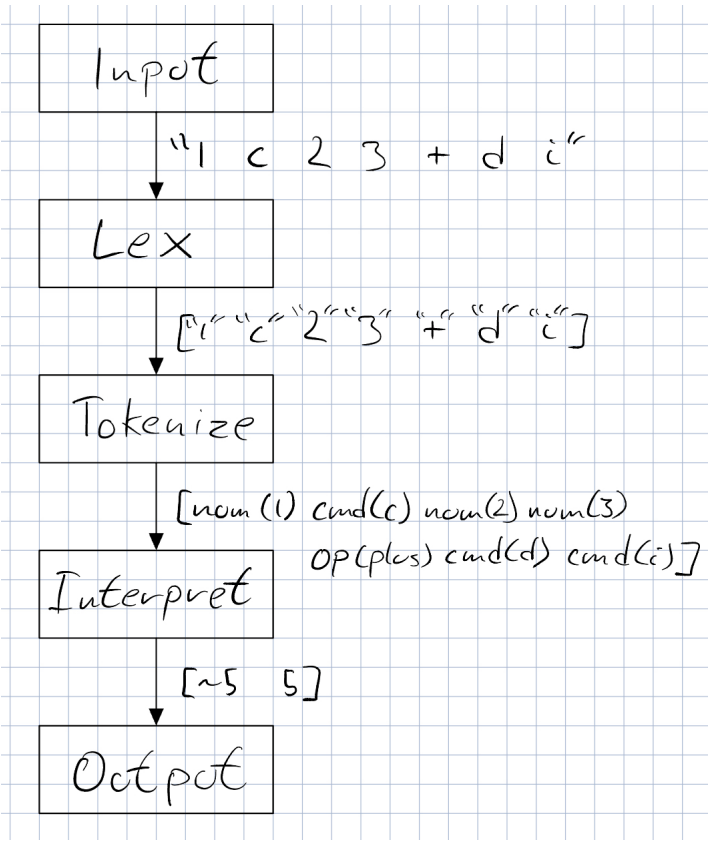
Programmeringsspråk TDT4165 - Assignment 2

Ola Kristoffer Hoff

27. September 2022

1 mdc

1.1 You also need to give a high level description of how your mdc works, i.e., how it takes the postfix expression and computes the result.



The figure above shows the flow of how my **mdc** works. First I take the input through the *Lex*-stage which seperates the input string into seperate lexemes. These are then sent through the *Tokenize*-stage where they are classified into *operator*, *command* and *number* records. The tokens are then sent through the main function, the *Interpret*-stage. Here I call a helper function to deal with the mdc internal stack, the helper function is called with a full token list and an empty stack. It then does the following: If there are no tokens, return the stack.

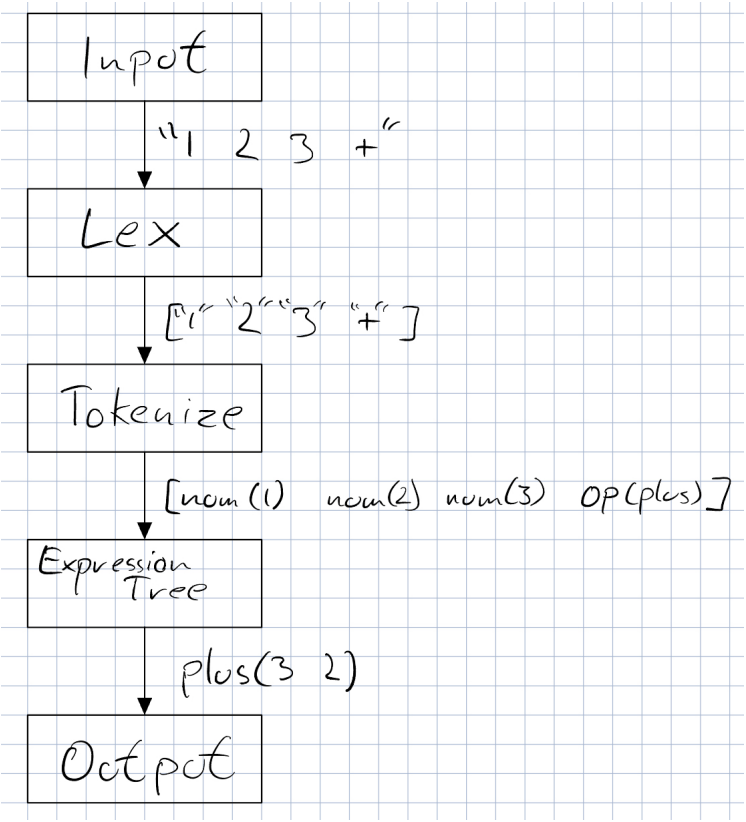
Otherwise check what the next token is. If token is an operator then the helper-function calls itself with the rest of the tokens (excluding the one just classified) and the product of the operation concatenated with the stack, excluding the two upper values. This is equivalent to takeing the operation of the two top numbers of the stack (and removing these elements from the stack) and adding the product to the stack.

