Table of Contents

Chapter 1

Introduction

Programming languages are notations for describing computations to people and to machines. The world as we know it depends on programming languages, because all the software running on all the computers was written in some programming language. But, before a program can be run, it first must be translated into a form in which it can be executed by a computer.

The software systems that do this translation are called *compilers.*

This book is about how to design and implement compilers. We shall discover that a few basic ideas can be used to construct translators for a wide variety of languages and machines. Besides compilers, the principles and techniques for compiler design are applicable to so many other domains that they are likely to be reused many times in the career of a computer scientist. The study of compiler writing touches upon programming languages, machine architecture, language theory, algorithms, and software engineering.

In this preliminary chapter, we introduce the different forms of language translators, give a high level overview of the structure of a typical compiler, and discuss the trends in programming languages and machine architecture that are shaping compilers. We include some observations on the relationship between compiler design and computer-science theory and an outline of the applications of compiler technology that go beyond compilation. We end with a brief outline of key programming-language concepts that will be needed for our study of compilers.

1.1 Language Processors

Simply stated, a compiler is a program that can read a program in one language — the *source* language — and translate it into an equivalent program in another language — the *target* language; see Fig. 1.1. An important role of the compiler is to report any errors in the source program that it detects during the translation process.

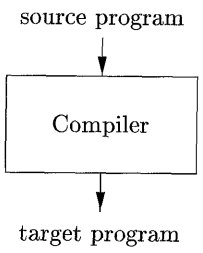


Figure 1.1: A compiler

If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs; see Fig. 1.2.

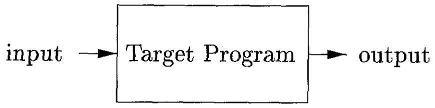


Figure 1.2: Running the target program

An *interpreter* is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user, as shown in Fig. 1.3.

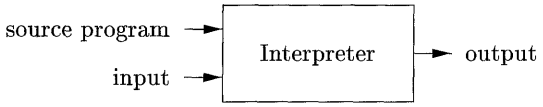


Figure 1.3: An interpreter

The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs . An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

Example 1.1 : Java language processors combine compilation and interpretation, as shown in Fig. 1.4. A Java source program may first be compiled into an intermediate form called *bytecodes.* The bytecodes are then interpreted by a virtual machine. A benefit of this arrangement is that bytecodes compiled on one machine can be interpreted on another machine, perhaps across a network.

In order to achieve faster processing of inputs to outputs, some Java compilers, called *just-in-time* compilers, translate the bytecodes into machine language immediately before they run the intermediate program to process the input. □

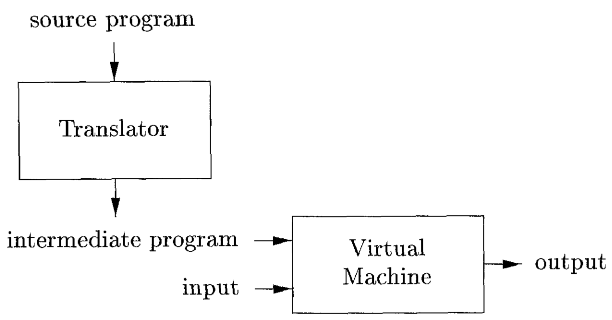


Figure 1.4: A hybrid compiler

In addition to a compiler, several other programs may be required to create an executable target program, as shown in Fig. 1.5. A source program may be divided into modules stored in separate files. The task of collecting the source program is sometimes entrusted to a separate program, called a *preprocessor*. The preprocessor may also expand shorthands, called macros, into source language statements.

The modified source program is then fed to a compiler. The compiler may produce an assembly-language program as its output, because assembly language is easier to produce as output and is easier to debug. The assembly language is then processed by a program called an *assembler* that produces relocatable machine code as its output.

Large programs are often compiled in pieces, so the relocatable machine code may have to be linked together with other relocatable object files and library files into the code that actually runs on the machine. The *linker* resolves external memory addresses, where the code in one file may refer to a location in another file. The *loader* then puts together all of the executable object files into memory for execution.

1.1.1 Exercises for Section 1.1

Exercise 1.1.1: What is the difference between a compiler and an interpreter?

Exercise 1.1.2: What are the advantages of (a) a compiler over an interpreter (b) an interpreter over a compiler?

Exercise 1.1.3: What advantages are there to a language-processing system in which the compiler produces assembly language rather than machine language?

Exercise 1.1.4: A compiler that translates a high-level language into another high-level language is called a *source-to-source* translator. What advantages are there to using C as a target language for a compiler?

Exercise 1.1.5: Describe some of the tasks that an assembler needs to perform.

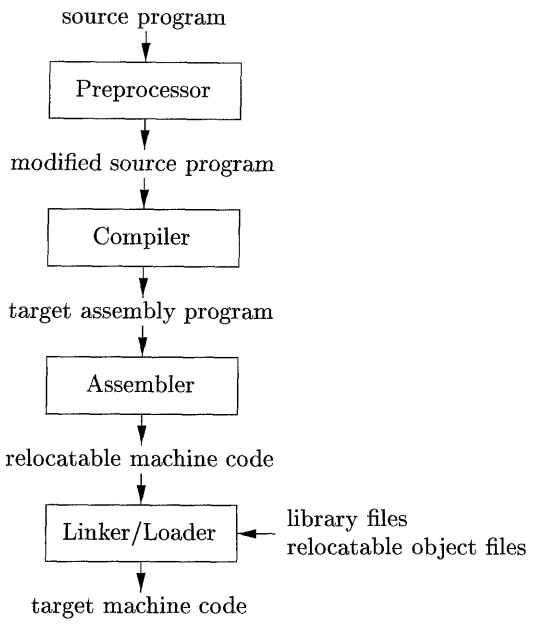


Figure 1.5: A language-processing system

1.2 The Structure of a Compiler

Up to this point we have treated a compiler as a single box that maps a source program into a semantically equivalent target program. If we open up this box a little, we see that there are two parts to this mapping: analysis and synthesis.

The *analysis* part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action. The analysis part also collects information about the source program and stores it in a data structure called a *symbol table*, which is passed along with the intermediate representation to the synthesis part.

The *synthesis* part constructs the desired target program from the intermediate representation and the information in the symbol table. The analysis part is often called the *front end* of the compiler; the synthesis part is the *back end*.

If we examine the compilation process in more detail, we see that it operates as a sequence of *phases*, each of which transforms one representation of the source program to another. A typical decomposition of a compiler into phases is shown in Fig. 1.6. In practice, several phases may be grouped together, and the intermediate representations between the grouped phases need not be constructed explicitly. The symbol table, which stores information about the

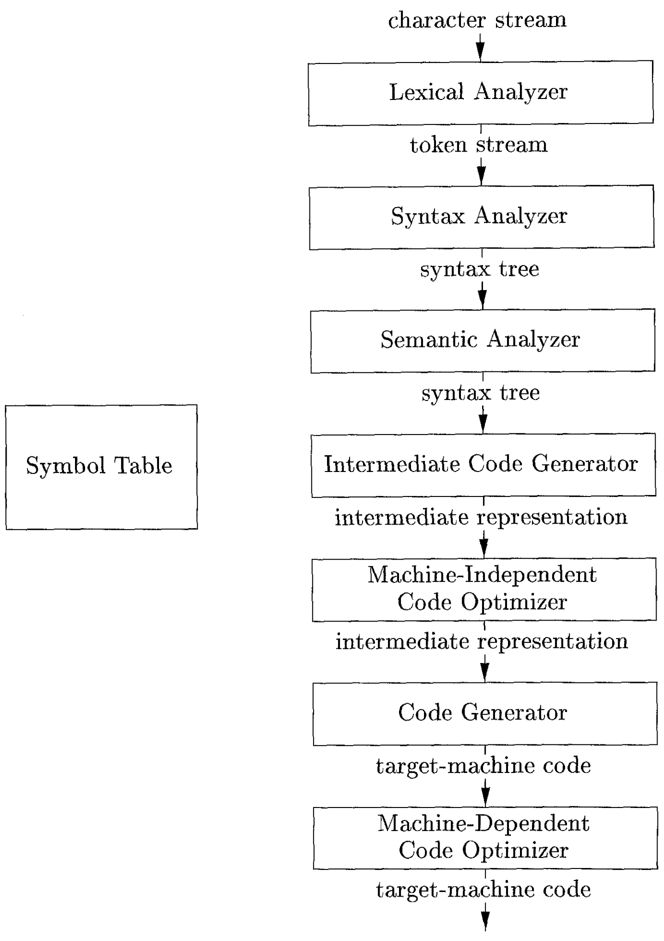


Figure 1.6: Phases of a compiler

entire source program, is used by all phases of the compiler.

Some compilers have a machine-independent optimization phase between the front end and the back end. The purpose of this optimization phase is to perform transformations on the intermediate representation, so that the back end can produce a better target program than it would have otherwise produced from an unoptimized intermediate representation. Since optimization is optional, one or the other of the two optimization phases shown in Fig. 1.6 may be missing.

1.2.1 Lexical Analysis

The first phase of a compiler is called *lexical analysis* or *scanning*. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called *lexemes*. For each lexeme, the lexical analyzer produces as output a *token* of the form

$$\langle \textit{token-name}, \textit{attribute-value} \rangle$$

that it passes on to the subsequent phase, syntax analysis. In the token, the first component *token-name* is an abstract symbol that is used during syntax analysis, and the second component *attribute-value* points to an entry in the symbol table for this token. Information from the symbol-table entry is needed for semantic analysis and code generation.

For example, suppose a source program contains the assignment statement

$$\texttt{position = initial + rate * 60} \tag{1.1}$$

The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyzer:

1. `position` is a lexeme that would be mapped into a token $\langle \mathbf{id}, 1 \rangle$, where **id** is an abstract symbol standing for *identifier* and 1 points to the symbol-table entry for `position`. The symbol-table entry for an identifier holds information about the identifier, such as its name and type.

2. The assignment symbol `=` is a lexeme that is mapped into the token $\langle = \rangle$. Since this token needs no attribute-value, we have omitted the second component. We could have used any abstract symbol such as **assign** for the token-name, but for notational convenience we have chosen to use the lexeme itself as the name of the abstract symbol.

3. `initial` is a lexeme that is mapped into the token $\langle \mathbf{id}, 2 \rangle$, where 2 points to the symbol-table entry for `initial`.

4. `+` is a lexeme that is mapped into the token $\langle + \rangle$.

5. `rate` is a lexeme that is mapped into the token $\langle \mathbf{id}, 3 \rangle$, where 3 points to the symbol-table entry for `rate`.

6. `*` is a lexeme that is mapped into the token $\langle * \rangle$.

7. `60` is a lexeme that is mapped into the token $\langle 60 \rangle$.[1]

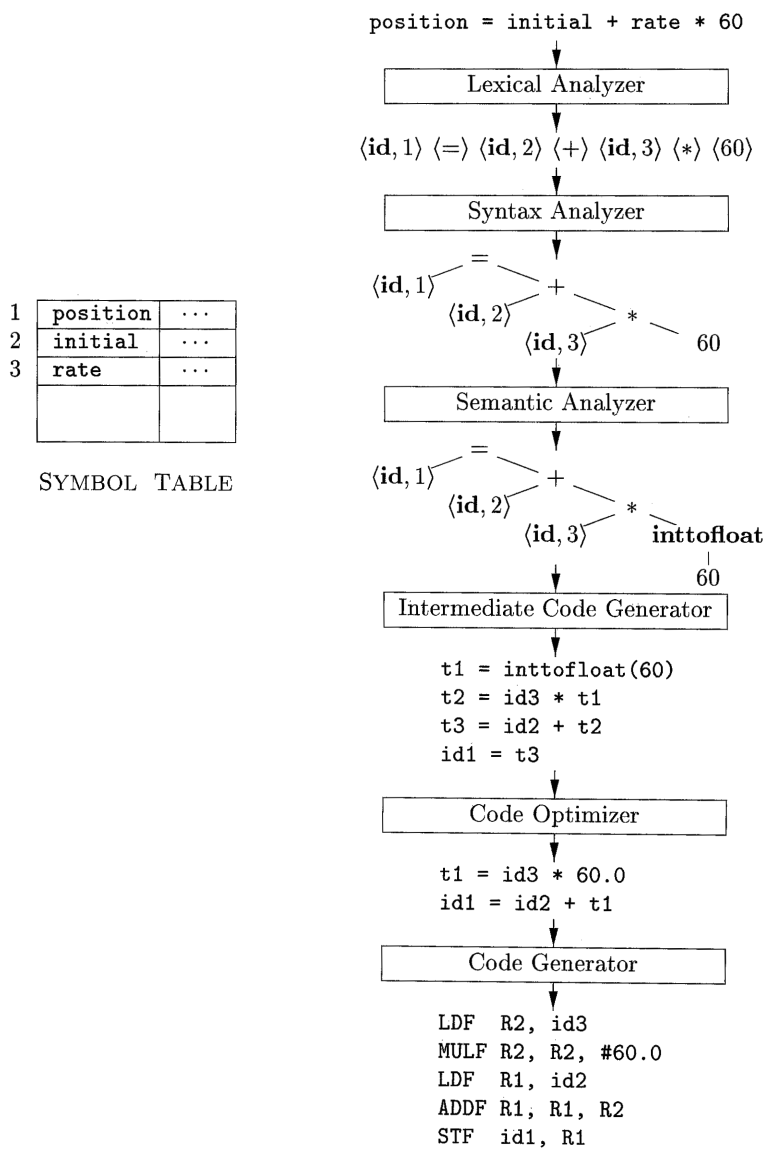
Blanks separating the lexemes would be discarded by the lexical analyzer.

Figure 1.7 shows the representation of the assignment statement (1.1) after lexical analysis as the sequence of tokens

$$\langle \mathbf{id}, 1 \rangle \; \langle = \rangle \; \langle \mathbf{id}, 2 \rangle \; \langle + \rangle \; \langle \mathbf{id}, 3 \rangle \; \langle * \rangle \; \langle 60 \rangle \tag{1.2}$$

In this representation, the token names $=$, $+$, and $*$ are abstract symbols for the assignment, addition, and multiplication operators, respectively.

[1]Technically speaking, for the lexeme `60` we should make up a token like $\langle \mathbf{number}, 4 \rangle$, where 4 points to the symbol table for the internal representation of integer 60 but we shall defer the discussion of tokens for numbers until Chapter 2. Chapter 3 discusses techniques for building lexical analyzers.



1	position	···
2	initial	···
3	rate	···

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
t1 = id3 * 60.0
id1 = id2 + t1
```

```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

Figure 1.7: Translation of an assignment statement

1.2.2 Syntax Analysis

The second phase of the compiler is *syntax analysis* or *parsing*. The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream. A typical representation is a *syntax tree* in which each interior node represents an operation and the children of the node represent the arguments of the operation. A syntax tree for the token stream (1.2) is shown as the output of the syntactic analyzer in Fig. 1.7.

This tree shows the order in which the operations in the assignment

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

are to be performed. The tree has an interior node labeled $*$ with $\langle \mathbf{id}, 3\rangle$ as its left child and the integer 60 as its right child. The node $\langle \mathbf{id}, 3\rangle$ represents the identifier `rate`. The node labeled $*$ makes it explicit that we must first multiply the value of `rate` by 60. The node labeled $+$ indicates that we must add the result of this multiplication to the value of `initial`. The root of the tree, labeled $=$, indicates that we must store the result of this addition into the location for the identifier `position`. This ordering of operations is consistent with the usual conventions of arithmetic which tell us that multiplication has higher precedence than addition, and hence that the multiplication is to be performed before the addition.

The subsequent phases of the compiler use the grammatical structure to help analyze the source program and generate the target program. In Chapter 4 we shall use context-free grammars to specify the grammatical structure of programming languages and discuss algorithms for constructing efficient syntax analyzers automatically from certain classes of grammars. In Chapters 2 and 5 we shall see that syntax-directed definitions can help specify the translation of programming language constructs.

1.2.3 Semantic Analysis

The *semantic analyzer* uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

An important part of semantic analysis is *type checking*, where the compiler checks that each operator has matching operands. For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array.

The language specification may permit some type conversions called *coercions*. For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers. If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number.

Such a coercion appears in Fig. 1.7. Suppose that `position`, `initial`, and `rate` have been declared to be floating-point numbers, and that the lexeme `60` by itself forms an integer. The type checker in the semantic analyzer in Fig. 1.7 discovers that the operator $*$ is applied to a floating-point number `rate` and an integer 60. In this case, the integer may be converted into a floating-point number. In Fig. 1.7, notice that the output of the semantic analyzer has an extra node for the operator **inttofloat**, which explicitly converts its integer argument into a floating-point number. Type checking and semantic analysis are discussed in Chapter 6.

1.2.4 Intermediate Code Generation

In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis.

After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine. This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into the target machine.

In Chapter 6, we consider an intermediate form called *three-address code*, which consists of a sequence of assembly-like instructions with three operands per instruction. Each operand can act like a register. The output of the intermediate code generator in Fig. 1.7 consists of the three-address code sequence

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```
(1.3)

There are several points worth noting about three-address instructions. First, each three-address assignment instruction has at most one operator on the right side. Thus, these instructions fix the order in which operations are to be done; the multiplication precedes the addition in the source program (1.1). Second, the compiler must generate a temporary name to hold the value computed by a three-address instruction. Third, some "three-address instructions" like the first and last in the sequence (1.3), above, have fewer than three operands.

In Chapter 6, we cover the principal intermediate representations used in compilers. Chapters 5 introduces techniques for syntax-directed translation that are applied in Chapter 6 to type checking and intermediate-code generation for typical programming language constructs such as expressions, flow-of-control constructs, and procedure calls.

1.2.5 Code Optimization

The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power. For example, a straightforward algorithm generates the intermediate code (1.3), using an instruction for each operator in the tree representation that comes from the semantic analyzer.

A simple intermediate code generation algorithm followed by code optimization is a reasonable way to generate good target code. The optimizer can deduce that the conversion of 60 from integer to floating point can be done once and for all at compile time, so the **inttofloat** operation can be eliminated by replacing the integer 60 by the floating-point number 60.0. Moreover, `t3` is used only once to transmit its value to `id1` so the optimizer can transform (1.3) into the shorter sequence

```
t1 = id3 * 60.0
id1 = id2 + t1
```
(1.4)

There is a great variation in the amount of code optimization different compilers perform. In those that do the most, the so-called "optimizing compilers," a significant amount of time is spent on this phase. There are simple optimizations that significantly improve the running time of the target program without slowing down compilation too much. The chapters from 8 on discuss machine-independent and machine-dependent optimizations in detail.

1.2.6 Code Generation

The code generator takes as input an intermediate representation of the source program and maps it into the target language. If the target language is machine code, registers or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task. A crucial aspect of code generation is the judicious assignment of registers to hold variables.

For example, using registers `R1` and `R2`, the intermediate code in (1.4) might get translated into the machine code

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

The first operand of each instruction specifies a destination. The `F` in each instruction tells us that it deals with floating-point numbers. The code in

(1.5) loads the contents of address `id3` into register `R2`, then multiplies it with floating-point constant 60.0. The `#` signifies that 60.0 is to be treated as an immediate constant. The third instruction moves `id2` into register `R1` and the fourth adds to it the value previously computed in register `R2`. Finally, the value in register `R1` is stored into the address of `id1`, so the code correctly implements the assignment statement (1.1). Chapter 8 covers code generation.

This discussion of code generation has ignored the important issue of storage allocation for the identifiers in the source program. As we shall see in Chapter 7, the organization of storage at run-time depends on the language being compiled. Storage-allocation decisions are made either during intermediate code generation or during code generation.

1.2.7 Symbol-Table Management

An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name. These attributes may provide information about the storage allocated for a name, its type, its scope (where in the program its value may be used), and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned.

The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name. The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly. Symbol tables are discussed in Chapter 2.

1.2.8 The Grouping of Phases into Passes

The discussion of phases deals with the logical organization of a compiler. In an implementation, activities from several phases may be grouped together into a *pass* that reads an input file and writes an output file. For example, the front-end phases of lexical analysis, syntax analysis, semantic analysis, and intermediate code generation might be grouped together into one pass. Code optimization might be an optional pass. Then there could be a back-end pass consisting of code generation for a particular target machine.

Some compiler collections have been created around carefully designed intermediate representations that allow the front end for a particular language to interface with the back end for a certain target machine. With these collections, we can produce compilers for different source languages for one target machine by combining different front ends with the back end for that target machine. Similarly, we can produce compilers for different target machines, by combining a front end with back ends for different target machines.

1.2.9 Compiler-Construction Tools

The compiler writer, like any software developer, can profitably use modern software development environments containing tools such as language editors, debuggers, version managers, profilers, test harnesses, and so on. In addition to these general software-development tools, other more specialized tools have been created to help implement various phases of a compiler.

These tools use specialized languages for specifying and implementing specific components, and many use quite sophisticated algorithms. The most successful tools are those that hide the details of the generation algorithm and produce components that can be easily integrated into the remainder of the compiler. Some commonly used compiler-construction tools include

1. *Parser generators* that automatically produce syntax analyzers from a grammatical description of a programming language.

2. *Scanner generators* that produce lexical analyzers from a regular-expression description of the tokens of a language.

3. *Syntax-directed translation engines* that produce collections of routines for walking a parse tree and generating intermediate code.

4. *Code-generator generators* that produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.

5. *Data-flow analysis engines* that facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data-flow analysis is a key part of code optimization.

6. *Compiler-construction toolkits* that provide an integrated set of routines for constructing various phases of a compiler.

We shall describe many of these tools throughout this book.

1.3 The Evolution of Programming Languages

The first electronic computers appeared in the 1940's and were programmed in machine language by sequences of 0's and 1's that explicitly told the computer what operations to execute and in what order. The operations themselves were very low level: move data from one location to another, add the contents of two registers, compare two values, and so on. Needless to say, this kind of programming was slow, tedious, and error prone. And once written, the programs were hard to understand and modify.

1.3.1 The Move to Higher-level Languages

The first step towards more people-friendly programming languages was the development of mnemonic assembly languages in the early 1950's. Initially, the instructions in an assembly language were just mnemonic representations of machine instructions. Later, macro instructions were added to assembly languages so that a programmer could define parameterized shorthands for frequently used sequences of machine instructions.

A major step towards higher-level languages was made in the latter half of the 1950's with the development of Fortran for scientific computation, Cobol for business data processing, and Lisp for symbolic computation. The philosophy behind these languages was to create higher-level notations with which programmers could more easily write numerical computations, business applications, and symbolic programs. These languages were so successful that they are still in use today.

In the following decades, many more languages were created with innovative features to help make programming easier, more natural, and more robust. Later in this chapter, we shall discuss some key features that are common to many modern programming languages.

Today, there are thousands of programming languages. They can be classified in a variety of ways. One classification is by generation. *First-generation languages* are the machine languages, *second-generation* the assembly languages, and *third-generation* the higher-level languages like Fortran, Cobol, Lisp, C, C++, C#, and Java. *Fourth-generation languages* are languages designed for specific applications like NOMAD for report generation, SQL for database queries, and Postscript for text formatting. The term *fifth-generation language* has been applied to logic- and constraint-based languages like Prolog and OPS5.

Another classification of languages uses the term *imperative* for languages in which a program specifies *how* a computation is to be done and *declarative* for languages in which a program specifies *what* computation is to be done. Languages such as C, C++, C#, and Java are imperative languages. In imperative languages there is a notion of program state and statements that change the state. Functional languages such as ML and Haskell and constraint logic languages such as Prolog are often considered to be declarative languages.

The term *von Neumann language* is applied to programming languages whose computational model is based on the von Neumann computer architecture. Many of today's languages, such as Fortran and C are von Neumann languages.

An *object-oriented language* is one that supports object-oriented programming, a programming style in which a program consists of a collection of objects that interact with one another. Simula 67 and Smalltalk are the earliest major object-oriented languages. Languages such as C++, C#, Java, and Ruby are more recent object-oriented languages.

Scripting languages are interpreted languages with high-level operators designed for "gluing together" computations. These computations were originally called "scripts." Awk, JavaScript, Perl, PHP, Python, Ruby, and Tcl are popular examples of scripting languages. Programs written in scripting languages are often much shorter than equivalent programs written in languages like C.

1.3.2 Impacts on Compilers

Since the design of programming languages and compilers are intimately related, the advances in programming languages placed new demands on compiler writers. They had to devise algorithms and representations to translate and support the new language features. Since the 1940's, computer architecture has evolved as well. Not only did the compiler writers have to track new language features, they also had to devise translation algorithms that would take maximal advantage of the new hardware capabilities.

Compilers can help promote the use of high-level languages by minimizing the execution overhead of the programs written in these languages. Compilers are also critical in making high-performance computer architectures effective on users' applications. In fact, the performance of a computer system is so dependent on compiler technology that compilers are used as a tool in evaluating architectural concepts before a computer is built.

Compiler writing is challenging. A compiler by itself is a large program. Moreover, many modern language-processing systems handle several source languages and target machines within the same framework; that is, they serve as collections of compilers, possibly consisting of millions of lines of code. Consequently, good software-engineering techniques are essential for creating and evolving modern language processors.

A compiler must translate correctly the potentially infinite set of programs that could be written in the source language. The problem of generating the optimal target code from a source program is undecidable in general; thus, compiler writers must evaluate tradeoffs about what problems to tackle and what heuristics to use to approach the problem of generating efficient code.

A study of compilers is also a study of how theory meets practice, as we shall see in Section 1.4.

The purpose of this text is to teach the methodology and fundamental ideas used in compiler design. It is not the intention of this text to teach all the algorithms and techniques that could be used for building a state-of-the-art language-processing system. However, readers of this text will acquire the basic knowledge and understanding to learn how to build a compiler relatively easily.

1.3.3 Exercises for Section 1.3

Exercise 1.3.1: Indicate which of the following terms:

a) imperative	b) declarative	c) von Neumann
d) object-oriented	e) functional	f) third-generation
g) fourth-generation	h) scripting	

apply to which of the following languages:

1) C	2) C++	3) Cobol	4) Fortran	5) Java
6) Lisp	7) ML	8) Perl	9) Python	10) VB.

1.4 The Science of Building a Compiler

Compiler design is full of beautiful examples where complicated real-world problems are solved by abstracting the essence of the problem mathematically. These serve as excellent illustrations of how abstractions can be used to solve problems: take a problem, formulate a mathematical abstraction that captures the key characteristics, and solve it using mathematical techniques. The problem formulation must be grounded in a solid understanding of the characteristics of computer programs, and the solution must be validated and refined empirically.

A compiler must accept all source programs that conform to the specification of the language; the set of source programs is infinite and any program can be very large, consisting of possibly millions of lines of code. Any transformation performed by the compiler while translating a source program must preserve the meaning of the program being compiled. Compiler writers thus have influence over not just the compilers they create, but all the programs that their compilers compile. This leverage makes writing compilers particularly rewarding; however, it also makes compiler development challenging.

1.4.1 Modeling in Compiler Design and Implementation

The study of compilers is mainly a study of how we design the right mathematical models and choose the right algorithms, while balancing the need for generality and power against simplicity and efficiency.

Some of most fundamental models are finite-state machines and regular expressions, which we shall meet in Chapter 3. These models are useful for describing the lexical units of programs (keywords, identifiers, and such) and for describing the algorithms used by the compiler to recognize those units. Also among the most fundamental models are context-free grammars, used to describe the syntactic structure of programming languages such as the nesting of parentheses or control constructs. We shall study grammars in Chapter 4. Similarly, trees are an important model for representing the structure of programs and their translation into object code, as we shall see in Chapter 5.

1.4.2 The Science of Code Optimization

The term "optimization" in compiler design refers to the attempts that a compiler makes to produce code that is more efficient than the obvious code. "Optimization" is thus a misnomer, since there is no way that the code produced by a compiler can be guaranteed to be as fast or faster than any other code that performs the same task.

In modern times, the optimization of code that a compiler performs has become both more important and more complex. It is more complex because processor architectures have become more complex, yielding more opportunities to improve the way code executes. It is more important because massively parallel computers require substantial optimization, or their performance suffers by orders of magnitude. With the likely prevalence of multicore machines (computers with chips that have large numbers of processors on them), all compilers will have to face the problem of taking advantage of multiprocessor machines.

It is hard, if not impossible, to build a robust compiler out of "hacks." Thus, an extensive and useful theory has been built up around the problem of optimizing code. The use of a rigorous mathematical foundation allows us to show that an optimization is correct and that it produces the desirable effect for all possible inputs. We shall see, starting in Chapter 9, how models such as graphs, matrices, and linear programs are necessary if the compiler is to produce well optimized code.

On the other hand, pure theory alone is insufficient. Like many real-world problems, there are no perfect answers. In fact, most of the questions that we ask in compiler optimization are undecidable. One of the most important skills in compiler design is the ability to formulate the right problem to solve. We need a good understanding of the behavior of programs to start with and thorough experimentation and evaluation to validate our intuitions.

Compiler optimizations must meet the following design objectives:

- The optimization must be correct, that is, preserve the meaning of the compiled program,
- The optimization must improve the performance of many programs,
- The compilation time must be kept reasonable, and
- The engineering effort required must be manageable.

It is impossible to overemphasize the importance of correctness. It is trivial to write a compiler that generates fast code if the generated code need not be correct! Optimizing compilers are so difficult to get right that we dare say that no optimizing compiler is completely error-free! Thus, the most important objective in writing a compiler is that it is correct.

The second goal is that the compiler must be effective in improving the performance of many input programs. Normally, performance means the speed of the program execution. Especially in embedded applications, we may also wish to minimize the size of the generated code. And in the case of mobile devices, it is also desirable that the code minimizes power consumption. Typically, the same optimizations that speed up execution time also conserve power. Besides performance, usability aspects such as error reporting and debugging are also important.

Third, we need to keep the compilation time short to support a rapid development and debugging cycle. This requirement has become easier to meet as machines get faster. Often, a program is first developed and debugged without program optimizations. Not only is the compilation time reduced, but more importantly, unoptimized programs are easier to debug, because the optimizations introduced by a compiler often obscure the relationship between the source code and the object code. Turning on optimizations in the compiler sometimes exposes new problems in the source program; thus testing must again be performed on the optimized code. The need for additional testing sometimes deters the use of optimizations in applications, especially if their performance is not critical.

Finally, a compiler is a complex system; we must keep the system simple to assure that the engineering and maintenance costs of the compiler are manageable. There is an infinite number of program optimizations that we could implement, and it takes a nontrivial amount of effort to create a correct and effective optimization. We must prioritize the optimizations, implementing only those that lead to the greatest benefits on source programs encountered in practice.

