CHAPTER 1

Introduction to Compiling

The principles and techniques of compiler writing are so pervasive that the ideas found in this book will be used many times in the career of a computer scientist. Compiler writing spans programming languages, machine architecture, language theory, algorithms, and software engineering. Fortunately, a few basic compiler-writing techniques can be used to construct translators for a wide variety of languages and machines. In this chapter, we introduce the subject of compiling by describing the components of a compiler, the environment in which compilers do their job, and some software tools that make it easier to build compilers.

1.1 COMPILERS

Simply stated, a compiler is a program that reads a program written in one language – the *source* language – and translates it into an equivalent program in another language – the *target* language (see Fig. 1.1). As an important part of this translation process, the compiler reports to its user the presence of errors in the source program.

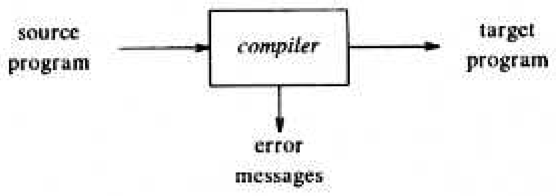


Fig. 1.1. A compiler.

At first glance, the variety of compilers may appear overwhelming. There are thousands of source languages, ranging from traditional programming languages such as Fortran and Pascal to specialized languages that have arisen in virtually every area of computer application. Target languages are equally as varied; a target language may be another programming language, or the machine language of any computer between a microprocessor and a supercomputer. Compilers are sometimes classified as single-pass, multi-pass, load-and-go, debugging, or optimizing, depending on how they have been constructed or on what function they are supposed to perform. Despite this apparent complexity, the basic tasks that any compiler must perform are essentially the same. By understanding these tasks, we can construct compilers for a wide variety of source languages and target machines using the same basic techniques.

Our knowledge about how to organize and write compilers has increased vastly since the first compilers started to appear in the early 1950's. It is difficult to give an exact date for the first compiler because initially a great deal of experimentation and implementation was done independently by several groups. Much of the early work on compiling dealt with the translation of arithmetic formulas into machine code.

Throughout the 1950's, compilers were considered notoriously difficult programs to write. The first Fortran compiler, for example, took 18 staff-years to implement (Backus et al. [1957]). We have since discovered systematic techniques for handling many of the important tasks that occur during compilation. Good implementation languages, programming environments, and software tools have also been developed. With these advances, a substantial compiler can be implemented even as a student project in a one-semester compiler-design course.

The Analysis-Synthesis Model of Compilation

There are two parts to compilation: analysis and synthesis. The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program. The synthesis part constructs the desired target program from the intermediate representation. Of the two parts, synthesis requires the most specialized techniques. We shall consider analysis informally in Section 1.2 and outline the way target code is synthesized in a standard compiler in Section 1.3.

During analysis, the operations implied by the source program are determined and recorded in a hierarchical structure called a tree. Often, a special kind of tree called a syntax tree is used, in which each node represents an operation and the children of a node represent the arguments of the operation. For example, a syntax tree for an assignment statement is shown in Fig. 1.2.

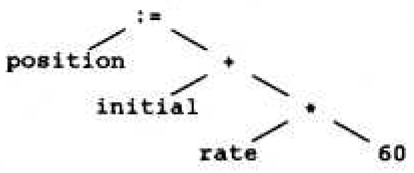


Fig. 1.2. Syntax tree for `position := initial + rate * 60`.

Many software tools that manipulate source programs first perform some kind of analysis. Some examples of such tools include:

1. *Structure editors.* A structure editor takes as input a sequence of commands to build a source program. The structure editor not only performs the text-creation and modification functions of an ordinary text editor, but it also analyzes the program text, putting an appropriate hierarchical structure on the source program. Thus, the structure editor can perform additional tasks that are useful in the preparation of programs. For example, it can check that the input is correctly formed, can supply keywords automatically (e.g., when the user types `while`, the editor supplies the matching `do` and reminds the user that a conditional must come between them), and can jump from a `begin` or left parenthesis to its matching `end` or right parenthesis. Further, the output of such an editor is often similar to the output of the analysis phase of a compiler.

2. *Pretty printers.* A pretty printer analyzes a program and prints it in such a way that the structure of the program becomes clearly visible. For example, comments may appear in a special font, and statements may appear with an amount of indentation proportional to the depth of their nesting in the hierarchical organization of the statements.

