TDT4165 Programming Languages Assignment 2

Martin Hegnum Johannessen

September 12, 2024

Contents

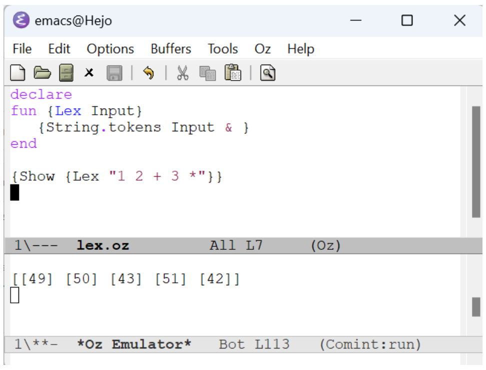
List of Figures

1 Task 1: mdc

The MDC is a postfix calculator that interprets an input expression in reverse Polish notation (RPN) using an internal stack. The process begins by splitting the input string into tokens, which are then converted into records representing numbers, operators, or commands. A recursive function processes each token: numbers are pushed onto the stack, operators perform calculations using the top two numbers from the stack, and commands modify the stack (e.g., print, duplicate, invert, or clear). The final state of the stack is returned as the output after all tokens are processed.

1.1 Lex Input

The Lex function takes a string as input and splits it into individual tokens (lexemes), assuming they are separated by spaces. This is the first stage of the MDC process where the input string is prepared for further processing.



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

{Show {Lex "1 2 + 3 *"}}
```

```
1\---  lex.oz          All L7      (Oz)
```

```
[[49] [50] [43] [51] [42]]
```

```
1\**-  *Oz Emulator*   Bot L113    (Comint:run)
```

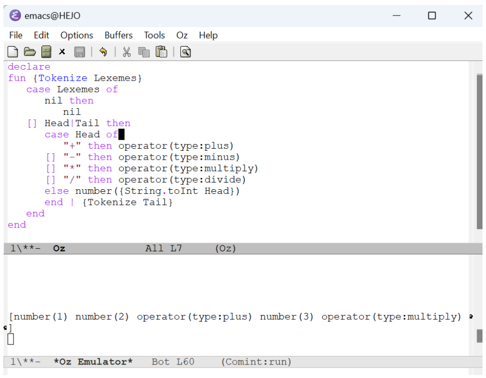
Figure 1: lex_input

1.2 Tokenize Lexemes

[h] The Tokenize function processes the list of lexemes (tokens) and converts them into structured records, such as number(N) for numbers and operator(type:plus) for the plus operator. It classifies the tokens into either numbers or operations/commands.

1.3 Interpret Tokens

The Interpret function takes the tokenized list from the previous stage and processes it. It modifies an internal stack based on the token types and operators, performing calculations where necessary.



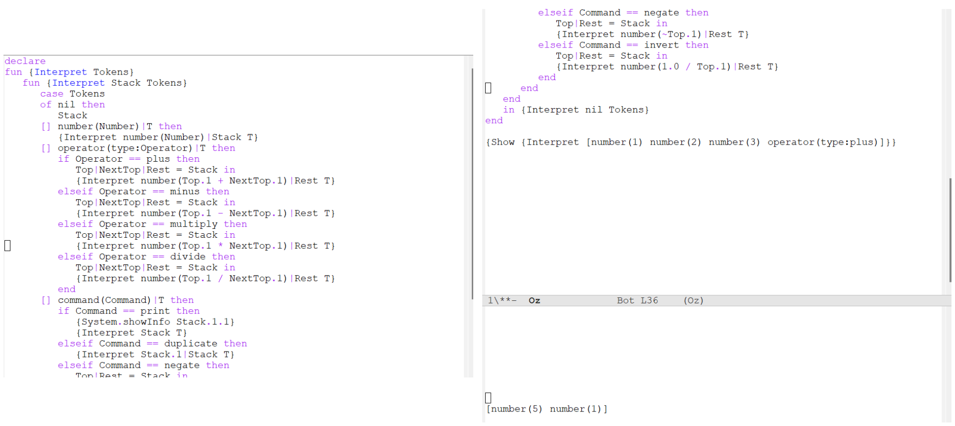
```
declare
fun {Tokenize Lexemes}
   case Lexemes of
      nil then
         nil
   [] Head|Tail then
      case Head of
         "+" then operator(type:plus)
      [] "-" then operator(type:minus)
      [] "*" then operator(type:multiply)
      [] "/" then operator(type:divide)
      else number({String.toInt Head})
      end | {Tokenize Tail}
   end
end
```