Thus, in studying compilers, we learn not only how to build a compiler, but also the general methodology of solving complex and open-ended problems. The approach used in compiler development involves both theory and experimentation. We normally start by formulating the problem based on our intuitions on what the important issues are.

1.5 Applications of Compiler Technology

Compiler design is not only about compilers, and many people use the technology learned by studying compilers in school, yet have never, strictly speaking, written (even part of) a compiler for a major programming language. Compiler technology has other important uses as well. Additionally, compiler design impacts several other areas of computer science. In this section, we review the most important interactions and applications of the technology.

1.5.1 Implementation of High-Level Programming Languages

A high-level programming language defines a programming abstraction: the programmer expresses an algorithm using the language, and the compiler must translate that program to the target language. Generally, higher-level programming languages are easier to program in, but are less efficient, that is, the target programs run more slowly. Programmers using a low-level language have more control over a computation and can, in principle, produce more efficient code. Unfortunately, lower-level programs are harder to write and — worse still — less portable, more prone to errors, and harder to maintain. Optimizing compilers include techniques to improve the performance of generated code, thus offsetting the inefficiency introduced by high-level abstractions.

Example 1.2: The **register** keyword in the C programming language is an early example of the interaction between compiler technology and language evolution. When the C language was created in the mid 1970s, it was considered necessary to let a programmer control which program variables reside in registers. This control became unnecessary as effective register-allocation techniques were developed, and most modern programs no longer use this language feature.

In fact, programs that use the **register** keyword may lose efficiency, because programmers often are not the best judge of very low-level matters like register allocation. The optimal choice of register allocation depends greatly on the specifics of a machine architecture. Hardwiring low-level resource-management decisions like register allocation may in fact hurt performance, especially if the program is run on machines other than the one for which it was written. □

The many shifts in the popular choice of programming languages have been in the direction of increased levels of abstraction. C was the predominant systems programming language of the 80's; many of the new projects started in the 90's chose C++; Java, introduced in 1995, gained popularity quickly in the late 90's. The new programming-language features introduced in each round spurred new research in compiler optimization. In the following, we give an overview on the main language features that have stimulated significant advances in compiler technology.

Practically all common programming languages, including C, Fortran and Cobol, support user-defined aggregate data types, such as arrays and structures, and high-level control flow, such as loops and procedure invocations. If we just take each high-level construct or data-access operation and translate it directly to machine code, the result would be very inefficient. A body of compiler optimizations, known as *data-flow optimizations*, has been developed to analyze the flow of data through the program and removes redundancies across these constructs. They are effective in generating code that resembles code written by a skilled programmer at a lower level.

Object orientation was first introduced in Simula in 1967, and has been incorporated in languages such as Smalltalk, C++, C#, and Java. The key ideas behind object orientation are

1. Data abstraction and

2. Inheritance of properties,

both of which have been found to make programs more modular and easier to maintain. Object-oriented programs are different from those written in many other languages, in that they consist of many more, but smaller, procedures (called *methods* in object-oriented terms). Thus, compiler optimizations must be able to perform well across the procedural boundaries of the source program. Procedure inlining, which is the replacement of a procedure call by the body of the procedure, is particularly useful here. Optimizations to speed up virtual method dispatches have also been developed.

Java has many features that make programming easier, many of which have been introduced previously in other languages. The Java language is type-safe; that is, an object cannot be used as an object of an unrelated type. All array accesses are checked to ensure that they lie within the bounds of the array. Java has no pointers and does not allow pointer arithmetic. It has a built-in garbage-collection facility that automatically frees the memory of variables that are no longer in use. While all these features make programming easier, they incur a run-time overhead. Compiler optimizations have been developed to reduce the overhead, for example, by eliminating unnecessary range checks and by allocating objects that are not accessible beyond a procedure on the stack instead of the heap. Effective algorithms also have been developed to minimize the overhead of garbage collection.

In addition, Java is designed to support portable and mobile code. Programs are distributed as Java bytecode, which must either be interpreted or compiled into native code dynamically, that is, at run time. Dynamic compilation has also been studied in other contexts, where information is extracted dynamically at run time and used to produce better-optimized code. In dynamic optimization, it is important to minimize the compilation time as it is part of the execution overhead. A common technique used is to only compile and optimize those parts of the program that will be frequently executed.

1.5.2 Optimizations for Computer Architectures

The rapid evolution of computer architectures has also led to an insatiable demand for new compiler technology. Almost all high-performance systems take advantage of the same two basic techniques: *parallelism* and *memory hierarchies*. Parallelism can be found at several levels: at the *instruction level*, where multiple operations are executed simultaneously and at the *processor level*, where different threads of the same application are run on different processors. Memory hierarchies are a response to the basic limitation that we can build very fast storage or very large storage, but not storage that is both fast and large.

Parallelism

All modern microprocessors exploit instruction-level parallelism. However, this parallelism can be hidden from the programmer. Programs are written as if all instructions were executed in sequence; the hardware dynamically checks for dependencies in the sequential instruction stream and issues them in parallel when possible. In some cases, the machine includes a hardware scheduler that can change the instruction ordering to increase the parallelism in the program. Whether the hardware reorders the instructions or not, compilers can rearrange the instructions to make instruction-level parallelism more effective.

Instruction-level parallelism can also appear explicitly in the instruction set. VLIW (Very Long Instruction Word) machines have instructions that can issue multiple operations in parallel. The Intel IA64 is a well-known example of such an architecture. All high-performance, general-purpose microprocessors also include instructions that can operate on a vector of data at the same time. Compiler techniques have been developed to generate code automatically for such machines from sequential programs.

Multiprocessors have also become prevalent; even personal computers often have multiple processors. Programmers can write multithreaded code for multiprocessors, or parallel code can be automatically generated by a compiler from conventional sequential programs. Such a compiler hides from the programmers the details of finding parallelism in a program, distributing the computation across the machine, and minimizing synchronization and communication among the processors. Many scientific-computing and engineering applications are computation-intensive and can benefit greatly from parallel processing. Parallelization techniques have been developed to translate automatically sequential scientific programs into multiprocessor code.

Memory Hierarchies

A memory hierarchy consists of several levels of storage with different speeds and sizes, with the level closest to the processor being the fastest but smallest. The average memory-access time of a program is reduced if most of its accesses are satisfied by the faster levels of the hierarchy. Both parallelism and the existence of a memory hierarchy improve the potential performance of a machine, but they must be harnessed effectively by the compiler to deliver real performance on an application.

Memory hierarchies are found in all machines. A processor usually has a small number of registers consisting of hundreds of bytes, several levels of caches containing kilobytes to megabytes, physical memory containing megabytes to gigabytes, and finally secondary storage that contains gigabytes and beyond. Correspondingly, the speed of accesses between adjacent levels of the hierarchy can differ by two or three orders of magnitude. The performance of a system is often limited not by the speed of the processor but by the performance of the memory subsystem. While compilers traditionally focus on optimizing the processor execution, more emphasis is now placed on making the memory hierarchy more effective.

Using registers effectively is probably the single most important problem in optimizing a program. Unlike registers that have to be managed explicitly in software, caches and physical memories are hidden from the instruction set and are managed by hardware. It has been found that cache-management policies implemented by hardware are not effective in some cases, especially in scientific code that has large data structures (arrays, typically). It is possible to improve the effectiveness of the memory hierarchy by changing the layout of the data, or changing the order of instructions accessing the data. We can also change the layout of code to improve the effectiveness of instruction caches.

1.5.3 Design of New Computer Architectures

In the early days of computer architecture design, compilers were developed after the machines were built. That has changed. Since programming in high-level languages is the norm, the performance of a computer system is determined not by its raw speed but also by how well compilers can exploit its features. Thus, in modern computer architecture development, compilers are developed in the processor-design stage, and compiled code, running on simulators, is used to evaluate the proposed architectural features.

RISC

One of the best known examples of how compilers influenced the design of computer architecture was the invention of the RISC (Reduced Instruction-Set Computer) architecture. Prior to this invention, the trend was to develop progressively complex instruction sets intended to make assembly programming easier; these architectures were known as CISC (Complex Instruction-Set Computer). For example, CISC instruction sets include complex memory-addressing modes to support data-structure accesses and procedure-invocation instructions that save registers and pass parameters on the stack.

Compiler optimizations often can reduce these instructions to a small number of simpler operations by eliminating the redundancies across complex instructions. Thus, it is desirable to build simple instruction sets; compilers can use them effectively and the hardware is much easier to optimize.

Most general-purpose processor architectures, including PowerPC, SPARC, MIPS, Alpha, and PA-RISC, are based on the RISC concept. Although the x86 architecture—the most popular microprocessor—has a CISC instruction set, many of the ideas developed for RISC machines are used in the implementation of the processor itself. Moreover, the most effective way to use a high-performance x86 machine is to use just its simple instructions.

Specialized Architectures

Over the last three decades, many architectural concepts have been proposed. They include data flow machines, vector machines, VLIW (Very Long Instruction Word) machines, SIMD (Single Instruction, Multiple Data) arrays of processors, systolic arrays, multiprocessors with shared memory, and multiprocessors with distributed memory. The development of each of these architectural concepts was accompanied by the research and development of corresponding compiler technology.

Some of these ideas have made their way into the designs of embedded machines. Since entire systems can fit on a single chip, processors need no longer be prepackaged commodity units, but can be tailored to achieve better cost-effectiveness for a particular application. Thus, in contrast to general-purpose processors, where economies of scale have led computer architectures to converge, application-specific processors exhibit a diversity of computer architectures. Compiler technology is needed not only to support programming for these architectures, but also to evaluate proposed architectural designs.

1.5.4 Program Translations

While we normally think of compiling as a translation from a high-level language to the machine level, the same technology can be applied to translate between different kinds of languages. The following are some of the important applications of program-translation techniques.

Binary Translation

Compiler technology can be used to translate the binary code for one machine to that of another, allowing a machine to run programs originally compiled for another instruction set. Binary translation technology has been used by various computer companies to increase the availability of software for their machines. In particular, because of the domination of the x86 personal-computer market, most software titles are available as x86 code. Binary translators have been developed to convert x86 code into both Alpha and Sparc code. Binary translation was also used by Transmeta Inc. in their implementation of the x86 instruction set. Instead of executing the complex x86 instruction set directly in hardware, the Transmeta Crusoe processor is a VLIW processor that relies on binary translation to convert x86 code into native VLIW code.

Binary translation can also be used to provide backward compatibility. When the processor in the Apple Macintosh was changed from the Motorola MC 68040 to the PowerPC in 1994, binary translation was used to allow PowerPC processors run legacy MC 68040 code.

Hardware Synthesis

Not only is most software written in high-level languages; even hardware designs are mostly described in high-level hardware description languages like Verilog and VHDL (Very high-speed integrated circuit Hardware Description Language). Hardware designs are typically described at the register transfer level (RTL), where variables represent registers and expressions represent combinational logic. Hardware-synthesis tools translate RTL descriptions automatically into gates, which are then mapped to transistors and eventually to a physical layout. Unlike compilers for programming languages, these tools often take hours optimizing the circuit. Techniques to translate designs at higher levels, such as the behavior or functional level, also exist.

Database Query Interpreters

Besides specifying software and hardware, languages are useful in many other applications. For example, query languages, especially SQL (Structured Query

Language), are used to search databases. Database queries consist of predicates containing relational and boolean operators. They can be interpreted or compiled into commands to search a database for records satisfying that predicate.

Compiled Simulation

Simulation is a general technique used in many scientific and engineering disciplines to understand a phenomenon or to validate a design. Inputs to a simulator usually include the description of the design and specific input parameters for that particular simulation run. Simulations can be very expensive. We typically need to simulate many possible design alternatives on many different input sets, and each experiment may take days to complete on a high-performance machine. Instead of writing a simulator that interprets the design, it is faster to compile the design to produce machine code that simulates that particular design natively. Compiled simulation can run orders of magnitude faster than an interpreter-based approach. Compiled simulation is used in many state-of-the-art tools that simulate designs written in Verilog or VHDL.

1.5.5 Software Productivity Tools

Programs are arguably the most complicated engineering artifacts ever produced; they consist of many many details, every one of which must be correct before the program will work completely. As a result, errors are rampant in programs; errors may crash a system, produce wrong results, render a system vulnerable to security attacks, or even lead to catastrophic failures in critical systems. Testing is the primary technique for locating errors in programs.

An interesting and promising complementary approach is to use data-flow analysis to locate errors statically (that is, before the program is run). Data-flow analysis can find errors along all the possible execution paths, and not just those exercised by the input data sets, as in the case of program testing. Many of the data-flow-analysis techniques, originally developed for compiler optimizations, can be used to create tools that assist programmers in their software engineering tasks.

The problem of finding all program errors is undecidable. A data-flow analysis may be designed to warn the programmers of all possible statements violating a particular category of errors. But if most of these warnings are false alarms, users will not use the tool. Thus, practical error detectors are often neither sound nor complete. That is, they may not find all the errors in the program, and not all errors reported are guaranteed to be real errors. Nonetheless, various static analyses have been developed and shown to be effective in finding errors, such as dereferencing null or freed pointers, in real programs. The fact that error detectors may be unsound makes them significantly different from compiler optimizations. Optimizers must be conservative and cannot alter the semantics of the program under any circumstances.

In the balance of this section, we shall mention several ways in which program analysis, building upon techniques originally developed to optimize code in compilers, have improved software productivity. Of special importance are techniques that detect statically when a program might have a security vulnerability.

Type Checking

Type checking is an effective and well-established technique to catch inconsistencies in programs. It can be used to catch errors, for example, where an operation is applied to the wrong type of object, or if parameters passed to a procedure do not match the signature of the procedure. Program analysis can go beyond finding type errors by analyzing the flow of data through a program. For example, if a pointer is assigned `null` and then immediately dereferenced, the program is clearly in error.

The same technology can be used to catch a variety of security holes, in which an attacker supplies a string or other data that is used carelessly by the program. A user-supplied string can be labeled with a type "dangerous." If this string is not checked for proper format, then it remains "dangerous," and if a string of this type is able to influence the control-flow of the code at some point in the program, then there is a potential security flaw.

Bounds Checking

It is easier to make mistakes when programming in a lower-level language than a higher-level one. For example, many security breaches in systems are caused by buffer overflows in programs written in C. Because C does not have array-bounds checks, it is up to the user to ensure that the arrays are not accessed out of bounds. Failing to check that the data supplied by the user can overflow a buffer, the program may be tricked into storing user data outside of the buffer. An attacker can manipulate the input data that causes the program to misbehave and compromise the security of the system. Techniques have been developed to find buffer overflows in programs, but with limited success.

Had the program been written in a safe language that includes automatic range checking, this problem would not have occurred. The same data-flow analysis that is used to eliminate redundant range checks can also be used to locate buffer overflows. The major difference, however, is that failing to eliminate a range check would only result in a small run-time cost, while failing to identify a potential buffer overflow may compromise the security of the system. Thus, while it is adequate to use simple techniques to optimize range checks, sophisticated analyses, such as tracking the values of pointers across procedures, are needed to get high-quality results in error detection tools.

Memory-Management Tools

Garbage collection is another excellent example of the tradeoff between efficiency and a combination of ease of programming and software reliability. Automatic memory management obliterates all memory-management errors (e.g., "memory leaks"), which are a major source of problems in C and C++ programs. Various tools have been developed to help programmers find memory management errors. For example, Purify is a widely used tool that dynamically catches memory management errors as they occur. Tools that help identify some of these problems statically have also been developed.

1.6 Programming Language Basics

In this section, we shall cover the most important terminology and distinctions that appear in the study of programming languages. It is not our purpose to cover all concepts or all the popular programming languages. We assume that the reader is familiar with at least one of C, C++, C#, or Java, and may have encountered other languages as well.

1.6.1 The Static/Dynamic Distinction

Among the most important issues that we face when designing a compiler for a language is what decisions can the compiler make about a program. If a language uses a policy that allows the compiler to decide an issue, then we say that the language uses a *static* policy or that the issue can be decided at *compile time*. On the other hand, a policy that only allows a decision to be made when we execute the program is said to be a *dynamic policy* or to require a decision at *run time*.

One issue on which we shall concentrate is the scope of declarations. The *scope* of a declaration of x is the region of the program in which uses of x refer to this declaration. A language uses *static scope* or *lexical scope* if it is possible to determine the scope of a declaration by looking only at the program. Otherwise, the language uses *dynamic scope*. With dynamic scope, as the program runs, the same use of x could refer to any of several different declarations of x.

Most languages, such as C and Java, use static scope. We shall discuss static scoping in Section 1.6.3.

Example 1.3: As another example of the static/dynamic distinction, consider the use of the term "static" as it applies to data in a Java class declaration. In Java, a variable is a name for a location in memory used to hold a data value. Here, "static" refers not to the scope of the variable, but rather to the ability of the compiler to determine the location in memory where the declared variable can be found. A declaration like

```
    public static int x;
```

makes x a *class variable* and says that there is only one copy of x, no matter how many objects of this class are created. Moreover, the compiler can determine a location in memory where this integer x will be held. In contrast, had "static" been omitted from this declaration, then each object of the class would have its own location where x would be held, and the compiler could not determine all these places in advance of running the program. □

1.6.2 Environments and States

Another important distinction we must make when discussing programming languages is whether changes occurring as the program runs affect the values of data elements or affect the interpretation of names for that data. For example, the execution of an assignment such as `x = y + 1` changes the value denoted by the name x. More specifically, the assignment changes the value in whatever location is denoted by x.

It may be less clear that the location denoted by x can change at run time. For instance, as we discussed in Example 1.3, if x is not a static (or "class") variable, then every object of the class has its own location for an instance of variable x. In that case, the assignment to x can change any of those "instance" variables, depending on the object to which a method containing that assignment is applied.

$$\begin{tikzcd}
\text{names} \arrow[r, bend left, "\textit{environment}"] & \text{locations (variables)} \arrow[r, bend left, "\textit{state}"] & \text{values}
\end{tikzcd}$$

Figure 1.8: Two-stage mapping from names to values

The association of names with locations in memory (the *store*) and then with values can be described by two mappings that change as the program runs (see Fig. 1.8):

1. The *environment* is a mapping from names to locations in the store. Since variables refer to locations ("l-values" in the terminology of C), we could alternatively define an environment as a mapping from names to variables.

2. The *state* is a mapping from locations in store to their values. That is, the state maps l-values to their corresponding r-values, in the terminology of C.

Environments change according to the scope rules of a language.

Example 1.4: Consider the C program fragment in Fig. 1.9. Integer i is declared a global variable, and also declared as a variable local to function f. When f is executing, the environment adjusts so that name i refers to the

```
...
int i;                    /* global i        */
...
void f(···) {
     int i;               /* local i         */
     ...
     i = 3;               /* use of local i  */
     ...
}
...
     x = i + 1;           /* use of global i */
```

Figure 1.9: Two declarations of the name i

location reserved for the i that is local to f, and any use of i, such as the assignment `i = 3` shown explicitly, refers to that location. Typically, the local i is given a place on the run-time stack.

Whenever a function g other than f is executing, uses of i cannot refer to the i that is local to f. Uses of name i in g must be within the scope of some other declaration of i. An example is the explicitly shown statement `x = i+1`, which is inside some procedure whose definition is not shown. The i in $i + 1$ presumably refers to the global i. As in most languages, declarations in C must precede their use, so a function that comes before the global i cannot refer to it. □

The environment and state mappings in Fig. 1.8 are dynamic, but there are a few exceptions:

1. *Static versus dynamic binding* of names to locations. Most binding of names to locations is dynamic, and we discuss several approaches to this binding throughout the section. Some declarations, such as the global i in Fig. 1.9, can be given a location in the store once and for all, as the compiler generates object code.[2]

2. *Static versus dynamic binding* of locations to values. The binding of locations to values (the second stage in Fig. 1.8), is generally dynamic as well, since we cannot tell the value in a location until we run the program. Declared constants are an exception. For instance, the C definition

```
#define ARRAYSIZE 1000
```

[2]Technically, the C compiler will assign a location in virtual memory for the global i, leaving it to the loader and the operating system to determine where in the physical memory of the machine i will be located. However, we shall not worry about "relocation" issues such as these, which have no impact on compiling. Instead, we treat the address space that the compiler uses for its output code as if it gave physical memory locations.

Names, Identifiers, and Variables

Although the terms "name" and "variable," often refer to the same thing, we use them carefully to distinguish between compile-time names and the run-time locations denoted by names.

An *identifier* is a string of characters, typically letters or digits, that refers to (identifies) an entity, such as a data object, a procedure, a class, or a type. All identifiers are names, but not all names are identifiers. Names can also be expressions. For example, the name $x.y$ might denote the field y of a structure denoted by x. Here, x and y are identifiers, while $x.y$ is a name, but not an identifier. Composite names like $x.y$ are called *qualified* names.

A *variable* refers to a particular location of the store. It is common for the same identifier to be declared more than once; each such declaration introduces a new variable. Even if each identifier is declared just once, an identifier local to a recursive procedure will refer to different locations of the store at different times.

binds the name `ARRAYSIZE` to the value 1000 statically. We can determine this binding by looking at the statement, and we know that it is impossible for this binding to change when the program executes.

1.6.3 Static Scope and Block Structure

Most languages, including C and its family, use static scope. The scope rules for C are based on program structure; the scope of a declaration is determined implicitly by where the declaration appears in the program. Later languages, such as C++, Java, and C#, also provide explicit control over scopes through the use of keywords like **public**, **private**, and **protected**.

In this section we consider static-scope rules for a language with blocks, where a *block* is a grouping of declarations and statements. C uses braces { and } to delimit a block; the alternative use of **begin** and **end** for the same purpose dates back to Algol.

Example 1.5: To a first approximation, the C static-scope policy is as follows:

1. A C program consists of a sequence of top-level declarations of variables and functions.

2. Functions may have variable declarations within them, where variables include local variables and parameters. The scope of each such declaration is restricted to the function in which it appears.

Procedures, Functions, and Methods

To avoid saying "procedures, functions, or methods," each time we want to talk about a subprogram that may be called, we shall usually refer to all of them as "procedures." The exception is that when talking explicitly of programs in languages like C that have only functions, we shall refer to them as "functions." Or, if we are discussing a language like Java that has only methods, we shall use that term instead.

A function generally returns a value of some type (the "return type"), while a procedure does not return any value. C and similar languages, which have only functions, treat procedures as functions that have a special return type "void," to signify no return value. Object-oriented languages like Java and C++ use the term "methods." These can behave like either functions or procedures, but are associated with a particular class.

3. The scope of a top-level declaration of a name x consists of the entire program that follows, with the exception of those statements that lie within a function that also has a declaration of x.

The additional detail regarding the C static-scope policy deals with variable declarations within statements. We examine such declarations next and in Example 1.6. □

In C, the syntax of blocks is given by

1. One type of statement is a block. Blocks can appear anywhere that other types of statements, such as assignment statements, can appear.

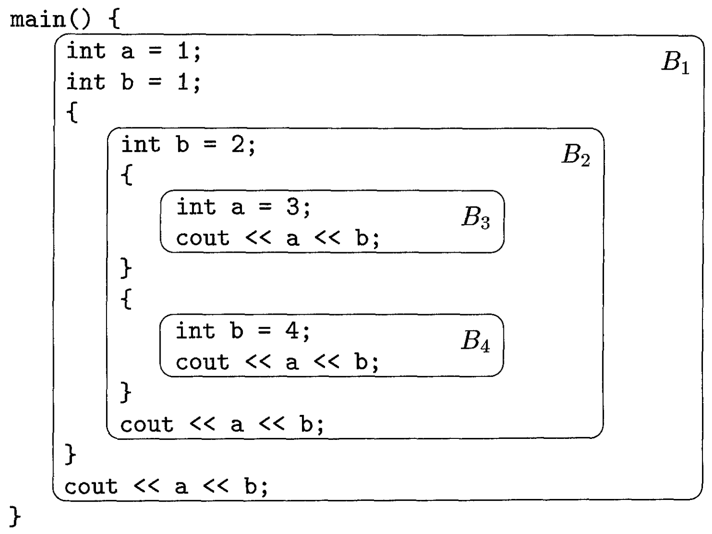
2. A block is a sequence of declarations followed by a sequence of statements, all surrounded by braces.

Note that this syntax allows blocks to be nested inside each other. This nesting property is referred to as *block structure*. The C family of languages has block structure, except that a function may not be defined inside another function.

We say that a declaration D "belongs" to a block B if B is the most closely nested block containing D; that is, D is located within B, but not within any block that is nested within B.

The static-scope rule for variable declarations in a block-structured languages is as follows. If declaration D of name x belongs to block B, then the scope of D is all of B, except for any blocks B' nested to any depth within B, in which x is redeclared. Here, x is redeclared in B' if some other declaration D' of the same name x belongs to B'.

An equivalent way to express this rule is to focus on a use of a name x. Let $B_1, B_2, \ldots, B_k$ be all the blocks that surround this use of x, with B_k the smallest, nested within B_{k-1}, which is nested within B_{k-2}, and so on. Search for the largest i such that there is a declaration of x belonging to B_i. This use of x refers to the declaration in B_i. Alternatively, this use of x is within the scope of the declaration in B_i.



```
main() {
    int a = 1;
    int b = 1;
    {
        int b = 2;
        {
            int a = 3;
            cout << a << b;
        }
        {
            int b = 4;
            cout << a << b;
        }
        cout << a << b;
    }
    cout << a << b;
}
```

Figure 1.10: Blocks in a C++ program

Example 1.6: The C++ program in Fig. 1.10 has four blocks, with several definitions of variables a and b. As a memory aid, each declaration initializes its variable to the number of the block to which it belongs.

For instance, consider the declaration `int a = 1` in block B_1. Its scope is all of B_1, except for those blocks nested (perhaps deeply) within B_1 that have their own declaration of a. B_2, nested immediately within B_1, does not have a declaration of a, but B_3 does. B_4 does not have a declaration of a, so block B_3 is the only place in the entire program that is outside the scope of the declaration of the name a that belongs to B_1. That is, this scope includes B_4 and all of B_2 except for the part of B_2 that is within B_3. The scopes of all five declarations are summarized in Fig. 1.11.

From another point of view, let us consider the output statement in block B_4 and bind the variables a and b used there to the proper declarations. The list of surrounding blocks, in order of increasing size, is B_4, B_2, B_1. Note that B_3 does not surround the point in question. B_4 has a declaration of b, so it is to this declaration that this use of b refers, and the value of b printed is 4. However, B_4 does not have a declaration of a, so we next look at B_2. That block does not have a declaration of a either, so we proceed to B_1. Fortunately,

DECLARATION	SCOPE
`int a = 1;`	$B_1 - B_3$
`int b = 1;`	$B_1 - B_2$
`int b = 2;`	$B_2 - B_4$
`int a = 3;`	B_3
`int b = 4;`	B_4

Figure 1.11: Scopes of declarations in Example 1.6

there is a declaration `int a = 1` belonging to that block, so the value of a printed is 1. Had there been no such declaration, the program would have been erroneous. □

1.6.4 Explicit Access Control

Classes and structures introduce a new scope for their members. If p is an object of a class with a field (member) x, then the use of x in $p.x$ refers to field x in the class definition. In analogy with block structure, the scope of a member declaration x in a class C extends to any subclass C', except if C' has a local declaration of the same name x.

Through the use of keywords like **public**, **private**, and **protected**, object-oriented languages such as C++ or Java provide explicit control over access to member names in a superclass. These keywords support *encapsulation* by restricting access. Thus, private names are purposely given a scope that includes only the method declarations and definitions associated with that class and any "friend" classes (the C++ term). Protected names are accessible to subclasses. Public names are accessible from outside the class.

In C++, a class definition may be separated from the definitions of some or all of its methods. Therefore, a name x associated with the class C may have a region of the code that is outside its scope, followed by another region (a method definition) that is within its scope. In fact, regions inside and outside the scope may alternate, until all the methods have been defined.

1.6.5 Dynamic Scope

Technically, any scoping policy is dynamic if it is based on factor(s) that can be known only when the program executes. The term *dynamic scope*, however, usually refers to the following policy: a use of a name x refers to the declaration of x in the most recently called procedure with such a declaration. Dynamic scoping of this type appears only in special situations. We shall consider two examples of dynamic policies: macro expansion in the C preprocessor and method resolution in object-oriented programming.

Declarations and Definitions

The apparently similar terms "declaration" and "definition" for programming-language concepts are actually quite different. Declarations tell us about the types of things, while definitions tell us about their values. Thus, `int i` is a declaration of i, while `i = 1` is a definition of i.

The difference is more significant when we deal with methods or other procedures. In C++, a method is declared in a class definition, by giving the types of the arguments and result of the method (often called the *signature* for the method. The method is then defined, i.e., the code for executing the method is given, in another place. Similarly, it is common to define a C function in one file and declare it in other files where the function is used.

Example 1.7: In the C program of Fig. 1.12, identifier a is a macro that stands for expression $(x + 1)$. But what is x? We cannot resolve x statically, that is, in terms of the program text.

```
#define a (x+1)

int x = 2;

void b() { int x = 1; printf("%d\n", a); }

void c() { printf("%d\n", a); }

void main() { b(); c(); }
```

Figure 1.12: A macro whose names must be scoped dynamically

In fact, in order to interpret x, we must use the usual dynamic-scope rule. We examine all the function calls that are currently active, and we take the most recently called function that has a declaration of x. It is to this declaration that the use of x refers.

In the example of Fig. 1.12, the function *main* first calls function b. As b executes, it prints the value of the macro a. Since $(x + 1)$ must be substituted for a, we resolve this use of x to the declaration `int x=1` in function b. The reason is that b has a declaration of x, so the $(x + 1)$ in the `printf` in b refers to this x. Thus, the value printed is 1.

After b finishes, and c is called, we again need to print the value of macro a. However, the only x accessible to c is the global x. The `printf` statement in c thus refers to this declaration of x, and value 2 is printed. □

Dynamic scope resolution is also essential for polymorphic procedures, those that have two or more definitions for the same name, depending only on the

Analogy Between Static and Dynamic Scoping

While there could be any number of static or dynamic policies for scoping, there is an interesting relationship between the normal (block-structured) static scoping rule and the normal dynamic policy. In a sense, the dynamic rule is to time as the static rule is to space. While the static rule asks us to find the declaration whose unit (block) most closely surrounds the physical location of the use, the dynamic rule asks us to find the declaration whose unit (procedure invocation) most closely surrounds the time of the use.

types of the arguments. In some languages, such as ML (see Section 7.3.3), it is possible to determine statically types for all uses of names, in which case the compiler can replace each use of a procedure name p by a reference to the code for the proper procedure. However, in other languages, such as Java and C++, there are times when the compiler cannot make that determination.

