A “Brief” Introduction to Haskell by Example

Adapted from slides by Dr. Kenneth Lambert

Side Effects

- Much of the complexity of software depends on the fact that programs change the state of the computer with time

- The number of interactions between states can grow exponentially with the number of states

- Many of these interactions, also called **side effects**, can lead to errors

Potential Solution - Drop the States

- Think of computation as a set of transformations of data values into other data values, rather than as a sequence of changes of state

- Each transformation guarantees the ***same results*** for the same data values

- You construct a set of transformations just like a proof – verification built in!

Functional Programming

- Programs consist of a set of cooperating **functions**
- Each function is **pure**, guaranteeing the **same results for the same data values**
- There is just **single assignment**, for parameters and for temporary variables
- **No side effects**!

Haskell

- Created by committee in 1987
  - Collected best features of existing functional languages
- First released to the public in 1990
- Named after Haskell Curry
- Notation more similar to mathematical notation
- Compiling tools
  - Ghc (Glasgow Haskell Compiler)
  - Ghci (Glasgow Haskell Compiler Interactive)


```
stalnakert@th001-1:~$ ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude>
```

Numbers, Arithmetic, and Comparisons

```
! !! // ^ **
* / `div` `mod` `rem` `quot`
+ -  :
/= < <= == > >= `elem`
&& ||
```

```
Prelude> 404
404
Prelude> 3.14
3.14
Prelude> 3 + 4 * 2
11
Prelude> 3 == 4
False
Prelude> 3 <= 4
True
Prelude> 3 /= 4
True
```

Function Calls

```
Prelude> sqrt 2
1.4142135623730951
Prelude> mod 3 2
1
Prelude> abs -2

<interactive>:13:1: error:
    • Non type-variable argument in the constraint: Num (a -> a)
      (Use FlexibleContexts to permit this)
    • When checking the inferred type
        it :: forall a. (Num a, Num (a -> a)) => a -> a
Prelude> abs (-2)
2
Prelude> 3 / 2
1.5
Prelude> 3 `div` 2
1
```

Interpreter Commands

Command	What It Does
`:browse[!] [[*]<mod>]`	Display the names defined by module <mod> (!: more details; *: all top-level names)
`:cd <dir>`	Change directory to <dir>
`:edit <file name>`	Edit <file name>
`:help, :?`	Display all commands
`:load <module>`	Load <module> and its dependents
`:quit`	Quit the GHCi
`:type <expression>`	Display the type of <expression>
`:!<command>`	Run a shell command

Basic Data Types

Data Type	Example Values	What It Is
`Bool`	`True`, `False`	The two truth values.
`Char`	`'a'`, `'!'`	The set of characters (on the Latin keyboard and others as well).
`Double`	3.14, 0.001	The set of double-precision floating-point numbers.
`Float`	3.14, 0.001	The set of single-precision floating-point numbers.
`Int`	0, 67, 1000000	The set of integers ranging from -2^{31} to $2^{31} - 1$.
`Integer`	Includes the `Int`s as well as greater magnitudes	A range of integers limited only by computer memory.

Check ‘em out with :type

```
Prelude> :type 'a'
'a' :: Char

Prelude> :type "Hi there"
"Hi there" :: [Char]

Prelude> :type 45
45 :: Num a => a

Prelude> :type odd
odd :: Integral a => a -> Bool

:type 3.14
3.14 :: Fractional a => a

Prelude> :type sqrt
sqrt :: Floating a => a -> a
```

First Functions

```
Prelude> square n = n * n

Prelude> square 3
9

Prelude> cube n = n * (square n)

Prelude> cube 3
27

Prelude> between n low high = low <= n && n <= high

Prelude> between 4 1 20
True
```

Notice that functions don’t need parentheses

Packaging a Module

```
{-
File: MyMath.hs
Author: Trevor Stalnaker
Purpose: provides some simple math functions
-}

module MyMath where

-- Function to square an integer
square n = n * n

-- Function to cube an integer
cube n = n * square n

-- Function to test for inclusion in a range
between n low high = low <= n && n <= high
```

