parser shifts. The parser has not yet found the right end of the handle. If the relation $\cdot>$ holds, a reduction is called for. Now the parser has found the right end of the handle, and the precedence relations can be used to find the left end of the handle in the stack.

If no precedence relation holds between a pair of terminals (indicated by a blank entry in Fig. 5.6), then a syntactic error has been detected and an error recovery routine, such as the one described in Chapter 11, is invoked.

Operator-Precedence Relations from Associativity and Precedence

We are always free to create operator-precedence relations any way we see fit and hope that a shift-reduce parser will work properly when guided by them. For a language of arithmetic expressions such as that generated by grammar (5.3) we can use the following heuristic to produce a useful set of precedence relations. Note that grammar (5.3) is ambiguous, and right-sentential forms could have many handles. Our rules are designed to select the "proper" handles to reflect a given set of associativity and precedence rules for binary operators.

1. If operator θ_1 has higher precedence than operator θ_2, make $\theta_1 \cdot> \theta_2$ and $\theta_2 <\cdot \theta_1$. For example, if * has higher precedence than +, make $* \cdot> +$ and $+ <\cdot *$. These relations ensure that, in an expression of the form $E+E*E+E$, the central $E*E$ is the handle that will be reduced first.

2. If θ_1 and θ_2 are operators of equal precedence (they may in fact be the same operator), then make $\theta_1 \cdot> \theta_2$ and $\theta_2 \cdot> \theta_1$ if the operators are left-associative, or make $\theta_1 <\cdot \theta_2$ and $\theta_2 <\cdot \theta_1$ if they are right-associative. For example, if + and − are left-associative, then make $+ \cdot> +$, $+ \cdot> -$, $- \cdot> -$, and $- \cdot> +$. If $\uparrow$ is right associative, then $\uparrow <\cdot \uparrow$. These relations ensure that $E-E+E$ will have handle $E-E$ selected and $E\uparrow E\uparrow E$ will have the last $E\uparrow E$ selected.

3. Make $\theta \lessdot \cdot$ id, id $\cdot\gtrdot \theta$, $\theta \lessdot \cdot$ (, ($\lessdot \cdot \theta$,) $\cdot\gtrdot \theta$, $\theta \cdot\gtrdot$), $\theta \cdot\gtrdot$ \$ and \$ $\lessdot \cdot \theta$ for all operators θ. Also, let

($\doteq$)	\$ $\lessdot \cdot$ (	\$ $\lessdot \cdot$ id
($\lessdot \cdot$ (	id $\cdot\gtrdot$ \$	) $\cdot\gtrdot$ \$
($\lessdot \cdot$ id	id $\cdot\gtrdot$)	) $\cdot\gtrdot$)

These rules ensure that id will be reduced to E wherever found and (E) will be reduced to E wherever found. Also, \$ will serve as both left and right endmarker, causing handles to be found between \$'s wherever possible.

Example 5.7. The operator-precedence relations for grammar (5.3), assuming

1. ↑ is of highest precedence and right-associative,
2. • and / are of next highest precedence and left-associative, and
3. + and − are of lowest precedence and left-associative,

are shown in Fig. 5.7. (Blanks denote error entries.) The reader should try out the table to see that it works correctly, ignoring problems with unary minus for the moment. Try input id•(id↑id)−id/id, for example. □

	+	−	•	/	↑	id	(	)	\$
+	$\cdot\gtrdot$	$\cdot\gtrdot$	$\lessdot\cdot$	$\lessdot\cdot$	$\lessdot\cdot$	$\lessdot\cdot$	$\lessdot\cdot$	$\cdot\gtrdot$	$\cdot\gtrdot$
−	$\cdot\gtrdot$	$\cdot\gtrdot$	$\lessdot\cdot$	$\lessdot\cdot$	$\lessdot\cdot$	$\lessdot\cdot$	$\lessdot\cdot$	$\cdot\gtrdot$	$\cdot\gtrdot$
•	$\cdot\gtrdot$	$\cdot\gtrdot$	$\cdot\gtrdot$	$\cdot\gtrdot$	$\lessdot\cdot$	$\lessdot\cdot$	$\lessdot\cdot$	$\cdot\gtrdot$	$\cdot\gtrdot$
/	$\cdot\gtrdot$	$\cdot\gtrdot$	$\cdot\gtrdot$	$\cdot\gtrdot$	$\lessdot\cdot$	$\lessdot\cdot$	$\lessdot\cdot$	$\cdot\gtrdot$	$\cdot\gtrdot$
↑	$\cdot\gtrdot$	$\cdot\gtrdot$	$\cdot\gtrdot$	$\cdot\gtrdot$	$\lessdot\cdot$	$\lessdot\cdot$	$\lessdot\cdot$	$\cdot\gtrdot$	$\cdot\gtrdot$
id	$\cdot\gtrdot$	$\cdot\gtrdot$	$\cdot\gtrdot$	$\cdot\gtrdot$	$\cdot\gtrdot$			$\cdot\gtrdot$	$\cdot\gtrdot$
(	$\lessdot\cdot$	$\lessdot\cdot$	$\lessdot\cdot$	$\lessdot\cdot$	$\lessdot\cdot$	$\lessdot\cdot$	$\lessdot\cdot$	$\doteq$	
)	$\cdot\gtrdot$	$\cdot\gtrdot$	$\cdot\gtrdot$	$\cdot\gtrdot$	$\cdot\gtrdot$			$\cdot\gtrdot$	$\cdot\gtrdot$
\$	$\lessdot\cdot$	$\lessdot\cdot$	$\lessdot\cdot$	$\lessdot\cdot$	$\lessdot\cdot$	$\lessdot\cdot$	$\lessdot\cdot$		