```
1\**-  Oz               All L7      (Oz)
```

```
[number(1) number(2) operator(type:plus) number(3) operator(type:multiply)
]
```

```
1\**-  *Oz Emulator*    Bot L60     (Comint:run)
```

Figure 2: tokenize_lexemes



```
declare
fun {Interpret Tokens}
   fun {Interpret Stack Tokens}
      case Tokens
      of nil then
         Stack
      [] number(Number)|T then
         {Interpret number(Number)|Stack T}
      [] operator(type:Operator)|T then
         if Operator == plus then
            Top|NextTop|Rest = Stack in
            {Interpret number(Top.1 + NextTop.1)|Rest T}
         elseif Operator == minus then
            Top|NextTop|Rest = Stack in
            {Interpret number(Top.1 - NextTop.1)|Rest T}
         elseif Operator == multiply then
            Top|NextTop|Rest = Stack in
            {Interpret number(Top.1 * NextTop.1)|Rest T}
         elseif Operator == divide then
            Top|NextTop|Rest = Stack in
            {Interpret number(Top.1 / NextTop.1)|Rest T}
         end
      [] command(Command)|T then
         if Command == print then
            {System.showInfo Stack.1.1}
            {Interpret Stack T}
         elseif Command == duplicate then
            {Interpret Stack.1|Stack T}
         elseif Command == negate then
            Top|Rest = Stack in
```

```
         elseif Command == negate then
            Top|Rest = Stack in
            {Interpret number(~Top.1)|Rest T}
         elseif Command == invert then
            Top|Rest = Stack in
            {Interpret number(1.0 / Top.1)|Rest T}
         end
      end
   end
   in {Interpret nil Tokens}
end

{Show {Interpret [number(1) number(2) number(3) operator(type:plus)]}}
```

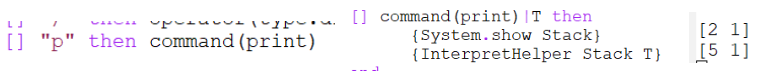
```
1\**-  Oz               Bot L36     (Oz)
```

```
[number(5) number(1)]
```

Figure 3: interpret_tokens

1.4 Add matching rule p

The p command prints the current state of the stack without modifying it. This is implemented as a special command in the interpret function.



```
[] "p" then command(print)
```

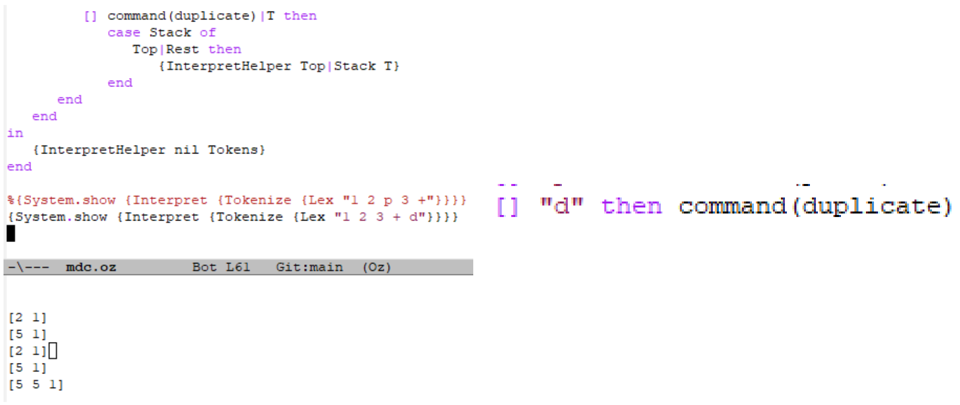
```
[] command(print)|T then
      {System.show Stack}
      {InterpretHelper Stack T}
```

