3

Lexical and Syntactic Analysis

Exercises

3.1 a.

$$\begin{aligned} Identifier &\rightarrow letter\ IdentRest \\ IdentRest &\rightarrow \mid\ letter\ IdentRest\ \mid\ digit\ IdentRest \end{aligned}$$

where *letter* and *digit* are treated as terminals. Note that individual letters and digits could be substituted into the above productions.

b.

$$\begin{aligned} Float &\rightarrow digit\ IntRest \\ IntRest &\rightarrow .\ digit\ DecRest\ \mid\ digit\ IntRest \\ DecRest &\rightarrow \mid\ digit\ DecRest \end{aligned}$$

where *digit* is treated as a terminal symbol. Note that the digits themselves could be substituted into the above production.

3.2 a.

$$Identifier \rightarrow letter\ \mid\ Identifier\ letter\ \mid Identifier\ digit$$

where *letter* and *digit* are treated as terminals. Note that individual letters and digits could be substituted into the above productions.

b.

$$Float \rightarrow IntPart\ .\ digit \mid Float\ digit$$
$$IntPart \mid \rightarrow digit \mid IntPart\ digit$$

where *digit* is treated as a terminal symbol. Note that the digits themselves could be substituted into the above productions.

3.3 As a right regular grammar the answer is:

$$Identifier \rightarrow digit\ Identifier \mid letter\ Rest$$
$$Rest \rightarrow \mid letter\ Rest \mid digit\ Rest$$

3.4 * The language $\{a^n b^n\}$ cannot be defined using a regular expression, since the generation of n b's requires "remembering" the value of n from the time the a's were generated. The regular expression a^*b^* generates the language $\{a^m b^n\}$ in which m may not equal n. A simple non-regular BNF grammar for generating $\{a^n b^n\}$ is:

$$S \rightarrow aSb \mid ab$$

3.5 * Assume that such a DFSA can be constructed and has K states. Consider the strings: $a^1, a^2, \ldots, a^K, a^{K=1}$. For at least two of the strings a^i and a^j for $i \neq j$ must wind up in the same state. Therefore they are indistinguishable, so that $a^i b^i$ and $a^j b^i$ must wind up being accepted. This is a contradiction.

3.6 Fortran and Cobol were developed on machines with a 6 bit character, so there were at most 64 distinct characters. With both upper and lower case letters and digits, you have 26 + 26 + 10 characters (= 62). There would not be enough room to add the period, plus, minus (or hyphen), comma, etc. So lower case was omitted.

When 8 bit characters were adopted, it seemed simpler to make these older languages case insensitive in order to remain backward compatible.

3.7 In PHP variable names are case sensitive, like all C-like languages. However, reserved words and function names are case insensitive. Note that this is an irregularity in the language and should generally be avoided.

3.8 This is an enormous task. You must include a test for every reserved word and every terminal symbol.

For even the simplest programs, you must have tests where whitespace occurs between every two consecutive tokens and a test where all unnecessary whitespace is eliminated. E.g., the parser used in the first edition of the text would not allow 2 or more comments between consecutive tokens.

For identifiers, the test must include 1- and 2-character cases, as well 3 or more character cases. The same applies to integers.

3.9 a.

$$\begin{aligned}(S, a\$) &\vdash (I, \$)\\ &\vdash (F,)\end{aligned}$$

b.

$$\begin{aligned}(S, a2\$) &\vdash (I, 2\$)\\ &\vdash (I, \$)\\ &\vdash (F,)\end{aligned}$$

c.

$$\begin{aligned}(S, a2i\$) &\vdash (I, 2i\$)\\ &\vdash (I, i\$)\\ &\vdash (I, \$)\\ &\vdash (F,)\end{aligned}$$

d.

$$\begin{aligned}(S, abc\$) &\vdash (I, bc\$)\\ &\vdash (I, c\$)\\ &\vdash (I, \$)\\ &\vdash (F,)\end{aligned}$$

3.10

$$\begin{aligned}Float &\rightarrow digit\ Float \mid .\ digit\ IntRest \mid Exponent\\ IntRest &\rightarrow digit\ IntRest \mid Exponent\\ Exponent &\rightarrow E\ +\ Expo \mid E\ -\ Expo \mid E\ Expo\\ Expo &\rightarrow digit\ Expo \mid digit\end{aligned}$$

A DFSA can be constructed directly from the above right regular grammar, as can a regular expression. Legal examples include: 2E3, 3.14E12, 3.14E+12, 3.14E-12. Illegal examples include: 2.E3, .14E12, +3.14E12, 3.14E12-.