Fig. 5.7. Operator precedence relations.

Handling Unary Operators

If we have a unary operator such as ¬ (logical negation), which is not also a binary operator, we can easily incorporate it into the above scheme for creating operator-precedence relations. Supposing ¬ to be a unary prefix operator, we make $\theta \lessdot \cdot$ ¬ for any operator θ, whether unary or binary. We make ¬ $\cdot\gtrdot \theta$ if ¬ has higher precedence than θ and ¬ $\lessdot \cdot \theta$ if not. For

example, if ¬ has higher precedence than &, and & is left-associative, we would group $E\&\neg E\&E$ as $(E\&(\neg E))\&E$, by these rules. The rule for unary postfix operators is analogous and is left as an exercise.

The situation changes when we have an operator like the minus sign − which is both unary prefix and binary infix. Even if we give unary and binary minus the same precedence, the table of Fig. 5.7 will fail to parse strings like **id***−**id** correctly. The best approach in this case is to use the lexical analyzer to distinguish between unary and binary minus, by having it return a different token when it sees unary minus. Unfortunately, the lexical analyzer cannot use lookahead to distinguish the two; it must remember the previous token. In FORTRAN, for example, a minus sign is unary if the previous token was an operator, a left parenthesis, a comma, or an assignment symbol.

Operator-Precedence Grammars

If we use an unambiguous operator grammar for arithmetic expressions or for a variety of other programming-language constructs, and our grammar satisfies certain conditions, we can construct a reliable operator-precedence table for the grammar. The resulting parser might accept strings not in the language of the grammar, but in practical cases that does not usually happen. Example 4.6 gives a typical construction of an unambiguous grammar for arithmetic expressions, although that grammar, which includes the unary minus operator, will turn out not to be an operator-precedence grammar (one for which the following construction works), unless unary minus is regarded as a distinct token.

We shall now define the term "operator-precedence grammar," show how to compute its precedence relations, and explain the details of shift-reduce parsing using precedence relations. To begin, let G be an ϵ-free operator grammar (i.e., no right side is ϵ and no right side side has a pair of adjacent nonterminals). For each two terminal symbols a and b, we say:

i) $a \doteq b$ if there is a right side of a production of the form $\alpha a \beta b \gamma$, where β is either ϵ or a single nonterminal. That is $a \doteq b$ if a appears immediately to the left of b in a right side, or if they appear separated by one nonterminal. For example, in grammar (5.1) of Example 5.1, the production $S \rightarrow iCtSeS$ implies that $i \doteq t$ and $t \doteq e$.

ii) $a <\cdot b$ if for some nonterminal A there is a right side of the form $\alpha a A \beta$, and $A \overset{+}{\Rightarrow} \gamma b \delta$, where γ is either ϵ or a single nonterminal. That is, $a <\cdot b$ if a nonterminal A appears immediately to the right of a and derives a string in which b is the first terminal symbol. For example, in grammar (5.1), we have i immediately to the left of C in $S \rightarrow iCtS$, and $C \overset{+}{\Rightarrow} b$, so $i <\cdot b$. The derivation is one step, and

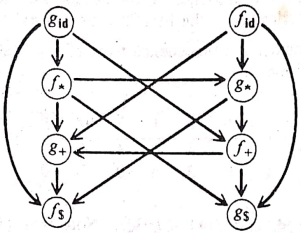


Fig. 5.15. Graph representing precedence functions.

	id	+	*	$
f	4	2	4	0
g	5	1	3	0

□

5.4 Top-Down Parsing

We now turn to another, rather different, parsing strategy, called top-down parsing. We first discuss a general form of top-down parsing that may involve backtracking, that is, making repeated scans of the input. We then discuss a special case, called recursive-descent parsing, which eliminates the need for backtracking over the input.

