

Forum:[Hack Forums](#)

Thread: [Shunting Yard Algorithm Calculator - Parses and solves math expressions](#)

Trisomy 21 - September 18, 2014, 8:17 pm

I actually made this awhile ago, but with HF being down I got bored and cleaned this code up a tiny bit, then I came on HF and searched and realized there were no SYA algorithm/calculator threads, in the Java section at least, so I thought I would post what I have here. It supports the following operations: +, -, *, /, ^ and parses parentheses correctly. If you have any questions about the code ask away, and also if you can recommend how I can do some things different/better please do, thanks.

All you do is send the 'solve(String in)' method a string like "6/2+(3^2)+4*3" and it will return the answer, in this case 24.

Also I have not implemented any error catching at all, if you give improper input the program will just crash.

I added some commenting but it screwed up the neat formatting :(

Main.java (just showing how you can use the class)

PHP Code

<?php

```
import java.util.*;
```

```
public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.solve("6/2+(3^2)+4*3"));
    }
}
```

Calculator.java

PHP Code

<?php

```
import java.util.*;
```

```
public class Calculator {
    HashMap<Character, Integer> operators = new HashMap<Character, Integer>();
    // holds operators and their precedence

    public Calculator() {
        /* set operator precedence */
        operators.put('+', 2);
        operators.put('-', 2);
        operators.put('*', 3);
        operators.put('/', 3);
        operators.put('^', 4);
    }
}
```

```

public double solve(String input) {
    ArrayList postfix = infixToPostfix(stringToArrayList(input));
// convert infix expression to postfix
    Stack stack = new Stack();
    for (Object token: postfix) {                // for all tokens in 'postfix'
        if (token instanceof Double)             // if token is a number
            stack.push(token);
        else if (operators.containsKey(token)) { // if token is an operator
            double temp = (double)(stack.pop());
            switch ((char)(token)) {
                case '+': temp = (double)(stack.pop()) + temp; break;
                case '-': temp = (double)(stack.pop()) - temp; break;
                case '*': temp = (double)(stack.pop()) * temp; break;
                case '/': temp = (double)(stack.pop()) / temp; break;
                case '^': temp = Math.pow((double)(stack.pop()), temp); break;
                default:;
            }
            stack.push(temp);
        }
    }
    return (double)(stack.pop());
}

private ArrayList stringToArrayList(String in) {
    ArrayList out = new ArrayList();
    String temp = "";
    for (int i = 0; i < in.length(); i++) {        // for all characters in 'in'
        char token = in.charAt(i);
// give value of next character to 'token' for easier readability
        if (isNumeric(Character.toString(token)) || token == '.') // if token is numeric
            temp += token;
        else if (operators.containsKey(token) || token == '(' || token == ')') { // if token is an operator
            if (!temp.isEmpty()) {
                out.add(Double.parseDouble(temp));
                temp = "";
            }
            out.add(token);
        }
    }
    /* if 'temp' is not empty parse it to a double and add it to 'out' */
    if (!temp.isEmpty())
        out.add(Double.parseDouble(temp));

    return out;
}

private ArrayList infixToPostfix(ArrayList in) {
    ArrayList out = new ArrayList();            // postfix output queue
    Stack stack = new Stack();                  // operator stack
    for (Object token: in) {                    // for next token of 'in'
        if (token instanceof Double)            // if token is a numbe

            out.add(token);

```

```

else if (operators.containsKey(token)) { // if token is an operator

    while ((!stack.empty() && (char)stack.peek() != '(') &&
        (((char)(token) != '^' && operators.get(token) <= operators.get(stack.peek())) ||
        operators.get(token) < operators.get(stack.peek()))
        out.add(stack.pop());
        stack.push(token);
    }
    else if ((char)(token) == '(') // if token is left parentheses
        stack.push(token);
    else if ((char)(token) == ')') { // if token is right parentheses
        while ((char)(stack.peek()) != '(')
            out.add(stack.pop());
        stack.pop();
    }
}
/* pop remaining operators off stack and into 'out' */
while (!stack.empty())
    out.add(stack.pop());

return out;
}

private boolean isNumeric(String s) {
    try {
        Double.parseDouble(s);
    }
    catch (NumberFormatException e) {
        return false;
    }
    return true;
}
}

```

??????M1kep?????? - September 19, 2014, 12:25 am

This is pretty awesome! One thing is I dint understand why people don't upload things to git hub, makes it much easier to read, but atm I'm on mobile but I'm going to for sure check this out tomorrow, is the code commented well?

Trisomy 21 - September 19, 2014, 7:58 am

(Sep 19, 2014 12:25 AM)??????M1kep?????? Wrote: [\[CITE\]](#) This is pretty awesome! One thing is I dint understand why people don't upload things to git hub, makes it much easier to read, but atm I'm on mobile but I'm going to for sure check this out tomorrow, is the code commented well?

Not at all commented lol, I'll get around to it later though. I think the only part that is hard to read without comments is the long while expression in the middle of infixToPostfix() that executes when the next token in the input array is an operator.

Heres some pseudo code to clear up what that part is doing.
 Code While (there is an operator at the top of the stack)
 and either
 {(token is left associative and its precedence is less than or equal to precedence of operator at top of stack)

Or

(Precedence of token is less than precedence of operator at top of stack)}

Cosa Nostra - *September 19, 2014, 1:00 pm*

I prefer using finite state machines to accomplish something like this, coupled with a fundamental language and recursive descent parsing; this results in much cleaner code. But, hey, if it works, don't knock it.