Compilers: Principles, Techiniques and Tools

Solutions Manual

Gautam Singh

May 22, 2023

Contents

1 Introduction **1**
1.1 Language Processors 1
1.3 The Evolution of Programming Languages 2
1.6 Programming Language Basics 3

2 A Simple Syntax-Directed Translator **7**
2.2 Syntax Definition 7
2.3 Syntax-Directed Translation 12
2.4 Parsing 17
2.6 Lexical Analysis 17
2.8 Intermediate Code Generation 18

3 Lexical Analysis **23**
3.1 The Role of the Lexical Analyzer 23
3.3 Specification of Tokens 24
3.4 Recognition of Tokens 31

List of Figures

2.1 Parse tree for the string aa+a*. 7
2.2 Parse tree for the input 9-5+2 for Exercise 2.3.1. 13
2.3 Parse tree for the input 9-5*2 for Exercise 2.3.1. 13
2.4 Parse tree for the input 95-2* for Exercise 2.3.2. 14
2.5 Parse tree for the input 952*- for Exercise 2.3.2. 14

List of Tables

1.1 Differences Between Compilers and Interpreters. 1
1.2 Scopes of variables declared in Listing 1.3. 5

3.1 Lexemes and their associated vales in Listing 3.1. 24
3.2 Lexemes and their associated vales in Listing 3.2. 25
3.3 $\delta(q, a)$ for DFA M. 28
3.4 Filename expressions used by the shell command `sh`. 30
3.5 Equivalent regular expressions for SQL patterns. 31
3.6 Failure function for `ababababaab`. 31
3.7 Failure function for `aaaaaa`. 32
3.8 Failure function for `abbaabb`. 32
3.9 Failure function for keyword `ababaa`. 33
3.10 KMP algorithm for `ababababaab`. 33
3.11 KMP algorithm for `ababababaab`. 33

List of Listings

1.1 Source code for Exercise 1.6.1 3
1.2 Source code for Exercise 1.6.2 4
1.3 Source code for Exercise 1.6.3 5
1.4 Source code for Exercise 1.6.4. 6
1.5 Output on running Listing 1.4. 6
2.1 Pseudocode for recursive descent parser for Exercise 2.4.1 a). 18
2.2 Pseudocode for recursive descent parser for Exercise 2.4.1 b). 18
2.3 Pseudocode for recursive descent parser for Exercise 2.4.1 c). 19
2.4 Implementation of the For class in Java. 20
2.5 Generation of three-address code for if-statments in C. 21
3.1 Source code for Exercise 3.1.1. 23
3.2 Source code for Exercise 3.1.2. 24
3.3 Algorithm to compute the failure function for keyword $b_1 b_2 \ldots b_n$. 32
3.4 The KMP Algorithm. 34

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.

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.

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

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

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`.

```
#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(); }
```

Listing 1.4: Source code for Exercise 1.6.4.

