Open4Tech Summer School 2019

All Things JavaScript
Coding Pro-Practices
Curry On Functional Programming
HTML, CSS & JS in the Real World
Java vs Python: Coding Deadmatch

OOP Techniques in a Simple Game
REST In Node.JS At The React & Angular SPA
Sneak Peek Into Next Level QA (Test Automation)
Windows App Development with .NET WPF
You'll Neversea Algorithms Like These

24 iunie - 12 iulie 2019
http://inf.ucv.ro/~summer-school/

DEPARTAMENTUL DE INFORMATICĂ

CAPHYON

NETROM premium software

Society for Computing Technologies

Welcome!

Open4Tech Summer School 2019

	Luni	Marti	Miercuri	Joi	Vineri
	24 iunie	25 iunie	26 iunie	27 iunie	28 iunie
2-4pm	All Things JavaScript	All Things JavaScript	REST in Node.JS at the React & Angular SPA	REST in Node.JS at the React & Angular SPA	All Things JavaScript
4-6pm	OOP Techniques in a Simple Game	OOP Techniques in a Simple Game	OOP Techniques in a Simple Game	Windows App Development with .NET WPF	Windows App Development with .NET WPF
6-8pm	HTML, CSS & JS in the Real World	HTML, CSS & JS in the Real World	HTML, CSS & JS in the Real World	Java vs Python: Coding Deathmatch	Java vs Python: Coding Deathmatch
	1 iulie	2 iulie	3 iulie	4 iulie	5 iulie
2-4pm			REST in Node.JS at the React & Angular SPA	REST in Node.JS at the React & Angular SPA	
4-6pm	Coding Pro-Practices	Coding Pro-Practices	Coding Pro-Practices	You'll Neversea Algorithms Like These	You'll Neversea Algorithms Like These
6-8pm	Java vs Python: Coding Deathmatch	Sneak Peek Into Next Level QA (Test Automation)	Sneak Peek Into Next Level QA (Test Automation)	Curry On Functional Programming	Curry On Functional Programming
	8 iulie	9 iulie	10 iulie	11 iulie	12 iulie
2-4pm					
4-6pm	You'll Neversea Algorithms Like These		REST in Node.JS at the React & Angular SPA	REST in Node.JS at the React & Angular SPA	
6-8pm	Curry On Functional Programming				

http://inf.ucv.ro/~summer-school/

https://www.caphyon.ro/open4tech-2019.html

Open4Tech
Summer School 2019
24 iunie - 12 iulie

Curry On Functional Programming

July, 2019
Craiova

CAPHYON

@ciura_victor

Victor Ciura
Technical Lead, Caphyon
www.caphyon.ro

Abstract

Can a language whose official motto is “Avoid Success at All Costs” teach us new tricks in modern programming languages?

If Haskell is so great, why hasn't it taken over the world? My claim is that it has. But not as a Roman legion loudly marching in a new territory, rather as distributed Trojan horses popping in at the gates, masquerading as modern features or novel ideas in today’s mainstream languages. Functional Programming ideas that have been around for over 40 years will be rediscovered to solve our current software complexity problems.
Indeed, modern programming languages have become more functional. From mundane concepts like lambdas & closures, function objects, values types and constants, to composability of algorithms, ranges, folding, mapping or even higher-order functions.

In this workshop we’ll analyze a bunch of FP techniques and see how they help make our code shorter, clearer and faster, by embracing a declarative vs. an imperative style. Brace yourselves for a bumpy ride including composition, lifting, currying, partial application, pure functions, maybe even pattern matching and lazy evaluation.

Spoiler: no unicorns here.

Who Am I ?

Advanced Installer

Clang Power Tools

@ciura_victor

Curry On
Functional Programming

What is it all about ?

Haskell

ranges

std::optional

C++

algorithms

STL

Maybe | Just

lifting

monoids

lambdas & closures

fold

values types

lazy evaluation

declarative vs imperative

algebraic data types

monads

higher order functions

map

composition

pattern matching

FP

expressions vs statements

pure functions

currying

partial application

category theory

recursion

Paradox of Programming

Machine/**Human** impedance mismatch:

- Local/Global perspective
- Progress/Goal oriented
- Detail/Idea
- Vast/Limited memory
- Pretty reliable/Error prone
- Machine language/Mathematics

Is it easier to think like a machine than to do math?

