TDT4165 Assignment 2

Task 2 - High Level Description

The function "Lex" takes the program, a string, as input and splits it on white space with the built-in function {String.tokens String} and outputs lexemes in a list. E.g "1 2 3" to ["1" "2" "3"]

The function "Tokenize" takes the list of lexemes generated by the "Lex" function, and tokenizes them. It calls itself recursively to iterate trough the list. The numbers are wrapped in a number() record, operators (+,-,*,/) are transformed into an operator(type) record, and commands (p,d,i,^) are transformed into a command(type) record. E.g ["1" "2" "3" "+" "+"] to [number(1) number(2) number(3) operator(type:plus) operator(type:plus)]

The function "Interpret" takes the list of tokens from the "Tokenize" function and executes the program. It calls itself recursively to iterate through the list. Numbers are converted to floats pushed to a stack. Operators take the two numbers from the top of the stack and preforms arithmetics. The result is pushed back on top of the stack. Commands read a singe value from the top of the stack. The print command simply prints the value. The duplicate pushes the same value it read to the top of the stack. The negate command pops the top of the stack and pushes back the negated number. The invert command pops the top of the stack and pushes back the inverted number (1/n). When the token list is empty, the stack is returned. E.g [number(1) number(2) operator(type:plus)] to [number(3)]

Task 3 - High Level Description

The function "Infix" takes a list of tokens from the "Tokenize" function and converts it to infix notation. It calls itself recursively to iterate trough the list. Each number i pushed to an expression stack. The binary operators pops the two top expressions from the stack, and pushes an appropriate expression back on the stack. E.g operator = plus Top = 1 NextTop = 2 expression = "(1+2)". The unary operators similarly pops the top of the stack and pushes an expression back on the stack. When the token list is empty, the stack, now hopefully only containing one expression, is returned.

Task 4

a)

$\Gamma = (\mathcal{V}, \mathcal{S}, \mathcal{R}, \nu_s)$

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

$\mathcal{V}$ = {S, B, I, C}

$\mathcal{R}$ = { S → ϵ, S → [0-9]+ S, S → [0-9]+ C, S → [0-9]+ B, C → [pdi^]{1} S, B → [+-*/]{1} S, }

ν_s = S

b)

```
<expression> ::== <int>|<float>|(<expression><binaryOperator><expression>)|<unaryOperator>(<expression>)
<binaryOperator> ::== +|-|*|/
<unaryOperator> ::== -|1/
<int> ::== <digit>|<digit><int>
<float> ::== <int>.<int>
<digit> ::== 0|1|2|3|4|5|6|7|8|9
```

Because of operator precedence, the grammar is not ambiguous and can only lead to one parse tree.

c)

In a context-free grammar, every production is on the form $\upsilon \rightarrow \chi$, whereas in a context-sensitive grammar, every production is on the form $\alpha\upsilon\beta \rightarrow \alpha\chi\beta$.

In other words - in a context-free grammar we only allow a single non-terminal on the left hand of a production. Whereas in a context-sensitive we can mix in more symbols on the left hand side.

E.g Lets say we have a rule for making a ⟨**digit**⟩. In a context-free grammar, this rule will not change depending on the surrounding context. In a context-sensitive grammar, this rule may be different in different contexts.

d)

I never encountered float-int problems as i convert all my numbers to floats by default. Oz does not automatically convert types, and arithmetic binary operators cannot operate on different types. This helps us ensure consistency in our programs by not allowing ambiguity. Certain programming languages, e.g. JavaScript, allows this, and it can lead to some interesting and frustrating bugs.