Indeed, the basic organization of the sample compiler in this text was suggested by that course, and the machine simulator of Appendix C is a descendant of the one he provided.

More directly, I would like to thank my colleagues Bill Giles and Sam Khuri at San Jose State for encouraging me in this project, reading and commenting on most of the text, and for using preliminary drafts in their classes. I would also like to thank the students at San Jose State University in both my own and other classes who provided useful input. Further, I would like to thank Mary T. Stone of PWS for gathering a great deal of information on compiler tools and for coordinating the very useful review process.

The following reviewers contributed many excellent suggestions, for which I am grateful:

Jeff Jenness
Arkansas State University

Joe Lambert
Penn State University

Joan Lukas
University of Masschusetts, Boston

Jerry Potter
Kent State University

Samuel A. Rebelsky
Dartmouth College

Of course I alone am responsible for any shortcomings of the text. I have tried to make this book as error-free as possible. Undoubtedly errors remain, and I would be happy to hear from any readers willing to point them out to me.

Finally, I would like to thank my wife Margreth for her understanding, patience, and support, and our son Andrew for encouraging me to finish this book.

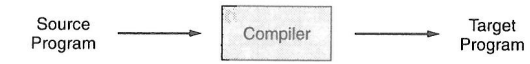
K.C.L.

Chapter 1

Introduction

1.1 Why Compilers? A Brief History
1.2 Programs Related to Compilers
1.3 The Translation Process
1.4 Major Data Structures in a Compiler
1.5 Other Issues in Compiler Structure
1.6 Bootstrapping and Porting
1.7 The TINY Sample Language and Compiler
1.8 C-Minus: A Language for a Compiler Project

Compilers are computer programs that translate one language to another. A compiler takes as its input a program written in its **source language** and produces an equivalent program written in its **target language**. Usually, the source language is a **high-level language**, such as C or C++, and the target language is **object code** (sometimes also called **machine code**) for the target machine, that is, code written in the machine instructions of the computer on which it is to be executed. We can view this process schematically as follows:



A compiler is a fairly complex program that can be anywhere from 10,000 to 1,000,000 lines of code. Writing such a program, or even understanding it, is not a simple task, and most computer scientists and professionals will never write a complete compiler. Nevertheless, compilers are used in almost all forms of computing, and anyone professionally involved with computers should know the basic organization and operation of a compiler. In addition, a frequent task in computer applications is the development of command interpreters and interface programs, which are smaller than compilers but which use the same techniques. A knowledge of these techniques is, therefore, of significant practical use.

It is the purpose of this text not only to provide such basic knowledge but also to give the reader all the necessary tools and practical experience to design and pro-

gram an actual compiler. To accomplish this, it is necessary to study the theoretical techniques, mainly from automata theory, that make compiler construction a manageable task. In covering this theory, we do not assume that the reader has previous knowledge of automata theory. Indeed, the viewpoint taken here is different from that in a standard automata theory text, in that it is aimed specifically at the compilation process. Nevertheless, a reader who has studied automata theory will find the theoretical material more familiar and will be able to proceed more quickly through those sections. In particular, Sections 2.2, 2.3, 2.4, and 3.2 may be skipped or skimmed by a reader with a good background in automata theory. In any case, the reader should be familiar with basic data structures and discrete mathematics. Some knowledge of machine architecture and assembly language is also essential, particularly for the chapter on code generation.

The study of the practical coding techniques themselves requires careful planning, since even with a good theoretical foundation the details of the code can be complex and overwhelming. This text contains a series of simple examples of programming language constructs that are used to elaborate the discussion of the techniques. The language we use for this discussion is called TINY. We also provide (in Appendix A) a more extensive example, consisting of a small but sufficiently complex subset of C, which we call C-Minus, which is suitable for a class project. In addition there are numerous exercises; these include simple paper-and-pencil exercises, extensions of code in the text, and more involved coding exercises.

In general, there is significant interaction between the structure of a compiler and the design of the programming language being compiled. In this text we will only incidentally study language design issues. Other texts are available that more fully treat programming language concepts and design issues. (See the Notes and References section at the end of this chapter.)

