# EXERCISES

## Section 5.1

1. Draw parse trees, according to the grammar in Fig. 5.2, for the following <id-list>s:

   a. `ALPHA`

   b. `ALPHA, BETA, GAMMA`

2. Draw parse trees, according to the grammar in Fig. 5.2, for the following <exp>s:

   a. `ALPHA + BETA`

   b. `ALPHA - BETA * GAMMA`

   c. `ALPHA DIV (BETA + GAMMA) - DELTA`

3. Suppose Rules 10 and 11 of the grammar in Fig. 5.2 were changed to

```
<exp>   ::= <term> | <exp> * <term> | <exp> DIV <term>
<term>  ::= <factor> | <term> + <factor> | <term> - <factor>
```

   Draw the parse trees for the <exp>s in Exercise 2 according to this modified grammar. How has the change in the grammar affected the precedence of the arithmetic operators?

4. Assume that Rules 10 and 11 of the grammar in Fig. 5.2 are deleted and replaced with the single rule

```
<exp> ::= <factor> | <exp> + <factor> | <exp> - <factor>
                   | <exp> * <factor> | <exp> DIV <factor>
```

   Draw the parse trees for the <exp>s in Exercise 2 according to this modified grammar. How has the change in the grammar affected the precedence of the arithmetic operators?

5. Modify the grammar in Fig. 5.2 to include exponentiation operations of the form X↑Y. Be sure that exponentiation has higher priority than any other arithmetic operation.

6. Modify the grammar in Fig. 5.2 to include statements of the form

```
IF condition THEN statement-1 ELSE statement-2
```

where the ELSE clause may be omitted. Assume that the condition must be of the form $a < b$, $a = b$, or $a > b$, where $a$ and $b$ are single identifiers or integers. You do not need to allow for nested IFs—that is, *statement-1* and *statement-2* may not be IF statements.

7. Modify the grammar in Fig. 5.2 so that the I/O list for a WRITE statement may include character strings enclosed in quotation marks, as well as identifiers.

8. Write an algorithm that scans an input stream, recognizing operators and identifiers. An identifier may be up to 10 characters long. It must start with a letter, and the remaining characters, if any, must be letters and digits. The operators to be recognized are +, –, *, DIV, and :=. Your algorithm should return an integer that represents the type of token found, using the coding scheme of Fig. 5.5. If an illegal combination of characters is found, the algorithm should return the value –1.

9. Modify the scanner you wrote in Exercise 8 so that it recognizes integers as well as identifiers. Integers may begin with a sign (+ or –); however, they may not begin with the digit 0 (except for the integer that consists of a single 0).

10. Draw a state diagram for a finite automaton to recognize a token type named "real constant." This token consists of a string of digits that contains a decimal point. There must be at least one digit before the decimal point.

11. Modify your answer to Exercise 10 so that a real constant may also contain a scale factor. The scale factor, which follows the string of digits, consists of the letter E followed by a positive or negative integer. A real constant must contain either a decimal point or a scale factor (or both). There must be at least one digit before the decimal point (if any).

12. Draw a state diagram for a finite automaton to recognize a token type named "write-element." Each such token must have one of the following forms:

```
name
name:n
name:n:m
'string'
'string':n
```

where

*name* must start with a letter (a–z); all characters after the first letter must be either letters (a–z) or digits (0–9).

*string* may contain any characters other than quote (').

*n,m* must be positive integers containing only digits (0–9), with no leading zeros allowed.

13. Write a program that simulates the operation of a finite automaton, using a tabular representation like the one illustrated in Fig. 5.10(b).

14. Select a high-level programming language with which you are familiar and write a lexical scanner for it.

15. Parse the following statements from the example program in Fig. 5.1, using the operator-precedence technique and the precedence matrix in Fig. 5.11:
    a. the assignment statement on line 11
    b. the declaration on line 3
    c. the FOR statement beginning on line 7

16. Parse the entire program for Fig. 5.1, using the operator-precedence technique and the precedence matrix in Fig. 5.11.

17. Parse the assignment statement on line 11 of Fig. 5.1, using the method of recursive descent and the procedures given in Fig. 5.17.

18. Write recursive-descent parsing procedures that correspond to the rules for <dec-list>, <dec>, and <type> in Fig. 5.15. Use these procedures to parse the declaration on line 3 of Fig. 5.1.

19. Write recursive-descent parsing procedures for the remaining nonterminals in the grammar of Fig. 5.15. Parse the entire program in Fig. 5.1, using the method of recursive descent.

20. Use the routines in Figs. 5.18–5.20 to generate code for the following statements from the example program in Fig. 5.1:
    a. the assignment statement on line 11
    b. the WRITE statement on line 15
    c. the FOR statement beginning on line 7

Refer to the parse tree in Fig. 5.4 to see the order in which the parser recognizes the various constructs involved in these statements.

