`$Id: asg3j-bitree-calc.mm,v 1.9 2013-10-17 18:36:17-07 - - $`
`PWD: /afs/cats.ucsc.edu/courses/cmps012b-wm/Assignments/.asg/asg3j-bitree-calc`
`URL: http://www2.ucsc.edu/courses/cmps012b-wm/:/Assignments/.asg/asg3j-bitree-calc/`

1. Overview

A binary tree is a recursive data structure. We can define a tree as either being ***null***, or containing data in its ***root***, and having a pair of trees called its ***left*** and ***right*** subtrees, each of which is itself a tree.

In this assignment you will implement a symbolic calculator which stores arithmetic expressions in the form of binary trees (abstract syntax trees) in a symbol table. Input will be in reverse Polish notation (RPN), named so after its invention by Polish mathematician Jan Łukasiewicz. Hewlett-Packard calculators commonly use this notation. Infix notation, such as is used in Java, is harder to parse and will not be used. The operators to be implemented are : assignment of an expression, assignment a value, evaluation of an expression, and the usual four arithmetic operators `+`, `-`, `*`, and `/`.

Reference : **`http://en.wikipedia.org/wiki/Reverse_Polish_notation`**

2. Program Specification

The program is specified in the format of a Unix `man`(1) page.

NAME

bitreecalc – binary tree RPN calculator

SYNOPSIS

`bitreecalc` [`-e`] [`-o`*outputfile*] [*inputfile . . .*]

DESCRIPTION

Each line of input is read as a separate statement, and executed as it is read. An input line may have one of four formats :

variable `=` *expression*

The RPN expression is converted into a tree and stored in the symbol table entry associated with the variable. A postorder traversal of the tree is then done in order to evaluate it, using the current values of all variables. This new value is then stored in the symbol table associated with the variable. The value and tree are then printed as for the query (`?`) operator.

variable : *number*

The number is stored in the symbol table and the tree is set to ***null***. The value and tree are then printed.

variable `?`

The tree is traversed and the value of the variable is recomputed, in case any of the values of the operand variables have changed. The variable is found in the symbol table and its value and tree are printed. The tree is printed using a fully-parenthesized in-order traversal.

`#` *anything*
: Any line whose first non-whitespace character is a hash is ignored as a comment.

A variable is any one of the 26 ASCII letters, either upper or lower case. An expression consists of either a variable, or two expressions followed by an operator. An operator is one of `+`, `-`, `*`, or `/`. White space is ignored, and any other character causes an error message to be printed. It is also an error if the conversion stack underflows or contains more than one element at the end of the line.

At the end of execution, all symbol table entries are printed to the output in the same format as individual-line output, including both the value and the tree.

OPTIONS

Any word of the command line which begins with a minus sign is an option. A word of the command line consisting only of a minus sign is an operand, not an option. Options and operands may appear in any order or mixed in any way as separate flags or single-word flags.

`-e` Each line of input is echoed to the output verbatim and preceded by four asterisks and a space immediately before being interpreted. Without this option, input is not echoed.

`-o` *outputfile*
: Output is written to *outputfile*. If this option is not specified, output is written to the standard output. Regardless of the `-o` flag, error messages are always written to the standard error. If the word specifying the `-o` option is not followed by a filename, then the next word from the command line is used as the output file. If the output file can not be opened, the program terminates immediately with an error message.

OPERANDS

If no input files are specified, the standard input is read. Otherwise each of the filenames specified are read in turn, one after the other, as if they were all concatenated together. If an input file is specified as a single-character word which is a minus sign (`-`), then the standard input is read at that point. If an input file can not be opened, an error message is printed, and reading continues with the next file.

EXIT STATUS

0 All input files were valid, no output errors were generated, and no invalid options were found.

1 An error occurred opening an input or output file.

3. Implementation Sequence — Phase 1

Following is the implementation sequence, divided up into several phases. Complete the implementation of each phase in the given sequence and state in your **`README`** how far you have gotten when you submit your program. You might also want to play with **`bitreecalc.perl`**, which is a partial implementation of your program and may provide some sample output.

(1) The most important part of the assignment is the ADTs used, namely the `bitree`, the `linked_stack`, and then `symbol_table`. For now, ignore all options and operands, and read all input from the standard input, write all non-error output to the standard output, and all error messages to the standard error. Write code for the `-e` option to be permanently turned on until a later phase. The function `Double.isNaN` is used to check to see if a number is a `NaN` or not.

(2) First, complete the parser :

  (a) Write a function which can be called to scan a line, skipping white space, and return the next non-whitespace character on the line. Indicate whether this has succeeded or run off the end of the line.

  (b) Then scan a line as follows, for each line :

    (i) Read the line, exit from the loop at end of file.

    (ii) Print the line.

    (iii) Scan the line using the function above and return the first (non-whitespace) character. If it is not a variable, print an error message and break out of the scan. Otherwise continue scanning.

    (iv) Scan for the operator, which must be one of equal (`=`), colon (`:`), or query (`?`). If not, print an error message and break out of the scan. Otherwise handle each of the three cases listed here by printing a stub message, to be filled in later :

      `=` Scan and print each (non-whitespace) character following the equals sign. Print error messages for invalid characters. Valid characters are upper or lower case letters or one of the four arithmetic operators.

      `:` Use `Double.parseDouble` to scan the rest of the line (following the colon). Print the resulting double, or error out if `NumberFormatException` is raised.

      `?` Stub for print statement.