A Crash Course in Category Theory - Bartosz Milewski https://www.youtube.com/watch?v=JH_Ou17_zyU

Semantics

- The meaning of a program
- Operational semantics: local, progress oriented
  - Execute program on an abstract machine in your brain
- Denotational semantics
  - Translate program to math
- Math: an ancient language developed for humans

A Crash Course in Category Theory - Bartosz Milewski https://www.youtube.com/watch?v=JH_Ou17_zyU

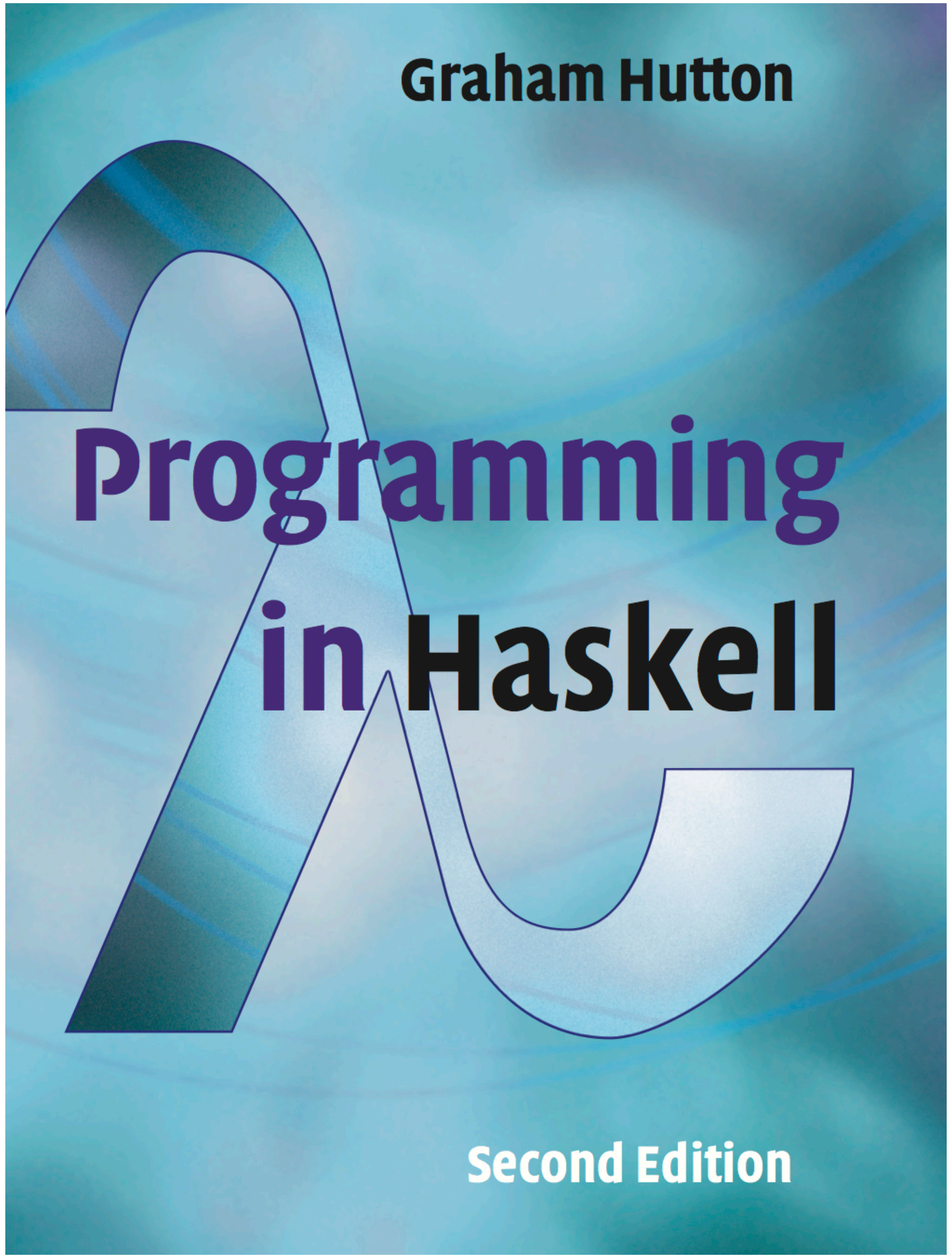
What is Functional Programming ?