We begin with a brief look at the history and the raison d'être of compilers, together with a description of programs related to compilers. Then, we examine the structure of a compiler and the various translation processes and associated data structures and tour this structure using a simple concrete example. Finally, we give an overview of other issues of compiler structure, including bootstrapping and porting, concluding with a description of the principal language examples used in the remainder of the book.

1.1 WHY COMPILERS? A BRIEF HISTORY

With the advent of the stored-program computer pioneered by John von Neumann in the late 1940s, it became necessary to write sequences of codes, or programs, that would cause these computers to perform the desired computations. Initially, these programs were written in **machine language**—numeric codes that represented the actual machine operations to be performed. For example,

```
C7 06 0000 0002
```

represents the instruction to move the number 2 to the location 0000 (in hexadecimal) on the Intel 8x86 processors used in IBM PCs. Of course, writing such codes is extremely time consuming and tedious, and this form of coding was soon replaced by

assembly language, in which instructions and memory locations are given symbolic forms. For example, the assembly language instruction

```
MOV X , 2
```

is equivalent to the previous machine instruction (assuming the symbolic memory location X is 0000). An **assembler** translates the symbolic codes and memory locations of assembly language into the corresponding numeric codes of machine language.

Assembly language greatly improved the speed and accuracy with which programs could be written, and it is still in use today, especially when extreme speed or conciseness of code is needed. However, assembly language has a number of defects: it is still not easy to write and it is difficult to read and understand. Moreover, assembly language is extremely dependent on the particular machine for which it was written, so code written for one computer must be completely rewritten for another machine. Clearly, the next major step in programming technology was to write the operations of a program in a concise form more nearly resembling mathematical notation or natural language, in a way that was independent of any one particular machine and yet capable of itself being translated by a program into executable code. For example, the previous assembly language code can be written in a concise, machine-independent form as

```
X = 2
```

At first, it was feared that this might not be possible, or if it was, then the object code would be so inefficient as to be useless.

The development of the FORTRAN language and its compiler by a team at IBM led by John Backus between 1954 and 1957 showed that both these fears were unfounded. Nevertheless, the success of this project came about only with a great deal of effort, since most of the processes involved in translating programming languages were not well understood at the time.

At about the same time that the first compiler was under development, Noam Chomsky began his study of the structure of natural language. His findings eventually made the construction of compilers considerably easier and even capable of partial automation. Chomsky's study led to the classification of languages according to the complexity of their **grammars** (the rules specifying their structure) and the power of the algorithms needed to recognize them. The **Chomsky hierarchy**, as it is now called, consists of four levels of grammars, called the type 0, type 1, type 2, and type 3 grammars, each of which is a specialization of its predecessor. The type 2, or **context-free grammars** proved to be the most useful for programming languages, and today they are the standard way to represent the structure of programming languages. The study of the **parsing problem** (the determination of efficient algorithms for the recognition of context-free languages) was pursued in the 1960s and 1970s and led to a fairly complete solution of this problem, which today has become a standard part of compiler theory. Context-free languages and parsing algorithms are studied in Chapters 3, 4, and 5.

Closely related to context-free grammars are **finite automata** and **regular expressions**, which correspond to Chomsky's type 3 grammars. Begun at about the same time as Chomsky's work, their study led to symbolic methods for expressing the structure of the words, or tokens, of a programming language. Chapter 2 discusses finite automata and regular expressions.

Much more complex has been the development of methods for generating efficient object code, which began with the first compilers and continues to this day. These techniques are usually misnamed **optimization techniques**, but they really should be called **code improvement techniques**, since they almost never result in truly optimal object code but only improve its efficiency. Chapter 8 describes the basics of these techniques.

As the parsing problem became well understood, a great deal of work was devoted to developing programs that would automate this part of compiler development. These programs were originally called **compiler-compilers**, but are more aptly referred to as **parser generators**, since they automate only one part of the compilation process. The best-known of these programs is Yacc (yet another compiler-compiler) written by Steve Johnson in 1975 for the Unix system. Yacc is studied in Chapter 5. Similarly, the study of finite automata led to the development of another tool called a **scanner generator**, of which Lex (developed for the Unix system by Mike Lesk about the same time as Yacc) is the best known. Lex is studied in Chapter 2.