(3) Bring in the code for `linkedstack` and `symboltable`. This has been written for you, so you only need compile and use it. Implement the query (`?`) operator. To do this, use the variable to look up entries in the symbol table and print the results. Now, all numbers will be `NaN`s and all trees will simple print the word `null`. To print the tree line, you can use: `out.printf ("␣␣␣%s%n", tree);` This will cause the word `null` to be printed if the variable `tree` is null, and call `tree.toString` if not. You will need to implement `bitree.toString` later. Note that the printing routine is called by all three operators, so don't write the code inline.

(4) Implement the numeric assignment operators. In this case, just put the number into the symbol table. Then thest the query operator again.

(5) Implement the `bitree` class :

(a) A single parameter ctor will be used for a letter, and will set the left and right child pointers to null.

(b) A three parameter ctor will take an operator character, followed by two tree pointers and make a new tree with the two parameter trees as the left and right child nodes, respectively.

(c) The method `toString` will return a fully parenthesized string representation of the tree. For a leaf node, the character field is just returned as a string. For a non-leaf node, call the `toString` methods recursively for each non-null child tree, and concatenate the left and right child strings and the node string in inorder fashion and enclose them in parentheses.

(d) The method `eval` with no parameters will return the `double` numeric value of the tree by doing a post-order traversal, switching based on the arithmetic operators or variables in the node.

(6) Now rewrite your stub for the equals operator. This will involve a scan over the variables and operators following the equal sign. First, ensure that the stack is empty. For each such character found :

(a) If you have a variable, make a one-node tree and push it onto the stack.

(b) If you have an operator, Pop two trees from the stack and make them children of the new node with the operator as the data item. Push the result onto the stack. Note that the right operand must be the one previously at the top of the stack and the left one must be the one previously under it. Print an error if the stack underflows, and quit the scan, continuing with the next line.

(c) If you have any other character, print a syntax error message, and continue with the next line.

(d) At end of line, pop the stack. If the stack is not now empty, print an error message and continue with the next line. Otherwise insert the tree into the symbol table and find its numeric value.

(7) Add code for the final symbol table dump. Now, you are done with the major work of the program.

4. Implementation Sequence — Phase 2

If you have completed the first phase, you are done with the bulk of the work. Now go back and retrofit your code.

(1) Implement an options analyzer. Note that options and operands can go in any order.

(2) Set the necessary flags so that the echo and end of file dump are done only if the options are specified. Note that the test runs may have these options turned on, so in the previous phase, don't print error messages if you don't have them implemented. But do perform the dump.

(3) Use the output option to reroute the standard output, but not the standard error.

(4) Redirect input so that it comes from the standard input if no input files are specified. But if they are, sequence through them in order. As you are sequencing through them in order, treat the filename `-` specially: read the standard input at that point.

5. Computation and IEEE-754 floating-point arithmetic

Computation will be done using IEEE Standard 754 double-precision floating point arithmetic (Java `double`), which can be used to represent values of up to 1.7976931348623157e308 in magnitude, and also `+Infinity`, `-Infinity`, and `NaN`. This last value means "not a number", and is the result of doing indeterminate computations such as dividing 0.0 by 0.0. Mathematically, `+Infinity` means $+\infty$, and `-Infinity` means $-\infty$,

Generally, an infinite value will result if the magnitude of a result exceeds the above mentioned maximum double-precision quantity. Division of a finite quantity by zero also produces an infinite value. You need not check for this, as the compiler will handle it automatically. Division by zero is not an error. `NaN` is an indeterminate quantity. It represents the result of doing things like dividing zero by zero or adding positive infinity to negative infinity.

6. Sample Input and Output

The following table contains some sample input in the column at the right, with some sample output from that input in the middle column, and a summary explanation in the third column.

`a: 1.7`	`a: 1.7`
	`    null`
		Store the number in `a` and set its tree to `null`. Then print out the value and tree of `a`.

`b= ac+ ac+*`	`b: NaN`
	`    ((a+c)*(a+c))`
		Store a pointer to the tree in `b`. Then evaluate the tree and store its value in `b`. The result is `NaN` because `c` is undefined.

`c:3`	`c: 3`
	`    null`
		Store `3` in `c`, set its tree to `null`, and print it out.

`b ?`	`b: 22.09`
	`    ((a+c)*(a+c))`
		`b` is re-evaluated and its new value is stored in the symbol table and printed.

`z :0`	`z: 0`
	`    null`
		Zero.

`y= a z/`	`y: Infinity`
	`    (a/z)`
		`y`'s tree is set and evaluated. Zerodivide produces (not a number).

7. What to Submit

Submit the following files. The names must be exactly as given. Every file must contain your name and username at the top of the file. As before, after you have done a submit, verify that everything necessary has been submitted.

`README`	your name and username and a statement of which major parts of the implementation strategy you have completed.
`Makefile`	has, among others, the following targets :
	`all` — must be the first and must build the file `bitreecalc`.
	`clean` — removes all class files generated.
	`spotless` — depends on `clean` and also removes the jar file.
	`submit` — submits your files.
`bitreecalc.java`	the calculator main function, scanner, and auxiliary functions.
`bitree.java`	containts the `bitree` ADT, its constructors, the `toString` method, etc.
`linkedstack.java`	the implementation of the `stack` ADT, taken from the examples.
`symboltable.java`	the simple symbol table implementation, provided in the code subdirectory.
`auxlib.java`	handles miscellaneous system interaction

8. Prohibit `java.util.*`

In this course, you are not permitted to use any class from `java.util` unless explicitly authorized. The grader will use the command `grep util` on your source code and investigate anything that pops up. If it appears in any statement other than one of the following, you will lose many points.

```
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.Scanner;
```

9. Miscellaneous

You may do pair programming if you like. Please read `/afs/cats.ucsc.edu/courses/cmps012b-wm/Syllabus/pair-programming/` for details. The directory `.score` has test data, and the program `misc/pbitreecalc.perl` has a reference implementation.