Compilers: Principles, Techiniques and Tools
Solutions Manual

GAUTAM SINGH

May 19, 2023

ii

Contents

List of Figures

List of Tables

List of Listings

1

Introduction

1.1 Language Processors

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

Solution The differences are illustrated in Table 1.1.

Compilers	Interpreters
It translates a program from source language to an equivalent program in the target language.	It executes the instructions in the source program directly without translation.
Inputs are provided to the target program.	Inputs are provided directly to the source program.
Produces worse error diagnostics since it translates the entire source program in one step without executing it.	Produces better error diagnostics since it executes the source program line by line.
Faster in mapping inputs to outputs.	Slower in mapping inputs to outputs.

Table 1.1: Differences Between Compilers and Interpreters.

Exercise 1.1.2 What are the advantages of:

(a) A compiler over an interpreter? (b) An interpreter over a compiler?

Solution See Table 1.1.

1

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

Solution   The advantages of a compiler producing assembly language instead of machine language are:

1. Assembly language is easier to produce as output.

2. Assembly language is easier to debug.

3. One can use available assemblers for the hardware to convert the assembly language into machine code.

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?

Solution   The advantages of using C as a target language for a compiler are as follows.

1. It may be less cumbersome to code in another language than in C.

2. The C language already has a compiler that can be readily used, and there is no need to create a new compiler for another language from scratch.

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

Solution   The tasks that an assembler needs to perform are as follows.

1. Produce relocatable machine code from assembly code.

2. Report errors in the assembly code.

1.3   The Evolution of Programming Languages

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

Solution

    a) *Imperative:* C, C++, Java

    b) *Declarative:* ML

    c) *von Neumann:* C, Fortran

    d) *Object-oriented:* C++, Java

    e) *Functional:* ML

    f) *Third-generation:* C, C++, Cobol, Fortran, Java, Lisp

    g) *Fourth-generation:* Perl, Python, VB

    h) *Scripting:* Lisp, Perl, Python, VB

1.6   Programming Language Basics

Exercise 1.6.1   For the block-structured C code of Listing 1.1, indicate the values assigned to w, x, y, and z.

```c
int w, x, y, z;
int i = 4; int j = 5;
{                    /* BLOCK 1 */
    int j = 7;
    i = 6;
    w = i + j;
}
x = i + j;          /* BLOCK 2 */
{                    /* BLOCK 3 */
    int i = 8;
    y = i + j;
}
z = i + j;          /* BLOCK 4 */
```

Listing 1.1: Source code for Exercise 1.6.1

Solution   Inside block 1, a new local variable `j` is initialized with the value 7, and the value of `i` in the global scope is changed to 6. Thus, the value of `w` is 13.

In block 2, `j` has the value 5 from its global definition, and the value of `i` was changed to 6 in block 1. Therefore, the value of `x` is 11.

In block 3, a new local variable `i` is initialized with the value 8, and the value of `j` is 5 from its global scope. Thus, the value of `y` is 13.

In block 4, the value of `z` is the same as `x`, since no further changes are made to the values of `i` or `j`. Thus, the value of `z` is 11.

Exercise 1.6.2   Repeat Exercise 1.6.1 for the code of Listing 1.2.

```
1   int w, x, y, z;
2   int i = 3; int j = 4;
3   {                   /* BLOCK 1 */
4       int i = 5;
5       w = i + j;
6   }
7   x = i + j;          /* BLOCK 2 */
8   {                   /* BLOCK 3 */
9       int j = 6;
10      i = 7;
11      y = i + j;
12  }
13  z = i + j;          /* BLOCK 4 */
```

Listing 1.2: Source code for Exercise 1.6.2

Solution   Inside block 1, a new local variable `i` is initialized with the value 5, and the value of `j` in the global scope is changed to 4. Thus, the value of `w` is 9.

In block 2, there are no changes to the values of the global variables `i` and `j`. Therefore, the value of `x` is 7.

In block 3, a new local variable `j` is initialized with the value 6, and the value of `i` is changed to 7 in its global scope. Thus, the value of `y` is 13.

In block 4, the value of `i` is now 7 from block 3, and the value of `j` is unchanged in its global scope. Thus, the value of `z` is 11.

Exercise 1.6.3   For the block-structured code of Listing 1.3, assuming the usual static scoping of declarations, give the scope of each of the twelve declarations.