During the late 1970s and early 1980s, a number of projects focused on automating the generation of other parts of a compiler, including code generation. These attempts have been less successful, possibly because of the complex nature of the operations and our less than perfect understanding of them. We do not study them in detail in this text.

More recent advances in compiler design have included the following. First, compilers have included the application of more sophisticated algorithms for inferring and/or simplifying the information contained in a program, and these have gone hand in hand with the development of more sophisticated programming languages that allow this kind of analysis. Typical of these is the unification algorithm of Hindley-Milner type checking, used in the compilation of functional languages. Second, compilers have become more and more a part of a window-based **interactive development environment**, or IDE, that includes editors, linkers, debuggers, and project managers. So far there has been little standardization of such IDEs, but the development of standard windowing environments is leading in that direction. The study of such topics is beyond the scope of this text (but see the next section for a brief description of some of the components of an IDE). For pointers to the literature, see the Notes and References section at the end of the chapter. Despite the amount of research activity in recent years, however, the basics of compiler design have not changed much in the last 20 years, and they have increasingly become a part of the standard core of the computer science curriculum.

1.2 PROGRAMS RELATED TO COMPILERS

In this section, we briefly describe other programs that are related to or used together with compilers and that often come together with compilers in a complete language development environment. (We have already mentioned some of these.)

INTERPRETERS

An interpreter is a language translator like a compiler. It differs from a compiler in that it executes the source program immediately rather than generating object code that is executed after translation is complete. In principle, any programming language can be either interpreted or compiled, but an interpreter may be preferred to a compiler depending on the language in use and the situation under which translation occurs. For example, BASIC is a language that is more usually interpreted than

compiled. Similarly, functional languages such as LISP tend to be interpreted. Interpreters are also often used in educational and software development situations, where programs are likely to be translated and retranslated many times. On the other hand, a compiler is to be preferred if speed of execution is a primary consideration, since compiled object code is invariably faster than interpreted source code, sometimes by a factor of 10 or more. Interpreters, however, share many of their operations with compilers, and there can even be translators that are hybrids, lying somewhere between interpreters and compilers. We will discuss interpreters intermittently, but our main focus in this text will be on compilation.

ASSEMBLERS

An assembler is a translator for the assembly language of a particular computer. As we have already noted, assembly language is a symbolic form of the machine language of the computer and is particularly easy to translate. Sometimes, a compiler will generate assembly language as its target language and then rely on an assembler to finish the translation into object code.

LINKERS

Both compilers and assemblers often rely on a program called a linker, which collects code separately compiled or assembled in different object files into a file that is directly executable. In this sense, a distinction can be made between object code—machine code that has not yet been linked—and executable machine code. A linker also connects an object program to the code for standard library functions and to resources supplied by the operating system of the computer, such as memory allocators and input and output devices. It is interesting to note that linkers now perform the task that was originally one of the principal activities of a compiler (hence the use of the word *compile*—to construct by collecting from different sources). We will not study the linking process in this text, since it is extremely dependent on the details of the operating system and processor. We will also not always make a clear distinction between unlinked object code and executable code, since this distinction will not be important for our study of compilation techniques.

LOADERS

Often a compiler, assembler, or linker will produce code that is not yet completely fixed and ready to execute, but whose principal memory references are all made relative to an undetermined starting location that can be anywhere in memory. Such code is said to be **relocatable**, and a loader will resolve all relocatable addresses relative to a given base, or starting, address. The use of a loader makes executable code more flexible, but the loading process often occurs behind the scenes (as part of the operating environment) or in conjunction with linking. Rarely is a loader an actual separate program.

PREPROCESSORS

A preprocessor is a separate program that is called by the compiler before actual translation begins. Such a preprocessor can delete comments, include other files, and perform **macro** substitutions (a macro is a shorthand description of a repeated sequence of text). Preprocessors can be required by the language (as in C) or can be later add-ons that provide additional facilities (such as the Ratfor preprocessor for FORTRAN).