Module names are capitalized

Running and Testing Module

```
stalnakert@th001-1:~$ stalnakert@th001-1:~$ nano MyMath.hs
stalnakert@th001-1:~$ stalnakert@th001-1:~$ ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :load MyMath
[1 of 1] Compiling MyMath           ( MyMath.hs, interpreted )
Ok, one module loaded.
*MyMath> cube 3
27
*MyMath> square 4
16
*MyMath> between 20 1 10
False
```

Type Inference

- Haskell will infer types from parameters
- But including type signatures is good practice

```haskell
-- Function to square an integer
square n = n * n

-- Function to cube an integer
cube n = n * square n

-- Function to test for inclusion in a range
between :: Integer -> Integer -> Integer -> Bool
between n low high = low <= n && n <= high
```

Type Signature Breakdown

between ::	Integer	->	Integer	->	Integer	->	Bool
between	n		low		high	=	low <= n && n <= high

More Examples

```
-- What's the signature?
square n = n * n

 -- What's the Signature?
cube n = n * square n
```

More Examples

```
square :: Integer -> Integer
square n = n * n

cube :: Integer -> Integer
cube n = n * square n
```

But what if we want to square integers **and** floating point numbers???

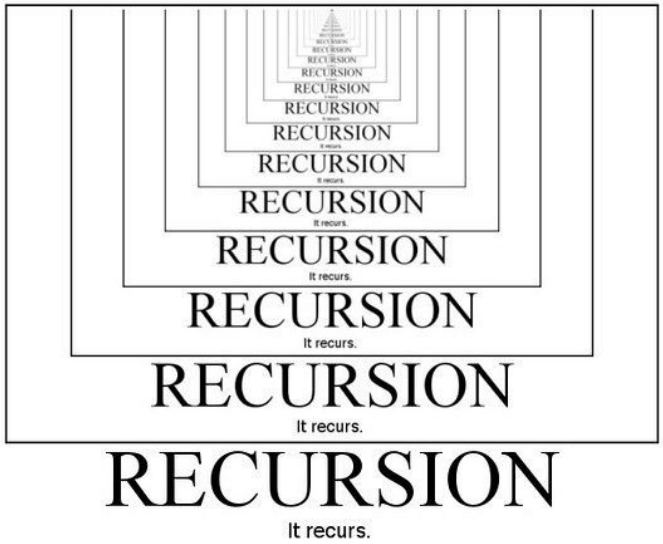
Generalizing the Types

```
{-
File: MyMath.hs
Author: Trevor Stalnaker
Purpose: provides some simple math functions
-}

module MyMath where

-- Function to square an integer
square :: Num a => a -> a
square n = n * n

-- Function to cube an integer
cube :: Num a=> a -> a
cube n = n * square n

-- Function to test for inclusion in a range
between :: Ord a => a -> a -> a -> Bool
between n low high = low <= n && n <= high
```

Recursion

Recursion

- Recursive functions have *at least* two equations or **clauses**
  - One is the base case
  - The other the recursive step
- Factorials
  - Base: n! = 1, when n =1
  - Recursive: n! = n * (n-1)!, when n >1
- To implement this, we need a way of making choices (control flow)…



Set of Function Clauses

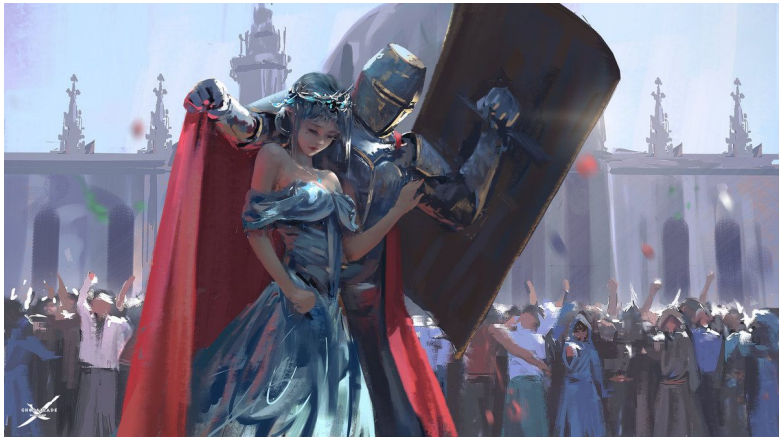
```
factorial :: Integer -> Integer
factorial 1 = 1
factorial n = n * factorial (n - 1)

fibonacci:: Integer -> Integer
fibonacci 1 = 1
fibonnaci 2 = 1
fibonacci n = fibonacci (n - 1) + fibonacci (n - 2)
```