Solution   Using set notation and indexing variables with the same identifier by block as in Listing 1.3, the scopes of the 12 variables are shown in Table 1.2.

```
1   {                       /* Block B1 */
2       int w, x, y, z;
3       {                   /* Block B2 */
4           int x, z;
5           {               /* Block B3 */
6               int w, x;
7           }
8       }
9       {                   /* Block B4 */
10          int w, x;
11          {               /* Block B5 */
12              int y, z;
13          }
14      }
15  }
```

Listing 1.3: Source code for Exercise 1.6.3

Variable	Scope
w1	$B_1 \setminus (B_3 \cup B_4)$
x1	$B_1 \setminus (B_2 \cup B_4)$
y1	$B_1 \setminus B_5$
z1	$B_1 \setminus (B_2 \cup B_5)$
x2	$B_2 \setminus B_3$
z2	B_2
w3	B_3
x3	B_3
w4	B_4
x4	B_4
y5	B_5
z5	B_5

Table 1.2: Scopes of variables declared in Listing 1.3.

Exercise 1.6.4 For the block-structured C code of Listing 1.4, indicate the values assigned to *w*, *x*, *y*, and *z*.

Solution When `main` calls `b`, the first statement reads `x = (x+1)`, which changes the value of `x` to 3 in the *global scope* as there are no declarations for `x` in the local scope of `b`. Hence, the `printf` outputs 3.
For the case of `c`, the latest definition of `x` is in its local scope, hence the `printf` will output 2 on substituting the macro for `a`.

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

Listing 1.4: Source code for Exercise 1.6.4.

The final output is shown in Listing 1.5.

```
1  2
2  3
```

Listing 1.5: Output on running Listing 1.4.

2

A Simple Syntax-Directed Translator

2.2 Syntax Definition

Exercise 2.2.1   Consider the context-free grammar

$$S \to S\ S\ \texttt{+}\ |\ S\ S\ \texttt{*}\ |\ \texttt{a}$$

a) Show how the string `aa+a*` can be generated by this grammar.

b) Construct a parse tree for this string.

c) What language does this grammar generate? Justify your answer.

Solution

a) We derive `aa+a*` using the following leftmost derivation.

$$S \to SS\texttt{*} \to SS\texttt{+}S\texttt{*} \to \texttt{a}S\texttt{+}S\texttt{*} \to \texttt{aa+}S\texttt{*} \to \texttt{aa+a*}$$
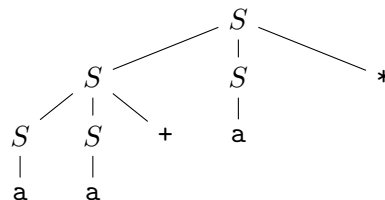
b) The parse tree for the above string is in Figure 2.1.

Figure 2.1: Parse tree for the string `aa+a*`.

c) We claim that the given context-free grammar generates all and only postfix expressions with operand `a` and operations `+` and `*`. We prove this claim by induction on the number of operations in the expression, which we call n.

Notice that at any step of the derivation, the length of the string either increases by 2 or remains constant. Since the smallest string in this grammar is $S \rightarrow$ `a`, we see that any string generated by this grammar has odd length.

The base case ($n = 0$) consists of the derivation $S \rightarrow$ `a`, so it holds.

Suppose that the claim holds for all postfix expressions having upto $n = k$ operations. Note that the last symbol for any postfix expression must be an operation symbol by definition (except for the base case), hence given any string $s = xy$`b` with $n = k + 1$ operations, we use the fact that x and y are postfix expressions with at most k operations and the induction hypothesis to arrive at the derivations

$$S \rightarrow SS\mathtt{b} \xrightarrow{*} xS\mathtt{b} \xrightarrow{*} xy\mathtt{b}$$

where `b` is one of the two operators. This proves the claim.

Exercise 2.2.2 What language is generated by the following grammars? In each case justify your answer.

a) $S \rightarrow 0 \; S \; 1 \mid 0 \; 1$

b) $S \rightarrow \mathtt{+} \; S \; S \mid \mathtt{-} \; S \; S \mid \mathtt{a}$

c) $S \rightarrow S \; (\; S \;) \; S \mid \epsilon.$

d) $S \rightarrow \mathtt{a} \; S \; \mathtt{b} \; S \mid \mathtt{b} \; S \; \mathtt{a} \; S \mid \epsilon$

e) $S \rightarrow \mathtt{a} \mid S \; \mathtt{+} \; S \mid S \; S \mid S \; \mathtt{*} \mid (\; S \;)$

Solution

a) We prove that the language generated is $\{0^n 1^n \mid n > 0\}$ by induction on n. For $n = 1$, we see that $S \rightarrow 01$. Suppose $S \xrightarrow{*} 0^k 1^k$ for some $k > 0$. Then, we have

$$S \rightarrow 0S1 \xrightarrow{*} 00^k 1^k 1 = 0^{k+1} 1^{k+1}$$

which proves the claim.

b) Similar to Exercise 2.2.1, the given grammar generates all prefix expressions containing only the `+` and `-` operands.

c) This grammar generates all balanced bracketing sequences containing only parentheses. We show this by using the definition.