EDITORS

Compilers usually accept source programs written using any editor that will produce a standard file, such as an ASCII file. More recently, compilers have been bundled together with editors and other programs into an interactive development environment, or IDE. In such a case, an editor, while still producing standard files, may be oriented toward the format or structure of the programming language in question. Such editors are called **structure based** and already include some of the operations of a compiler, so that, for example, the programmer may be informed of errors as the program is written rather than when it is compiled. The compiler and its companion programs can also be called from within the editor, so that the programmer can execute the program without leaving the editor.

DEBUGGERS

A debugger is a program that can be used to determine execution errors in a compiled program. It is also often packaged with a compiler in an IDE. Running a program with a debugger differs from straight execution in that the debugger keeps track of most or all of the source code information, such as line numbers and names of variables and procedures. It can also halt execution at prespecified locations called **breakpoints** as well as provide information on what functions have been called and what the current values of variables are. To perform these functions, the debugger must be supplied with appropriate symbolic information by the compiler, and this can sometimes be difficult, especially in a compiler that tries to optimize the object code. Thus, debugging becomes a compiler question, which, however, is beyond the scope of this book.

PROFILERS

A profiler is a program that collects statistics on the behavior of an object program during execution. Typical statistics that may be of interest to the programmer are the number of times each procedure is called and the percentage of execution time spent in each procedure. Such statistics can be extremely useful in helping the programmer to improve the execution speed of the program. Sometimes the compiler will even use the output of the profiler to automatically improve the object code without intervention by the programmer.

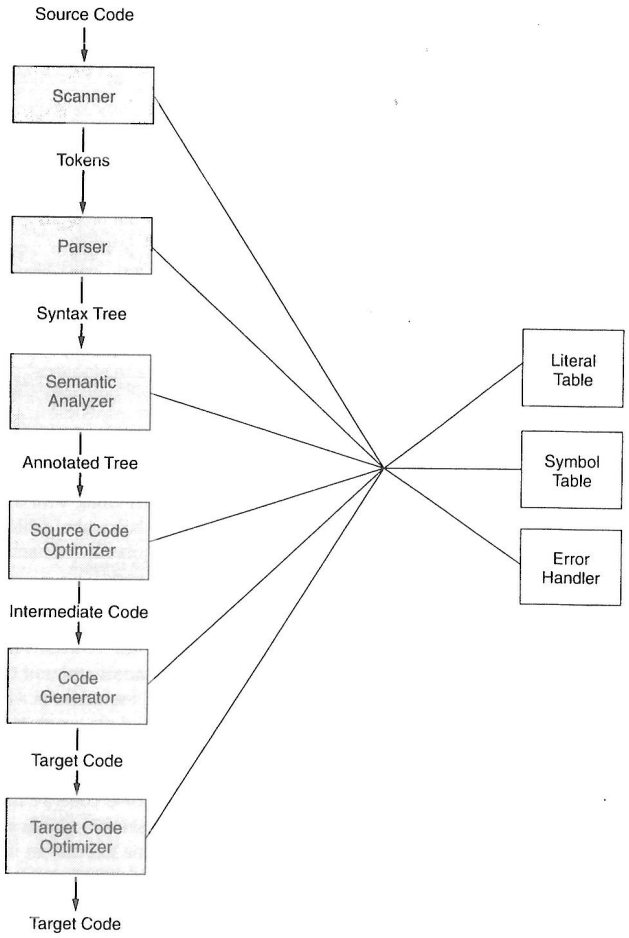
PROJECT MANAGERS

Modern software projects are usually so large that they are undertaken by groups of programmers rather than a single programmer. In such cases, it is important that the files being worked on by different people are coordinated, and this is the job of a project manager program. For example, a project manager should coordinate the merging of separate versions of the same file produced by different programmers. It should also maintain a history of changes to each of a group of files, so that coherent versions of a program under development can be maintained (this is something that can also be useful to the one-programmer project). A project manager can be written in a language-independent way, but when it is bundled together with a compiler, it can maintain information on the specific compiler and linker operations needed to build a complete executable program. Two popular project manager programs on Unix systems are **sccs (source code control system)** and **rcs (revision control system)**.

