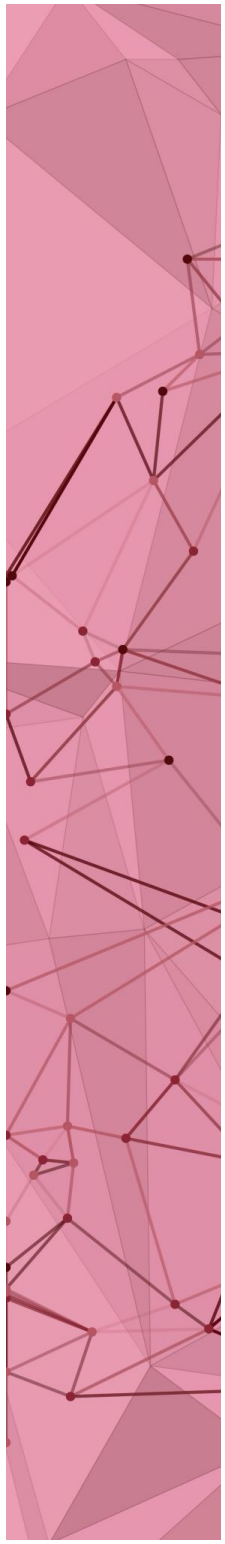
CSCI 2320
Functional Programming with Haskell

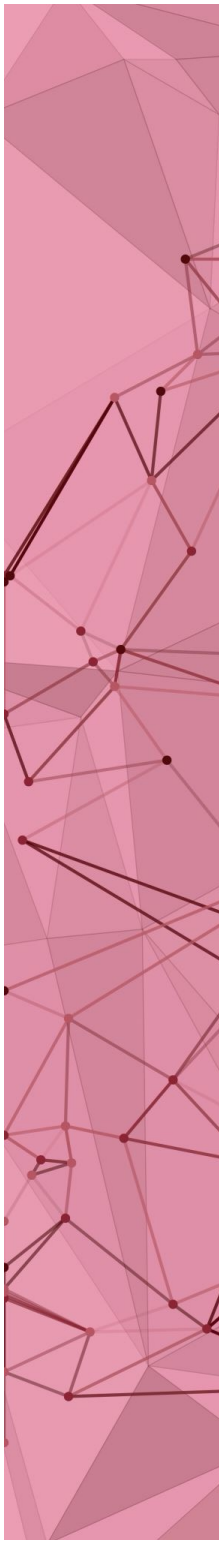
Mohammad T. Irfan

Functional Programming

- Mimic mathematical functions
- No variables in C/Java sense
- No assignment statements in C/Java sense
- How about loops?



- Recursion
- Extensive polymorphism
- Functions are first-class citizens
- Data structure: list



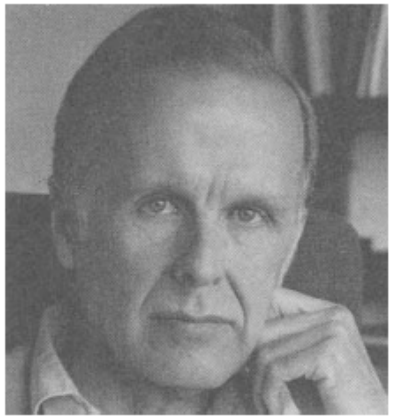
"It certainly seems like the kind of cognitive act that we are unlikely to see from any other species."

– John R. Anderson (Psychologist, CMU) on recursion

Functional Programming
“Backus’s apology for creating FORTRAN”

Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus
IBM Research Laboratory, San Jose

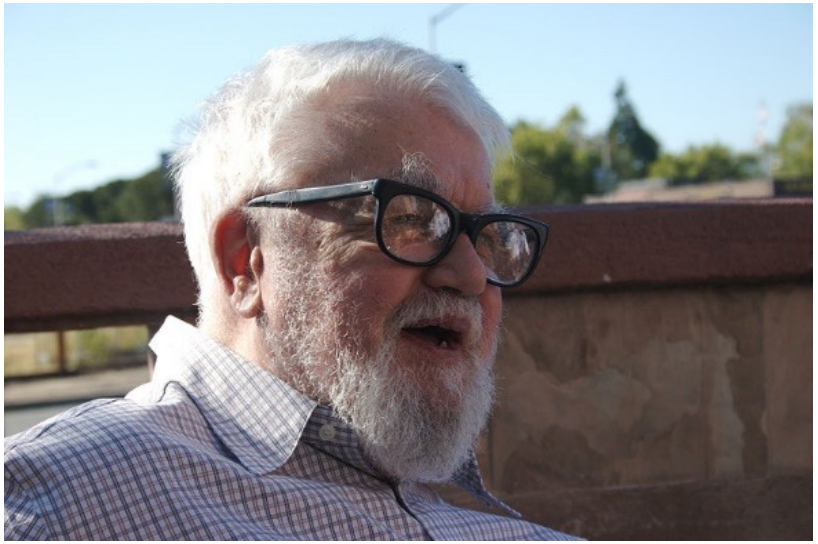


Author's address: 91 Saint Germain Ave., San Francisco, CA 94114.

Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

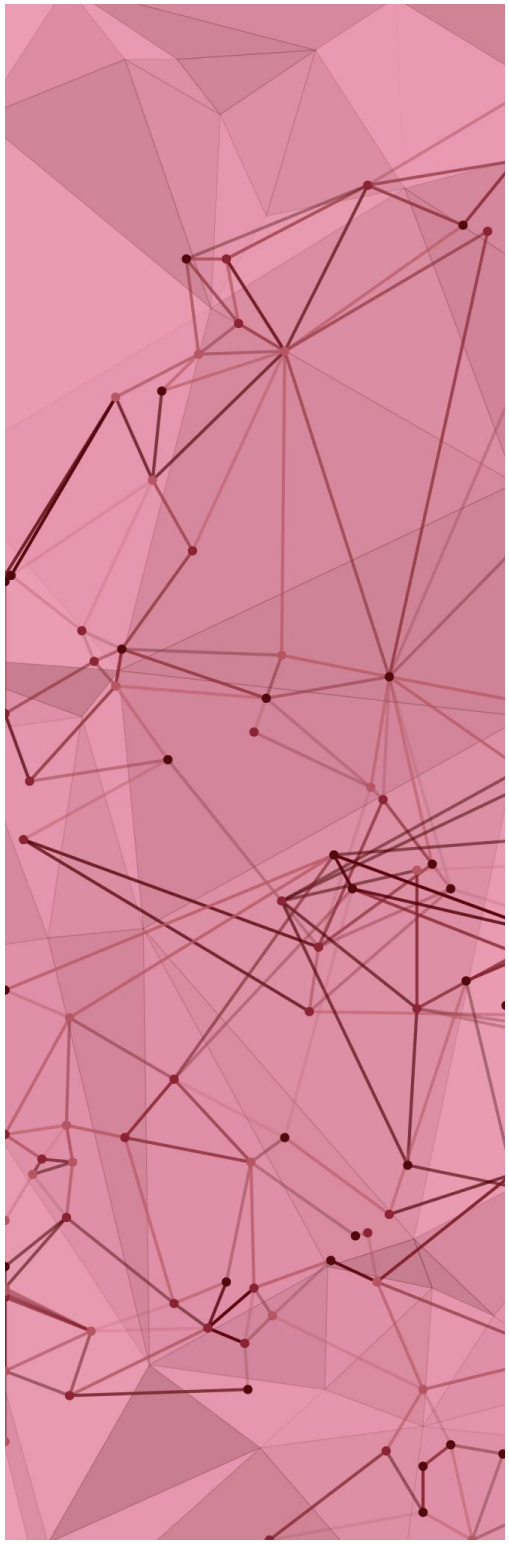
An alternative functional style of programming is founded on the use of combining forms for creating programs. Functional programs deal with structured data, are often nonrepetitive and nonrecursive, are hierarchically constructed, do not name their arguments, and do not require the complex machinery of procedure declarations to become generally applicable. Combining forms can use high level programs to build still higher level ones in a style not possible in conventional languages.