Clearly, we see that $S \to \epsilon$, so the base case holds. Now, suppose that x is a balanced bracketing sequence in this language. Then, we also see that

$$S \to S(S)S \to (S)S \to (S) \xrightarrow{*} (x)$$

or (x) is also in this language. Suppose also that $x = (x')$ and $y = (y')$ are balanced bracketing sequences in this language without loss of generality. Then,

$$S \to S(S)S \to (S)S \to (S)S(S)S \xrightarrow{*} (x')(y') = xy.$$

which proves the claim. Note that if x and y are not in the required form, then we can apply the above derivation multiple times as needed to get the required form.

d) This grammar generates all strings with equal occurrences of a and b (including the empty string). This is because, in any rule, whenever an a is added, a b is also added.

e) This grammar generates regular expressions of a, the union and star operations and parentheses. We prove this claim by the definition of regular expressions. Clearly, $S \to$ a is in this language. If any regular expressions R and T are in this language, then

$$S \to (S) \xrightarrow{*} (R)$$
$$S \to S * \xrightarrow{*} R *$$
$$S \to S + S \xrightarrow{*} R + T$$
$$S \to SS \xrightarrow{*} RT$$

and this proves the claim.

Exercise 2.2.3 Which of the grammars in Exercise 2.2.2 are ambiguous?

Solution

1. This grammar is not ambiguous.

2. This grammar is not ambiguous.

3. This grammar is ambiguous. Consider the string ()(). We show two leftmost derivations for it.

$$S \to S(S)S \to (S)S \to ()S \to ()S(S)S \xrightarrow{*} ()()$$
$$S \to S(S)S \to S(S)S(S)S \xrightarrow{*} ()()$$

4. This grammar is ambiguous. Consider the string `ab`. We show two leftmost derivations for it.

$$S \to \text{a}S\text{b}S \to \text{ab}S \to \text{ab}S \xrightarrow{*} \text{abab}$$
$$S \to \text{a}S\text{b}S \to \text{ab}S\text{a}S\text{b}S \xrightarrow{*} \text{abab}$$

5. This grammar is ambiguous. Consider the string `a+a+a`. We show two leftmost derivations for it.

$$S \to S\text{+}S \to S\text{+}S\text{+}S \xrightarrow{*} \text{a+a+a}$$
$$S \to S\text{+}S \to \text{a+}S \to \text{a+}S\text{+}S \xrightarrow{*} \text{a+a+a}$$

Exercise 2.2.4 Construct unambiguous context-free grammars for each of the following languages. In each case show that your grammar is correct.

a) Arithmetic expressions in postfix notation.

b) Left-associative lists of identifiers separated by commas.

c) Right-associative lists of identifiers separated by commas.

d) Arithmetic expressions of integers and identifiers with the four binary operators +, -, *, /.

e) Add unary plus and minus to the operators of (d).

Solution

a) The grammar is

$$E \to E \ E \ op \mid num$$
$$op \to \text{+} \mid \text{-} \mid \text{*} \mid \text{/}$$

We prove that each string generated by this grammar is generated unambiguously by induction on the number of operations. For zero operations, the expression $E \to num$ is unambiguously generated. Suppose that all expressions with upto k operations are unambiguously generated. Then, any postfix operation with $k+1$ operations is of the form $E_1 \ E_2 \ op$ where E_1 and E_2 are unambiguously generated postfix expressions with upto k operations. Thus, E is derived unambiguously as

$$E \to E \ E \ op \xrightarrow{*} E_1 \ E \ op \xrightarrow{*} E_1 \ E_2 \ op$$

which proves the claim.

b) The grammar is $L \rightarrow L,\ id \mid id$. We show that this grammar is unambiguous by induction on the number of identifiers. For one identifier, $L \rightarrow id$ is derived unambiguously. Consider a list $L = L_1,\ id$ of $k+1$ identifiers. We derive L as shown.

$$L \rightarrow L,\ id \xrightarrow{*} L_1,\ id$$

which proves the claim.

c) The grammar is $L \rightarrow id,\ L \mid id$. Proof is similar to the left associative case.

d) The grammar is

$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T * F \mid T\ /\ F \mid F$$
$$F \rightarrow id \mid num \mid (E)$$

We see that this grammar is unambiguous because of the fact that the expression is built term by term from the rightmost end (represented by T).

e) The grammar is

$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T * U \mid T\ /\ U \mid U$$
$$U \rightarrow + F \mid - F \mid F$$
$$F \rightarrow id \mid num \mid (E)$$

Proof of unambiguity is similar to the above exercise.