- Functional programming is a **style** of programming in which the basic method of computation is the ***application of functions*** to arguments

- A functional **language** is one that supports and encourages the ***functional style***

Let's address the 🐘 in the room...

Haskell



https://www.amazon.com/Programming-Haskell-Graham-Hutton/dp/1316626229/

A functional language is one that supports and encourages the functional style

What do you mean ?

Summing the integers 1 to 10 in C++/Java/C#

```
int total = 0;
for (int i = 1; i ≤ 10; i++)
   total = total + i;
```

The computation method is variable assignment.

Summing the integers 1 to 10 in Haskell

```
sum [1..10]
```

The computation method is function application.

Sneak Peek Into Next Level QA (Test Automation) - Antonio Valent

Historical Background

Historical Background

Most of the "new" ideas and innovations in modern programming languages are actually very old...

Historical Background

1930s

Alonzo Church develops the lambda calculus,
a simple but powerful *theory of functions*

Historical Background

1950s

John McCarthy develops Lisp, the *first functional language*, with some influences from the lambda calculus, but retaining *variable assignments*

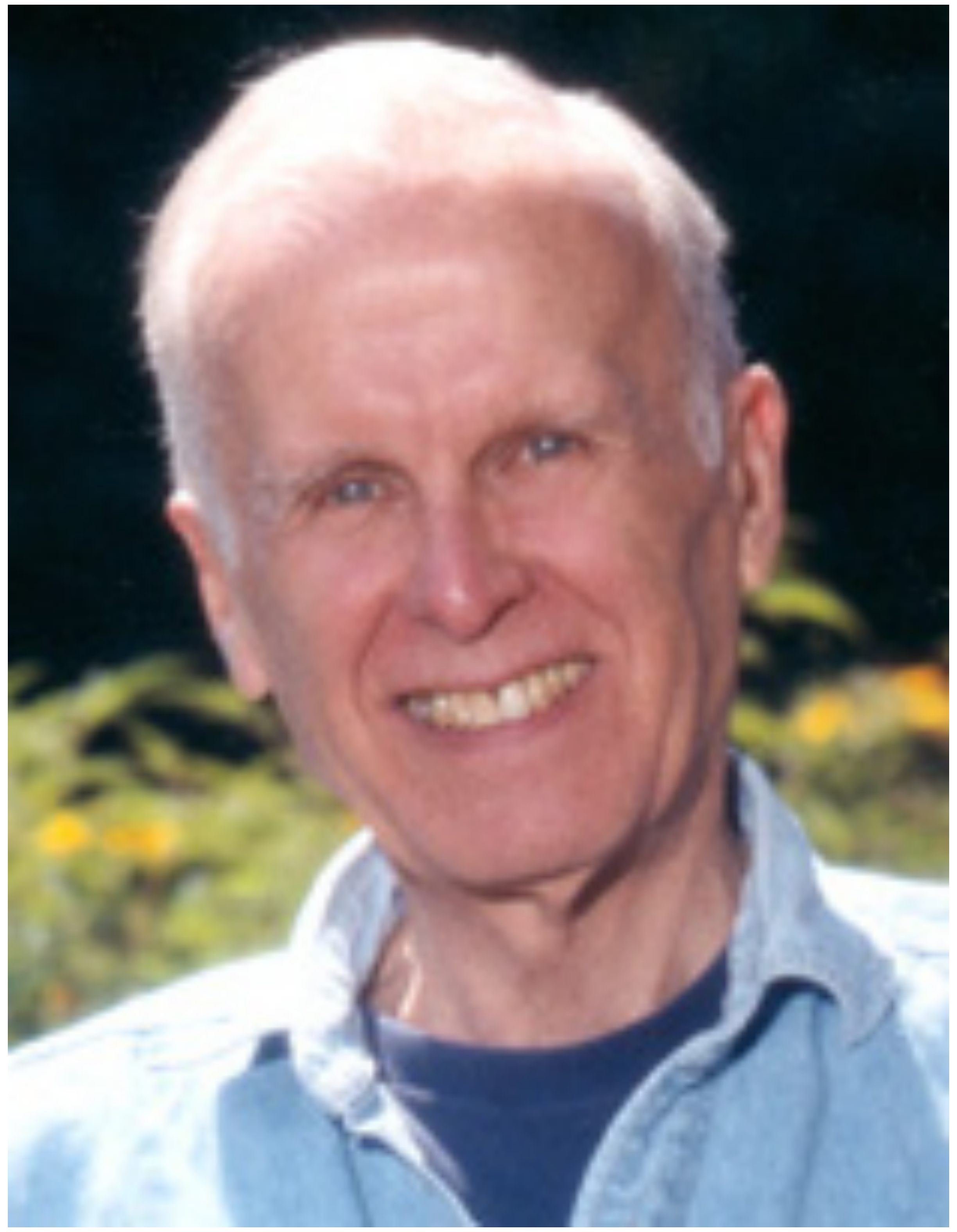
Historical Background

1960s

Peter Landin develops ISWIM, the first *pure functional language*, based strongly on the lambda calculus, with *no assignments*