Example 1.8: A distinguishing feature of object-oriented programming is the ability of each object to invoke the appropriate method in response to a message. In other words, the procedure called when $x.m()$ is executed depends on the class of the object denoted by x at that time. A typical example is as follows:

1. There is a class C with a method named $m()$.
2. D is a subclass of C, and D has its own method named $m()$.
3. There is a use of m of the form $x.m()$, where x is an object of class C.

Normally, it is impossible to tell at compile time whether x will be of class C or of the subclass D. If the method application occurs several times, it is highly likely that some will be on objects denoted by x that are in class C but not D, while others will be in class D. It is not until run-time that it can be decided which definition of m is the right one. Thus, the code generated by the compiler must determine the class of the object x, and call one or the other method named m. □

1.6.6 Parameter Passing Mechanisms

All programming languages have a notion of a procedure, but they can differ in how these procedures get their arguments. In this section, we shall consider how the *actual parameters* (the parameters used in the call of a procedure) are associated with the *formal parameters* (those used in the procedure definition). Which mechanism is used determines how the calling-sequence code treats parameters. The great majority of languages use either "call-by-value," or "call-by-reference," or both. We shall explain these terms, and another method known as "call-by-name," that is primarily of historical interest.

Call-by-Value

In *call-by-value*, the actual parameter is evaluated (if it is an expression) or copied (if it is a variable). The value is placed in the location belonging to the corresponding formal parameter of the called procedure. This method is used in C and Java, and is a common option in C++, as well as in most other languages. Call-by-value has the effect that all computation involving the formal parameters done by the called procedure is local to that procedure, and the actual parameters themselves cannot be changed.

Note, however, that in C we can pass a pointer to a variable to allow that variable to be changed by the callee. Likewise, array names passed as parameters in C, C++, or Java give the called procedure what is in effect a pointer or reference to the array itself. Thus, if a is the name of an array of the calling procedure, and it is passed by value to corresponding formal parameter x, then an assignment such as `x[i] = 2` really changes the array element $a[2]$. The reason is that, although x gets a copy of the value of a, that value is really a pointer to the beginning of the area of the store where the array named a is located.

Similarly, in Java, many variables are really references, or pointers, to the things they stand for. This observation applies to arrays, strings, and objects of all classes. Even though Java uses call-by-value exclusively, whenever we pass the name of an object to a called procedure, the value received by that procedure is in effect a pointer to the object. Thus, the called procedure is able to affect the value of the object itself.

Call-by-Reference

In *call-by-reference*, the address of the actual parameter is passed to the callee as the value of the corresponding formal parameter. Uses of the formal parameter in the code of the callee are implemented by following this pointer to the location indicated by the caller. Changes to the formal parameter thus appear as changes to the actual parameter.

If the actual parameter is an expression, however, then the expression is evaluated before the call, and its value stored in a location of its own. Changes to the formal parameter change this location, but can have no effect on the data of the caller.

Call-by-reference is used for "ref" parameters in C++ and is an option in many other languages. It is almost essential when the formal parameter is a large object, array, or structure. The reason is that strict call-by-value requires that the caller copy the entire actual parameter into the space belonging to the corresponding formal parameter. This copying gets expensive when the parameter is large. As we noted when discussing call-by-value, languages such as Java solve the problem of passing arrays, strings, or other objects by copying only a reference to those objects. The effect is that Java behaves as if it used call-by-reference for anything other than a basic type such as an integer or real.

Call-by-Name

A third mechanism — call-by-name — was used in the early programming language Algol 60. It requires that the callee execute as if the actual parameter were substituted literally for the formal parameter in the code of the callee, as if the formal parameter were a macro standing for the actual parameter (with renaming of local names in the called procedure, to keep them distinct). When the actual parameter is an expression rather than a variable, some unintuitive behaviors occur, which is one reason this mechanism is not favored today.

1.6.7 Aliasing

There is an interesting consequence of call-by-reference parameter passing or its simulation, as in Java, where references to objects are passed by value. It is possible that two formal parameters can refer to the same location; such variables are said to be *aliases* of one another. As a result, any two variables, which may appear to take their values from two distinct formal parameters, can become aliases of each other, as well.

Example 1.9: Suppose a is an array belonging to a procedure p, and p calls another procedure $q(x, y)$ with a call $q(a, a)$. Suppose also that parameters are passed by value, but that array names are really references to the location where the array is stored, as in C or similar languages. Now, x and y have become aliases of each other. The important point is that if within q there is an assignment `x[10] = 2`, then the value of $y[10]$ also becomes 2. □

It turns out that understanding aliasing and the mechanisms that create it is essential if a compiler is to optimize a program. As we shall see starting in Chapter 9, there are many situations where we can only optimize code if we can be sure certain variables are not aliased. For instance, we might determine that `x = 2` is the only place that variable x is ever assigned. If so, then we can replace a use of x by a use of 2; for example, replace `a = x+3` by the simpler `a = 5`. But suppose there were another variable y that was aliased to x. Then an assignment `y = 4` might have the unexpected effect of changing x. It might also mean that replacing `a = x+3` by `a = 5` was a mistake; the proper value of a could be 7 there.

1.6.8 Exercises for Section 1.6

Exercise 1.6.1: For the block-structured C code of Fig. 1.13(a), indicate the values assigned to w, x, y, and z.

Exercise 1.6.2: Repeat Exercise 1.6.1 for the code of Fig. 1.13(b).

Exercise 1.6.3: For the block-structured code of Fig. 1.14, assuming the usual static scoping of declarations, give the scope for each of the twelve declarations.

```
int w, x, y, z;
int i = 4; int j = 5;
{   int j = 7;
    i = 6;
    w = i + j;
}
x = i + j;
{   int i = 8;
    y = i + j;
}
z = i + j;
```

(a) Code for Exercise 1.6.1

```
int w, x, y, z;
int i = 3; int j = 4;
{   int i = 5;
    w = i + j;
}
x = i + j;
{   int j = 6;
    i = 7;
    y = i + j;
}
z = i + j;
```

(b) Code for Exercise 1.6.2

Figure 1.13: Block-structured code

```
{   int w, x, y, z;         /* Block B1 */
    {   int x, z;           /* Block B2 */
        {   int w, x;       /* Block B3 */ }
    }
    {   int w, x;           /* Block B4 */
        {   int y, z;       /* Block B5 */ }
    }
}
```

Figure 1.14: Block structured code for Exercise 1.6.3

Exercise 1.6.4: What is printed by the following C code?

```
#define a (x+1)
int x = 2;
void b() { x = a; printf("%d\n", x); }
void c() { int x = 1; printf("%d\n"), a; }
void main() { b(); c(); }
```

1.7 Summary of Chapter 1

- *Language Processors.* An integrated software development environment includes many different kinds of language processors such as compilers, interpreters, assemblers, linkers, loaders, debuggers, profilers.

- *Compiler Phases.* A compiler operates as a sequence of phases, each of which transforms the source program from one intermediate representation to another.

- *Machine and Assembly Languages.* Machine languages were the first-generation programming languages, followed by assembly languages. Programming in these languages was time consuming and error prone.

- *Modeling in Compiler Design.* Compiler design is one of the places where theory has had the most impact on practice. Models that have been found useful include automata, grammars, regular expressions, trees, and many others.

- *Code Optimization.* Although code cannot truly be "optimized," the science of improving the efficiency of code is both complex and very important. It is a major portion of the study of compilation.

- *Higher-Level Languages.* As time goes on, programming languages take on progressively more of the tasks that formerly were left to the programmer, such as memory management, type-consistency checking, or parallel execution of code.

- *Compilers and Computer Architecture.* Compiler technology influences computer architecture, as well as being influenced by the advances in architecture. Many modern innovations in architecture depend on compilers being able to extract from source programs the opportunities to use the hardware capabilities effectively.

- *Software Productivity and Software Security.* The same technology that allows compilers to optimize code can be used for a variety of program-analysis tasks, ranging from detecting common program bugs to discovering that a program is vulnerable to one of the many kinds of intrusions that "hackers" have discovered.

- *Scope Rules.* The *scope* of a declaration of x is the context in which uses of x refer to this declaration. A language uses *static scope* or *lexical scope* if it is possible to determine the scope of a declaration by looking only at the program. Otherwise, the language uses *dynamic scope.*

- *Environments.* The association of names with locations in memory and then with values can be described in terms of *environments*, which map names to locations in store, and *states*, which map locations to their values.

- *Block Structure.* Languages that allow blocks to be nested are said to have *block structure.* A name x in a nested block B is in the scope of a declaration D of x in an enclosing block if there is no other declaration of x in an intervening block.

- *Parameter Passing.* Parameters are passed from a calling procedure to the callee either by value or by reference. When large objects are passed by value, the values passed are really references to the objects themselves, resulting in an effective call-by-reference.

- *Aliasing.* When parameters are (effectively) passed by reference, two formal parameters can refer to the same object. This possibility allows a change in one variable to change another.

1.8 References for Chapter 1

For the development of programming languages that were created and in use by 1967, including Fortran, Algol, Lisp, and Simula, see [7]. For languages that were created by 1982, including C, C++, Pascal, and Smalltalk, see [1].

The GNU Compiler Collection, gcc, is a popular source of open-source compilers for C, C++, Fortran, Java, and other languages [2]. Phoenix is a compiler-construction toolkit that provides an integrated framework for building the program analysis, code generation, and code optimization phases of compilers discussed in this book [3].

For more information about programming language concepts, we recommend [5,6]. For more on computer architecture and how it impacts compiling, we suggest [4].

1. Bergin, T. J. and R. G. Gibson, *History of Programming Languages*, ACM Press, New York, 1996.

2. `http://gcc.gnu.org/` .

3. `http://research.microsoft.com/phoenix/default.aspx` .

4. Hennessy, J. L. and D. A. Patterson, *Computer Organization and Design: The Hardware/Software Interface*, Morgan-Kaufmann, San Francisco, CA, 2004.

5. Scott, M. L., *Programming Language Pragmatics, second edition*, Morgan-Kaufmann, San Francisco, CA, 2006.

6. Sethi, R., *Programming Languages: Concepts and Constructs*, Addison-Wesley, 1996.

7. Wexelblat, R. L., *History of Programming Languages*, Academic Press, New York, 1981.

Chapter 3

Lexical Analysis

In this chapter we show how to construct a lexical analyzer. To implement a lexical analyzer by hand, it helps to start with a diagram or other description for the lexemes of each token. We can then write code to identify each occurrence of each lexeme on the input and to return information about the token identified.

We can also produce a lexical analyzer automatically by specifying the lexeme patterns to a *lexical-analyzer generator* and compiling those patterns into code that functions as a lexical analyzer. This approach makes it easier to modify a lexical analyzer, since we have only to rewrite the affected patterns, not the entire program. It also speeds up the process of implementing the lexical analyzer, since the programmer specifies the software at the very high level of patterns and relies on the generator to produce the detailed code. We shall introduce in Section 3.5 a lexical-analyzer generator called *Lex* (or *Flex* in a more recent embodiment).

We begin the study of lexical-analyzer generators by introducing regular expressions, a convenient notation for specifying lexeme patterns. We show how this notation can be transformed, first into nondeterministic automata and then into deterministic automata. The latter two notations can be used as input to a "driver," that is, code which simulates these automata and uses them as a guide to determining the next token. This driver and the specification of the automaton form the nucleus of the lexical analyzer.

3.1 The Role of the Lexical Analyzer

As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program. The stream of tokens is sent to the parser for syntax analysis. It is common for the lexical analyzer to interact with the symbol table as well. When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table. In some cases, information regarding the kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser.

These interactions are suggested in Fig. 3.1. Commonly, the interaction is implemented by having the parser call the lexical analyzer. The call, suggested by the *getNextToken* command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.

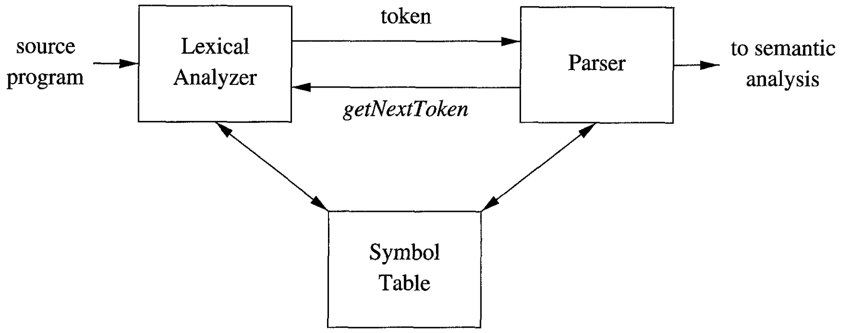


Figure 3.1: Interactions between the lexical analyzer and the parser

Since the lexical analyzer is the part of the compiler that reads the source text, it may perform certain other tasks besides identification of lexemes. One such task is stripping out comments and *whitespace* (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input). Another task is correlating error messages generated by the compiler with the source program. For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message. In some compilers, the lexical analyzer makes a copy of the source program with the error messages inserted at the appropriate positions. If the source program uses a macro-preprocessor, the expansion of macros may also be performed by the lexical analyzer.

Sometimes, lexical analyzers are divided into a cascade of two processes:

a) *Scanning* consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.

b) *Lexical analysis* proper is the more complex portion, where the scanner produces the sequence of tokens as output.

3.1.1 Lexical Analysis Versus Parsing

There are a number of reasons why the analysis portion of a compiler is normally separated into lexical analysis and parsing (syntax analysis) phases.

1. Simplicity of design is the most important consideration. The separation of lexical and syntactic analysis often allows us to simplify at least one of these tasks. For example, a parser that had to deal with comments and whitespace as syntactic units would be considerably more complex than one that can assume comments and whitespace have already been removed by the lexical analyzer. If we are designing a new language, separating lexical and syntactic concerns can lead to a cleaner overall language design.

2. Compiler efficiency is improved. A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.

3. Compiler portability is enhanced. Input-device-specific peculiarities can be restricted to the lexical analyzer.

3.1.2 Tokens, Patterns, and Lexemes

When discussing lexical analysis, we use three related but distinct terms:

- A *token* is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. In what follows, we shall generally write the name of a token in boldface. We will often refer to a token by its token name.

- A *pattern* is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is *matched* by many strings.

- A *lexeme* is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

Example 3.1: Figure 3.2 gives some typical tokens, their informally described patterns, and some sample lexemes. To see how these concepts are used in practice, in the C statement

```
printf("Total = %d\n", score);
```

both `printf` and `score` are lexemes matching the pattern for token **id**, and `"Total = %d\n"` is a lexeme matching **literal**. □

In many programming languages, the following classes cover most or all of the tokens:

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	< or > or <= or >= or == or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but `"`, surrounded by `"`'s	`"core dumped"`

Figure 3.2: Examples of tokens

1. One token for each keyword. The pattern for a keyword is the same as the keyword itself.
2. Tokens for the operators, either individually or in classes such as the token `comparison` mentioned in Fig. 3.2.
3. One token representing all identifiers.
4. One or more tokens representing constants, such as numbers and literal strings.
5. Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.

3.1.3 Attributes for Tokens

When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phases additional information about the particular lexeme that matched. For example, the pattern for token **number** matches both 0 and 1, but it is extremely important for the code generator to know which lexeme was found in the source program. Thus, in many cases the lexical analyzer returns to the parser not only a token name, but an attribute value that describes the lexeme represented by the token; the token name influences parsing decisions, while the attribute value influences translation of tokens after the parse.

We shall assume that tokens have at most one associated attribute, although this attribute may have a structure that combines several pieces of information. The most important example is the token **id**, where we need to associate with the token a great deal of information. Normally, information about an identifier — e.g., its lexeme, its type, and the location at which it is first found (in case an error message about that identifier must be issued) — is kept in the symbol table. Thus, the appropriate attribute value for an identifier is a pointer to the symbol-table entry for that identifier.

Tricky Problems When Recognizing Tokens

Usually, given the pattern describing the lexemes of a token, it is relatively simple to recognize matching lexemes when they occur on the input. However, in some languages it is not immediately apparent when we have seen an instance of a lexeme corresponding to a token. The following example is taken from Fortran, in the fixed-format still allowed in Fortran 90. In the statement

```
DO 5 I = 1.25
```

it is not apparent that the first lexeme is `DO5I`, an instance of the identifier token, until we see the dot following the 1. Note that blanks in fixed-format Fortran are ignored (an archaic convention). Had we seen a comma instead of the dot, we would have had a do-statement

```
DO 5 I = 1,25
```

in which the first lexeme is the keyword `DO`.

Example 3.2: The token names and associated attribute values for the Fortran statement

```
E = M * C ** 2
```

are written below as a sequence of pairs.

<**id**, pointer to symbol-table entry for `E`>
<**assign_op**>
<**id**, pointer to symbol-table entry for `M`>
<**mult_op**>
<**id**, pointer to symbol-table entry for `C`>
<**exp_op**>
<**number**, integer value 2>

Note that in certain pairs, especially operators, punctuation, and keywords, there is no need for an attribute value. In this example, the token **number** has been given an integer-valued attribute. In practice, a typical compiler would instead store a character string representing the constant and use as an attribute value for **number** a pointer to that string. □

3.1.4 Lexical Errors

It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error. For instance, if the string `fi` is encountered for the first time in a C program in the context:

```
    fi ( a == f(x)) ...
```

a lexical analyzer cannot tell whether `fi` is a misspelling of the keyword `if` or an undeclared function identifier. Since `fi` is a valid lexeme for the token **id**, the lexical analyzer must return the token **id** to the parser and let some other phase of the compiler — probably the parser in this case — handle an error due to transposition of the letters.

However, suppose a situation arises in which the lexical analyzer is unable to proceed because none of the patterns for tokens matches any prefix of the remaining input. The simplest recovery strategy is "panic mode" recovery. We delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left. This recovery technique may confuse the parser, but in an interactive computing environment it may be quite adequate.

Other possible error-recovery actions are:

1. Delete one character from the remaining input.

2. Insert a missing character into the remaining input.

3. Replace a character by another character.

4. Transpose two adjacent characters.

Transformations like these may be tried in an attempt to repair the input. The simplest such strategy is to see whether a prefix of the remaining input can be transformed into a valid lexeme by a single transformation. This strategy makes sense, since in practice most lexical errors involve a single character. A more general correction strategy is to find the smallest number of transformations needed to convert the source program into one that consists only of valid lexemes, but this approach is considered too expensive in practice to be worth the effort.

3.1.5 Exercises for Section 3.1

Exercise 3.1.1: Divide the following C++ program:

```
float limitedSquare(x) float x {
    /* returns x-squared, but never more than 100 */
    return (x<=-10.0||x>=10.0)?100:x*x;
}
```

into appropriate lexemes, using the discussion of Section 3.1.2 as a guide. Which lexemes should get associated lexical values? What should those values be?

! **Exercise 3.1.2:** Tagged languages like HTML or XML are different from conventional programming languages in that the punctuation (tags) are either very numerous (as in HTML) or a user-definable set (as in XML). Further, tags can often have parameters. Suggest how to divide the following HTML document:

```
Here is a photo of <B>my house</B>:
<P><IMG SRC = "house.gif"><BR>
See <A HREF = "morePix.html">More Pictures</A> if you
liked that one.<P>
```

into appropriate lexemes. Which lexemes should get associated lexical values, and what should those values be?

3.2 Input Buffering

Before discussing the problem of recognizing lexemes in the input, let us examine some ways that the simple but important task of reading the source program can be speeded. This task is made difficult by the fact that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme. The box on "Tricky Problems When Recognizing Tokens" in Section 3.1 gave an extreme example, but there are many situations where we need to look at least one additional character ahead. For instance, we cannot be sure we've seen the end of an identifier until we see a character that is not a letter or digit, and therefore is not part of the lexeme for **id**. In C, single-character operators like `-`, `=`, or `<` could also be the beginning of a two-character operator like `->`, `==`, or `<=`. Thus, we shall introduce a two-buffer scheme that handles large lookaheads safely. We then consider an improvement involving "sentinels" that saves time checking for the ends of buffers.

3.2.1 Buffer Pairs

Because of the amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character. An important scheme involves two buffers that are alternately reloaded, as suggested in Fig. 3.3.

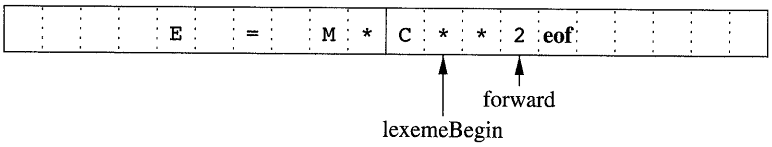


Figure 3.3: Using a pair of input buffers

Each buffer is of the same size N, and N is usually the size of a disk block, e.g., 4096 bytes. Using one system read command we can read N characters into a buffer, rather than using one system call per character. If fewer than N characters remain in the input file, then a special character, represented by **eof**, marks the end of the source file and is different from any possible character of the source program.

Two pointers to the input are maintained:

1. Pointer `lexemeBegin`, marks the beginning of the current lexeme, whose extent we are attempting to determine.

2. Pointer `forward` scans ahead until a pattern match is found; the exact strategy whereby this determination is made will be covered in the balance of this chapter.

Once the next lexeme is determined, `forward` is set to the character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, `lexemeBegin` is set to the character immediately after the lexeme just found. In Fig. 3.3, we see `forward` has passed the end of the next lexeme, `**` (the Fortran exponentiation operator), and must be retracted one position to its left.

Advancing `forward` requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move `forward` to the beginning of the newly loaded buffer. As long as we never need to look so far ahead of the actual lexeme that the sum of the lexeme's length plus the distance we look ahead is greater than N, we shall never overwrite the lexeme in its buffer before determining it.

3.2.2 Sentinels

If we use the scheme of Section 3.2.1 as described, we must check, each time we advance `forward`, that we have not moved off one of the buffers; if we do, then we must also reload the other buffer. Thus, for each character read, we make two tests: one for the end of the buffer, and one to determine what character is read (the latter may be a multiway branch). We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a *sentinel* character at the end. The sentinel is a special character that cannot be part of the source program, and a natural choice is the character **eof**.

Figure 3.4 shows the same arrangement as Fig. 3.3, but with the sentinels added. Note that **eof** retains its use as a marker for the end of the entire input. Any **eof** that appears other than at the end of a buffer means that the input is at an end. Figure 3.5 summarizes the algorithm for advancing `forward`. Notice how the first test, which can be part of a multiway branch based on the character pointed to by `forward`, is the only test we make, except in the case where we actually are at the end of a buffer or the end of the input.

3.3 Specification of Tokens

Regular expressions are an important notation for specifying lexeme patterns. While they cannot express all possible patterns, they are very effective in spec-

Can We Run Out of Buffer Space?

In most modern languages, lexemes are short, and one or two characters of lookahead is sufficient. Thus a buffer size N in the thousands is ample, and the double-buffer scheme of Section 3.2.1 works without problem. However, there are some risks. For example, if character strings can be very long, extending over many lines, then we could face the possibility that a lexeme is longer than N. To avoid problems with long character strings, we can treat them as a concatenation of components, one from each line over which the string is written. For instance, in Java it is conventional to represent long strings by writing a piece on each line and concatenating pieces with a + operator at the end of each piece.

A more difficult problem occurs when arbitrarily long lookahead may be needed. For example, some languages like PL/I do not treat keywords as *reserved*; that is, you can use identifiers with the same name as a keyword like DECLARE. If the lexical analyzer is presented with text of a PL/I program that begins DECLARE (ARG1, ARG2,... it cannot be sure whether DECLARE is a keyword, and ARG1 and so on are variables being declared, or whether DECLARE is a procedure name with its arguments. For this reason, modern languages tend to reserve their keywords. However, if not, one can treat a keyword like DECLARE as an ambiguous identifier, and let the parser resolve the issue, perhaps in conjunction with symbol-table lookup.

ifying those types of patterns that we actually need for tokens. In this section we shall study the formal notation for regular expressions, and in Section 3.5 we shall see how these expressions are used in a lexical-analyzer generator. Then, Section 3.7 shows how to build the lexical analyzer by converting regular expressions to automata that perform the recognition of the specified tokens.

3.3.1 Strings and Languages

An *alphabet* is any finite set of symbols. Typical examples of symbols are letters, digits, and punctuation. The set $\{0, 1\}$ is the *binary alphabet*. ASCII is an important example of an alphabet; it is used in many software systems. Uni-

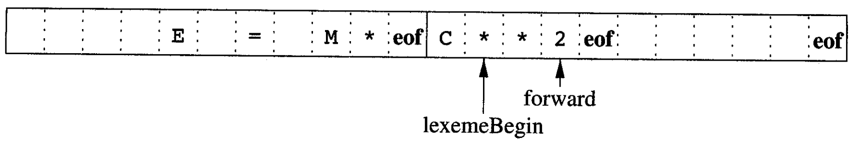


Figure 3.4: Sentinels at the end of each buffer

```
switch ( *forward++ ) {
    case eof:
        if (forward is at end of first buffer ) {
            reload second buffer;
            forward = beginning of second buffer;
        }
        else if (forward is at end of second buffer ) {
            reload first buffer;
            forward = beginning of first buffer;
        }
        else /* eof within a buffer marks the end of input */
            terminate lexical analysis;
        break;
    Cases for the other characters
}
```

Figure 3.5: Lookahead code with sentinels

Implementing Multiway Branches

We might imagine that the switch in Fig. 3.5 requires many steps to execute, and that placing the case **eof** first is not a wise choice. Actually, it doesn't matter in what order we list the cases for each character. In practice, a multiway branch depending on the input character is be made in one step by jumping to an address found in an array of addresses, indexed by characters.

code, which includes approximately 100,000 characters from alphabets around the world, is another important example of an alphabet.

A *string* over an alphabet is a finite sequence of symbols drawn from that alphabet. In language theory, the terms "sentence" and "word" are often used as synonyms for "string." The length of a string s, usually written $|s|$, is the number of occurrences of symbols in s. For example, banana is a string of length six. The *empty string*, denoted ϵ, is the string of length zero.

A *language* is any countable set of strings over some fixed alphabet. This definition is very broad. Abstract languages like $\emptyset$, the *empty set*, or $\{\epsilon\}$, the set containing only the empty string, are languages under this definition. So too are the set of all syntactically well-formed C programs and the set of all grammatically correct English sentences, although the latter two languages are difficult to specify exactly. Note that the definition of "language" does not require that any meaning be ascribed to the strings in the language. Methods for defining the "meaning" of strings are discussed in Chapter 5.

Terms for Parts of Strings

The following string-related terms are commonly used:

1. A *prefix* of string s is any string obtained by removing zero or more symbols from the end of s. For example, **ban**, **banana**, and ϵ are prefixes of **banana**.

2. A *suffix* of string s is any string obtained by removing zero or more symbols from the beginning of s. For example, **nana**, **banana**, and ϵ are suffixes of **banana**.

3. A *substring* of s is obtained by deleting any prefix and any suffix from s. For instance, **banana**, **nan**, and ϵ are substrings of **banana**.

4. The *proper* prefixes, suffixes, and substrings of a string s are those, prefixes, suffixes, and substrings, respectively, of s that are not ϵ or not equal to s itself.

5. A *subsequence* of s is any string formed by deleting zero or more not necessarily consecutive positions of s. For example, **baan** is a subsequence of **banana**.

If x and y are strings, then the *concatenation* of x and y, denoted xy, is the string formed by appending y to x. For example, if x = **dog** and y = **house**, then xy = **doghouse**. The empty string is the identity under concatenation; that is, for any string s, $\epsilon s = s\epsilon = s$.

If we think of concatenation as a product, we can define the "exponentiation" of strings as follows. Define s^0 to be ϵ, and for all $i > 0$, define s^i to be $s^{i-1}s$. Since $\epsilon s = s$, it follows that $s^1 = s$. Then $s^2 = ss$, $s^3 = sss$, and so on.

3.3.2 Operations on Languages

In lexical analysis, the most important operations on languages are union, concatenation, and closure, which are defined formally in Fig. 3.6. Union is the familiar operation on sets. The concatenation of languages is all strings formed by taking a string from the first language and a string from the second language, in all possible ways, and concatenating them. The (*Kleene*) *closure* of a language L, denoted L^*, is the set of strings you get by concatenating L zero or more times. Note that L^0, the "concatenation of L zero times," is defined to be $\{\epsilon\}$, and inductively, L^i is $L^{i-1}L$. Finally, the positive closure, denoted L^+, is the same as the Kleene closure, but without the term L^0. That is, ϵ will not be in L^+ unless it is in L itself.

OPERATION	DEFINITION AND NOTATION
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

Figure 3.6: Definitions of operations on languages

Example 3.3: Let L be the set of letters $\{\texttt{A}, \texttt{B}, \ldots, \texttt{Z}, \texttt{a}, \texttt{b}, \ldots, \texttt{z}\}$ and let D be the set of digits $\{\texttt{0}, \texttt{1}, \ldots \texttt{9}\}$. We may think of L and D in two, essentially equivalent, ways. One way is that L and D are, respectively, the alphabets of uppercase and lowercase letters and of digits. The second way is that L and D are languages, all of whose strings happen to be of length one. Here are some other languages that can be constructed from languages L and D, using the operators of Fig. 3.6:

1. $L \cup D$ is the set of letters and digits — strictly speaking the language with 62 strings of length one, each of which strings is either one letter or one digit.

2. LD is the set of 520 strings of length two, each consisting of one letter followed by one digit.

3. L^4 is the set of all 4-letter strings.

4. L^* is the set of all strings of letters, including ϵ, the empty string.

5. $L(L \cup D)^*$ is the set of all strings of letters and digits beginning with a letter.

6. D^+ is the set of all strings of one or more digits.

□

3.3.3 Regular Expressions

Suppose we wanted to describe the set of valid C identifiers. It is almost exactly the language described in item (5) above; the only difference is that the underscore is included among the letters.

In Example 3.3, we were able to describe identifiers by giving names to sets of letters and digits and using the language operators union, concatenation, and closure. This process is so useful that a notation called *regular expressions* has come into common use for describing all the languages that can be built from these operators applied to the symbols of some alphabet. In this notation, if *letter_* is established to stand for any letter or the underscore, and *digit_* is established to stand for any digit, then we could describe the language of C identifiers by:

$$\mathit{letter}\_ \; (\; \mathit{letter}\_ \mid \mathit{digit} \;)^*$$

The vertical bar above means union, the parentheses are used to group subexpressions, the star means "zero or more occurrences of," and the juxtaposition of *letter_* with the remainder of the expression signifies concatenation.