Exercise 2.2.5

a) Show that all binary strings generated by the following grammar have values divisible by 3. *Hint.* Use induction on the number of nodes in a parse tree.

$$num \rightarrow 11 \mid 1001 \mid num\ 0 \mid num\ num$$

b) Does the grammar generate all binary strings with values divisible by 3?

Solution

a) We prove the claim by induction on the number of nodes in a parse tree. For two nodes (including the root), we have two possibilities: $S \to (11|1001)$ and both numbers are divisible by 3. This is the base case. Suppose that the claim holds for all parse trees with upto k nodes. Any parse tree with $k + 1$ nodes is of one of the two following forms.

  i. $num \to num'\ 0$, where num' has $k - 1$ nodes in its parse tree.
  ii. $num \to num'\ num''$, where num' and num'' have upto k nodes each in their parse trees, since the total number of nodes in both parse trees sums up to k.

Clearly, in both cases, num will be divisible by 3 by using the inductive hypothesis, which proves our claim.

Exercise 2.2.6   Construct a context-free grammar for roman numerals.

Solution   The context free grammar is as shown.

$$num \to Th\ H\ T\ U$$
$$Th \to \text{M} \mid \text{MM} \mid \text{MMM} \mid \epsilon$$
$$H \to H' \mid \text{D}H' \mid \text{CD} \mid \text{CM}$$
$$H' \to \text{C} \mid \text{CC} \mid \text{CCC} \mid \epsilon$$
$$T \to T' \mid \text{L}T' \mid \text{XL} \mid \text{XC}$$
$$T' \to \text{X} \mid \text{XX} \mid \text{XXX} \mid \epsilon$$
$$U \to U' \mid \text{V}U' \mid \text{IV} \mid \text{IX}$$
$$U' \to \text{I} \mid \text{II} \mid \text{III} \mid \epsilon$$

2.3   Syntax-Directed Translation

Exercise 2.3.1   Construct a syntax directed translation scheme that translates arithmetic expressions from infix notation into prefix notation in which an operator appears before its operands; e.g., $-xy$ is the prefix notation for $x - y$. Give annotated parse trees for the inputs `9-5+2` and `9-5*2`.

Solution   The translation scheme is given below.

$$E \to \{\text{print('+')}\}\ E + T \mid \{\text{print('-')}\}\ E - T \mid T$$
$$T \to \{\text{print('*')}\}\ T * F \mid \{\text{print('/')}\}\ T\ /\ F \mid F$$
$$F \to \{\text{print(id)}\}\ id \mid \{\text{print(num)}\}\ num \mid (E)$$

The parse trees for the inputs `9-5+2` and `9-5*2` are shown in Figure 2.2 and Figure 2.3 respectively.
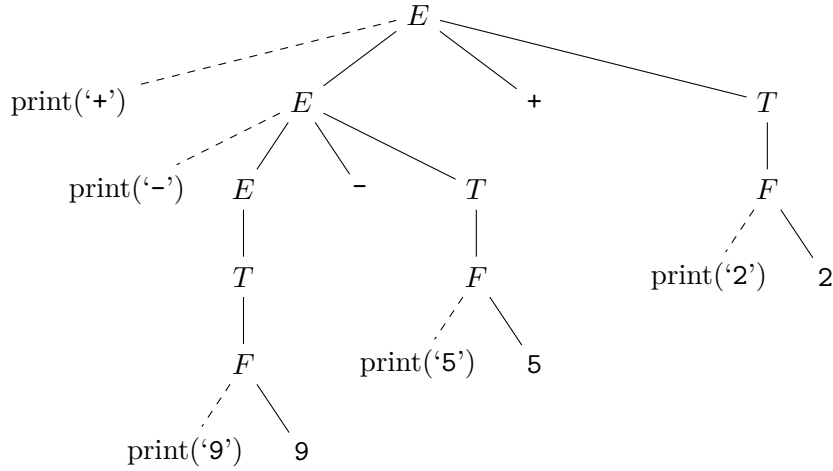
Figure 2.2: Parse tree for the input `9-5+2` for Exercise 2.3.1.

Figure 2.3: Parse tree for the input `9-5*2` for Exercise 2.3.1.

Exercise 2.3.2 Construct a syntax directed translation scheme that translates arithmetic expressions from postfix notation into infix notation. Give annotated parse trees for the inputs `95-2*` and `952*-`.

Solution The translation scheme is given below.

$$
\begin{aligned}
E \rightarrow\ & E\ \{\text{print('+')}\}\ E\ \texttt{+} \\
\mid\ & E\ \{\text{print('-')}\}\ E\ \texttt{-} \\
\mid\ & E\ \{\text{print('*')}\}\ E\ \texttt{*} \\
\mid\ & E\ \{\text{print('/')}\}\ E\ \texttt{/} \\
\mid\ & \{\text{print('num')}\}\ num
\end{aligned}
$$

The parse trees for the inputs `95-2*` and `952*-` are shown in Figure 2.4 and Figure 2.5.

