

INTRODUCTION TO COMPILING¹⁾

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 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 reference, 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 COMPILERS

Simply stated, a compiler is a program that reads a program written in one language – the *source* language – and translate it into an equivalent program in another language – the *target* language (see Figure 1). As an important part of this translation process, the compiler reports to its user the presence of errors in the source program.

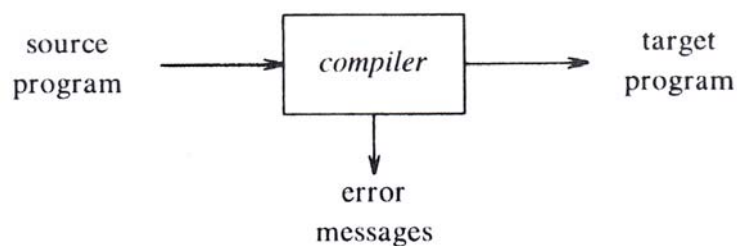


Figure 1. A Compiler

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

1) Selectively from "Compilers Principles, Techniques, and Tools," written by Alfred V.Aho, et al.

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.

1.1 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 technique. We shall consider analysis informally in Section 2 and outline the way target code is synthesized in a standard compiler in Section 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 Figure 2.

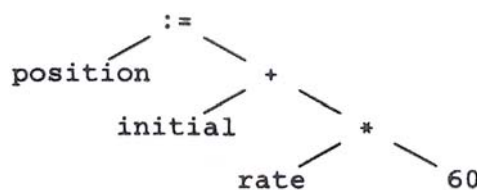


Figure 2. Syntax Tree for `position := initial + rate * 60`

1.2 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 3 shows a typical "compilation." The target program created by the compiler may require further processing before it can be run. The compiler in Figure 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 Figure 3 are discussed in Section 4.

2 ANALYSIS OF THE SOURCE PROGRAM

In this section, we introduce analysis and illustrate its use in some text formatting language. 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.
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.

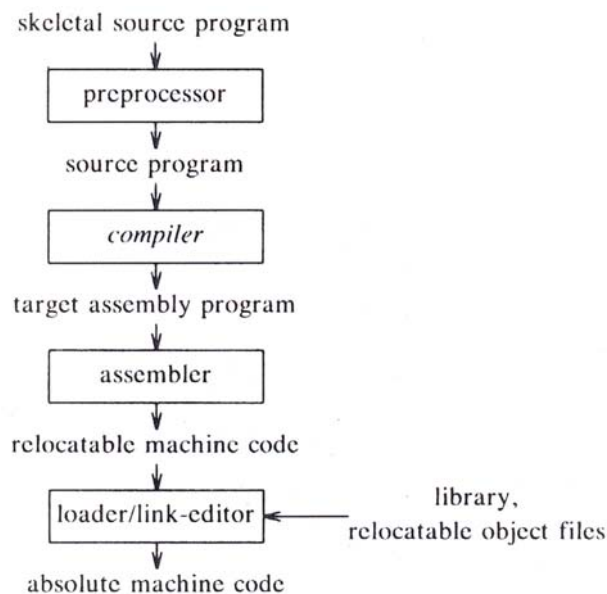


Figure 3. A Language-Processing System

2.1 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.

2.2 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 Figure 4.

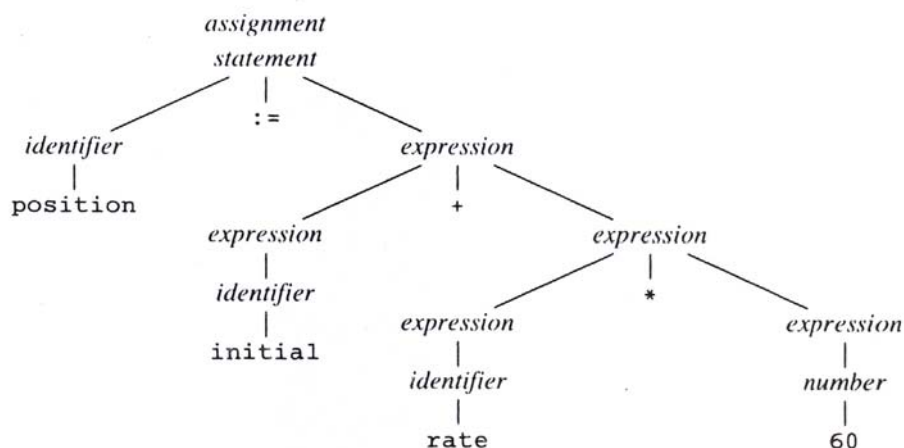


Figure 4. Parse Tree for `Position := initial + rate * 60`

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 Figure 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

