Text answers for exercise 2

High-level description of code

mdc

Firstly, the `Lex`-function takes in a space-separated string of numbers, commands or operators.All it does is splitting the string on the spaces, and returning a list containing the lexemes. Then, the `Tokenize`-function takes in a list of lexemes and converts them to tokens, which are represented as records. It returns a list of tokens. It makes use of a implementation of the `Map`-function. For each lexeme, it matches it (with pattern matching) to a set of vocabulary and returns a record representing this lexeme. The `Map`-function returns a list. `Interpret` is the interpreter and calculator. It takes in a list of tokens, and uses a helper-function, `NextInStack`, to interpret and calculate. `NextInStack` is a recursive function that returns the stack created by the list of tokens. The base case of the recursion is when Tokens are `nil`, in which case the function returns the stack. Otherwise, it uses pattern matching to decide whether we have a number, operator or command. Depending on what the token is, a correct action is taken. For example, the operators pop the two first elements of the stack list, performs the corresponding operation, and pushes it back to the stack. In these steps, the function "consumes" a token, so the arguments for the recursion call is the (modified) stack and a popped version of the tokens. The commands only modify the tokens (consumes the token).

`ExpressionTree`

The conversion from a list of tokens to a expression tree is quite simple. `ExpressionTree` also uses a helper function, `ExpressionTreeInternal` to modify the expression stack. `ExpressionTreeInternal` works in the same way as `NextInStack`, with a few tweeks. It is also recursive, and the base case is still when tokens are nil. In the base case it returns the first element of the stack, which will be the expression tree for the expression. The function also consumes tokens. When it encounters a number, it will just push the number onto the stack. When it encounters a operator, it creates a record with the label `{operator_name}`. The record values are the top two elements in the expression stack list. The top two elements are popped from the stack, and the resulting operation-record is pushed onto the stack. Since elements on the stack are consumed and combined into a new element, as long as we have a valid postfix notation (and expression), the resulting stack at the base case will only be the expression tree, and it's therefore safe to just peek the top element of the stack.

Theory questions

a

Formal description of the regular grammer (the rules are regular):

$$ \begin{align*} V &= {v_1} \ S &= {number(n), operation(o), command(c)} \ R &= {\ &(v_1, \epsilon) \ &(v_1, nv_1) \ &(v_1, ov_1) \ &(v_1, cv_1) \ }\ v_s &= v_1 \end{align*} $$ where $$ \begin{align*} number &= {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}\ operation &= {+, - , , /}\ *command &= {p, d, i, c}* *end{align*}* $$

b

$$ \begin{align*} V &= {exp} \ S &= {op, num} \ R &= {\ &\langle exp \rangle ::= op(\langle exp \rangle \langle exp \rangle) \ &\langle exp \rangle ::= num\ }\ v_s &= exp \end{align*} $$

c

The grammar in a) is a regular grammar, since every rule is on the form: $$ \begin{align*} v &::= sw \ v &::= s \ v &::= \epsilon \end{align*} $$

The grammar in b) is context-free, since every rule is on the form: $$ v ::= \gamma $$