Pattern matching selects option

Function Guards

```
<function name> <parameters>
    | <Boolean expression-1> = <expression-1>
    …
    | <Boolean expression-n> = <expression-n>
    | otherwise = <default expression>
```

```
myMax :: Ord a => a -> a -> Bool
myMax x y
    | x > y = x
    | otherwise = y

sum :: Integer -> Integer -> Integer
sum lower upper
    | lower == upper = lower
    | otherwise = lower + sum (lower + 1) upper
```

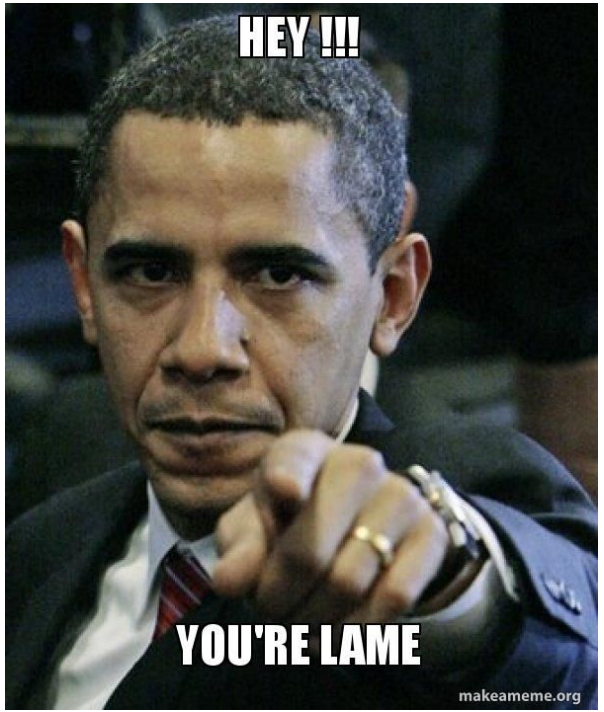


Indentation matters!

Or If You're Lame… if-else

```
<function name> <parameters> =
    if <Boolean expression-1> then
        <expression-1>
    …
    else if <Boolean expression-n> then
        <expression-n>
    else
        <default expression>
```

```
myMax :: Ord a => a -> a -> Bool
myMax x y =
    if x > y then
        x
    else
        y
```

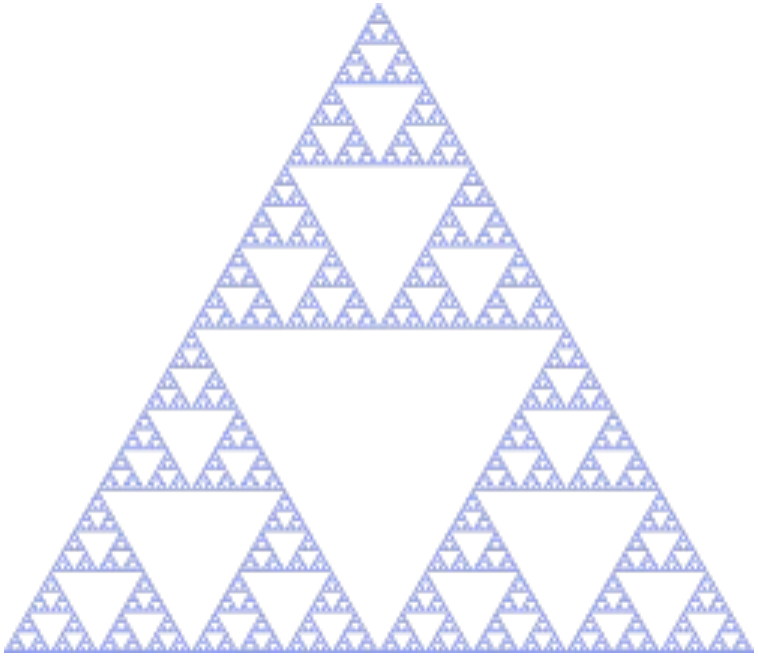


Just ***syntactic sugar*** for function guards

Lastly, The Case Expression

- Like switch statement in Java, C, etc.
- _ character is the wildcard

```
interpretCommand :: Char -> String
interpretCommand letter =
    case toUpper letter of
        'N' -> newFile
        'O' -> openFile
        'S' -> saveFile
        'Q' -> quitProgram
        _   -> handleError
```



Now Back to Recursion

How Expensive is Recursion?

