By the inductive hypothesis, $s_n = 1 + 3 + 5 + ... + (2n - 1) = n^2$. Hence,

$$1 + 3 + 5 + ... + (2n - 1) + (2(n + 1) - 1) = n^2 + 2n + 1 = (n + 1)^2.$$

Consequently, s_{n+1} holds, and the inductive proof is completed.

■

1.2 Compilation

A *compiler* reads a *source program* written in a *source language* and translates this program into a *target program* written in a *target language* so that both programs are functionally equivalent—that is, they specify the same computational task to perform. As a rule, the source language is a high-level language, such as Pascal or C, while the target language is the machine language of a particular computer or an assembly language, which is easy to transform to the machine language. During the translation, the compiler first *analyzes* the source program to verify that the source program is correctly written in the source language. If so, the compiler generates the target program; otherwise, the compiler reports the errors and unsuccessfully ends the translation.

Compilation Phases

In greater detail, the compiler first makes the lexical, syntax, and semantic analysis of the source program. Then, from the information gathered during this threefold analysis, it generates the intermediate code of the source program, makes its optimization, and creates the resulting target code. As a whole, the *compilation* thus consists of these six *compilation phases*, each of which transforms the source program from one inner representation to another:

- lexical analysis
- syntax analysis
- semantic analysis
- intermediate code generation
- optimized intermediate code generation
- target code generation

Lexical analysis breaks up the source program into *lexemes*—that is, logically cohesive lexical entities, such as identifiers or integers. It verifies that these entities are well-formed, produces *tokens* that uniformly represent lexemes in a fixed-sized way, and sends these tokens to the syntax analysis. If necessary, the tokens are associated with attributes to specify them in more detail. The lexical analysis recognizes every single lexeme by its *scanner*, which reads the sequence of characters that make up the source program to recognize the next portion of this sequence that forms the lexeme. Having recognized the lexeme in this way, the lexical analysis creates its tokenized representation and sends it to the syntax analysis.

Syntax analysis determines the syntax structure of the tokenized source program, provided by the lexical analysis. This compilation phase makes use of the concepts and techniques developed by modern mathematical linguistics. Indeed, the source-language syntax is specified by *grammatical rules*, from which the syntax analysis constructs a *parse*—that is, a sequence of rules that generates the program. The way by which a *parser*, which is the syntax-analysis component responsible for this construction, works is usually explained graphically. That is, a parse is displayed as a *parse tree* whose leaves are labeled with the tokens and each of its parent-children portion forms a *rule tree* that graphically represents a rule. The parser constructs this tree by

smartly selecting and composing appropriate rule trees. Depending on the way it makes this construction, we distinguish two fundamental types of parsers. A *top-down parser* builds the parse tree from the root and proceeds down toward the frontier while a *bottom-up parser* starts from the frontier and works up toward the root. If the parser eventually obtains a complete parse tree for the source program, it not only verifies that the program is syntactically correct but also obtains its syntax structure. On the other hand, if this tree does not exist, the source program is syntactically erroneous.

Semantic analysis checks that the source program satisfies the semantic conventions of the source language. Perhaps most importantly, it performs type checking to verify that each operator has operands permitted by the source-language specification. If the operands are not permitted, this compilation phase takes an appropriate action to handle this incompatibility. That is, it either indicates an error or makes type coercion, during which the operands are converted so they are compatible.

Intermediate code generation turns the tokenized source program to a functionally equivalent program in a uniform intermediate language. As its name indicates, this language is at a level intermediate between the source language and the target language because it is completely independent of any particular machine code, but its conversion to the target code represents a relatively simple task. The intermediate code fulfills a particularly important role in a *retargetable* compiler, which is adapt or retarget for several different computers. Indeed, an installation of a compiler like this on a specific computer only requires the translation of the intermediate code to the computer's machine code while all the compiler part preceding this simple translation remains unchanged.

As a matter of fact, this generation usually makes several conversions of the source program from one internal representation to another. Typically, this compilation phase first creates the *abstract syntax tree*, which is easy to generate by using the information obtained during syntax analysis. Indeed, this tree compresses the essential syntactical structure of the parse tree. Then, the abstract syntax tree is transformed to the *three-address code*, which represents every single source-program statement by a short sequence of simple instructions. This kind of representation is particularly convenient for the optimization.

Optimized intermediate code generation or, briefly, *optimization* reshapes the intermediate code so it works in a more efficient way. This phase usually involves numerous subphases, many of which are applied repeatedly. It thus comes as no surprise that this phase slows down the translation significantly, so a compiler usually allows optimization to be turned off.

In greater detail, we distinguish two kinds of optimizations—*machine-independent optimization* and *machine-dependent optimization*. The former operates on the intermediate code while the latter is applied to the target code, whose generation is sketched next.

