7 Constructing a Parser for a Python Subset

```
1 print(-59 + 20*3)
2 a = 2
3 bb_1 = -(a) + 12
4 print(a*bb_1 + a*3*(-1 + -1 + -1))
```

Figure 7.1

```
<program>           → <stmt>* EOF
<stmt>              → <simplestmt> NEWLINE
<simplestmt>        → <assignmentstmt>
<simplestmt>        → <printstmt>
<assignmentstmt>    → NAME '=' <expr>
<printstmt>         → 'print' '(' <expr> ')'
<expr>              → <term> ('+' <term>)*
<term>              → <factor> ('*' <factor>)*
<factor>            → '+' <factor>
<factor>            → '-' <factor>
<factor>            → UNSIGNEDINT
<factor>            → NAME
<factor>            → '(' <expr> ')'
```

Figure 7.3

Switches

```
1 debug = False    # controls debugging display
2 heading = False  # controls heading display (only in "sh" shells)
3 pyprog = False   # controls python program display (only in "sh" shells)
```

```
1 # advances to the next token in the list tokens
2 def advance():
3     global token, tokenindex
4     tokenindex += 1
5     if tokenindex >= len(tokenlist):
6         raise RuntimeError('Unexpected end of file')
7     token = tokenlist[tokenindex]
```

It is important to remember that any time during a parse, the variable token contains the current token, and tokenindex is the index of that token in tokenlist, the list of tokens created by the tokenzier. Also remember that token is an object that has the variables `line`, `column`, `category`, and `lexeme`.

- `token.line` is the line number of the line in the source program in which the current token appears.
- `token.column` is the column number of the column in the source program in which the current token starts.
- `token.category` is the category of the current token.
- `token.lexeme` is the lexeme of the current token.

```
1 # advances if current token is the expected token
2 def consume(expectedcat):
3     if (token.category == expectedcat):
4         advance()
5     else:
6         raise RuntimeError('Expecting ' + catnames[expectedcat])
```

```
1 # <program> → <stmt>* EOF
2 def program():
3     while token.category in [NAME, PRINT]:
4         stmt()
5     if token.category != EOF:
6         raise RuntimeError('Expecting end of file')
```

```
<program>                → <stmt>* EOF
```

```
1 def stmt():
2     simplestmt()
3     consume(NEWLINE)
```

```
    ⟨stmt⟩                → ⟨simplestmt⟩ NEWLINE
```

```
1 def simplestmt():
2    if token.category == NAME:
3       assignmentstmt()
4    elif token.category == PRINT:
5       printstmt()
6    else:
7       raise RuntimeError('Expecting stmt')
```

```
<simplestmt>          → <assignmentstmt>
<simplestmt>          → <printstmt>
```

```
1 def printstmt():
2       advance()          # advance past PRINT token
3       consume(LEFTPAREN)
4       expr()
5       consume(RIGHTPAREN)
```

```
   <printstmt>          → 'print' '(' <expr> ')'
```

```
1 def assignmentstmt():
2       advance()          # advance past PRINT token
3       consume(ASSIGNOP)
4       expr()
```

```
   <assignmentstmt>      → NAME '=' <expr>
```

```
1 def expr():
2     term()
3     while token.category == PLUS:
4         advance()    # no need for consume() here
5         term()
```

```
    <expr>                  → <term> ('+' <term>)*
```

```
1 def term():
2     factor()
3     while token.category == TIMES:
4         advance()    # no need for consume() here
5         factor()
```

```
    <term>                  → <factor> ('*' <factor>)*
```

```
 1 def factor():
 2    if token.category == PLUS:
 3       advance()
 4       factor()
 5    elif token.category == MINUS:
 6       advance()
 7       factor()
 8    elif token.category == UNSIGNEDINT:
 9       advance()
10    elif token.category == NAME:
11       advance()
12    elif token.category == LEFTPAREN:
13       advance()
14       expr()
15       consume(RIGHTPAREN)
16    else:
17       raise RuntimeError('Expecting factor')
```

```
<factor>              → '+' <factor>
<factor>              → '-' <factor>
<factor>              → UNSIGNEDINT
<factor>              → NAME
<factor>              → '(' <expr> ')'
```

```
 1 def main():
 2    global source
 3
 4    if len(sys.argv) == 2:   # check if correct number of cmd line args
 5       try:
 6          infile = open(sys.argv[1], 'r')
 7          source = infile.read()  # read source program
 8       except IOError:
 9          print('Cannot read input file ' + sys.argv[1])
10          sys.exit(1)
11    else:
12       print('Wrong number of command line arguments')
13       print('Format: python p1.py <infile>')
14       sys.exit(1)
15
16    if source[-1] != '\n':  # add newline to end if missing
17       source = source + '\n'
```

```
1     try:
2        tokenizer()    # tokenize source code in source
3        parser()
4
5     # on an error, display an error message
6     # token is the token object on which the error was detected
7     except RuntimeError as emsg:
8       # output slash n in place of newline
9       lexeme = token.lexeme.replace('\n', '\\n')
10      print('\nError on '+ "'" + lexeme + "'" + ' line ' +
11         str(token.line) + ' column ' + str(token.column))
12      print(emsg)      # message from RuntimeError object
13      sys.exit(1)
```

Figure 7.4