3. *Static checkers.* A static checker reads a program, analyzes it, and attempts to discover potential bugs without running the program. The analysis portion is often similar to that found in optimizing compilers of the type discussed in Chapter 10. For example, a static checker may detect that parts of the source program can never be executed, or that a certain variable might be used before being defined. In addition, it can catch logical errors such as trying to use a real variable as a pointer, employing the type-checking techniques discussed in Chapter 6.

4. *Interpreters.* Instead of producing a target program as a translation, an interpreter performs the operations implied by the source program. For an assignment statement, for example, an interpreter might build a tree like Fig. 1.2, and then carry out the operations at the nodes as it "walks" the tree. At the root it would discover it had an assignment to perform, so it would call a routine to evaluate the expression on the right, and then store the resulting value in the location associated with the identifier `position`. At the right child of the root, the routine would discover it had to compute the sum of two expressions. It would call itself recursively to compute the value of the expression `rate * 60`. It would then add that value to the value of the variable `initial`.

   Interpreters are frequently used to execute command languages, since each operator executed in a command language is usually an invocation of a complex routine such as an editor or compiler. Similarly, some "very high-level" languages, like APL, are normally interpreted because there are many things about the data, such as the size and shape of arrays, that cannot be deduced at compile time.

Traditionally, we think of a compiler as a program that translates a source language like Fortran into the assembly or machine language of some computer. However, there are seemingly unrelated places where compiler technology is regularly used. The analysis portion in each of the following examples is similar to that of a conventional compiler.

1. *Text formatters.* A text formatter takes input that is a stream of characters, most of which is text to be typeset, but some of which includes commands to indicate paragraphs, figures, or mathematical structures like subscripts and superscripts. We mention some of the analysis done by text formatters in the next section.

2. *Silicon compilers.* A silicon compiler has a source language that is similar or identical to a conventional programming language. However, the variables of the language represent, not locations in memory, but, logical signals (0 or 1) or groups of signals in a switching circuit. The output is a circuit design in an appropriate language. See Johnson [1983], Ullman [1984], or Trickey [1985] for a discussion of silicon compilation.

3. *Query interpreters.* A query interpreter translates a predicate containing relational and boolean operators into commands to search a database for records satisfying that predicate. (See Ullman [1982] or Date [1986].)

The Context of a Compiler

In addition to a compiler, several other programs may be required to create an executable target program. A source program may be divided into modules stored in separate files. The task of collecting the source program is sometimes entrusted to a distinct program, called a preprocessor. The preprocessor may also expand shorthands, called macros, into source language statements.

Figure 1.3 shows a typical "compilation." The target program created by the compiler may require further processing before it can be run. The compiler in Fig. 1.3 creates assembly code that is translated by an assembler into machine code and then linked together with some library routines into the code that actually runs on the machine.

We shall consider the components of a compiler in the next two sections; the remaining programs in Fig. 1.3 are discussed in Section 1.4.

1.2 ANALYSIS OF THE SOURCE PROGRAM

In this section, we introduce analysis and illustrate its use in some text-formatting languages. The subject is treated in more detail in Chapters 2-4 and 6. In compiling, analysis consists of three phases:

1. *Linear analysis*, in which the stream of characters making up the source program is read from left-to-right and grouped into *tokens* that are sequences of characters having a collective meaning.

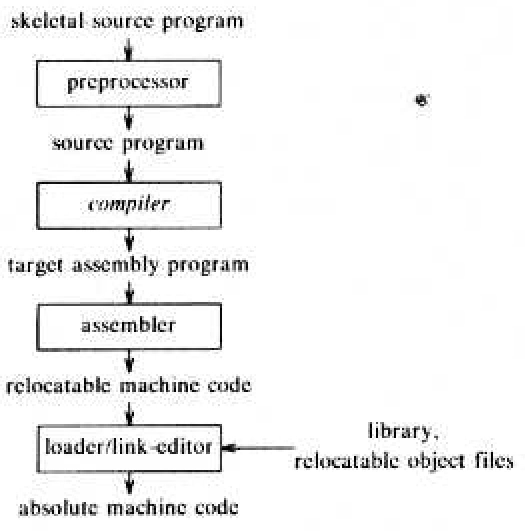


Fig. 1.3. A language-processing system.

2. *Hierarchical analysis*, in which characters or tokens are grouped hierarchically into nested collections with collective meaning.
3. *Semantic analysis*, in which certain checks are performed to ensure that the components of a program fit together meaningfully.

