pass consists of processing the intermediate representation, adding information to it, altering its structure, or producing a different representation. Passes may or may not correspond to phases—often a pass will consist of several phases. Indeed, depending on the language, a compiler may be **one pass**, in that all phases occur during a single pass. This results in efficient compilation but also in (typically) less efficient target code. Pascal and C are both languages that permit one-pass compilation. (Modula-2 is a language whose structure requires that a compiler have at least two passes.) Most compilers with optimizations use more than one pass; a typical arrangement is one pass for scanning and parsing, one pass for semantic analysis and source-level optimization, and a third pass for code generation and target-level optimization. Heavily optimizing compilers may use even more passes: five, six, or even eight passes are not unknown.

LANGUAGE DEFINITION AND COMPILERS

We noted in Section 1.1 that the lexical and syntactic structures of a programming language are usually specified in formal terms and use regular expressions and context-free grammars. The semantics of a programming language, however, are still commonly specified using English (or other natural language) descriptions. These descriptions (together with the formal lexical and syntactic structure) are usually collected into a **language reference manual**, or **language definition**. With a new language, a language definition and a compiler are often developed simultaneously, since the techniques available to the compiler writer can have a major impact on the definition of the language. Similarly, the way in which a language is defined will have a major impact on the techniques that are needed to construct the compiler.

A more common situation for the compiler writer is that the language being implemented is well known and has an existing language definition. Sometimes this language definition has been raised to the level of a **language standard** that has been approved by one of the official standardization organizations, such as ANSI (American National Standards Institute) or ISO (International Organization for Standardization). For example, FORTRAN, Pascal, and C have ANSI standards. Ada has a standard approved by the U.S. government. In this case, the compiler writer must interpret the language definition and implement a compiler that conforms to the language definition. This is often not an easy task, but it is sometimes made easier by the existence of a set of standard test programs (a **test suite**) against which a compiler can be tested (such a test suite exists for Ada). The TINY example language used in the text has its lexical, syntactic, and semantic structure specified in Sections 2.5, 3.7, and 6.5, respectively. Appendix A contains a minimal language reference manual for the C-Minus compiler project language.

Occasionally, a language will have its semantics given by a **formal definition** in mathematical terms. Several methods that are currently used do this, and no one method has achieved the level of a standard, although so-called **denotational semantics** has become one of the more common methods, especially in the functional programming community. When a formal definition exists for a language, then it is (in theory) possible to give a mathematical proof that a compiler conforms to the definition. However, this is such a difficult undertaking that it is almost never done. In any case, the techniques for doing so are beyond the scope of this text, and formal semantic techniques will not be studied here.

One aspect of compiler construction that is particularly affected by the language definition is the structure and behavior of the runtime environment. Runtime environments are studied in detail in Chapter 7. It is worth noting here, however, that the structure of data allowed in a programming language, and particularly the kinds of function calls and returned values allowed, have a decisive effect on the complexity of the runtime system. In particular, the three basic types of runtime environments, in increasing order of complexity, are as follows:

First, FORTRAN77, with no pointers or dynamic allocation and no recursive function calls, allows a completely static runtime environment, where all memory allocation is done prior to execution. This makes the job of allocation particularly easy for the compiler writer, as no code needs to be generated to maintain the environment. Second, Pascal, C, and other so-called Algol-like languages allow a limited form of dynamic allocation and recursive function calls and require a "semi-dynamic" or stack-based runtime environment with an additional dynamic structure called a *heap* from which the programmer can schedule dynamic allocation. Finally, functional and most object-oriented languages, such as LISP and Smalltalk, require a "fully dynamic" environment in which all allocation is performed automatically by code generated by the compiler. This is complicated, because it requires that memory also be freed automatically, and this in turn requires complex "garbage collection" algorithms. We will survey such methods along with our study of runtime environments, but a complete account of this area is beyond the scope of this book.

COMPILER OPTIONS AND INTERFACES

An important aspect of compiler construction is the inclusion of mechanisms for interfacing with the operating system and for providing options to the user for various purposes. Examples of interface mechanisms are the provision of input and output facilities as well as access to the file system of the target machine. Examples of user options include the specification of listing characteristics (length, error messages, cross-reference tables) and code optimization options (performance of certain optimizations but not others). Both interfacing and options are collectively referred to as compiler **pragmatics**. Sometimes a language definition will specify that certain pragmatics must be provided. For example, Pascal and C both specify certain input/output procedures (in Pascal they are part of the language proper, whereas in C they are part of the specification of a standard library). In Ada, a number of compiler directives, called **pragmas**, are part of the language definition. For example, the Ada statements

```
pragma LIST(ON);
...
pragma LIST(OFF);
```

generate a compiler listing for the part of the program contained within the pragmas. In this text we will see compiler directives only in the context of generating listing information for compiler debugging purposes. Also, we will not treat issues in input/output and operating system interfacing, since these involve considerable detail and vary so much from one operating system to another.