- Factorial is O(n) in running time and memory
- The growth of memory is caused by cells being pushed onto the stack for each function call

```
factorial :: Integer -> Integer
factorial 1 = 1
factorial n = n * factorial (n - 1)
```

Tail Recursion

- Leaves no work to be done after recursive calls
- Compiler can translate this to a loop with a single record on the stack

```
factorial :: Integer -> Integer
factorial 1 = 1
factorial n = n * factorial (n - 1)
```

```
tailFactorial :: Integer -> Integer -> Integer
tailFactorial 1 result = result
tailFactorial n result = tailFactorial (n - 1) (n * result)
```

What's the Big Difference?

```
factorial 3 ->
    3 * factorial 2 ->
        2 * factorial 1 ->
            <- 1
        <- 2
    <- 6
```

```
tailFactorial 3 1 ->
    tailFactorial 2 3 ->
        tailFactorial 1 6 ->
            <- 6
        <- 6
    <- 6
```

What's the Big Difference?

Yuck!
Now we need an extra variable….

```
factorial 3 ->
    3 * factorial 2 ->
        2 * factorial 1 ->
            <- 1
        <- 2
    <- 6
```

```
tailFactorial 3 1 ->
    tailFactorial 2 3 ->
        tailFactorial 1 6 ->
            <- 6
        <- 6
    <- 6
```

Solution: Maintain the Interface with a Helper

```
factorial :: Integer -> Integer
factorial n = tailFactorial n 1

tailFactorial :: Integer -> Integer -> Integer
tailFactorial 1 result = result
tailFactorial n result = tailFactorial (n - 1) (result * n)
```

Even Better Solution: Hiding the Helper

- Nest the helper with the ***where*** clause
- Scope of **where** determined by indentation

```
factorial :: Integer -> Integer
factorial n = tailFactorial n 1
    where
    tailFactorial :: Integer -> Integer -> Integer
    tailFactorial 1 result = result
    tailFactorial n result = tailFactorial (n - 1) (result * n)
```

Lists

List Literals

- Like Python, but all items must be of the same type
- Can be infinite (lazy evaluation)

```
Prelude> []
[]

Prelude> [1, 2, 3]
[1,2,3]

Prelude> [1..10]
[1,2,3,4,5,6,7,8,9,10]
```

```
Prelude> take 2 [1,2,3,4]
[1,2]

Prelude> take 2 [1..4]
[1,2]

Prelude> take 2 [1..]      -- An infinite list!
[1,2]

Prelude> [1..]  -- Builds a list until you hit control-c
```

Basic List Operations

`null list`	Returns **`true`** if **`list`** is empty or **`false`** otherwise.
`head list`	Returns the first item (at position 0) in **`list`**.
`tail list`	Returns a list of the items after the first item in **`list`**.
`length list`	Returns the number of items in **`list`**.
`item : list`	Returns a list whose head is **`item`** and whose tail is **`list`**.
`list1 ++ list2`	Returns a list containing the items in **`list1`** followed by the items in **`list2`**.
`list !! index`	Returns the item at position **`index`** in **`list`**. Positions are counted from 0.