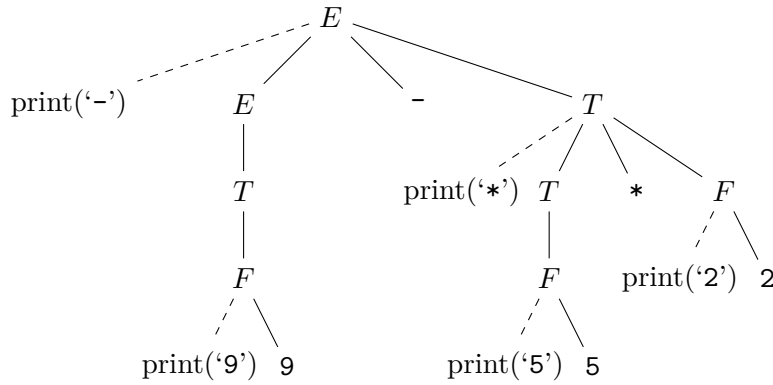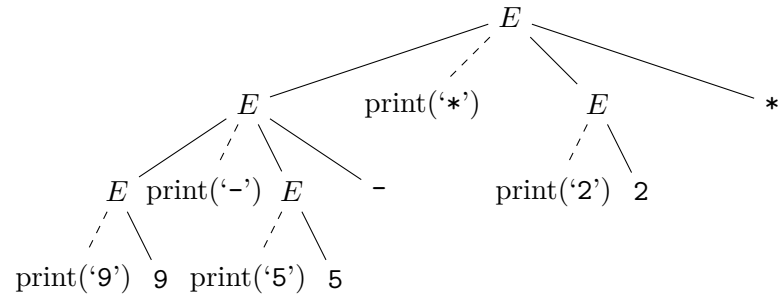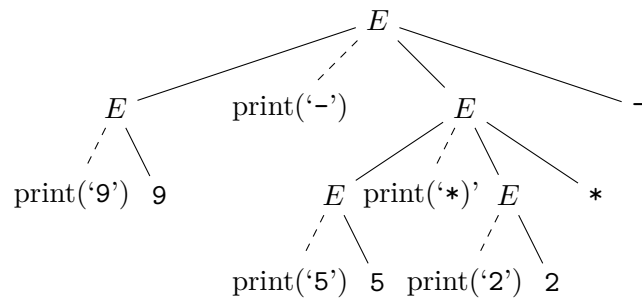Figure 2.4: Parse tree for the input `95-2*` for Exercise 2.3.2.

Figure 2.5: Parse tree for the input `952*-` for Exercise 2.3.2.

Exercise 2.3.3   Construct a syntax directed translation scheme that translates integers into roman numerals.

Solution   Using the grammar from Exercise 2.2.6, the translation scheme is

$$num \rightarrow Th\ H\ T\ U\ \ \{num.rm = Th.rm\ \|\ H.rm\|\ T.rm\|\ U.rm\}$$
$$Th \rightarrow low\ \ \{Th.rm = rep(\text{‘M’}, low.val)\}$$
$$H \rightarrow low\ \ \{H.rm = rep(\text{‘C’}, low.val)\}$$
$$|\ 4\ \{H.rm = \text{‘CD’}\}$$
$$|\ high\ \ \{H.rm = \text{‘D’}\ \|\ rep(\text{‘C’}, high.val - 5)\}$$
$$|\ 9\ \{H.rm = \text{‘CM’}\}$$
$$T \rightarrow low\ \ \{T.rm = rep(\text{‘X’}, low.val)\}$$
$$|4\ \{T.rm = \text{‘XL’}\}$$
$$|high\ \ \{T.rm = \text{‘L’}\ \|\ rep(\text{‘X’}, high.val - 5)\}$$
$$|9\ \{T.rm = \text{‘XC’}\}$$
$$U \rightarrow low\ \ \{U.rm = rep(\text{‘I’}, low.val)\}$$
$$|4\ \{U.rm = \text{‘IV’}\}$$
$$|high\ \ \{U.rm = \text{‘V’}\ \|\ rep(\text{‘I’}, high.val - 5)\}$$
$$|9\ \{U.rm = \text{‘IX’}\}$$
$$low \rightarrow 0\ \{low.val = 0\}$$
$$|1\ \{low.val = 1\}$$
$$|2\ \{low.val = 2\}$$
$$|3\ \{low.val = 3\}$$
$$high \rightarrow 5\ \{high.val = 5\}$$
$$|6\ \{high.val = 6\}$$
$$|7\ \{high.val = 7\}$$
$$|8\ \{high.val = 8\}$$

Exercise 2.3.4   Construct a syntax directed translation scheme that translates roman numerals up to 2000 into integers.