```
[2 1]
[5 1]
```

Figure 4: Matching rule print

1.5 Add matching rule d

The d command duplicates the top element of the stack. This means that the topmost number is pushed again onto the stack.



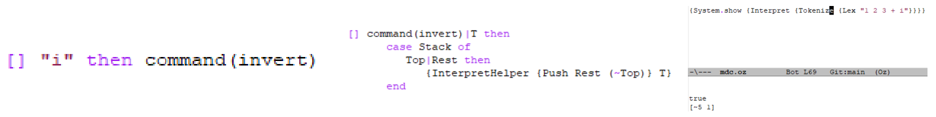
```
        [] command(duplicate)|T then
           case Stack of
              Top|Rest then
                 {InterpretHelper Top|Stack T}
           end
        end
     end
in
   {InterpretHelper nil Tokens}
end

%{System.show {Interpret {Tokenize {Lex "1 2 p 3 +"}}}}
{System.show {Interpret {Tokenize {Lex "1 2 3 + d"}}}}

-\---  mdc.oz          Bot L61    Git:main   (Oz)

[2 1]
[5 1]
[2 1]
[5 1]
[5 5 1]
```

```
[] "d" then command(duplicate)
```

Figure 5: Matching rule duplicate

1.6 Add matching rule i

The i command inverts the sign of the top element of the stack. It replaces the top value with its negative. In other words, the top value is negated (Top) and placed back on the stack.



```
[] "i" then command(invert)
```

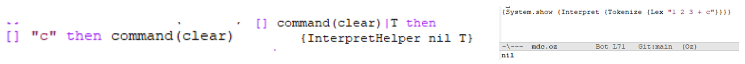
```
[] command(invert)|T then
   case Stack of
      Top|Rest then
         {InterpretHelper {Push Rest (~Top)} T}
   end
```

```
{System.show {Interpret {Tokenize {Lex "1 2 3 + i"}}}}

-\---  mdc.oz          Bot L69    Git:main   (Oz)

true
[~5 1]
```

Figure 6: Matching rule inverse

1.7 Add matching rule c

The c command clears the entire stack, leaving it empty. It does so by setting the stack to nil (empty), then interpretation continues with the remaining tokens.



```
[] "c" then command(clear)
```

```
[] command(clear)|T then
   {InterpretHelper nil T}
```

```
{System.show {Interpret {Tokenize {Lex "1 2 3 + c"}}}}

-\---  mdc.oz          Bot L71    Git:main   (Oz)
nil
```

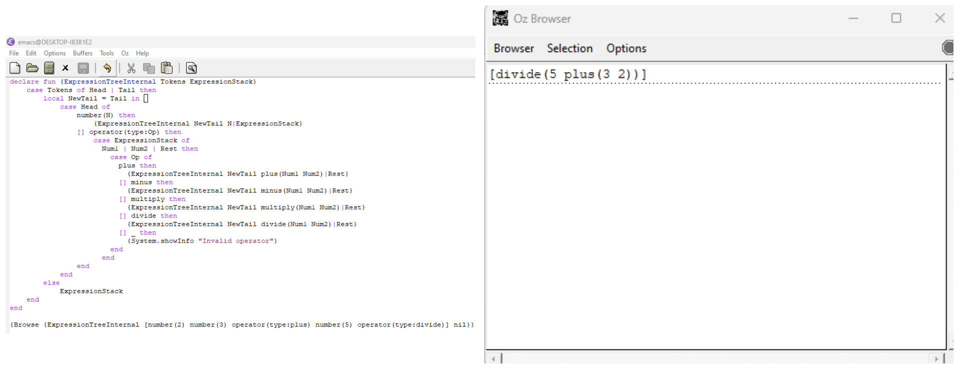
Figure 7: Matching rule clear

2 Task 2: Convert postfix notation into an expression tree

In postfix notation, operators follow their operands. To convert a postfix expression into an expression tree, we use a stack to store intermediate results while processing tokens left to right. Numbers are pushed onto the stack. Then, when an operator is encountered, two elements are popped. These are then combined into a new expression, and pushed back onto the stack. Once all tokens are processed, the final stack element is the root of the expression tree.

2.1 ExpressionTreeInternal Tokens ExpressionStack

The ExpressionTreeInternal function recursively processes tokens and manages the expression stack. Numbers are pushed onto the stack, while operators pop the top two elements. These form a new node, and push the result back onto the stack. The recursion continues until all tokens are processed, leaving the final expression tree on the stack.


emacs@DESKTOP-I83B1E2
File Edit Options Buffers Tools Oz Help
Oz Browser
Browser Selection Options

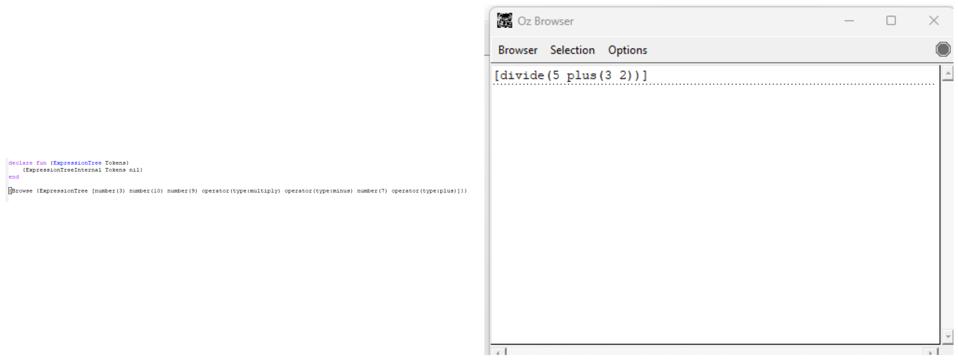
```
declare fun {ExpressionTreeInternal Tokens ExpressionStack}
    case Tokens of Head | Tail then
        local NewTail = Tail in 
            case Head of
                number(N) then
                    {ExpressionTreeInternal NewTail N|ExpressionStack}
                [] operator(type:Op) then
                    case ExpressionStack of
                      Num1 | Num2 | Rest then
                        case Op of
                          plus then
                            {ExpressionTreeInternal NewTail plus(Num1 Num2)|Rest}
                          [] minus then
                            {ExpressionTreeInternal NewTail minus(Num1 Num2)|Rest}
                          [] multiply then
                            {ExpressionTreeInternal NewTail multiply(Num1 Num2)|Rest}
                          [] divide then
                            {ExpressionTreeInternal NewTail divide(Num1 Num2)|Rest}
                          [] _ then
                            {System.showInfo "Invalid operator"}
                        end
                      end
                end
            end
        else
            ExpressionStack
    end
end