Interesting facts:
LISP and LISP Machine



John McCarthy
LISP(1960)

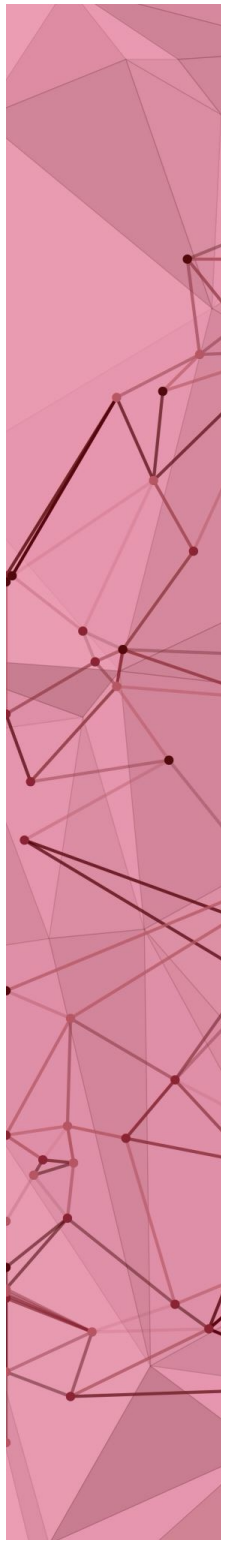
Knight machine
(1980s)



Haskell

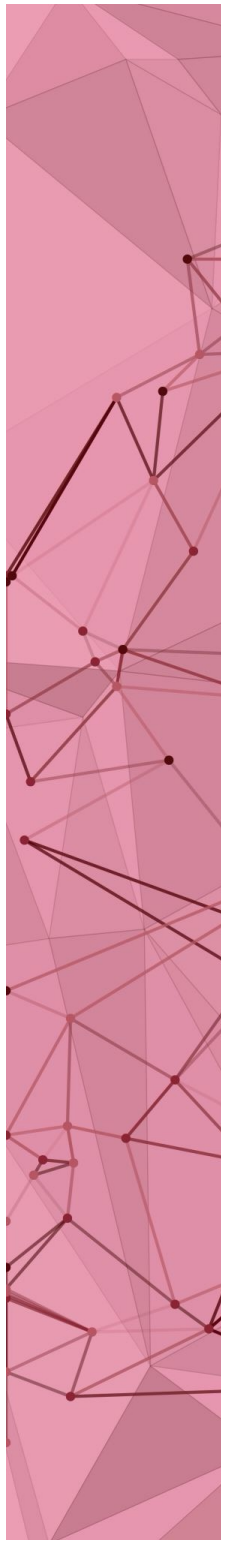
Installation

- Installation
  - https://www.haskell.org/ghcup/
- Haskell (GHCi) commands
  - http://www.haskell.org/ghc/docs/7.4.1/html/users_guide/ghci-commands.html



Learning

- Best book: Miran Lipovaca's *Learn You a Haskell for Great Good!*
  - http://learnyouahaskell.com/ (free online version)
- Gentle introduction to Haskell
  - https://www.haskell.org/tutorial/index.html
- Useful how-to page
  - http://www.haskell.org/haskellwiki/Category:How_to
- Haskell Wiki
  - http://www.haskell.org/haskellwiki/Learning_Haskell



Aaron Contorer

CEO and Founder, FP Complete; former executive with Microsoft Corporation

Haskell, the Language Most Likely to Change the Way you Think About Programming

Posted: 11/08/2013 4:28 pm

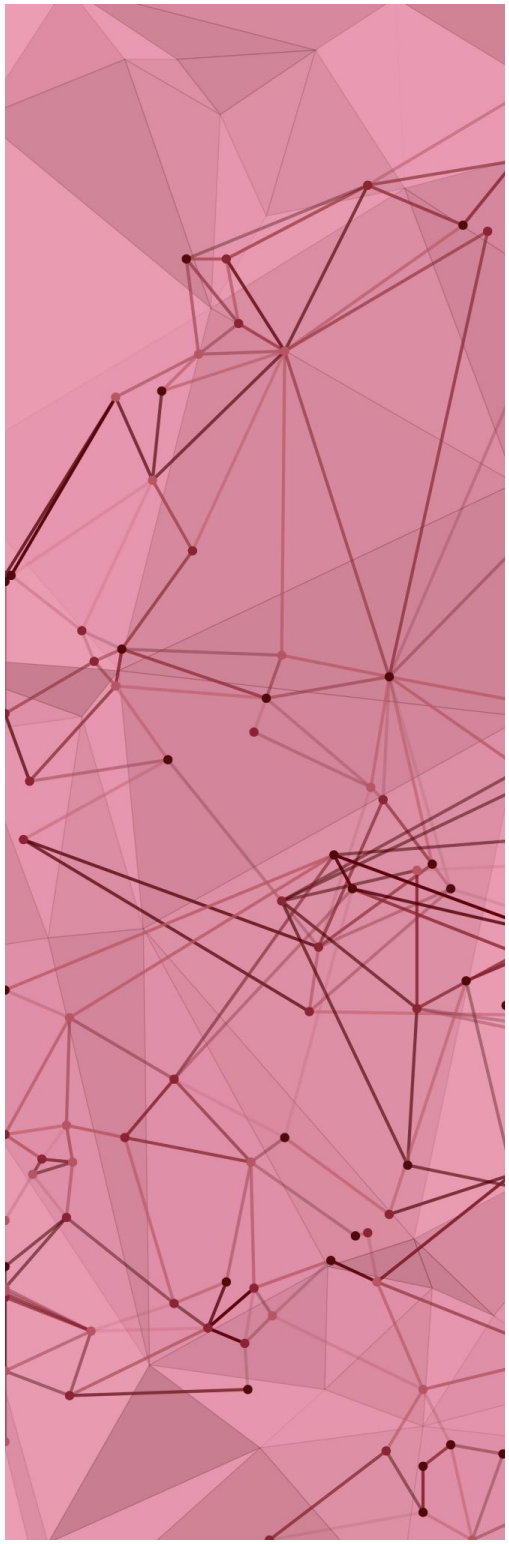
https://www.huffingtonpost.com/aaroncontorer/haskell-the-language-most_b_4242119.html

Read more > Huffpost Code News

When asked to rank programming languages based on their strengths and weaknesses, developers ranked Haskell number one for the following statements:

- "Learning this language significantly changed how I use other languages."
- "Learning this language improved my ability as a programmer."