Historical Background

1970s



John Backus develops FP, a functional language that emphasizes *higher-order functions* and reasoning about programs

Historical Background

1970s

Robin Milner and others develop ML, the first modern functional language, which introduced *type inference* and *polymorphic types*

Historical Background

1970-80s

David Turner develops a number of lazy *functional languages*, culminating in the Miranda system

Historical Background

1987

An advanced purely-functional programming language

An international committee starts the development of Haskell,
a standard *lazy functional language*

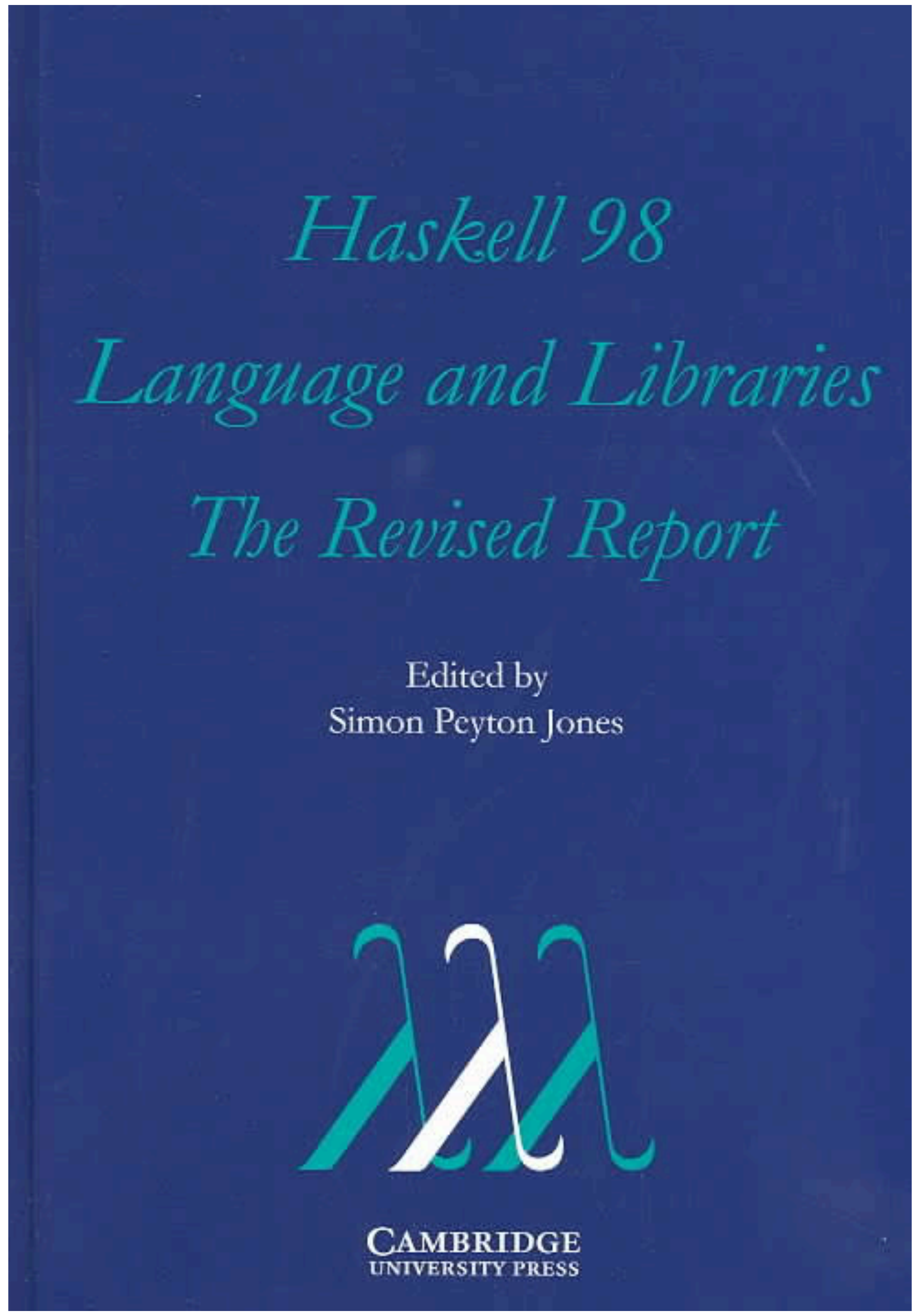
Historical Background

1990s

Phil Wadler and others develop type classes and monads, two of the main innovations of Haskell

Historical Background

2003
2010



The committee publishes the Haskell Report, defining a stable version of the language; an updated version was published in 2010

Historical Background

2010-2019

Haskell Platform

Haskell with batteries included

- standard distribution
- library support
- new language features
- development tools
- use in industry
- influence on other languages

A Taste of Haskell

```
f []     = []
f (x:xs) = f ys ++ [x] ++ f zs
           where
              ys = [a | a ← xs, a ≤ x]
              zs = [b | b ← xs, b > x]
```

What does f do ?

Standard Prelude

Haskell comes with a large number of standard library functions

Select the first element of a list:

```
> head [1,2,3,4,5]
1
```

Remove the first element from a list:

```
> tail [1,2,3,4,5]
[2,3,4,5]
```

Standard Prelude

Select the nth element of a list:

```
> [1,2,3,4,5] !! 2
3
```

Select the first n elements of a list:

```
> take 3 [1,2,3,4,5]
[1,2,3]
```

Standard Prelude

Remove the first n elements from a list:

```
> drop 3 [1,2,3,4,5]
[4,5]
```

Calculate the length of a list:

```
> length [1,2,3,4,5]
5
```

Calculate the sum of a list of numbers:

```
> sum [1,2,3,4,5]
15
```

Standard Prelude

Calculate the product of a list of numbers:

```
> product [1,2,3,4,5]
120
```

Append two lists:

```
> [1,2,3] ++ [4,5]
[1,2,3,4,5]
```

Reverse a list:

```
> reverse [1,2,3,4,5]
[5,4,3,2,1]
```

Function Application

```
f a b + c*d
```

f applied to a and b

Function application is assumed to have higher priority than all other operators:

```
f a + b
```

means (f a) + b rather than f (a + b)

Function Application

Mathematics	Haskell
`f(x)`	`f x`
`f(x,y)`	`f x y`
`f(g(x))`	`f (g x)`
`f(x,g(y))`	`f x (g y)`
`f(x) g(y)`	`f x * g y`

My First Function

```
double x = x + x

quadruple x = double (double x)
```

```
> quadruple 10
40

> take (double 2) [1,2,3,4,5,6]
[1,2,3,4]
```

Infix Functions

```
average ns = sum ns `div` length ns
```

`x `f` y` is just syntactic sugar for `f x y`

The Layout Rule

The layout rule avoids the need for explicit syntax to indicate the grouping of definitions

```
a = b + c
    where
      b = 1
      c = 2
d = a * 2
```

means

```
a = b + c
    where
    {b = 1;
     c = 2}
 d = a * 2
```

implicit grouping

explicit grouping

Types in Haskell

If evaluating an expression `e` would produce a value of type `t`,

then `e` has type `t`, written as `e::t`

Every well formed expression has a type, which can be automatically calculated at compile time using a process called type inference

All type errors are found at compile time,
=> makes programs safer and faster by removing the need for type checks at run time

List Types

A list is sequence of values of the same type:

```
[False,True,False] :: [Bool]

['a','b','c','d'] :: [Char]

[['a'],['b','c']] :: [[Char]]
```

Tuple Types

```
(False,True) :: (Bool,Bool)

(False,'a',True) :: (Bool,Char,Bool)


('a',(False,'b')) :: (Char,(Bool,Char))

(True,['a','b']) :: (Bool,[Char])
```

Function Types

A function is a mapping from values of one type to values of another type:

```
not :: Bool → Bool

even :: Int → Bool
```

Function Types

```
add :: (Int,Int) → Int
add (x,y) = x+y

zeroto :: Int → [Int]
zeroto n = [0..n]
```

Curried Functions

Functions with multiple arguments are also possible by returning functions as results:

```
add' :: Int → (Int → Int)
add' x y = x+y
```

`add'` takes an integer **x** and returns a function `add' x`

In turn, this new function takes an integer **y** and returns the result **x+y**

Curried Functions

add and add' produce the same final result,
but add takes its two arguments *at the same time*,
whereas add' takes them *one at a time*:

```
add :: (Int,Int) → Int

add' :: Int → (Int → Int)
```

Functions that take their arguments *one at a time* are called curried functions,
celebrating the work of Haskell Curry on such functions.