Lists are recursive

- A list is either
  - Empty (null is true)
  - An item (the head) followed by another list (the tail)

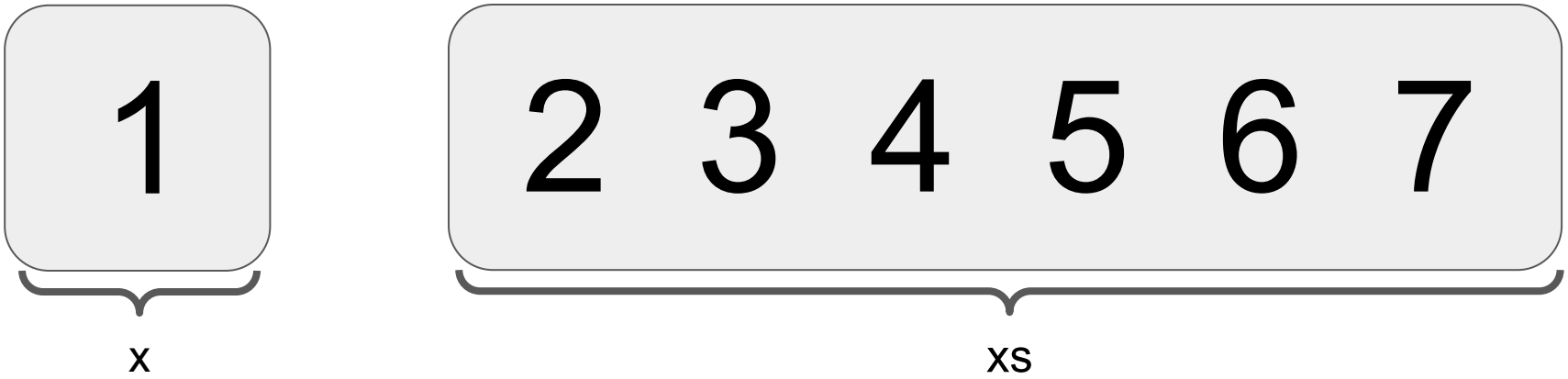
Recursive List Processing Example: Length

- The length of a list is
  - 0 if the list is empty
  - 1 + length of the tail of the list, otherwise

```
myLength :: [a] -> Int
myLength list
    | null list = 0
    | otherwise = 1 + myLength (tail list)
```

Pattern Matching

```
Prelude> 1:[2,3,4] --Use (:) as constructor
[1,2,3,4]

Prelude> (x:xs) = [1,2,3,4] -- Use (:) as selector

Prelude> x
1

Prelude> xs
[2,3,4]
```



x extracts the head, and **xs** extracts the tail

myLength with Pattern Matching

```
myLength :: [a] -> Int
myLength list
    | null list = 0
    | otherwise = 1 + myLength (tail list)
```

```
myLength :: [a] -> Int
myLength [] = 0
myLength (x:xs) = 1 + myLength xs
```

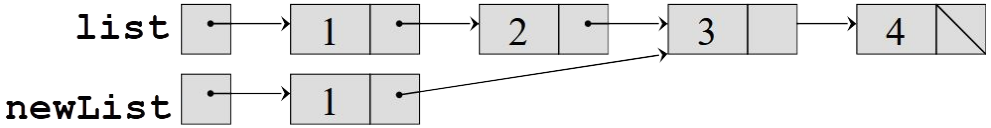
List Comprehensions

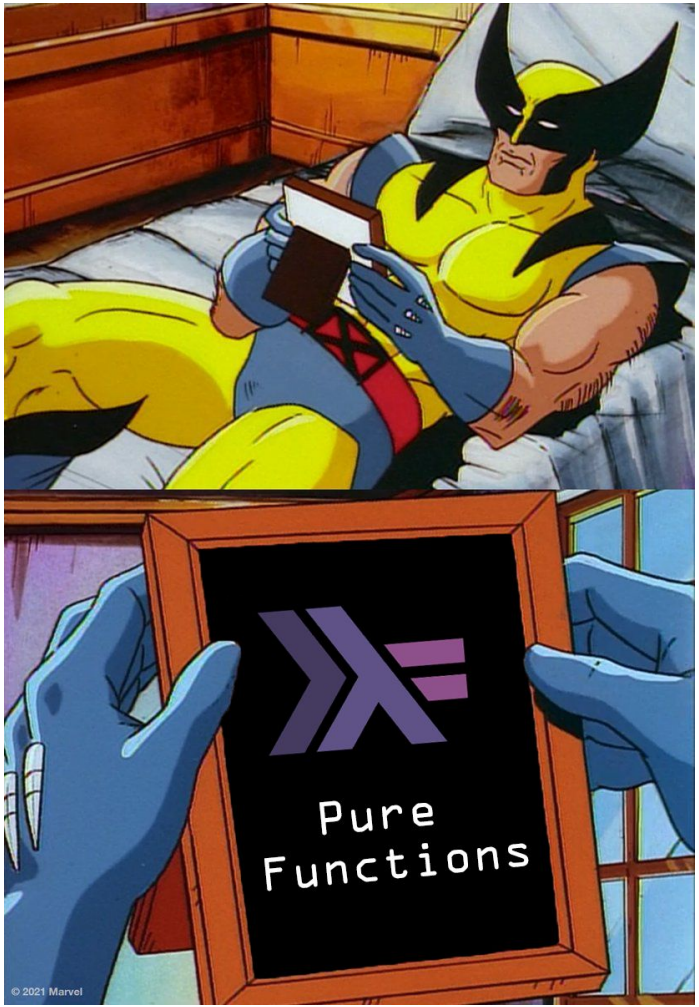
- Basic syntax:
  - [**<expression to apply to each element>** | **<element name>** <- **<list>**]
- The symbol <- suggests set membership (∈)

```
Prelude> evens = [2*x | x <- [0..10]]
Prelude> evens
[0,2,4,6,8,10,12,14,16,18,20]

Prelude> squares = [x*x | x <- [0..5]]
Prelude> squares
[0,1,4,9,16,25]
```