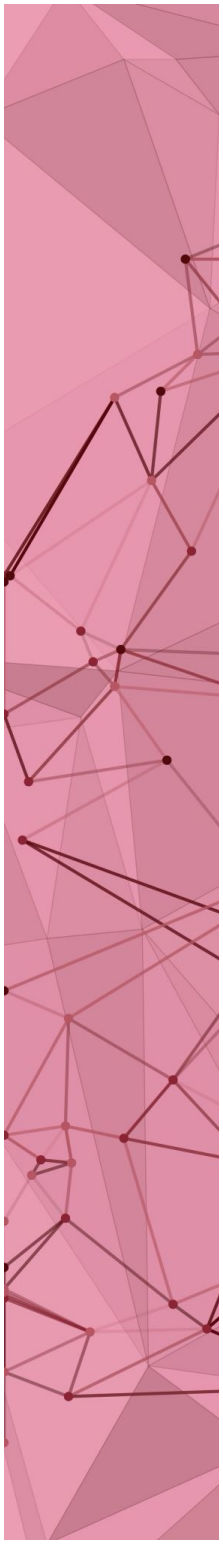
Haskell is highly regarded for its ability to transform the way developers think about programming. Many developers have told us they never really saw the holes in the imperative languages they were using until they started using Haskell. After learning Haskell, they feel more confident their code will work correctly and will have longevity. No other language provides the same perspective altering experience learners of Haskell espouse. Recently the Editor-and-Chief of Dr. Dobb's Journal, Andrew Binstock tweeted "I've noticed several times when someone says 'X really changed the way I think about programming,' frequently X=Haskell."



Coding in Haskell

Elementary functions

- Open your Haskell program folder on VS Code
- Make a *source.hs* file and write the following functions
  - doubleMe x = x + x
  - addSquares x y = x*x + y*y
- Execute this command in terminal: ghci
- Load the .hs file
  - :load source.hs (or, :l source.hs)
- Use your functions
  - addSquares 5 10
- If you change the .hs file => Execute *:r* to reload



if-then-else

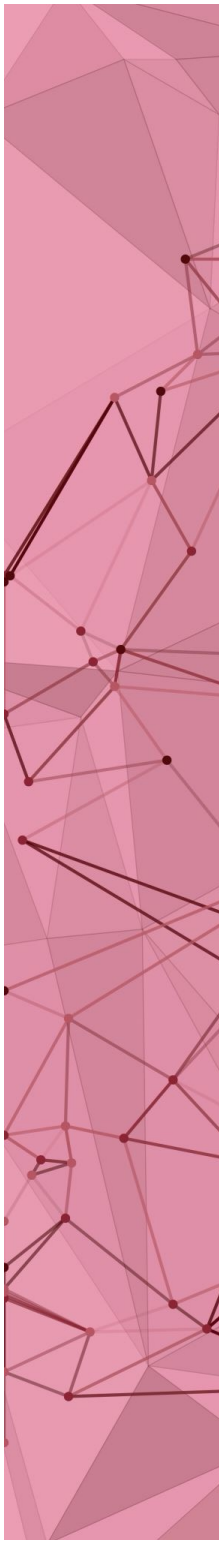
- Indentation is important

```
if boolean
    then expr1
    else expr2
```

- Same line is fine

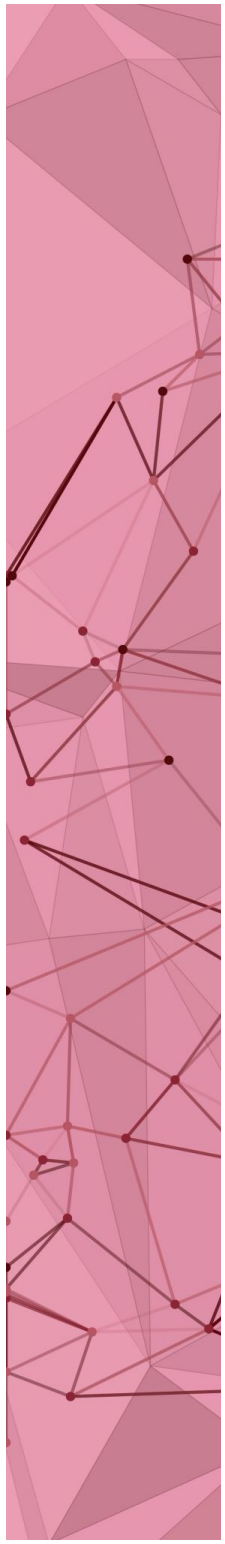
```
if boolean then expr1 else expr2
```

- else is a must! Following doesn't make sense:
  - `let x = if a then b`
- Nested if? Yes!
- Better alternative to if-then-else
  - "guard"



Problem: calculate factorial of n
Version 1 – note indentation

```
factorialV1 n =
    if n == 0
        then 1
        else if n > 0
                then n * factorialV1(n-1)
                else 0
```



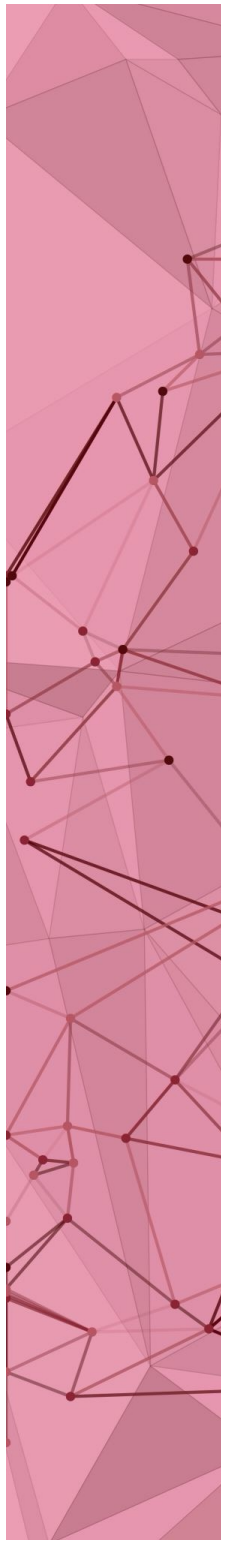
Factorial
Version 2 – using guard

```
factorial n
          |    n == 0 = 1
          |    n > 0    = n * factorial (n-1)
                                -- why not factorial n – 1 ?
```

