

# Project 4: Postfix Calculator

CSC 1351, Spring 2006

Due: 23 April 2006

Design and implement a calculator for *postfix* expressions. Your interpreter should read a sequence of postfix expressions and print each expression in *infix* form together with its value.

A *postfix* expression is an expression in which the operands are followed by the operator. An *infix* expression is an expression in the familiar format, where the operator is in between the operands. Similarly, a *prefix* expression is an expression where the operator comes before the operands. For example, the infix expression  $6 * (5 - 2 + 4)$  can be written in postfix form as follows:

6 5 2 - 4 + \*

Hewlett-Packard pocket calculators used postfix input expression. The programming languages Lisp and Scheme use prefix expressions.

Your calculator should recognize positive integer constants, the binary operators  $+$ ,  $-$ ,  $*$ , and  $/$ , the unary operator  $\sim$  (negation), and the equal sign  $=$  at the end of an expression.

For each expression, your calculator should build a tree representation of the expression, then print it in infix form and calculate and print its value. Each expression should be printed on a line by itself. Parentheses should be inserted as necessary. Expressions are only needed if addition or subtraction operators are used in the operands of multiplication or division expressions or if expressions are right-associative. Negation should be printed as a unary minus sign.

Your calculator should read from `System.in` and write to `System.out`. Any error messages for incorrect input or division by zero should be printed to `System.err`.

For example, the input

```
6 5 2 - 4 + * =
2 3 + = 4 5 ~ * =
1 2 3 4 5 * * * * =
1 2 * 3 * 4 * 5 * =
1 2 + 3 * 4 + 5 * =
```

should result in the following output

```
6*(5-2+4) = 42
2+3 = 5
4*(-5) = -20
1*(2*(3*(4*5))) = 120
1*2*3*4*5 = 120
((1+2)*3+4)*5 = 65
```

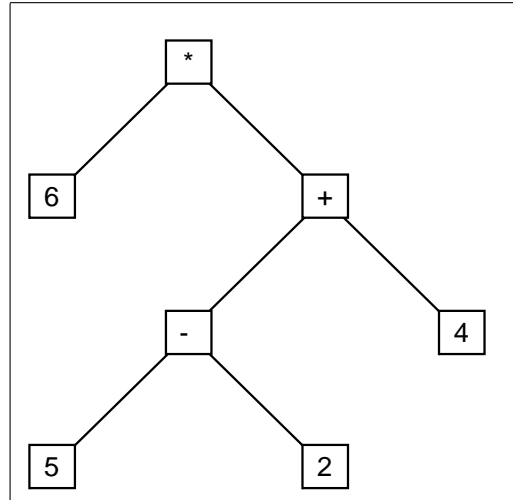


Figure 1: Expression tree for  $6 * (5 - 2 + 4)$ .

## Implementation

For reading the input, you can use a tokenizer similar to the one used at the end of Chapter 18 in the book or you could use class `Scanner` directly. Allow any white space as delimiter, i.e., "`( | \t | \n | \r | \f ) +`".

A postfix *expression* has one of the following three forms:

*integer-constant*  
*expression* ~  
*expression expression binary-operator*

Each input expression is followed by an equal sign.

After parsing, you should get a parse tree data structure representing the expression. In your parse tree, you have leaf nodes, unary nodes, and binary nodes. The typical data structure for representing such trees is built using a class hierarchy as follows:

```

enum Operator { PLUS, MINUS, TIMES, DIVIDE }
abstract class Tree { ... }
class Leaf extends Tree { private int value; ... }
class Unary extends Tree { private Tree exp; ... }
class Binary extends Tree {
    private Operator op;
    private Tree left;
    private Tree right;
    ...
}
  
```

Figure 1 shows the expression tree for the expression  $6 * (5 - 2 + 4)$ . Equal signs are not represented in the tree.

For parsing the input, the recursive structure used in class `Evaluator` won't work because an expression can start with an expression. You wouldn't know how many recursive calls to make until you have seen the entire expression. We therefore need to use a *stack* data structure for parsing.

A stack is a collection of objects like a queue, but with a stack you add and remove objects on one end of the stack. You can easily implement the stack with an array and a *top-of-stack* index into the array.

You would use a stack of expressions. When you encounter a number, you'd put a `Leaf` node onto the stack. When you encounter a unary operator, you'd remove the expression on the top of the stack, construct a tree with a `Unary` node at the top, and put it onto the stack. When you encounter a binary operator, you'd remove the top two expressions from the stack and put a `Binary` tree onto the stack. For simplicity, you can limit the stack size to as little as ten elements (HP calculators can have only four elements on the stack), but you need to print an error message if you exhaust the stack space.

Print all error messages to `System.err`. For grading convenience, please start all error messages with the string `Error:.` When you encounter an error in an input expression, the best error recovery approach is to remove all the tokens up to and including the next equal sign.

## **Administrative Stuff**

Put your files in the directory `~/prog4` in your `cs1351xx` account on `byte` and submit it using

```
~cs1351c/bin/r_copy 4
```

Your main program should be in class `main.Main`. Keep the tree data structure in a separate package. Also submit a `README` file in which you briefly describe how you designed your code whatever else you want the grader to know.