Expression Compiler Project

Introduction

One of the most common tasks in computer science is processing instructions written as mathematical expressions. For example, compilers, scripting languages, and applications that allow the user to define calculations for information to be processed.

In mathematics we generally use infix notation: *leftOperand operator rightOperand*.

As described in the textbook we can use ***stacks*** to assist us in converting infix notation into a form which is more easily processed by a computer program. We generally create ***expression trees*** for this purpose. We also have the need to keep track of data by assigning values to named *variables*. To allow us to store values for the variables and then easily retrieve the values when given the name of the variable we can use ***dictionaries*** or ***maps***. As you can see, this process uses quite a few of the Abstract Data Types that we have been discussing in this class.

For this assignment you will write a *simple* compiler (of sorts). It will accept arithmetic expressions and execute them, one by one, to produce a final result. The expressions can contain assignment statements so that results can be saved in named variables. Your program will prompt the user for an expression (remember that in languages like C, C++, C#, and Java assignment is an expression and not just a statement). The expression will be evaluated (and if it is an assignment the value of the variable as the left operand will be changed). Variables can be accessed (referenced) in the expressions (or on the right side of an assignment expression) once the variable name has been defined (variables need not be declared as all variables will be assumed to contain integer values). The value of the expression, once evaluated, will be displayed to the user and another prompt will be presented. Numeric literals and variable names may be used in the expressions. Numeric literals and variable names, alone, may also be used as expressions and their values will be displayed. If, in response to the prompt, the user just enters a return (or new line) character the program will terminate. Likewise, if the input source (the keyboard by default or, optionally, a redirected data file) reaches the end of file (on the keyboard this is indicated by a <*ctrl-Z*> in Windows or a <*ctrl-D*> in Unix/Linux/MacOS) then the program will terminate.

I will provide you with a sample data file that you may use to verify the correctness of your program before submitting it for grading.

In essence, you will be writing a simple, integer only, desk calculator. This may be modified to a more advanced calculator by replacing the integer declarations with floating point declarations.

For the sake of simplicity you will not need to deal with unary operators in your expressions. These can easily be implemented using binary operators such that "***-a***" can be rewritten as "***0 – a***".

Parentheses are permitted so as to modify the order of operations (without parentheses the usual *PEMDAS* rules apply). Exponentiation will be implemented using the *caret* "^" operator (note: Java does NOT use the *caret* operator for *raise to the power* but instead uses it for *exclusive or*). Since Java does not have a *raise to the power* operator and the provided library function *Math.pow()* processes *double* argument to produce a *double* value (the algorithm uses *Math.log()* and *Math.exp()* functions) you will need to write your own function *int ipow(base, exp)* (I will provide the code for this – see below).

Input Line Processing

The input will consist of a series of text lines containing expressions written to the specified standards. Each line will represent an expression to be evaluated. The lines will contain ***tokens*** (numeric literals, variable names, operators, punctuation characters – in this case parentheses used for grouping). Your program will read in the line and split the line with white space as the delimiter between tokens. However, the user need not include this white space around tokens so your program will need to process the input line to replace each operator with the character sequence: *space operator space*. This can easily be accomplished by using statements such as:

```
String orig, line;
orig = inp.nextLine();
line = orig.replaceAll("([-+*/%^()=])", " $1 ");
```

The first argument to *replaceAll()* is a regular expression (*regex*) to define a pattern to search for. The square brackets (`[`, `]`) surround a set of characters that will be searched for (note that this set can include ranges with the dash character (`-`) used to indicate a range, e.g., `a-z`, therefore, to stop the pattern compiler from considering the dash as implying a range we *must* put the dash first or last in the set, as done here). Any member of the set will be acceptable to satisfy a match. The parentheses around the set in the *regex* tell the pattern compiler to save whichever character it was that matched the pattern into a temporary storage cell. The replacement string (second argument) contains the code `$1` which refers back to the saved matching character stored in the temporary storage cell so that the character is replaced with *space operator space*. (See any programming book, including the Javadoc, for a discussion of regular expressions and pattern matching.)

To split the now modified line into tokens we can use the following Java statement:

```
String[] tokens = line.split("\\s+");
```

The *regex* pattern "\s" matches any single white space character (*space*, *backspace*, *horizontal tab*, *vertical tab*, *form feed*, *new line*, *return*). But, to get a backslash character into a string we need to quote the quote character (backslash) so we enter a double backslash. Also, we need to consider the case where we may have multiple white space characters (remember we added white space around the operators, but the user might have used white space already, in which case we get multiple white space characters between the tokens). The plus (`+`) symbol modifies the pattern from *match a single white space character* to *match one or more white space characters*. So, the *regex* pattern "`\\s+`" will now match any run of one or more whitespace characters (which may be mixed, e.g., *space tab space space*).

Class Hierarchy for Parsed Tokens

As discussed above, we will be splitting the line into *tokens*. These tokens need to be categorized. To facilitate the categorization we will create a class hierarchy of tokens. We start off with the parent abstract class:

```
public abstract class Token
{
     public abstract int eval();
}
```

The method *eval()* will return the *value* of the token.