Top-down parsing can be viewed as an attempt to find a leftmost derivation for an input string. Equivalently, it can be viewed as attempting to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder. For example, consider the grammar

$$S \rightarrow cAd$$
$$A \rightarrow ab \mid a \tag{5.7}$$

and the input $w = cad$. To construct a parse tree for this sentence top-down, we initially create a tree consisting of a single node labeled S. An input pointer points to c, the first symbol of w. We then use the first production for S to expand the tree and obtain

The leftmost leaf, labeled c, matches the first symbol of w, so we now advance the input pointer to a, the second symbol of w, and consider the next leaf, labeled A. We can then expand A using the first alternate for A to obtain the tree

We now have a match for the second input symbol.

We now consider d, the third input symbol, and the next leaf, labeled b. Since b does not match d, we report failure and go back to A to see whether there is another alternate for A that we have not tried but which might produce a match.

In going back to A we must reset the input pointer to position 2, the position it had when we first came to A. We now try the second alternate for A to obtain the tree

The leaf a matches the second symbol of w and the leaf d matches the third symbol. Since we have now produced a parse tree for w, we halt and announce successful completion of parsing.

An easy way to implement such a parser is to create a procedure for each nonterminal. In the case of this simple grammar, generating only two strings, there is no need for recursion among the procedures, but in practical cases, where grammars derive an infinite number of strings, recursive procedures are essential.

In many compiler-writing systems based on top-down parsing with backtrack [e.g. META (Schorre, 1964) or TMG (McClure, 1965)], an interpreter is used to simulate a collection of recursive procedures. In Section 5.5 we shall discuss predictive parsers, which, in effect, enable us to interpret recursive procedures having no backtrack.

```
procedure S( );
begin
    if input symbol = 'c' then
        begin
            ADVANCE( );
            if A( ) then
                if input symbol = 'd' then
                    begin ADVANCE( ); return true end
        end;
    return false
end
```

(a) Procedure S.

```
procedure A( );
begin
    isave := input-pointer;
    if input symbol = 'a' then
        begin
            ADVANCE( );
            if input symbol = 'b' then
                begin ADVANCE( ); return true end
        end
    input-pointer := isave;
        /* failure to find ab */
    if input-symbol = 'a' then
        begin ADVANCE( ); return true end
    else return false
end
```

(b) Procedure A.

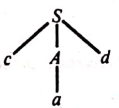
Fig. 5.16. Recursive procedures for top-down parsing.

Example 5.12. Let us suppose the procedure ADVANCE moves the input pointer to the next input symbol. "Input symbol" is the one currently pointed to by the input pointer. Figure 5.16 gives the procedures for S and A corresponding to the informal discussion above. Procedures return value **true** (success) or **false** (failure), depending on whether or not they have found on the input a string derived by the corresponding nonterminal. Note that, on failure, each procedure leaves the input pointer where it was when the procedure was invoked, and that, on success, it moves the input pointer over the substring recognized. □

There are several difficulties with top-down parsing as just presented. The first concerns left-recursion. A grammar G is said to be *left-recursive* if it has a nonterminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some α. A left-recursive grammar can cause a top-down parser to go into an infinite loop. That is, when we try to expand A, we may eventually find ourselves again trying to expand A without having consumed any input. This cycling will surely occur on an erroneous input string, and it may also occur on legal inputs, depending on the order in which the alternates for A are tried. Therefore, to use top-down parsing, we must eliminate all left-recursion from the grammar. We shall show how to do this shortly.

A second problem concerns backtracking. If we make a sequence of erroneous expansions and subsequently discover a mismatch, we may have to undo the semantic effects of making these erroneous expansions. For example, entries made in the symbol table might have to be removed. Since undoing semantic actions requires a substantial overhead, it is reasonable to consider top-down parsers that do no backtracking. The recursive-descent and predictive parsers discussed next are types of top-down parsers that avoid backtracking. They compensate somewhat for the lack of backtracking by using the next input symbol to guide parsing actions.

A third problem with top-down backtracking parsers is that the order in which alternates are tried can affect the language accepted. For example, if, in grammar (5.7), we used a and then ab as the order of the alternates for A, we could fail to accept *cabd*. That is, with parse tree



and *ca* already matched, the failure of the next input symbol, b, to match, would imply that the alternate *cAd* for S was wrong, leading to rejection of *cabd*.

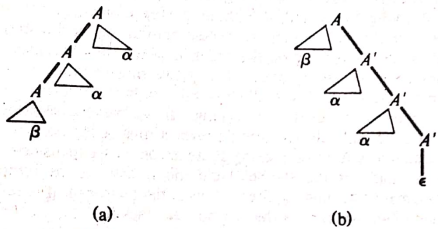
Yet another problem is that when failure is reported, we have very little idea where the error actually occurred. In the form given here, a top-down parser with backtrack simply returns failure no matter what the error is.

Elimination of Left-Recursion

If we have the left-recursive pair of productions $A \rightarrow A\alpha \mid \beta$, where β does not begin with an A, then we can eliminate the left-recursion by replacing this pair of productions with

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \epsilon$$

Notice that we do not change the set of strings derivable from A. Figure 5.17 illustrates the nature of this transformation. Parse tree (a) is in the original grammar, (b) in the new.