$$\begin{aligned} & \textit{expression}_1 + \textit{expression}_2 \\ & \textit{expression}_1 * \textit{expression}_2 \\ & (\textit{expression}_1) \end{aligned}$$

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

$$\textit{identifier}_1 := \textit{expression}_2$$

is a statement.

2. If *expression*₁ is an expression and *statement*₂ is a statement, then

$$\mathbf{while} (\textit{expression}_1) \mathbf{do} \textit{statement}_2$$

$$\mathbf{if} (\textit{expression}_1) \mathbf{then} \textit{statement}_2$$

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.

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.

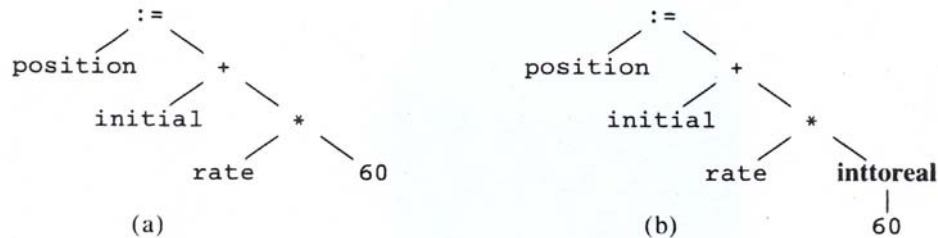


Figure 5. Semantic Analysis Inserts a Conversion from Integer to Real

The parse tree in Figure 4 describes the syntactic structure of the input. A more common internal representation of this syntactic structure is given by the syntax tree in Figure 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.

2.3 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.

Example 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 Figure 5 have been declared to be reals and that `60` by itself is assumed to be an integer. Type checking of Figure 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 Figure 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.

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 Figure 6. In practice, some of the phases may be grouped together, as mentioned in Section 5, and the intermediate representations between the grouped phases need not be explicitly constructed.

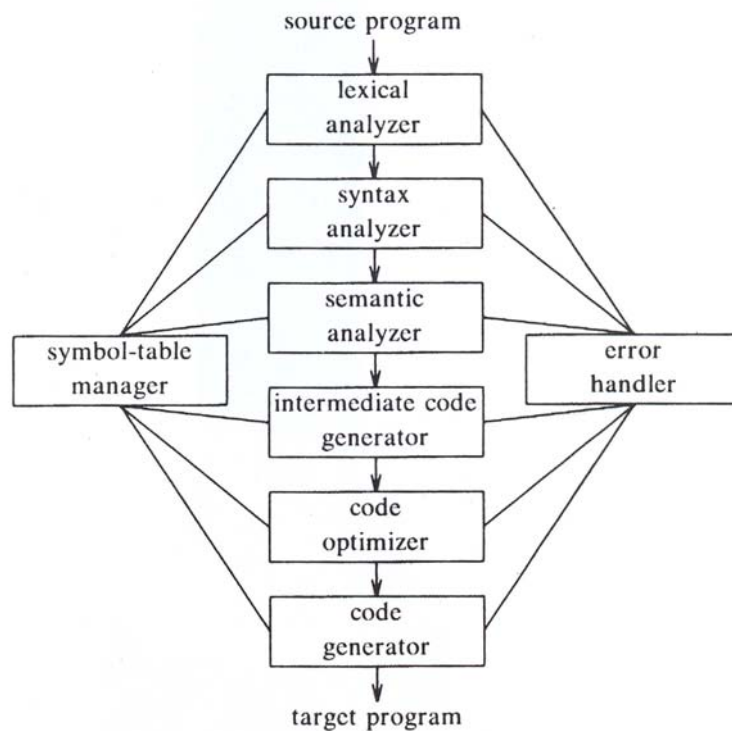


Figure 6. Phases of a Compiler.