Target code generation maps the optimized intermediate representation to the target language, such as a specific assembly language. That is, it translates this intermediate representation into a sequence of the assembly instructions that perform the same task. As obvious, this generation requires detailed information about the target machine, such as memory locations available for each variable used in the program. As already noted, the optimized target code generation attempts to make this translation as economically as possible so the resulting instructions do not waste space or time. Specifically, considering only a tiny target-code fragment at a time, this optimization shortens a sequence of target-code instructions without any functional change by some simple improvements. Specifically, it eliminates useless operations, such as a load of a value into a register when this value already exists in another register.

All the six compilation phases make use of error handler and symbol table management, sketched next.

Error Handler. The three analysis phases can encounter various errors. For instance, the lexical analysis can find out that the upcoming sequence of numeric characters represents no number in the source language. The syntax analysis can find out that the tokenized version of the source program cannot be parsed by the grammatical rules. Finally, the semantic analysis may detect an incompatibility regarding the operands attached to an operator. The error handler must be able to detect any error of this kind. After issuing an error diagnostic, however, it must somehow recover from the error so the compiler can complete the analysis of the entire source program. On the other hand, the error handler is no mind reader, so it can hardly figure out what the author actually meant by an erroneous passage in the source program code. As a result, no mater how sophisticatedly the compiler handles the errors, the author cannot expect that a compiler turns an erroneous program to a properly coded source program. Therefore, no generation of the target program follows the analysis of an erroneous program.

Symbol table management is a mechanism that associates each identifier with relevant information, such as its name, type, and scope. Most of this information is collected during the analysis; for instance, the identifier type is obtained when its declaration is processed. This mechanism assists almost every compilation phase, which can obtain the information about an identifier whenever needed. Perhaps most importantly, it provides the semantic analyzer with information to check the source-program semantic correctness, such as the proper declaration of identifiers. Furthermore, it aids the proper code generation. Therefore, the symbol-table management must allow the compiler to add new entries and find existing entries in a speedy and effective way. In addition, it has to reflect the source-program structure, such as identifier scope in nested program blocks. Therefore, a compiler writer should carefully organize the symbol-table so it meets all these criteria. Linked lists, binary search trees, and hash tables belong to commonly used symbol-table data structures.

Convention 1.4. For a variable x, ☞x denotes a pointer to the symbol-table entry recording the information about x throughout this book.

■

Case Study 1/35 FUN *Programming Language.* While discussing various methods concerning compilers in this book, we simultaneously illustrate how they are used in practice by designing a new Pascal-like programming language and its compiler. This language is called FUN because it is particularly suitable for the computation of mathematical *fun*ctions.

In this introductory part of the case study, we consider the following trivial FUN program that multiplies an integer by two. With this source program, we trace the six fundamental compilation phases described above. Although we have introduced all the notions used in these phases quite informally so far, they should be intuitively understood in terms of this simple program.

```
program DOUBLE;
{This FUN program reads an integer value and multiplies it by two.}

integer u;

begin
    read(u);
    u = u * 2;
    write(u);
end.
```

Lexical analyzer divides the source program into lexemes and translates them into tokens, some of which have attributes. In general, an attributed token has the form $t\{a\}$, where t is a token and a

represents *t*'s attribute that provides further information about *t*. Specifically, the FUN lexical analyzer represents each identifier *x* by an attributed token of the form $i\{\text{☞}x\}$, where *i* is the token specifying an identifier as a generic type of lexemes and the attribute ☞*x* is a pointer to the symbol-table entry that records all needed information about this particular identifier *x*, such as its type. Furthermore, $\#\{n\}$ is an attributed token, where # represents an integer in general and its attribute *n* is the integer value of the integer in question. Next, we give the tokenized version of program DOUBLE, where | separates the tokens of this program. Figure 1.5 gives the symbol table created for DOUBLE's identifiers.

program | $i\{\text{☞DOUBLE}\}$ | ; | ***integer*** | $i\{\text{☞}u\}$ | ; | ***begin*** | ***read*** | (| $i\{\text{☞}u\}$ |) | ; | $i\{\text{☞}u\}$ | = | $i\{\text{☞}u\}$ | * | $\#\{2\}$ | ; | ***write*** | (| $i\{\text{☞}u\}$ |) | ***end*** | .

Name	***Type***	…
DOUBLE		
u	***integer***	
⋮		

Figure 1.5 ***Symbol Table.***

Syntax analyzer reads the tokenized source program from left to right and verifies its syntactical correctness by grammatical rules. Graphically, this grammatical verification is expressed by constructing a parse tree, in which each parent-children portion represents a rule. This analyzer works with tokens without any attributes, which play no role during the syntax analysis. In DOUBLE, we restrict our attention just to the expression $i\{\text{☞}u\}$ * $\#\{2\}$, which becomes *i* * # without the attributes. Figure 1.6 gives the parse tree for this expression.