Solution   The translation scheme is shown below.

$$R \to Th\ H\ T\ U\ \{R.val = Th.val \parallel H.val \parallel T.val \parallel U.val\}$$
$$Th \to \text{M}\ \{Th.val = 1\}$$
$$\mid \text{MM}\ \{Th.val = 2\}$$
$$\mid \text{MMM}\ \{Th.val = 3\}$$
$$\mid \epsilon\ \{Th.val = 0\}$$
$$H \to H'\ \{H.val = H'.val\}$$
$$\mid \text{D}H'\ \{H.val = H'.val + 5\}$$
$$\mid \text{CD}\ \{H.val = 4\}$$
$$\mid \text{CM}\ \{H.val = 9\}$$
$$H' \to \text{C}\ \{H'.val = 1\}$$
$$\mid \text{CC}\ \{H'.val = 2\}$$
$$\mid \text{CCC}\ \{H'.val = 3\}$$
$$\mid \epsilon\ \{H'.val = 0\}$$
$$T \to T'\ \{T.val = T'.val\}$$
$$\mid \text{L}T'\ \{T.val = T'.val + 5\}$$
$$\mid \text{XL}\ \{T.val = 4\}$$
$$\mid \text{XC}\ \{T.val = 9\}$$
$$T' \to \text{X}\ \{T'.val = 1\}$$
$$\mid \text{XX}\ \{T'.val = 2\}$$
$$\mid \text{XXX}\ \{T'.val = 3\}$$
$$\mid \epsilon\ \{T'.val = 0\}$$
$$U \to U'\ \{U.val = U'.val\}$$
$$\mid \text{V}U'\ \{U.val = U'.val + 5\}$$
$$\mid \text{IV}\ \{U.val = 4\}$$
$$\mid \text{IX}\ \{U.val = 9\}$$
$$U' \to \text{I}\ \{U'.val = 1\}$$
$$\mid \text{II}\ \{U'.val = 2\}$$
$$\mid \text{III}\ \{U'.val = 3\}$$
$$\mid \epsilon\ \{U'.val = 0\}$$

Exercise 2.3.5   Construct a syntax directed translation scheme to translate postfix arithmetic expressions into equivalent prefix arithmetic expressions.

Solution The translation scheme is given below.

$$E \rightarrow E_1 \ E_2 \ \texttt{+} \ \{E.expr = \texttt{+} \parallel E_1.expr \parallel E_2.expr\}$$
$$| \ E_1 \ E_2 \ \texttt{-} \ \{E.expr = \texttt{-} \parallel E_1.expr \parallel E_2.expr\}$$
$$| \ E_1 \ E_2 \ \texttt{*} \ \{E.expr = \texttt{*} \parallel E_1.expr \parallel E_2.expr\}$$
$$| \ E_1 \ E_2 \ \texttt{/} \ \{E.expr = \texttt{/} \parallel E_1.expr \parallel E_2.expr\}$$
$$| \ id \ \{E.expr = id.val\}$$

2.4   Parsing

Exercise 2.4.1   Construct recursive-descent parsers, starting with the following grammars:

a) $S \rightarrow$ `+` $S \ S \ | $ `-` $S \ S \ | \ $`a`

b) $S \rightarrow S \ (\ S \ ) \ S \ | \ \epsilon$

c) $S \rightarrow 0 \ S \ 1 \ | \ 0 \ 1$

Solution

a) The pseudocode is shown in Listing 2.1.

b) The pseudocode is shown in Listing 2.2.

c) The pseudocode is shown in Listing 2.3.

2.6   Lexical Analysis

Exercise 2.6.1   Extend the lexical analyzer in Section 2.6.5 to remove comments, defined as follows:

a) A comment begins with `//` and includes all characters until the end of that line.

b) A comment begins with `/*` and includes all characters through the next occurrence of the character sequence `*/`.

Exercise 2.6.2   Extend the lexical analyzer in Section 2.6.5 to recognize the relational operators `<`, `<=`, `==`, `!=`, `>=`, `>`.

Exercise 2.6.3   Extend the lexical analyzer in Section 2.6.5 to recognize floating point numbers such as `2.`, `3.14`, and `.5`.