Guard

Comment

Try this: `factorial 100`

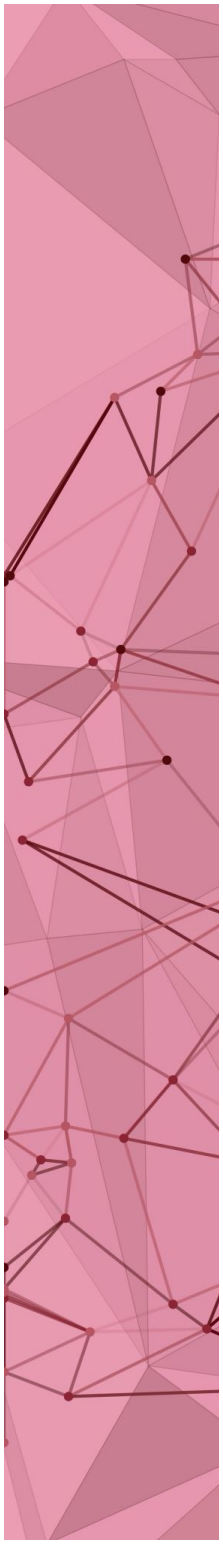


List

- `Want: evens = [0, 2, 4, 6, 8, 10]`
- In terminal (ghci)
  - `let evens = [0, 2 .. 10]`
  - `let evens = [2*x | x <- [0..5] ]`
- Infinite list
  - `let allEvens = [0, 2 ..]`

let defines functions within terminal or within another functions

Are these assignment statements?



```
let evens = [2*x | x <- [0..5] ]
```

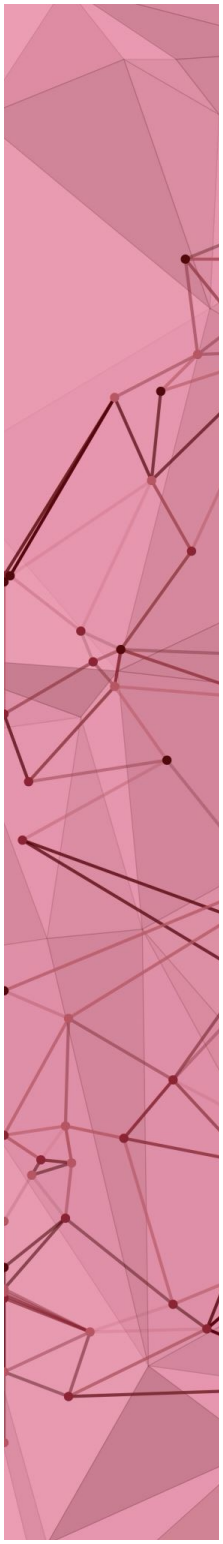
Is it a loop? `x <- [0..5]`

No; recursion is key!

```
list_gen  input_list = [2*x | x <- input_list]
```

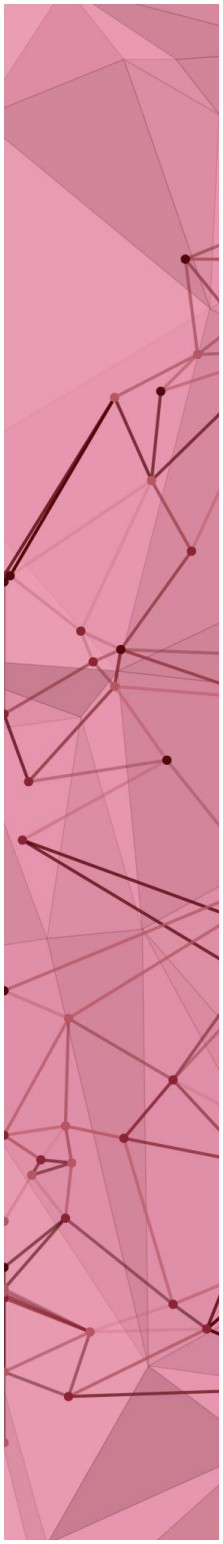
is defined as

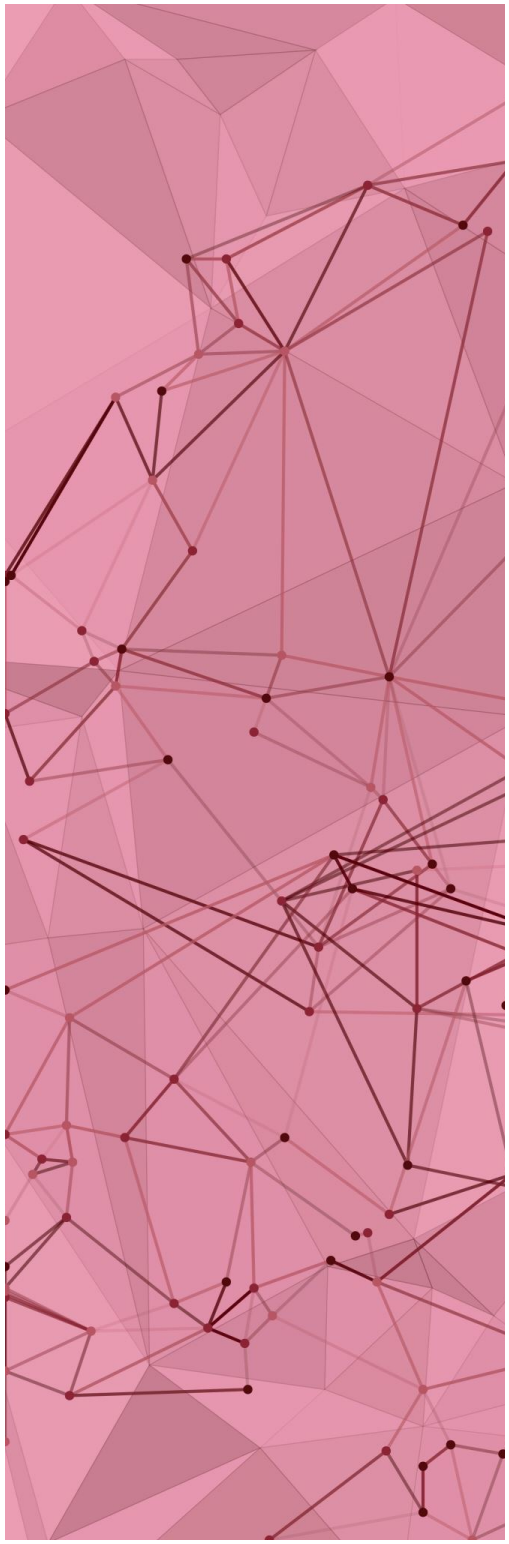
```
list_gen  [ ] = [ ]
list_gen (x:xs) = [2*x]  ++ list_gen xs
```



Anatomy of a list

- Two parts
  - Head (typically named x)
  - Tail (typically named xs): list of the remaining elements
- Functions `head` and `tail` return these
  - `head evens`
  - `tail evens`
- Joining head and tail by : operator
  - 0 : [2, 4 .. 10] will give [0, 2, 4, 6, 8, 10]
- Indexing function is !!
  - evens !! 2
- last list gives the last element of a list
- Reference
  - http://www.haskell.org/haskellwiki/How_to_work_on_lists





Additional Units

Time Permitting

Problem solving using Haskell
Main ingredient: recurrence relation

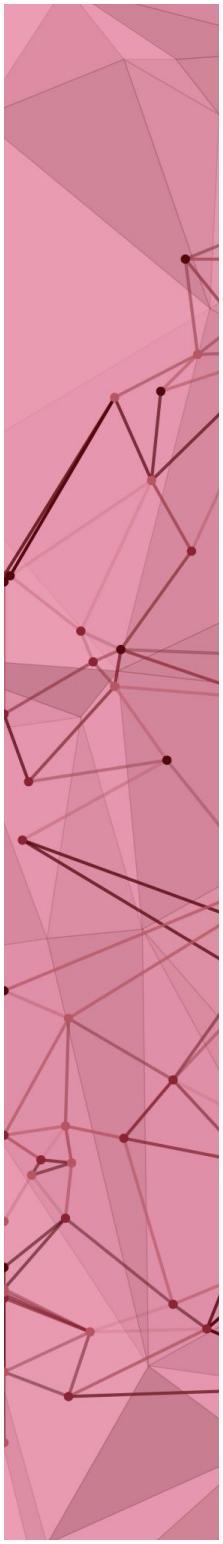
Problem: n-th Fibonacci number

```
fib n =
    if n == 0
        then 1
        else if n == 1
            then 1
            else if n > 1
                then fib(n-1) + fib(n-2)
                else 0

 fib2 n
    | (n == 0) || (n == 1) = 1
    | n > 1 = fib(n-1) + fib(n-2)
    | otherwise = 0
```

Otherwise is alias for True

How about memoized version of fibonacci?