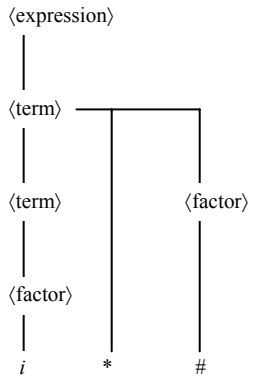

Figure 1.6 ***Parse Tree.***

Semantic analyzer checks semantic aspects of the source program, such as type checking. In DOUBLE, it consults the symbol table to find out that *u* is declared as **integer**.

Intermediate code generator produces the intermediate code of DOUBLE. First, it implements its syntax tree (see Figure 1.7).

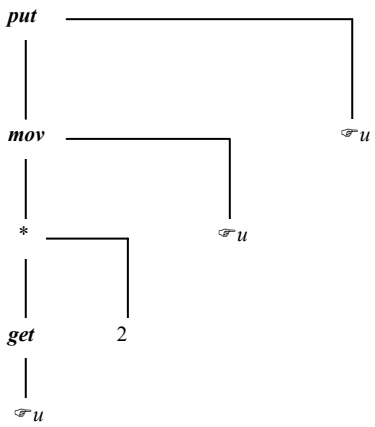

Figure 1.7 ***Syntax Tree.***

Then, it transforms this tree to the following three-address code, which makes use of a temporary variable *t* produced by the compiler. The ***get*** instruction moves the input integer value into *u*. The ***mul*** instruction multiplies the value of *u* by 2 and sets *t* to the result of this multiplication. The ***mov*** instruction moves the value of *t* to *u*. Finally, the ***put*** instruction prints the value of *u* out.

[***get***, , , ☞*u*]
[***mul***, ☞*u*, 2, ☞*t*]
[***mov***, ☞*t*, , ☞*u*]
[***put***, , , ☞*u*]

Optimizer reshapes the intermediate code to perform the computational task more efficiently. Specifically, in the above three-address program, it replaces ☞*t* with ☞*u*, and removes the third instruction to obtain this shorter one-variable three-address program

[***get***, , , ☞*u*]
[***mul***, ☞*u*, 2, ☞ *u*]
[***put***, , , ☞*u*]

Target code generator turns the optimized three-address code into a target program, which performs the computational task that is functionally equivalent to the source program. Of course, like the previous optimizer, the target code generator produces the target program code as succinctly as possible. Specifically, the following hypothetical assembly-language program, which is functionally equivalent to DOUBLE, consists of three instructions and works with a single register, *R*. First, instruction *GET R* reads the input integer value into *R*. Then, instruction *MUL R*, 2 multiplies the contents of *R* by 2 and places the result back into *R*, which the last instruction *PUT R* prints out.

```
GET     R
MUL     R, 2
PUT     R
```

■

Compiler Construction

The six fundamental compilation phases—lexical analysis, syntax analysis, semantic analysis, intermediate code generation, optimization, and target code generation—are abstracted from the

translation process made by a real compiler, which does not execute these phases strictly consecutively. Rather, their execution somewhat overlaps in order to complete the whole compilation process as fast as possible (see Figure 1.8). Since the source-program syntax structure represents probably the most important information to the analysis as a whole, the syntax analyzer guides the performance of all the analysis phases as well as the intermediate code generator. Indeed, the lexical analyzer goes into operation only when the syntax analyzer requests the next token. The syntax analyzer also calls the semantic analysis routines to make their semantic-related checks. Perhaps most importantly, the syntax analyzer directs the intermediate code generation actions, each of which translates a bit of the tokenized source program to a functionally equivalent portion of the intermediate code. This *syntax-directed translation* is based on grammatical rules with associated actions over *attributes* attached to symbols occurring in these rules to provide the intermediate code generator with specific information needed to produce the intermediate code. For instance, these actions generate some intermediate code operations with operands addressed by the attributes. When this generation is completed, the resulting intermediate code usually contains some useless or redundant instructions, which are removed by a machine-independent optimizer. Finally, in a close cooperation with a machine-dependent optimizer, the target code generator translates the optimized intermediate program into the target program and, thereby, completes the compilation process.

Passes. Several compilation phases may be grouped into a single *pass* consisting of reading an internal version of the program from a file and writing an output file. As passes obviously slow down the translation, *one-pass compilers* are usually faster than *multi-pass compilers*. Nevertheless, some aspects concerning the source language, the target machine, or the compiler design often necessitate an introduction of several passes. Regarding the source language, some questions raised early in the source program may remain unanswered until the compiler has read the rest of the program. For example, there may exist references to procedures that appear later in the source code. Concerning the target machine, unless there is enough memory available to hold all the intermediate results obtained during compilation, these results are stored into a file, which the compiler reads during a subsequent pass. Finally, regarding the compiler design, the compilation process is often divided into two passes corresponding to the two ends of a compiler as explained next.

