**Formal Languages and Automata Theory, SS 2018. Project (due Week 14)**

# 1 Preliminaries

The **objective** is to implement an algorithm for the evaluation of an arithmetic expression. As **input**, we have a string of characters representing an arithmetic expression. As **output**, we want the result of evaluating the expression. For example:

- input data: file `eval.in` contains on its first line a string of characters composed by numbers ('0', '1', ..., '9'), operators ('+', '-', '*', '/') and parentheses ('(', ')' ).
- output data: file `eval.out` contains a number representing the value obtained by evaluating the expression from the input file.
- **Example**: `eval.in`: $(1 + 1) * 13 + 10/2$, `eval.out`: 31.

**Methods**:

1. (Reversed) Polish notation
2. Binary trees

## 1.1 (Reversed) Polish notation

Arithmetic expressions are everywhere in computer science, and not only[1]. Typically, an arithmetic expression is written in *infix notation*: `Operand1 op Operand2`, where `op`$\in \{+, -, *, /, \%\}$. Note that + and − can be used as binary and unary operators, but here we assume that all our operations are binary.

There are rules to indicate which operations take precedence over others, and we often use parentheses to override those rules. Example: $3 * 4 + 5$ vs. $3 * (4 + 5)$.

It is also possible to write arithmetic expressions using:

- *prefix notation (Polish notation)*: `op Operand1 Operand2`
- *postfix notation (reversed Polish notation)*: `Operand1 Operand2 op`.

It is possible to use a *stack* to find the overall value of an expression. In case of prefix notation, if you push an operator, then its operands, you need to have forward knowledge of when the operator has all its operands. Basically one needs to keep track of when operators pushed have all their operands so that you can unwind the stack and evaluate. With suffix, when you push an operator, the operands are (should) already on the stack, simply pop the operands and evaluate. You only need a stack that can handle operands, and no other data structure is necessary.

The idea is to think of an expression as a list or sequence of items, each of which is a left parenthesis, right parenthesis, operand, or operator. An operand can be a constant or the name of a variable. Presumably it would be necessary at some point to replace each variable with its value.

There are two algorithms involved. One converts an infix expression to postfix form, and the other evaluates a postfix expression. Each uses a stack.

[1] This section is mainly based on `http://faculty.cs.niu.edu/~hutchins/csci241/eval.htm`

### 1.1.1 Transform an infix expression to postfix notation

Suppose $Q$ is an arithmetic expression in infix notation. We will create an equivalent postfix expression $P$ by adding items to on the right of $P$. The new expression $P$ will not contain any parentheses. We will use a stack in which each item may be a left parenthesis or the symbol for an operation.

```
Start with an empty stack.  We scan Q from left to right.
While (we have not reached the end of Q)
     If (an operand is found)
          Add it to P
     End-If
     If (a left parenthesis is found)
          Push it onto the stack
     End-If
     If (a right parenthesis is found)
          While (the stack is not empty AND the top item is not a left parenthesis)
              Pop the stack and add the popped value to P
          End-While
          Pop the left parenthesis from the stack and discard it
     End-If
     If (an operator is found)
          If (the stack is empty or if the top element is a left parenthesis)
               Push the operator onto the stack
          Else
               While (the stack is not empty AND
                 the top of the stack is not a left parenthesis AND
                 precedence of the  operator <= precedence of the top of the stack)
                     Pop the stack and add the top value to P
                End-While
                Push the latest operator onto the stack
          End-If
     End-If
End-While
While (the stack is not empty)
     Pop the stack and add the popped value to P
End-While
```

At the end, if there is still a left parenthesis at the top of the stack, or if we find a right parenthesis when the stack is empty, then $Q$ contained unbalanced parentheses and is in error.

### 1.1.2 Evaluate a postfix expression

Suppose $P$ is an arithmetic expression in postfix notation. We will evaluate it using a stack to hold the operands.