- Recursive sum

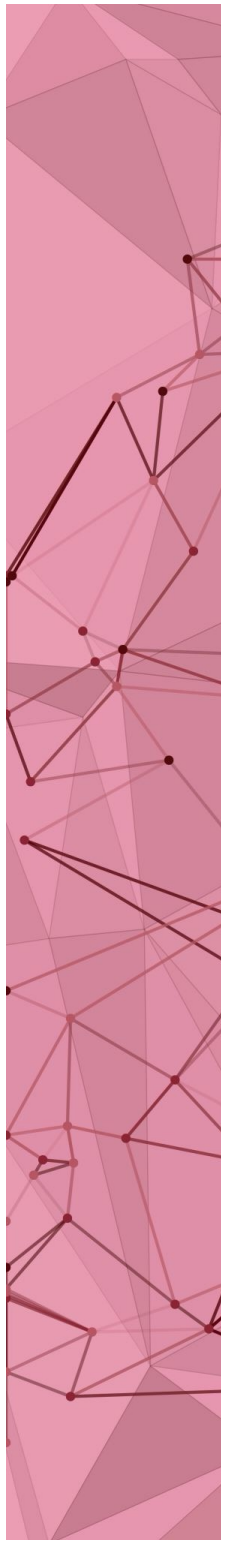
```
recursiveSum [] = 0
recursiveSum (x:xs) =
          x + recursiveSum xs
```

- Built-in function that does the above: sum

```
sum [1 .. 5]
```

- Similar built-in function: product

```
factorial n = product [1 .. n]
```

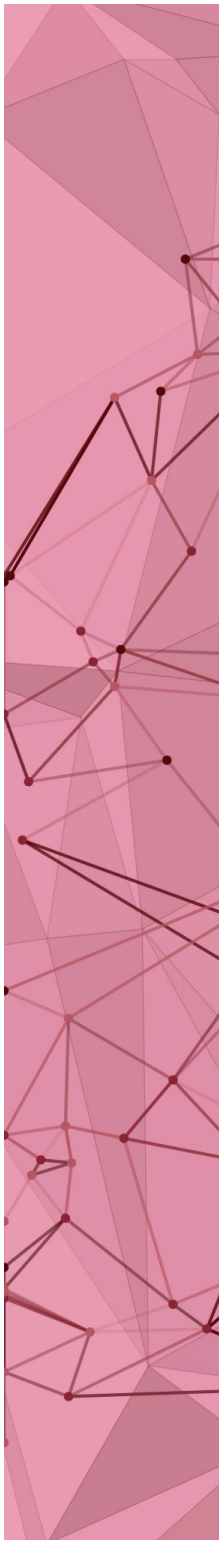


Problem: sort a list (low to high)

```
sort2 [] = []
sort2 (x:xs) =
        sort2 [low | low <- xs, low < x]
        ++   [x]  ++
        sort2 [high | high <- xs, high >= x]
```

Empty list

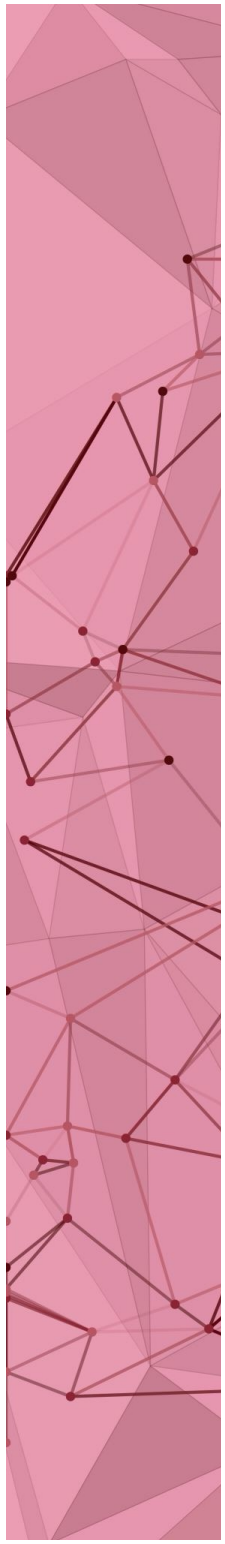
List generator



n-queens problem

One of the 92 solutions of 8-queens: [3,1,6,2,5,7,4,0]

	Col 0	Col 1	Col 2	Col 3	Col 4	Col 5	Col 6	Col 7
Row 0								X
Row 1		X						
Row 2				X				
Row 3	X							
Row 4							X	
Row 5					X			
Row 6			X					
Row 7						X		



n-queens problem
Generate all possible solutions

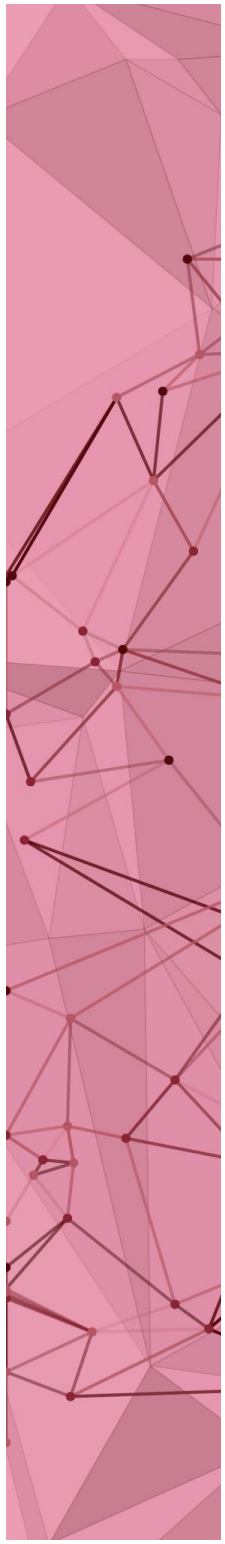
```
queens n = solve n
        where
        solve k
          | k <= 0 = [ [] ]
          | otherwise = [ h:partial | partial <- solve (k-1), h <- [0..(n-1)], safe h partial]
        safe h partial = and [ not (checks h partial i) | i <- [0..(length partial - 1)]]
        checks h partial i = h == partial!!i || abs(h - partial!!i) == i+1
```

where allows us to use the value of n here (context)

More examples

Factors of a number n

```
factors n = [f | f <- [1 .. n],
                  mod n f == 0]
```



More examples

- Compute ALL prime numbers!