To Mutate or Not to Mutate

- Pure functions
  - List operations return new lists and never modify arguments
- Data structures can share memory





Strings

String Literals and The String Type

```
Prelude> "Hi there!"
"Hi there!"

Prelude> :type "Hi there!"          -- A list of characters
"Hi there!" :: [Char]

Prelude> putStrLn "Hi there!"       -- Like Python's print
Hi there!

Prelude> putStrLn "Hi\nthere!"
Hi
there!
```

Strings are just lists of characters!

Some Built-in Functions

```
Prelude> import Data.String

Prelude Data.String> words "Hi there, Ken"          -- split
["Hi","there,","Ken"]

Prelude Data.String> unwords ["Hi","there,","Ken"] -- join
"Hi there, Ken"

Prelude Data.String> lines "Hi\nthere, Ken"
["Hi","there, Ken"]
```

More operations can be found in Data.List

Tuples

Lists vs. Tuples

A list is a sequence of items of the same type

```
Prelude> numbers = [100, 34, 67]

Prelude> :type numbers
numbers :: Num a => [a]
```

A tuple is a sequence of items of any type

```
Prelude> studentInfo = ("Stanley", 19, 3.56)
Prelude> :type studentInfo
studentInfo :: (Num b, Fractional c) => ([Char], b, c)
```

Pattern Matching with Tuples

```
Prelude> studentInfo = ("Stanley", 19, 3.56)
Prelude> :type studentInfo
studentInfo :: (Num b, Fractional c) => ([Char], b, c)]
```

```
Prelude> (name, age, gpa) = studentInfo

Prelude> name
"Stanley"

Prelude> age
19

Prelude> gpa
3.56
```

Association Lists

- Like a Python dictionary, associates a set of keys with values
- The key/value pairs are tuples within a list

```
Prelude> students = [("Stanley", 3.56), ("Ann", 4.0),
                     ("Bill", 2.95)]

Prelude> :type students
students :: Fractional b => [([Char], b)]

Prelude> (name, gpa) = head students

Prelude> name
"Stanley"

Prelude> gpa
3.56
```

Built-in Functions to Build A-Lists

zip – turns a list of keys and a list of values into an association list

unzip – turns an association list into a tuple of a list of keys and a list of values

```
Prelude> :type zip
zip :: [a] -> [b] -> [(a, b)]

Prelude> :type unzip
unzip :: [(a, b)] -> ([a], [b])
```

```
*MyMath> zip ["a","b","c"] [1,2,3]
[("a",1),("b",2),("c",3)]
*MyMath>
```

Higher Order Functions

Transforming Lists

Head/tail recursion

```
roots :: Floating a => [a] -> [a]
roots [] = []
roots (x:xs) = sqrt x : roots xs
```

List comprehension

```
roots :: Floating a => [a] -> [a]
roots list = [sqrt x | x <- list]
```