Ends. The *front end* of a compiler contains the compilation portion that heavily depends on the source language and has no concern with the target machine. On the other hand, the *back end* is primarily dependent on the target machine and largely independent of the source language. As a result, the former contains all the three analysis phases, the intermediate code generation, and the machine-independent optimization while the latter includes the machine-dependent optimization and the target code generator. In this two-end way, we almost always organize a retargetable compiler. Indeed, to adapt it for various target machines, we use the same front end and only redo its back end as needed. On the other hand, to obtain several compilers that translate different programming languages to the same target language, we use the same back end with different front ends.

Compilation in Computer Context. To sketch where the compiler fits into the overall context of writing and executing programs, we sketch the computational tasks that usually precede or follow a compilation process.

Before compilation, a source program may be stored in several separate files, so a preprocessor collects them together to create a single source program, which is subsequently translated as a whole.

After compilation, several post-compilation tasks are often needed to run the generated program on computer. If a compiler generates assembly code as its target language, the resulting target program is translated into the machine code by an assembler. Then, the resulting machine code is

usually linked together with some library routines, such as numeric functions, character string operations, or file handling routines. That is, the required library services are identified, *loaded* into memory, and *linked* together with the machine code program to create an executable code (the discussion of linkers and loaders is beyond the scope of this book). Finally, the resulting executable code is placed in memory and executed, or by a specific request, this code is stored on a disk and executed later on.