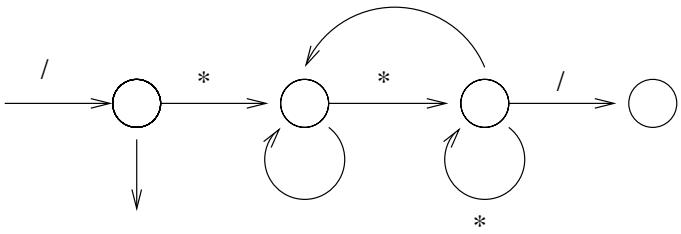
3.11
```
public Token next( ) { // Return next token
        do {
            if (isLetter(ch)) { // ident or keyword
                ...
            } else if (isDigit(ch)) { // int or float literal
                String number = concat(digits);
                if (ch != '.' && ch != 'E' && ch != 'e')  // int Literal
                    return Token.mkIntLiteral(number);
                if (ch == '.')
                    number += concat(digits);
                if (ch == 'E' || ch == 'e') {
                    number += concat("Ee");
                    number += concat(digits);
                }
                return Token.mkFloatLiteral(number);
            }
```

3.12 This notation for based numbers comes from Ada. As a regular grammar:

$$\begin{aligned} BasedNo &\rightarrow Base\ \#\ BasedInteger\ \# \\ Base &\rightarrow Integer \\ BasedInteger &\rightarrow BasedInteger\ BasedDigit \mid BasedDigit \\ BasedDigit &\rightarrow Digit \mid a \mid b \mid c \mid d \mid e \end{aligned}$$

The DFSA, right regular grammar, and regular expression can easily be constructed from the above grammar. Note that without a lot of work the grammar cannot enforce the restriction that each digit be legal in the given base. Consider the number: 8#19#; the digit 9 is illegal in base 8.

3.13 Here is the DFSA:



One of the authors once used a computer mnaufacturer's Pascal compiler, which used a `(* comment *)` convention, in which a comment of the form:

```
(***********)
```

was recognized properly only if the number of stars between the opening and closing comment markers was an even number. They probably never drew the DFSA, but rather wrote the code directly without a DFSA design.

3.14
```
public Token next( ) {
   ...
   switch (ch) {
   ...
      case '/':
          ch = nextChar();
          if (ch != '*' && ch != '/')  return Token.divideToken;
          if (ch == '*') {
             getComment( );
             continue;
          }
          ...
          break;
   ...
} // next( )

private void getComment( ) {
   repeat
          ch = nextChar( );
          // flush the * of the "( *" or the non )
      while (ch <> '*') do
          ch = nextChar( );
      ch = nextChar( );        // flush the *
      while (ch == '*') do
          ch = nextChar( );
   until ( ch == ')' );
   ch = nextChar( );        // flush the ) of the "* )"
}
```

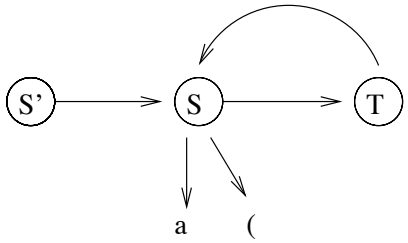
3.15 Note that the solution to this problem requires more than a single character lookahead. One early Pascal compiler, after advancing to the second period, literally replaced it by a colon, thus also accepting the illegal subrange 10:81.

Several solutions are possible: (a) implementing an unlimited forward lookahead feature; (b) implementing an unlimited push back feature (in this case, you never need to back up over a line boundary); (c) adding a one character peek function to a lexer that stays one character ahead.

Since our lexer is based on the one ahead implementation model, the simplest solution is to add a one character peek function (see Figure 3.5), which is given below.

```
} else if (isDigit(ch)) {
   String number = concat(digits);
   if (ch != '.') return Token.mkIntLiteral(number);
   if (peek( ) == '.') return Token.mkIntLiteral(number);
   ...
```

3.16 a. Here is the left dependency graph:



The grammar must be in augmented form, which means that we must add a new start symbol and production:

$$S' \rightarrow S \ \$$$

b. nullable = {S T}.
First(S) = {a (}
First(T) = {a , (}

3.17 See the program `Calculate.java` in the accompanying *ch03code* directory, along with the related files `Lexer.java` and `Token.java` in the *Clite* directory in the software distribution for Chapters 2-8.

In this solution the code which builds an abstract syntax tree was replaced by code which directly interprets the token stream. Contrast this approach with the one use in Exercise 3.21 in which an abstract syntax tree is built and then interpreted.

3.18
```
private int match (TokenType[] t) {
    for (int i=0; i<t.length; i++)
        if (token.type().equals(t[i])) {
            return i;
    }
    return -1;
}
```

3.19 PL/I was such a language; there were no reserved words. It allowed such constructs as:

```
if if = then then then = else; else do do = end; end
```

where `do/end` were similar to C's use of open/close brace.

3.20 When assigning this problem, the instructor should supply students with an abridged version of the file `AbstractSyntax.java` by removing from it the class `Indenter` and all calls to the method `display`. This file is part of the *Clite* interpreter's Java sources, which are available for downloading only at the instructor's web site.

3.21 See the program `PolishPrefixExpr.java` for the Parser and `Expression.java` for the abstract syntax, in the accompanying *ch03code* directory.