TDT4165 - Programming Languages

Assignment 2: Introduction to Language Theory

Benjamin Zubača

September 19, 2024

Task 1: mdc

a) In the function `Lex`, we utilize the built-in function `String.tokens` to extract lexemes from the input string, specifying whitespace as the delimiter for token separation.

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

b) → g) In the `Tokenize` function, we employ the `Map` function along with pattern matching to transform each lexeme into its corresponding record type based on its content. We first check each lexeme against predefined operators, and if none match, we attempt to convert the lexeme into a `number` record using `String.toInt` for the conversion.

```
fun {Tokenize Lexemes}
   {Map Lexemes
    fun {$ Lexeme}
       case Lexeme
       of "+" then operator(type:plus)
       [] "-" then operator(type:minus)
       [] "*" then operator(type:multiply)
       [] "/" then operator(type:divide)
       [] "p" then command(print)
       [] "d" then command(duplicate)
       [] "i" then command(negate)
       [] "c" then command(clear)
       else
          try number({String.toInt Lexeme})
          catch _ then raise "invalid lexeme" end
          end
       end
    end}
end
```

c) → g) In the `Interpret` function, we recursively process the list of tokens using a stack to manage operands and results, primarily through the `DoInterpret` function. Within this function, we evaluate specific commands and operators, adjusting the stack accordingly. When encountering an operator, we ensure that there are sufficient operands on the stack before performing the operation using the `Compute` function, which handles the arithmetic operations. Additional commands such as print, duplicate, negate, and clear are implemented using similar stack manipulation techniques. The `Unwrap` function simplifies the stack by extracting the values from each token, returning a list of the underlying numbers.

```
fun {Interpret Tokens}
   {Unwrap {DoInterpret nil Tokens}}
end

fun {DoInterpret Stack Tokens}
   case Tokens
   of nil then Stack
   [] Head|Tail then
      case Head
      of number(_) then {DoInterpret Head|Stack Tail}
      [] operator(type:Op) then
         if {Length Stack} < 2 then
            raise "Not enough operands for operator" end
         else
            case Stack of Right|Left|Rest then
               {DoInterpret {Compute Left Right Op}|Rest Tail}
            end
         end
      [] command(print) then
         {Show {Unwrap Stack}}
         {DoInterpret Stack Tail}
      [] command(duplicate) then
           case Stack of Top|Rest then
             {DoInterpret Top|Top|Rest Tail}
         end
      [] command(negate) then
         case Stack of number(N)|Rest then
             {DoInterpret number(~N)|Rest Tail}
         end
      [] command(clear) then {DoInterpret nil Tail}
      end
   end
end

fun {Compute Left Right Op}
   case Op
   of plus then number(Left.1 + Right.1)
   [] minus then number(Left.1 - Right.1)
   [] multiply then number(Left.1 * Right.1)
   [] divide then number(Left.1 / Right.1)
   else raise "Unknown operator" end
   end
end

fun {Unwrap List}
   {Map List fun {$ Token} Token.1 end}
end
```

A High-Level Description of the `mdc` Program

The `mdc` program starts by accepting a string input that represents a RPN expression. First, the `Lex` function is invoked to tokenize the input string, breaking it down into individual lexemes based on whitespace. This results in a list of tokens that represent either numbers or operators.

Next, the `Tokenize` function takes these lexemes and transforms them into structured records. Each token is categorized as either a numerical value (encapsulated in a `number` record) or an operator (encapsulated in an `operator` record).

During the interpretation phase, the `Interpret` function processes the list of token records recursively using a stack data structure. The core of this implementation lies within the `DoInterpret` function: As tokens are evaluated, numbers are pushed onto the stack, while operators initiate arithmetic operations. When an operator is encountered, the program checks for sufficient operands on the stack, executes the specified operation through the `Compute` function, and updates the stack with the result.

Additionally, the program includes commands for managing the stack, such as printing its current contents, duplicating the top element, negating the top element, and clearing the entire stack.

Once all tokens have been processed, the remaining stack contains the final computed values, which can be displayed or further manipulated as needed.

Task 2: Convert Postfix Notation Into an Expression Tree

a) The function `ExpressionTreeInternal` processes tokens of a postfix expression recursively and builds the expression tree. It keeps track of numbers and intermediate results using an expression stack. When an operator is encountered, it pops the necessary operands from the stack, combines them into a tree node, and pushes the result back onto the stack.

```
fun {ExpressionTreeInternal Tokens ExpressionStack}
   case Tokens
   of nil then
      case ExpressionStack
      of Head|_ then Head
      else raise "invalid expression" end
      end
   [] Head|Tail then
      case Head
      of number(N) then {ExpressionTreeInternal Tail N|ExpressionStack}
      [] operator(type:Op) then
         if {Length ExpressionStack} < 2 then
             raise "not enough operands for operator" end
         else
            case ExpressionStack of Left|Right|Rest then
               case Op
               of plus then {ExpressionTreeInternal Tail plus(Left Right)|Rest}
               [] minus then {ExpressionTreeInternal Tail minus(Left Right)|Rest}
               [] multiply then {ExpressionTreeInternal Tail multiply(Left
                   Right)|Rest}
               [] divide then {ExpressionTreeInternal Tail divide(Left Right)|Rest}
               else raise "unknown operator" end
               end
            end
         end
      else raise "invalid token" end
      end
   end
end
```