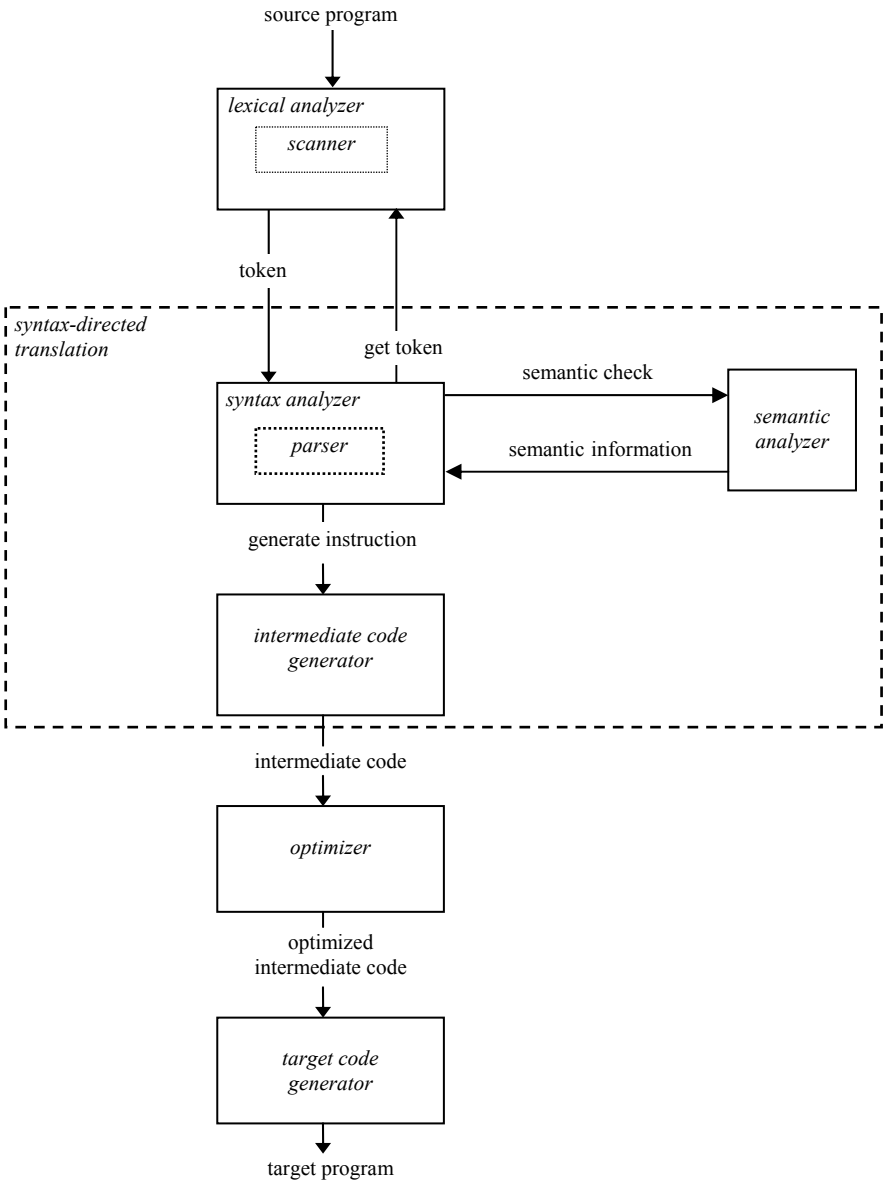

Figure 1.8 ***Compiler Construction*****.**

1.3 Rewriting Systems

As explained in the previous section, each compilation phase actually transforms the source program from one compiler inner representation to another. In other words, it *rewrites* a string that represents an inner form of the source program to a string representing another inner form that is closer to the target program, and this rewriting is obviously ruled by an algorithm. It is thus only natural to formalize these phases by rewriting systems, which are based on finite many rules that abstractly represent the algorithms according to which compilation phases are performed.

Definition 1.5 *Rewriting System*. A *rewriting system* is a pair, $M = (\Sigma, R)$, where Σ is an alphabet, and R is a finite relation on Σ^*. Σ is called the *total alphabet of M* or, simply, *M*'s *alphabet*. A member of R is called a *rule of M*, so R is referred to as *M*'s *set of rules*.

■

Convention 1.6. Each rule $(x, y) \in R$ is written as $x \to y$ throughout this book. For brevity, we often denote $x \to y$ with a label r as $r\colon x \to y$, and instead of $r\colon x \to y \in R$, we sometimes write $r \in R$. For $r\colon x \to y \in R$, x and y represent r's *left-hand side*, denoted by ***lhs***(r), and r's *right-hand side*, denoted by ***rhs***(r), respectively. R^* denotes the set of all *sequences of rules* from R; as a result, by $\rho \in R^*$, we briefly express that ρ is a sequence consisting of $|\rho|$ rules from R. By analogy with strings (see Convention 1.1), in sequences of rules, we simply juxtapose the rules and omit the parentheses as well as all separating commas in them. That is, if $\rho = (r_1, r_2, \ldots, r_n)$, we simply write ρ as $r_1 r_2 \ldots r_n$. To explicitly express that Σ and R represent the components of M, we write ${}_M\Sigma$ and ${}_M R$ instead of Σ and R, respectively.

■

Definition 1.7 *Rewriting Relation.* Let $M = (\Sigma, R)$ be a rewriting system. The *rewriting relation* over Σ^* is denoted by $\Rightarrow$ and defined so that for every $u, v \in \Sigma^*$, $u \Rightarrow v$ in M if and only if there exist $x \to y \in R$ and $w, z \in \Sigma^*$ such that $u = wxz$ and $v = wyz$.

■

Let $u, v \in \Sigma^*$. If $u \Rightarrow v$ in M, we say that *M directly rewrites u to v*. As usual, for every $n \geq 0$, the n-fold product of $\Rightarrow$ is denoted by $\Rightarrow^n$. If $u \Rightarrow^n v$, *M rewrites u to v in n steps*. Furthermore, the transitive-reflexive closure and the transitive closure of $\Rightarrow$ are $\Rightarrow^*$ and $\Rightarrow^+$, respectively. If $u \Rightarrow^* v$, we simply say that *M rewrites u to v*, and if $u \Rightarrow^+ v$, *M rewrites u to v in a nontrivial way*. In this book, we sometimes need to explicitly specify the rules used during rewriting. Suppose M makes $u \Rightarrow v$ so that $u = wxz$, $v = wyz$ and M replaces x with y by applying $r\colon x \to y \in R$. To express this application, we write $u \Rightarrow v\ [r]$ or, in greater detail, $wxz \Rightarrow wyz\ [r]$ in M and say that *M directly rewrites uxv to uyv by r*. More generally, let n be a non-negative integer, $w_0, w_1, \ldots, w_n$ be a sequence with $w_i \in \Sigma^*$, $0 \leq i \leq n$, and $r_j \in R$ for $1 \leq j \leq n$. If $w_{j-1} \Rightarrow w_j\ [r_j]$ in M for $1 \leq j \leq n$, *M rewrites* w_0 *to* w_n *in n steps by* $r_1 r_2 \ldots r_n$, symbolically written as $w_0 \Rightarrow^n w_n\ [r_1 r_2 \ldots r_n]$ in M ($n = 0$ means $w_0 \Rightarrow^0 w_0\ [\varepsilon]$). By $u \Rightarrow^* v\ [\rho]$, where $\rho \in R^*$, we express that M makes $u \Rightarrow^* v$ by using ρ; $u \Rightarrow^+ v\ [\rho]$ has an analogical meaning. Of course, whenever the specification of applied rules is superfluous, we omit it and write $u \Rightarrow v$, $u \Rightarrow^n v$, and $u \Rightarrow^* v$ for brevity.