(a) (b)

Fig. 5.17. Equivalent parse trees.

Example 5.13. Consider the following grammar

$$\begin{aligned} E &\to E + T \mid T \\ T &\to T * F \mid F \\ F &\to (E) \mid \mathbf{id} \end{aligned} \tag{5.8}$$

Eliminating the *immediate left-recursion* (productions of the form $A \to A\alpha$), we obtain

$$\begin{aligned} E &\to TE' \\ E' &\to +TE' \mid \epsilon \\ T &\to FT' \\ T' &\to *FT' \mid \epsilon \\ F &\to (E) \mid \mathbf{id} \end{aligned} \tag{5.9}$$

□

In general, to eliminate immediate left-recursion among all A-productions we group the A productions as

$$A \to A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

where no β_i begins with an A. Then we replace the A-productions by

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

This process will eliminate all immediate left-recursion (provided no α_i is ϵ), but it will not eliminate left-recursion involving derivations of two or more steps. For example, consider

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Sd \mid e \tag{5.10}$$

The nonterminal S is left-recursive because $S \Rightarrow Aa \Rightarrow Sda$.

It is possible, although difficult, to eliminate all left-recursion from any grammar. Here we shall present an algorithm to eliminate left-recursion that is guaranteed to work if the grammar has no cycles (derivations of the form $A \overset{+}{\Rightarrow} A$) or ϵ-productions (productions of the form $A \rightarrow \epsilon$). The algorithm is presented in Fig. 5.18. Note that the resulting non-left-recursive grammar may have ϵ-productions.

1. Arrange the nonterminals of G in some order $A_1, A_2, \ldots, A_n$.
2. **for** $i := 1$ **to** n **do**
   begin
   for $j := 1$ **to** $i-1$ **do**
   replace each production of the form $A_i \rightarrow A_j\gamma$
   by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma$,
   where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ are all the
   current A_j-productions;
   eliminate the immediate left-recursion among the
   A_i-productions
   end

Fig. 5.18. Algorithm to eliminate left-recursion from a grammar with no cycles or ϵ-productions.

Example 5.14. Let us apply this procedure to grammar (5.10). We order the nonterminals S, A. There is no immediate left-recursion among the S-productions. We then substitute the S-productions in $A \rightarrow Sd$ to obtain the following A-productions.

$$A \rightarrow Ac \mid Aad \mid bd \mid e$$

Eliminating the immediate left-recursion among the A-productions yields the following grammar.

$$S \rightarrow Aa \mid b$$
$$A \rightarrow bdA' \mid eA'$$
$$A' \rightarrow cA' \mid adA' \mid \epsilon$$

□

Recursive-Descent Parsing

In many practical cases a top-down parser needs no backtrack. In order that no backtracking be required, we must know, given the current input symbol a and the nonterminal A to be expanded, which one of the alternates of production $A \to \alpha_1 | \alpha_2 | \cdots | \alpha_n$ is the unique alternate that derives a string beginning with a. That is, the proper alternate is detectable by looking at only the first symbol it derives. For example, control constructs, with their distinguishing keywords, are detectable in this way. Suppose we have productions:

statement → **if** condition **then** statement **else** statement
| **while** condition **do** statement
| **begin** statement-list **end**

Then the keywords **if**, **while**, and **begin** tell us which alternate is the only one that could possibly succeed if we are to find a statement.

One nuance concerns the empty string. If one alternate for A is ϵ, and none of the other alternates derives a string beginning with a, then on input a we may expand A by $A \to \epsilon$, that is, we succeed without further ado in recognizing an A.

A parser that uses a set of recursive procedures to recognize its input with no backtracking is called a *recursive-descent* parser. The recursive procedures can be quite easy to write and fairly efficient if written in a language that implements procedure calls efficiently.

To avoid the necessity of a recursive language, we shall also consider a tabular implementation of recursive descent, called predictive parsing, where a stack is maintained by the parser, rather than by the language in which the parser is written.

Example 5.15. Let us consider grammar (5.9), which is suitable for a nonbacktracking recursive-descent parser. The procedures making up this parser are shown in Fig. 5.19. The same conventions as in Example 5.12 are used, but there is no need for the procedures to return an indication of success or failure, since the calling procedure has no intention of trying another alternate. Rather, on failure, an error-correcting routine, which we here name ERROR, can be invoked. Section 11.3 discusses the sorts of error actions that could be taken to resume parsing. □