1.3 THE TRANSLATION PROCESS

A compiler consists internally of a number of steps, or **phases**, that perform distinct logical operations. It is helpful to think of these phases as separate pieces within the compiler, and they may indeed be written as separately coded operations although in practice they are often grouped together. The phases of a compiler are shown in Figure 1.1, together with three auxiliary components that interact with some or all of

Figure 1.1
The phases of a compiler



the phases: the literal table, the symbol table, and the error handler. We will briefly describe each phase here; they will be studied in greater detail in the following chapters. (The literal and symbol tables will be discussed in more detail in the next section and the error handler in Section 1.5.)

THE SCANNER

This phase of the compiler does the actual reading of the source program, which is usually in the form of a stream of characters. The scanner performs what is called **lexical analysis**: it collects sequences of characters into meaningful units called **tokens**, which are like the words of a natural language such as English. Thus, a scanner can be thought to perform a function similar to spelling.

As an example, consider the following line of code, which could be part of a C program:

```
a[index] = 4 + 2
```

This code contains 12 nonblank characters but only 8 tokens:

`a`	identifier
`[`	left bracket
`index`	identifier
`]`	right bracket
`=`	assignment
`4`	number
`+`	plus sign
`2`	number

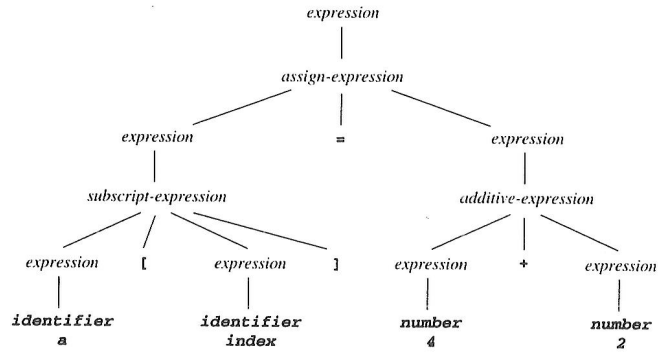
Each token consists of one or more characters that are collected into a unit before further processing takes place.

A scanner may perform other operations along with the recognition of tokens. For example, it may enter identifiers into the symbol table, and it may enter **literals** into the literal table (literals include numeric constants such as 3.1415926535 and quoted strings of text such as "Hello, world!").

THE PARSER

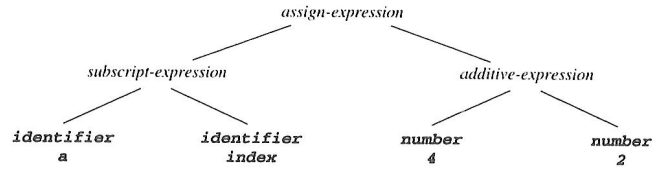
The parser receives the source code in the form of tokens from the scanner and performs **syntax analysis**, which determines the structure of the program. This is similar to performing grammatical analysis on a sentence in a natural language. Syntax analysis determines the structural elements of the program as well as their relationships. The results of syntax analysis are usually represented as a **parse tree** or a **syntax tree**.

As an example, consider again the line of C code we have already given. It represents a structural element called an expression, which is an assignment expression consisting of a subscripted expression on the left and an integer arithmetic expression on the right. This structure can be represented as a parse tree in the following form:



Note that the internal nodes of the parse tree are labeled by the names of the structures they represent and that the leaves of the parse tree represent the sequence of tokens from the input. (Names of structures are written in a different typeface to distinguish them from tokens.)

A parse tree is a useful aid to visualizing the syntax of a program or program element, but it is inefficient in its representation of that structure. Parsers tend to generate a syntax tree instead, which is a condensation of the information contained in the parse tree. (Sometimes syntax trees are called **abstract syntax trees**, since they represent a further abstraction from parse trees.) An abstract syntax tree for our example of a C assignment expression is as follows:



Note that in the syntax tree many of the nodes have disappeared (including token nodes). For example, if we know that an expression is a subscript operation, then it is no longer necessary to keep the brackets `[` and `]` that represent this operation in the original input.

