COP 4020 Programming Assignment 4: Calculator Parser

Educational Objectives: After completing this assignment, the student should be able to do the following:

- Draw parse trees for legal expressions
- Given an LL(1) grammar, implement a recursive descent parser as a Java class:
    i. Define functions (methods) for each non-terminal in the grammar
    ii. Use sequencing and recursion as defined in the productions of the grammar
- Explain how legal expressions are parsed by the code
- Explain why non-legal expressions are not fully parsed and where the parser detects the non-legality

Operational Objectives: Implement a recursive descent parser in Java for a calculator language based on a BNF grammar.

Deliverables: One file `Parser4.java`

1. Arithmetic operators in a programming language are typically *left associative* with the notable exception of exponentiation (^) which is *right associative*. (However, this rule of thumb is not universal.)

   Associativity can be captured in a grammar. For a left associative binary operator `lop` we can have a production of the form:

   ```
   <expr> -> <term>
           | <expr> lop <term>
   ```

   For example, `a+b+c` is evaluated from the left to the right by summing `a` and `b` first. Assuming that `<term>` represents identifiers, the parse tree of `a+b+c` with the grammar above is:

   ```
                <expr>
              /   |   \
         <expr>  +  <term>
        /   |   \       |
   <expr>  +  <term>  c
     |           |
   <term>       b
     |
     a
   ```

   As you can see, the left subtree represents `a+b` which is a subsexpression of `a+b+c`, because `a+b+c` is parsed as `(a+b)+c`.

   Note that the production for a left associative operator is left recursive. To eliminate left recursion, we can rewrite the grammar into:

   ```
   <expr>      -> <term> <term_tail>
   <term_tail> -> lop <term> <term_tail>
                | empty
   ```

   This (part of the) grammar is LL(1) and therefore suitable for recursive descent parsing. However, the parse tree structure does not capture the left-associativity of the `lop` operator.

Draw the parse tree of `a+b+c` using the LL(1) grammar shown above. You may assume that `<term>` represents identifiers. Hint: draw the tree from the top down by simulating a top-down predictive parser.

2. For a right associative operator `rop` we can create a grammar production of the form:

```
<expr> -> <term>
        | <term> rop <expr>
```

An example right associative operator is exponentiation `^`, so `a^b^c` is evaluated from the right to the left such that `b^c` is evaluated first.

Draw the parse tree of `a^b^c`. You may assume that `<term>` represents identifiers.

3. The *precedence* of an operator indicates the priority of applying the operator relative to other operators. For example, multiplication has a higher precedence than addition, so `a+b*c` is evaluated by multiplying `b` and `c` first. In other words, multiplication groups more tightly compared to addition. The rules of operator precedence vary from one programming language to another.

The relative precedences between operators can be captured in a grammar as follows. A nonterminal is introduced for every group of operators with identical precedence. The nonterminal of the group of operators with lowest precedence is the nonterminal for the expression as a whole. Productions for (left associative) binary operators with lowest to highest precedences are written of the form suitable for recursive descent parsing. Here is an outline:

```
<expr>    -> <e1> <e1_tail>
<e1>      -> <e2> <e2_tail>
<e1_tail> -> <lowest_op> <e1> <e1_tail>
           | empty
<e2>      -> <e3> <e3_tail>
<e2_tail> -> <second_lowest_op> <e2> <e2_tail>
           | empty
...
<eN>      -> '(' <expr> ')'
           | '-' <eN>
           | identifier
           | number
<eN_tail> -> <highest_op> <eN> <eN_tail>
           | empty
```

where `<lowest_op>` is a nonterminal denoting all operators with the same lowest precedence, etc.