The regular expressions are built recursively out of smaller regular expressions, using the rules described below. Each regular expression r denotes a language $L(r)$, which is also defined recursively from the languages denoted by r's subexpressions. Here are the rules that define the regular expressions over some alphabet Σ and the languages that those expressions denote.

BASIS: There are two rules that form the basis:

1. ϵ is a regular expression, and $L(\epsilon)$ is $\{\epsilon\}$, that is, the language whose sole member is the empty string.

2. If a is a symbol in Σ, then $\mathbf{a}$ is a regular expression, and $L(\mathbf{a}) = \{a\}$, that is, the language with one string, of length one, with a in its one position. Note that by convention, we use italics for symbols, and boldface for their corresponding regular expression.[1]

INDUCTION: There are four parts to the induction whereby larger regular expressions are built from smaller ones. Suppose r and s are regular expressions denoting languages $L(r)$ and $L(s)$, respectively.

1. $(r)|(s)$ is a regular expression denoting the language $L(r) \cup L(s)$.

2. $(r)(s)$ is a regular expression denoting the language $L(r)L(s)$.

3. $(r)^*$ is a regular expression denoting $\big(L(r)\big)^*$.

4. (r) is a regular expression denoting $L(r)$. This last rule says that we can add additional pairs of parentheses around expressions without changing the language they denote.

As defined, regular expressions often contain unnecessary pairs of parentheses. We may drop certain pairs of parentheses if we adopt the conventions that:

a) The unary operator $*$ has highest precedence and is left associative.

b) Concatenation has second highest precedence and is left associative.

[1]However, when talking about specific characters from the ASCII character set, we shall generally use teletype font for both the character and its regular expression.

c) $|$ has lowest precedence and is left associative.

Under these conventions, for example, we may replace the regular expression $(\mathbf{a})|((\mathbf{b})^*(\mathbf{c}))$ by $\mathbf{a}|\mathbf{b}^*\mathbf{c}$. Both expressions denote the set of strings that are either a single a or are zero or more b's followed by one c.

Example 3.4: Let $\Sigma = \{a, b\}$.

1. The regular expression $\mathbf{a}|\mathbf{b}$ denotes the language $\{a, b\}$.

2. $(\mathbf{a}|\mathbf{b})(\mathbf{a}|\mathbf{b})$ denotes $\{aa, ab, ba, bb\}$, the language of all strings of length two over the alphabet Σ. Another regular expression for the same language is $\mathbf{aa}|\mathbf{ab}|\mathbf{ba}|\mathbf{bb}$.

3. $\mathbf{a}^*$ denotes the language consisting of all strings of zero or more a's, that is, $\{\epsilon, a, aa, aaa, \dots\}$.

4. $(\mathbf{a}|\mathbf{b})^*$ denotes the set of all strings consisting of zero or more instances of a or b, that is, all strings of a's and b's: $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$. Another regular expression for the same language is $(\mathbf{a}^*\mathbf{b}^*)^*$.

5. $\mathbf{a}|\mathbf{a}^*\mathbf{b}$ denotes the language $\{a, b, ab, aab, aaab, \dots\}$, that is, the string a and all strings consisting of zero or more a's and ending in b.

□

A language that can be defined by a regular expression is called a *regular set*. If two regular expressions r and s denote the same regular set, we say they are *equivalent* and write $r = s$. For instance, $(\mathbf{a}|\mathbf{b}) = (\mathbf{b}|\mathbf{a})$. There are a number of algebraic laws for regular expressions; each law asserts that expressions of two different forms are equivalent. Figure 3.7 shows some of the algebraic laws that hold for arbitrary regular expressions r, s, and t.

LAW	DESCRIPTION
$r\|s = s\|r$	$\|$ is commutative
$r\|(s\|t) = (r\|s)\|t$	$\|$ is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s\|t) = rs\|rt$; $(s\|t)r = sr\|tr$	Concatenation distributes over $\|$
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
$r^* = (r\|\epsilon)^*$	ϵ is guaranteed in a closure
$r^{**} = r^*$	$*$ is idempotent

Figure 3.7: Algebraic laws for regular expressions

3.3.4 Regular Definitions

For notational convenience, we may wish to give names to certain regular expressions and use those names in subsequent expressions, as if the names were themselves symbols. If Σ is an alphabet of basic symbols, then a *regular definition* is a sequence of definitions of the form:

$$\begin{array}{ccc} d_1 & \rightarrow & r_1 \\ d_2 & \rightarrow & r_2 \\ & \cdots & \\ d_n & \rightarrow & r_n \end{array}$$

where:

1. Each d_i is a new symbol, not in Σ and not the same as any other of the d's, and
2. Each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$.

By restricting r_i to Σ and the previously defined d's, we avoid recursive definitions, and we can construct a regular expression over Σ alone, for each r_i. We do so by first replacing uses of d_1 in r_2 (which cannot use any of the d's except for d_1), then replacing uses of d_1 and d_2 in r_3 by r_1 and (the substituted) r_2, and so on. Finally, in r_n we replace each d_i, for $i = 1, 2, \ldots, n-1$, by the substituted version of r_i, each of which has only symbols of Σ.

Example 3.5: C identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifiers. We shall conventionally use italics for the symbols defined in regular definitions.

$$\begin{array}{rcl} \mathit{letter\_} & \rightarrow & \mathtt{A} \mid \mathtt{B} \mid \cdots \mid \mathtt{Z} \mid \mathtt{a} \mid \mathtt{b} \mid \cdots \mid \mathtt{z} \mid \mathtt{\_} \\ \mathit{digit} & \rightarrow & \mathtt{0} \mid \mathtt{1} \mid \cdots \mid \mathtt{9} \\ \mathit{id} & \rightarrow & \mathit{letter\_} \; (\; \mathit{letter\_} \mid \mathit{digit} \;)^* \end{array}$$

□

Example 3.6: Unsigned numbers (integer or floating point) are strings such as `5280`, `0.01234`, `6.336E4`, or `1.89E-4`. The regular definition

$$\begin{array}{rcl} \mathit{digit} & \rightarrow & \mathtt{0} \mid \mathtt{1} \mid \cdots \mid \mathtt{9} \\ \mathit{digits} & \rightarrow & \mathit{digit} \; \mathit{digit}^* \\ \mathit{optionalFraction} & \rightarrow & \mathtt{.} \; \mathit{digits} \mid \epsilon \\ \mathit{optionalExponent} & \rightarrow & (\; \mathtt{E} \; (\; \mathtt{+} \mid \mathtt{-} \mid \epsilon \;) \; \mathit{digits} \;) \mid \epsilon \\ \mathit{number} & \rightarrow & \mathit{digits} \; \mathit{optionalFraction} \; \mathit{optionalExponent} \end{array}$$

is a precise specification for this set of strings. That is, an *optionalFraction* is either a decimal point (dot) followed by one or more digits, or it is missing (the empty string). An *optionalExponent*, if not missing, is the letter E followed by an optional + or − sign, followed by one or more digits. Note that at least one digit must follow the dot, so *number* does not match `1.`, but does match `1.0`.
□

3.3.5 Extensions of Regular Expressions

Since Kleene introduced regular expressions with the basic operators for union, concatenation, and Kleene closure in the 1950s, many extensions have been added to regular expressions to enhance their ability to specify string patterns. Here we mention a few notational extensions that were first incorporated into Unix utilities such as Lex that are particularly useful in the specification lexical analyzers. The references to this chapter contain a discussion of some regular-expression variants in use today.

1. *One or more instances.* The unary, postfix operator $^+$ represents the positive closure of a regular expression and its language. That is, if r is a regular expression, then $(r)^+$ denotes the language $\big(L(r)\big)^+$. The operator $^+$ has the same precedence and associativity as the operator $*$. Two useful algebraic laws, $r^* = r^+|\epsilon$ and $r^+ = rr^* = r^*r$ relate the Kleene closure and positive closure.

2. *Zero or one instance.* The unary postfix operator ? means "zero or one occurrence." That is, $r?$ is equivalent to $r|\epsilon$, or put another way, $L(r?) = L(r) \cup \{\epsilon\}$. The ? operator has the same precedence and associativity as $*$ and $+$.

3. *Character classes.* A regular expression $a_1|a_2|\cdots|a_n$, where the a_i's are each symbols of the alphabet, can be replaced by the shorthand $[a_1a_2\cdots a_n]$. More importantly, when $a_1, a_2, \ldots, a_n$ form a logical sequence, e.g., consecutive uppercase letters, lowercase letters, or digits, we can replace them by a_1-a_n, that is, just the first and last separated by a hyphen. Thus, [**abc**] is shorthand for **a**|**b**|**c**, and [**a-z**] is shorthand for **a**|**b**|$\cdots$|**z**.

Example 3.7: Using these shorthands, we can rewrite the regular definition of Example 3.5 as:

$$\begin{array}{rcl} \mathit{letter\_} & \rightarrow & \texttt{[A-Za-z\_]} \\ \mathit{digit} & \rightarrow & \texttt{[0-9]} \\ \mathit{id} & \rightarrow & \mathit{letter\_}\ (\ \mathit{letter}\ |\ \mathit{digit}\ )^* \end{array}$$

The regular definition of Example 3.6 can also be simplified:

$$\begin{array}{rcl} \mathit{digit} & \rightarrow & \texttt{[0-9]} \\ \mathit{digits} & \rightarrow & \mathit{digit}^+ \\ \mathit{number} & \rightarrow & \mathit{digits}\ (\texttt{.}\ \mathit{digits})?\ (\ \texttt{E}\ \texttt{[+-]}?\ \mathit{digits}\ )? \end{array}$$

□

3.3.6 Exercises for Section 3.3

Exercise 3.3.1: Consult the language reference manuals to determine (*i*) the sets of characters that form the input alphabet (excluding those that may only appear in character strings or comments), (*ii*) the lexical form of numerical constants, and (*iii*) the lexical form of identifiers, for each of the following languages: (a) C (b) C++ (c) C# (d) Fortran (e) Java (f) Lisp (g) SQL.

! **Exercise 3.3.2:** Describe the languages denoted by the following regular expressions:

a) $\mathbf{a}(\mathbf{a}|\mathbf{b})^*\mathbf{a}$.

b) $((\epsilon|\mathbf{a})\mathbf{b}^*)^*$.

c) $(\mathbf{a}|\mathbf{b})^*\mathbf{a}(\mathbf{a}|\mathbf{b})(\mathbf{a}|\mathbf{b})$.

d) $\mathbf{a}^*\mathbf{b}\mathbf{a}^*\mathbf{b}\mathbf{a}^*\mathbf{b}\mathbf{a}^*$.

!! e) $(\mathbf{aa}|\mathbf{bb})^*((\mathbf{ab}|\mathbf{ba})(\mathbf{aa}|\mathbf{bb})^*(\mathbf{ab}|\mathbf{ba})(\mathbf{aa}|\mathbf{bb})^*)^*$.

Exercise 3.3.3: In a string of length n, how many of the following are there?

a) Prefixes.

b) Suffixes.

c) Proper prefixes.

! d) Substrings.

! e) Subsequences.

Exercise 3.3.4: Most languages are *case sensitive*, so keywords can be written only one way, and the regular expressions describing their lexeme is very simple. However, some languages, like SQL, are *case insensitive*, so a keyword can be written either in lowercase or in uppercase, or in any mixture of cases. Thus, the SQL keyword `SELECT` can also be written `select`, `Select`, or `sElEcT`, for instance. Show how to write a regular expression for a keyword in a case-insensitive language. Illustrate the idea by writing the expression for "select" in SQL.

! **Exercise 3.3.5:** Write regular definitions for the following languages:

a) All strings of lowercase letters that contain the five vowels in order.

b) All strings of lowercase letters in which the letters are in ascending lexicographic order.

c) Comments, consisting of a string surrounded by `/*` and `*/`, without an intervening `*/`, unless it is inside double-quotes (`"`).

!! d) All strings of digits with no repeated digits. *Hint*: Try this problem first with a few digits, such as $\{0, 1, 2\}$.

!! e) All strings of digits with at most one repeated digit.

!! f) All strings of a's and b's with an even number of a's and an odd number of b's.

g) The set of Chess moves, in the informal notation, such as `p-k4` or `kbp`$\times$`qn`.

!! h) All strings of a's and b's that do not contain the substring abb.

i) All strings of a's and b's that do not contain the subsequence abb.

Exercise 3.3.6: Write character classes for the following sets of characters:

a) The first ten letters (up to "j") in either upper or lower case.

b) The lowercase consonants.

c) The "digits" in a hexadecimal number (choose either upper or lower case for the "digits" above 9).

d) The characters that can appear at the end of a legitimate English sentence (e.g., exclamation point).

The following exercises, up to and including Exercise 3.3.10, discuss the extended regular-expression notation from `Lex` (the lexical-analyzer generator that we shall discuss extensively in Section 3.5). The extended notation is listed in Fig. 3.8.

Exercise 3.3.7: Note that these regular expressions give all of the following symbols (*operator characters*) a special meaning:

```
\ " . ^ $ [ ] * + ? { } | /
```

Their special meaning must be turned off if they are needed to represent themselves in a character string. We can do so by quoting the character within a string of length one or more; e.g., the regular expression `"**"` matches the string `**`. We can also get the literal meaning of an operator character by preceding it by a backslash. Thus, the regular expression `\*\*` also matches the string `**`. Write a regular expression that matches the string `"\`.

Exercise 3.3.8: In `Lex`, a *complemented character class* represents any character except the ones listed in the character class. We denote a complemented class by using ^ as the first character; this symbol (caret) is not itself part of the class being complemented, unless it is listed within the class itself. Thus, `[^A-Za-z]` matches any character that is not an uppercase or lowercase letter, and `[^\^]` represents any character but the caret (or newline, since newline cannot be in any character class). Show that for every regular expression with complemented character classes, there is an equivalent regular expression without complemented character classes.

EXPRESSION	MATCHES	EXAMPLE
c	the one non-operator character c	`a`
`\`c	character c literally	`\*`
`"`s`"`	string s literally	`"**"`
`.`	any character but newline	`a.*b`
`^`	beginning of a line	`^abc`
`$`	end of a line	`abc$`
`[`s`]`	any one of the characters in string s	`[abc]`
`[^`s`]`	any one character not in string s	`[^abc]`
$r*$	zero or more strings matching r	`a*`
$r+$	one or more strings matching r	`a+`
$r?$	zero or one r	`a?`
$r\{m, n\}$	between m and n occurrences of r	`a[1,5]`
$r_1 r_2$	an r_1 followed by an r_2	`ab`
$r_1 \mid r_2$	an r_1 or an r_2	`a|b`
(r)	same as r	`(a|b)`
r_1/r_2	r_1 when followed by r_2	`abc/123`

Figure 3.8: `Lex` regular expressions

! **Exercise 3.3.9:** The regular expression $r\{m, n\}$ matches from m to n occurrences of the pattern r. For example, `a[1,5]` matches a string of one to five a's. Show that for every regular expression containing repetition operators of this form, there is an equivalent regular expression without repetition operators.

! **Exercise 3.3.10:** The operator ^ matches the left end of a line, and $ matches the right end of a line. The operator ^ is also used to introduce complemented character classes, but the context always makes it clear which meaning is intended. For example, `^[^aeiou]*$` matches any complete line that does not contain a lowercase vowel.

a) How do you tell which meaning of ^ is intended?

b) Can you always replace a regular expression using the ^ and $ operators by an equivalent expression that does not use either of these operators?

! **Exercise 3.3.11:** The UNIX shell command `sh` uses the operators in Fig. 3.9 in filename expressions to describe sets of file names. For example, the filename expression `*.o` matches all file names ending in `.o`; `sort1.?` matches all filenames of the form `sort.`c, where c is any character. Show how `sh` filename

EXPRESSION	MATCHES	EXAMPLE
$'s'$	string s literally	`'\'`
$\backslash c$	character c literally	`\'`
$*$	any string	`*.o`
?	any character	`sort1.?`
$[s]$	any character in s	`sort1.[cso]`

Figure 3.9: Filename expressions used by the shell command `sh`

expressions can be replaced by equivalent regular expressions using only the basic union, concatenation, and closure operators.

! **Exercise 3.3.12:** SQL allows a rudimentary form of patterns in which two characters have special meaning: underscore (_) stands for any one character and percent-sign (%) stands for any string of 0 or more characters. In addition, the programmer may define any character, say e, to be the escape character, so e preceding an e preceding _, %, or another e gives the character that follows its literal meaning. Show how to express any SQL pattern as a regular expression, given that we know which character is the escape character.

3.4 Recognition of Tokens

In the previous section we learned how to express patterns using regular expressions. Now, we must study how to take the patterns for all the needed tokens and build a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns. Our discussion will make use of the following running example.

$$\begin{array}{rcl}
\mathit{stmt} & \rightarrow & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \\
 & | & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \textbf{ else } \mathit{stmt} \\
 & | & \epsilon \\
\mathit{expr} & \rightarrow & \mathit{term} \textbf{ relop } \mathit{term} \\
 & | & \mathit{term} \\
\mathit{term} & \rightarrow & \textbf{id} \\
 & | & \textbf{number}
\end{array}$$

Figure 3.10: A grammar for branching statements

Example 3.8: The grammar fragment of Fig. 3.10 describes a simple form of branching statements and conditional expressions. This syntax is similar to that of the language Pascal, in that **then** appears explicitly after conditions.

For **relop**, we use the comparison operators of languages like Pascal or SQL, where `=` is "equals" and `<>` is "not equals," because it presents an interesting structure of lexemes.

The terminals of the grammar, which are **if**, **then**, **else**, **relop**, **id**, and **number**, are the names of tokens as far as the lexical analyzer is concerned. The patterns for these tokens are described using regular definitions, as in Fig. 3.11. The patterns for *id* and *number* are similar to what we saw in Example 3.7.

$$\begin{array}{rcl}
\mathit{digit} & \to & \texttt{[0-9]} \\
\mathit{digits} & \to & \mathit{digit}^+ \\
\mathit{number} & \to & \mathit{digits}\ (\texttt{.}\ \mathit{digits})?\ (\ \texttt{E}\ \texttt{[+-]}?\ \mathit{digits}\ )? \\
\mathit{letter} & \to & \texttt{[A-Za-z]} \\
\mathit{id} & \to & \mathit{letter}\ (\ \mathit{letter} \mid \mathit{digit}\ )^* \\
\mathit{if} & \to & \texttt{if} \\
\mathit{then} & \to & \texttt{then} \\
\mathit{else} & \to & \texttt{else} \\
\mathit{relop} & \to & \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{=} \mid \texttt{<>}
\end{array}$$

Figure 3.11: Patterns for tokens of Example 3.8

For this language, the lexical analyzer will recognize the keywords `if`, `then`, and `else`, as well as lexemes that match the patterns for *relop*, *id*, and *number*. To simplify matters, we make the common assumption that keywords are also *reserved words*: that is, they are not identifiers, even though their lexemes match the pattern for identifiers.

In addition, we assign the lexical analyzer the job of stripping out white-space, by recognizing the "token" *ws* defined by:

$$\mathit{ws} \to (\ \textbf{blank} \mid \textbf{tab} \mid \textbf{newline}\ )^+$$

Here, **blank**, **tab**, and **newline** are abstract symbols that we use to express the ASCII characters of the same names. Token *ws* is different from the other tokens in that, when we recognize it, we do not return it to the parser, but rather restart the lexical analysis from the character that follows the whitespace. It is the following token that gets returned to the parser.

Our goal for the lexical analyzer is summarized in Fig. 3.12. That table shows, for each lexeme or family of lexemes, which token name is returned to the parser and what attribute value, as discussed in Section 3.1.3, is returned. Note that for the six relational operators, symbolic constants `LT`, `LE`, and so on are used as the attribute value, in order to indicate which instance of the token **relop** we have found. The particular operator found will influence the code that is output from the compiler. □

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
`<`	**relop**	`LT`
`<=`	**relop**	`LE`
`=`	**relop**	`EQ`
`<>`	**relop**	`NE`
`>`	**relop**	`GT`
`>=`	**relop**	`GE`

Figure 3.12: Tokens, their patterns, and attribute values

3.4.1 Transition Diagrams

As an intermediate step in the construction of a lexical analyzer, we first convert patterns into stylized flowcharts, called "transition diagrams." In this section, we perform the conversion from regular-expression patterns to transition diagrams by hand, but in Section 3.6, we shall see that there is a mechanical way to construct these diagrams from collections of regular expressions.

Transition diagrams have a collection of nodes or circles, called *states*. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns. We may think of a state as summarizing all we need to know about what characters we have seen between the *lexemeBegin* pointer and the *forward* pointer (as in the situation of Fig. 3.3).

Edges are directed from one state of the transition diagram to another. Each edge is *labeled* by a symbol or set of symbols. If we are in some state s, and the next input symbol is a, we look for an edge out of state s labeled by a (and perhaps by other symbols, as well). If we find such an edge, we advance the *forward* pointer and enter the state of the transition diagram to which that edge leads. We shall assume that all our transition diagrams are *deterministic*, meaning that there is never more than one edge out of a given state with a given symbol among its labels. Starting in Section 3.5, we shall relax the condition of determinism, making life much easier for the designer of a lexical analyzer, although trickier for the implementer. Some important conventions about transition diagrams are:

1. Certain states are said to be *accepting*, or *final*. These states indicate that a lexeme has been found, although the actual lexeme may not consist of all positions between the *lexemeBegin* and *forward* pointers. We always

indicate an accepting state by a double circle, and if there is an action to be taken — typically returning a token and an attribute value to the parser — we shall attach that action to the accepting state.

2. In addition, if it is necessary to retract the *forward* pointer one position (i.e., the lexeme does not include the symbol that got us to the accepting state), then we shall additionally place a * near that accepting state. In our example, it is never necessary to retract *forward* by more than one position, but if it were, we could attach any number of *'s to the accepting state.

3. One state is designated the *start state*, or *initial state*; it is indicated by an edge, labeled "start," entering from nowhere. The transition diagram always begins in the start state before any input symbols have been read.

Example 3.9: Figure 3.13 is a transition diagram that recognizes the lexemes matching the token **relop**. We begin in state 0, the start state. If we see < as the first input symbol, then among the lexemes that match the pattern for **relop** we can only be looking at <, <>, or <=. We therefore go to state 1, and look at the next character. If it is =, then we recognize lexeme <=, enter state 2, and return the token **relop** with attribute LE, the symbolic constant representing this particular comparison operator. If in state 1 the next character is >, then instead we have lexeme <>, and enter state 3 to return an indication that the not-equals operator has been found. On any other character, the lexeme is <, and we enter state 4 to return that information. Note, however, that state 4 has a * to indicate that we must retract the input one position.

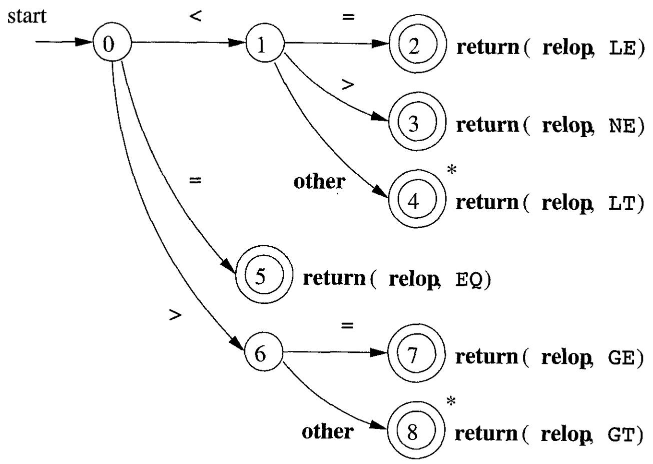


Figure 3.13: Transition diagram for **relop**

On the other hand, if in state 0 the first character we see is =, then this one character must be the lexeme. We immediately return that fact from state 5.

The remaining possibility is that the first character is `>`. Then, we must enter state 6 and decide, on the basis of the next character, whether the lexeme is `>=` (if we next see the `=` sign), or just `>` (on any other character). Note that if, in state 0, we see any character besides `<`, `=`, or `>`, we can not possibly be seeing a `relop` lexeme, so this transition diagram will not be used. □

3.4.2 Recognition of Reserved Words and Identifiers

Recognizing keywords and identifiers presents a problem. Usually, keywords like `if` or `then` are reserved (as they are in our running example), so they are not identifiers even though they *look* like identifiers. Thus, although we typically use a transition diagram like that of Fig. 3.14 to search for identifier lexemes, this diagram will also recognize the keywords `if`, `then`, and `else` of our running example.

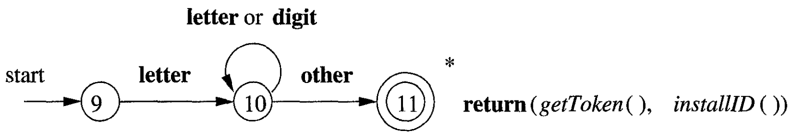


Figure 3.14: A transition diagram for **id**'s and keywords

There are two ways that we can handle reserved words that look like identifiers:

1. Install the reserved words in the symbol table initially. A field of the symbol-table entry indicates that these strings are never ordinary identifiers, and tells which token they represent. We have supposed that this method is in use in Fig. 3.14. When we find an identifier, a call to *installID* places it in the symbol table if it is not already there and returns a pointer to the symbol-table entry for the lexeme found. Of course, any identifier not in the symbol table during lexical analysis cannot be a reserved word, so its token is **id**. The function *getToken* examines the symbol table entry for the lexeme found, and returns whatever token name the symbol table says this lexeme represents — either **id** or one of the keyword tokens that was initially installed in the table.

2. Create separate transition diagrams for each keyword; an example for the keyword `then` is shown in Fig. 3.15. Note that such a transition diagram consists of states representing the situation after each successive letter of the keyword is seen, followed by a test for a "nonletter-or-digit," i.e., any character that cannot be the continuation of an identifier. It is necessary to check that the identifier has ended, or else we would return token **then** in situations where the correct token was **id**, with a lexeme like `thenextvalue` that has `then` as a proper prefix. If we adopt this approach, then we must prioritize the tokens so that the reserved-word

tokens are recognized in preference to **id**, when the lexeme matches both patterns. We *do not* use this approach in our example, which is why the states in Fig. 3.15 are unnumbered.

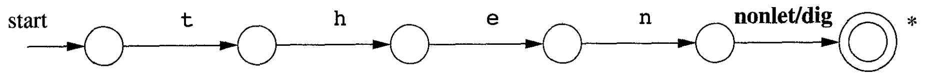


Figure 3.15: Hypothetical transition diagram for the keyword `then`

3.4.3 Completion of the Running Example

The transition diagram for **id**'s that we saw in Fig. 3.14 has a simple structure. Starting in state 9, it checks that the lexeme begins with a letter and goes to state 10 if so. We stay in state 10 as long as the input contains letters and digits. When we first encounter anything but a letter or digit, we go to state 11 and accept the lexeme found. Since the last character is not part of the identifier, we must retract the input one position, and as discussed in Section 3.4.2, we enter what we have found in the symbol table and determine whether we have a keyword or a true identifier.

The transition diagram for token **number** is shown in Fig. 3.16, and is so far the most complex diagram we have seen. Beginning in state 12, if we see a digit, we go to state 13. In that state, we can read any number of additional digits. However, if we see anything but a digit or a dot, we have seen a number in the form of an integer; 123 is an example. That case is handled by entering state 20, where we return token **number** and a pointer to a table of constants where the found lexeme is entered. These mechanics are not shown on the diagram but are analogous to the way we handled identifiers.

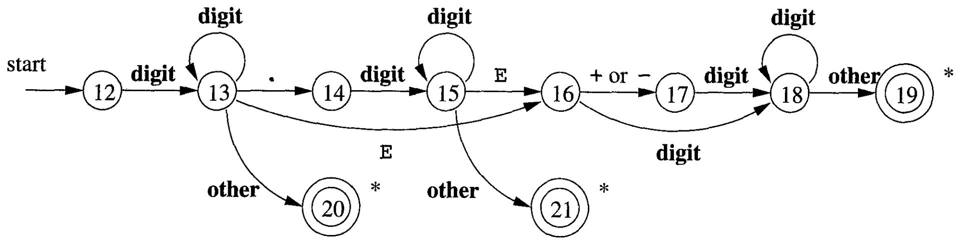


Figure 3.16: A transition diagram for unsigned numbers

If we instead see a dot in state 13, then we have an "optional fraction." State 14 is entered, and we look for one or more additional digits; state 15 is used for that purpose. If we see an E, then we have an "optional exponent," whose recognition is the job of states 16 through 19. Should we, in state 15, instead see anything but E or a digit, then we have come to the end of the fraction, there is no exponent, and we return the lexeme found, via state 21.

The final transition diagram, shown in Fig. 3.17, is for whitespace. In that diagram, we look for one or more "whitespace" characters, represented by **delim** in that diagram — typically these characters would be blank, tab, newline, and perhaps other characters that are not considered by the language design to be part of any token.

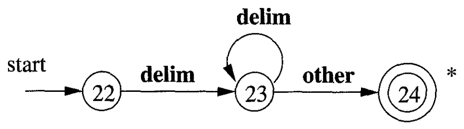


Figure 3.17: A transition diagram for whitespace

Note that in state 24, we have found a block of consecutive whitespace characters, followed by a nonwhitespace character. We retract the input to begin at the nonwhitespace, but we do not return to the parser. Rather, we must restart the process of lexical analysis after the whitespace.

3.4.4 Architecture of a Transition-Diagram-Based Lexical Analyzer

There are several ways that a collection of transition diagrams can be used to build a lexical analyzer. Regardless of the overall strategy, each state is represented by a piece of code. We may imagine a variable `state` holding the number of the current state for a transition diagram. A switch based on the value of `state` takes us to code for each of the possible states, where we find the action of that state. Often, the code for a state is itself a switch statement or multiway branch that determines the next state by reading and examining the next input character.

Example 3.10: In Fig. 3.18 we see a sketch of `getRelop()`, a C++ function whose job is to simulate the transition diagram of Fig. 3.13 and return an object of type `TOKEN`, that is, a pair consisting of the token name (which must be **relop** in this case) and an attribute value (the code for one of the six comparison operators in this case). `getRelop()` first creates a new object `retToken` and initializes its first component to `RELOP`, the symbolic code for token **relop**.