Left-Factoring

Often the grammar one writes down is not suitable for recursive-descent parsing, even if there is no left-recursion. For example, if we have the two productions

```
procedure E( );
begin
    T( );
    EPRIME( )
end;

procedure EPRIME( );
if input-symbol = '+' then
    begin
        ADVANCE( );
        T( );
        EPRIME( )
    end;

procedure T( );
begin
    F( );
    TPRIME( )
end;

procedure TPRIME( );
if input-symbol = '*' then
    begin
        ADVANCE( );
        F( );
        TPRIME( )
    end;

procedure F( );
if input-symbol = 'id' then
    ADVANCE( )
else if input-symbol = '(' then
    begin
        ADVANCE( );
        E( );
        if input-symbol = ')' then
            ADVANCE( )
        else ERROR( )
    end
else ERROR( )
```

Fig. 5.19. Mutually recursive procedures to recognize arithmetic expressions.

statement → **if** condition **then** statement **else** statement

| **if** condition **then** statement

we could not, on seeing input symbol **if**, tell which to choose to expand statement. A useful method for manipulating grammars into a form suitable for recursive-descent parsing is *left-factoring*, the process of factoring out the common prefixes of alternates.

If $A \rightarrow \alpha\beta \mid \alpha\gamma$ are two A-productions, and the input begins with a nonempty string derived from α, we do not know whether to expand A to $\alpha\beta$ or to $\alpha\gamma$. We may defer the decision by expanding A to $\alpha A'$. Then, after seeing the input derived from α, we expand A' to β or to γ. That is, left-factored, the original productions become

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta \mid \gamma$$

Example 5.16. Consider again the grammar

$$S \rightarrow iCtS \mid iCtSeS \mid a$$
$$C \rightarrow b$$

which abstracts the dangling **else** problem. Left-factored, this grammar becomes:

$$S \rightarrow iCtSS' \mid a$$
$$S' \rightarrow eS \mid \epsilon$$
$$C \rightarrow b$$

Thus we may expand S to $iCtSS'$ on input i, and wait until $iCtS$ has been seen to decide whether to expand S' to eS or to ϵ. □

Transition Diagrams

Just as a transition diagram was seen in Section 3.2 to be a useful plan, or flowchart, for a lexical analyzer, we can use a transition diagram as a plan for a recursive-descent parser. Several differences are immediately apparent. In the case of the parser, there is one transition diagram for each nonterminal. The labels of edges are not characters but tokens or nonterminals. A transition on a token (terminal) means we should take that transition if that token is the next input symbol. Edges may also be labeled by nonterminals A, implying that the transition diagram for A should be "called."

There is no guarantee that one can correctly choose a path through a transition diagram, although if we eliminate left-recursion and then left-

factor, we have a fair chance of success if we do the following for each nonterminal A:

1. Create an initial and final (return) state.
2. For each production $A \rightarrow X_1 X_2 \cdots X_n$, create a path from the initial to the final state, with edges labeled $X_1, X_2, \ldots, X_n$.

Example 5.17. Grammar (5.9) has the collection of transition diagrams shown in Fig. 5.20. If we interpret the edges out of the initial state for E' as saying: take the transition on + whenever that is the next input and take the transition on ϵ otherwise, and make the analogous assumption for T', we have a program that functions as a correct recursive-descent parser for grammar (5.9). □

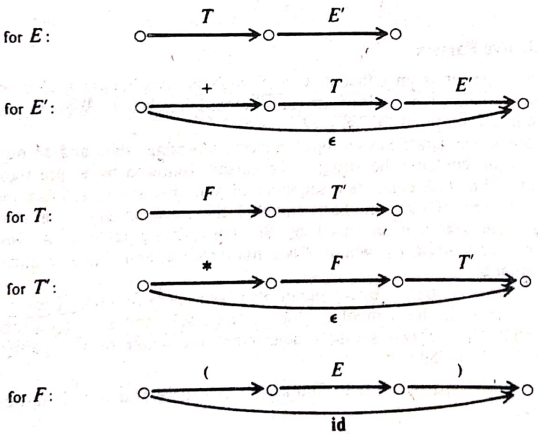


Fig. 5.20. Transition diagrams.

It should be observed that if there is "nondeterminism" in which there are two transitions out of some state labeled by the same symbol, or there are transitions labeled by a terminal symbol and a nonterminal symbol, then we cannot necessarily apply the subset construction of Chapter 3 to make the transition diagrams deterministic. Intuitively, the reason the subset construction does not work is that it cannot remember how many

recursive calls are made. In cases with nondeterminism, the method fails to produce a recursive-descent parser, but it may be used to help design one with backtrack.

It is possible to simplify transition diagrams by substituting diagrams in one another. For example, in Fig. 5.20, the call of E' on itself can be replaced by a jump to the beginning of the diagram for E', as shown in Fig. 5.21(a).

Fig. 5.21(b) shows an equivalent transition diagram for E'. We may then substitute the diagram of Fig. 5.21(b) for the transition on E' in the diagram for E, yielding the diagram of Fig. 5.21(c). Lastly, we observe that the first and third nodes in Fig. 5.21(c) are equivalent and we merge them.