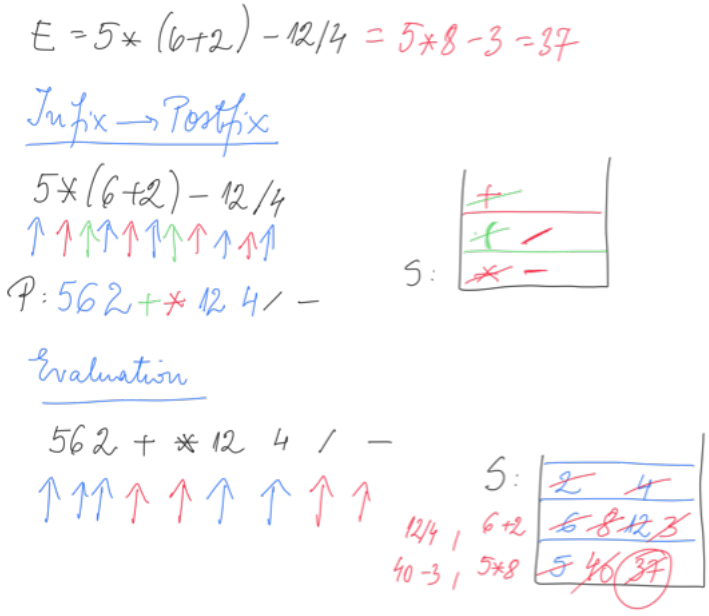
```
Start with an empty stack.  We scan P from left to right.
While (we have not reached the end of P)
     If an operand is found
```

```
        push it onto the stack
    End-If
    If an operator is found
        Pop the stack and call the value A
        Pop the stack and call the value B
        Evaluate B op A using the operator just found.
        Push the resulting value onto the stack
    End-If
End-While
Pop the stack (this is the final value)
```

At the end, there should be only one element left on the stack. This assumes the postfix expression is valid.

### 1.1.3 Example

**Example 1.** Let $E$ being the following infix expression: $5*(6+2)-12/4$. The equivalent postfix expression $E'$ is: $562+*124/-$. It was obtained using the algorithms above

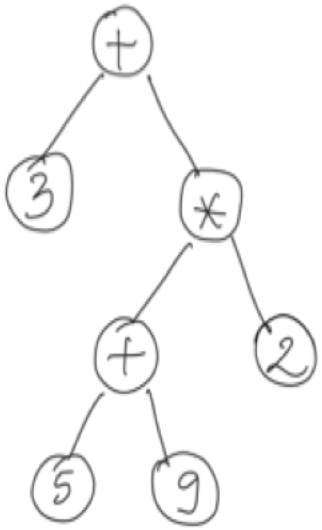



**Other examples**

| Infix Notation | Prefix Notation | Postfix Notation |
|---|---|---|
| $A+B$ | $+AB$ | $AB+$ |
| $A+B*C$ | $+A*BC$ | $ABC*+$ |
| $(A+B)*C$ | $*+ABC$ | $AB+C*$ |
| $(a-b)*c$ | $*c-ab$ | $ab-c*$ |
| $a-b*(c+d)$ | $-a*b+cd$ | $abcd+*-$ |

## 1.2 Binary Trees

Expression tree is a binary tree in which each internal *node* corresponds to operator and each *leaf* node corresponds to operand. For example, the tree for the expression $3 + ((5 + 9) * 2)$ is:




*Inorder* traversal ((Left, Root, Right)) of expression tree produces infix version of given postfix expression (same with *preorder* ((Root, Left, Right)) traversal it gives prefix expression).

Also in this case, there are two algorithms involved for evaluating an expression: *construction of expression tree* and *evaluation of the expression represented by expression tree.*

### 1.2.1 Construction of Expression Tree

For constructing expression tree we use a stack. We loop through input expression and do following for every character:

- If character is operand push that into stack
- If character is operator pop two values from stack make them its child and push current node again.

At the end the only element of stack will be root of expression tree.

### 1.2.2 Evaluating the expression represented by expression tree

```
solve(t)
     Let t be the expression tree
     If t is not null then
          If t.value is operand then
               return  t.value
          A = solve(t.left)
          B = solve(t.right)
          // calculate applies operator 't.value' on A and B, and returns value
          return calculate(A, B, t.value)
```

# 2 Homework

1. Write an algorithm in your favorite programming language which performs the evaluation of an arithmetic expression using the reversed Polish notation.

2. (Optional) Same requirement as above, but by using binary trees.