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."

3.1 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.

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.

3.2 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.

3.3 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

$$\text{position} := \text{initial} + \text{rate} * 60 \quad (1)$$

Figure 7 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 **id₁**, **id₂**, and **id₃** 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) after lexical analysis is therefore suggested by:

$$\mathbf{id_1} := \mathbf{id_2} + \mathbf{id_3} * 60 \quad (2)$$

We should also make up tokens for the multi-character operator *:=* and the number 60 to reflect their internal representation.

The second and third phases, syntax and semantic analysis, have also been introduced in Section 2. Syntax analysis imposes a hierarchical structure on the token stream, which we shall portray by syntax trees as in Figure 8(a). A typical data structure for the tree is shown in Figure 8(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.

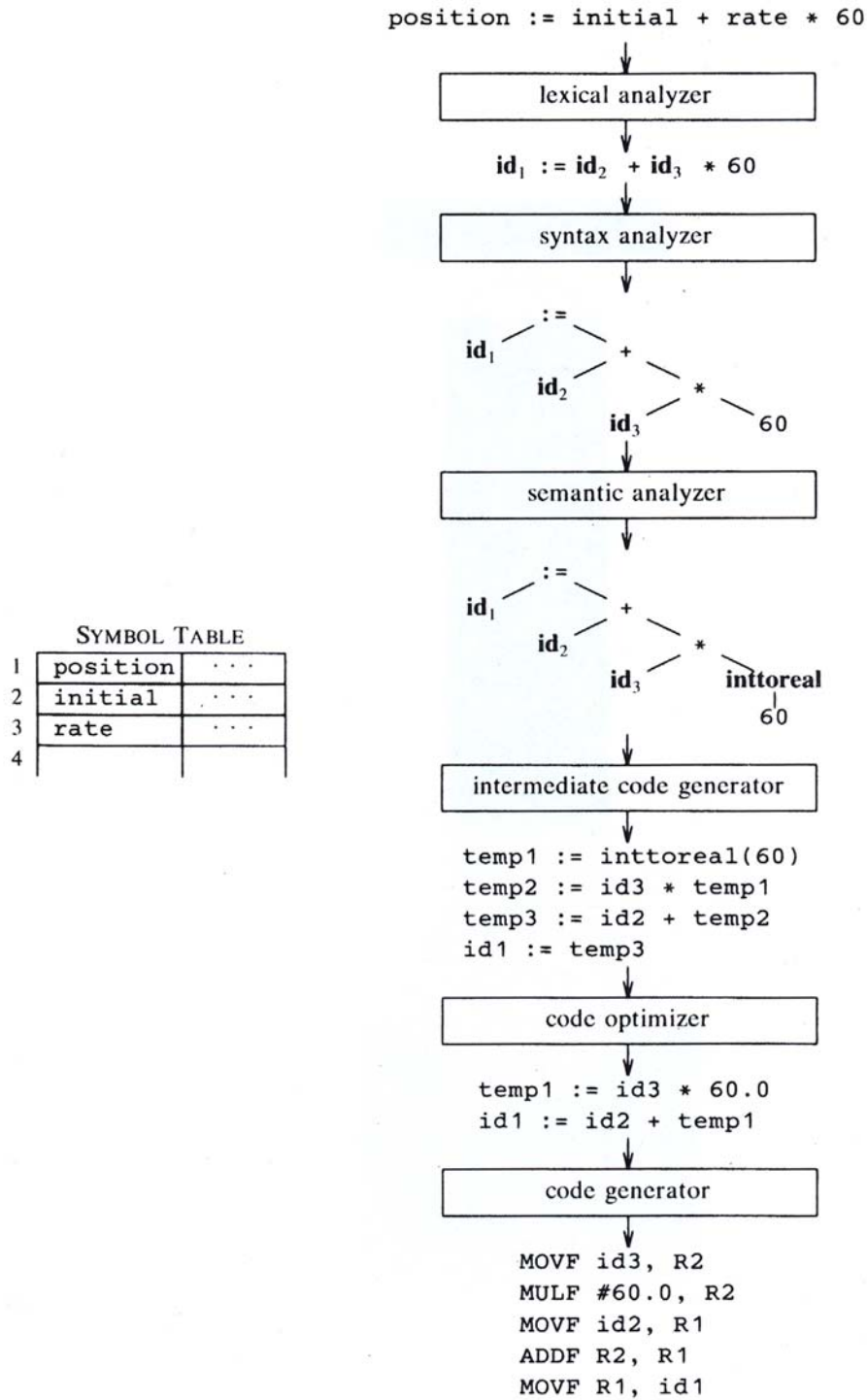


Figure 7. Translation of a Statement.

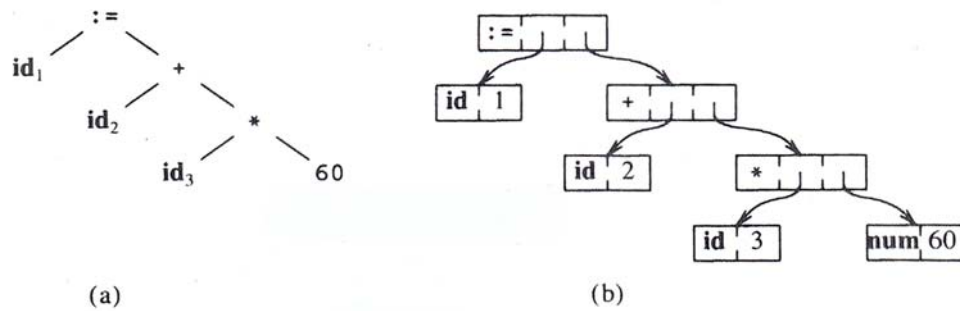


Figure 8. The Data Structure in (b) is for the Tree in (a).

3.4 Intermediate Code Generation

After syntax and semantic analysis, some compilers generate an explicit inter-mediate representation of the source program. We can think of this inter-mediate 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. 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) might appear in three-address code as

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

(3)

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). 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 (3).

In general, these representations must do more than compute expressions; they must also handle flow-of-control constructs and procedure calls.

3.5 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 (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
```

 (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 intto-real 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 (3) is not needed and the code of (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.

3.6 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 (4) might become

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

 (5)

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 address²⁾ `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 Figure 7.

4 COUSINS OF THE COMPILER