Language Models

The language constructs used during some compilation phases, such as the lexical and syntax analysis, are usually represented by formal languages defined by a special case of rewriting systems, customarily referred to as *language-defining models* underlying the phase. Accordingly, the compiler parts that perform these phases are usually based upon algorithms that implement the corresponding language models.

Convention 1.8. Throughout this book, the language defined by a model M is denoted by $L(M)$. In an algorithm that implements M working with a string, w, we write **ACCEPT** to announce that $w \in L(M)$ while **REJECT** means $w \notin L(M)$.

■

By the investigation of the language models, we obtain a systematized body of knowledge about the compiler component that performs the compilation phase, and making use of this valuable knowledge, we try to design the component as sophisticatedly as possible. In particular, the language models underlying the phases that are completely independent of the target machine, such as the analysis phases, allow us to approach these phases in a completely general and rigorous way. This approach to the study of compilation phases has become so common and fruitful that it has given rise to several types of models, some of which define the same languages. We refer to the models that define the same language as *equivalent models*, and from a broader perspective, we say that some types of models are *equally powerful* if the family of languages defined by models of each of these types is the same.

Synopsis of the Book

Specifically, regarding the compilation process discussed in this book, the lexical analysis is explained by equally powerful language-defining models called *finite automata* and *regular expressions* (see Chapter 2). The syntax analysis is expressed in terms of equally powerful *pushdown automata* and *grammars* (see Chapters 3 through 5), and the syntax-directed translation is explained by *attribute grammars* (see Chapter 6), which represent an extension of the grammars underlying the syntax analysis. The optimization and the target code generation are described without any formal models in this introductory book (see Chapter 7). In its conclusion, we suggest the contents of an advanced course about compilers following a class based upon the present book (see Chapter 8). In the appendix, we demonstrate how to implement a real compiler.

Exercises

1.1. Let $L = \{a^n | \, n \geq 2\}$ be a language over an alphabet, Σ. Determine *complement*(L) for $\Sigma = \{a\}$, $\Sigma = \{a, b\}$, and $\Sigma = \{a, b, \ldots, z\}$.

1.2. By using the notions introduced in Section 1.1, such as various language operations, define integer and real numbers in your favorite programming language.

1.3. Let Σ be a subset of the set of all non-negative integers, and let ϕ be the total function from the set of all non-negative integers to $\{0, 1\}$ defined by this equivalence $\phi(i) = 1$ if and only if $i \in \Sigma$, for all non-negative integers, i. Then, ϕ is the *characteristic function* of Σ. Illustrate this notion in terms of formal languages.

1.4. Let $X = \{i | \, i$ is a positive integer$\} \cup \{$*and*, *are*, *even*, *Integers*, *odd*, *respectively*$\}$. Describe the language of all well-constructed English sentences consisting of X's members, commas and a period. For instance, this language contains *Integers* 2 *and* 5 *are even and odd*, *respectively*. Is this language infinite? Can you define this language by using X and operations concatenation, union, and iteration?

1.5. Let Σ and Ω be two sets, and let ρ and ρ' be two relations from Σ to Ω. If ρ and ρ' represent two identical subsets of $\Sigma \times \Omega$, then ρ *equals* ρ', symbolically written as $\rho = \rho'$. Illustrate this definition in terms of language translations.

1.6. Let Σ be a set, and let ρ be a relation on Σ. Then,

- if for all $a \in \Sigma$, $a\rho a$, then ρ is *reflexive*;
- if for all $a, b \in \Sigma$, $a\rho b$ implies $b\rho a$, then ρ is *symmetric*;
- if for all $a, b \in \Sigma$, ($a\rho b$ and $b\rho a$) implies $a = b$, then ρ is *antisymmetric*;
- if for all $a, b, c \in \Sigma$, ($a\rho b$ and $b\rho c$) implies $a\rho c$, then ρ is *transitive*.

Illustrate these notions in terms of language translations.

1.7. Let $\Sigma = \{1, 2, \ldots, 8\}$. Consider the following binary relations over Σ:

- $\varnothing$;
- $\{(1, 3), (3, 1), (8, 8)\}$;
- $\{(1, 1), (2, 2), (8, 8)\}$;
- $\{(a, a) | \ a \in \Sigma\}$;
- $\{(a, b) | \ a, b \in \Sigma, a < b\}$;
- $\{(a, b) | \ a, b \in \Sigma, a \leq b\}$;
- $\{(a, b) | \ a, b \in \Sigma, a + b = 9\}$;
- $\{(a, b) | \ a, b \in \Sigma, b$ is divisible by $a\}$;
- $\{(a, b) | \ a, b \in \Sigma, a - b$ is divisible by $3\}$.

Note that a is divisible by b if there exists a positive integer c such that $a = bc$. For each of these relations, determine whether it is reflexive, symmetric, antisymmetric, or transitive.