The following concrete classes will be developed:

`LitToken`	Represents an integer numeric literal. *eval()* returns the value of that literal.
`VarToken`	Represent a variable. *HashMap*.
`OprToken`	Represents an operator character. e*val()* returns the integer value of the character operator.
`ExprToken`	Represents a binary operator expression. It contains three (3) fields: • `Token left;` • `char  opr;` • `Token right;` Where the left and right tokens may be Lit, Var, or Expr tokens (the last allows for recursive descent expressions). e*val()* returns the expression value after all subexpressions have been evaluated.

All of the token classes will override the *toString()* method. This method will create the string that contains the type of token (e.g., "`LIT:`" followed by the result of calling *eval()* on the object instance).

In the case of a VarToken it will display "`VAR:`*name->eval()*".

In the case of an ExprToken it will display "`EXPR:`*left opr right*".

All of these concrete classes should include a *main()* method to *unit test* the class.

Since the ExprToken represents expression evaluation you will also need to write the method:

```
int apply(int left, char opr, int right)
```

This method will, based on the *opr* value, compute the result of this expression and return it.

The method *iPow()* (reasonably efficient) is presented here:

```
//  Integer power function (not supplied in library).
private int ipow(int base, int exp)
{
    int result = 1;
    while (exp > 0) {
        if ((exp & 1) == 1) result *= base;
        exp >>= 1;
        base *= base;
    }
    return result;
}
```

The VarToken class should also contain the method:

```
void assign(int value)
```

This method will *put()* the argument *value* into the symbol table with the key *this.name*.

Note: that this class will contain the following declaration:

```
public static final Map<String, Integer> SYMTAB = new HashMap<>();
```

Please remember to import the appropriate classes into this class source file.

Handling Operator Priorities

In the main ***Compiler*** class you will need to access a mapping from operator characters to priorities. I recommend using a *Map* for this. Consider the following code:

```
//  Declare and define Operator Priority map.
public static final Map<Character, Integer> OPR_PRIO;

static
{
    OPR_PRIO = new HashMap<>();
    OPR_PRIO.put(')', 5);
    OPR_PRIO.put('^', 4);
    OPR_PRIO.put('*', 3);
    OPR_PRIO.put('/', 3);
    OPR_PRIO.put('%', 3);
    OPR_PRIO.put('+', 2);
    OPR_PRIO.put('-', 2);
    OPR_PRIO.put('=', 1);
    OPR_PRIO.put('(', 0);
}
```

The method that has no name, just the *static* modifier is a static constructor. This constructor is called when the class is loaded into the JVM and the *Class* object is first created. Its purpose is to initialize *static* data. Since there's a fair amount of work being done here you cannot do it with a single assignment statement, thus the *static* constructor. You will need to access the priority of operators to determine when to perform an operation versus delay the operation (i.e., stack the operator). If the first character a token that represents an operator is stored in *first*:

```
char first = token.charAt(0);
```

and *topOpr* contains the character code for the operator at the top of the *operators* stack:

```
char topOpr = '\0';
if ( ! operators.empty())
     topOpr = (char)operators.peek().eval();
```

then, to compare the relative priorities you can use:

```
if (OPR_PRIO.get(first) > OPR_PRIO.get(topOpr)) …
```

Displaying Debugging Information and Reporting Results

I recommend the use of the following code in your *Compiler* class:

```
private static boolean DEBUG = false;

private static void DEBUG(String fmt, Object... args)
{ if (DEBUG) System.err.printf(fmt, args); }

private static void REPORT(String fmt, Object... args)
{ System.out.printf(fmt, args); }
```

Then, debugging code can look something like this:

```
DEBUG(">>>  orig. line: %s%n", orig);
```

This code will only display if the DEBUG flag is set to true, otherwise it will be ignored. The flag can be turned on/off whenever needed either for the whole file or just for sections of code to see selected debugging information.

Using Stacks to Keep Track of Saved Information

I would suggest that you create two *Stack*s in your program, one for operands and one for operators. For example:

```
//  Stacks for processing infix operands and operators.
Stack<Token> operands = new Stack<>();
Stack<OprToken> operators = new Stack<>();
```

The operands stack can contain *Lit/Var/ExprToken* object instances while the operator stack can only contain *OprToken* object instances.

Parenthesis Processing

Open parentheses are always pushed onto the operators stack when they occur. When close parentheses are seen they are not pushed onto the operators stack but instead trigger a backtracking by performing all pending operations until the open parenthesis is found. This stops the backtracking. The open parenthesis should then be popped from the operations stack and the close parenthesis thrown away. The remainder of the input line can now be processed.

As a suggestion to facilitate the processing of input lines, insert an open parenthesis followed by a space in front of the input line and a close parenthesis preceded by a space at the end of the input line. Since you are doing parenthesis processing anyway, this means that you no longer need any special processing to handle the end of line situation.