Lexical Analysis

In a compiler, linear analysis is called *lexical analysis* or *scanning*. For example, in lexical analysis the characters in the assignment statement

```
    position := initial + rate * 60
```

would be grouped into the following tokens:

1. The identifier `position`.
2. The assignment symbol `:=`.
3. The identifier `initial`.
4. The plus sign.
5. The identifier `rate`.
6. The multiplication sign.
7. The number `60`.

The blanks separating the characters of these tokens would normally be eliminated during lexical analysis.

Syntax Analysis

Hierarchical analysis is called *parsing* or *syntax analysis*. It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output. Usually, the grammatical phrases of the source program are represented by a parse tree such as the one shown in Fig. 1.4.

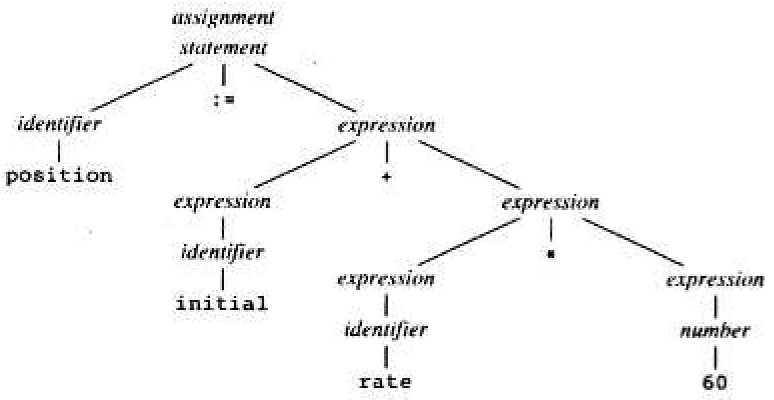


Fig. 1.4. Parse tree for `position := initial + rate * 60`.

In the expression `initial + rate * 60`, the phrase `rate * 60` is a logical unit because the usual conventions of arithmetic expressions tell us that multiplication is performed before addition. Because the expression `initial + rate` is followed by a `*`, it is not grouped into a single phrase by itself in Fig. 1.4.

The hierarchical structure of a program is usually expressed by recursive rules. For example, we might have the following rules as part of the definition of expressions:

1. Any *identifier* is an expression.
2. Any *number* is an expression.
3. If $expression_1$ and $expression_2$ are expressions, then so are

   $expression_1$ + $expression_2$
   $expression_1$ * $expression_2$
   ($expression_1$)

Rules (1) and (2) are (nonrecursive) basis rules, while (3) defines expressions in terms of operators applied to other expressions. Thus, by rule (1), `initial` and `rate` are expressions. By rule (2), `60` is an expression, while by rule (3), we can first infer that `rate * 60` is an expression and finally that `initial + rate * 60` is an expression.

Similarly, many languages define statements recursively by rules such as:

1. If *identifier*$_1$ is an identifier, and *expression*$_2$ is an expression, then

   identifier$_1$:= *expression*$_2$

   is a statement.

2. If *expression*$_1$ is an expression and *statement*$_2$ is a statement, then

   while (*expression*$_1$) **do** *statement*$_2$
   if (*expression*$_1$) **then** *statement*$_2$

   are statements.

The division between lexical and syntactic analysis is somewhat arbitrary. We usually choose a division that simplifies the overall task of analysis. One factor in determining the division is whether a source language construct is inherently recursive or not. Lexical constructs do not require recursion, while syntactic constructs often do. Context-free grammars are a formalization of recursive rules that can be used to guide syntactic analysis. They are introduced in Chapter 2 and studied extensively in Chapter 4.

For example, recursion is not required to recognize identifiers, which are typically strings of letters and digits beginning with a letter. We would normally recognize identifiers by a simple scan of the input stream, waiting until a character that was neither a letter nor a digit was found, and then grouping all the letters and digits found up to that point into an identifier token. The characters so grouped are recorded in a table, called a symbol table, and removed from the input so that processing of the next token can begin.

On the other hand, this kind of linear scan is not powerful enough to analyze expressions or statements. For example, we cannot properly match parentheses in expressions, or begin and end in statements, without putting some kind of hierarchical or nesting structure on the input.

