CSC461 Programming Languages - Fall 2017
Programming Assignment #3: Recursive Descent Parser (Java)

Recursive descent parsing is often used in compilers to determine whether the input source code is syntactically correct. Recursive descent parsing consists of writing a subprogram (possibly recursive) for each nonterminal element of the language. An input token is parsed by calling the appropriate subprogram, which decides if it is a valid language element.

Write a recursive descent parser in Java for the expression grammar given below. Your program should read each line of input and print a message to standard output, telling the user whether or not it is a valid expression. Continue processing input until a blank line is entered.

To make it easier to perform lexical scanning for valid tokens, each expression will appear on a single line of input. Spaces are permitted (but not required) between tokens. A Java StringTokenizer may be used to tokenize the input stream.

To implement the parser, write a method corresponding to each nonterminal element in the EBNF grammar. Each method should return *true* if it accepts a syntactic element, otherwise it should return *false*. Ensure that all input characters are "consumed" before declaring a string to be a valid expression. For example, the string "x + (y * z / 4)" is syntactically valid, but "x + (y * z / 4))" is not, because it has an extra right parenthesis.

EBNF expression grammar:

```
<expr>     -> <term> { <addop> <term> }
<term>     -> <factor> { <mulop> <factor> }
<factor>   -> <integer> | <float> | <id> | '(' <expr> ')' | [-] <factor>
<integer>  -> <digit> { <digit> }
<float>    -> <integer> . <integer>
<id>       -> <letter> { <letter> | <digit> }
<letter>   -> A | B | C | D | E | F | G | H | I | J | K | L | M |
              N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
              a | b | c | d | e | f | g | h | i | j | k | l | m |
              n | o | p | q | r | s | t | u | v | w | x | y | z | _
<digit>    -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<addop>    -> + | -
<mulop>    -> * | / | %
```

If you prefer writing recursive routines, you may rewrite this as a BNF grammar:

```
<expr> -> <expr> <addop> <term> | <term>
<term> -> <term> <mulop> <factor> | <factor>
         etc.
```

Name your main Java class *Parser*, and your source file *Parser.java*. Prompt the user to enter expressions, one per line, ending execution with a blank line.

Usage: `java Parser [-t]`

The –t switch prints out a list of tokens in addition to the validity message.

Program submission notes

- Complete your program by the due date (Monday November 6) in order to receive credit for this assignment. Late programs will not be accepted for partial credit unless prior arrangements have been made with the instructor.

- To receive full credit, your code must be readable, modular, nicely formatted, and adequately documented (use *javadoc*!), as well as complete and correct. It must build and run successfully using a reasonably current Java SE 8 distribution on both Windows and Linux. If your program does not run correctly, indicate why. This will make it easier to give you partial credit.
- You must write a recursive descent parser, implementing a method for each nonterminal production rule. No other approach is acceptable for this assignment.
- Work in teams of two students on this assignment. Your program will be pulled from your GitLab repository on the due date at midnight. Name the project *CSC461_F17_PA3*. Be sure to add your instructor as a developer on the project. This will enable me to monitor progress and pull projects for grading.

  Partners for PA#3
  Ian Beamer + Nathan Ducasse
  Christopher Blumer + Sierra Wahlin-Rhoades
  Allison Bodvig + Noah Brubaker
  Joey Brown + Collin Chick
  Kathleen Brown + Isaac Rath
  Aaron Campbell + Jake Davidson
  Lucas Carpenter + Michael Pfeifer
  Darla Drenckhahn + Ryan McCaskell
  Benjamin Garcia + Katherine MacMillan
  Aaron Gibbs + Mathias Wingert
  Jeremy Goens + Kyle Lorenz
  Naomi Green + Andrew Housh
  Chad Heath + Garret Odegaard
  Ryan Hinrichs + Soham Naik
  Lawrence Hoffman + Logan Lembke
  Tanner Holthus + Matthew Schallenkamp
  Cameron Javaheri + Ryley Sutton
  Kyle MacMillan + Hannah Wegehaupt

When you finish the assignment, email me a brief description of team member contributions (both you and your partner) using the form *Teamwork Evaluation.docx*. Your comments will be kept private, but I may choose to grade team members individually.

Example

```
% java Parser
Enter expression: A + B
"A + B" is a valid expression

Enter expression: ((A+B) *   C    )
"((A + B) * C)" is a valid expression

Enter expression: ((A + B) * C))
"((A + B) * C))" is not a valid expression

Enter expression: A + * B
"A + * B" is not a valid expression

Enter expression: hi * -3.14159
"hi * -3.14159" is a valid expression

Enter expression:
(end of input)
```

```
% java Parser -t
Enter expression: A + B
"A + B" is a valid expression.
tokens: A,+,B

Enter expression: ((A+B) *   C    )
"((A+B) * C )" is a valid expression.
tokens: (,(,A,+,B,),*,C,

Enter expression: ((A + B) * C))
"((A + B) * C))" is not a valid expression.
tokens: (,(,A,+,B,),*,C,),)

Enter expression: A + * B
"A + * B" is not a valid expression.
tokens: A,+,*

Enter expression: hi * -3.14159
 "hi * -3.14159" is a valid expression.
tokens: hi,*,-,3,.,14159

Enter expression:
(end of input)
```