Sander Lindberg, TDT4165, Assignment 2

Task 2 - High level description of my MDC

My function “Lex” splits the input-string on spaces with the oz function {String.tokens String}, so that I get a list. I.E [“1” “2” “+” “3” “*”]

The function Tokenize checks if the head of the Lexemes list, generated by Lex, is a number or an operator, wraps number() or operator(type:) around the head. Then it calls itself recursively while adding the head, with either number() or operator(), to a list.

The function Interpret iterates over the Tokens list with the function Iterate. First time Iterate runs, the stack is empty. It checks if the head of Tokens is number(N), operator(type:operator) or command(command). If it’s a number, it converts number(N) to a float and puts it in the stack, then calls itself recursively with Tokens’s tail and the stack. If it encounters an operator, it pops the two first elements of the stack, does the operation, puts it back into the stack and calls itself recursively. If it encounters a command(command) it does the appropriate operation on the stack (print, duplicate, flip and 1/head). If we encounter a token that is not valid, such as “a”, we simply skip it and call Tokenize recursively with Tokens’s tail.

Task 3

InfixInternal starts with an empty ExpressionStack. It checks if the head of the Tokens list is a number(N) or an operator(operator). If it is a number, it adds it to the stack and calls itself recursively. (The first time, when the stack is empty it creates the stack). If we encounter an operator, we pop the two first elements of the stack and places the operator between them (as a string) and encloses it with parentheses. Then push it back into the stack. Then the function calls itself recursively with the “new” stack and does the same thing until the Tokens list is empty.

Infix simply calls InfixInternal (with nil as ExpressionStack), and returns the first element in InfixInternals’s return value, which is our expression.

Task 4

a) $G = \{[+-]?([0-9]+([.][0-9]*)?|[.][0-9]+), p, d, i, \wedge, +, -, *, /\}$ Where $[+-]?([0-9]+([.][0-9]*)?|[.][0-9]+)$ is the regular expression for floats.

b) <Expression> ::== <integer>|<float>|(<expression><operator><expression>)|(<UnaryOperator><expression>)
<operator> ::== +|-|*|/
<UnaryOperator> ::== i|^|p|d

No it is not ambiguous due to the operator precedence and that it only has one parsing tree.

c) In a context-free grammar, every production has the form A -> v where in a context-sensitive grammar, every production has the form vAw -> vyw. That is – if the grammar have something else than just one nonterminal on the left hand side, it is a context-sensitive grammar, else, it’s a context-free. In my example v and w on the left hand side is the context. That’s why it’s called *context*-sensitive.

d) This happens because integers and floats are non-compatible because floats cannot be represented accurately in a computer. This error message is useful because if we are to round 2.5, we get 2 and if we are to round 1.5 we get 2. So if we have 1 + 1.5 and 1 + 2.5 and we convert each of the floats to ints, we get the same answer.