THE SEMANTIC ANALYZER

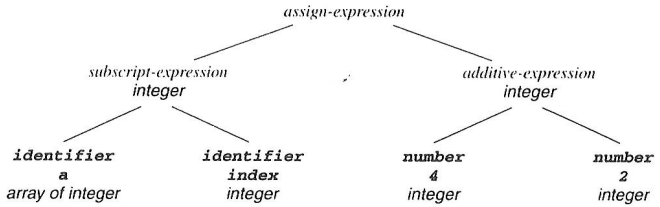
The semantics of a program are its "meaning," as opposed to its syntax, or structure. The semantics of a program determine its runtime behavior, but most programming languages have features that can be determined prior to execution and yet cannot be conveniently expressed as syntax and analyzed by the parser. Such features are referred to as **static semantics**, and the analysis of such semantics is

the task of the semantic analyzer. (The "dynamic" semantics of a program—those properties of a program that can only be determined by executing it—cannot be determined by a compiler, since it does not execute the program.) Typical static semantic features of common programming languages include declarations and type checking. The extra pieces of information (such as data types) computed by the semantic analyzer are called **attributes**, and these are often added to the tree as annotations, or "decorations." (Attributes may also be entered into the symbol table.)

In our running example of the C expression

```
a[index] = 4 + 2
```

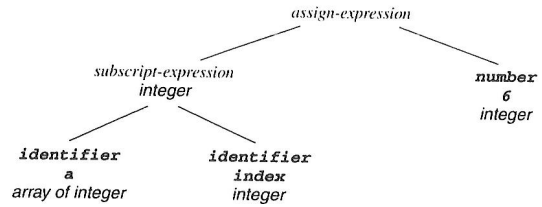
typical type information that would be gathered prior to the analysis of this line might be that `a` is an array of integer values with subscripts from a subrange of the integers and that `index` is an integer variable. Then the semantic analyzer would annotate the syntax tree with the types of all the subexpressions and then check that the assignment makes sense for these types, declaring a type mismatch error if not. In our example, all the types make sense, and the result of semantic analysis on the syntax tree could be represented by the following annotated tree:



THE SOURCE CODE OPTIMIZER

Compilers often include a number of code improvement, or optimization, steps. The earliest point at which most optimization steps can be performed is just after semantic analysis, and there may be possibilities for code improvement that depend only on the source code. We indicate this possibility by providing this operation as a separate phase in the compilation process. Individual compilers exhibit a wide variation not only in the kinds of optimizations performed but also in the placement of the optimization phases.

In our example, we have included an opportunity for source-level optimization; namely, the expression `4 + 2` can be precomputed by the compiler to the result `6`. (This particular optimization is known as **constant folding**.) Of course, much more complex possibilities exist (some of these are mentioned in Chapter 8). In our example, this optimization can be performed directly on the (annotated) syntax tree by collapsing the right-hand subtree of the root node to its constant value:



Many optimizations can be performed directly on the tree, but in a number of cases, it is easier to optimize a linearized form of the tree that is closer to assembly code. Many different varieties of such code exist, but a standard choice is **three-address code**, so called because it contains the addresses of (up to) three locations in memory. Another popular choice is **P-code**, which has been used in many Pascal compilers.

In our example, three-address code for the original C expression might look like this:

```
t = 4 + 2
a[index] = t
```

(Note the use of an extra temporary variable `t` to store the intermediate result of the addition.) Now the optimizer would improve this code in two steps, first computing the result of the addition

```
t = 6
a[index] = t
```

and then replacing `t` by its value to get the three-address statement

```
a[index] = 6
```

In Figure 1.1 we have indicated the possibility that the source code optimizer may use three-address code by referring to its output as **intermediate code**. Intermediate code historically referred to a form of code representation intermediate between source code and object code, such as three-address code or a similar linear representation. However, it can also more generally refer to *any* internal representation for the source code used by the compiler. In this sense, the syntax tree can also be referred to as intermediate code, and indeed the source code optimizer could continue to use this representation in its output. Sometimes this more general sense is indicated by referring to the intermediate code as the **intermediate representation**, or **IR**.

THE CODE GENERATOR

The code generator takes the intermediate code or IR and generates code for the target machine. In this text we will write target code in assembly language form for ease of understanding, although most compilers generate object code directly. It is

in this phase of compilation that the properties of the target machine become the major factor. Not only is it necessary to use instructions as they exist on the target machine but decisions about the representation of data will now also play a major role, such as how many bytes or words variables of integer and floating-point data types occupy in memory.