If the token was a command the helper-function does what the command should: p: print the stack, d: duplicate the top element of the stack, i: flips the sign og the top element on the stack, and c: clear the stack. It then calls itself with the respective stack and the remaining tokens.

Otherwise, if the token was not a operator nor a command, it must be a number, in this case it is added to the stack and the helper-function calls itself with the remaining tokens and extended stack.

2 Convert postfix notation into an expression tree

2.1 In addition, you need to give a high level description of how you convert postfix notation to infix notation.



In the figure above we see the control flow of how I implemented the *ExpressionTree*. This works very similarly to the previous task. The only difference is the fourth block, here we push the token to the stack if it is a number and when we encounter a operator we pop the to top elements and put them into a new record/expression that consists of: $<$***Operator***$>$***(Stack***$_1$ ***Stack***$_2$***)***. The new expression is added to the stack and it continues until there are no more tokens left, then it returns the top element in the stack.

This is essentialy how one would convert a postfix notation to an infix notation. We can say that since this method takes a postfix notation as input and returns a infix notation.

3 Theory

3.1 Formally describe the regular grammar of the lexemes in Task 1.

$V = \{c\}$
$S = \{+, -, *, /, p, d, i, c, \mathbb{Z}\}$, $\mathbb{Z}$ *is the set of integers*
$R = \{(c, \epsilon), (c, s_i\ c)\}$
$v_s = c$

Given the formal grammar above we can create the all possible lexemes from task 1. E.g. We can create:

$$"1","2","3","+"$$

with the following steps:

$$c \leftarrow (c, 1\ c)$$
$$"1", c \leftarrow (c, 2\ c)$$

,

$$"1","2", c \leftarrow (c, 3\ c)$$

$$"1","2","3", c \leftarrow (c, + c)$$
$$"1","2","3"," + ", c \leftarrow (c, \epsilon)$$
$$"1","2","3"," + "$$

3.2 Describe the grammar of the records returned by the ExpressionTree function in Task 3, using (E)BNF.

$$< expression > ::= < operator > (< expression > < expression >) \mid < number >$$
$$< operator > ::= + \mid - \mid * \mid /$$
$$< number > ::= \mathbb{Z}, \quad \mathbb{Z} \textit{ is the set of integers}$$

I am not quite sure if an empty string is allowed (my grammer does not allow it). I could have added ϵ to the $< expression >$ rule, but that would allow for something line $plus(\epsilon\ 1)$ to exist, which I am more certain is not allowed.

3.3 Which kind of grammar is the grammar you defined in step a)? Is it regular, context-free, context-sensitive, or unconstrained? What about the one from step b)?

In *Task a* the task was to "describe the regular grammer" so I hope that I used regular grammer there. We can also check that it is within the defintion of a regular grammar:

$$v ::= s\omega$$
$$v ::= s$$
$$v ::= \epsilon$$

where v and ω are any variable (i.e. non-terminals) in V and s is any symbol S. (These are right-regular grammars; in left-regular grammars the right-hand side of the first rule has inverse order.) [Taken directly from Lecture 2: Syntax, slide 68] Since it only needs these rules it is the most spesific grammar, it is a regular grammar.

In the other case I think it is a context-free grammar. This is because it needs the support of the format:

$$v ::= \gamma$$

where v is a variable in V and γ is any sequence of variables and symbols from $V \cup S$. [Taken directly from Lecture 2: Syntax, slide 70] It need atleast these rules because it has a sequence of variables and expressions in the *operator(expression expression)*-rule. This is not supported in regular grammar.

However it does not need the rules for context-sensative languages:

$$\alpha v \beta ::= \alpha \gamma \beta$$

where v is a variable in V and α, β and γ are any sequence of variables and symbols from $V \cup S$. [Taken directly from Lecture 2: Syntax, slide 71]

Therefore I conclude that it must be a context-free grammar.