The following Java program uses these concepts to implement a recursive descent parser for a calculator language:

```
/* Parser.java
   Implementes a parser for a calculator language
   Uses java.io.StreamTokenizer and recursive descent parsing

   Compile:
   javac Parser.java
*/
import java.io.*;
/* Calculator language grammar:

   <expr>        -> <term> <term_tail>
   <term>        -> <factor> <factor_tail>
```

```
   <term_tail>   -> <add_op> <term> <term_tail>
                  | empty
   <factor>      -> '(' <expr> ')'
                  | '-' <factor>
                  | identifier
                  | number
   <factor_tail> -> <mult_op> <factor> <factor_tail>
                  | empty
   <add_op>      -> '+' | '-'
   <mult_op>     -> '*' | '/'
*/
public class Parser
{
  private static StreamTokenizer tokens;
  private static int token;
  public static void main(String argv[]) throws IOException
  {
    InputStreamReader reader;
    if (argv.length > 0)
      reader = new InputStreamReader(new FileInputStream(argv[0]));
    else
      reader = new InputStreamReader(System.in);
    // create the tokenizer:
    tokens = new StreamTokenizer(reader);
    tokens.ordinaryChar('.');
    tokens.ordinaryChar('-');
    tokens.ordinaryChar('/');
    // advance to the first token on the input:
    getToken();
    // check if expression:
    expr();
    // check if expression ends with ';'
    if (token == (int)';')
      System.out.println("Syntax ok");
    else
      System.out.println("Syntax error");
  }
  // getToken - advance to the next token on the input
  private static void getToken() throws IOException
  {
    token = tokens.nextToken();
  }
  // expr - parse <expr> -> <term> <term_tail>
  private static void expr() throws IOException
  {
    term();
    term_tail();
  }
  // term - parse <term> -> <factor> <factor_tail>
  private static void term() throws IOException
  {
    factor();
    factor_tail();
  }
  // term_tail - parse <term_tail> -> <add_op> <term> <term_tail> |
empty
  private static void term_tail() throws IOException
  {
    if (token == (int)'+' || token == (int)'-')
    {
      add_op();
      term();
      term_tail();
```

```
      }
    }
    // factor - parse <factor> -> '(' <expr> ')' | '-' <expr> | identifier
  | number
    private static void factor() throws IOException
    {
      if (token == (int)'(')
      {
        getToken();
        expr();
        if (token == (int)')')
          getToken();
        else System.out.println("closing ')' expected");
      }
      else if (token == (int)'-')
      {
        getToken();
        factor();
      }
      else if (token == tokens.TT_WORD)
        getToken();
      else if (token == tokens.TT_NUMBER)
        getToken();
      else System.out.println("factor expected");
    }
    // factor_tail - parse <factor_tail> -> <mult_op> <factor>
  <factor_tail> | empty
    private static void factor_tail() throws IOException
    {
      if (token == (int)'*' || token == (int)'/')
      {
        mult_op();
        factor();
        factor_tail();
      }
    }
    // add_op - parse <add_op> -> '+' | '-'
    private static void add_op() throws IOException
    {
      if (token == (int)'+' || token == (int)'-')
        getToken();
    }
    // mult_op - parse <mult_op> -> '*' | '/'
    private static void mult_op() throws IOException
    {
      if (token == (int)'*' || token == (int)'/')
        getToken();
    }
  }
```

Copy (and download if needed) this example parser program from:

```
~cop4020p/fall08/examples/
```

Compile and execute:

```
javac Parser.java
java Parser
```

Give the output of the program when you type `2*(1+3)/x;` and explain why this expression is accepted by the parser by drawing the parse tree. Give the output of the program when you type

`2x+1;` and explain why it is not accepted. At what point in the program does the parser fail?

4. Extend the parser program to include syntax checking of function calls with one argument, given by the new production for `<factor>`:

```
<factor> -> '(' <expr> ')'
          | '-' <factor>
          | identifier '(' <expr> ')'
          | identifier
          | number
```

   Test your implementation with `2*f(1+a);`. Also draw the parse tree of `2*f(1+a);`.

5. Extend the parser to include syntax checking of the exponentiation operator `^`, so that expressions like `-a^2` and `-(a^b)^(c*d)^(e+f)` can be parsed. Note that exponentation is right associative and has the highest precedence, even higher than unary minus, so `-a^2` is evaluated by evaluating `a^2` first. To implemented this, you must add a `<power>` nonterminal and also change the production of `<factor>` so that the parse tree of `-a^2` is:

```
<factor>
 /    \
-   <power>
    /  |  \
   a   ^   <power>
             |
             2
```

Turn in this assignment as a working Java program named `"Parser4.java"`, using the submit script `"pr4submit.sh"`. Your answers for the non-programming questions should be inserted in the documentation at the top of your file. Use plain text to draw the trees as required.