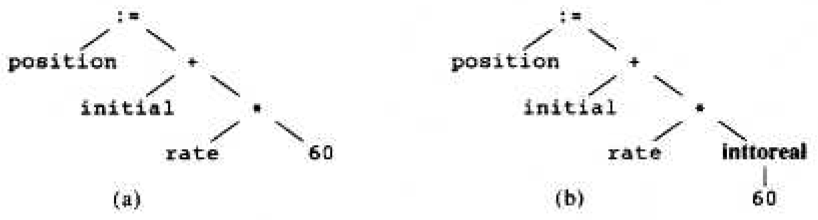


Fig. 1.5. Semantic analysis inserts a conversion from integer to real.

The parse tree in Fig. 1.4 describes the syntactic structure of the input. A more common internal representation of this syntactic structure is given by the syntax tree in Fig. 1.5(a). A syntax tree is a compressed representation of the parse tree in which the operators appear as the interior nodes, and the operands of an operator are the children of the node for that operator. The construction of trees such as the one in Fig. 1.5(a) is discussed in Section 5.2.

We shall take up in Chapter 2, and in more detail in Chapter 5, the subject of *syntax-directed translation*, in which the compiler uses the hierarchical structure on the input to help generate the output.

Semantic Analysis

The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent code-generation phase. It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operands of expressions and statements.

An important component of semantic analysis is type checking. Here the compiler checks that each operator has operands that are permitted by the source language specification. For example, many programming language definitions require a compiler to report an error every time a real number is used to index an array. However, the language specification may permit some operand coercions, for example, when a binary arithmetic operator is applied to an integer and real. In this case, the compiler may need to convert the integer to a real. Type checking and semantic analysis are discussed in Chapter 6.

Example 1.1. Inside a machine, the bit pattern representing an integer is generally different from the bit pattern for a real, even if the integer and the real number happen to have the same value. Suppose, for example, that all identifiers in Fig. 1.5 have been declared to be reals and that `60` by itself is assumed to be an integer. Type checking of Fig. 1.5(a) reveals that `*` is applied to a real, `rate`, and an integer, `60`. The general approach is to convert the integer into a real. This has been achieved in Fig. 1.5(b) by creating an extra node for the operator **inttoreal** that explicitly converts an integer into a real. Alternatively, since the operand of **inttoreal** is a constant, the compiler may instead replace the integer constant by an equivalent real constant. □

MO

Analysis in Text Formatters

It is useful to regard the input to a text formatter as specifying a hierarchy of *boxes* that are rectangular regions to be filled by some bit pattern, representing light and dark pixels to be printed by the output device.

For example, the TeX system (Knuth [1984a]) views its input this way. Each character that is not part of a command represents a box containing the bit pattern for that character in the appropriate font and size. Consecutive characters not separated by "white space" (blanks or newline characters) are grouped into words, consisting of a sequence of horizontally arranged boxes, shown schematically in Fig. 1.6. The grouping of characters into words (or commands) is the linear or lexical aspect of analysis in a text formatter.

Boxes in TeX may be built from smaller boxes by arbitrary horizontal and vertical combinations. For example,

```
\hbox{ <list of boxes> }
```

```
a sub {i sup 2}
```

results in a_{i^2}. Grouping the operators **sub** and **sup** into tokens is part of the lexical analysis of EQN text. However, the syntactic structure of the text is needed to determine the size and placement of a box.

1.3 THE PHASES OF A COMPILER

Conceptually, a compiler operates in *phases*, each of which transforms the source program from one representation to another. A typical decomposition of a compiler is shown in Fig. 1.9. In practice, some of the phases may be grouped together, as mentioned in Section 1.5, and the intermediate representations between the grouped phases need not be explicitly constructed.