The final output is shown in Listing 1.5.

```
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

$$S \rightarrow S\ S\ \texttt{+} \mid S\ S\ \texttt{*} \mid \texttt{a}$$

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.

$$S \rightarrow SS\texttt{*} \rightarrow SS\texttt{+}S\texttt{*} \rightarrow \texttt{a}S\texttt{+}S\texttt{*} \rightarrow \texttt{aa+}S\texttt{*} \rightarrow \texttt{aa+a*}$$

b) The parse tree for the above string is in Figure 2.1.

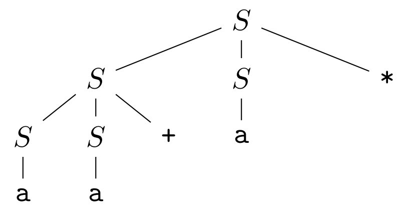

Figure 2.1: Parse tree for the string `aa+a*`.

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 $S \to \texttt{a}$, we see that any string generated by this grammar has odd length.

   The base case ($n = 0$) consists of the derivation $S \to \texttt{a}$, 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 $s = xy\texttt{b}$ 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

$$S \to SS\texttt{b} \overset{*}{\to} xS\texttt{b} \overset{*}{\to} xy\texttt{b}$$

   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) $S \to 0\ S\ 1 \mid 0\ 1$

b) $S \to \texttt{+}\ S\ S \mid \texttt{-}\ S\ S \mid \texttt{a}$

c) $S \to S\ \texttt{(}\ S\ \texttt{)}\ S \mid \epsilon$.

d) $S \to \texttt{a}\ S\ \texttt{b}\ S \mid \texttt{b}\ S\ \texttt{a}\ S \mid \epsilon$

e) $S \to \texttt{a} \mid S\ \texttt{+}\ S \mid S\ S \mid S\ {}^{*} \mid \texttt{(}\ S\ \texttt{)}$

Solution

a) We prove that the language generated is $\{0^n1^n \mid n > 0\}$ by induction on n. For $n = 1$, we see that $S \to 01$. Suppose $S \overset{*}{\to} 0^k1^k$ for some $k > 0$. Then, we have

$$S \to 0S1 \overset{*}{\to} 00^k1^k1 = 0^{k+1}1^{k+1}$$

   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

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

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

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

   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, $S \to \texttt{a}$ is in this language. If any regular expressions R and T are in this language, then

$$\begin{aligned}
&S \to (S) \xrightarrow{*} (R)\\
&S \to S\,{}^{*} \xrightarrow{*} R\,{}^{*}\\
&S \to S + S \xrightarrow{*} R + T\\
&S \to SS \xrightarrow{*} RT
\end{aligned}$$

   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.

$$\begin{aligned}
&S \to S\texttt{(}S\texttt{)}S \to \texttt{(}S\texttt{)}S \to \texttt{()}S \to \texttt{()}S\texttt{(}S\texttt{)}S \xrightarrow{*} \texttt{()()}\\
&S \to S\texttt{(}S\texttt{)}S \to S\texttt{(}S\texttt{)}S\texttt{(}S\texttt{)}S \xrightarrow{*} \texttt{()()}
\end{aligned}$$

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

$$S \rightarrow \mathtt{a}S\mathtt{b}S \rightarrow \mathtt{ab}S \rightarrow \mathtt{ab}S \xrightarrow{*} \mathtt{abab}$$
$$S \rightarrow \mathtt{a}S\mathtt{b}S \rightarrow \mathtt{ab}S\mathtt{a}S\mathtt{b}S \xrightarrow{*} \mathtt{abab}$$

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

$$S \rightarrow S\mathtt{+}S \rightarrow S\mathtt{+}S\mathtt{+}S \xrightarrow{*} \mathtt{a+a+a}$$
$$S \rightarrow S\mathtt{+}S \rightarrow \mathtt{a+}S \rightarrow \mathtt{a+}S\mathtt{+}S \xrightarrow{*} \mathtt{a+a+a}$$

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

$$E \rightarrow E\ E\ op \mid num$$
$$op \rightarrow \mathtt{+} \mid \mathtt{-} \mid \mathtt{*} \mid \mathtt{/}$$

