TDT4165 Programming Languages Assignment 2

Ola Lømo Ellingsen

September 19, 2024

Task 1:

The **Lex** function takes a string as input and splits the string (on whitespace) into a list of tokens, this is done using the String.tokens function. In order to convert each Lexeme into a corresponding record, the **Tokenize** function was implemented. The following records were used: *operator(type:plus), operator(type:minus), operator(type:multiply), operator(type:divide), number(N) where N is any number, command(type:print), command(type:duplicate), command(type:invert) and command(type:clear).*

In order to take in postfix expressions and compute the result, the **Interpret** function was implemented. In this function we're using a recursive function *Interpreter* to check weather a token is a number, an operator or a command. An empty stack is used to recursively add the computed numbers.

If the token is a number, it will be pushed onto the stack. If the token is an operator, the top two elements of the stack will be popped and the type of operator will be applied (plus, minus, multiply or divide). The function calls itself (recursively) and the result of the operation is pushed back on the stack.

If the token is a command (print, duplicate, invert or clear), the function call itself (recursively) using a combination of functions (Push, Peek, Pop and Print) to update the stack. The function returns the stack of computed numbers after all the tokens has been interpreted.

The code is provided in the file 'task1.oz'.

Task 2:

In the function **ExpressionTree**, we take in a list of Tokens with postfix notation, and we initialize an empty stack that will be used to build the expression tree. Using the function **ExpressionTreeInternal**, we recursively iterate through the list of Tokens, and for each token we check weather it's a number or an operator. If it's a number, we push it to the stack.

If it's an operator, we pop the top two elements from the stack and apply them in the expression along with the operator. We then push this expression back to the stack. We continue this process until we have processed all tokens in the list, and the top element of the stack will be the final expression tree. We return this as the result.

The code is provided in the file 'task2.oz'.

Task 3:

a)

Each expression needs to start with at least two numbers followed by an operator. An expression (exp) can therefore be both a number, or two expressions followed by an operator.

Variables (non-terminals): V = {exp}

Symbols (terminals): S = {digit, +, -, *, /}, where digit represents a numeric value (can be a set of digits)

Rules: R = {(exp, digit), (exp, exp exp operator)} where digit is a number and operator is +, - ,* or /

Start variable: Vs = exp

b)

In the ExpressionTree, an expression can either be a number or an operator applied to two expressions.
E.g. plus(7 minus(3 9)), where *exp = op(num op(num num)).*

Thus, the result of the **ExpressionTree** function will always be an expression on this form. The valid operators in the expressions are plus, minus, multiply and divide. In this case the non-terminals will be expression, operator and number ('exp', 'op' and 'num'). The terminals will be the valid operators: 'plus', 'minus', 'multiply' and 'divide', as well as 'digit' which represents a number.

This is the grammar with (E)BNF:

<exp> ::= <num> | <op>(<exp> <exp>)
<num> ::= <digit>
<op> ::= plus | minus | multiply | divide

c) The grammar in step a) for the lexemes is **context-free** because it has recursive rules with only one no-terminal on the left-hand side and no context dependencies, so the expansion of a no-terminal is the same no matter where it's used. Similarly, the grammar in step b) for the expression tree is **also context-free**, as it follows the same structure of single and recursive rules. Both grammars is in the context-free category as they do not require complexity or dependencies found in context-sensitive or unconstrained grammars, e.g. declare-before-use restriction on variables.