We see the typical behavior of a state in case 0, the case where the current state is 0. A function `nextChar()` obtains the next character from the input and assigns it to local variable c. We then check c for the three characters we expect to find, making the state transition dictated by the transition diagram of Fig. 3.13 in each case. For example, if the next input character is `=`, we go to state 5.

If the next input character is not one that can begin a comparison operator, then a function `fail()` is called. What `fail()` does depends on the global error-recovery strategy of the lexical analyzer. It should reset the `forward` pointer to `lexemeBegin`, in order to allow another transition diagram to be applied to

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

Figure 3.18: Sketch of implementation of **relop** transition diagram

the true beginning of the unprocessed input. It might then change the value of `state` to be the start state for another transition diagram, which will search for another token. Alternatively, if there is no other transition diagram that remains unused, `fail()` could initiate an error-correction phase that will try to repair the input and find a lexeme, as discussed in Section 3.1.4.

We also show the action for state 8 in Fig. 3.18. Because state 8 bears a *, we must retract the input pointer one position (i.e., put c back on the input stream). That task is accomplished by the function `retract()`. Since state 8 represents the recognition of lexeme >=, we set the second component of the returned object, which we suppose is named `attribute`, to `GT`, the code for this operator. □

To place the simulation of one transition diagram in perspective, let us consider the ways code like Fig. 3.18 could fit into the entire lexical analyzer.

1. We could arrange for the transition diagrams for each token to be tried sequentially. Then, the function `fail()` of Example 3.10 resets the pointer `forward` and starts the next transition diagram, each time it is called. This method allows us to use transition diagrams for the individual keywords, like the one suggested in Fig. 3.15. We have only to use these before we use the diagram for **id**, in order for the keywords to be reserved words.

2. We could run the various transition diagrams "in parallel," feeding the next input character to all of them and allowing each one to make whatever transitions it required. If we use this strategy, we must be careful to resolve the case where one diagram finds a lexeme that matches its pattern, while one or more other diagrams are still able to process input. The normal strategy is to take the longest prefix of the input that matches any pattern. That rule allows us to prefer identifier `thenext` to keyword `then`, or the operator `->` to `-`, for example.

3. The preferred approach, and the one we shall take up in the following sections, is to combine all the transition diagrams into one. We allow the transition diagram to read input until there is no possible next state, and then take the longest lexeme that matched any pattern, as we discussed in item (2) above. In our running example, this combination is easy, because no two tokens can start with the same character; i.e., the first character immediately tells us which token we are looking for. Thus, we could simply combine states 0, 9, 12, and 22 into one start state, leaving other transitions intact. However, in general, the problem of combining transition diagrams for several tokens is more complex, as we shall see shortly.

3.4.5 Exercises for Section 3.4

Exercise 3.4.1: Provide transition diagrams to recognize the same languages as each of the regular expressions in Exercise 3.3.2.

Exercise 3.4.2: Provide transition diagrams to recognize the same languages as each of the regular expressions in Exercise 3.3.5.

The following exercises, up to Exercise 3.4.12, introduce the Aho-Corasick algorithm for recognizing a collection of keywords in a text string in time proportional to the length of the text and the sum of the length of the keywords. This algorithm uses a special form of transition diagram called a *trie*. A trie is a tree-structured transition diagram with distinct labels on the edges leading from a node to its children. Leaves of the trie represent recognized keywords.

Knuth, Morris, and Pratt presented an algorithm for recognizing a single keyword $b_1 b_2 \cdots b_n$ in a text string. Here the trie is a transition diagram with n states, 0 through n. State 0 is the initial state, and state n represents acceptance, that is, discovery of the keyword. From each state s from 0 through $n-1$, there is a transition to state $s+1$, labeled by symbol b_{s+1}. For example, the trie for the keyword `ababaa` is:

$$\begin{tikzcd}
\textcircled{0} \arrow[r, "a"] & \textcircled{1} \arrow[r, "b"] & \textcircled{2} \arrow[r, "a"] & \textcircled{3} \arrow[r, "b"] & \textcircled{4} \arrow[r, "a"] & \textcircled{5} \arrow[r, "a"] & \textcircled{\textcircled{6}}
\end{tikzcd}$$

In order to process text strings rapidly and search those strings for a keyword, it is useful to define, for keyword $b_1 b_2 \cdots b_n$ and position s in that keyword (corresponding to state s of its trie), a *failure function*, $f(s)$, computed as in

Fig. 3.19. The objective is that $b_1 b_2 \cdots b_{f(s)}$ is the longest proper prefix of $b_1 b_2 \cdots b_s$ that is also a suffix of $b_1 b_2 \cdots b_s$. The reason $f(s)$ is important is that if we are trying to match a text string for $b_1 b_2 \cdots b_n$, and we have matched the first s positions, but we then fail (i.e., the next position of the text string does not hold b_{s+1}), then $f(s)$ is the longest prefix of $b_1 b_2 \cdots b_n$ that could possibly match the text string up to the point we are at. Of course, the next character of the text string must be $b_{f(s)+1}$, or else we still have problems and must consider a yet shorter prefix, which will be $b_{f(f(s))}$.

```
1)  $t = 0$;
2)  $f(1) = 0$;
3)  for ($s = 1$; $s < n$; $s++$) {
4)          while ($t > 0$ && $b_{s+1}$ != $b_{t+1}$) $t = f(t)$;
5)          if ($b_{s+1}$ == $b_{t+1}$) {
6)                  $t = t + 1$;
7)                  $f(s+1) = t$;
            }
8)          else $f(s+1) = 0$;
    }
```

Figure 3.19: Algorithm to compute the failure function for keyword $b_1 b_2 \cdots b_n$

As an example, the failure function for the trie constructed above for ababaa is:

s	1	2	3	4	5	6
$f(s)$	0	0	1	2	3	1

For instance, states 3 and 1 represent prefixes aba and a, respectively. $f(3) = 1$ because a is the longest proper prefix of aba that is also a suffix of aba. Also, $f(2) = 0$, because the longest proper prefix of ab that is also a suffix is the empty string.

Exercise 3.4.3: Construct the failure function for the strings:

a) abababaab.

b) aaaaaa.

c) abbaabb.

! **Exercise 3.4.4:** Prove, by induction on s, that the algorithm of Fig. 3.19 correctly computes the failure function.

!! **Exercise 3.4.5:** Show that the assignment $t = f(t)$ in line (4) of Fig. 3.19 is executed at most n times. Show that therefore, the entire algorithm takes only $O(n)$ time on a keyword of length n.

Having computed the failure function for a keyword $b_1b_2\cdots b_n$, we can scan a string $a_1a_2\cdots a_m$ in time $O(m)$ to tell whether the keyword occurs in the string. The algorithm, shown in Fig. 3.20, slides the keyword along the string, trying to make progress by matching the next character of the keyword with the next character of the string. If it cannot do so after matching s characters, then it "slides" the keyword right $s - f(s)$ positions, so only the first $f(s)$ characters of the keyword are considered matched with the string.

```
1)  $s = 0;$
2)  for ($i = 1; i \le m; i$++) {
3)          while ($s > 0$ && $a_i$ != $b_{s+1}$) $s = f(s);$
4)          if ($a_i == b_{s+1}$) $s = s + 1;$
5)          if ($s == n$) return "yes";
    }
6)  return "no";
```

Figure 3.20: The KMP algorithm tests whether string $a_1a_2\cdots a_m$ contains a single keyword $b_1b_2\cdots b_n$ as a substring in $O(m + n)$ time

Exercise 3.4.6: Apply Algorithm KMP to test whether keyword ababaa is a substring of:

a) abababaab.

b) abababbaa.

!! **Exercise 3.4.7:** Show that the algorithm of Fig. 3.20 correctly tells whether the keyword is a substring of the given string. *Hint*: proceed by induction on i. Show that for all i, the value of s after line (4) is the length of the longest prefix of the keyword that is a suffix of $a_1a_2\cdots a_i$.

!! **Exercise 3.4.8:** Show that the algorithm of Fig. 3.20 runs in time $O(m + n)$, assuming that function f is already computed and its values stored in an array indexed by s.

Exercise 3.4.9: The *Fibonacci strings* are defined as follows:

1. $s_1 =$ b.

2. $s_2 =$ a.

3. $s_k = s_{k-1}s_{k-2}$ for $k > 2$.

For example, $s_3 =$ ab, $s_4 =$ aba, and $s_5 =$ abaab.

a) What is the length of s_n?

b) Construct the failure function for s_6.

c) Construct the failure function for s_7.

!! d) Show that the failure function for any s_n can be expressed by $f(1) = f(2) = 0$, and for $2 < j \leq |s_n|$, $f(j)$ is $j - |s_{k-1}|$, where k is the largest integer such that $|s_k| \leq j + 1$.

!! e) In the KMP algorithm, what is the largest number of consecutive applications of the failure function, when we try to determine whether keyword s_k appears in text string s_{k+1}?

Aho and Corasick generalized the KMP algorithm to recognize any of a set of keywords in a text string. In this case, the trie is a true tree, with branching from the root. There is one state for every string that is a prefix (not necessarily proper) of any keyword. The parent of a state corresponding to string $b_1 b_2 \cdots b_k$ is the state that corresponds to $b_1 b_2 \cdots b_{k-1}$. A state is accepting if it corresponds to a complete keyword. For example, Fig. 3.21 shows the trie for the keywords `he`, `she`, `his`, and `hers`.

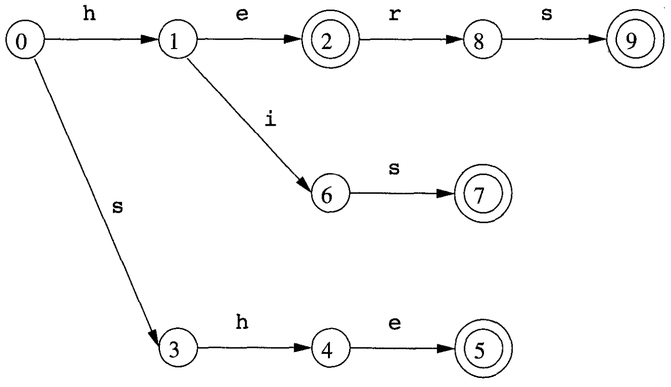


Figure 3.21: Trie for keywords **he**, **she**, **his**, **hers**

The failure function for the general trie is defined as follows. Suppose s is the state that corresponds to string $b_1 b_2 \cdots b_n$. Then $f(s)$ is the state that corresponds to the longest proper suffix of $b_1 b_2 \cdots b_n$ that is also a prefix of *some* keyword. For example, the failure function for the trie of Fig. 3.21 is:

s	1	2	3	4	5	6	7	8	9
$f(s)$	0	0	0	1	2	0	3	0	3

! **Exercise 3.4.10:** Modify the algorithm of Fig. 3.19 to compute the failure function for general tries. *Hint*: The major difference is that we cannot simply test for equality or inequality of b_{s+1} and b_{t+1} in lines (4) and (5) of Fig. 3.19. Rather, from any state there may be several transitions out on several characters, as there are transitions on both `e` and `i` from state 1 in Fig. 3.21. Any of those transitions could lead to a state that represents the longest suffix that is also a prefix.

Exercise 3.4.11: Construct the tries and compute the failure function for the following sets of keywords:

a) `aaa`, `abaaa`, and `ababaaa`.

b) `all`, `fall`, `fatal`, `llama`, and `lame`.

c) `pipe`, `pet`, `item`, `temper`, and `perpetual`.

! **Exercise 3.4.12:** Show that your algorithm from Exercise 3.4.10 still runs in time that is linear in the sum of the lengths of the keywords.

3.5 The Lexical-Analyzer Generator Lex

In this section, we introduce a tool called `Lex`, or in a more recent implementation `Flex`, that allows one to specify a lexical analyzer by specifying regular expressions to describe patterns for tokens. The input notation for the `Lex` tool is referred to as the *Lex language* and the tool itself is the *Lex compiler*. Behind the scenes, the Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called `lex.yy.c`, that simulates this transition diagram. The mechanics of how this translation from regular expressions to transition diagrams occurs is the subject of the next sections; here we only learn the Lex language.

3.5.1 Use of Lex

Figure 3.22 suggests how `Lex` is used. An input file, which we call `lex.l`, is written in the Lex language and describes the lexical analyzer to be generated. The Lex compiler transforms `lex.l` to a C program, in a file that is always named `lex.yy.c`. The latter file is compiled by the C compiler into a file called `a.out`, as always. The C-compiler output is a working lexical analyzer that can take a stream of input characters and produce a stream of tokens.

The normal use of the compiled C program, referred to as `a.out` in Fig. 3.22, is as a subroutine of the parser. It is a C function that returns an integer, which is a code for one of the possible token names. The attribute value, whether it be another numeric code, a pointer to the symbol table, or nothing, is placed in a global variable `yylval`,[2] which is shared between the lexical analyzer and parser, thereby making it simple to return both the name and an attribute value of a token.

[2]Incidentally, the `yy` that appears in `yylval` and `lex.yy.c` refers to the `Yacc` parser-generator, which we shall describe in Section 4.9, and which is commonly used in conjunction with `Lex`.

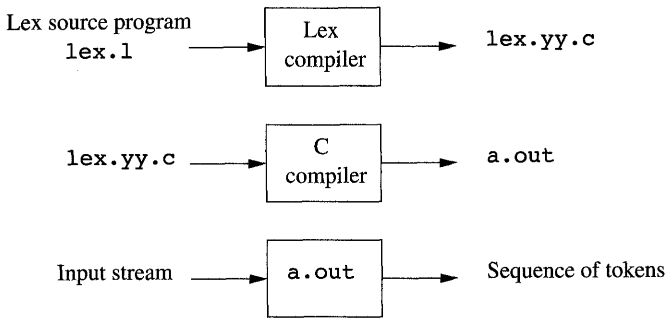


Figure 3.22: Creating a lexical analyzer with Lex

3.5.2 Structure of Lex Programs

A Lex program has the following form:

```
declarations
%%
translation rules
%%
auxiliary functions
```

The declarations section includes declarations of variables, *manifest constants* (identifiers declared to stand for a constant, e.g., the name of a token), and regular definitions, in the style of Section 3.3.4.

The translation rules each have the form

```
Pattern    { Action }
```

Each pattern is a regular expression, which may use the regular definitions of the declaration section. The actions are fragments of code, typically written in C, although many variants of Lex using other languages have been created.

The third section holds whatever additional functions are used in the actions. Alternatively, these functions can be compiled separately and loaded with the lexical analyzer.

The lexical analyzer created by Lex behaves in concert with the parser as follows. When called by the parser, the lexical analyzer begins reading its remaining input, one character at a time, until it finds the longest prefix of the input that matches one of the patterns P_i. It then executes the associated action A_i. Typically, A_i will return to the parser, but if it does not (e.g., because P_i describes whitespace or comments), then the lexical analyzer proceeds to find additional lexemes, until one of the corresponding actions causes a return to the parser. The lexical analyzer returns a single value, the token name, to the parser, but uses the shared, integer variable `yylval` to pass additional information about the lexeme found, if needed.

Example 3.11: Figure 3.23 is a Lex program that recognizes the tokens of Fig. 3.12 and returns the token found. A few observations about this code will introduce us to many of the important features of Lex.

In the declarations section we see a pair of special brackets, %{ and %}. Anything within these brackets is copied directly to the file lex.yy.c, and is not treated as a regular definition. It is common to place there the definitions of the manifest constants, using C #define statements to associate unique integer codes with each of the manifest constants. In our example, we have listed in a comment the names of the manifest constants, LT, IF, and so on, but have not shown them defined to be particular integers.[3]

Also in the declarations section is a sequence of regular definitions. These use the extended notation for regular expressions described in Section 3.3.5. Regular definitions that are used in later definitions or in the patterns of the translation rules are surrounded by curly braces. Thus, for instance, *delim* is defined to be a shorthand for the character class consisting of the blank, the tab, and the newline; the latter two are represented, as in all UNIX commands, by backslash followed by t or n, respectively. Then, *ws* is defined to be one or more delimiters, by the regular expression {delim}+.

Notice that in the definition of *id* and *number*, parentheses are used as grouping metasymbols and do not stand for themselves. In contrast, E in the definition of *number* stands for itself. If we wish to use one of the Lex metasymbols, such as any of the parentheses, +, *, or ?, to stand for themselves, we may precede them with a backslash. For instance, we see \. in the definition of *number*, to represent the dot, since that character is a metasymbol representing "any character," as usual in UNIX regular expressions.

In the auxiliary-function section, we see two such functions, installID() and installNum(). Like the portion of the declaration section that appears between %{...%}, everything in the auxiliary section is copied directly to file lex.yy.c, but may be used in the actions.

Finally, let us examine some of the patterns and rules in the middle section of Fig. 3.23. First, *ws*, an identifier declared in the first section, has an associated empty action. If we find whitespace, we do not return to the parser, but look for another lexeme. The second token has the simple regular expression pattern if. Should we see the two letters if on the input, and they are not followed by another letter or digit (which would cause the lexical analyzer to find a longer prefix of the input matching the pattern for **id**), then the lexical analyzer consumes these two letters from the input and returns the token name IF, that is, the integer for which the manifest constant IF stands. Keywords then and else are treated similarly.

The fifth token has the pattern defined by *id*. Note that, although keywords like if match this pattern as well as an earlier pattern, Lex chooses whichever

[3]If Lex is used along with Yacc, then it would be normal to define the manifest constants in the Yacc program and use them without definition in the Lex program. Since lex.yy.c is compiled with the Yacc output, the constants thus will be available to the actions in the Lex program.

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim      [ \t\n]
ws         {delim}+
letter     [A-Za-z]
digit      [0-9]
id         {letter}({letter}|{digit})*
number     {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%

{ws}       {/* no action and no return */}
if         {return(IF);}
then       {return(THEN);}
else       {return(ELSE);}
{id}       {yylval = (int) installID(); return(ID);}
{number}   {yylval = (int) installNum(); return(NUMBER);}
"<"        {yylval = LT; return(RELOP);}
"<="       {yylval = LE; return(RELOP);}
"="        {yylval = EQ; return(RELOP);}
"<>"       {yylval = NE; return(RELOP);}
">"        {yylval = GT; return(RELOP);}
">="       {yylval = GE; return(RELOP);}

%%

int installID() {/* function to install the lexeme, whose
                    first character is pointed to by yytext,
                    and whose length is yyleng, into the
                    symbol table and return a pointer
                    thereto */
}

int installNum() {/* similar to installID, but puts numer-
                     ical constants into a separate table */
}
```

Figure 3.23: Lex program for the tokens of Fig. 3.12

pattern is listed first in situations where the longest matching prefix matches two or more patterns. The action taken when *id* is matched is threefold:

1. Function `installID()` is called to place the lexeme found in the symbol table.

2. This function returns a pointer to the symbol table, which is placed in global variable `yylval`, where it can be used by the parser or a later component of the compiler. Note that `installID()` has available to it two variables that are set automatically by the lexical analyzer that `Lex` generates:

   (a) `yytext` is a pointer to the beginning of the lexeme, analogous to `lexemeBegin` in Fig. 3.3.

   (b) `yyleng` is the length of the lexeme found.

3. The token name `ID` is returned to the parser.

The action taken when a lexeme matching the pattern *number* is similar, using the auxiliary function `installNum()`. □

3.5.3 Conflict Resolution in `Lex`

We have alluded to the two rules that `Lex` uses to decide on the proper lexeme to select, when several prefixes of the input match one or more patterns:

1. Always prefer a longer prefix to a shorter prefix.

2. If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the `Lex` program.

Example 3.12: The first rule tells us to continue reading letters and digits to find the longest prefix of these characters to group as an identifier. It also tells us to treat `<=` as a single lexeme, rather than selecting `<` as one lexeme and `=` as the next lexeme. The second rule makes keywords reserved, if we list the keywords before **id** in the program. For instance, if **then** is determined to be the longest prefix of the input that matches any pattern, and the pattern **then** precedes `{id}`, as it does in Fig. 3.23, then the token `THEN` is returned, rather than `ID`. □

3.5.4 The Lookahead Operator

`Lex` automatically reads one character ahead of the last character that forms the selected lexeme, and then retracts the input so only the lexeme itself is consumed from the input. However, sometimes, we want a certain pattern to be matched to the input only when it is followed by a certain other characters. If so, we may use the slash in a pattern to indicate the end of the part of the pattern that matches the lexeme. What follows / is additional pattern that must be matched before we can decide that the token in question was seen, but what matches this second pattern is not part of the lexeme.

Example 3.13: In Fortran and some other languages, keywords are not reserved. That situation creates problems, such as a statement

```
IF(I,J) = 3
```

where `IF` is the name of an array, not a keyword. This statement contrasts with statements of the form

```
IF( condition ) THEN ...
```

where `IF` is a keyword. Fortunately, we can be sure that the keyword `IF` is always followed by a left parenthesis, some text — the condition — that may contain parentheses, a right parenthesis and a letter. Thus, we could write a `Lex` rule for the keyword `IF` like:

```
IF / \( .* \) {letter}
```

This rule says that the pattern the lexeme matches is just the two letters `IF`. The slash says that additional pattern follows but does not match the lexeme. In this pattern, the first character is the left parentheses. Since that character is a `Lex` metasymbol, it must be preceded by a backslash to indicate that it has its literal meaning. The dot and star match "any string without a newline." Note that the dot is a `Lex` metasymbol meaning "any character except newline." It is followed by a right parenthesis, again with a backslash to give that character its literal meaning. The additional pattern is followed by the symbol *letter*, which is a regular definition representing the character class of all letters.

Note that in order for this pattern to be foolproof, we must preprocess the input to delete whitespace. We have in the pattern neither provision for whitespace, nor can we deal with the possibility that the condition extends over lines, since the dot will not match a newline character.

For instance, suppose this pattern is asked to match a prefix of input:

```
IF(A<(B+C)*D)THEN...
```

the first two characters match `IF`, the next character matches `\(`, the next nine characters match `.*`, and the next two match `\)` and *letter*. Note the fact that the first right parenthesis (after `C`) is not followed by a letter is irrelevant; we only need to find some way of matching the input to the pattern. We conclude that the letters `IF` constitute the lexeme, and they are an instance of token **if**. □

3.5.5 Exercises for Section 3.5

Exercise 3.5.1: Describe how to make the following modifications to the `Lex` program of Fig. 3.23:

a) Add the keyword `while`.

b) Change the comparison operators to be the C operators of that kind.

c) Allow the underscore (_) as an additional letter.

! d) Add a new pattern with token `STRING`. The pattern consists of a double-quote ("), any string of characters and a final double-quote. However, if a double-quote appears in the string, it must be escaped by preceding it with a backslash (\), and therefore a backslash in the string must be represented by two backslashes. The lexical value, which is the string without the surrounding double-quotes, and with backslashes used to escape a character removed. Strings are to be installed in a table of strings.

Exercise 3.5.2: Write a `Lex` program that copies a file, replacing each nonempty sequence of white space by a single blank.

Exercise 3.5.3: Write a `Lex` program that copies a C program, replacing each instance of the keyword `float` by `double`.

! **Exercise 3.5.4:** Write a `Lex` program that converts a file to "Pig latin." Specifically, assume the file is a sequence of words (groups of letters) separated by whitespace. Every time you encounter a word:

1. If the first letter is a consonant, move it to the end of the word and then add `ay`.

2. If the first letter is a vowel, just add `ay` to the end of the word.

All nonletters are copied intact to the output.

! **Exercise 3.5.5:** In SQL, keywords and identifiers are case-insensitive. Write a `Lex` program that recognizes the keywords `SELECT`, `FROM`, and `WHERE` (in any combination of capital and lower-case letters), and token `ID`, which for the purposes of this exercise you may take to be any sequence of letters and digits, beginning with a letter. You need not install identifiers in a symbol table, but tell how the "install" function would differ from that described for case-sensitive identifiers as in Fig. 3.23.

3.6 Finite Automata

We shall now discover how Lex turns its input program into a lexical analyzer. At the heart of the transition is the formalism known as *finite automata*. These are essentially graphs, like transition diagrams, with a few differences:

1. Finite automata are *recognizers*; they simply say "yes" or "no" about each possible input string.

2. Finite automata come in two flavors:

   (a) *Nondeterministic finite automata* (NFA) have no restrictions on the labels of their edges. A symbol can label several edges out of the same state, and ϵ, the empty string, is a possible label.

   (b) *Deterministic finite automata* (DFA) have, for each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.

Both deterministic and nondeterministic finite automata are capable of recognizing the same languages. In fact these languages are exactly the same languages, called the *regular languages*, that regular expressions can describe.[4]

3.6.1 Nondeterministic Finite Automata

A *nondeterministic finite automaton* (NFA) consists of:

1. A finite set of states S.

2. A set of input symbols Σ, the *input alphabet*. We assume that ϵ, which stands for the empty string, is never a member of Σ.

3. A *transition function* that gives, for each state, and for each symbol in $\Sigma \cup \{\epsilon\}$ a set of *next states*.

4. A state s_0 from S that is distinguished as the *start state* (or *initial state*).

5. A set of states F, a subset of S, that is distinguished as the *accepting states* (or *final states*).

We can represent either an NFA or DFA by a *transition graph*, where the nodes are states and the labeled edges represent the transition function. There is an edge labeled a from state s to state t if and only if t is one of the next states for state s and input a. This graph is very much like a transition diagram, except:

[4]There is a small lacuna: as we defined them, regular expressions cannot describe the empty language, since we would never want to use this pattern in practice. However, finite automata *can* define the empty language. In the theory, $\emptyset$ is treated as an additional regular expression for the sole purpose of defining the empty language.

a) The same symbol can label edges from one state to several different states, and

b) An edge may be labeled by ϵ, the empty string, instead of, or in addition to, symbols from the input alphabet.

Example 3.14: The transition graph for an NFA recognizing the language of regular expression $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$ is shown in Fig. 3.24. This abstract example, describing all strings of a's and b's ending in the particular string abb, will be used throughout this section. It is similar to regular expressions that describe languages of real interest, however. For instance, an expression describing all files whose name ends in `.o` is **any***`.o`, where **any** stands for any printable character.

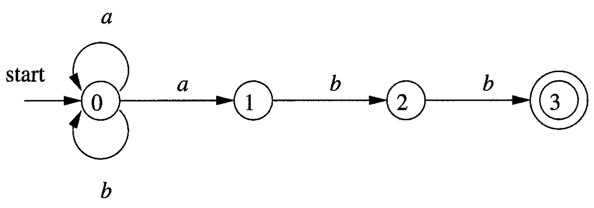


Figure 3.24: A nondeterministic finite automaton

Following our convention for transition diagrams, the double circle around state 3 indicates that this state is accepting. Notice that the only ways to get from the start state 0 to the accepting state is to follow some path that stays in state 0 for a while, then goes to states 1, 2, and 3 by reading abb from the input. Thus, the only strings getting to the accepting state are those that end in abb. □

3.6.2 Transition Tables

We can also represent an NFA by a *transition table*, whose rows correspond to states, and whose columns correspond to the input symbols and ϵ. The entry for a given state and input is the value of the transition function applied to those arguments. If the transition function has no information about that state-input pair, we put $\emptyset$ in the table for the pair.

Example 3.15: The transition table for the NFA of Fig. 3.24 is shown in Fig. 3.25. □

The transition table has the advantage that we can easily find the transitions on a given state and input. Its disadvantage is that it takes a lot of space, when the input alphabet is large, yet most states do not have any moves on most of the input symbols.

STATE	a	b	ϵ
0	$\{0,1\}$	$\{0\}$	$\emptyset$
1	$\emptyset$	$\{2\}$	$\emptyset$
2	$\emptyset$	$\{3\}$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$

Figure 3.25: Transition table for the NFA of Fig. 3.24

3.6.3 Acceptance of Input Strings by Automata

An NFA *accepts* input string x if and only if there is some path in the transition graph from the start state to one of the accepting states, such that the symbols along the path spell out x. Note that ϵ labels along the path are effectively ignored, since the empty string does not contribute to the string constructed along the path.

Example 3.16: The string *aabb* is accepted by the NFA of Fig. 3.24. The path labeled by *aabb* from state 0 to state 3 demonstrating this fact is:

$$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$$

Note that several paths labeled by the same string may lead to different states. For instance, path

$$0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$$

is another path from state 0 labeled by the string *aabb*. This path leads to state 0, which is not accepting. However, remember that an NFA accepts a string as long as *some* path labeled by that string leads from the start state to an accepting state. The existence of other paths leading to a nonaccepting state is irrelevant. □

The *language defined* (or *accepted*) by an NFA is the set of strings labeling some path from the start to an accepting state. As was mentioned, the NFA of Fig. 3.24 defines the same language as does the regular expression $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$, that is, all strings from the alphabet $\{a, b\}$ that end in *abb*. We may use $L(A)$ to stand for the language accepted by automaton A.

Example 3.17: Figure 3.26 is an NFA accepting $L(\mathbf{aa}^*|\mathbf{bb}^*)$. String *aaa* is accepted because of the path

$$0 \xrightarrow{\varepsilon} 1 \xrightarrow{a} 2 \xrightarrow{a} 2 \xrightarrow{a} 2$$

Note that ϵ's "disappear" in a concatenation, so the label of the path is *aaa*.
□

3.6.4 Deterministic Finite Automata

A *deterministic finite automaton* (DFA) is a special case of an NFA where:

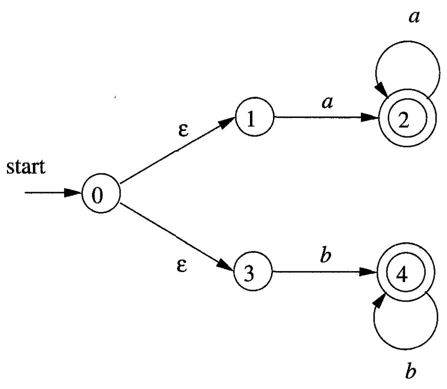


Figure 3.26: NFA accepting **aa***|**bb***

1. There are no moves on input ϵ, and

2. For each state s and input symbol a, there is exactly one edge out of s labeled a.

If we are using a transition table to represent a DFA, then each entry is a single state. we may therefore represent this state without the curly braces that we use to form sets.

While the NFA is an abstract representation of an algorithm to recognize the strings of a certain language, the DFA is a simple, concrete algorithm for recognizing strings. It is fortunate indeed that every regular expression and every NFA can be converted to a DFA accepting the same language, because it is the DFA that we really implement or simulate when building lexical analyzers. The following algorithm shows how to apply a DFA to a string.

Algorithm 3.18: Simulating a DFA.

INPUT: An input string x terminated by an end-of-file character **eof**. A DFA D with start state s_0, accepting states F, and transition function *move*.

OUTPUT: Answer "yes" if D accepts x; "no" otherwise.

METHOD: Apply the algorithm in Fig. 3.27 to the input string x. The function $move(s, c)$ gives the state to which there is an edge from state s on input c. The function *nextChar* returns the next character of the input string x. □

Example 3.19: In Fig. 3.28 we see the transition graph of a DFA accepting the language (**a**|**b**)***abb**, the same as that accepted by the NFA of Fig. 3.24. Given the input string *ababb*, this DFA enters the sequence of states $0, 1, 2, 1, 2, 3$ and returns "yes." □