As we saw in Figure 3, the input to a compiler may be produced by one or more preprocessors, and further processing of the compiler's output may be needed before running machine code is obtained. In this section, we discuss the context in which a compiler typically operates.

4.1 Preprocessors

Preprocessors produce input to compilers. They may perform the following functions:

1. *Macro processing.* A preprocessor may allow a user to define macros that are shorthands for longer constructs.
2. *File inclusion.* A preprocessor may include header files into the program text. For example, the C preprocessor causes the contents of the file `<global.h>` to replace the statement `#include <global.h>` when it processes a file containing this statement.
3. *"Rational" preprocessors.* These processors augment older languages with more modern flow-of-control and data-structuring facilities. For example, such a preprocessor might provide the user with built-in macros for constructs like while-statements or if-statements, where none exist in the programming language itself.

Macro processors deal with two kinds of statement: macro definition and macro use. Definitions are normally indicated by some unique character or keyword, like `define` or `macro`. They consist of a name for the macro being defined and a *body*, forming its definition. Often, macro processors permit *formal parameters* in their definition, that is,

2) We have side-stepped the important issue of storage allocation for the identifiers in the source program. 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.

symbols to be replaced by values (a "value" is a string of characters, in this context). The use of a macro consists of naming the macro and supplying *actual parameters*, that is, values for its formal parameters. The macro processor substitutes the actual parameters for the formal parameters in the body of the macro; the transformed body then replaces the macro use itself.

4.2 Assemblers

Some compilers produce assembly code, as in (5), that is passed to an assembler for further processing. Other compilers perform the job of the assembler, producing relocatable machine code that can be passed directly to the loader/link-editor. We assume the reader has some familiarity with what an assembly language looks like and what an assembler does; here we shall review the relationship between assembly and machine code.

Assembly code is a mnemonic version of machine code, in which names are used instead of binary codes for operations, and names are also given to memory addresses. A typical sequence of assembly instructions might be