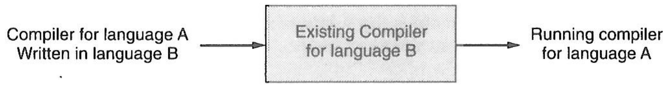
ERROR HANDLING

One of the most important functions of a compiler is its response to errors in a source program. Errors can be detected during almost every phase of compilation. These **static** (or **compile-time**) **errors** must be reported by a compiler, and it is important that a compiler be able to generate meaningful error messages and resume compilation after each error. Each phase of a compiler will need a slightly different kind of error handling, and so an **error handler** must contain different operations, each appropriate for a specific phase and situation. Error handling techniques for each phase will therefore be studied separately in the appropriate chapter.

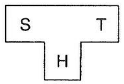
A language definition will usually require not only that static errors be caught by a compiler but also that certain execution errors be caught as well. This requires a compiler to generate extra code that will perform suitable runtime tests to guarantee that all such errors will cause an appropriate event during execution. The simplest such event will halt the execution of the program. Often, however, this is inadequate, and a language definition may require the presence of **exception handling** mechanisms. These can substantially complicate the management of a runtime system, especially if a program may continue to execute from the point where the error occurred. We will not consider the implementation of such a mechanism, but we will show how a compiler can generate test code to ensure that specified runtime errors will cause execution to halt.

1.6 BOOTSTRAPPING AND PORTING

We have discussed the source language and target language as determining factors in the structure of a compiler and the usefulness of separating source and target language issues into front and back ends. But we have not mentioned the third language involved in the compiler construction process: the language in which the compiler itself is written. For the compiler to execute immediately, this implementation (or **host**) language would have to be machine language. This was indeed how the first compilers were written, since essentially no compilers existed yet. A more reasonable approach today is to write the compiler in another language for which a compiler already exists. If the existing compiler already runs on the target machine, then we need only compile the new compiler using the existing compiler to get a running program:

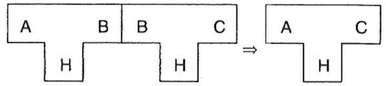


If the existing compiler for the language B runs on a machine different from the target machine, then the situation is a bit more complicated. Compilation then produces a **cross compiler**, that is, a compiler that generates target code for a different machine from the one on which it runs. This and other more complex situations are best described by drawing a compiler as a **T-diagram** (named after its shape). A compiler written in language H (for host language) that translates language S (for source language) into language T (for target language) is drawn as the following T-diagram:

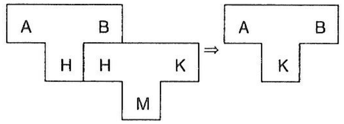


Note that this is equivalent to saying that the compiler runs on "machine" H (if H is not machine code, then we consider it to be the executable code for a hypothetical machine). Typically, we expect H to be the same as T (that is, the compiler produces code for the same machine as the one on which it runs), but this needn't be the case.

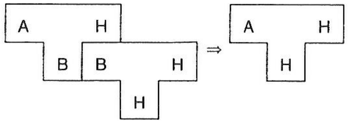
T-diagrams can be combined in two ways. First, if we have two compilers that run on the same machine H, one of which translates language A to language B and the other of which translates language B to language C, then we can combine them by letting the output of the first be the input to the second. The result is a compiler from A to C on machine H. We express this as follows:



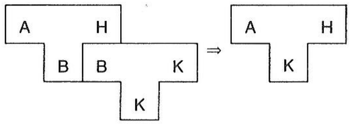
Second, we can use a compiler from "machine" H to "machine" K to translate the implementation language of another compiler from H to K. We express this as follows:



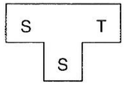
Now the first scenario we described previously—that is, using an existing compiler for language B on machine H to translate a compiler from language A to H written in B—can be viewed as the following diagram, which is just a special case of the previous diagram:



The second scenario we have described—where the compiler of language B generates code for (and runs on) a different machine, which results in a cross compiler for A—can similarly be described as follows:



It is common to write a compiler in the same language that it is to compile:



While this appears to be a blunder of circularity—since, if no compiler for the source language yet exists, the compiler itself cannot be compiled—there are important advantages to be gained from this approach.