We prove that each string generated by this grammar is generated unambiguously by induction on the number of operations. For zero operations, the expression $E \rightarrow num$ 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

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

which proves the claim.

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

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

which proves the claim.

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

d) The grammar is

$$\begin{aligned} E &\to E \texttt{ + } T \mid E \texttt{ - } T \mid T \\ T &\to T \texttt{ * } F \mid T \texttt{ / } F \mid F \\ F &\to id \mid num \mid (E) \end{aligned}$$

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

$$\begin{aligned} E &\to E \texttt{ + } T \mid E \texttt{ - } T \mid T \\ T &\to T \texttt{ * } U \mid T \texttt{ / } U \mid U \\ U &\to \texttt{+ } F \mid \texttt{- } F \mid F \\ F &\to id \mid num \mid (E) \end{aligned}$$

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.

$$num \to 11 \mid 1001 \mid num\ 0 \mid num\ num$$

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.

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

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.

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

The parse trees for the inputs `9-5+2` and `9-5*2` are shown in Figure 2.2 and Figure 2.3 respectively.

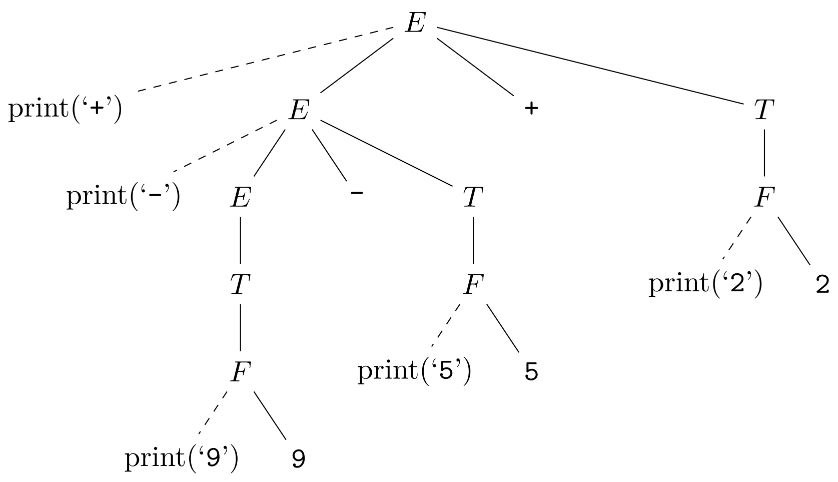

Figure 2.2: Parse tree for the input `9-5+2` for Exercise 2.3.1.

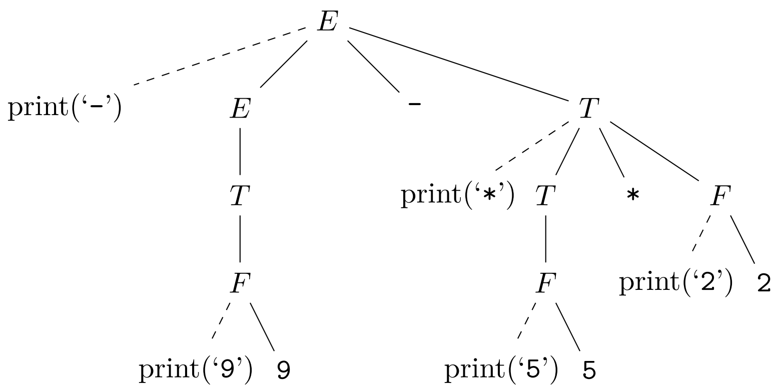

Figure 2.3: Parse tree for the input `9-5*2` for Exercise 2.3.1.

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.

$$\begin{aligned} E \to\ & E\ \{\text{print}(\text{`}\texttt{+}\text{'})\}\ E\ \texttt{+} \\ |\ & E\ \{\text{print}(\text{`}\texttt{-}\text{'})\}\ E\ \texttt{-} \\ |\ & E\ \{\text{print}(\text{`}\texttt{*}\text{'})\}\ E\ \texttt{*} \\ |\ & E\ \{\text{print}(\text{`}\texttt{/}\text{'})\}\ E\ \texttt{/} \\ |\ & \{\text{print}(\text{`}\texttt{num}\text{'})\}\ \mathit{num} \end{aligned}$$

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

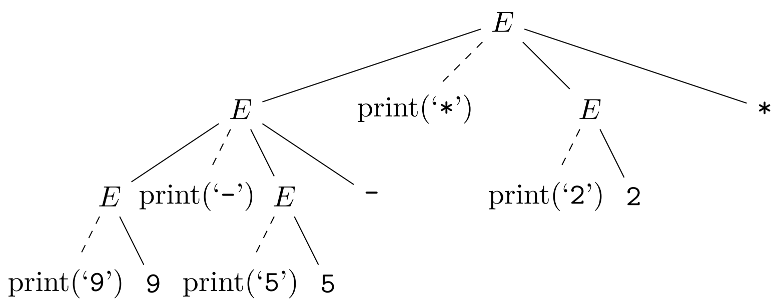

Figure 2.4: Parse tree for the input `95-2*` for Exercise 2.3.2.

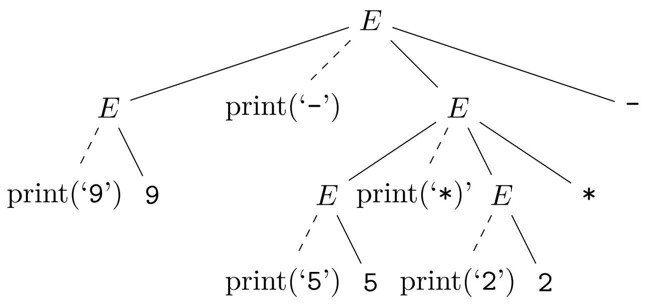

Figure 2.5: Parse tree for the input `952*-` for Exercise 2.3.2.

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

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

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.

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

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.

$$\begin{aligned}E \rightarrow\ & E_1\ E_2\ \texttt{+}\ \{E.expr = \texttt{+} \parallel E_1.expr \parallel E_2.expr\}\\ \mid\ & E_1\ E_2\ \texttt{-}\ \{E.expr = \texttt{-} \parallel E_1.expr \parallel E_2.expr\}\\ \mid\ & E_1\ E_2\ \texttt{*}\ \{E.expr = \texttt{*} \parallel E_1.expr \parallel E_2.expr\}\\ \mid\ & E_1\ E_2\ \texttt{/}\ \{E.expr = \texttt{/} \parallel E_1.expr \parallel E_2.expr\}\\ \mid\ & id\ \{E.expr = id.val\}\end{aligned}$$

2.4 Parsing

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

a) $S \rightarrow\ \texttt{+}\ S\ S \mid \texttt{-}\ S\ S \mid \texttt{a}$

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

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

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.