Solution   The extended lexical analyzer code can be viewed here.

```
1  void S() {
2      switch (lookahead) {
3          case '+':
4              match('+'); S(); S();
5              break;
6          case '-':
7              match('-'); S(); S();
8              break;
9          case 'a':
10             match('a');
11             break;
12         default:
13             report("syntax error");
14     }
15 }
16
17 void match(terminal t) {
18     if (lookahead == t) lookahead = nextTerminal;
19     else report("syntax error");
20 }
```

Listing 2.1: Pseudocode for recursive descent parser for Exercise 2.4.1 a).

```
1  void S() {
2      if (lookahead == '(') {
3          match('('); S(); match(')'); S();
4      }
5  }
6
7  void match(terminal t) {
8      if (lookahead == t) lookahead = nextTerminal;
9      else report("syntax error");
10 }
```

Listing 2.2: Pseudocode for recursive descent parser for Exercise 2.4.1 b).

2.8   Intermediate Code Generation

Exercise 2.8.1   For-statements in C and Java have the form:

$$\texttt{for} \ (\ expr_1 \ ; \ expr_2 \ ; \ expr_3 \ ) \ stmt$$

```
1   void S() {
2       switch (lookahead) {
3           case '0':
4               match('0'); S(); match('1');
5           case '1':        // Fallthrough!
6               break;
7           default:
8               report("syntax error");
9       }
10  }
11
12  void match(terminal t) {
13      if (lookahead == t) lookahead = nextTerminal;
14      else report("syntax error");
15  }
```

Listing 2.3: Pseudocode for recursive descent parser for Exercise 2.4.1 c).

The first expression is executed before the loop; it is typically used for initializing the loop index. The second expression is a test made before each iteration of the loop; the loop is exited if the expression becomes 0. The loop itself can be thought of as the statement { *stmt expr*$_3$; }. The third expression is executed at the end of each iteration; it is typically used to increment the loop index. The meaning of the for-statement is similar to

$$expr_1; \texttt{ while (} expr_2 \texttt{) \{ } stmt\ expr_3; \texttt{ \} }$$

Define a class *For* for for-statements, similar to class *If* in Fig. 2.43.

Solution   The class is shown in Listing 2.4.

Exercise 2.8.2   The programming language C does not have a boolean type. Show how a C compiler might translate an if-statement into three-address code.

Solution   Suppose that the if-statment is of the form `if ( ` *expr* ` ) ` *stmt*;. The equivalent three-address code would be as follows.

```java
class For extends Stmt {
    Expr E1, E2, E3;
    Stmt S;
    public For(Expr e1, Expr e2, Expr e3, Stmt s) {
        E1 = e1; E2 = e2; E3 = e3; S = s;
    }
    public void gen() {
        E1.gen();
        // Give a label for start and end
        Label start = new Label();
        Label end = new Label();
        emit(start + " ifFalse" + E2.rvalue().toString() + "
 ↪  goto " + end);
        S.gen();
        E3.gen();
        // Unconditional jump back to label for checking
        emit("goto " + start);
        // Code after the loop
        emit(end + ":");
    }
}
```

Listing 2.4: Implementation of the *For* class in Java.

```
1  class If extends Stmt {
2      Expr E;
3      Stmt S;
4      public For(Expr e, Stmt s) {
5          E = e; S = s;
6      }
7      public void gen() {
8          E.gen();
9          Label end = new Label();
10         emit("ifEqual " + E.rvalue().toString() + " 0 goto "
           ↪  + end);
11         S.gen();
12         // Code after the loop
13         emit(end + ":");
14      }
15 }
```

Listing 2.5: Generation of three-address code for if-statments in C.

3

Lexical Analysis

3.1 The Role of the Lexical Analyzer

Exercise 3.1.1 Divide the following C++ program:

```
1  float limitedSquare(x) float x; {
2      /* returns x-squared, but never more than 100 */
3      return (x<=-10.0||x>=10.0)?100:x*x;
4  }
```

Listing 3.1: Source code for Exercise 3.1.1.

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?

Solution The lexemes are divided as shown in Table 3.1.

Exercise 3.1.2 Tagged languages like HTML or XML are different from
conventional programming languages in that the punctuation (tags) are ei-
ther very numerous (as in HTML) or a user-definable set (as in XML). Fur-
ther, tags can often have parameters. Suggest how to divide the following
HTML document:
into appropriate lexemes. Which lexemes should get associated lexical val-
ues, and what should those values be?

Solution The lexemes are divided as in Table 3.2.

<**float**>

<**id**, pointer to symbol-table entry for `limitedSquare`>

<**(**>

<**id**, pointer to symbol-table entry for `x`>

<**)**>

<**;**>

<**{**>

<**return**>

<**<=**>

<**number**, float value `-10.0`>

<**||**>

<**>=**>

<**number**, float value `10.0`>

<**?**>

<**number**, integer value `100`>

<**:**>

<*****>

<**}**>

Table 3.1: Lexemes and their associated vales in Listing 3.1.

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

Listing 3.2: Source code for Exercise 3.1.2.

3.3   Specification of Tokens

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) `a(a|b)*a`.

b) `((ε|a)b*)*`.