1.8. Let Σ be a set, and let ρ be a relation on Σ. If ρ is reflexive, symmetric, and transitive, then ρ is an *equivalence relation*. Let ρ be an equivalence relation on Σ. Then, ρ partitions Σ into disjoint subsets, called *equivalence classes*, so that for each $a \in \Sigma$, the equivalence class of a is denoted by $[a]$ and defined as $[a] = \{b | \ a\rho b\}$. Prove that for all a and b in Σ, either $[a] = [b]$ or $[a] \cap [b] = \varnothing$.

1.9*Solved*. Let Σ be a set, and let ρ be a relation on Σ. If ρ is reflexive, antisymmetric, and transitive, then ρ is a *partial order*. If ρ is a partial order satisfying for all $a, b \in \Sigma$, $a\rho b$, $b\rho a$, or $a = b$, then ρ is a *linear order*. Formalize the usual dictionary order as a *lexicographic order* based upon a linear order.

1.10. Write a program that implements the lexicographic order constructed in Exercise 1.9. Test this program on a large file of English words.

1.11. Let Σ be a set. Define the relation ρ on $Power(\Sigma)$ as $\rho = \{(A, B) | \ A, B \in Power(\Sigma), A \subseteq B\}$ (see Section 1.1 for $Power(\Sigma)$). Prove that ρ represents a partial order.

1.12. By induction, prove that for any set Σ, $card(Power(\Sigma)) = 2^{card(\Sigma)}$.

1.13*Solved*. Prove Theorem 1.9, given next.

Theorem 1.9. Let Σ be a set, ρ be a relation on Σ, and ρ^+ be the transitive closure of ρ. Then, (1) ρ^+ is a transitive relation, and (2) if ρ' is a transitive relation such that $\rho \subseteq \rho'$, then $\rho^+ \subseteq \rho'$.

1.14. Prove Theorem 1.10, given next.

Theorem 1.10. Let Σ be a set, ρ be a relation on Σ, and ρ^* be the transitive and reflexive closure of ρ. Then, (1) ρ^* is a transitive and reflexive relation, and (2) if ρ' be a transitive and reflexive relation such that $\rho \subseteq \rho'$, then $\rho^* \subseteq \rho'$.

1.15. Generalize the notion of a binary relation to the notion of an *n*-ary relation, where *n* is a natural number.

1.16*Solved*. Define the prefix Polish notation.

1.17*Solved*. In Section 1.1, we translate the infix expression $(a + b) * c$ into the postfix notation $ab+c*$. Translate $(a + b) * c$ into the prefix notation. Describe this translation in a step-by-step way.

1.18. Consider the one-dimensional representation of trees, $\Re$ (see Section 1.1). Prove that the prefix Polish notation is the same as $\Re$ with parentheses deleted.

1.19. Design a one-dimensional representation for trees, different from $\Re$.

1.20. Introduce the notion of an *n*-ary function, where *n* is a natural number. Illustrate this notion in terms of compilation.

1.21. Consider the directed graphs defined and discussed in Section 1.1. Intuitively, *undirected graphs* are similar to these graphs except that their edges are undirected. Define them rigorously.

1.22. Recall that Section 1.1 has defined a tree as an acyclic graph, $G = (\Sigma, \rho)$, satisfying these three properties:

(1) G has a specified node whose in-degree is 0; this node represents the *root* of G, denoted by *root*(G).
(2) If $a \in \Sigma$ and $a \neq root(G)$, then a is a descendent of *root*(G) and the in-degree of a is 1.
(3) Each node, $a \in \Sigma$, has its direct descendents, b_1 through b_n, ordered from the left to the right so that b_1 is the leftmost direct descendent of a and b_n is the rightmost direct descendent of a.

Reformulate (3) by using the notion of a partial order, defined in Exercise 1.9.

1.23. A *tautology* is a statement that is true for all possible truth values of the statement variables. Explain why every theorem of a formal mathematical system represents a tautology.

1.24. Prove that the contrapositive law represents a tautology. State a theorem and prove this theorem by using the contrapositive law.

1.25. A *Boolean algebra* is a formal mathematical system, which consists of a set, Σ, and operations *and*, *or*, and *not*. The axioms of Boolean algebra follow next.

Associativity. (1) *a or* (*b or c*) = (*a or b*) *or c*, and
(2) *a and* (*b and c*) = (*a and b*) *and c*, for all $a, b, c \in \Sigma$.
Commutativity. (1) *a or b* = *b or a*, and
(2) *a and b* = *b and a*, for all $a, b \in \Sigma$.
Distributivity. (1) *a and* (*b or c*) = (*a and b*) *or* (*a and c*), and
(2) *a or* (*b and c*) = (*a or b*) *and* (*a or c*), for all $a, b \in \Sigma$.