```
$s = s_0$;
$c = nextChar()$;
while ( $c$ != eof ) {
        $s = move(s, c)$;
        $c = nextChar()$;
}
if ( $s$ is in $F$ ) return "yes";
else return "no";
```

Figure 3.27: Simulating a DFA

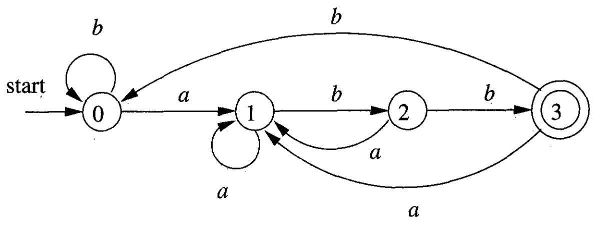


Figure 3.28: DFA accepting (**a**|**b**)***abb**

3.6.5 Exercises for Section 3.6

! **Exercise 3.6.1:** Figure 3.19 in the exercises of Section 3.4 computes the failure function for the KMP algorithm. Show how, given that failure function, we can construct, from a keyword $b_1 b_2 \cdots b_n$ an $n+1$-state DFA that recognizes $.^* b_1 b_2 \cdots b_n$, where the dot stands for "any character." Moreover, this DFA can be constructed in $O(n)$ time.

Exercise 3.6.2: Design finite automata (deterministic or nondeterministic) for each of the languages of Exercise 3.3.5.

Exercise 3.6.3: For the NFA of Fig. 3.29, indicate all the paths labeled *aabb*. Does the NFA accept *aabb*?

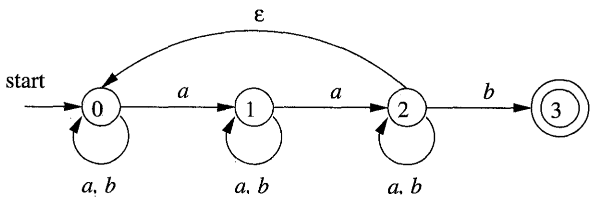


Figure 3.29: NFA for Exercise 3.6.3

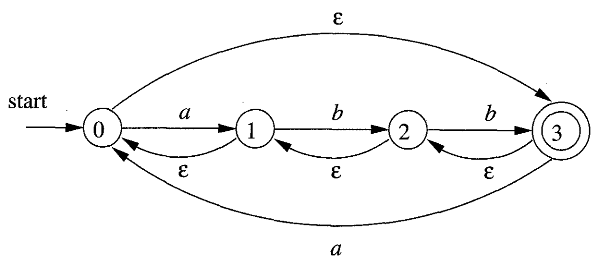


Figure 3.30: NFA for Exercise 3.6.4

Exercise 3.6.4: Repeat Exercise 3.6.3 for the NFA of Fig. 3.30.

Exercise 3.6.5: Give the transition tables for the NFA of:

a) Exercise 3.6.3.

b) Exercise 3.6.4.

c) Figure 3.26.

3.7 From Regular Expressions to Automata

The regular expression is the notation of choice for describing lexical analyzers and other pattern-processing software, as was reflected in Section 3.5. However, implementation of that software requires the simulation of a DFA, as in Algorithm 3.18, or perhaps simulation of an NFA. Because an NFA often has a choice of move on an input symbol (as Fig. 3.24 does on input a from state 0) or on ϵ (as Fig. 3.26 does from state 0), or even a choice of making a transition on ϵ or on a real input symbol, its simulation is less straightforward than for a DFA. Thus often it is important to convert an NFA to a DFA that accepts the same language.

In this section we shall first show how to convert NFA's to DFA's. Then, we use this technique, known as "the subset construction," to give a useful algorithm for simulating NFA's directly, in situations (other than lexical analysis) where the NFA-to-DFA conversion takes more time than the direct simulation. Next, we show how to convert regular expressions to NFA's, from which a DFA can be constructed if desired. We conclude with a discussion of the time-space tradeoffs inherent in the various methods for implementing regular expressions, and see how to choose the appropriate method for your application.

3.7.1 Conversion of an NFA to a DFA

The general idea behind the subset construction is that each state of the constructed DFA corresponds to a set of NFA states. After reading input

$a_1a_2 \cdots a_n$, the DFA is in that state which corresponds to the set of states that the NFA can reach, from its start state, following paths labeled $a_1a_2 \cdots a_n$.

It is possible that the number of DFA states is exponential in the number of NFA states, which could lead to difficulties when we try to implement this DFA. However, part of the power of the automaton-based approach to lexical analysis is that for real languages, the NFA and DFA have approximately the same number of states, and the exponential behavior is not seen.

Algorithm 3.20: The *subset construction* of a DFA from an NFA.

INPUT: An NFA N.

OUTPUT: A DFA D accepting the same language as N.

METHOD: Our algorithm constructs a transition table *Dtran* for D. Each state of D is a set of NFA states, and we construct *Dtran* so D will simulate "in parallel" all possible moves N can make on a given input string. Our first problem is to deal with ϵ-transitions of N properly. In Fig. 3.31 we see the definitions of several functions that describe basic computations on the states of N that are needed in the algorithm. Note that s is a single state of N, while T is a set of states of N.

OPERATION	DESCRIPTION
ϵ-*closure*(s)	Set of NFA states reachable from NFA state s on ϵ-transitions alone.
ϵ-*closure*(T)	Set of NFA states reachable from some NFA state s in set T on ϵ-transitions alone; $= \cup_{s \text{ in } T}$ ϵ-*closure*(s).
move(T, a)	Set of NFA states to which there is a transition on input symbol a from some state s in T.

Figure 3.31: Operations on NFA states

We must explore those sets of states that N can be in after seeing some input string. As a basis, before reading the first input symbol, N can be in any of the states of ϵ-*closure*(s_0), where s_0 is its start state. For the induction, suppose that N can be in set of states T after reading input string x. If it next reads input a, then N can immediately go to any of the states in *move*(T, a). However, after reading a, it may also make several ϵ-transitions; thus N could be in any state of ϵ-*closure*$(\textit{move}(T, a))$ after reading input xa. Following these ideas, the construction of the set of D's states, *Dstates*, and its transition function *Dtran*, is shown in Fig. 3.32.

The start state of D is ϵ-*closure*(s_0), and the accepting states of D are all those sets of N's states that include at least one accepting state of N. To complete our description of the subset construction, we need only to show how

```
initially, $\epsilon$-closure($s_0$) is the only state in Dstates, and it is unmarked;
while ( there is an unmarked state $T$ in Dstates ) {
        mark $T$;
        for ( each input symbol $a$ ) {
                $U = \epsilon\text{-}closure(move(T, a))$;
                if ( $U$ is not in Dstates )
                        add $U$ as an unmarked state to Dstates;
                $Dtran[T, a] = U$;
        }
}
```

Figure 3.32: The subset construction

ϵ-closure(T) is computed for any set of NFA states T. This process, shown in Fig. 3.33, is a straightforward search in a graph from a set of states. In this case, imagine that only the ϵ-labeled edges are available in the graph. □

```
push all states of $T$ onto stack;
initialize $\epsilon$-closure($T$) to $T$;
while ( stack is not empty ) {
        pop $t$, the top element, off stack;
        for ( each state $u$ with an edge from $t$ to $u$ labeled $\epsilon$ )
                if ( $u$ is not in $\epsilon$-closure($T$) ) {
                        add $u$ to $\epsilon$-closure($T$);
                        push $u$ onto stack;
                }
}
```

Figure 3.33: Computing *ϵ-closure*(T)

Example 3.21 : Figure 3.34 shows another NFA accepting (**a**|**b**)***abb**; it happens to be the one we shall construct directly from this regular expression in Section 3.7. Let us apply Algorithm 3.20 to Fig. 3.29.

The start state A of the equivalent DFA is *ϵ-closure*(0), or $A = \{0, 1, 2, 4, 7\}$, since these are exactly the states reachable from state 0 via a path all of whose edges have label ϵ. Note that a path can have zero edges, so state 0 is reachable from itself by an ϵ-labeled path.

The input alphabet is $\{a, b\}$. Thus, our first step is to mark A and compute $Dtran[A, a] = \epsilon\text{-}closure(move(A, a))$ and $Dtran[A, b] = \epsilon\text{-}closure(move(A, b))$. Among the states 0, 1, 2, 4, and 7, only 2 and 7 have transitions on a, to 3 and 8, respectively. Thus, $move(A, a) = \{3, 8\}$. Also, $\epsilon\text{-}closure(\{3, 8\} = \{1, 2, 3, 4, 6, 7, 8\}$, so we conclude

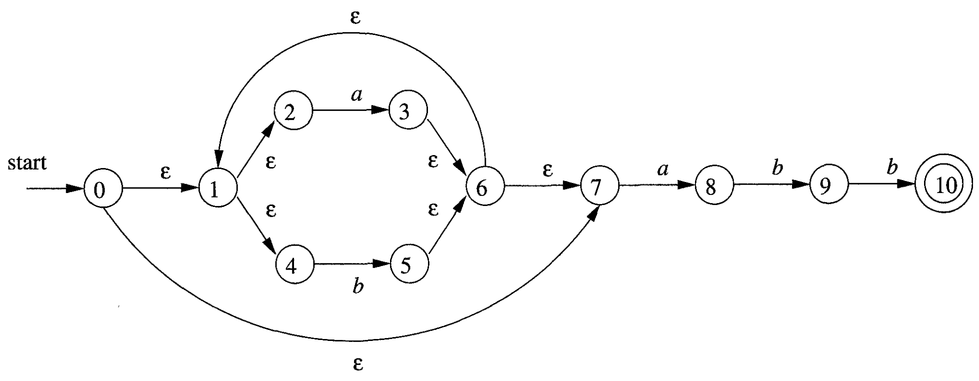


Figure 3.34: NFA N for $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$

$$Dtran[A,a] = \epsilon\text{-}closure\big(move(A,a)\big) = \epsilon\text{-}closure(\{3,8\}) = \{1,2,3,4,6,7,8\}$$

Let us call this set B, so $Dtran[A,a] = B$.

Now, we must compute $Dtran[A,b]$. Among the states in A, only 4 has a transition on b, and it goes to 5. Thus,

$$Dtran[A,b] = \epsilon\text{-}closure(\{5\}) = \{1,2,4,6,7\}$$

Let us call the above set C, so $Dtran[A,b] = C$.

NFA State	DFA State	a	b
$\{0,1,2,4,7\}$	A	B	C
$\{1,2,3,4,6,7,8\}$	B	B	D
$\{1,2,4,5,6,7\}$	C	B	C
$\{1,2,4,5,6,7,9\}$	D	B	E
$\{1,2,3,5,6,7,10\}$	E	B	C

Figure 3.35: Transition table $Dtran$ for DFA D

If we continue this process with the unmarked sets B and C, we eventually reach a point where all the states of the DFA are marked. This conclusion is guaranteed, since there are "only" 2^{11} different subsets of a set of eleven NFA states. The five different DFA states we actually construct, their corresponding sets of NFA states, and the transition table for the DFA D are shown in Fig. 3.35, and the transition graph for D is in Fig. 3.36. State A is the start state, and state E, which contains state 10 of the NFA, is the only accepting state.

Note that D has one more state than the DFA of Fig. 3.28 for the same language. States A and C have the same move function, and so can be merged. We discuss the matter of minimizing the number of states of a DFA in Section 3.9.6. □

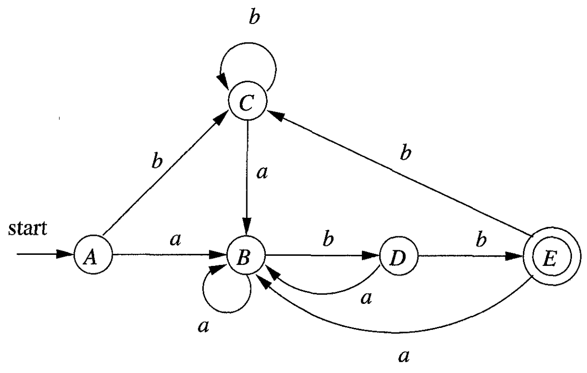

$$\begin{tikzcd}
 & & C \arrow[loop above, "b"] \arrow[d, "a"] & & \\
\text{start} \arrow[r] & A \arrow[r, "a"] \arrow[ur, "b"] & B \arrow[loop below, "a"] \arrow[r, "b"] & D \arrow[r, "b"] \arrow[l, bend left, "a"] & E \arrow[ull, "b"'] \arrow[ll, bend left=40, "a"]
\end{tikzcd}$$

Figure 3.36: Result of applying the subset construction to Fig. 3.34

3.7.2 Simulation of an NFA

A strategy that has been used in a number of text-editing programs is to construct an NFA from a regular expression and then simulate the NFA using something like an on-the-fly subset construction. The simulation is outlined below.

Algorithm 3.22: Simulating an NFA.

INPUT: An input string x terminated by an end-of-file character **eof**. An NFA N with start state s_0, accepting states F, and transition function *move*.

OUTPUT: Answer "yes" if M accepts x; "no" otherwise.

METHOD: The algorithm keeps a set of current states S, those that are reached from s_0 following a path labeled by the inputs read so far. If c is the next input character, read by the function *nextChar*(), then we first compute $move(S, c)$ and then close that set using ϵ-*closure*(). The algorithm is sketched in Fig. 3.37. □

```
1)  $S = \epsilon\text{-}closure(s_0)$;
2)  $c = nextChar()$;
3)  while ( $c$ != eof ) {
4)          $S = \epsilon\text{-}closure(move(S, c))$;
5)          $c = nextChar()$;
6)  }
7)  if ( $S \cap F$ != $\emptyset$ ) return "yes";
8)  else return "no";
```

Figure 3.37: Simulating an NFA

3.7.3 Efficiency of NFA Simulation

If carefully implemented, Algorithm 3.22 can be quite efficient. As the ideas involved are useful in a number of similar algorithms involving search of graphs, we shall look at this implementation in additional detail. The data structures we need are:

1. Two stacks, each of which holds a set of NFA states. One of these stacks, *oldStates*, holds the "current" set of states, i.e., the value of S on the right side of line (4) in Fig. 3.37. The second, *newStates*, holds the "next" set of states — S on the left side of line (4). Unseen is a step where, as we go around the loop of lines (3) through (6), *newStates* is transferred to *oldStates*.

2. A boolean array *alreadyOn*, indexed by the NFA states, to indicate which states are in *newStates*. While the array and stack hold the same information, it is much faster to interrogate *alreadyOn*$[s]$ than to search for state s on the stack *newStates*. It is for this efficiency that we maintain both representations.

3. A two-dimensional array *move*$[s, a]$ holding the transition table of the NFA. The entries in this table, which are sets of states, are represented by linked lists.

To implement line (1) of Fig. 3.37, we need to set each entry in array *alreadyOn* to `FALSE`, then for each state s in ϵ-*closure*(s_0), push s onto *oldStates* and set *alreadyOn*$[s]$ to `TRUE`. This operation on state s, and the implementation of line (4) as well, are facilitated by a function we shall call *addState*(s). This function pushes state s onto *newStates*, sets *alreadyOn*$[s]$ to `TRUE`, and calls itself recursively on the states in *move*$[s, \epsilon]$ in order to further the computation of ϵ-*closure*(s). However, to avoid duplicating work, we must be careful never to call *addState* on a state that is already on the stack *newStates*. Figure 3.38 sketches this function.

```
 9)  addState(s) {
10)          push s onto newStates;
11)          alreadyOn[s] = TRUE;
12)          for ( t on move[s, ε] )
13)                  if ( !alreadyOn(t) )
14)                          addState(t);
15)  }
```

Figure 3.38: Adding a new state s, which is known not to be on *newStates*

We implement line (4) of Fig. 3.37 by looking at each state s on *oldStates*. We first find the set of states *move*$[s, c]$, where c is the next input, and for each of those states that is not already on *newStates*, we apply *addState* to it. Note that *addState* has the effect of computing the ϵ-*closure* and adding all those states to *newStates* as well, if they were not already on. This sequence of steps is summarized in Fig. 3.39.

```
16)  for ( s on oldStates ) {
17)      for ( t on move[s,c] )
18)          if ( !alreadyOn[t] )
19)              addState(t);
20)      pop s from oldStates;
21)  }

