Command-line Calculator

Project n°2

Formal Languages
Summer semester 2025

HAUTEFAYE Corentin

Contents

1 Introduction

Within the scope of this class on *Formal Languages*, the second assignment I had to work on was to implement a command-line calculator in Java, C or Python. The goal of this project was to understand how context-free grammars can be parsed then evaluated using a stack.

This program should able to handle expressions that include basic arithmetic operators, such as $+, -, \times, \div, \%$. It should also take into account nested expressions as well as associativity and usual precedence. Moreover, it takes integers and variables as parameters in expressions. If a variable is encountered while parsing, then its value should be prompted to the user.

Furthermore this calculator should use a stack-base approach for the evaluation of expressions. Therefore it must display any transformation onto that stack, then the result.

2 Algorithms & Implementation

I have chosen to implement this project in C99 as I am much more comfortable in using that language. Moreover, I only needed structures and not classes to program, thus the choice was quite quick.

Finally, this project has been compiled on Windows with CLion. It should be able to compile also on any Linux distribution with GCC installed.

2.1 Stack

In order to evaluate arithmetic expressions using stacks, we first need to define such structures, whence : $\texttt{StackNode} : \begin{cases} \texttt{value} & : \texttt{integer} \\ \texttt{next} & : \texttt{StackNode} \end{cases}$ and $\texttt{Stack} : \begin{cases} \texttt{size} & : \texttt{unsigned integer} \\ \texttt{top} & : \texttt{StackNode} \end{cases}$.

Usually, stacks are defined as a continuous and bounded block in memory, with a base address and a pointer to the top of the stack. Here, I have chosen to represent stacks as non-continuous and unbounded blocks in memory. I do not need indeed to have all data stored continuously as pushing and popping values using dynamic allocation could be inefficient. Moreover, since I do not know how much data I will need to store, I cannot bound the size of the stack.

2.2 Expressions Checking

Let $\mathcal{A} = (0-9)^* \cup (A-Z)$ and $\mathcal{O} = \{(,), +, -, *, \%\}$. Thus we can formally define a grammar $G = (\mathcal{A} \cup \mathcal{O}, \{X, T, F, N, D, I\}, \rightarrow, X)$ such that we have the following axioms:

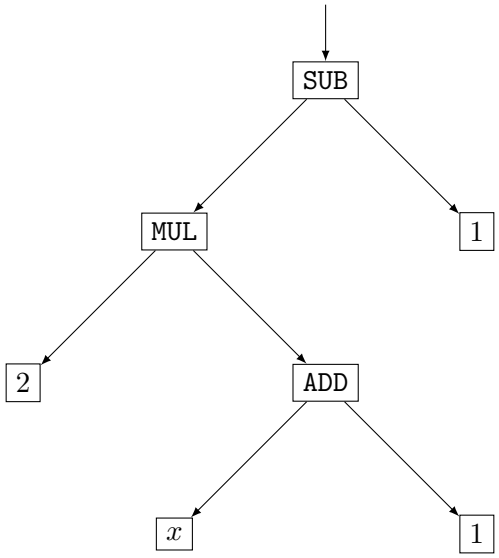
$$\begin{cases} X & \rightarrow X + T \mid X - T \mid T \\ T & \rightarrow T * F \mid T \,/\, F \mid T \,\%\, F \mid F \\ F & \rightarrow (X) \mid N \mid I \\ N & \rightarrow DN \mid D \\ D & \rightarrow 0 - 9 \\ I & \rightarrow A - Z \end{cases}$$

Thus, our program should be able to handle arithmetic expressions described by G. Using a top-down recursion allows us to check whether a given expression belongs to $L(G)$. As you can see, variables are one letter only. Therefore my current implement only allows for the use of each letter of the standard Latin alphabet, whence 26. Besides the deepest part of the recursion (*for variables*) also takes care of getting the value of each variable that has not been seen before.

Furthermore, I have decided that any whitespace encountered while parsing an expression should be discarded. Thus the amount of spaces does not matter. Also the current implementation is case insensitive for variables (*lowercase letters are replaced by their uppercase counterpart*).

2.3 Parsing

Let's consider $E = 2 * (x + 1) - 1$. Thus we can draw the following binary tree T_E:



Let T be a binary tree. For a given node N in T, we will respectively denote its left and right children by L and R. Thus, it is possible to read this tree in different ways:

Infix We go through T in the following order LNR. Therefore, for T_E we would get $((2) * ((x) + (1))) - (1)$ whence the starting expression E. Please mind the fact that in this kind of traversal, it is necessary to add parenthesis not to make any kind of mistake, only then some may be removed.

Prefix We go through T in the following order NLR. Therefore, for T_E we would get $- * 2 + x\ 1\ 1$. Let's notice that we have operators before their operands, and we do not need parenthesis. This traversal is great but not enough to be able to use stacks as it is in the wrong order.

Postfix We go through T in the following order LRN. Therefore, for T_E we would get $2\ x\ 1\ +\ *\ 1\ -$. Here, we have operands before operators, and still no parenthesis.