In addition, Σ contains two distinguished members, 0 and 1, which satisfy these laws for all $a \in \Sigma$,

(1) *a or* 0 = *a*, (2) *a and* 1 = *a*, (3) *a or* (*not a*) = 1, and (4) *a and* (*not a*) = 0.

The rule of inference is substitution of equals for equals. Discuss the Boolean algebra in which Σ's only members are 0 and 1, representing falsehood and truth, respectively. Why is this algebra important to the mathematical foundations of computer science?

1.26. Consider your favorite programming language, such as Pascal or C. Define its lexical units by the language operations introduced in Section 1.1. Can the syntax be defined in the same way? Justify your answer.

1.27. Learn the *syntax diagram* from a good high-level programming language manual. Design a simple programming language and describe its syntax by these diagrams.

1.28$_{Solved}$**.** Recall that a rewriting system is a pair, $M = (\Sigma, R)$, where Σ is an alphabet, and R is a finite relation on Σ^* (see Definition 1.5). Furthermore, the rewriting relation over Σ^* is denoted by $\Rightarrow$ and defined so that for every $u, v \in \Sigma^*$, $u \Rightarrow v$ in M if and only if there exist $x \to y \in R$ and $w, z \in \Sigma^*$ such that $u = wxz$ and $v = wyz$ (see Definition 1.7). For every $n \geq 0$, the n-fold product of $\Rightarrow$ is denoted by $\Rightarrow^n$. Determine a non-negative integer, $m \geq 0$, satisfying for all $u, v \in \Sigma^*$, if $u \Rightarrow^n v$ in M, then $|v| \leq nm|u|$.

Solutions to Selected Exercises

1.9. Let Σ be a set, and let β be a linear order on Σ. We extend β to Σ^* so that for any $x, y \in \Sigma^*$, $x \beta y$ if $x \in prefixes(y) - \{y\}$, or for some $k \geq 1$ such that $|x| > k$ and $|y| > k$, $prefix(x, k-1) = prefix(y, k-1)$ and $symbol(x, k)\ \beta\ symbol(y, k)$. This extended definition of β is referred to as the *lexicographic order* β on Σ^*. Take, for instance, Σ as the English alphabet and β as its alphabetical order. Then, the lexical order β extended in the above way represents the usual dictionary order on Σ^*.

1.13. To demonstrate that Theorem 1.9 holds, we next prove that (1) and (2) hold true.

(1) To prove that ρ^+ is a transitive relation, we demonstrate that if $a\rho^+b$ and $b\rho^+c$, then $a\rho^+c$. As $a\rho^+b$, there exist $x_1, \ldots, x_n$ in Σ so $x_1\rho x_2, \ldots, x_{n-1}\rho x_n$, where $x_1 = a$ and $x_n = b$. As $b\rho^+c$, there also exist $y_1, \ldots, y_m$ in Σ so $y_1\rho y_2, \ldots, y_{m-1}\rho y_m$, where $y_1 = b$ and $y_m = c$. Consequently, $x_1\rho x_2, \ldots, x_{n-1}\rho x_n, y_1\rho y_2, \ldots, y_{m-1}\rho y_m$, where $x_1 = a$, $x_n = b = y_1$, and $y_m = c$. As a result, $a\rho^+c$.
(2) We demonstrate that if ρ' is a transitive relation such that $\rho \subseteq \rho'$, then $\rho^+ \subseteq \rho'$. Less formally, this implication means that ρ^+ is the smallest transitive relation that includes ρ. Let ρ' be a transitive relation such that $\rho \subseteq \rho'$, and let $a\rho^+b$. Then, there exist $x_1, \ldots, x_n$ in Σ so $x_1\rho x_2, \ldots, x_{n-1}\rho x_n$, where $x_1 = a$ and $x_n = b$. As $\rho \subseteq \rho'$, $x_1\rho' x_2, \ldots, x_{n-1}\rho' x_n$ where $x_1 = a$ and $x_n = b$. Because ρ' is transitive, $a\rho' b$. Consequently, $a\rho^+b$ implies $a\rho' b$.

1.16. The prefix notation is defined recursively as follows. Let Ω be a set of binary operators, and let Σ be a set of operands.

- Every $a \in \Sigma$ is a prefix representation of a.
- Let AoB be an infix expression, where $o \in \Omega$, and A, B are infix expressions. Then, oCD is the prefix representation of AoB, where C and D are the prefix representations of A and B, respectively.
- Let C be the prefix representation of an infix expression A. Then, C is the prefix representation of (A).

1.17. The expression $(a + b) * c$ is of the form $A * B$ with $A = (a + b)$ and $B = c$. The prefix notation for B is c. The prefix notation for A is $+ab$. Thus the prefix expression for $(a + b) * c$ is $*+abc$.

1.28. Take m as any non-negative integer satisfying $m \geq |y| - |x|$, for all $x \to y \in R$.