Compilers: Principles, Techiniques and Tools

Solutions Manual

GAUTAM SINGH

May 9, 2023

Contents

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`.

```c
#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.

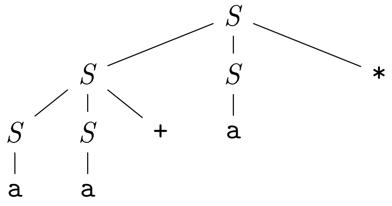

Figure 2.1: Parse tree for the string `aa+a*`.

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

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

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 $\{0^n 1^n \mid n > 0\}$ by induction on n. For $n = 1$, we see that $S \to 01$. Suppose $S \xrightarrow{*} 0^k 1^k$ for some $k > 0$. Then, we have

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

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

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

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,

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

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 containing a and the union and star operations along with parentheses. We prove this claim by definition. Clearly, $S \rightarrow \texttt{a}$ is in this language. If any regular expressions R and T are in this language, then

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

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.

$$S \rightarrow S\texttt{(}S\texttt{)}S \rightarrow \texttt{(}S\texttt{)}S \rightarrow \texttt{()}S \rightarrow \texttt{()}S\texttt{(}S\texttt{)}S \xrightarrow{*} \texttt{()()}$$
$$S \rightarrow S\texttt{(}S\texttt{)}S \rightarrow S\texttt{(}S\texttt{)}S\texttt{(}S\texttt{)}S \xrightarrow{*} \texttt{()()}$$

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 $L \rightarrow L,\ id \mid id$. 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.

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

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

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

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

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

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.

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

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: $S \rightarrow (11|1001)$ 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. $num \rightarrow num'\ 0$, where num' has $k-1$ nodes in its parse tree.

   ii. $num \rightarrow num'\ num''$, 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.

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