22)  for ( s on newStates ) {
23)      pop s from newStates;
24)      push s onto oldStates;
25)      alreadyOn[s] = FALSE;
26)  }
```

Figure 3.39: Implementation of step (4) of Fig. 3.37

Now, suppose that the NFA N has n states and m transitions; i.e., m is the sum over all states of the number of symbols (or ϵ) on which the state has a transition out. Not counting the call to *addState* at line (19) of Fig. 3.39, the time spent in the loop of lines (16) through (21) is $O(n)$. That is, we can go around the loop at most n times, and each step of the loop requires constant work, except for the time spent in *addState*. The same is true of the loop of lines (22) through (26).

During one execution of Fig. 3.39, i.e., of step (4) of Fig. 3.37, it is only possible to call *addState* on a given state once. The reason is that whenever we call *addState*(s), we set *alreadyOn*$[s]$ to `TRUE` at line (11) of Fig. 3.39. Once *alreadyOn*$[s]$ is `TRUE`, the tests at line (13) of Fig. 3.38 and line (18) of Fig. 3.39 prevent another call.

The time spent in one call to *addState*, exclusive of the time spent in recursive calls at line (14), is $O(1)$ for lines (10) and (11). For lines (12) and (13), the time depends on how many ϵ-transitions there are out of state s. We do not know this number for a given state, but we know that there are at most m transitions in total, out of all states. As a result, the aggregate time spent in lines (11) over all calls to *addState* during one execution of the code of Fig. 3.39 is $O(m)$. The aggregate for the rest of the steps of *addState* is $O(n)$, since it is a constant per call, and there are at most n calls.

We conclude that, implemented properly, the time to execute line (4) of Fig. 3.37 is $O(n+m)$. The rest of the while-loop of lines (3) through (6) takes $O(1)$ time per iteration. If the input x is of length k, then the total work in that loop is $O((k(n+m))$. Line (1) of Fig. 3.37 can be executed in $O(n+m)$ time, since it is essentially the steps of Fig. 3.39 with *oldStates* containing only

Big-Oh Notation

An expression like $O(n)$ is a shorthand for "at most some constant times n." Technically, we say a function $f(n)$, perhaps the running time of some step of an algorithm, is $O\big(g(n)\big)$ if there are constants c and n_0, such that whenever $n \geq n_0$, it is true that $f(n) \leq cg(n)$. A useful idiom is "$O(1)$," which means "some constant." The use of this *big-oh notation* enables us to avoid getting too far into the details of what we count as a unit of execution time, yet lets us express the rate at which the running time of an algorithm grows.

the state s_0. Lines (2), (7), and (8) each take $O(1)$ time. Thus, the running time of Algorithm 3.22, properly implemented, is $O\big((k(n+m)\big)$. That is, the time taken is proportional to the length of the input times the size (nodes plus edges) of the transition graph.

3.7.4 Construction of an NFA from a Regular Expression

We now give an algorithm for converting any regular expression to an NFA that defines the same language. The algorithm is syntax-directed, in the sense that it works recursively up the parse tree for the regular expression. For each subexpression the algorithm constructs an NFA with a single accepting state.

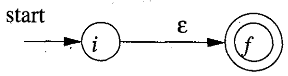
Algorithm 3.23: The McNaughton-Yamada-Thompson algorithm to convert a regular expression to an NFA.

INPUT: A regular expression r over alphabet Σ.

OUTPUT: An NFA N accepting $L(r)$.

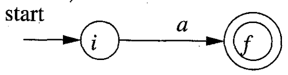
METHOD: Begin by parsing r into its constituent subexpressions. The rules for constructing an NFA consist of basis rules for handling subexpressions with no operators, and inductive rules for constructing larger NFA's from the NFA's for the immediate subexpressions of a given expression.

BASIS: For expression ϵ construct the NFA



Here, i is a new state, the start state of this NFA, and f is another new state, the accepting state for the NFA.

For any subexpression a in Σ, construct the NFA



where again i and f are new states, the start and accepting states, respectively. Note that in both of the basis constructions, we construct a distinct NFA, with new states, for every occurrence of ϵ or some a as a subexpression of r.

INDUCTION: Suppose $N(s)$ and $N(t)$ are NFA's for regular expressions s and t, respectively.

a) Suppose $r = s|t$. Then $N(r)$, the NFA for r, is constructed as in Fig. 3.40. Here, i and f are new states, the start and accepting states of $N(r)$, respectively. There are ϵ-transitions from i to the start states of $N(s)$ and $N(t)$, and each of their accepting states have ϵ-transitions to the accepting state f. Note that the accepting states of $N(s)$ and $N(t)$ are not accepting in $N(r)$. Since any path from i to f must pass through either $N(s)$ or $N(t)$ exclusively, and since the label of that path is not changed by the ϵ's leaving i or entering f, we conclude that $N(r)$ accepts $L(s) \cup L(t)$, which is the same as $L(r)$. That is, Fig. 3.40 is a correct construction for the union operator.

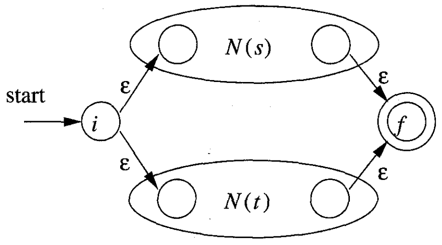


Figure 3.40: NFA for the union of two regular expressions

b) Suppose $r = st$. Then construct $N(r)$ as in Fig. 3.41. The start state of $N(s)$ becomes the start state of $N(r)$, and the accepting state of $N(t)$ is the only accepting state of $N(r)$. The accepting state of $N(s)$ and the start state of $N(t)$ are merged into a single state, with all the transitions in or out of either state. A path from i to f in Fig. 3.41 must go first through $N(s)$, and therefore its label will begin with some string in $L(s)$. The path then continues through $N(t)$, so the path's label finishes with a string in $L(t)$. As we shall soon argue, accepting states never have edges out and start states never have edges in, so it is not possible for a path to re-enter $N(s)$ after leaving it. Thus, $N(r)$ accepts exactly $L(s)L(t)$, and is a correct NFA for $r = st$.

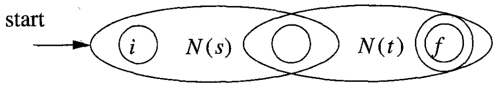


Figure 3.41: NFA for the concatenation of two regular expressions

c) Suppose $r = s^*$. Then for r we construct the NFA $N(r)$ shown in Fig. 3.42. Here, i and f are new states, the start state and lone accepting state of $N(r)$. To get from i to f, we can either follow the introduced path labeled ϵ, which takes care of the one string in $L(s)^0$, or we can go to the start state of $N(s)$, through that NFA, then from its accepting state back to its start state zero or more times. These options allow $N(r)$ to accept all the strings in $L(s)^1$, $L(s)^2$, and so on, so the entire set of strings accepted by $N(r)$ is $L(s^*)$.

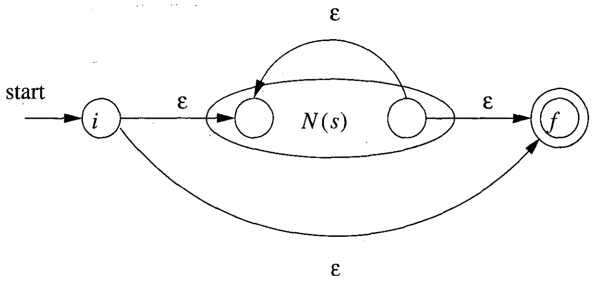


Figure 3.42: NFA for the closure of a regular expression

d) Finally, suppose $r = (s)$. Then $L(r) = L(s)$, and we can use the NFA $N(s)$ as $N(r)$.

□

The method description in Algorithm 3.23 contains hints as to why the inductive construction works as it should. We shall not give a formal correctness proof, but we shall list several properties of the constructed NFA's, in addition to the all-important fact that $N(r)$ accepts language $L(r)$. These properties are interesting in their own right, and helpful in making a formal proof.

1. $N(r)$ has at most twice as many states as there are operators and operands in r. This bound follows from the fact that each step of the algorithm creates at most two new states.

2. $N(r)$ has one start state and one accepting state. The accepting state has no outgoing transitions, and the start state has no incoming transitions.

3. Each state of $N(r)$ other than the accepting state has either one outgoing transition on a symbol in Σ or two outgoing transitions, both on ϵ.

Example 3.24: Let us use Algorithm 3.23 to construct an NFA for $r = (\mathbf{a}|\mathbf{b})^*\mathbf{abb}$. Figure 3.43 shows a parse tree for r that is analogous to the parse trees constructed for arithmetic expressions in Section 2.2.3. For subexpression r_1, the first **a**, we construct the NFA:

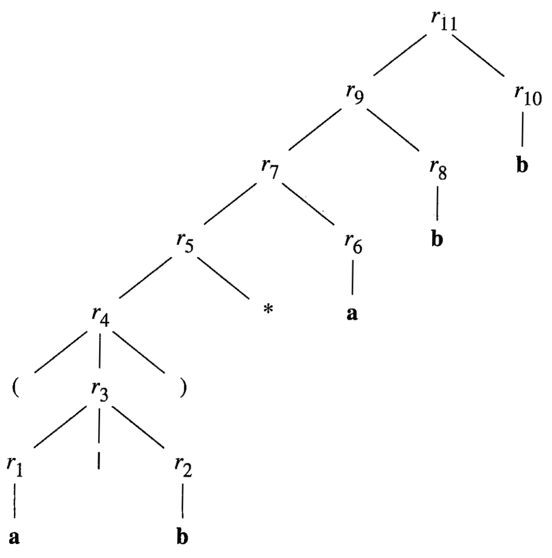
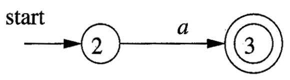
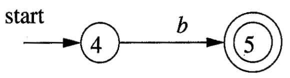


Figure 3.43: Parse tree for $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$



State numbers have been chosen for consistency with what follows. For r_2 we construct:



We can now combine $N(r_1)$ and $N(r_2)$, using the construction of Fig. 3.40 to obtain the NFA for $r_3 = r_1|r_2$; this NFA is shown in Fig. 3.44.

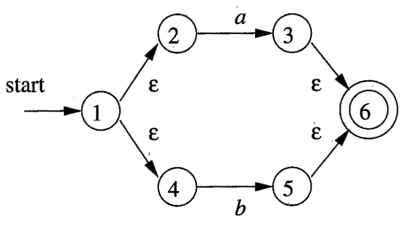


Figure 3.44: NFA for r_3

The NFA for $r_4 = (r_3)$ is the same as that for r_3. The NFA for $r_5 = (r_3)^*$ is then as shown in Fig. 3.45. We have used the construction in Fig. 3.42 to build this NFA from the NFA in Fig. 3.44.

Now, consider subexpression r_6, which is another **a**. We use the basis construction for a again, but we must use new states. It is not permissible to reuse

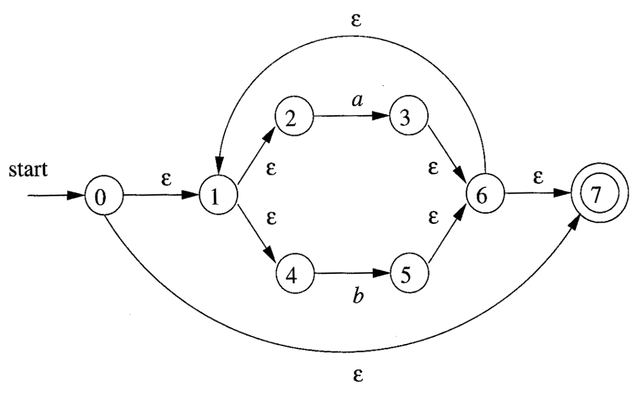
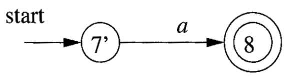


Figure 3.45: NFA for r_5

the NFA we constructed for r_1, even though r_1 and r_6 are the same expression. The NFA for r_6 is:



To obtain the NFA for $r_7 = r_5r_6$, we apply the construction of Fig. 3.41. We merge states 7 and $7'$, yielding the NFA of Fig. 3.46. Continuing in this fashion with new NFA's for the two subexpressions **b** called r_8 and r_{10}, we eventually construct the NFA for (**a**|**b**)***abb** that we first met in Fig. 3.34. □

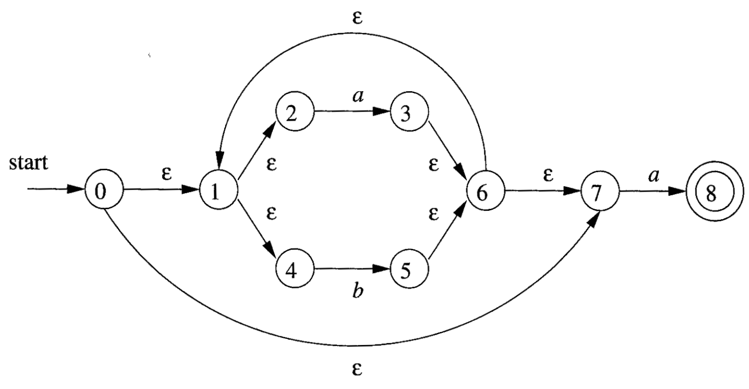


Figure 3.46: NFA for r_7

3.7.5 Efficiency of String-Processing Algorithms

We observed that Algorithm 3.18 processes a string x in time $O(|x|)$, while in Section 3.7.3 we concluded that we could simulate an NFA in time proportional to the product of $|x|$ and the size of the NFA's transition graph. Obviously, it is faster to have a DFA to simulate than an NFA, so we might wonder whether it ever makes sense to simulate an NFA.

One issue that may favor an NFA is that the subset construction can, in the worst case, exponentiate the number of states. While in principle, the number of DFA states does not influence the running time of Algorithm 3.18, should the number of states become so large that the transition table does not fit in main memory, then the true running time would have to include disk I/O and therefore rise noticeably.

Example 3.25: Consider the family of languages described by regular expressions of the form $L_n = (\mathbf{a}|\mathbf{b})^*\mathbf{a}(\mathbf{a}|\mathbf{b})^{n-1}$, that is, each language L_n consists of strings of a's and b's such that the nth character to the left of the right end holds a. An $n+1$-state NFA is easy to construct. It stays in its initial state under any input, but also has the option, on input a, of going to state 1. From state 1, it goes to state 2 on any input, and so on, until in state n it accepts. Figure 3.47 suggests this NFA.

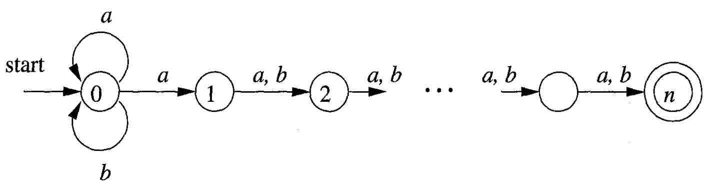


Figure 3.47: An NFA that has many fewer states than the smallest equivalent DFA

However, any DFA for the language L_n must have at least 2^n states. We shall not prove this fact, but the idea is that if two strings of length n can get the DFA to the same state, then we can exploit the last position where the strings differ (and therefore one must have a, the other b) to continue the strings identically, until they are the same in the last $n-1$ positions. The DFA will then be in a state where it must both accept and not accept. Fortunately, as we mentioned, it is rare for lexical analysis to involve patterns of this type, and we do not expect to encounter DFA's with outlandish numbers of states in practice. □

However, lexical-analyzer generators and other string-processing systems often start with a regular expression. We are faced with a choice of converting the regular expression to an NFA or DFA. The additional cost of going to a DFA is thus the cost of executing Algorithm 3.23 on the NFA (one could go directly from a regular expression to a DFA, but the work is essentially the same). If the string-processor is one that will be executed many times, as is the case for lexical analysis, then any cost of converting to a DFA is worthwhile. However, in other string-processing applications, such as `grep`, where the user specifies one regular expression and one or several files to be searched for the pattern of that expression, it may be more efficient to skip the step of constructing a DFA, and simulate the NFA directly.

Let us consider the cost of converting a regular expression r to an NFA by Algorithm 3.23. A key step is constructing the parse tree for r. In Chapter 4 we shall see several methods that are capable of constructing this parse tree in linear time, that is, in time $O(|r|)$, where $|r|$ stands for the *size* of r — the sum of the number of operators and operands in r. It is also easy to check that each of the basis and inductive constructions of Algorithm 3.23 takes constant time, so the entire time spent by the conversion to an NFA is $O(|r|)$.

Moreover, as we observed in Section 3.7.4, the NFA we construct has at most $|r|$ states and at most $2|r|$ transitions. That is, in terms of the analysis in Section 3.7.3, we have $n \leq |r|$ and $m \leq 2|r|$. Thus, simulating this NFA on an input string x takes time $O(|r| \times |x|)$. This time dominates the time taken by the NFA construction, which is $O(|r|)$, and therefore, we conclude that it is possible to take a regular expression r and string x, and tell whether x is in $L(r)$ in time $O(|r| \times |x|)$.

The time taken by the subset construction is highly dependent on the number of states the resulting DFA has. To begin, notice that in the subset construction of Fig. 3.32, the key step, the construction of a set of states U from a set of states T and an input symbol a, is very much like the construction of a new set of states from the old set of states in the NFA simulation of Algorithm 3.22. We already concluded that, properly implemented, this step takes time at most proportional to the number of states and transitions of the NFA.

Suppose we start with a regular expression r and convert it to an NFA. This NFA has at most $|r|$ states and at most $2|r|$ transitions. Moreover, there are at most $|r|$ input symbols. Thus, for every DFA state constructed, we must construct at most $|r|$ new states, and each one takes at most $O(|r| + 2|r|)$ time. The time to construct a DFA of s states is thus $O(|r|^2 s)$.

In the common case where s is about $|r|$, the subset construction takes time $O(|r|^3)$. However, in the worst case, as in Example 3.25, this time is $O(|r|^2 2^{|r|})$. Figure 3.48 summarizes the options when one is given a regular expression r and wants to produce a recognizer that will tell whether one or more strings x are in $L(r)$.

AUTOMATON	INITIAL	PER STRING
NFA	$O(\|r\|)$	$O(\|r\| \times \|x\|)$
DFA typical case	$O(\|r\|^3)$	$O(\|x\|)$
DFA worst case	$O(\|r\|^2 2^{\|r\|})$	$O(\|x\|)$

Figure 3.48: Initial cost and per-string-cost of various methods of recognizing the language of a regular expression

If the per-string cost dominates, as it does when we build a lexical analyzer, we clearly prefer the DFA. However, in commands like `grep`, where we run the automaton on only one string, we generally prefer the NFA. It is not until $|x|$ approaches $|r|^3$ that we would even think about converting to a DFA.

There is, however, a mixed strategy that is about as good as the better of the NFA and the DFA strategy for each expression r and string x. Start off simulating the NFA, but remember the sets of NFA states (i.e., the DFA states) and their transitions, as we compute them. Before processing the current set of NFA states and the current input symbol, check to see whether we have already computed this transition, and use the information if so.

3.7.6 Exercises for Section 3.7

Exercise 3.7.1: Convert to DFA's the NFA's of:

a) Fig. 3.26.

b) Fig. 3.29.

c) Fig. 3.30.

Exercise 3.7.2: use Algorithm 3.22 to simulate the NFA's:

a) Fig. 3.29.

b) Fig. 3.30.

on input *aabb*.

Exercise 3.7.3: Convert the following regular expressions to deterministic finite automata, using algorithms 3.23 and 3.20:

a) $(\mathbf{a}|\mathbf{b})^*$.

b) $(\mathbf{a}^*|\mathbf{b}^*)^*$.

c) $\left((\epsilon|\mathbf{a})\mathbf{b}^*\right)^*$.

d) $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}(\mathbf{a}|\mathbf{b})^*$.

3.8 Design of a Lexical-Analyzer Generator

In this section we shall apply the techniques presented in Section 3.7 to see how a lexical-analyzer generator such as `Lex` is architected. We discuss two approaches, based on NFA's and DFA's; the latter is essentially the implementation of `Lex`.

3.8.1 The Structure of the Generated Analyzer

Figure 3.49 overviews the architecture of a lexical analyzer generated by `Lex`. The program that serves as the lexical analyzer includes a fixed program that simulates an automaton; at this point we leave open whether that automaton is deterministic or nondeterministic. The rest of the lexical analyzer consists of components that are created from the `Lex` program by `Lex` itself.

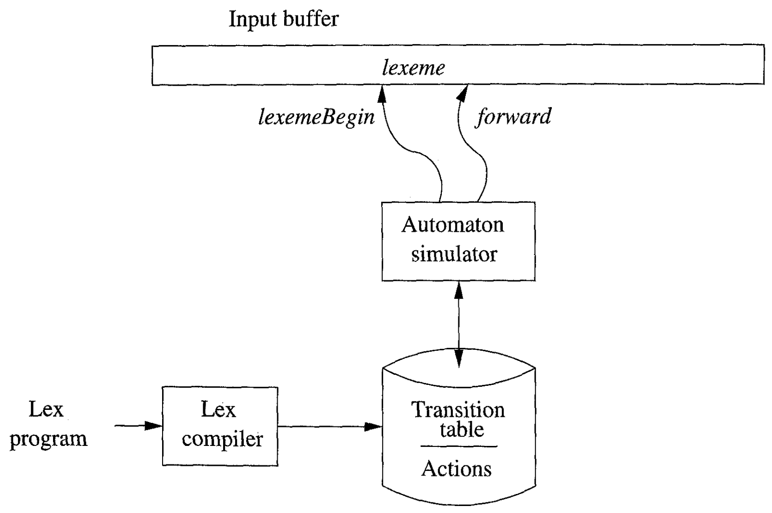


Figure 3.49: A `Lex` program is turned into a transition table and actions, which are used by a finite-automaton simulator

These components are:

1. A transition table for the automaton.
2. Those functions that are passed directly through `Lex` to the output (see Section 3.5.2).
3. The actions from the input program, which appear as fragments of code to be invoked at the appropriate time by the automaton simulator.

To construct the automaton, we begin by taking each regular-expression pattern in the `Lex` program and converting it, using Algorithm 3.23, to an NFA. We need a single automaton that will recognize lexemes matching any of the patterns in the program, so we combine all the NFA's into one by introducing a new start state with ϵ-transitions to each of the start states of the NFA's N_i for pattern p_i. This construction is shown in Fig. 3.50.

Example 3.26: We shall illustrate the ideas of this section with the following simple, abstract example:

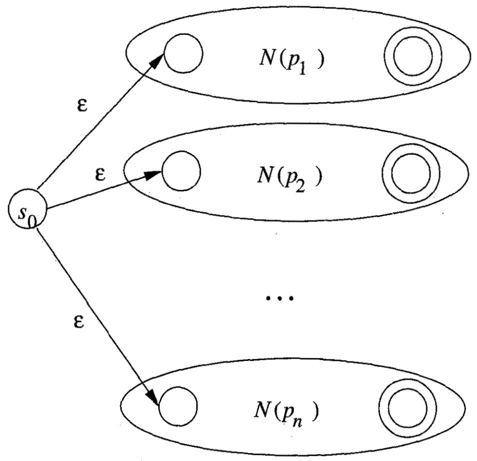


Figure 3.50: An NFA constructed from a Lex program

a	{ action A_1 for pattern p_1 }
abb	{ action A_2 for pattern p_2 }
a*b$^+$	{ action A_3 for pattern p_3 }

Note that these three patterns present some conflicts of the type discussed in Section 3.5.3. In particular, string *abb* matches both the second and third patterns, but we shall consider it a lexeme for pattern p_2, since that pattern is listed first in the above Lex program. Then, input strings such as $aabbb\cdots$ have many prefixes that match the third pattern. The Lex rule is to take the longest, so we continue reading b's, until another a is met, whereupon we report the lexeme to be the initial a's followed by as many b's as there are.

Figure 3.51 shows three NFA's that recognize the three patterns. The third is a simplification of what would come out of Algorithm 3.23. Then, Fig. 3.52 shows these three NFA's combined into a single NFA by the addition of start state 0 and three ϵ-transitions. □

3.8.2 Pattern Matching Based on NFA's

If the lexical analyzer simulates an NFA such as that of Fig. 3.52, then it must read input beginning at the point on its input which we have referred to as *lexemeBegin*. As it moves the pointer called *forward* ahead in the input, it calculates the set of states it is in at each point, following Algorithm 3.22.

Eventually, the NFA simulation reaches a point on the input where there are no next states. At that point, there is no hope that any longer prefix of the input would ever get the NFA to an accepting state; rather, the set of states will always be empty. Thus, we are ready to decide on the longest prefix that is a lexeme matching some pattern.

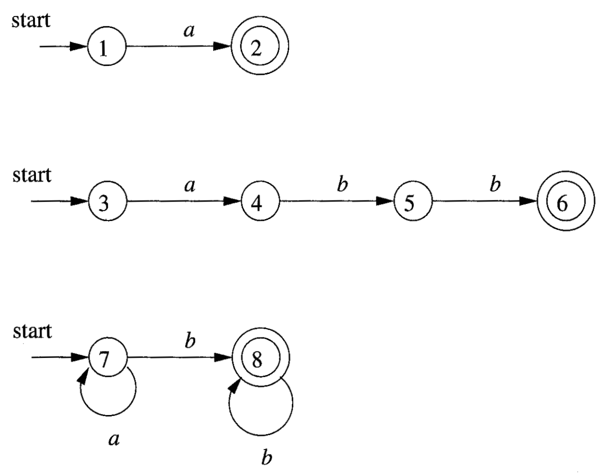


Figure 3.51: NFA's for **a**, **abb**, and $\mathbf{a}^*\mathbf{b}^+$

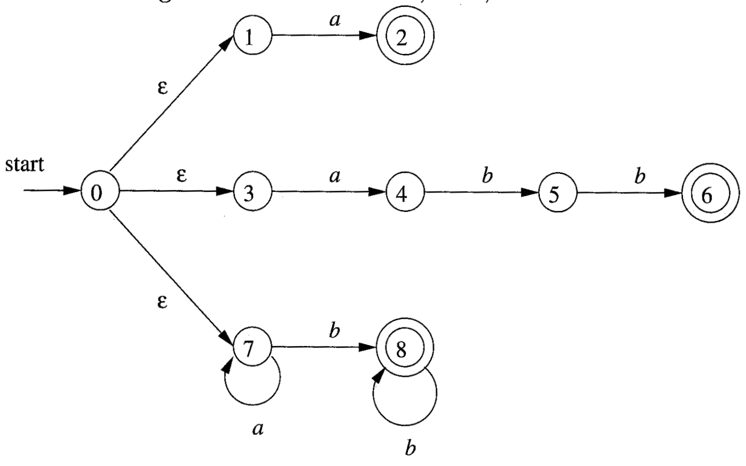


Figure 3.52: Combined NFA

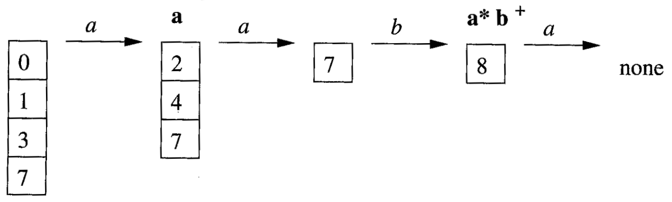


Figure 3.53: Sequence of sets of states entered when processing input *aaba*

We look backwards in the sequence of sets of states, until we find a set that includes one or more accepting states. If there are several accepting states in that set, pick the one associated with the earliest pattern p_i in the list from the Lex program. Move the *forward* pointer back to the end of the lexeme, and perform the action A_i associated with pattern p_i.

Example 3.27: Suppose we have the patterns of Example 3.26 and the input begins *aaba*. Figure 3.53 shows the sets of states of the NFA of Fig. 3.52 that we enter, starting with ϵ-*closure* of the initial state 0, which is $\{0, 1, 3, 7\}$, and proceeding from there. After reading the fourth input symbol, we are in an empty set of states, since in Fig. 3.52, there are no transitions out of state 8 on input a.

Thus, we need to back up, looking for a set of states that includes an accepting state. Notice that, as indicated in Fig. 3.53, after reading a we are in a set that includes state 2 and therefore indicates that the pattern **a** has been matched. However, after reading *aab*, we are in state 8, which indicates that $\mathbf{a}^*\mathbf{b}^+$ has been matched; prefix *aab* is the longest prefix that gets us to an accepting state. We therefore select *aab* as the lexeme, and execute action A_3, which should include a return to the parser indicating that the token whose pattern is $p_3 = \mathbf{a}^*\mathbf{b}^+$ has been found. □

3.8.3 DFA's for Lexical Analyzers

Another architecture, resembling the output of Lex, is to convert the NFA for all the patterns into an equivalent DFA, using the subset construction of Algorithm 3.20. Within each DFA state, if there are one or more accepting NFA states, determine the first pattern whose accepting state is represented, and make that pattern the output of the DFA state.

Example 3.28: Figure 3.54 shows a transition diagram based on the DFA that is constructed by the subset construction from the NFA in Fig. 3.52. The accepting states are labeled by the pattern that is identified by that state. For instance, the state $\{6, 8\}$ has two accepting states, corresponding to patterns **abb** and $\mathbf{a}^*\mathbf{b}^+$. Since the former is listed first, that is the pattern associated with state $\{6, 8\}$. □

We use the DFA in a lexical analyzer much as we did the NFA. We simulate the DFA until at some point there is no next state (or strictly speaking, the next state is $\emptyset$, the *dead state* corresponding to the empty set of NFA states). At that point, we back up through the sequence of states we entered and, as soon as we meet an accepting DFA state, we perform the action associated with the pattern for that state.

Example 3.29: Suppose the DFA of Fig. 3.54 is given input *abba*. The sequence of states entered is $0137, 247, 58, 68$, and at the final a there is no transition out of state 68. Thus, we consider the sequence from the end, and in this case, 68 itself is an accepting state that reports pattern $p_2 = \mathbf{abb}$. □

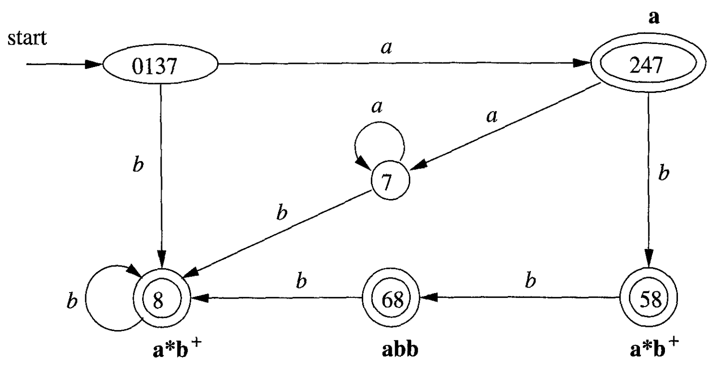


Figure 3.54: Transition graph for DFA handling the patterns **a**, **abb**, and **a*****b**$^+$

3.8.4 Implementing the Lookahead Operator

Recall from Section 3.5.4 that the `Lex` lookahead operator / in a `Lex` pattern r_1/r_2 is sometimes necessary, because the pattern r_1 for a particular token may need to describe some trailing context r_2 in order to correctly identify the actual lexeme. When converting the pattern r_1/r_2 to an NFA, we treat the / as if it were ϵ, so we do not actually look for a / on the input. However, if the NFA recognizes a prefix xy of the input buffer as matching this regular expression, the end of the lexeme is not where the NFA entered its accepting state. Rather the end occurs when the NFA enters a state s such that

1. s has an ϵ-transition on the (imaginary) /,

2. There is a path from the start state of the NFA to state s that spells out x.

3. There is a path from state s to the accepting state that spells out y.

4. x is as long as possible for any xy satisfying conditions 1-3.

If there is only one ϵ-transition state on the imaginary / in the NFA, then the end of the lexeme occurs when this state is entered for the last time as the following example illustrates. If the NFA has more than one ϵ-transition state on the imaginary /, then the general problem of finding the correct state s is much more difficult.

Example 3.30: An NFA for the pattern for the Fortran `IF` with lookahead, from Example 3.13, is shown in Fig. 3.55. Notice that the ϵ-transition from state 2 to state 3 represents the lookahead operator. State 6 indicates the presence of the keyword `IF`. However, we find the lexeme `IF` by scanning backwards to the last occurrence of state 2, whenever state 6 is entered. □

Dead States in DFA's

Technically, the automaton in Fig. 3.54 is not quite a DFA. The reason is that a DFA has a transition from every state on every input symbol in its input alphabet. Here, we have omitted transitions to the dead state $\emptyset$, and we have therefore omitted the transitions from the dead state to itself on every input. Previous NFA-to-DFA examples did not have a way to get from the start state to $\emptyset$, but the NFA of Fig. 3.52 does.

However, when we construct a DFA for use in a lexical analyzer, it is important that we treat the dead state differently, since we must know when there is no longer any possibility of recognizing a longer lexeme. Thus, we suggest always omitting transitions to the dead state and eliminating the dead state itself. In fact, the problem is harder than it appears, since an NFA-to-DFA construction may yield several states that cannot reach any accepting state, and we must know when any of these states have been reached. Section 3.9.6 discusses how to combine all these states into one dead state, so their identification becomes easy. It is also interesting to note that if we construct a DFA from a regular expression using Algorithms 3.20 and 3.23, then the DFA will not have any states besides $\emptyset$ that cannot lead to an accepting state.

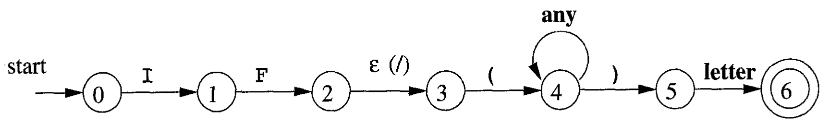


Figure 3.55: NFA recognizing the keyword `IF`

3.8.5 Exercises for Section 3.8

Exercise 3.8.1: Suppose we have two tokens: (1) the keyword `if`, and (2) identifiers, which are strings of letters other than `if`. Show:

a) The NFA for these tokens, and

b) The DFA for these tokens.

Exercise 3.8.2: Repeat Exercise 3.8.1 for tokens consisting of (1) the keyword `while`, (2) the keyword `when`, and (3) identifiers consisting of strings of letters and digits, beginning with a letter.

! **Exercise 3.8.3:** Suppose we were to revise the definition of a DFA to allow zero or one transition out of each state on each input symbol (rather than exactly one such transition, as in the standard DFA definition). Some regular expressions would then have smaller "DFA's" than they do under the standard definition of a DFA. Give an example of one such regular expression.

!! **Exercise 3.8.4:** Design an algorithm to recognize Lex-lookahead patterns of the form r_1/r_2, where r_1 and r_2 are regular expressions. Show how your algorithm works on the following inputs:

a) $(abcd|abc)/d$

b) $(a|ab)/ba$

c) $aa*/a*$

3.9 Optimization of DFA-Based Pattern Matchers

In this section we present three algorithms that have been used to implement and optimize pattern matchers constructed from regular expressions.

1. The first algorithm is useful in a Lex compiler, because it constructs a DFA directly from a regular expression, without constructing an intermediate NFA. The resulting DFA also may have fewer states than the DFA constructed via an NFA.

2. The second algorithm minimizes the number of states of any DFA, by combining states that have the same future behavior. The algorithm itself is quite efficient, running in time $O(n \log n)$, where n is the number of states of the DFA.

3. The third algorithm produces more compact representations of transition tables than the standard, two-dimensional table.

3.9.1 Important States of an NFA

To begin our discussion of how to go directly from a regular expression to a DFA, we must first dissect the NFA construction of Algorithm 3.23 and consider the roles played by various states. We call a state of an NFA *important* if it has a non-ϵ out-transition. Notice that the subset construction (Algorithm 3.20) uses only the important states in a set T when it computes $\epsilon\text{-}closure(move(T, a))$, the set of states reachable from T on input a. That is, the set of states $move(s, a)$ is nonempty only if state s is important. During the subset construction, two sets of NFA states can be identified (treated as if they were the same set) if they:

1. Have the same important states, and

2. Either both have accepting states or neither does.

When the NFA is constructed from a regular expression by Algorithm 3.23, we can say more about the important states. The only important states are those introduced as initial states in the basis part for a particular symbol position in the regular expression. That is, each important state corresponds to a particular operand in the regular expression.

The constructed NFA has only one accepting state, but this state, having no out-transitions, is not an important state. By concatenating a unique right endmarker # to a regular expression r, we give the accepting state for r a transition on #, making it an important state of the NFA for $(r)\#$. In other words, by using the *augmented* regular expression $(r)\#$, we can forget about accepting states as the subset construction proceeds; when the construction is complete, any state with a transition on # must be an accepting state.

The important states of the NFA correspond directly to the positions in the regular expression that hold symbols of the alphabet. It is useful, as we shall see, to present the regular expression by its *syntax tree*, where the leaves correspond to operands and the interior nodes correspond to operators. An interior node is called a *cat-node*, *or-node*, or *star-node* if it is labeled by the concatenation operator (dot), union operator |, or star operator $*$, respectively. We can construct a syntax tree for a regular expression just as we did for arithmetic expressions in Section 2.5.1.

Example 3.31: Figure 3.56 shows the syntax tree for the regular expression of our running example. Cat-nodes are represented by circles. □

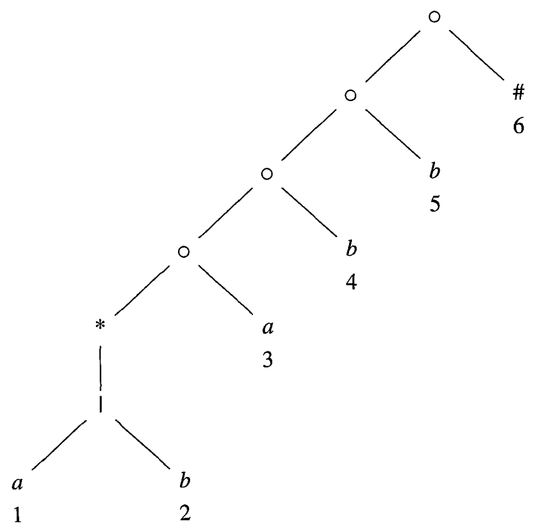


Figure 3.56: Syntax tree for $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}\#$

Leaves in a syntax tree are labeled by ϵ or by an alphabet symbol. To each leaf not labeled ϵ, we attach a unique integer. We refer to this integer as the *position* of the leaf and also as a position of its symbol. Note that a symbol can have several positions; for instance, a has positions 1 and 3 in Fig. 3.56. The positions in the syntax tree correspond to the important states of the constructed NFA.

Example 3.32: Figure 3.57 shows the NFA for the same regular expression as Fig. 3.56, with the important states numbered and other states represented by letters. The numbered states in the NFA and the positions in the syntax tree correspond in a way we shall soon see. □

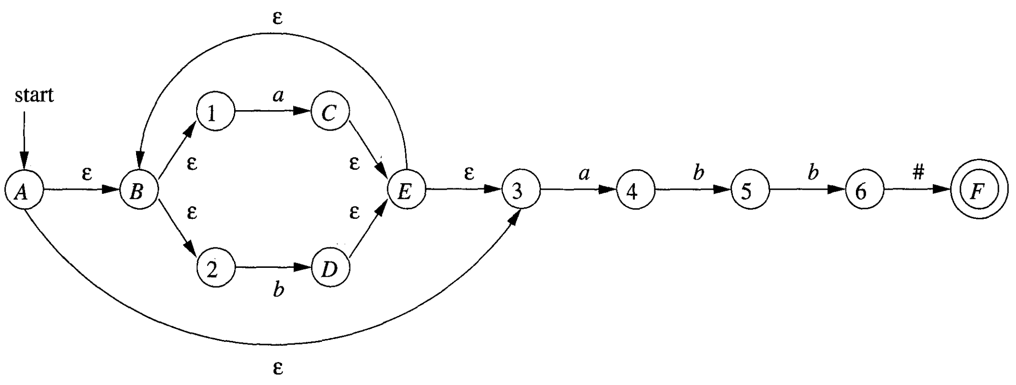


Figure 3.57: NFA constructed by Algorithm 3.23 for $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}\#$

3.9.2 Functions Computed From the Syntax Tree

To construct a DFA directly from a regular expression, we construct its syntax tree and then compute four functions: *nullable*, *firstpos*, *lastpos*, and *followpos*, defined as follows. Each definition refers to the syntax tree for a particular augmented regular expression $(r)\#$.

1. *nullable*(n) is true for a syntax-tree node n if and only if the subexpression represented by n has ϵ in its language. That is, the subexpression can be "made null" or the empty string, even though there may be other strings it can represent as well.

2. *firstpos*(n) is the set of positions in the subtree rooted at n that correspond to the first symbol of at least one string in the language of the subexpression rooted at n.

3. *lastpos*(n) is the set of positions in the subtree rooted at n that correspond to the last symbol of at least one string in the language of the subexpression rooted at n.

4. *followpos*(p), for a position p, is the set of positions q in the entire syntax tree such that there is some string $x = a_1a_2 \cdots a_n$ in $L\big((r)\#\big)$ such that for some i, there is a way to explain the membership of x in $L\big((r)\#\big)$ by matching a_i to position p of the syntax tree and a_{i+1} to position q.

Example 3.33: Consider the cat-node n in Fig. 3.56 that corresponds to the expression $(\mathbf{a}|\mathbf{b})^*\mathbf{a}$. We claim *nullable*$(n)$ is false, since this node generates all strings of a's and b's ending in an a; it does not generate ϵ. On the other hand, the star-node below it is nullable; it generates ϵ along with all other strings of a's and b's.

firstpos$(n) = \{1,2,3\}$. In a typical generated string like aa, the first position of the string corresponds to position 1 of the tree, and in a string like ba, the first position of the string comes from position 2 of the tree. However, when the string generated by the expression of node n is just a, then this a comes from position 3.

lastpos$(n) = \{3\}$. That is, no matter what string is generated from the expression of node n, the last position is the a from position 3 of the tree.

followpos is trickier to compute, but we shall see the rules for doing so shortly. Here is an example of the reasoning: *followpos*$(1) = \{1,2,3\}$. Consider a string $\cdots ac \cdots$, where the c is either a or b, and the a comes from position 1. That is, this a is one of those generated by the $\mathbf{a}$ in expression $(\mathbf{a}|\mathbf{b})^*$. This a could be followed by another a or b coming from the same subexpression, in which case c comes from position 1 or 2. It is also possible that this a is the last in the string generated by $(\mathbf{a}|\mathbf{b})^*$, in which case the symbol c must be the a that comes from position 3. Thus, 1, 2, and 3 are exactly the positions that can follow position 1. □

3.9.3 Computing *nullable*, *firstpos*, and *lastpos*

We can compute *nullable*, *firstpos*, and *lastpos* by a straightforward recursion on the height of the tree. The basis and inductive rules for *nullable* and *firstpos* are summarized in Fig. 3.58. The rules for *lastpos* are essentially the same as for *firstpos*, but the roles of children c_1 and c_2 must be swapped in the rule for a cat-node.

Example 3.34: Of all the nodes in Fig. 3.56 only the star-node is nullable. We note from the table of Fig. 3.58 that none of the leaves are nullable, because they each correspond to non-ϵ operands. The or-node is not nullable, because neither of its children is. The star-node is nullable, because every star-node is nullable. Finally, each of the cat-nodes, having at least one nonnullable child, is not nullable.

The computation of *firstpos* and *lastpos* for each of the nodes is shown in Fig. 3.59, with *firstpos*(n) to the left of node n, and *lastpos*(n) to its right. Each of the leaves has only itself for *firstpos* and *lastpos*, as required by the rule for non-ϵ leaves in Fig. 3.58. For the or-node, we take the union of *firstpos* at the

NODE n	*nullable*(n)	*firstpos*(n)
A leaf labeled ϵ	**true**	$\emptyset$
A leaf with position i	**false**	$\{i\}$
An or-node $n = c_1 \| c_2$	*nullable*(c_1) **or** *nullable*(c_2)	*firstpos*$(c_1) \cup$ *firstpos*(c_2)
A cat-node $n = c_1c_2$	*nullable*(c_1) **and** *nullable*(c_2)	**if** (*nullable*(c_1)) *firstpos*$(c_1) \cup$ *firstpos*(c_2) **else** *firstpos*(c_1)
A star-node $n = c_1{}^*$	**true**	*firstpos*(c_1)

Figure 3.58: Rules for computing *nullable* and *firstpos*

children and do the same for *lastpos*. The rule for the star-node says that we take the value of *firstpos* or *lastpos* at the one child of that node.

Now, consider the lowest cat-node, which we shall call n. To compute *firstpos*(n), we first consider whether the left operand is nullable, which it is in this case. Therefore, *firstpos* for n is the union of *firstpos* for each of its children, that is $\{1, 2\} \cup \{3\} = \{1, 2, 3\}$. The rule for *lastpos* does not appear explicitly in Fig. 3.58, but as we mentioned, the rules are the same as for *firstpos*, with the children interchanged. That is, to compute *lastpos*(n) we must ask whether its right child (the leaf with position 3) is nullable, which it is not. Therefore, *lastpos*(n) is the same as *lastpos* of the right child, or $\{3\}$. $\square$

3.9.4 Computing *followpos*

Finally, we need to see how to compute *followpos*. There are only two ways that a position of a regular expression can be made to follow another.

1. If n is a cat-node with left child c_1 and right child c_2, then for every position i in *lastpos*(c_1), all positions in *firstpos*(c_2) are in *followpos*(i).

2. If n is a star-node, and i is a position in *lastpos*(n), then all positions in *firstpos*(n) are in *followpos*(i).

Example 3.35: Let us continue with our running example; recall that *firstpos* and *lastpos* were computed in Fig. 3.59. Rule 1 for *followpos* requires that we look at each cat-node, and put each position in *firstpos* of its right child in *followpos* for each position in *lastpos* of its left child. For the lowest cat-node in Fig. 3.59, that rule says position 3 is in *followpos*(1) and *followpos*(2). The next cat-node above says that 4 is in *followpos*(3), and the remaining two cat-nodes give us 5 in *followpos*(4) and 6 in *followpos*(5).

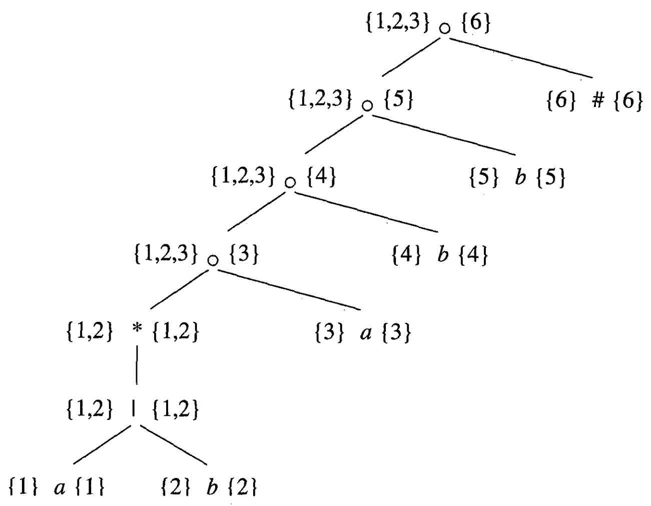


Figure 3.59: *firstpos* and *lastpos* for nodes in the syntax tree for **(a|b)*abb#**

We must also apply rule 2 to the star-node. That rule tells us positions 1 and 2 are in both *followpos*(1) and *followpos*(2), since both *firstpos* and *lastpos* for this node are $\{1, 2\}$. The complete sets *followpos* are summarized in Fig. 3.60. □

NODE n	*followpos*(n)
1	$\{1, 2, 3\}$
2	$\{1, 2, 3\}$
3	$\{4\}$
4	$\{5\}$
5	$\{6\}$
6	$\emptyset$

Figure 3.60: The function *followpos*

We can represent the function *followpos* by creating a directed graph with a node for each position and an arc from position i to position j if and only if j is in *followpos*(i). Figure 3.61 shows this graph for the function of Fig. 3.60.

It should come as no surprise that the graph for *followpos* is almost an NFA without ϵ-transitions for the underlying regular expression, and would become one if we:

1. Make all positions in *firstpos* of the root be initial states,

2. Label each arc from i to j by the symbol at position i, and

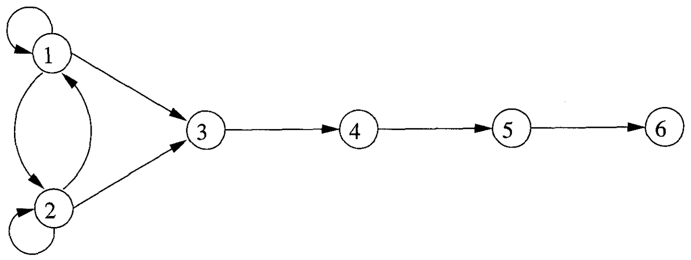


Figure 3.61: Directed graph for the function *followpos*

3. Make the position associated with endmarker # be the only accepting state.

3.9.5 Converting a Regular Expression Directly to a DFA

Algorithm 3.36: Construction of a DFA from a regular expression r.

INPUT: A regular expression r.

OUTPUT: A DFA D that recognizes $L(r)$.

METHOD:

1. Construct a syntax tree T from the augmented regular expression $(r)\#$.

2. Compute *nullable*, *firstpos*, *lastpos*, and *followpos* for T, using the methods of Sections 3.9.3 and 3.9.4.

3. Construct *Dstates*, the set of states of DFA D, and *Dtran*, the transition function for D, by the procedure of Fig. 3.62. The states of D are sets of positions in T. Initially, each state is "unmarked," and a state becomes "marked" just before we consider its out-transitions. The start state of D is *firstpos*(n_0), where node n_0 is the root of T. The accepting states are those containing the position for the endmarker symbol #.

□

Example 3.37: We can now put together the steps of our running example to construct a DFA for the regular expression $r = (\mathbf{a}|\mathbf{b})^*\mathbf{abb}$. The syntax tree for $(r)\#$ appeared in Fig. 3.56. We observed that for this tree, *nullable* is true only for the star-node, and we exhibited *firstpos* and *lastpos* in Fig. 3.59. The values of *followpos* appear in Fig. 3.60.

The value of *firstpos* for the root of the tree is $\{1, 2, 3\}$, so this set is the start state of D. Call this set of states A. We must compute *Dtran*$[A, a]$ and *Dtran*$[A, b]$. Among the positions of A, 1 and 3 correspond to a, while 2 corresponds to b. Thus, *Dtran*$[A, a]$ = *followpos*$(1) \cup$ *followpos*$(3) = \{1, 2, 3, 4\}$,