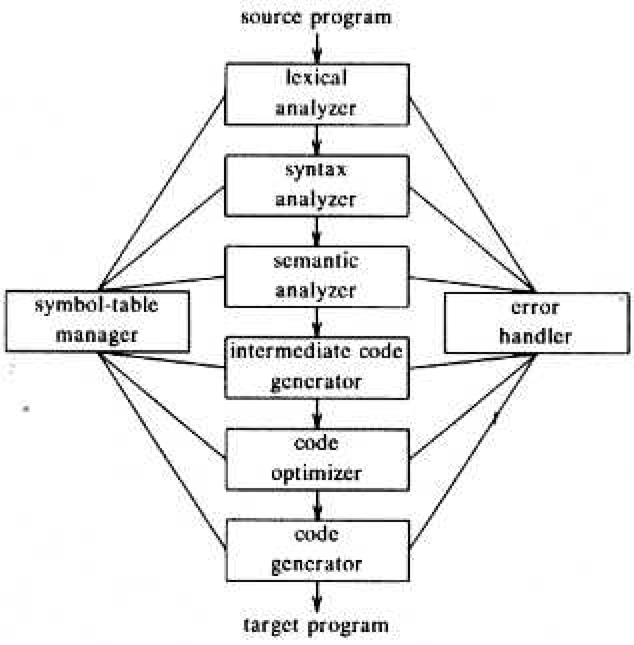


Fig. 1.9. Phases of a compiler.

The first three phases, forming the bulk of the analysis portion of a compiler, were introduced in the last section. Two other activities, symbol-table management and error handling, are shown interacting with the six phases of lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization, and code generation. Informally, we shall also call the symbol-table manager and the error handler "phases."

Symbol-Table Management

An essential function of a compiler is to record the identifiers used in the source program and collect information about various attributes of each identifier. These attributes may provide information about the storage allocated for an identifier, its type, its scope (where in the program it is valid), and, in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (e.g., by reference), and the type returned, if any.

A *symbol table* is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly. Symbol tables are discussed in Chapters 2 and 7.

When an identifier in the source program is detected by the lexical analyzer, the identifier is entered into the symbol table. However, the attributes of an identifier cannot normally be determined during lexical analysis. For example, in a Pascal declaration like

```
var position, initial, rate : real ;
```

the type real is not known when `position`, `initial`, and `rate` are seen by the lexical analyzer.

The remaining phases enter information about identifiers into the symbol table and then use this information in various ways. For example, when doing semantic analysis and intermediate code generation, we need to know what the types of identifiers are, so we can check that the source program uses them in valid ways, and so that we can generate the proper operations on them. The code generator typically enters and uses detailed information about the storage assigned to identifiers.

Error Detection and Reporting

Each phase can encounter errors. However, after detecting an error, a phase must somehow deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected. A compiler that stops when it finds the first error is not as helpful as it could be.

The syntax and semantic analysis phases usually handle a large fraction of the errors detectable by the compiler. The lexical phase can detect errors where the characters remaining in the input do not form any token of the language. Errors where the token stream violates the structure rules (syntax) of the language are determined by the syntax analysis phase. During semantic analysis the compiler tries to detect constructs that have the right syntactic structure but no meaning to the operation involved, e.g., if we try to add two identifiers, one of which is the name of an array, and the other the name of a procedure. We discuss the handling of errors by each phase in the part of the book devoted to that phase.

The Analysis Phases

As translation progresses, the compiler's internal representation of the source program changes. We illustrate these representations by considering the translation of the statement

```
    position := initial + rate * 60                          (1.1)
```

Figure 1.10 shows the representation of this statement after each phase.

The lexical analysis phase reads the characters in the source program and groups them into a stream of tokens in which each token represents a logically cohesive sequence of characters, such as an identifier, a keyword (`if`, `while`, etc.), a punctuation character, or a multi-character operator like `:=`. The character sequence forming a token is called the *lexeme* for the token.

Certain tokens will be augmented by a "lexical value." For example, when an identifier like `rate` is found, the lexical analyzer not only generates a token, say **id**, but also enters the lexeme `rate` into the symbol table, if it is not already there. The lexical value associated with this occurrence of **id** points to the symbol-table entry for `rate`.

In this section, we shall use $\mathbf{id}_1$, $\mathbf{id}_2$, and $\mathbf{id}_3$ for `position`, `initial`, and `rate`, respectively, to emphasize that the internal representation of an identifier is different from the character sequence forming the identifier. The representation of (1.1) after lexical analysis is therefore suggested by:

$$\mathbf{id}_1 \ := \ \mathbf{id}_2 \ + \ \mathbf{id}_3 \ * \ 60 \tag{1.2}$$

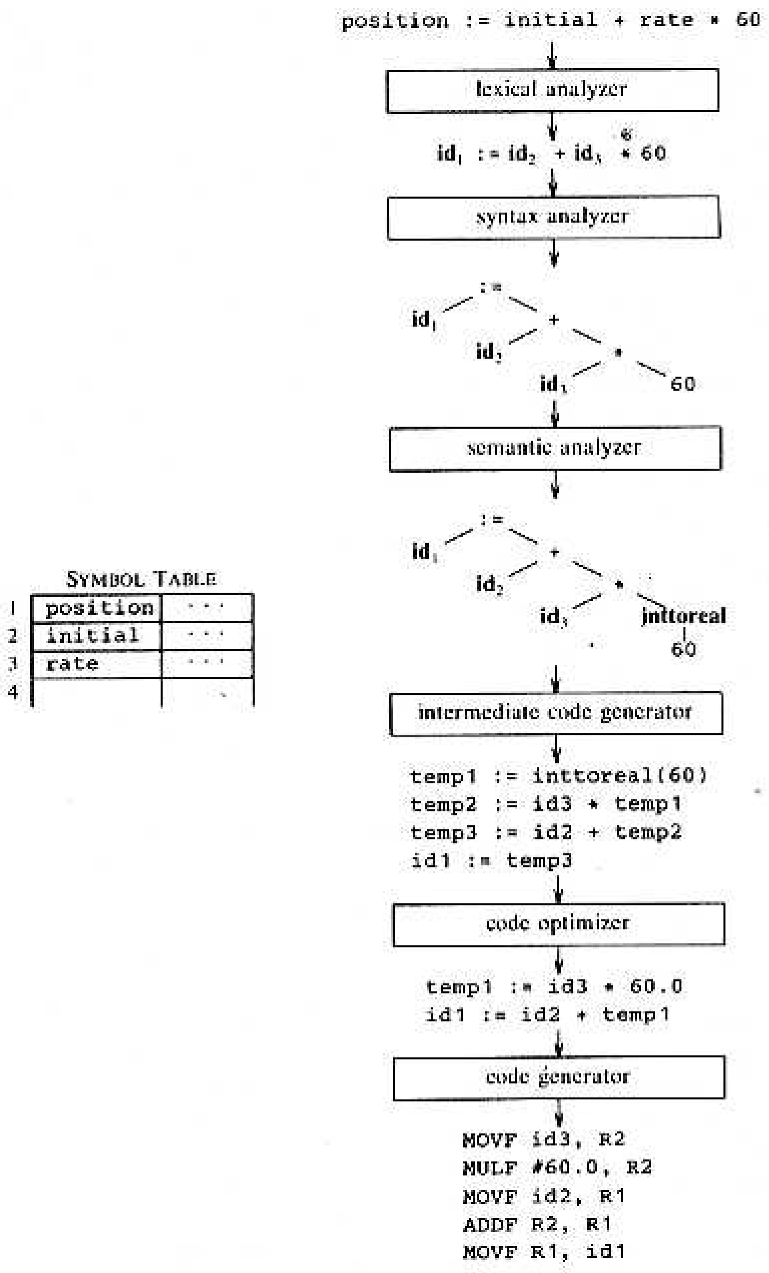
We should also make up tokens for the multi-character operator `:=` and the number `60` to reflect their internal representation, but we defer that until Chapter 2. Lexical analysis is covered in detail in Chapter 3.

The second and third phases, syntax and semantic analysis, have also been introduced in Section 1.2. Syntax analysis imposes a hierarchical structure on the token stream, which we shall portray by syntax trees as in Fig. 1.11(a). A typical data structure for the tree is shown in Fig. 1.11(b) in which an interior node is a record with a field for the operator and two fields containing pointers to the records for the left and right children. A leaf is a record with two or more fields, one to identify the token at the leaf, and the others to record information about the token. Additional information about language constructs can be kept by adding more fields to the records for nodes. We discuss syntax and semantic analysis in Chapters 4 and 6, respectively.

Intermediate Code Generation

After syntax and semantic analysis, some compilers generate an explicit intermediate representation of the source program. We can think of this intermediate representation as a program for an abstract machine. This intermediate representation should have two important properties; it should be easy to produce, and easy to translate into the target program.