The same techniques apply to the diagrams for T and T'. The resulting set of diagrams is shown in Fig. 5.21(d).

5.5 Predictive Parsers

A predictive parser is an efficient way of implementing recursive-descent parsing by handling the stack of activation records explicitly. We can picture a predictive parser as in Fig. 5.22.

The predictive parser has an input, a stack, a parsing table, and an output. The input contains the string to be parsed, followed by \$, the right endmarker. The stack contains a sequence of grammar symbols, preceded by \$, the bottom-of-stack marker. Initially, the stack contains the start symbol of the grammar preceded by \$. The parsing table is a two-dimensional array $M[A, a]$, where A is a nonterminal, and a is a terminal or the symbol \$.

The parser is controlled by a program that behaves as follows. The program determines X, the symbol on top of the stack, and a, the current input symbol. These two symbols determine the action of the parser. There are three possibilities.

1. If $X = a = \$$, the parser halts and announces successful completion of parsing.
2. If $X = a \neq \$$, the parser pops X off the stack and advances the input pointer to the next input symbol.
3. If X is a nonterminal, the program consults entry $M[X, a]$ of the parsing table M. This entry will be either an X-production of the grammar or an error entry. If $M[X, a] = \{X \rightarrow UVW\}$, the parser replaces X on top of the stack by WVU (with U on top). As output, the grammar does the semantic action associated with this production, which, for the time being, we shall assume is just printing the production used. If $M[X, a] =$ error, the parser calls an error recovery routine.

blank

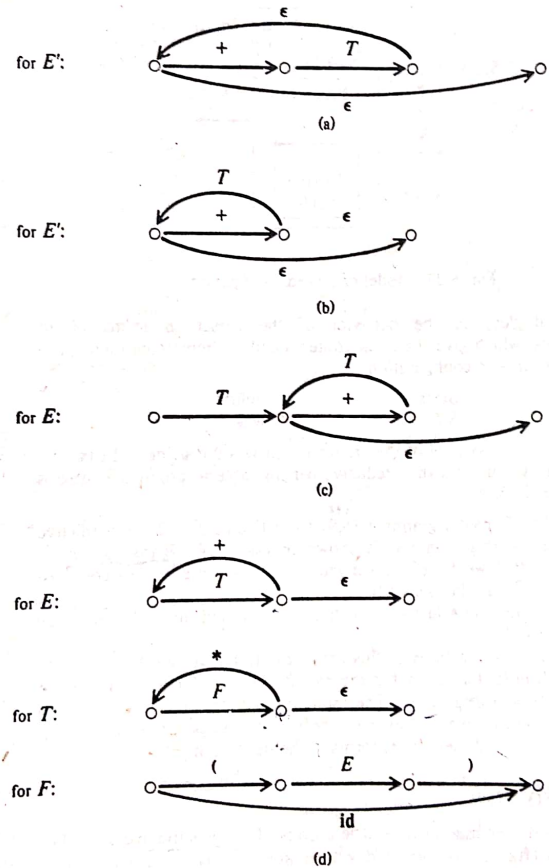


Fig. 5.21. Revised transition diagrams.

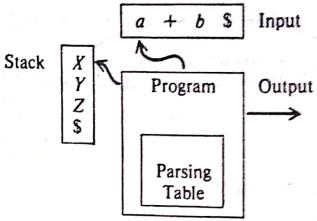


Fig. 5.22. Model of a predictive parser.

We shall describe the behavior of the parser in terms of its configurations, which give the stack contents and the remaining input. Initially, the parser is in configuration

Stack	Input
$ S	w $

where S is the start symbol of the grammar and w is the string to be parsed. The program that utilizes the predictive parsing table to produce a parse is shown in Fig. 5.23.

Example 5.18. Consider grammar (5.9) from Example 5.13. A predictive parsing table for this grammar is shown in Fig. 5.24. Blanks are error entries. Note that we have not yet indicated how these entries could be selected, but we shall do so shortly.

With input **id + id * id** the predictive parser would make the sequence of moves in Fig. 5.25.

If we observe the actions of this parser carefully, we see that it is tracing out a leftmost derivation for the input. The productions output are those of a leftmost derivation. The input symbols that have already been scanned, followed by the grammar symbols on the stack (from top to bottom), make up the left-sentential forms in the derivation. □

FIRST and FOLLOW

Now let us consider how to fill in the entries of a predictive parsing table. We need two functions associated with a grammar G. These functions, FIRST and FOLLOW, will indicate the proper entries in the table for G, if such a parsing table for G exists. If α is any string of grammar symbols, let $FIRST(\alpha)$ be the set of terminals that begin strings derived from α. If $\alpha \overset{*}{\Rightarrow} \epsilon$, then ϵ is also in FIRST(α).

```
repeat
    begin
        let X be the top stack symbol and a the next input symbol;
        if X is a terminal or $ then
            if X = a then
                pop X from the stack and remove a from the input
            else
                ERROR( )
        else /* X is a nonterminal */
            if M[X, a] = X → Y1 Y2 ··· Yk then
                begin
                    pop X from the stack;
                    push Yk, Yk-1, . . . , Y1 onto the stack, Y1 on top
                end
            else
                ERROR( )
    end
until
    X = $ /* stack has emptied */
```

Fig. 5.23. Predictive parsing program.