Consider, for example, how we might approach the circularity problem. We might write a "quick and dirty" compiler in assembly language, translating only those features of the language that are actually used in the compiler (having, of course, limited our use of those features when writing the "good" compiler). This "quick and dirty" compiler may also produce extremely inefficient code (it only needs to be correct!). Once we have the running "quick and dirty" compiler, we use it to compile the "good" compiler. Then we recompile the "good" compiler to produce the final version. This process is called **bootstrapping**. This process is illustrated in Figures 1.2(a) and 1.2(b).

After bootstrapping, we have a compiler in both source code and executing code. The advantage to this is that any improvement to the source code of the compiler can be immediately bootstrapped to a working compiler by applying the same two-step process as before.

Figure 1.2(a)
The first step in a bootstrap process

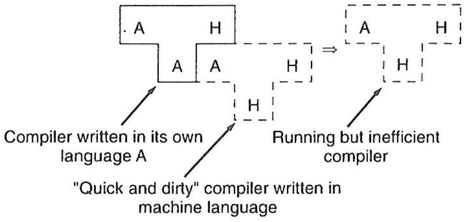
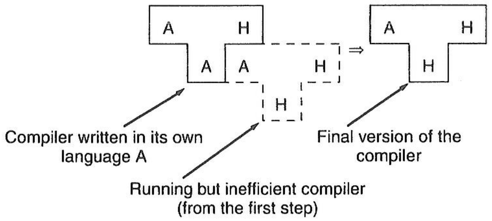

A
H
A
A
H
H
⇒
A
H
H
Compiler written in its own language A
Running but inefficient compiler
"Quick and dirty" compiler written in machine language

Figure 1.2(b)
The second step in a bootstrap process


A
H
A
A
H
H
⇒
A
H
H
Compiler written in its own language A
Final version of the compiler
Running but inefficient compiler (from the first step)

But there is another advantage. Porting the compiler to a new host computer now only requires that the back end of the source code be rewritten to generate code for the new machine. This is then compiled using the old compiler to produce a cross compiler, and the compiler is again recompiled by the cross compiler to produce a working version for the new machine. This is illustrated in Figures 1.3(a) and 1.3(b).

Figure 1.3(a)
Porting a compiler written in its own source language (step 1)

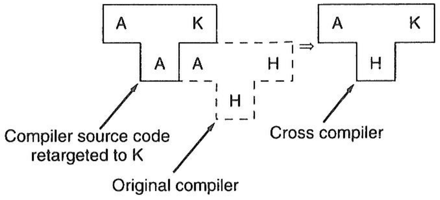
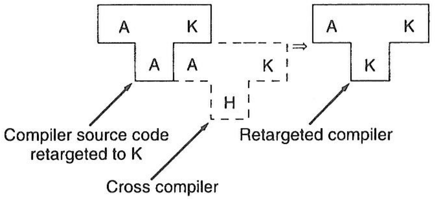


Figure 1.3(b)
Porting a compiler written in its own source language (step 2)



1.7 THE TINY SAMPLE LANGUAGE AND COMPILER

A book on compiler construction would be incomplete without examples for each step in the compilation process. In many cases we will illustrate techniques with examples that are abstracted from existing languages, such as C, C++, Pascal, and Ada. These examples, however, are not enough to show how all the parts of a compiler fit together. For that, it is also necessary to exhibit a complete compiler and provide a commentary on its operation.

This requirement—that an actual compiler be demonstrated—is a difficult one. A "real" compiler—that is, one that we would expect to use in everyday programming—has far too much detail and would be overwhelming to study within the framework of a text. On the other hand, a compiler for a very small language, whose listing would comprise 10 or so pages of text, could not hope to demonstrate adequately all the features that a "real" compiler needs.

We will attempt to satisfy these requirements by giving complete source code in (ANSI) C for a small language whose compiler can be easily comprehended once the techniques are understood. We will call this language TINY and will use it as a running example for the techniques studied in each chapter. The code for its compiler will be discussed as the techniques are covered. In this section we will give an overview of the language and its compiler. The complete compiler code is collected in Appendix B.