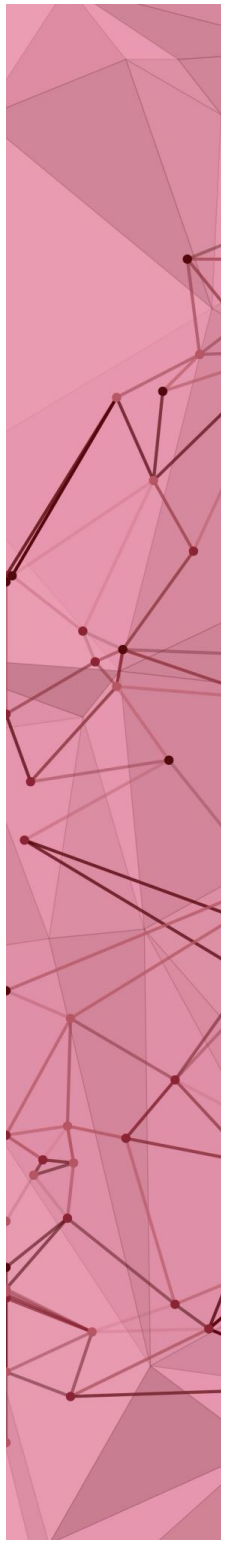
```
primes = primeGen' [2 ..]
  where
  primeGen' (p:xs) = p : primeGen’ [q|
  q <- xs, mod q p /= 0] -- same line
```

- Getting the first 10 prime numbers

```
take 10 primes
```

Lazy evaluation

- Side note: This is not really the sieve of Eratosthenes
  http://www.cs.hmc.edu/~oneill/papers/Sieve-JFP.pdf



Higher-order function/ curried function

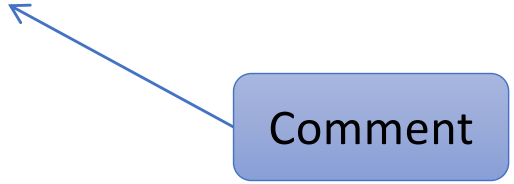
- One function can be a parameter of another functions
- One function can return another function

Haskell Curry

Higher-order function/ curried function

- In GHCI command area:

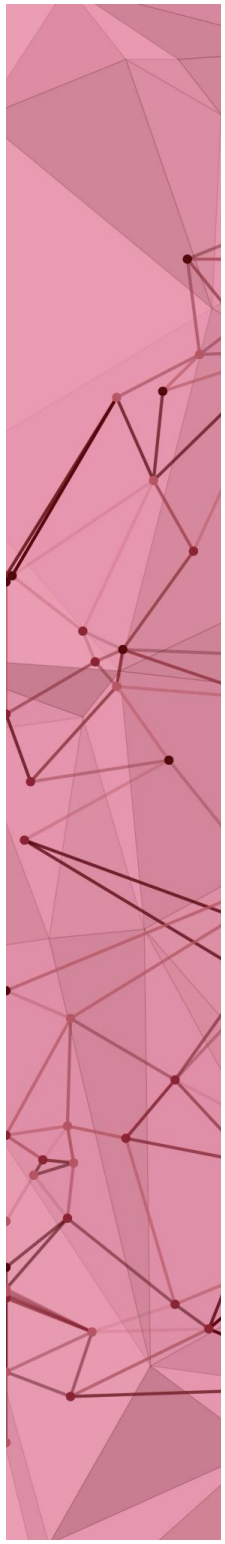
```
let fun3 x y z = x+y-z -- 3 param function
fun3 1 2 3 -- outputs 0
```



- This is how Haskell deals with the fun3 function:

```
let fun2 = fun3 1
let fun1 = fun2 2
fun1 3 -- outputs 0
```

- Functions can return a (partial) function



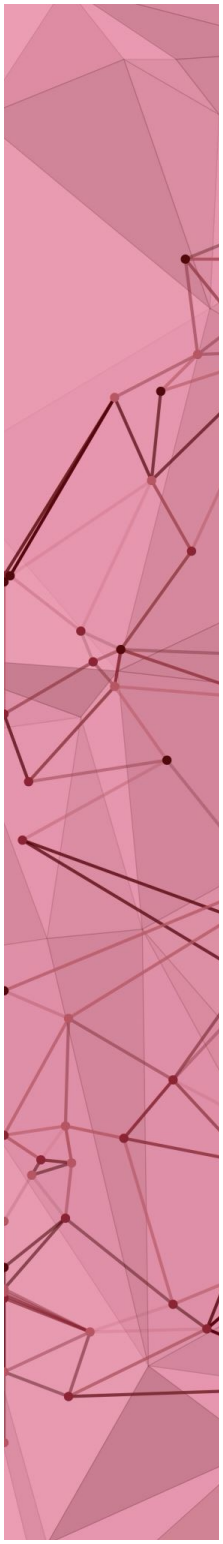
Useful built-in functions

1. map function list
   - `map ( > 0)  [2, -50, 100]` -- → [True, False, True]
     - Equivalent: `[ x > 0 | x <- [2, -50, 100] ]`
   - `map (`mod` 2) [13, 14, 15]` -- → [1, 0, 1]
2. filter condition list
   - Examples
   - Sorting function (from last class)

```
qsort [] = []
qsort (x:xs) =
        qsort (filter (< x) xs)
        ++ [x] ++
        qsort (filter (>= x) xs)
```

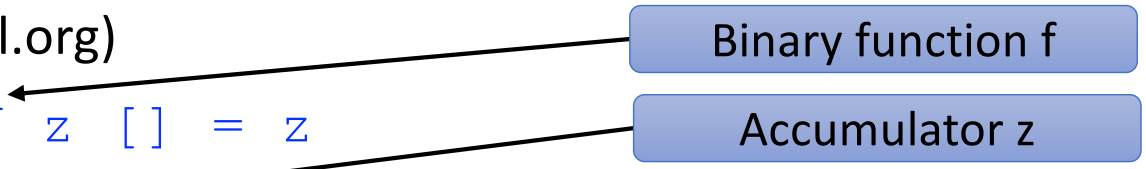
`qsort [b| b <- xs, b < x]`

Difference between map and filter?