```
initialize *Dstates* to contain only the unmarked state $firstpos(n_0)$,
        where $n_0$ is the root of syntax tree $T$ for $(r)\#$;
**while** ( there is an unmarked state $S$ in *Dstates* ) {
        mark $S$;
        **for** ( each input symbol $a$ ) {
                let $U$ be the union of $followpos(p)$ for all $p$
                        in $S$ that correspond to $a$;
                **if** ( $U$ is not in *Dstates* )
                        add $U$ as an unmarked state to *Dstates*;
                $Dtran[S, a] = U$;
        }
}
```

Figure 3.62: Construction of a DFA directly from a regular expression

and $Dtran[A, b] = followpos(2) = \{1, 2, 3\}$. The latter is state A, and so does not have to be added to *Dstates*, but the former, $B = \{1, 2, 3, 4\}$, is new, so we add it to *Dstates* and proceed to compute its transitions. The complete DFA is shown in Fig. 3.63. □

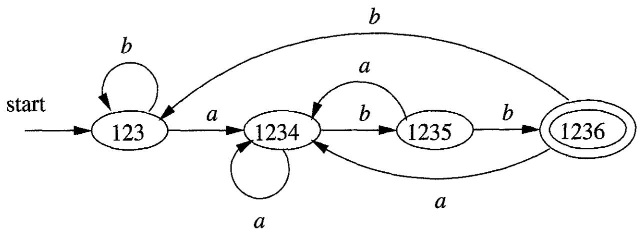


Figure 3.63: DFA constructed from Fig. 3.57

3.9.6 Minimizing the Number of States of a DFA

There can be many DFA's that recognize the same language. For instance, note that the DFA's of Figs. 3.36 and 3.63 both recognize language $L\big((\mathbf{a}|\mathbf{b})^*\mathbf{abb}\big)$. Not only do these automata have states with different names, but they don't even have the same number of states. If we implement a lexical analyzer as a DFA, we would generally prefer a DFA with as few states as possible, since each state requires entries in the table that describes the lexical analyzer.

The matter of the names of states is minor. We shall say that two automata are *the same up to state names* if one can be transformed into the other by doing nothing more than changing the names of states. Figures 3.36 and 3.63 are not the same up to state names. However, there is a close relationship between the states of each. States A and C of Fig. 3.36 are actually equivalent, in the sense that neither is an accepting state, and on any input they transfer to the same state — to B on input a and to C on input b. Moreover, both states A and C behave like state 123 of Fig. 3.63. Likewise, state B of Fig. 3.36 behaves like state 1234 of Fig. 3.63, state D behaves like state 1235, and state E behaves like state 1236.

It turns out that there is always a unique (up to state names) minimum state DFA for any regular language. Moreover, this minimum-state DFA can be constructed from any DFA for the same language by grouping sets of equivalent states. In the case of $L\big((\mathbf{a}|\mathbf{b})^*\mathbf{abb}\big)$, Fig. 3.63 is the minimum-state DFA, and it can be constructed by partitioning the states of Fig. 3.36 as $\{A, C\}\{B\}\{D\}\{E\}$.

In order to understand the algorithm for creating the partition of states that converts any DFA into its minimum-state equivalent DFA, we need to see how input strings distinguish states from one another. We say that string x *distinguishes* state s from state t if exactly one of the states reached from s and t by following the path with label x is an accepting state. State s is *distinguishable* from state t if there is some string that distinguishes them.

Example 3.38: The empty string distinguishes any accepting state from any nonaccepting state. In Fig. 3.36, the string bb distinguishes state A from state B, since bb takes A to a nonaccepting state C, but takes B to accepting state E. □

The state-minimization algorithm works by partitioning the states of a DFA into groups of states that cannot be distinguished. Each group of states is then merged into a single state of the minimum-state DFA. The algorithm works by maintaining a partition, whose groups are sets of states that have not yet been distinguished, while any two states from different groups are known to be distinguishable. When the partition cannot be refined further by breaking any group into smaller groups, we have the minimum-state DFA.

Initially, the partition consists of two groups: the accepting states and the nonaccepting states. The fundamental step is to take some group of the current partition, say $A = \{s_1, s_2, \ldots, s_k\}$, and some input symbol a, and see whether a can be used to distinguish between any states in group A. We examine the transitions from each of $s_1, s_2, \ldots, s_k$ on input a, and if the states reached fall into two or more groups of the current partition, we split A into a collection of groups, so that s_i and s_j are in the same group if and only if they go to the same group on input a. We repeat this process of splitting groups, until for no group, and for no input symbol, can the group be split further. The idea is formalized in the next algorithm.

Algorithm 3.39: Minimizing the number of states of a DFA.

INPUT: A DFA D with set of states S, input alphabet Σ, state state s_0, and set of accepting states F.

OUTPUT: A DFA D' accepting the same language as D and having as few states as possible.

Why the State-Minimization Algorithm Works

We need to prove two things: that states remaining in the same group in Π_{final} are indistinguishable by any string, and that states winding up in different groups are distinguishable. The first is an induction on i that if after the ith iteration of step (2) of Algorithm 3.39, s and t are in the same group, then there is no string of length i or less that distinguishes them. We shall leave the details of the induction to you.

The second is an induction on i that if states s and t are placed in different groups at the ith iteration of step (2), then there is a string that distinguishes them. The basis, when s and t are placed in different groups of the initial partition, is easy: one must be accepting and the other not, so ϵ distinguishes them. For the induction, there must be an input a and states p and q such that s and t go to states p and q, respectively, on input a. Moreover, p and q must already have been placed in different groups. Then by the inductive hypothesis, there is some string x that distinguishes p from q. Therefore, ax distinguishes s from t.

METHOD:

1. Start with an initial partition Π with two groups, F and $S - F$, the accepting and nonaccepting states of D.

2. Apply the procedure of Fig. 3.64 to construct a new partition Π_{new}.

```
initially, let $\Pi_{\text{new}} = \Pi$;
for ( each group $G$ of $\Pi$ ) {
        partition $G$ into subgroups such that two states $s$ and $t$
                are in the same subgroup if and only if for all
                input symbols $a$, states $s$ and $t$ have transitions on $a$
                to states in the same group of $\Pi$;
        /* at worst, a state will be in a subgroup by itself */
        replace $G$ in $\Pi_{\text{new}}$ by the set of all subgroups formed;
}
```

Figure 3.64: Construction of Π_{new}

3. If $\Pi_{\text{new}} = \Pi$, let $\Pi_{\text{final}} = \Pi$ and continue with step (4). Otherwise, repeat step (2) with Π_{new} in place of Π.

4. Choose one state in each group of Π_{final} as the *representative* for that group. The representatives will be the states of the minimum-state DFA D'. The other components of D' are constructed as follows:

Eliminating the Dead State

The minimization algorithm sometimes produces a DFA with one dead state — one that is not accepting and transfers to itself on each input symbol. This state is technically needed, because a DFA must have a transition from every state on every symbol. However, as discussed in Section 3.8.3, we often want to know when there is no longer any possibility of acceptance, so we can establish that the proper lexeme has already been seen. Thus, we may wish to eliminate the dead state and use an automaton that is missing some transitions. This automaton has one fewer state than the minimum-state DFA, but is strictly speaking not a DFA, because of the missing transitions to the dead state.

(a) The state state of D' is the representative of the group containing the start state of D.

(b) The accepting states of D' are the representatives of those groups that contain an accepting state of D. Note that each group contains either only accepting states, or only nonaccepting states, because we started by separating those two classes of states, and the procedure of Fig. 3.64 always forms new groups that are subgroups of previously constructed groups.

(c) Let s be the representative of some group G of Π_{final}, and let the transition of D from s on input a be to state t. Let r be the representative of t's group H. Then in D', there is a transition from s to r on input a. Note that in D, every state in group G must go to some state of group H on input a, or else, group G would have been split according to Fig. 3.64.

□

Example 3.40: Let us reconsider the DFA of Fig. 3.36. The initial partition consists of the two groups $\{A, B, C, D\}\{E\}$, which are respectively the nonaccepting states and the accepting states. To construct Π_{new}, the procedure of Fig. 3.64 considers both groups and inputs a and b. The group $\{E\}$ cannot be split, because it has only one state, so $\{E\}$ will remain intact in Π_{new}.

The other group $\{A, B, C, D\}$ can be split, so we must consider the effect of each input symbol. On input a, each of these states goes to state B, so there is no way to distinguish these states using strings that begin with a. On input b, states A, B, and C go to members of group $\{A, B, C, D\}$, while state D goes to E, a member of another group. Thus, in Π_{new}, group $\{A, B, C, D\}$ is split into $\{A, B, C\}\{D\}$, and Π_{new} for this round is $\{A, B, C\}\{D\}\{E\}$.

In the next round, we can split $\{A, B, C\}$ into $\{A, C\}\{B\}$, since A and C each go to a member of $\{A, B, C\}$ on input b, while B goes to a member of another group, $\{D\}$. Thus, after the second round, $\Pi_{\text{new}} = \{A, C\}\{B\}\{D\}\{E\}$. For the third round, we cannot split the one remaining group with more than one state, since A and C each go to the same state (and therefore to the same group) on each input. We conclude that $\Pi_{\text{final}} = \{A, C\}\{B\}\{D\}\{E\}$.

Now, we shall construct the minimum-state DFA. It has four states, corresponding to the four groups of Π_{final}, and let us pick A, B, D, and E as the representatives of these groups. The initial state is A, and the only accepting state is E. Figure 3.65 shows the transition function for the DFA. For instance, the transition from state E on input b is to A, since in the original DFA, E goes to C on input b, and A is the representative of C's group. For the same reason, the transition on b from state A is to A itself, while all other transitions are as in Fig. 3.36. □

STATE	a	b
A	B	A
B	B	D
D	B	E
E	B	A

Figure 3.65: Transition table of minimum-state DFA

3.9.7 State Minimization in Lexical Analyzers

To apply the state minimization procedure to the DFA's generated in Section 3.8.3, we must begin Algorithm 3.39 with the partition that groups together all states that recognize a particular token, and also places in one group all those states that do not indicate any token. An example should make the extension clear.

Example 3.41 : For the DFA of Fig. 3.54, the initial partition is

$$\{0137, 7\}\{247\}\{8, 58\}\{7\}\{68\}\{\emptyset\}$$

That is, states 0137 and 7 belong together because neither announces any token. States 8 and 58 belong together because they both announce token $\mathbf{a}^*\mathbf{b}^+$. Note that we have added a dead state $\emptyset$, which we suppose has transitions to itself on inputs a and b. The dead state is also the target of missing transitions on a from states 8, 58, and 68.

We must split 0137 from 7, because they go to different groups on input a. We also split 8 from 58, because they go to different groups on b. Thus, all states are in groups by themselves, and Fig. 3.54 is the minimum-state DFA recognizing its three tokens. Recall that a DFA serving as a lexical analyzer will normally drop the dead state, while we treat missing transitions as a signal to end token recognition. □

3.9.8 Trading Time for Space in DFA Simulation

The simplest and fastest way to represent the transition function of a DFA is a two-dimensional table indexed by states and characters. Given a state and next input character, we access the array to find the next state and any special action we must take, e.g., returning a token to the parser. Since a typical lexical analyzer has several hundred states in its DFA and involves the ASCII alphabet of 128 input characters, the array consumes less than a megabyte.

However, compilers are also appearing in very small devices, where even a megabyte of storage may be too much. For such situations, there are many methods that can be used to compact the transition table. For instance, we can represent each state by a list of transitions — that is, character-state pairs — ended by a default state that is to be chosen for any input character not on the list. If we choose as the default the most frequently occurring next state, we can often reduce the amount of storage needed by a large factor.

There is a more subtle data structure that allows us to combine the speed of array access with the compression of lists with defaults. We may think of this structure as four arrays, as suggested in Fig. 3.66.[5] The *base* array is used to determine the base location of the entries for state s, which are located in the *next* and *check* arrays. The *default* array is used to determine an alternative base location if the *check* array tells us the one given by *base*[s] is invalid.

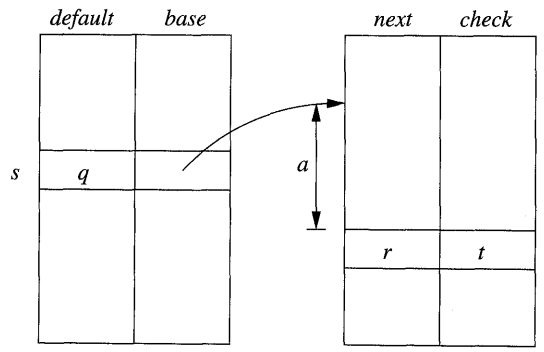


Figure 3.66: Data structure for representing transition tables

To compute *nextState*(s, a), the transition for state s on input a, we examine the *next* and *check* entries in location l = *base*[s]$+a$, where character a is treated as an integer, presumably in the range 0 to 127. If *check*[l] $= s$, then this entry

[5]In practice, there would be another array indexed by states to give the action associated with that state, if any.

is valid, and the next state for state s on input a is $next[l]$. If $check[l] \neq s$, then we determine another state $t = default[s]$ and repeat the process as if t were the current state. More formally, the function *nextState* is defined as follows:

```
int $nextState(s, a)$ {
        if ( $check[base[s] + a] = s$ ) return $next[base[s] + a]$;
        else return $nextState(default[s], a)$;
}
```

The intended use of the structure of Fig. 3.66 is to make the *next-check* arrays short by taking advantage of the similarities among states. For instance, state t, the default for state s, might be the state that says "we are working on an identifier," like state 10 in Fig. 3.14. Perhaps state s is entered after seeing the letters `th`, which are a prefix of keyword `then` as well as potentially being the prefix of some lexeme for an identifier. On input character `e`, we must go from state s to a special state that remembers we have seen `the`, but otherwise, state s behaves as t does. Thus, we set $check[base[s] + \texttt{e}]$ to s (to confirm that this entry is valid for s) and we set $next[base[s] + \texttt{e}]$ to the state that remembers `the`. Also, $default[s]$ is set to t.

While we may not be able to choose *base* values so that no *next-check* entries remain unused, experience has shown that the simple strategy of assigning *base* values to states in turn, and assigning each $base[s]$ value the lowest integer so that the special entries for state s are not previously occupied utilizes little more space than the minimum possible.

3.9.9 Exercises for Section 3.9

Exercise 3.9.1: Extend the table of Fig. 3.58 to include the operators (a) ? and (b) $^+$.

Exercise 3.9.2: Use Algorithm 3.36 to convert the regular expressions of Exercise 3.7.3 directly to deterministic finite automata.

! **Exercise 3.9.3:** We can prove that two regular expressions are equivalent by showing that their minimum-state DFA's are the same up to renaming of states. Show in this way that the following regular expressions: $(\mathbf{a}|\mathbf{b})^*$, $(\mathbf{a}^*|\mathbf{b}^*)^*$, and $\big((\epsilon|\mathbf{a})\mathbf{b}^*\big)^*$ are all equivalent. *Note*: You may have constructed the DFA's for these expressions in response to Exercise 3.7.3.

! **Exercise 3.9.4:** Construct the minimum-state DFA's for the following regular expressions:

a) $(\mathbf{a}|\mathbf{b})^*\mathbf{a}(\mathbf{a}|\mathbf{b})$.

b) $(\mathbf{a}|\mathbf{b})^*\mathbf{a}(\mathbf{a}|\mathbf{b})(\mathbf{a}|\mathbf{b})$.

c) $(\mathbf{a}|\mathbf{b})^*\mathbf{a}(\mathbf{a}|\mathbf{b})(\mathbf{a}|\mathbf{b})(\mathbf{a}|\mathbf{b})$.

Do you see a pattern?

!! **Exercise 3.9.5:** To make formal the informal claim of Example 3.25, show that any deterministic finite automaton for the regular expression

$$(\mathbf{a}|\mathbf{b})^*\mathbf{a}(\mathbf{a}|\mathbf{b})(\mathbf{a}|\mathbf{b})\cdots(\mathbf{a}|\mathbf{b})$$

where $(\mathbf{a}|\mathbf{b})$ appears $n-1$ times at the end, must have at least 2^n states. *Hint*: Observe the pattern in Exercise 3.9.4. What condition regarding the history of inputs does each state represent?

3.10 Summary of Chapter 3

- *Tokens.* The lexical analyzer scans the source program and produces as output a sequence of tokens, which are normally passed, one at a time to the parser. Some tokens may consist only of a token name while others may also have an associated lexical value that gives information about the particular instance of the token that has been found on the input.

- *Lexemes.* Each time the lexical analyzer returns a token to the parser, it has an associated lexeme — the sequence of input characters that the token represents.

- *Buffering.* Because it is often necessary to scan ahead on the input in order to see where the next lexeme ends, it is usually necessary for the lexical analyzer to buffer its input. Using a pair of buffers cyclicly and ending each buffer's contents with a sentinel that warns of its end are two techniques that accelerate the process of scanning the input.

- *Patterns.* Each token has a pattern that describes which sequences of characters can form the lexemes corresponding to that token. The set of words, or strings of characters, that match a given pattern is called a language.

- *Regular Expressions.* These expressions are commonly used to describe patterns. Regular expressions are built from single characters, using union, concatenation, and the Kleene closure, or any-number-of, operator.

- *Regular Definitions.* Complex collections of languages, such as the patterns that describe the tokens of a programming language, are often defined by a regular definition, which is a sequence of statements that each define one variable to stand for some regular expression. The regular expression for one variable can use previously defined variables in its regular expression.

- *Extended Regular-Expression Notation.* A number of additional operators may appear as shorthands in regular expressions, to make it easier to express patterns. Examples include the + operator (one-or-more-of), ? (zero-or-one-of), and character classes (the union of the strings each consisting of one of the characters).

- *Transition Diagrams.* The behavior of a lexical analyzer can often be described by a transition diagram. These diagrams have states, each of which represents something about the history of the characters seen during the current search for a lexeme that matches one of the possible patterns. There are arrows, or transitions, from one state to another, each of which indicates the possible next input characters that cause the lexical analyzer to make that change of state.

- *Finite Automata.* These are a formalization of transition diagrams that include a designation of a start state and one or more accepting states, as well as the set of states, input characters, and transitions among states. Accepting states indicate that the lexeme for some token has been found. Unlike transition diagrams, finite automata can make transitions on empty input as well as on input characters.

- *Deterministic Finite Automata.* A DFA is a special kind of finite automaton that has exactly one transition out of each state for each input symbol. Also, transitions on empty input are disallowed. The DFA is easily simulated and makes a good implementation of a lexical analyzer, similar to a transition diagram.

- *Nondeterministic Finite Automata.* Automata that are not DFA's are called nondeterministic. NFA's often are easier to design than are DFA's. Another possible architecture for a lexical analyzer is to tabulate all the states that NFA's for each of the possible patterns can be in, as we scan the input characters.

- *Conversion Among Pattern Representations.* It is possible to convert any regular expression into an NFA of about the same size, recognizing the same language as the regular expression defines. Further, any NFA can be converted to a DFA for the same pattern, although in the worst case (never encountered in common programming languages) the size of the automaton can grow exponentially. It is also possible to convert any nondeterministic or deterministic finite automaton into a regular expression that defines the same language recognized by the finite automaton.

- *Lex.* There is a family of software systems, including `Lex` and `Flex`, that are lexical-analyzer generators. The user specifies the patterns for tokens using an extended regular-expression notation. `Lex` converts these expressions into a lexical analyzer that is essentially a deterministic finite automaton that recognizes any of the patterns.

- *Minimization of Finite Automata.* For every DFA there is a minimum-state DFA accepting the same language. Moreover, the minimum-state DFA for a given language is unique except for the names given to the various states.

3.11 References for Chapter 3

Regular expressions were first developed by Kleene in the 1950's [9]. Kleene was interested in describing the events that could be represented by McCullough and Pitts' [12] finite-automaton model of neural activity. Since that time regular expressions and finite automata have become widely used in computer science.

Regular expressions in various forms were used from the outset in many popular Unix utilities such as `awk`, `ed`, `egrep`, `grep`, `lex`, `sed`, `sh`, and `vi`. The IEEE 1003 and ISO/IEC 9945 standards documents for the Portable Operating System Interface (POSIX) define the POSIX extended regular expressions which are similar to the original Unix regular expressions with a few exceptions such as mnemonic representations for character classes. Many scripting languages such as Perl, Python, and Tcl have adopted regular expressions but often with incompatible extensions.

The familiar finite-automaton model and the minimization of finite automata, as in Algorithm 3.39, come from Huffman [6] and Moore [14]. Nondeterministic finite automata were first proposed by Rabin and Scott [15]; the subset construction of Algorithm 3.20, showing the equivalence of deterministic and nondeterministic finite automata, is from there.

McNaughton and Yamada [13] first gave an algorithm to convert regular expressions directly to deterministic finite automata. Algorithm 3.36 described in Section 3.9 was first used by Aho in creating the Unix regular-expression matching tool `egrep`. This algorithm was also used in the regular-expression pattern matching routines in `awk` [3]. The approach of using nondeterministic automata as an intermediary is due Thompson [17]. The latter paper also contains the algorithm for the direct simulation of nondeterministic finite automata (Algorithm 3.22), which was used by Thompson in the text editor `QED`.

Lesk developed the first version of `Lex` and then Lesk and Schmidt created a second version using Algorithm 3.36 [10]. Many variants of `Lex` have been subsequently implemented. The GNU version, `Flex`, can be downloaded, along with documentation at [4]. Popular Java versions of `Lex` include `JFlex` [7] and `JLex` [8].

The KMP algorithm, discussed in the exercises to Section 3.4 just prior to Exercise 3.4.3, is from [11]. Its generalization to many keywords appears in [2] and was used by Aho in the first implementation of the Unix utility `fgrep`.

The theory of finite automata and regular expressions is covered in [5]. A survey of string-matching techniques is in [1].

1. Aho, A. V., "Algorithms for finding patterns in strings," in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), Vol. A, Ch. 5, MIT

Press, Cambridge, 1990.

2. Aho, A. V. and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Comm. ACM* **18**:6 (1975), pp. 333–340.

3. Aho, A. V., B. W. Kernighan, and P. J. Weinberger, *The AWK Programming Language*, Addison-Wesley, Boston, MA, 1988.

4. `Flex` home page `http://www.gnu.org/software/flex/`, Free Software Foundation.

5. Hopcroft, J. E., R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Boston MA, 2006.

6. Huffman, D. A., "The synthesis of sequential machines," *J. Franklin Inst.* **257** (1954), pp. 3–4, 161, 190, 275–303.

7. `JFlex` home page `http://jflex.de/` .

8. `http://www.cs.princeton.edu/~appel/modern/java/JLex` .

9. Kleene, S. C., "Representation of events in nerve nets," in [16], pp. 3–40.

10. Lesk, M. E., "Lex – a lexical analyzer generator," Computing Science Tech. Report 39, Bell Laboratories, Murray Hill, NJ, 1975. A similar document with the same title but with E. Schmidt as a coauthor, appears in Vol. 2 of the *Unix Programmer's Manual*, Bell laboratories, Murray Hill NJ, 1975; see `http://dinosaur.compilertools.net/lex/index.html` .

11. Knuth, D. E., J. H. Morris, and V. R. Pratt, "Fast pattern matching in strings," *SIAM J. Computing* **6**:2 (1977), pp. 323–350.

12. McCullough, W. S. and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bull. Math. Biophysics* **5** (1943), pp. 115–133.

13. McNaughton, R. and H. Yamada, "Regular expressions and state graphs for automata," *IRE Trans. on Electronic Computers* **EC-9**:1 (1960), pp. 38–47.

14. Moore, E. F., "Gedanken experiments on sequential machines," in [16], pp. 129–153.

15. Rabin, M. O. and D. Scott, "Finite automata and their decision problems," *IBM J. Res. and Devel.* **3**:2 (1959), pp. 114–125.

16. Shannon, C. and J. McCarthy (eds.), *Automata Studies*, Princeton Univ. Press, 1956.

17. Thompson, K., "Regular expression search algorithm," *Comm. ACM* **11**:6 (1968), pp. 419–422.