Exercise 4

NTNU

TDT4165 fall 2018

1 RPN calculator

1.1 splitOn

takeWhile, dropWhile Implement the functions **takeWhile, dropWhile :: (a → Bool) → [a] → [a]**, which takes a predicate and a list and returns the elements of the list or drops them as long as the predicate is satisfied, respectively.

break Implement the function **break :: (a → Bool) → [a] → ([a], [a])** which takes a predicate and a list. It splits the list on the first occurrence where the predicate is satisfied and returns a tuple with the two parts.

splitOn Implement the function **splitOn :: Eq a ⇒ a → [a] → [[a]]** that splits a list on a given element and returns a list of lists. It should remove duplicates of the element you split on. Using dropWhile, takeWhile and/or break, might help you get to the finish line on this one.

```
--example
Prelude> splitOn '.' "....lambda...the..ultimate..."
["lambda", "the", "ultimate"]
```

1.2 Lexer

Implement the function **lex :: String → [String]** which splits a list on the space character. **lex " Don't panic ! "** should return **["Don't", "panic", "!"]**

1.3 Tokenizer

Implement the function **tokenize :: [String] -> [Token]** which takes a list of strings and turns them into tokens. It might be helpful to create a helper function which tokenizes one element, **String -> Token** and then map over it in the tokenize function. Remember to account for the token type TokErr. TokErr

circumvents the type system in a similar way to that of null. The compiler will not let you know if you forget to check for it, but the test cases will. The tokenizer should return [TokErr] if any of the tokens are erroneous.

1.4 Interpreter

Implement a function **interpret :: [Token] → [Token]** that takes a list of tokens and interprets them. **interpret . tokenize . lex $ "3 10 9 * - 3 +"** should return [TokInt -84]
Hint: foldl could be a useful function for this task

1.5 Add operators

Add token types for # and – which duplicates an element and takes the additive inverse of a number.

```
--example
*Lib> interpret . tokenize . lex $ "3 # +"
[TokInt 6]
*Lib> interpret . tokenize . lex $ "2 -- # +"
[TokInt (-4)]
```

2 Shunting-Yard algorithm

Since humans are (usually) most proficient in infix notation, it would be useful if we could use this with our calculator. We will therefore be using the Shunting-Yard algorithm to convert infix to postfix notation.

2.1 Order of operations

Implement a function that checks if one operator has a higher precedence than another.

2.2 shuntInternal

Implement the function **shuntInternal :: [Token] → [Token] → [Token] → [Token]**. It takes three lists, where one is the input stack, the second is the output stack and the third is the operator stack. It will be called recursively and follow these rules when inspecting the head of the input stack:

- If the input stack is empty, return the output stack in the correct order
- If the element is a number, push it on top of the output stack
- If the element is an operator, check if it has a lower precedence than the top operator on the operator stack. If it does, push the operator with

higher precedence on the output stack. Then repeat this step. If not, push the operator on the operator stack.

2.3 shunt

Implement the function **shunt :: [Token] → [Token]** which calls shunt with the appropriate arguments.

3 Try it out

To try out (and demonstrate) your infix calculator, you can use the main function in app/Main.hs. It will be loaded when using **stack ghci**.

4 Theory

- Formally describe the regular grammar of the lexemes in task 2.
- Describe the grammar of the infix notation in task 3 using (E)BNF. Beware of operator precedence. Is the grammar ambiguous? Explain why it is or is not ambiguous?
- What is the difference between a context-sensitive and a context-free grammar?
- Given the grammar below, determine which of the strings are legal in the language:

```
1    <S> ::= <Z> | <X>
2    <Z> ::= z <Z> y | z <Y> y | e
3    <Y> ::= z <Y> y x | e
4    <X> ::= x <X> x | e
```

a) zzyy
b) xxzyxxx
c) xxxx
d) zzyxyx
e) zzzyxyxy
f) zzyxy
g) zxxy