In our example, we must now decide how integers are to be stored to generate code for the array indexing. For example, a possible sample code sequence for the given expression might be (in a hypothetical assembly language)

```
MOV   R0, index   ;; value of index -> R0
MUL   R0, 2       ;; double value in R0
MOV   R1, &a      ;; address of a -> R1
ADD   R1, R0      ;; add R0 to R1
MOV   *R1, 6      ;; constant 6 -> address in R1
```

In this code we have used a C-like convention for addressing modes, so that `&a` is the address of `a` (i.e., the base address of the array) and that `*R1` means indirect register addressing (so that the last instruction stores the value 6 to the address contained in R1). In this code we have also assumed that the machine performs byte addressing and that integers occupy two bytes of memory (hence the use of 2 as the multiplication factor in the second instruction).

THE TARGET CODE OPTIMIZER

In this phase, the compiler attempts to improve the target code generated by the code generator. Such improvements include choosing addressing modes to improve performance, replacing slow instructions by faster ones, and eliminating redundant or unnecessary operations.

In the sample target code given, there are a number of improvements possible. One is to use a shift instruction to replace the multiplication in the second instruction (which is usually expensive in terms of execution time). Another is to use a more powerful addressing mode, such as indexed addressing to perform the array store. With these two optimizations, our target code becomes

```
MOV R0, index     ;; value of index -> R0
SHL R0            ;; double value in R0
MOV &a[R0], 6     ;; constant 6 -> address a + R0
```

This completes our brief description of the phases of a compiler. We want to emphasize that this description is only schematic and does not necessarily represent the actual organization of a working compiler. Indeed, compilers exhibit a wide variation in their organizational details. Nevertheless, the phases we have described are present in some form in nearly all compilers.

We have also discussed only tangentially the data structures required to maintain the information needed by each phase, such as the syntax tree, the intermediate code (assuming these are not the same), the literal table, and the symbol table. We devote the next section to a brief overview of the major data structures in a compiler.

1.4 MAJOR DATA STRUCTURES IN A COMPILER

The interaction between the algorithms used by the phases of a compiler and the data structures that support these phases is, of course, a strong one. The compiler writer strives to implement these algorithms in as efficient a manner as possible, without incurring too much extra complexity. Ideally, a compiler should be capable of compiling a program in time proportional to the size of the program, that is, in $O(n)$ time, where n is a measure of program size (usually, the number of characters). In this section, we indicate a few of the principal data structures that are needed by the phases as part of their operation and that serve to communicate information among the phases.

TOKENS

When a scanner collects characters into a token, it generally represents the token symbolically, that is, as a value of an enumerated data type representing the set of tokens of the source language. Sometimes it is also necessary to preserve the string of characters itself or other information derived from it, such as the name associated with an identifier token or the value of a number token. In most languages the scanner needs only to generate one token at a time (this is called **single symbol lookahead**). In this case, a single global variable can be used to hold the token information. In other cases (most notably FORTRAN), an array of tokens may be needed.

THE SYNTAX TREE

If the parser does generate a syntax tree, it is usually constructed as a standard pointer-based structure that is dynamically allocated as parsing proceeds. The entire tree can then be kept as a single variable pointing to the root node. Each node in the structure is a record whose fields represent the information collected both by the parser and, later, by the semantic analyzer. For example, the data type of an expression may be kept as a field in the syntax tree node for the expression. Sometimes, to save space, these fields are also dynamically allocated, or they are stored in other data structures, such as the symbol table, that allow selective allocation and deallocation. Indeed, each syntax tree node itself may require different attributes to be stored, depending on the kind of language structure it represents (for example, an expression node has different requirements from a statement node or a declaration node). In this case, each node in the syntax tree may be represented by a variant record, with each node kind containing only the information necessary for that case.

THE SYMBOL TABLE

This data structure keeps information associated with identifiers: functions, variables, constants, and data types. The symbol table interacts with almost every phase of the compiler: the scanner, parser, or semantic analyzer may enter identifiers into the table; the semantic analyzer will add data type and other information; and the optimization and code generation phases will use the information provided by the symbol table to make appropriate object code choices. Since the symbol table will be accessed so frequently, insertion, deletion, and access operations need to be efficient, preferably constant-time operations. A standard data structure for this purpose is the hash table, although various tree structures can also be used. Sometimes several tables are used and maintained in a list or stack.