Therefore we will use the postfix notation of any given expression to proceed with a stack-based evaluation. Thus, we need to parse it in such a way that we get that kind of notation at the end.

Thereby we need to use a pushdown automaton N_P to perform such transformations. Thus we will denote states by $(S_i)_{i\in[\![0;7]\!]}$. Moreover, our automaton will use a stack P to hold operators and will write the postfix expression into a string which we will call S. Both S_6 and S_7 are final states. Once the whole string has been processed, if N_P is not in state S_6, then it makes a transition to state S_7. Thus, we have the following transition table:

State	Character	Next state
S_0	' '	S_0
S_0	$0-9$	S_1
S_0	$+,-,*,/,\%$	S_2
S_0	'('	S_3
S_0	')'	S_4
S_0	$A-Z$	S_5
S_0	ϵ	S_6
S_1	$0-9$	S_1
S_1	ϵ	S_0
S_2	ϵ	S_0
S_3	ϵ	S_0
S_4	ϵ	S_0
S_4	ϵ	S_6
S_5	ϵ	S_0

Table 1: Transition table (Transitions to S_7 are not in this table)

State S_0 This is the starting state of this automaton, and also the general state. This part skips all whitespace and decides which action to apply regarding the current character that is being read. Any invalid or unknown character yields the state S_6.

State S_1 This state parses integers. Once the whole integer has been processed, it is written into the buffer S and the automaton returns in state S_0.

State S_2 This state parses operators and performs two operations. First, if P is not empty, it will compare the precedence of the operator O at the top of P with the current one being parsed. If O has a greater precedence, then it is removed from P and written to the buffer S. This operation is repeated until either P is empty or the operator at its top has a lower precedence. Finally, the given operator is pushed onto P, and the automaton returns to general state S_0.

State S_3 This state parses left parenthesis and pushes this token onto P.

State S_4 This state parses right parenthesis. It will remove each operator from P and write them to the buffer S until P is either empty or its top is a left parenthesis. If no left parenthesis is encountered in P, then N_P transitions to S_6 otherwise it returns to state S_0.

State S_5 This state parses user-defined variables and write it directly to the buffer S. N_P goes back to state S_0 afterwards.

State S_6 This is the error state. Here memory is freed and N_P does not perform any more transitions.

State S_7 N_P is put in this state once the whole string has been processed and no error has occurred. If P is not empty, it will remove each element from it and write them to the buffer S. Memory is freed and the final string S is returned.

Nota bene: Each part of the final string is separated by a whitespace so that everything is clear. Indeed, if we have 42, we do not know if we should interpret it as 42 or 4 and 2, whence 4 2.

2.4 Evaluation

Let E be an expression written in the postfix notation. The evaluation of such a form is trivial, whence the following algorithm:

Algorithm 1: Evaluate a string in postfix notation

```
Input: S: string, V: array<integer>[26]
Output: Integer
Function evaluate(S: string, V: array<integer>[26]) : integer
Begin
    If S = ∅ Then
        // Error
        evaluate ← 0;
    End If
    P : Stack ← ∅;
    i : unsigned integer ← 0;
    While S[i] ≠ '\0' Do
        If S[i] = ' ' Then
            i ← i + 1;
        Else
            If S[i] ∈ (A − Z) Then
                // S[i] is an ASCII character and 'A' = 65
                P ← push(P, V[S[i] − 65]);
                i ← i + 1;
            Else
                If S[i] ∈ (0 − 9) Then
                    val : integer ← 0 While S[i] ≠ '\0' AND S[i] ≠ ' ' Do
                        val ← val ∗ 10;
                        val ← val + (S[i] − '0');
                        i ← i + 1;
                    End While
                    P ← push(P, val);
                Else
                    If is_operator(S[i]) Then
                        b : integer ← pop(P);
                        a : integer ← pop(P);
                        // get_result returns the result (integer) of a by b
                           with the operator S[i]
                        tmp : integer ← get_result(a, b, S[i]);
                        P ← push(P, tmp);
                        i ← i + 1;
                    Else
                        // Error here
                        free(P);
                        evaluate ← 0;
                    End If
                End If
            End If
        End If
    End While
    res : integer ← pop(P);
    free(P);
    evaluate ← res;
End
```

3 Example

```
Please enter an arithmetic expression:(x+3)*6-2/4
Enter value for 'X':5
======================
Expression in stack-compatible form: X 3 + 6 * 2 4 / -
----------------------
->|5|
->|3|5|
(+) ->|8|
->|6|8|
(*) ->|48|
->|2|48|
->|4|2|48|
(/) ->|0|48|
(-) ->|48|
----------------------
Thus, (X+3)*6-2/4=48
======================
Would you like to evaluate another expression (Y/N):n
```

4 Conclusion & Feedback

4.1 Difficulties

I did not really have any kind of difficulty for this project.

4.2 Suggestions

There are some parts of this program which could be improved. For instance, we could add other operators and/or functions such as $\hat{}$(power), $\exp, \sin, \arctan, \ldots$ We could also add the possibility to handle floating numbers and precision. Finally, being able to use whole identifiers for variables could be a good idea.