Useful built-in functions

3. Fold: produces single value from a list
(From Haskell.org)




- ```
  foldl f z [] = z
  foldl f z (x:xs) = foldl f (f z x) xs
  ```

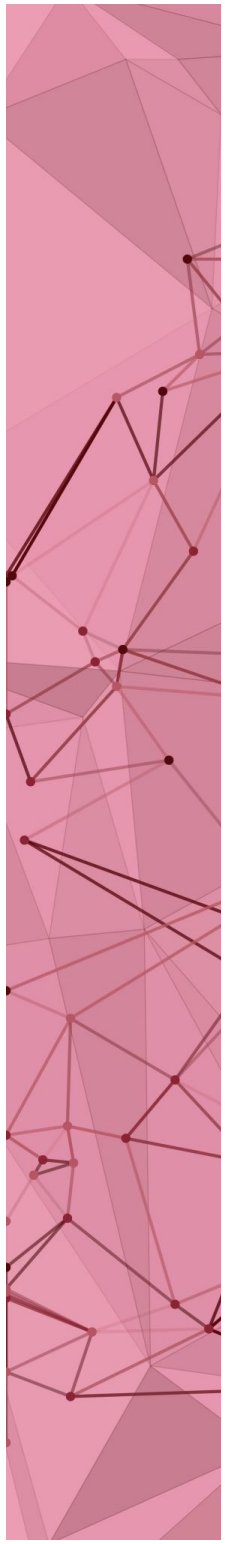
- ```
  foldr f z []     = z
  foldr f z (x:xs) = f x (foldr f z xs)
  ```

- Examples
  - `foldl (-) 1 [2, 3, 4]` -- 1 -2 -3 -4
  - `foldr (-) 1 [2, 3, 4]` -- 2 – [3 – [4 - 1]]
  - `anyTrue = foldr (||) False [True, True ..]`
  - `anyTrue = foldl (||) False [True, True ..]`

Useful built-in functions

- zipWith
  - Arguments: a function and two lists
  - Applies that function to the corresponding elements of the list and produces a new list

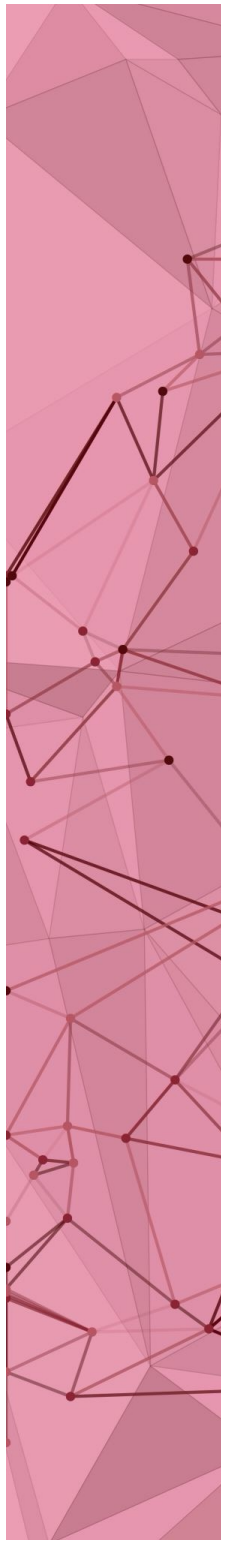
```
ghci> zipWith (+) [1,2,3,4] [5,6,7,8]
[6,8,10,12]
ghci> zipWith max [6,3,2,1] [7,3,1,5]
[7,3,2,5]
ghci> zipWith (++) ["foo ", "bar ", "baz "] ["fighters", "hoppers", "aldrin"]
["foo fighters","bar hoppers","baz aldrin"]
```



Evaluate Polish prefix expression

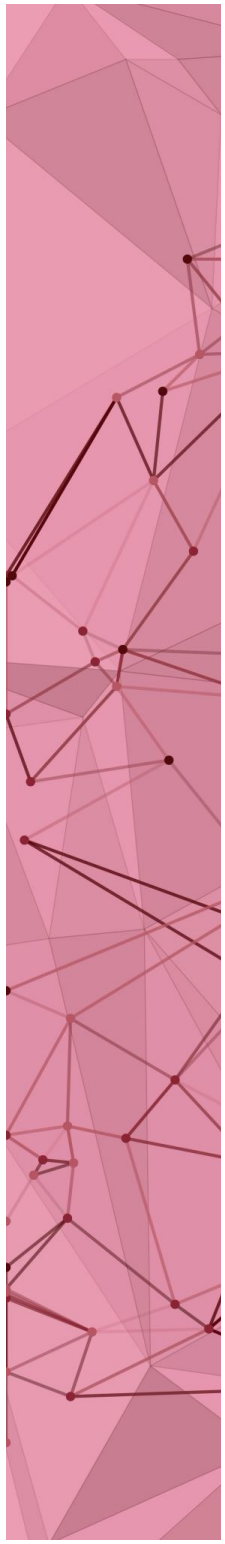
- 10 + (15 – 5) * 4 is in prefix: + 10 * - 15 5 4
- 10 + 15 – 5 * 4 is in prefix: + 10 - 15 * 5 4
- Input: String (list of characters) of Polish prefix expression
- Output: Value of expression
- Solution
  - Reverse the expr
  - Traverse it from left to right
  - Stack operations (implement by a list: head is TOS)
    - If the current string is an operator, pop the top two operands and push the result of applying the operator

foldl



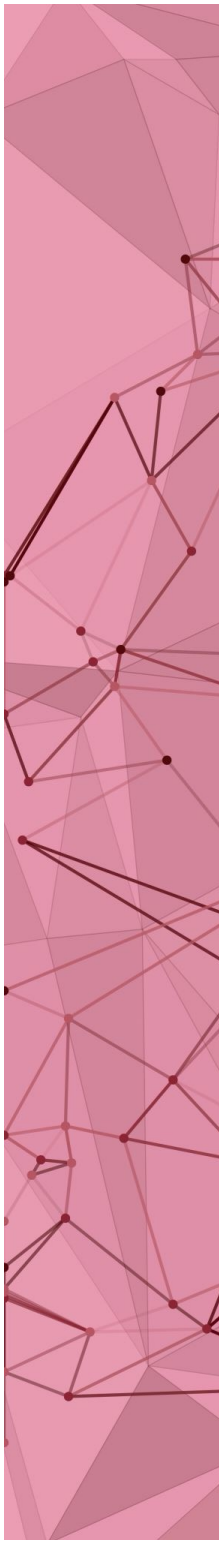
Solution

```
evaluate expr = head(foldl stackOperation []
                        (reverse(words(expr))))
 where
 stackOperation (x:y:xs) curString
              | curString == "+" = (x+y):xs
                    | curString == "-" = (x-y):xs
                    | curString == "*" = (x*y):xs
 stackOperation xs curString = read curString:xs
```



