TDT4165 - Programming Languages

Assignment 2: Introduction to Language Theory

Task 1

You will implement mdc: a program that interprets reverse-polish notation and applies mathematical operations on numbers. See slides from the second lecture for more information. The code from lecture 2 can supplement your understanding of mdc.

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.

Code

Link to code for task 1 here

High level description

The mdc command is split by space characther, turned into records corresponding to the mdc action, and finally run trough a set of functions that return the updated stack given the command until every command has been processed.

a)

Split by space character (32 in ASCII)

```
declare fun {Lex Input} {String.tokens Input 32} end
```

b)

Simple elseif function for returning records corresponding to lexem. Use built in Map function for converting Lexemes array.

```
declare fun {Token Lexem}
    if     Lexem == "+" then operator(type:plus)
    elseif Lexem == "-" then operator(type:minus)
    elseif Lexem == "*" then operator(type:multiply)
    elseif Lexem == "/" then operator(type:divide)
    elseif Lexem == "p" then command(print)
    elseif Lexem == "d" then command(duplicate)
    elseif Lexem == "i" then command(inverse)
    elseif Lexem == "c" then command(clear)
    else number({String.toInt Lexem}) end
end

declare fun {Tokenize Lexemes} {Map Lexemes Token} end
```

c)

1. Initial interpret function creates and empty stack and calls `InterpetRec`
2. `InterpetRec` goes trough each token and completes it until there are no more tokens remaining.
3. `ParseToken` uses case pattern matching to figure out what type of token it is and runs their corresponding job.

4. `Calc` does the mathimatical operation passed trough its Operator parameter.

```
declare fun {Calc Operator S} N1=S.1 N2=S.2.1 in
    if     Operator == plus     then N1 + N2
    elseif Operator == minus    then N1 - N2
    elseif Operator == multiply then N1 * N2
    elseif Operator == divide   then N1 / N2
    end
end

declare fun {Command Command S} ... end

declare fun {ParseToken T S}
    case T of
       number(N)        then N | S
    [] operator(type:O) then {Calc O S} | S.2.2
    [] command(C)       then {Command C S}
    end
end

declare fun {InterpretRec Token Stack} if Token==nil then Stack else
    {InterpretRec Token.2 {ParseToken Token.1 Stack}} end
end

declare fun {Interpret Tokens} {InterpretRec Tokens {List.make 0}} end
```

d) e) f) g)

The `Command` function performs the extra rules added in d, e, f, and g.

```
declare fun {Command Command S}
    if     Command == print     then {Show S} S
    elseif Command == duplicate then S.1 | S
    elseif Command == inverse   then S.1 * ~1 | S.2
    elseif Command == clear     then {List.make 0}
    end
end
```

Task 2

You will implement fun {ExpressionTree Tokens}, where Tokens is a list of tokens as defined in Task 1. The function will return a record organized as a tree, representing the expression parsed from the postfix representation of the mdc-style input string.

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

Code solution

```
\insert '/Users/jesperhustad/Documents/assignments/pl_4165/2/task1.oz'

declare fun {MakeExpression Operator S} N1=S.1 N2=S.2.1 in
    if     Operator == plus     then plus(N1 N2)
    elseif Operator == minus    then minus(N1 N2)
    elseif Operator == multiply then multiply(N1 N2)
    elseif Operator == divide   then divide(N1 N2)
    end
end

declare fun {ExpressionTreeInternal Tokens ExpressionStack} T=Tokens S=ExpressionStack in

    % all tokens already processed
    if T==nil then S.1 else

    % add numbers to expression stack
    case T.1 of number(N) then {ExpressionTreeInternal T.2  N|S}

    % if operator: make corresponding expression and remove 2 from stack
    [] operator(type:O) then {ExpressionTreeInternal T.2 {MakeExpression O S}|S.2.2}

    end end
end

declare fun {ExpressionTree Tokens} {ExpressionTreeInternal Tokens {List.make 0}} end
```

High level description

The solution follows closely the algorithm given in the assignment:

> •When you encounter a non-operator token in the input list, push it to the expression stack.
> •When you encounter an operator token in the input list, remove two expressions from the expression stack and use them as operands, in a newly built expression. Push this expression to the stack.
> •When all the input tokens have been processed, return the element at the top of the expression stack.

To make the function recursive the final check that the stack contains no more elements is performed first.

MakeExpression function simply returns an expression, inspired from the Calc function already implemented in Task1

Task 3: Theory

a) Formally describe the regular grammar of the lexemes in Task 1.

They are one character lexemes divided by single space characthers. They can be divided into three types of tokens: numbers, operators and commands. Other than the order each lexem is context-free. It can therefore be described as:
`v ::= γ`

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

The expressions tree in (Extended) Backus-Naur Form can be described as a set of grouping with parantheses. The token to the left of the grouping describes a function which is performed on the grouping.

*c) 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)?*

The grammer can be defined as context-sensetive. Because the order of the lexemes has a big effect on the output it can not be defined as context-free. The grammer in task 2 can be described as regular because (E)BNF ideas such as grouping and functions have been implemented in the grammer.