A further problem is the choice of the machine language to use as the target language of the TINY compiler. Again, the complexity of using actual machine code for an exisiting processor makes such a choice difficult. But the choice of a specific processor also has the effect of limiting the execution of the resulting target code to these machines. Instead, we simplify the target code to be the assembly language for a simple hypothetical processor, which we will call the TM machine (for tiny machine). We will take a quick look at this machine here but will delay a more extensive description until Chapter 8 (code generation). A TM simulator listing in C appears in Appendix C.

1.7.1 The TINY Language

A program in TINY has a very simple structure: it is just a sequence of statements separated by semicolons in a syntax similar to that of Ada or Pascal. There are no procedures and no declarations. All variables are integer variables, and variables are declared simply by assigning values to them (somewhat like FORTRAN or BASIC). There are only two control statements: an if-statement and a repeat-statement. Both control statements may themselves contain statement sequences. An if-statement has an optional else part and must be terminated by the keyword **end**. There are also read and write statements that perform input/output. Comments are allowed within curly brackets; comments cannot be nested.

Expressions in TINY are also limited to Boolean and integer arithmetic expressions. A Boolean expression consists of a comparison of two arithmetic expressions using either of the two comparison operators **<** and **=**. An arithmetic expression may involve integer constants, variables, parentheses, and any of the four integer operators **+**, **-**, *****, and **/** (integer division), with the usual mathematical properties. Boolean expressions may appear only as tests in control statements—there are no Boolean variables, assignment, or I/O.

Figure 1.4 gives a sample program in this language for the well-known factorial function. We will use this program as a running example throughout the text.

Figure 1.4
A TINY language program that outputs the factorial of its input

```
{ Sample program
  in TINY language -
  computes factorial
}
read x; { input an integer }
if x > 0 then { don't compute if x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  until x = 0;
  write fact { output factorial of x }
end
```

While TINY lacks many features necessary for real programming languages—procedures, arrays, and floating-point values are some of the more serious omissions—it is still large enough to exemplify most of the essential features of a compiler.

1.7.2 The TINY Compiler

The TINY compiler consists of the following C files, where we list header files (for inclusion) on the left and code files on the right:

globals.h	main.c
util.h	util.c
scan.h	scan.c
parse.h	parse.c
symtab.h	symtab.c
analyze.h	analyze.c
code.h	code.c
cgen.h	cgen.c

The source code for these files is listed in Appendix B, with line numbers, and in the order given, except that **main.c** is listed before **globals.h**. The **globals.h** header file is included in all the code files. It contains the definitions of data types and global variables used throughout the compiler. The file **main.c** contains the main program that drives the compiler, and it allocates and initializes the global variables. The remaining files consist of header/code file pairs, with the externally available function prototypes given in the header file and implemented (with possibly additional static local functions) in the associated code file. The **scan**, **parse**, **analyze**, and **cgen** files correspond exactly to the scanner, parser, semantic analyzer, and code generator phases of Figure 1.1. The **util** files contain utility functions needed to generate the internal representation of the source code (the syntax tree) and display listing and error information. The **symtab** files contain a hash-table implementation of a symbol table suitable for use with TINY. The **code** files contain utilities for code generation that are dependent on the target machine (the TM machine, described in Section 1.7.3). The remaining components of Figure 1.1 are absent: there is no separate error handler or literal table and there are no optimization phases. Also, there is no intermediate code separate from the syntax tree. Further, the symbol table interacts only with the semantic analyzer and the code generator (so we delay a discussion of it until Chapter 6).

To reduce the interaction among these files, we have also made the compiler four-pass: the first pass consists of the scanner and parser, which construct the syntax tree; the second and third passes undertake semantic analysis, with the second pass constructing the symbol table and the third pass performing type checking; the final pass is the code generator. The code in **main.c** that drives these passes is particularly simple. Ignoring flags and conditional compilation, the central code is as follows (see lines 69, 77, 79, and 94 of Appendix B):

```
syntaxTree = parse();
buildSymtab(syntaxTree);
typeCheck(syntaxTree);
codeGen(syntaxTree,codefile);
```

For flexibility, we have also built in conditional compilation flags that make it possible to build partial compilers. The flags, with their effect, are as follows:

FLAG	EFFECT IF SET	FILES NEEDED FOR COMPILATION (CUMULATIVE)
NO_PARSE	Builds a scanner-only compiler.	**globals.h**, **main.c**, **util.h**, **util.c**, **scan.h**, **scan.c**
NO_ANALYZE	Builds a compiler that parses and scans only.	**parse.h**, **parse.c**
NO_CODE	Builds a compiler that performs semantic analysis but generates no code.	**symtab.h**, **symtab.c**, **analyze.h**, **analyze.c**

Although this design for the TINY compiler is somewhat unrealistic, it has the pedagogical advantage that separate files correspond roughly to phases, and they can be discussed (and compiled and executed) individually in the chapters that follow.