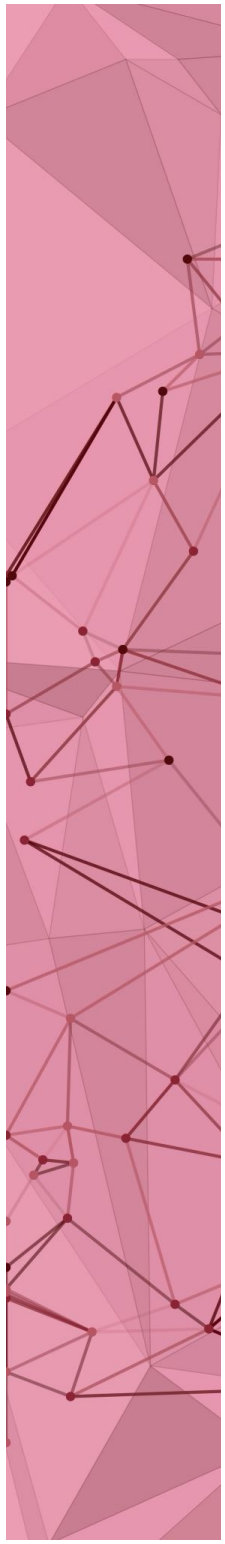
Alternative solution

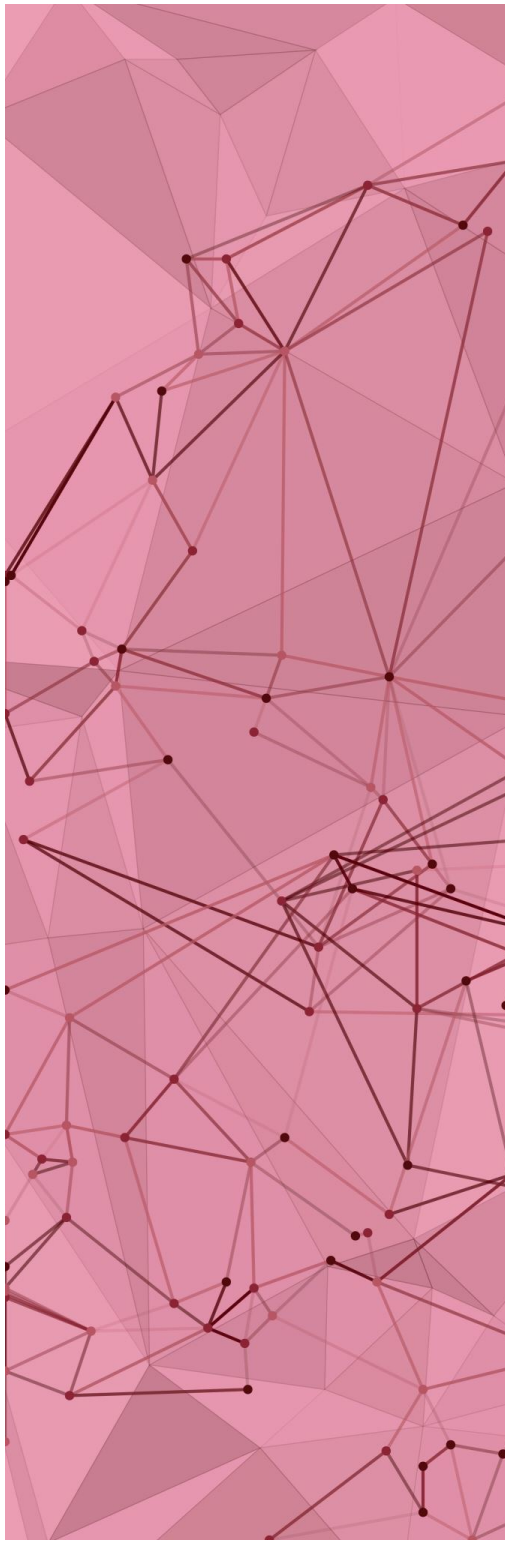
```
evaluatePrefix expr
     = head ( foldl f [] ( reverse (words expr) )
                )
       where      f (x:y:ys) "*" = (x*y):ys
                        f (x:y:ys) "+" = (x+y):ys
                        f (x:y:ys) "-" = (x-y):ys
                        f xs str = read str:xs
```



Memoization for Fibonacci

```
-- Memoized fibonacci numbers
fibs = [fibMem n | n <- [0..]] --infinite list
fibMem n
    | n == 0 = 1
    | n == 1 = 1
    | otherwise = fibs!!(n-1) + fibs!!(n-2)
```



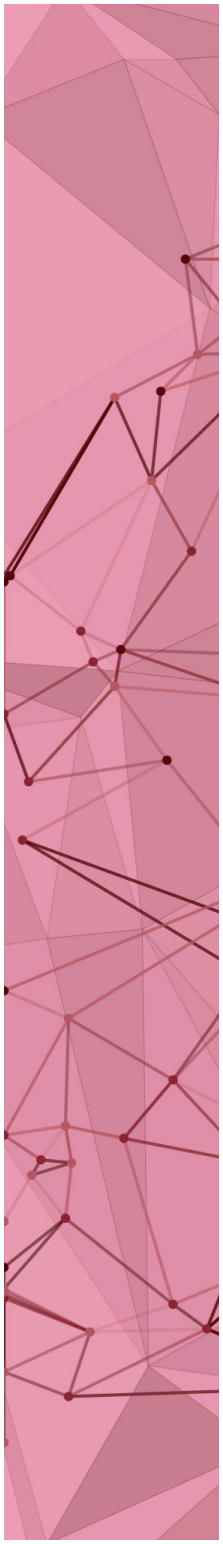


Theoretical Foundation

Optional

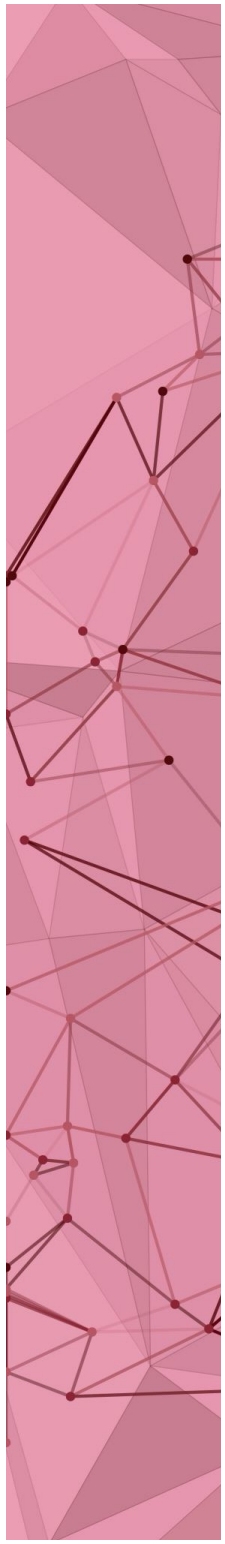
Theoretical foundation: λ calculus

- Optional reading: Scott's section 11.2.4* (on Canvas)
- Alonzo Church's λ calculus
  - Goal: express computation using mathematical functions [1936—1940]
  - Think about sqrt: R → R
    - Where's the algorithm?
  - Key ideas in λ calculus
    - Function abstraction and application
    - Name/"variable" binding and substitution
- λ calculus examples
  - λx . x * x * x (known as λ abstraction)
  - (λx . x * x * x) 2 (known as function application)
    - yields 8



Building block: λ expression (expr) Recursive definition

- A name
  - times (that is, arithmetic * operation)
  - x
- (expr)
- A λ abstraction: λ name.expr
  - λx.x (identity)
  - λx.5 (constant)
  - λx.times x x (times x x is another expr– see below)
- A function application: two adjacent expr, the first is a function applied to the second
  - times x x <--> curried function (times x) x



Examples

- Can name λ expressions
  - e.g., hypotenuse below
- hypotenuse =
  λx.λy.sqrt (plus (square x) (square y))
- hypotenuse 3 4
  - yields 5
  - How? By substitution: x = 3, y = 4 (an instance of "beta reduction")
  - Two other rules
    - alpha conversion: nothing changes if you rename a "variables" (under certain conditions)
    - eta reduction: if E has nothing to do with x in λx.E, then λx.E is the same as just E