THE LITERAL TABLE

Quick insertion and lookup are essential as well to the literal table, which stores constants and strings used in a program. However, a literal table need not allow deletions, since its data applies globally to the program and a constant or string will appear only once in this table. The literal table is important in reducing the size of a program in memory by allowing the reuse of constants and strings. It is also needed by the code generator to construct symbolic addresses for literals and for entering data definitions in the target code file.

INTERMEDIATE CODE

Depending on the kind of intermediate code (e.g., three-address code and P-code) and the kinds of optimizations performed, this code may be kept as an array of text strings, a temporary text file, or as a linked list of structures. In compilers that perform complex optimizations, particular attention must be given to choosing representations that permit easy reorganization.

TEMPORARY FILES

Historically, computers did not possess enough memory for an entire program to be kept in memory during compilation. This problem was solved by using temporary files to hold the products of intermediate steps during translation or by compiling "on the fly," that is, keeping only enough information from earlier parts of the source program to enable translation to proceed. Memory constraints are now a much smaller problem, and it is possible to require that an entire compilation unit be maintained in memory, especially if separate compilation is available in the language. Still, compilers occasionally find it useful to generate intermediate files during some of the processing steps. Typical among these is the need to **backpatch** addresses during code generation. For example, when translating a conditional statement such as

```
if x = 0 then ... else ...
```

a jump from the test to the else-part must be generated before the location of the code for the else-part is known:

```
CMP X,0
JNE NEXT ;; location of NEXT not yet known
<code for then-part>
NEXT:
<code for else-part>
```

Typically, a blank must be left for the value of `NEXT`, which is filled in once that value becomes known. This is easily accomplished with the use of a temporary file.

1.5 OTHER ISSUES IN COMPILER STRUCTURE

The structure of a compiler may be viewed from many different angles. In Section 1.3 we described its phases, which represent the logical structure of a compiler. Other viewpoints are possible: the physical structure of the compiler, the sequencing of the operations, and so on. The compiler writer should be familiar with as many views of compiler structure as possible, since the structure of the compiler will have a major impact on its reliability, efficiency, usefulness, and maintainability. In this section we will consider other aspects of compiler structure and indicate how each view applies.

ANALYSIS AND SYNTHESIS

In this view, compiler operations that analyze the source program to compute its properties are classified as the **analysis** part of the compiler, while operations involved in producing translated code are called the **synthesis** part of the compiler. Naturally, lexical analysis, syntax analysis, and semantic analysis belong to the analysis part, while code generation is synthesis. Optimization steps may involve both analysis and synthesis. Analysis tends to be more mathematical and better understood, while synthesis requires more specialized techniques. Thus, it is helpful to separate analysis steps from synthesis steps so each can be changed independently of the other.

FRONT END AND BACK END

This view regards the compiler as separated into those operations that depend only on the source language (the **front end**) and those operations that depend only on the target language (the **back end**). This is similar to the division into analysis and synthesis: the scanner, parser, and semantic analyzer are part of the front end, while the code generator is part of the back end. However, some optimization analysis can be target dependent, and therefore part of the back end, while intermediate code synthesis is often target independent and thus part of the front end. Ideally, the compiler would be strictly divided into these two sections, with the intermediate representation as the medium of communication between them:

This structure is especially important for compiler **portability**, in which the compiler is designed with a view toward changing either the source code (which involves rewriting the front end) or the target code (which involves rewriting the back end). In practice this has proven difficult to achieve, and so-called portable compilers tend still to have features that depend on both source and target languages. In part, this can be blamed on rapid and fundamental changes in both programming languages and machine architectures, but it is also difficult to efficiently retain all the information one might need in moving to a new target language or in making the data structures suitably general to permit a change to a new source language. Nevertheless, a consistent attempt to separate front and back ends will pay off in easier portability.

PASSES

A compiler often finds it convenient to process the entire source program several times before generating code. These repetitions are referred to as **passes**. After the initial pass, which constructs a syntax tree or intermediate code from the source, a