```
void S() {
    switch (lookahead) {
        case '+':
            match('+'); S(); S();
            break;
        case '-':
            match('-'); S(); S();
            break;
        case 'a':
            match('a');
            break;
        default:
            report("syntax error");
    }
}

void match(terminal t) {
    if (lookahead == t) lookahead = nextTerminal;
    else report("syntax error");
}
```

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

```
void S() {
    if (lookahead == '(') {
        match('('); S(); match(')'); S();
    }
}

void match(terminal t) {
    if (lookahead == t) lookahead = nextTerminal;
    else report("syntax error");
}
```

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:

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

```
void S() {
    switch (lookahead) {
        case '0':
            match('0'); S(); match('1');
        case '1':       // Fallthrough!
            break;
        default:
            report("syntax error");
    }
}

void match(terminal t) {
    if (lookahead == t) lookahead = nextTerminal;
    else report("syntax error");
}
```

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

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

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.

```java
class If extends Stmt {
    Expr E;
    Stmt S;
    public For(Expr e, Stmt s) {
        E = e; S = s;
    }
    public void gen() {
        E.gen();
        Label end = new Label();
        emit("ifEqual " + E.rvalue().toString() + " 0 goto "
        ↪  + end);
        S.gen();
        // Code after the loop
        emit(end + ":");
    }
}
```

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:

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

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 di erent from conventional programming languages in that the punctuation (tags) are either very numerous (as in HTML) or a user-de nable set (as in XML). Further, tags can often have parameters. Suggest how to divide the following HTML document:

into appropriate lexemes. Which lexemes should get associated lexical values, 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.

```
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>
```

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 $s = s_1 s_2 \dots s_n$, the required regular expression is

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

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) $\Sigma = \{$`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) $q_0 = q_s$.

   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) $\Sigma = \{$`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) $q_0 = q_s$.

   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) $\Sigma = \{$`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) The regular expression is `b*(a+b?)*`.

i) The regular expression is `b*a*b?a*`.

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).

Solution

a) The character class is `[a-jA-J]`.

b) The character class is `[a-z]`.

c) The character class is `[0-9a-fA-F]`.

d) The character class is `[\.?!]`.

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 `"\`.

Solution The required regular expression is `\"\\`.

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.

Solution We can simply substitute the complemented character classes with character classes that contain all but the complemented characters. Since the character class of a language is finite, we can always do this, and not change the meaning of the regular expression.

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.

Solution We use the zero or one operator (`?`) to construct the equivalent regular expression

$$r\,\{m,n\} \equiv \underbrace{rr\ldots r}_{m \text{ times}}\underbrace{r?r?\ldots r?}_{n-m \text{ times}}$$

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?

Solution

a) We can distinguish the two meanings by checking whether the caret is present inside a character class right after the opening square bracket. If so, then the caret denotes negation, otherwise it denotes the start of a line.

b)

Exercise 3.3.11 The UNIX shell command `sh` uses the operators in Table 3.4 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 `sort1.`c, where c is any character. Show how `sh` filename expressions can be replaced by equivalent regular expressions using only the basic union, concatenation, and closure operators.

EXPRESSION	MATCHES	EXAMPLE	REGULAR EXPRESSION
's'	string s literally	`'\'`	(s)
'\\c'	character c literally	`\'`	(c)
*	any string	`*.o`	`(a|b|...)*.o`
?	any character	`sort1.?`	`sort1.a| sort1.b|...`
[s]	any character in s	`sort1.[cso]`	`sort1.c| sort1.s| sort1.o`

Table 3.4: Filename expressions used by the shell command `sh`.

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 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.

Solution Using `Lex` regular expression, we have the following equivalent regular expressions in Table 3.5.

SQL Pattern	Regular Expression
`_`	`.`
`%`	`.*`
ec	$\backslash c$

Table 3.5: Equivalent regular expressions for SQL patterns.

3.4 Recognition of Tokens

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.

Exercise 3.4.3 Construct the failure function for the strings:

a) `ababaabaab`.

b) `aaaaaa`.

c) `abbaabb`.

Solution

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

Table 3.6: Failure function for `ababaabaab`.

a)

b)

c)

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

Table 3.7: Failure function for `aaaaaa`.

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

Table 3.8: Failure function for `abbaabb`.

Exercise 3.4.4 Prove, by induction on s, that the algorithm of Listing 3.3 correctly computes the failure function.