<**text**, Here is a photo of >
<**tag_start**, B>
<**text**, my house>
<**tag_end**, B>
<**text**, :>
<**tag_start**, P>
<**self_end_tag**, IMG>
<**attribute**, {SRC: "house.gif"}>
<**self_end_tag**, BR>
<**text**, See >
<**tag_start**, A>
<**attribute**, {HREF: "morePix.html"}>
<**text**, More Pictures>
<**tag_end**, A>
<**text**, if you liked that one.>
<**tag_end**, P>

Table 3.2: Lexemes and their associated vales in Listing 3.2.

c) `(a|b)*a(a|b)(a|b)`.

d) `a*ba*ba*ba*`.

e) `(aa|bb)*((ab|ba)(aa|bb)*(ab|ba)(aa|bb)*)*`.

Solution

a) Strings that begin and end with `a`.

b) `(a|b)*`.

c) Strings whose third to last character is `a`.

d) Strings that contain exactly three occurrences of `b`.

e) Strings that have an even difference in the occurrences of `a` and `b`.

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.

Solution

    a) There are n prefixes, determined by the ending index.

    b) There are n suffixes, determined by the starting index.

    c) Since the string itsef is a prefox but not a proper prefix of itself, we have $n-1$ proper prefixes.

    d) Any two indices determine the start and end of a substring. Thus, we have $\binom{n+1}{2}$ substrings.

    e) For each character of the string, we can choose to either include or exclude it, for a total of 2^n subsequences.

Exercise 3.3.4 Most languages are *case sensitive*, so keywords can be written only one way, and the regular expressions describing their lexemes are 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.

Solution For a keyword $s = s_1 s_2 \ldots s_n$, the required regular expression is

$$s' = (s_1|S_1)\,(s_2|S_2)\ldots(s_n|S_n)$$

where S_i denotes the opposite case of s_i. For "select", the regular expression is `(s|S)(e|E)(l|L)(e|E)(c|C)(t|T)`.

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×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`.

Solution

a) The regular expression is `C*a[Ca]*e[Ce]*i[Ci]*o[Co]*u[Cu]*`. C is the shorthand for all consonants `[^aeiou]`.

b) The regular expression is `a*b*...z*`.

c) The regular expression is `\/\*(\".*\"|[^\*\"]*|\*[^\/]*)*\*\/`.

d) We describe the regular definition by describing an appropriate NFA $N(Q, \Sigma, \delta, q_s, F)$.

   i) $Q = \{q_s, q_0, \ldots, q_9\}$, where each state except q_s represents the last scanned digit.

   ii) $\Sigma = \{\texttt{0-9}\}$.

   iii) δ is given by

$$\delta(q, i) = \begin{cases} \{q_i\} & q = q_s, \ i \in \Sigma \\ \{q_i\} & q = q_j, \ j \in \Sigma, \ i \neq j \end{cases}$$

   iv) q_s is the start state.

   v) $F = Q \setminus \{q_s\}$ are the accepting states.

e) Similar to the above expression, we describe the regular definition by describing an appropriate NFA $N(Q, \Sigma, \delta, q_s, F)$.

   i) $Q = \{q_s, q_0, q_0', \ldots, q_9, q_9'\}$, where each state except q_s represents the last scanned digit. Additionally, the states marked with a ' denote that there has been a repeated digit.

   ii) $\Sigma = \{\texttt{0-9}\}$.

   iii) δ is given by

$$\delta(q, i) = \begin{cases} \{q_i\} & q = q_s, \ i \in \Sigma \\ \{q_i'\} & q = q_i, \ i \in \Sigma \\ \{q_i\} & q = q_j, \ j \in \Sigma, \ i \neq j \\ \{q_i'\} & q = q_j', \ j \in \Sigma, \ i \neq j \end{cases}$$

   iv) q_s is the start state.

      v) $F = Q \setminus \{q_s\}$ are the accepting states.

f) We describe a DFA $M(Q, \Sigma, \delta, q_0, F)$ that recognizes the required language.

    i) $Q = \{q_{ij} \mid 0 \leq i, j \leq 1\}$. Here, the first index represents the parity of `a` and the second index represents the parity of `b` in the string.

    ii) $\Sigma = \{\texttt{a-b}\}$.

    iii) δ is given in Table 3.3.

$\delta\,(q, a)$	a	b
q_{00}	q_{10}	q_{01}
q_{01}	q_{11}	q_{00}
q_{10}	q_{00}	q_{11}
q_{11}	q_{01}	q_{10}

Table 3.3: $\delta(q, a)$ for DFA M.

.

    iv) $q_0 = q_{00}$.

    v) $F = \{q_{01}\}$.

g)

h)