Map

```
roots :: Floating a => [a] -> [a]
roots list = map sqrt list
```

Filtering

Head/tail recursion

```
allEvens :: Integral a => [a] -> [a]
allEvens [] = []
allEvens (x:xs)
    | even x = x : allEvens xs
    | otherwise = allEvens xs
```

List comprehension

```
allEvens :: Integral a => [a] -> [a]
allEvens list = [x | x  <- list, even x]
```

Map

```
allEvens :: Integral a => [a] -> [a]
allEvens list = filter even list
```

Reducing a List to a Single Item

```
Prelude> sum [1..10]
55

Prelude> product [1..10]
3628800
```

In Haskell, this is also called ***folding***

Input / Output


CAUTION
SIDE EFFECTS
AHEAD

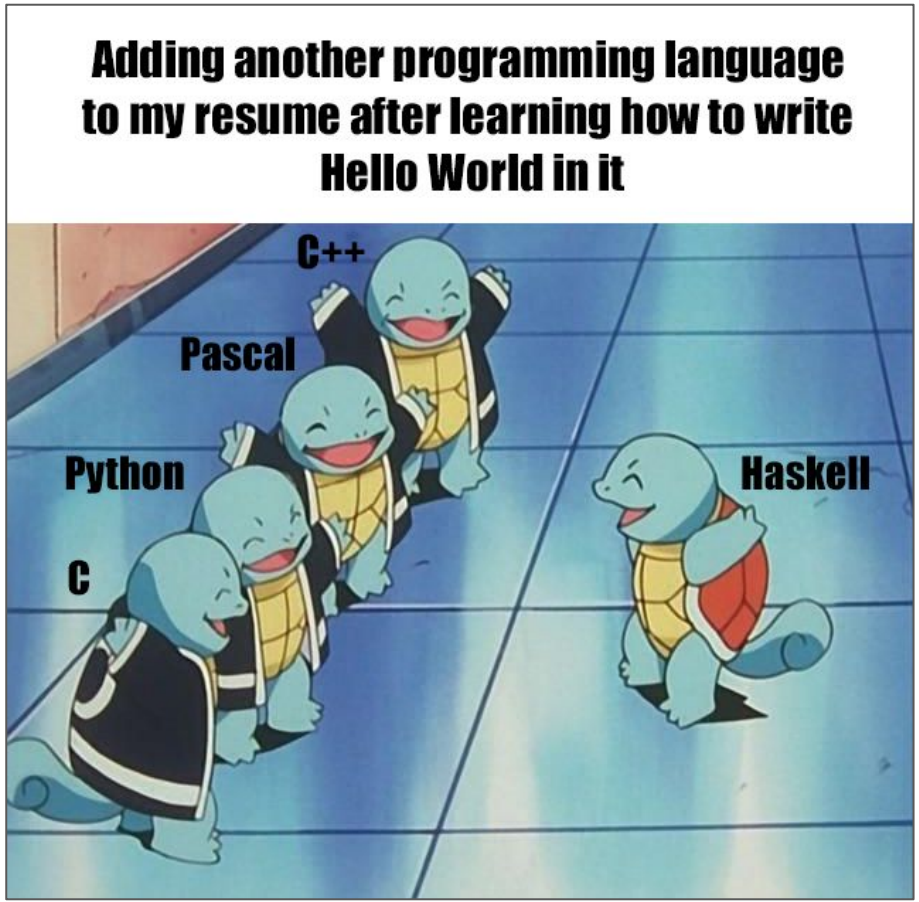
Output - putStrLn

- Takes **String** as an argument, prints it, and returns nothing!
- **IO** is a type class for types associated with I/O functions
  - **()** is the empty type
- Run only for its side effect, producing output to the terminal

```
Prelude> putStrLn "1 more week of class!"
1 more week of class!

Prelude> :type putStrLn
putStrLn :: String -> IO ()
```

Finally, Hello World!

```
module Hello where

main :: IO ()
main = putStrLn “Hello World!”
```



Input

- Takes no arguments, waits, and returns the string entered at keyboard
- Different calls can produce different results (***not pure!***)

```
Prelude> getLine   -- Input is in italic on next line
Some input text
"Some input text"
Prelude> :type getLine
getLine :: IO String
```

Questions?