b) The function `ExpressionTree` serves as a wrapper for `ExpressionTreeInternal`, initializing the function with an empty stack. It returns the root of the fully constructed expression tree once all tokens have been processed.

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

A High-Level Description of the Postfix to Expression Tree Conversion

The process of converting postfix notation into an expression tree follows a stack-based approach similar to evaluating postfix expressions. The algorithm processes each token in the input:

- Numbers are pushed onto the stack as leaf nodes.
- Operators pop two nodes from the stack (representing operands), create a new tree node where the operator is the parent and the two operands are its children, and then push the resulting subtree back onto the stack.

This continues until all tokens are processed, leaving a single node on the stack, which represents the root of the expression tree.

Task 3: Theory

a) The grammar of the lexemes from Task 1 can be formalized as $\Gamma = (V, S, R, v_s)$, where

$$V = \{\varepsilon, \lambda, \nu, \omega, \kappa\}$$

$$S = \{+, -, *, /, p, d, i, c, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \sqcup\}$$

$$\begin{aligned} R = \{&(\varepsilon, \lambda), (\varepsilon, \lambda \sqcup \varepsilon),\\ &(\lambda, \nu), (\lambda, \omega), (\lambda, \kappa),\\ &(\nu, 0), (\nu, 1), (\nu, 2), (\nu, 3), (\nu, 4), (\nu, 5), (\nu, 6), (\nu, 7), (\nu, 8), (\nu, 9), (\nu, \nu\nu),\\ &(\omega, +), (\omega, -), (\omega, *), (\omega, /),\\ &(\kappa, p), (\kappa, d), (\kappa, i), (\kappa, c)\} \end{aligned}$$

$$v_s = \varepsilon.$$

Here, $v_s = \varepsilon$ represents the start variable, λ represents a single lexeme, ν represents an integer, ω represents an operator, κ represents a command, and $\sqcup$ represents a whitespace symbol.

This grammar captures the structure of the lexemes in Task 1. An expression can be a single lexeme or multiple lexemes separated by whitespace. Integers can be single digits or multiple digits (represented by the recursive production $\nu \to \nu\nu$). Operators are the four basic arithmetic operations. Commands are single-letter instructions for print, duplicate, negate, and clear. Whitespace ($\sqcup$) is explicitly included to separate lexemes in multi-lexeme expressions.

b) The grammar of the records returned by the `ExpressionTree` function in Task 2 can be formalized as the following:

⟨*expr*⟩ ::= `integer`
| ⟨*oper*⟩

⟨*oper*⟩ ::= ⟨*plus*⟩
| ⟨*minus*⟩
| ⟨*multiply*⟩
| ⟨*divide*⟩

⟨*plus*⟩ ::= `plus(`⟨*expr*⟩, ⟨*expr*⟩`)`

⟨*minus*⟩ ::= `minus(`⟨*expr*⟩, ⟨*expr*⟩`)`

⟨*multiply*⟩ ::= `multiply(`⟨*expr*⟩, ⟨*expr*⟩`)`

⟨*divide*⟩ ::= `divide(`⟨*expr*⟩, ⟨*expr*⟩`)`

c) The grammar in step a) can be classified as context-free because each production rule contains a single non-terminal on the left-hand side, and the right-hand side consists of a sequence of terminals and/or non-terminals. For instance, the recursive production $\nu \to \nu\nu$ enables the generation of multi-digits integers. In contrast, this grammar cannot be considered regular, as regular grammars restrict rules to those where the left-hand non-terminal is replaced by at most one terminal and possibly one-terminal. The recursion in $\nu \to \nu\nu$ violates this constraint, making the grammar non-regular.

Similarly, the grammar in part (b) is also context-free, as it describes tree-like structures for mathematical expressions. For example,

$\langle plus \rangle$ ::= `plus(`$\langle expr \rangle$, $\langle expr \rangle$`).`

This rule follows the context-free pattern, with a single non-terminal on the left-hand side and a combination of terminals and/or non-terminals on the right-hand side. Like the grammar in part a), this one cannot be classified as regular either, because it involves non-linear structures and recursive patterns. Regular grammars, being more restrictive, only allow productions with limited right-hand sides, typically a single terminal followed by at most one non-terminal.