A#3CSC401F14 Due on or before November 6, 2013 ks rajasethupathy

In this project you will be making a few additions/deletions and improvements to your asignment#2.

Add semantic actions for **arithmetic** expressions and assignment statements. To implement the semantic tasks, you need to do the following:

* 1. Define an integer stack called semanticActionStack (SAS). You are allowed to use any existing code for stack class that you have access to.
  2. **Define a list ( using Java’s HashMap, which implements the map interface ) of elements, where each element is a <key, value> pair.** The first data field contains the name of the variable used in your program and the second contains its integer value. At the start of the program you may choose arbitrary integer value to initialize each variable. You can also assume that every arithmetic action results in integer only.
  3. Whenever the parser detects an <ident>, it should get the value of the <ident> from the map, and push the value on the SAS.
  4. **Whenever the parser encounters an operator, it just remembers it; no other action is needed.**
  5. **Once appropriate number of operands have been recognized (two if the operator is binary, or one if the operator is unary, etc) it should pop appropriate number of values** from the SAS, perform the operation, and push the result back on the SAS. See the example given at the end of this document.
  6. When the input expression is validated,

1. Printout the top of the stack, and
2. If appropriate, assign the value on the top of the stack to the variable on the left of the assignment statement and also output that value.

Notes:

1. Do not change the grammar on your own. Talk to me if you think there are compelling reasons to do so.
2. Prior to the code segment of each parsing method, provide the grammar rule(s) that it implements (as documentation).
3. Make sure you read one line at a time from the input file. You should neither read one token at a time nor read and store all tokens in one go.
4. Make sure you implement user-defined exceptions to handle error situations. If you don’t know how to create “user-defined exceptions” search the Internet for examples. To demonstrate the correctness of your implementation of exception handling provide the test results of any five error cases.
5. Make sure that getNextToken( ) provides an enumerated type to the parsing methods. The enumerated type that getNextToken( ) provides should be used in subsequent parsing actions.
6. Where appropriate, replace “while…do” loops by “do…while” loops.
7. All methods in the parser, excepting the constructor and the main methods, must be private.
8. Implementing semantics for an iterative construct or a selection construct or relational expressions will each fetch an additional 5 points.
9. If you implement JUnit testing that will fetch additional 5 points.
10. As usual, you need to submit:
11. Hard copies of the source code, test cases and test results, and
12. A flash drive containing all the above along with the executable version of the source code.

Consider the rule <factor> 🡪 a | b

**Parsing function alone**:

boolean factor( )

{

if ( nT == aToken ) { getNextToken( ); return true; }

if ( nT == bToken ) { getNextToken( ); return true; }

return false;

}

# Parsing with semantic action

boolean factor( )

{

if ( nT == aToken) { push (getValue(aToken)) ; getNextToken( ); return true; }

if ( nT == bToken) { push (getValue(bToken)) ; getNextToken( ); return true; }

return false;

}

Consider

<expression> 🡪 <term> <nextT>

<nextT> 🡪 + <term> <nextT>

Parsing action alone for <nextT> is:

boolean nextT {

if (nT == plusToken) {

getNextToken( ); //call to lexical analyzer

if (<term> )

if (<nextT>)

return true;

}

return false;

}

Parsing function with semantic action for <nextT> is:

boolean <nextT> {

if (nT == plusToken) {

getNextToken( ); //call to lexical analyzer

if (<term> ) {

//sas is the semantic action stack

//appropriate type casting is needed when dealing with stack class.

int op1 = sas.pop( );

int op2 = sas.pop( );

sas.push(op1+op2);

if (<nextT>) return true;

}

}

return false;

}