	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ id			$F \to (E)$		

Fig. 5.24. Parsing table for grammar (5.9).

Define $FOLLOW(A)$, for nonterminal A, to be the set of terminals a that can appear immediately to the right of A in some sentential form, that is, $S \Rightarrow \alpha A a \beta$ for some α and β. If A can be the rightmost symbol in some sentential form, then we add $ to FOLLOW(A).

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ϵ can be added to any FIRST set.

Stack	Input	Output
$E	id + id * id$	
$ET	id + id * id$	$E \rightarrow TE'$
$ETF	id + id * id$	$T \rightarrow FT'$
$ET'id	id + id * id$	$F \rightarrow$ id
$ET'	+ id * id$	
$E'	+ id * id$	$T' \rightarrow \epsilon$
$ET+	+ id * id$	$E' \rightarrow +TE'$
$ET	id * id$	
$ETF	id * id$	$T \rightarrow FT'$
$ET'id	id * id$	$F \rightarrow$ id
$ET'	* id$	
$ETF*	* id$	$T' \rightarrow *FT'$
$ETF	id$	
$ET'id	id$	$F \rightarrow$ id
$ET'	$	
$E'	$	$T' \rightarrow \epsilon$
$	$	$E' \rightarrow \epsilon$

Fig. 5.25. Moves by predictive parser.

1. If X is terminal, then FIRST(X) is $\{X\}$.
2. If X is nonterminal and $X \rightarrow a\alpha$ is a production, then add a to FIRST(X). If $X \rightarrow \epsilon$ is a production, then add ϵ to FIRST(X).
3. If $X \rightarrow Y_1 Y_2 \cdots Y_k$ is a production, then for all i such that all of $Y_1, \ldots, Y_{i-1}$ are nonterminals and FIRST(Y_j) contains ϵ for $j = 1, 2, \ldots, i-1$ (i.e. $Y_1 Y_2 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$), add every non-$\epsilon$ symbol in FIRST(Y_i) to FIRST(X). If ϵ is in FIRST(Y_j), for all $j = 1, 2, \ldots, k$, then add ϵ to FIRST(X).

Now, we can compute FIRST for any string $X_1 X_2 \cdots X_n$ as follows. Add to FIRST($X_1 X_2 \cdots X_n$) all the non-ϵ symbols of FIRST(X_1). Also add the non-ϵ symbols of FIRST(X_2) if ϵ is in FIRST(X_1), the non-ϵ symbols of FIRST(X_3) if ϵ is in both FIRST(X_1) and FIRST(X_2), and so on. Finally, add ϵ to FIRST($X_1 X_2 \cdots X_n$) if, for all i, FIRST(X_i) contains ϵ, or if $n = 0$.

To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. \$ is in FOLLOW(S), where S is the start symbol.
2. If there is a production $A \rightarrow \alpha B \beta$, $\beta \neq \epsilon$, then everything in FIRST(β) but ϵ is in FOLLOW(B).
3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains ϵ (i.e., $\beta \stackrel{*}{\Rightarrow} \epsilon$), then everything in FOLLOW(A) is in FOLLOW(B).

Example 5.19. Consider again grammar (5.9)

$$
\begin{aligned}
E &\rightarrow TE' \\
E' &\rightarrow +TE' \mid \epsilon \\
T &\rightarrow FT' \\
T' &\rightarrow *FT' \mid \epsilon \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

Then:

$$
\begin{aligned}
&\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{(, \mathbf{id}\}. \\
&\text{FIRST}(E') = \{+, \epsilon\} \\
&\text{FIRST}(T') = \{*, \epsilon\} \\
&\text{FOLLOW}(E) = \text{FOLLOW}(E') = \{), \$\} \\
&\text{FOLLOW}(T) = \text{FOLLOW}(T') = \{+,), \$\} \\
&\text{FOLLOW}(F) = \{+, *,), \$\}
\end{aligned}
$$

For example, **id** and left parenthesis are added to FIRST(F) by rule (2) in the definition of FIRST. Then by rule (3) with $i = 1$, the production $T \rightarrow FT'$ implies that **id** and left parenthesis are in FIRST(T) as well.