The TINY compiler can be compiled by any ANSI C compiler. Assuming that the name of the executable file is **tiny**, it can be used to compile a TINY source program in the text file **sample.tny** by issuing the command

```
tiny sample.tny
```

(The compiler will also add the **.tny** suffix if it is omitted.) This will print a program listing to the screen (which can be redirected to a file) and (if code generation is activated) will also generate the target code file **sample.tm** (for use with the TM machine, described next).

There are several options for the information in the compilation listing. The following flags are available:

FLAG	EFFECT IF SET
EchoSource	Echoes the TINY source program to the listing together with line numbers.
TraceScan	Displays information on each token as the scanner recognizes it.
TraceParse	Displays the syntax tree in a linearized format.
TraceAnalyze	Displays summary information on the symbol table and type checking.
TraceCode	Prints code generation–tracing comments to the code file.

1.7.3 The TM Machine

We use the assembly language for this machine as the target language for the TINY compiler. The TM machine has just enough instructions to be an adequate target for a small language such as TINY. In fact, TM has some of the properties of Reduced Instruction Set Computers (or RISCs), in that all arithmetic and testing must take place in registers and the addressing modes are extremely limited. To give some idea of the simplicity of this machine, we translate the code for the C expression

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

into TM assembly language (compare this with the hypothetical assembly language for the same statement in Section 1.3, page 12):

```
LDC  1,0(0)        load 0 into reg 1
* the following instruction
* assumes index is at location 10 in memory
LD   0,10(1)       load val at 10+R1 into R0
LDC  1,2(0)        load 2 into reg 1
MUL  0,1,0         put R1*R0 into R0
LDC  1,0(0)        load 0 into reg 1
* the following instruction
* assumes a is at location 20 in memory
LDA  1,20(1)       load 20+R1 into R0
ADD  0,1,0         put R1+R0 into R0
LDC  1,6(0)        load 6 into reg 1
ST   1,0(0)        store R1 at 0+R0
```

We note that there are three addressing modes for the load operation, all given by different instructions: **LDC** is "load constant," **LD** is "load from memory," and **LDA** is "load address." We note also that addresses must always be given as "register+offset" values, as in **10(1)** (instruction 2 of the preceding code), which stands for the address computed by adding the offset 10 to the contents of register 1. (Since 0 was loaded into register 1 in the previous instruction, this actually refers to the absolute location 10.)[1] We also note that the arithmetic instructions **MUL** and **ADD** can have only register operands and are "three-address" instructions, in that the target register of the result can be specified independently of the operands (contrast this with the code in Section 1.3, page 12, where the operations were "two address").

Our simulator for the TM machine reads the assembly code directly from a file and executes it. Thus, we avoid the added complexity of translating the assembly language to machine code. However, our simulator is not a true assembler, in that there are no symbolic addresses or labels. Thus, the TINY compiler must still compute absolute addresses for jumps. Also, to avoid the extra complexity of linking with external input/output routines, the TM machine contains built-in I/O facilities for integers; these are read from and written to the standard devices during simulation.

The TM simulator can be compiled from the **tm.c** source code using any ANSI C compiler. Assuming the executable file is called **tm**, it can be used by issuing the command

```
tm sample.tm
```

where **sample.tm** is, for example, the code file produced by the TINY compiler from the **sample.tny** source file. This command causes the code file to be assembled and loaded; then the TM simulator can be run interactively. For example, if **sample.tny** is the sample program of Figure 1.4, then the factorial of 7 can be computed with the following interaction:

```
tm sample.tm
TM  simulation (enter h for help)...
Enter command: go
Enter value for IN instruction: 7
OUT instruction prints: 5040
HALT: 0,0,0
Halted
Enter command: quit
Simulation done.
```

1.8 C-MINUS: A LANGUAGE FOR A COMPILER PROJECT

A more extensive language than TINY, suitable for a compiler project, is described in Appendix A. It is a considerably restricted subset of C, which we will call C-Minus. It contains integers, integer arrays, and functions (including procedures, or void functions). It has local and global (static) declarations and (simple) recursive functions. It has an if-statement and a while-statement. It lacks almost everything else. A program consists of a sequence of function and variable declarations. A **main** function must be declared last. Execution begins with a call to **main**.[2]

[1] The **LDC** command also requires a register+offset format, but the register is ignored and the offset itself is loaded as a constant. This is due to the simple uniform format of the TM assembler.