```
MOV a, R1
ADD #2, R1
MOV R1, b
```

(6)

This code moves the contents of the address *a* into register1, then adds the constant 2 to it, treating the contents of register 1 as a fixed-point number, and finally stores the result in the location named by *b*. Thus, it computes $b := a + 2$.

It is customary for assembly languages to have macro facilities that are similar to those in the macro preprocessors discussed above.

4.3 Two-Pass Assembly

The simplest form of assembler makes two passes over the input, where a pass consists of reading an input file once. In the first pass, all the identifiers that denote storage locations are found and stored in a symbol table (separate from that of the compiler). Identifiers are assigned storage locations as they are encountered for the first time, so after reading (6), for example, the symbol table might contain the entries shown in Figure 9. In that figure, we have assumed that a word, consisting of four bytes, is set aside for each identifier, and that addresses are assigned starting from byte 0.

Identifier	Address
------------	---------

a	0
b	4

Figure 9. An Assembler's Symbol Table with Identifiers of (6).

In the second pass, the assembler scans the input again. This time, it translates each operation code into the sequence of bits representing that operation in machine language, and it translates each identifier representing allocation into the address given for that identifier in the symbol table.

The output of the second pass is usually *relocatable* machine code, meaning that it can be loaded starting at any location L in memory; i.e., if L is added to all addresses in the code, then all references will be correct. Thus, the out-put of the assembler must distinguish those portions of instructions that refer to addresses that can be relocated.

Example 2. The following is a hypothetical machine code into which the assembly instructions (6) might be translated.

```

0001 01 00 00000000 *
0011 01 10 00000010
0010 01 00 00000100 *

```

(7)

We envision a tiny instruction word, in which the first four bits are the instruction code, with 0001, 0010, and 0011 standing for load, store, and add, respectively. By load and store we mean moves from memory into a register and vice versa. The next two bits designate a register, and 01 refers to register 1 in each of the three above instructions. The two bits after that represent a "tag," with 00 standing for the ordinary address mode, where the last eight bits refer to a memory address. The tag 10 stands for the "immediate" mode, where the last eight bits are taken literally as the operand. This mode appears in the second instruction of (7).

We also see in (7) a * associated with the first and third instructions. This * represents the *relocation bit* that is associated with each operand in relocatable machine code. Suppose that the address space containing the data is to be loaded starting at location L . The presence of the * means that L must be added to the address of the instruction. Thus, if $L = 00001111$, i.e., 15, then a and b would be at locations 15 and 19, respectively, and the instructions of (7) would appear as

```

0001 01 00 00001111
0011 01 10 00000010
0010 01 00 00010011

```

(8)

in *absolute*, or unrelocatable, machine code. Note that there is no * associated with the

second instruction in (7), so L has not been added to its address in (8), which is exactly right because the bits represents the constant 2, not the location 2.

4.3 Loaders and Link-Editors

Usually, a program called a *loader* performs the two functions of loading and link-editing. The process of loading consists of talking relocatable machine code, altering the relocatable addresses as discussed in Example 2, and placing the altered instructions and data in memory at the proper locations.

The link-editor allows us to make a single program from several files of relocatable machine code. These files may have been the result of several different compilations, and one or more may be library files of routines provided by the system and available to any program that needs them.

If the files are to be used together in a useful way, there may be some *external* references, in which the code of one file refers to a location in another file. This reference may be to a data location defined in one file and used in another, or it may be to the entry point of a procedure that appears in the code for one file and is called from another file. The relocatable machine code file must retain the information in the symbol table for each data location or instruction label that is referred to externally. If we do not know in advance what might be referred to, we in effect must include the entire assembler symbol table as part of the relocatable machine code.

For example, the code of (7) would be preceded by

```
a    0
b    4
```

If a file loaded with (7) referred to b, then that reference would be replaced by 4 plus the offset by which the data locations in file (7) were relocated.