To compute FOLLOW sets, we put \$ in FOLLOW(E) by rule (1). By rule (2) applied to production $F \rightarrow (E)$, the right parenthesis is also in FOLLOW(E). By rule (3) applied to production $E \rightarrow TE'$, \$ and right parenthesis are in FOLLOW(E'). Since $E' \stackrel{*}{\Rightarrow} \epsilon$, they are also in FOLLOW(T). For a last example of how the FOLLOW rules are applied, the production $E \rightarrow TE'$ implies, by rule (2), that everything other than ϵ in FIRST(E') must be placed in FOLLOW(T). We have already seen that \$ is in FOLLOW(T). □

Construction of Parsing Tables

The following algorithm can be used to construct a predictive parsing table for a grammar G. The idea behind the algorithm is simple. Suppose $A \to \alpha$ is a production with a in FIRST(α). Then, whenever the parser has A on top of the stack with a the current input symbol, the parser will expand A by α. The only complication occurs when $\alpha = \epsilon$ or $\alpha \stackrel{*}{\Rightarrow} \epsilon$. In this case, we should also expand A by α if the current input symbol is in FOLLOW(A), or if the \$ on the input has been reached and \$ is in FOLLOW(A).

Algorithm 5.4. Constructing a predictive parsing table.

Input. Grammar G.

Output. Parsing table M.

Method.

1. For each production $A \to \alpha$ of the grammar, do steps 2 and 3.
2. For each terminal a in FIRST(α), add $A \to \alpha$ to $M[A, a]$.
3. If ϵ is in FIRST(α), add $A \to \alpha$ to $M[A, b]$ for each terminal b in FOLLOW(A). If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \to \alpha$ to $M[A, \$]$.
4. Make each undefined entry of M **error**. □

Example 5.20. Let us apply Algorithm 5.4 to grammar (5.9). Since FIRST(TE') = $FIRST(T)$ = {(, **id**}, production $E \to TE'$ causes $M[E, (]$ and $M[E,$ **id**$]$ to acquire the entry $E \to TE'$.

Production $E' \to +TE'$ causes $M[E', +]$ to acquire $E' \to +TE'$. Production $E' \to \epsilon$ causes $M[E',)]$ and $M[E', \$]$ to acquire $E' \to \epsilon$ since FOLLOW(E') = {), \$}.

The parsing table produced by Algorithm 5.4 for G was shown in Fig. 5.24. □

LL(1) Grammars

Algorithm 5.4 can be applied to any grammar G to produce a parsing table M. For some grammars, however, M may have some entries that are multiply-defined. For example, if G is left-recursive or ambiguous, then M will have at least one multiply-defined entry.

Example 5.21. Consider the grammar from Example 5.16.

$$\begin{aligned} S &\to iCtSS' \mid a \\ S' &\to eS \mid \epsilon \\ C &\to b \end{aligned} \tag{5.11}$$

The parsing table for grammar (5.11) is shown in Fig. 5.26.

	a	b	e	i	t	$
S	$S \rightarrow a$			$S \rightarrow iCtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
C		$C \rightarrow b$				

Fig. 5.26. Parsing table.

The entry for $M[S', e]$ contains both $S' \rightarrow eS$ and $S' \rightarrow \epsilon$, since FOLLOW$(S') = \{e, \$\}$. The grammar is ambiguous and the ambiguity is manifested by a choice in what production to use when an e (**else**) is seen. We can resolve the ambiguity if we choose $S' \rightarrow eS$. This choice corresponds to associating **else**'s with the closest previous **then**'s. Note that the choice $S' \rightarrow \epsilon$ would prevent e from ever being put on the stack or removed from the input, and is therefore surely wrong. □

There arises the question of what should be done when a parsing table has multiply-defined entries. The easiest recourse is to transform the grammar by eliminating all left-recursion and then left-factoring whenever possible, hopefully to produce a grammar for which the parsing table has no multiply-defined entries.

A grammar whose parsing table has no multiply-defined entries is said to be *LL*(1). It can be shown that Algorithm 5.4 produces a parsing table for every LL(1) grammar, and that this table parses all and only the sentences of the grammar. It can also be shown that a grammar G is LL(1) if and only if whenever $A \rightarrow \alpha \mid \beta$ are two distinct productions of G the following conditions hold:

1. For no terminal a do α and β derive strings beginning with a.
2. At most one of α and β can derive the empty string.
3. If $\beta \overset{*}{\Rightarrow} \epsilon$, then α does not derive any strings beginning with a terminal in FOLLOW(A).

Clearly, the grammar of Example 5.19 for arithmetic expressions is LL(1). The grammar (5.11) of Example 5.21 modeling **if-then-else** statements is not.

Unfortunately, there are some grammars for which no amount of rewriting will yield an LL(1) grammar. Grammar (5.11) is one such example. As we saw, we can still parse (5.11) with a predictive parser by arbitrarily making $M[S', e] = \{S' \rightarrow eS\}$. In general, however, there are no universal rules by which multiply-defined entries can be made single-valued without affecting the language recognized by the parser.