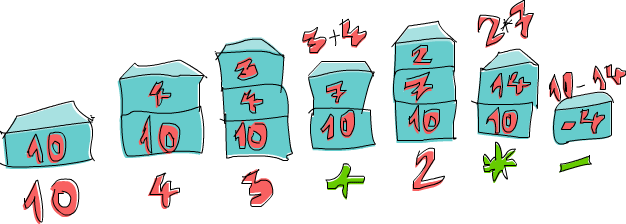
CS 5035 (Fall 2016)

### Project 8. Functionally Solving Problems

Based on chapter [10 of LYH](http://learnyouahaskell.com/functionally-solving-problems).  **(no videos)**

#### **[Reverse Polish notation calculator](http://learnyouahaskell.com/functionally-solving-problems" \l "reverse-polish-notation-calculator) (first attempt by Nov 28).**

#### **How do we evaluate a Reverse Polish notation expression such as “10 4 3 + 2 \* -” ? Well, think of a stack. You go over the expression from left to right. Every time a number is encountered, push it onto the stack. When we encounter an operator, take the two numbers that are on top of the stack (we also say that we pop them), use the operator and those two and then push the resulting number back onto the stack. When you reach the end of the expression, you should be left with a single number if the expression was well-formed and that number represents the result.**



**Let's go over the expression “10 4 3 + 2 \* -” together! First we push 10 onto the stack and the stack is now 10. The next item is 4, so we push it to the stack as well. The stack is now 10, 4. We do the same with 3 and the stack is now 10, 4, 3. And now, we encounter an operator, namely +! We pop the two top numbers from the stack (so now the stack is just 10), add those numbers together and push that result to the stack. The stack is now 10, 7. We push 2 to the stack, the stack for now is 10, 7, 2. We've encountered an operator again, so let's pop 7 and 2 off the stack, multiply them and push that result to the stack. Multiplying 7 and 2 produces a 14, so the stack we have now is 10, 14. Finally, there's a -. We pop 10 and 14 from the stack, subtract 14 from 10 and push that back. The number on the stack is now -4 and because there are no more numbers or operators in our expression, that's our result!**

This is the entire(!) code from the book.

**import** Data.List

solveRPN :: String -> Float

solveRPN = head . foldl foldingFunction [] . words

**where**  foldingFunction (x:y:ys) "\*" = (x \* y):ys

foldingFunction (x:y:ys) "+" = (x + y):ys

foldingFunction (x:y:ys) "-" = (y - x):ys

foldingFunction (x:y:ys) "/" = (y / x):ys

foldingFunction (x:y:ys) "^" = (y \*\* x):ys

foldingFunction (x:xs) "ln" = log x:xs

foldingFunction xs "sum" = [sum xs]

foldingFunction xs numberString = read numberString:xs

After loading the above code you can enter:

> solveRPN "10 4 3 + 2 \* -"

-4.0

Why are spaces required in the input? Could you modify solveRPN so that this works?

> solveRPN "10,4,3,+,2,\*,-"

-4.0

What about this?

> solveRPN "10, 4, 3, +, 2, \*, -"

-4.0

Or this?

> solveRPN "10,4, 3, +,2, \*,-"

-4.0

Don’t assume the maximum number is 2 digits.

What happens if you include a number with a decimal point in the input?

How does read know what type to return?

What happens if you leave out the **import** Data.List statement?

What happens if you leave out the solveRPN type declaration?

What about leaving out the head function?