{Browse {ExpressionTreeInternal [number(2) number(3) operator(type:plus) number(5) operator(type:divide)] nil}}
```

```
[divide(5 plus(3 2))]
```

Figure 8: ExpressionTreeInternal Tokens ExpressionStack

2.2 ExpressionTree

The ExpressionTree function initializes an empty stack and calls ExpressionTreeInternal to process tokens recursively. After all tokens are processed, it returns the final expression tree.



```
declare fun {ExpressionTree Tokens}
    {ExpressionTreeInternal Tokens nil}
end

{Browse {ExpressionTree [number(3) number(10) number(9) operator(type:multiply) operator(type:minus) number(7) operator(type:plus)]}}
```

```
[divide(5 plus(3 2))]
```

Figure 9: ExpressionTree

3 Task 3: Theory

3.1 Regular grammar

The grammar for the lexemes in Task 1 can be described formally as follows:

Set of Non-Terminals:

$$V = \{c\}$$

Set of Terminals:

$$S = \{+, -, *, /, p, d, i, c, Z\}$$

Where Z represents integers.

Production Rules:

$$c \to \epsilon \quad \text{(Empty string)}$$

$$c \to s\,c \quad \text{(Where } s \text{ is any terminal from } S\text{)}$$

This grammar generates all possible lexemes in Task 1, such as numbers and operators. For example, starting from c, we can generate lexemes like "1", "2", "3", and "+" by applying the production rules.

3.2 Grammar records

The grammar for the records returned by the `ExpressionTree` function in Task 2 can be described using (E)BNF as follows:

```
<expression> ::= <operator> ( <expression> <expression> ) | <number>
<operator> ::= + | - | * | /
<number> ::= Z  (Z represents integers)
```

This grammar describes how an expression tree is constructed, with operators having two sub-expressions and numbers being the leaves of the tree.

3.3 Grammar type

The grammar in **step a** is **regular** because it follows the form of a right-regular grammar, where a non-terminal is replaced by a terminal followed by another non-terminal, or an empty string.

The grammar in **step b** is **context-free** because it allows recursion with sequences of non-terminals and terminals (e.g., an operator followed by two sub-expressions), which requires context-free grammar rules. It is not regular because it involves recursive structures like `operator(expression expression)`.