As an example of a program in C-Minus, in Figure 1.5 we write the factorial program of Figure 1.4 using a recursive function. Input/output in this program is provided by a **read** function and a **write** function that can be defined in terms of the standard C functions **scanf** and **printf**.

C-Minus is a more complex language than TINY, particularly in its code generation requirements, but the TM machine is still a reasonable target for its compiler. In Appendix A we provide guidance on how to modify and extend the TINY compiler to C-Minus.

Figure 1.5
A C-Minus program that outputs the factorial of its input

```
int fact( int x )
/* recursive factorial function */
{ if (x > 1)
    return x * fact(x-1);
  else
    return 1;
}

void main( void )
{ int x;
  x = read();
  if (x > 0) write( fact(x) );
}
```

EXERCISES

1.1 Pick a familiar compiler that comes packaged with a development environment, and list all of the companion programs that are available with the compiler together with a brief description of their functions.

1.2 Given the C assignment

```
a[i+1] = a[i] + 2
```

draw a parse tree and a syntax tree for the expression, using the similar example in Section 1.3 as a guide.

1.3 Compilation errors can be loosely divided into two categories: syntax errors and semantic errors. Syntax errors include missing or incorrectly placed tokens, such as the missing

[2] For consistency with other functions in C-Minus, **main** is declared as a **void** function with a **void** parameter list. While this differs from ANSI C, many C compilers will accept this notation.

right parenthesis in the arithmetic expression `(2+3` . Semantic errors include incorrect types in expressions and undeclared variables (in most languages), such as the assignment `x = 2`, where `x` is an array variable.

a. Give two more examples of errors of each kind in a language of your choice.

b. Pick a compiler with which you are familiar and determine if it lists all syntax errors before semantic errors or if syntax and semantic errors are intermixed. What implication does this have for the number of passes?

1.4 This question assumes that you have a compiler that has an option to produce assembly language output.

a. Determine if your compiler performs constant folding optimizations.

b. A related but more advanced optimization is that of constant propagation: a variable that currently has a constant value is replaced by that value in expressions. For example, the code (in C syntax)

```
x = 4;
y = x + 2;
```

would, using constant propagation (and constant folding), be replaced by the code

```
x = 4;
y = 6;
```

Determine if your compiler performs constant propagation.

c. Give as many reasons as you can why constant propagation is more difficult than constant folding.

d. A situation related to constant propagation and constant folding is the use of named constants in a program. Using a named constant `x` instead of a variable, we can translate the above example as the following C code:

```
const int x = 4;
...
y = x + 2;
...
```

Determine if your compiler performs propagation/folding under these circumstances. How is this different from part (b)?

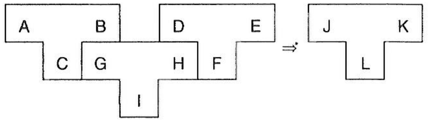
1.5 If your compiler will accept input directly from the keyboard, determine if your compiler reads the entire program before generating error messages or generates error messages as it encounters them. What implication does this have for the number of passes?

1.6 Describe the tasks performed by the following programs, and explain how these programs resemble or are related to compilers:

a. A language preprocessor **b.** A pretty-printer **c.** A text formatter

1.7 Suppose you have a Pascal-to-C translator written in C and a working C compiler. Use T-diagrams to describe the steps you would take to create a working Pascal compiler.

1.8 We have used an arrow ⇒ to indicate the reduction of a pattern of two T-diagrams to a single T-diagram. We may consider this arrow to be a "reduction relation" and form its transitive closure ⇒*, in which we allow a sequence of reductions to take place. Given the following diagram, in which letters stand for arbitrary languages, determine which languages must be equal for the reduction to be valid, and show the single reduction steps that make it valid:



Give a practical example of the reduction described by this diagram.