```
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      }
9      else f(s + 1) = 0;
10 }
```

Listing 3.3: Algorithm to compute the failure function for keyword $b_1 b_2 \dots b_n$.

Solution We prove the following loop invariant: *At the end of the s^{th} iteration, $t = f(s+1)$ and the values of $f(k)$, $1 \le k \le s+1$.* are computed correctly.
Putting $s = 0$, we see from lines 1 and 2 of Listing 3.3 that the base case is trivial. Suppose the loop invariant holds upto the $(s-1)^{\text{th}}$ iteration $(s > 0)$. Then, t contains the value of $f(s)$ at the start of the s^{th} iteration.
We claim that the `while` loop of line 4 computes the maximal length t of the prefix of $b_1 b_2 \dots b_s$ that is also a suffix of this substring such that $b_{t+1} = b_{s+1}$ or returns $t = 0$. Since $t = f(s)$ at the start, in any iteration of the `while` loop, we have a prefix of length t that is also a suffix of $b_1 b_2 \dots b_s$. Note that this loop decreases the value of t. If at any point, we find a suitable t such that $b_{t+1} = b_{s+1}$, it must be maximal, and hence $t = f(s+1)$ is computed properly. If no such t is found, clearly $t = f(s+1) = 0$.
Putting $s = n-1$ in the loop invariant, we see that the algorithm is correct.

Exercise 3.4.5 Show that the assignment $t = f(t)$ in line 4 of Listing 3.3 is executed at most n times. Show that therefore, the entire algorithm takes only $O(n)$ time on a keyword of length n.

Solution Observe that the `if...else` statement following the `while` loop is executed $O(n)$ times. Each execution may increase the value of t by $O(n)$. On the other hand, each execution of the `while` loop decreases the value of t. Since $t \geq 0$, we must have that the `while` loop was executed $O(n)$ times, for a total complexity of $O(n)$.

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

a) `abababaab`.

b) `abababbaa`.

Solution We compute the failure function for `ababaa` in Table 3.9.

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

Table 3.9: Failure function for keyword `ababaa`.

a) We see in Table 3.10 that $s = 6$ when $m = 8$, hence the keyword is a substring.

m	1	2	3	4	5	6	7	8	9
s	1	2	3	4	5	4	5	6	

Table 3.10: KMP algorithm for `abababaab`.

b) We see in Table 3.11 that s is never equal to 6, hence the keyword is not a substring.

m	1	2	3	4	5	6	7	8	9
s	1	2	3	4	5	4	0	1	0

Table 3.11: KMP algorithm for `abababaab`.

Exercise 3.4.7 Show that the algorithm in Listing 3.4 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 $a_1a_2\ldots a_i$.

```
1 s = 0;
2 for (i = 1; i <= 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 }
7 return "no";
```

Listing 3.4: The KMP Algorithm.

Solution We claim 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 $a_1 a_2 \dots a_i$. We prove this claim by induction on i.
For $i = 1$, we must have $s = 1$ if $b_1 = a_1$ and $s = 0$ otherwise, which proves the base case. Suppose the claim holds upto some i. Then, for $i + 1$, we need to find the maximal s such that $a_{i-s+k} = b_k$, $1 \leq k \leq s + 1$. The `while` loop in line 3 reduces `s` to this maximal value. If no such value of s is found, then we must have $s = 0$. In either case, the induction hypothesis holds, and the claim is proved.
At each iteration of the `for` loop, we check whether $s = n$. This implies that keyword b is a suffix of $a_1 a_2 \dots a_i$ for some i. Therefore, we have found a match. If this situation does not occur, control exits the `for` loop and there is no match. This proves the correctness of the KMP algorithm.

Exercise 3.4.8 Show that the algorithm of Listing 3.4 runs in time $O(m+n)$, assuming that function f is already computed and its values stored in an array indexed by s.

Solution Notice that s can increase $O(n)$ times, since the algorithm will terminate if $s = n$. Thus, the `while` loop in line 3 can decrease it only $O(n)$ times, hence it executes $O(n)$ times in total. Since there are $O(m)$ iterations of lines 4 and 5, it follows that the total runtime is $O(m + n)$, as required.

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`, $s4 =$ `aba`, and $s5 =$ `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}?