21. Use the routines in Figs. 5.18–5.20 to generate code for the entire program in Fig. 5.1.

22. Write code-generation routines for the new rules that you added to the grammar in Exercise 6 to define the IF statement.

23. Suppose that the grammar in Fig. 5.2 is modified to allow floating-point variables (i.e., the <type> REAL) as well as integers. How would the code-generation routines given in the text need to be changed? Assume that mixed-mode arithmetic expressions are allowed according to the usual rules of Pascal.

24. The code-generation routines in the text use immediate addressing for integers written by the programmer in arithmetic expressions (for example, the 100 in the expression SUM DIV 100). How could such constants be handled by a compiler for a machine that does not have immediate addressing?

25. What kinds of source program errors would be detected during lexical analysis?

26. What kinds of source program errors would be detected during syntactic analysis?

27. What kinds of source program errors would be detected during code generation?

28. In what ways might the symbol table used by a compiler be different from the symbol table used by an assembler?

29. Suppose you have a one-pass Pascal compiler similar to the one described in Section 5.1. Now you want to add a simple macro capability to this compiler. The macro processing should be integrated into the rest of the compiler, not implemented as a preprocessor. Describe how the macro processing routines would interact with the rest of the compiler. For example, would the routine that processes macro definitions be called by the scanner, the parser, or the code generator? Which of these phases of the compiler would interact with the routines that recognize and expand macro invocation statements?

## Section 5.2

1. Rewrite the code-generation routines given in Figs. 5.18 and 5.19 to produce quadruples instead of object code.
2. Write a set of routines to generate object code from the quadruples produced by your routines in Exercise 1. (Hint: You will need a routine that is similar in function to the GETA procedure in Fig. 5.19.)
3. Use the routines you wrote in Exercise 1 to produce quadruples for the following program fragment:

```
READ(X,Y);
Z := 3 * X - 5 * Y + X * Y;
```

4. Use the routines you wrote in Exercise 2 to produce object code from the quadruples generated in Exercise 3.
5. Rewrite the code-generation routines given in Fig. 5.20 to produce quadruples instead of object code.
6. Use the routines you wrote in Exercises 1 and 5 to produce quadruples for the program in Fig. 5.1.
7. Divide the quadruples you produced in Exercise 6 into basic blocks and draw a flow graph for the program.
8. Assume that you are generating SIC/XE object code from the quadruples produced in Exercise 6. Show one way of performing register assignments to optimize the object code, using registers S and T to hold variable values and intermediate results.

## Section 5.3

1. Write an algorithm for the prologue of a procedure, assuming the activation record format shown in Fig. 5.30.
2. Write an algorithm for the epilogue of a procedure, assuming the activation record format shown in Fig. 5.30.
3. Suggest a way of using the activation record stack to perform dynamic storage allocation for controlled variables. What would be the advantages and disadvantages of such a technique as compared to using a separate area of free storage to perform these allocations?

4. Assume the array C is declared as

```
C: ARRAY[5..20] OF INTEGER
```

   Generate quadruples for the statement

```
C[I] := 0
```

5. Assume the array D is declared as

```
D: ARRAY [-10..10,2..12] OF INTEGER
```

   and is stored in *row-major* order. Generate quadruples for the statement

```
D[I,J] := 0
```

6. Assume the array D declared in Exercise 5 is stored in *column-major* order. Generate quadruples for the statement

```
D[I,J] := 0
```

7. Generalize the methods given in Section 5.3.1 to the storage of three-dimensional arrays in row-major order. Assuming the array declaration

```
E : ARRAY[1..5, 1..10, 0..8] OF INTEGER
```

   generate quadruples for the statement

```
E[I,J,K] := 0
```

8. How could the base address for the array A defined in Fig. 5.26(a) be modified to avoid the need for subtracting 1 from the subscript value (quadruple 1)?
9. How could the technique derived in Exercise 8 be extended to two-dimensional arrays?

10. Assume the array declaration

```
T : ARRAY[1..5, 1..100] of INTEGER
```

Translate the following statements into quadruples and perform elimination of common subexpressions on the result.

```
K := J-1;
FOR I := 1 TO 5 DO
   BEGIN
      T[I,J] := K * K;
      J := J + K;
      T [I,J] := K * K - 1;
   END
```

11. Modify the quadruples produced in Exercise 10 to remove loop invariants.

12. Write an algorithm to construct the proper display when a procedure is invoked. Your algorithm may use the old display (i.e., the current display before the call), the address of the activation record created for the procedure being called, and the block-nesting level of the procedure being called.