1.9 An alternative to the method of porting a compiler described in Section 1.6 and Figure 1.3 is to use an interpreter for the intermediate code produced by the compiler and to do away with a back end altogether. Such a method is used by the **Pascal P-system**, which includes a Pascal compiler that produces P-code, a kind of assembly code for a "generic" stack machine, and a P-code interpreter that simulates the execution of the P-code. Both the Pascal compiler and the P-code interpreter are written in P-code.

a. Describe the steps needed to obtain a working Pascal compiler on an arbitrary machine, given a Pascal P-system.

b. Describe the steps needed to obtain a working native-code compiler from your system in (a) (i.e., a compiler that produces executable code for the host machine, rather than using the P-code interpreter).

1.10 The process of porting a compiler can be considered as two distinct operations: **retargeting** (modifying the compiler to produce target code for a new machine) and **rehosting** (modifying the compiler to run on a new machine). Discuss the distinctness of these two operations in terms of T-diagrams.

NOTES AND REFERENCES

Most of the topics mentioned in this chapter are treated in more detail in subsequent chapters, and the Notes and References of those chapters will provide suitable references. For instance, Lex is studied in Chapter 2; Yacc in Chapter 5; type checking, symbol tables, and attribute analysis in Chapter 6; code generation, three-address code, and P-code in Chapter 8; and error handling in Chapters 4 and 5.

A standard comprehensive reference on compilers is Aho [1986], particularly for theory and algorithms. A text that gives many useful implementation hints is Fischer and LeBlanc [1991]. Complete descriptions of C compilers can be found in Fraser and Hanson [1995] and Holub [1990]. A popular C/C++ compiler whose source code is widely available through the internet is the Gnu compiler. It is described in detail in Stallman [1994].

For a survey of programming language concepts, with information on their interactions with translators, see Louden [1993] or Sethi [1996].

A useful reference for automata theory from a mathematical view (as opposed to the practical view taken here) is Hopcroft and Ullman [1979]. More on the Chomsky hierarchy can also be found there (as well as in Chapter 3).

A description of the early FORTRAN compilers can be found in Backus [1957] and Backus [1981]. A description of an early Algol60 compiler can be found in Randell and Russell [1964]. Pascal compilers are described in Barron [1981], where a description of the Pascal P-system can also be found (Nori [1981]).

The Ratfor preprocessor mentioned in Section 1.2 is described in Kernighan [1975]. The T-diagrams of Section 1.6 were introduced by Bratman [1961].

This text focuses on standard translation techniques useful for the translation of most languages. Additional techniques may be needed for efficient translation of languages outside the main tradition of Algol-based imperative languages. In particular, the translation of functional languages such as ML and Haskell has been the source of many new techniques, some of which may become important general techniques in the future. Descriptions of these techniques can be found in Appel [1992], Peyton Jones [1992], and Peyton Jones [1987]. The latter contains a description of Hindley-Milner type checking (mentioned in Section 1.1).

Chapter 2

Scanning

2.1 The Scanning Process
2.2 Regular Expressions
2.3 Finite Automata
2.4 From Regular Expressions to DFAs
2.5 Implementation of a TINY Scanner
2.6 Use of Lex to Generate a Scanner Automatically

The **scanning**, or **lexical analysis**, phase of a compiler has the task of reading the source program as a file of characters and dividing it up into tokens. Tokens are like the words of a natural language: each token is a sequence of characters that represents a unit of information in the source program. Typical examples are **keywords**, such as `if` and `while`, which are fixed strings of letters; **identifiers**, which are user-defined strings, usually consisting of letters and numbers and beginning with a letter; **special symbols**, such as the arithmetic symbols + and *; as well as a few multicharacter symbols, such as >= and <>. In each case a token represents a certain pattern of characters that is recognized, or matched, by the scanner from the beginning of the remaining input characters.

Since the task performed by the scanner is a special case of pattern matching, we need to study methods of pattern specification and recognition as they apply to the scanning process. These methods are primarily those of **regular expressions** and **finite automata**. However, a scanner is also the part of the compiler that handles the input of the source code, and since this input often involves a significant time overhead, the scanner must operate as efficiently as possible. Thus, we need also to pay close attention to the practical details of the scanner structure.

We divide the study of scanner issues as follows. First, we give an overview of the operation of a scanner and the structures and concepts involved. Then, we study regular expressions, a standard notation for representing the patterns in strings that form the lexical structure of a programming language. Following that, we study finite-state machines, or finite automata, which represent algorithms for recognizing string patterns given by regular expressions. We also study the process of constructing