The intermediate representation can have a variety of forms. In Chapter 8,



```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

Fig. 1.10. Translation of a statement.

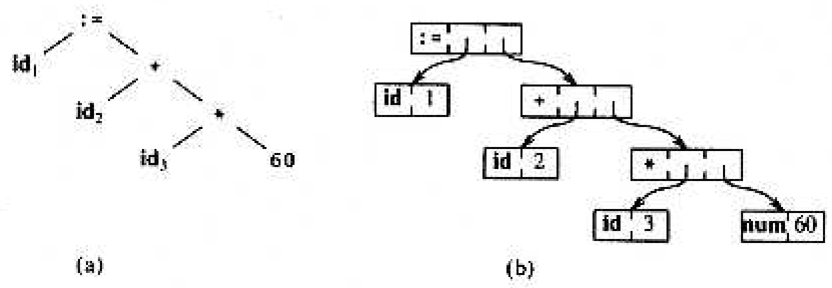


Fig. 1.11. The data structure in (b) is for the tree in (a).

we consider an intermediate form called "three-address code," which is like the assembly language for a machine in which every memory location can act like a register. Three-address code consists of a sequence of instructions, each of which has at most three operands. The source program in (1.1) might appear in three-address code as

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2                    (1.3)
id1 := temp3
```

This intermediate form has several properties. First, each three-address instruction has at most one operator in addition to the assignment. Thus, when generating these instructions, the compiler has to decide on the order in which operations are to be done; the multiplication precedes the addition in the source program of (1.1). Second, the compiler must generate a temporary name to hold the value computed by each instruction. Third, some "three-address" instructions have fewer than three operands, e.g., the first and last instructions in (1.3).

In Chapter 8, we cover the principal intermediate representations used in compilers. In general, these representations must do more than compute expressions; they must also handle flow-of-control constructs and procedure calls. Chapters 5 and 8 present algorithms for generating intermediate code for typical programming language constructs.

Code Optimization

The code optimization phase attempts to improve the intermediate code, so that faster-running machine code will result. Some optimizations are trivial. For example, a natural algorithm generates the intermediate code (1.3), using an instruction for each operator in the tree representation after semantic analysis, even though there is a better way to perform the same calculation, using the two instructions

```
    temp1 := id3 * 60.0
    id1 := id2 + temp1                                    (1.4)
```

There is nothing wrong with this simple algorithm, since the problem can be fixed during the code-optimization phase. That is, the compiler can deduce that the conversion of 60 from integer to real representation can be done once and for all at compile time, so the inttoreal operation can be eliminated. Besides, temp3 is used only once, to transmit its value to id1. It then becomes safe to substitute id1 for temp3, whereupon the last statement of (1.3) is not needed and the code of (1.4) results.

There is great variation in the amount of code optimization different compilers perform. In those that do the most, called "optimizing compilers," a significant fraction of the time of the compiler is spent on this phase. However, there are simple optimizations that significantly improve the running time of the target program without slowing down compilation too much. Many of these are discussed in Chapter 9, while Chapter 10 gives the technology used by the most powerful optimizing compilers.

Code Generation

The final phase of the compiler is the generation of target code, consisting normally of relocatable machine code or assembly code. Memory locations are selected for each of the variables used by the program. Then, intermediate instructions are each translated into a sequence of machine instructions that perform the same task. A crucial aspect is the assignment of variables to registers.

For example, using registers 1 and 2, the translation of the code of (1.4) might become

```
    MOVF id3, R2
    MULF #60.0, R2
    MOVF id2, R1                                          (1.5)
    ADDF R2, R1
    MOVF R1, id1
```

The first and second operands of each instruction specify a source and destination, respectively. The F in each instruction tells us that instructions deal with floating-point numbers. This code moves the contents of the address[1] id3 into register 2, then multiplies it with the real-constant 60.0. The # signifies that 60.0 is to be treated as a constant. The third instruction moves id2 into register 1 and adds to it the value previously computed in register 2. Finally, the value in register 1 is moved into the address of id1, so the code implements the assignment in Fig. 1.10. Chapter 9 covers code generation.

[1] We have side-stepped the important issue of storage allocation for the identifiers in the source program. As we shall see in Chapter 7, the organization of storage at run-time depends on the language being compiled. Storage-allocation decisions are made either during intermediate code generation or during code generation.

Compilers

Principles, Techniques, and Tools

ALFRED V. AHO

AT&T Bell Laboratories
Murray Hill, New Jersey

RAVI SETHI

AT&T Bell Laboratories
Murray Hill, New Jersey

JEFFREY D. ULLMAN

Stanford University
Stanford, California

ADDISON-WESLEY PUBLISHING COMPANY
Reading, Massachusetts • Menlo Park, California
Don Mills, Ontario • Wokingham, England • Amsterdam • Sydney
Singapore • Tokyo • Madrid • Bogotá • Santiago • San Juan