Curried Functions

Functions with more than two arguments can be curried by returning *nested functions*:

```
mult :: Int → (Int → (Int → Int))
mult x y z = x*y*z
```

mult takes an integer x and returns a function mult x, which in turn takes an integer y and returns a function mult x y, which finally takes an integer z and returns the result x*y*z

Curried Functions

Curried functions are more *flexible* than functions on tuples, because useful functions can often be made by partially applying a curried function.

```
add' 1 :: Int → Int

take 5 :: [Int] → [Int]

drop 5 :: [Int] → [Int]
```

Currying Conventions

To avoid excess parentheses when using curried functions, two simple conventions are adopted:

`Int → Int → Int → Int`

The arrow → associates to the right

same as: `Int → (Int → (Int → Int))`

Currying Conventions

As a consequence, it is then natural for function application to associate to the left

```
mult x y z
```

```
means ((mult x) y) z
```

Unless *tupling* is explicitly required, all functions in Haskell are normally defined in *curried form*

Polymorphic Functions

A function is called polymorphic if its type contains one or more *type variables*

```
length :: [a] → Int
```

For any type a, `length` takes a list of values of type a and returns an integer

Polymorphic Functions

Type variables can be instantiated to different types in different circumstances:

```
> length [False,True]
2
```

a = Bool

```
> length [1,2,3,4]
4
```

a = Int

Type variables must begin with a **lower-case letter**, and are usually named a, b, c...

Polymorphic Functions

Many of the functions defined in the standard prelude are polymorphic:

```
fst :: (a,b) → a

head :: [a] → a

take :: Int → [a] → [a]

zip :: [a] → [b] → [(a,b)]

id :: a → a
```

Guarded Equations

```
abs :: Int → Int
abs n = if n ≥ 0 then n else -n
```

As an alternative to **conditionals**,
functions can also be defined using **guarded equations**

```
abs n | n ≥ 0     = n
      | otherwise = -n
```

Guarded Equations

```
signum :: Int → Int
signum n = if n < 0 then -1 else
              if n == 0 then 0 else 1
```

Guarded equations can be used to make definitions involving multiple conditions easier to read:

```
signum n | n < 0     = -1
         | n == 0    = 0
         | otherwise = 1
```

The catch all condition otherwise is defined in the prelude by `otherwise = True`

Pattern Matching

```haskell
not :: Bool → Bool
not False = True
not True  = False
```

Pattern Matching

```
(&&) :: Bool → Bool → Bool
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

can be defined more compactly by:

```
(&&) :: Bool → Bool → Bool
True && True = True
_    && _    = False
```

underscore symbol _ is a **wildcard** pattern that matches any argument value

Pattern Matching

However, the following definition is more efficient,

because it avoids evaluating the second argument if the first argument is False

```
(&&) :: Bool → Bool → Bool
True  && b = b
False && _ = False
```

underscore symbol _ is a **wildcard** pattern that matches any argument value

Pattern Matching

Patterns are matched **in order**.

The following definition always returns False:

```
_    && _    = False
True && True = True
```

List Patterns

Internally, every non-empty list is constructed by repeated use of an `operator (:)` called “**cons**” that adds an element to the *start of a list*

`[1,2,3,4]`

means `1:(2:(3:(4:[])))`

List Patterns (x:xs)

Functions on lists can be defined using `x:xs` patterns

```
head :: [a] → a
head (x:_) = x

tail :: [a] → [a]
tail (_:xs) = xs
```

x:xs patterns only match non-empty lists:

```
> head []
*** Exception: empty list
```

Lambda Expressions

$\lambda x \rightarrow x + x$

$\backslash x \rightarrow x + x$

the nameless function that takes a number **x** and returns the result **x + x**

Lambda Expressions

Lambda expressions can be used to avoid naming functions that are only referenced once

```
odds n = map f [0..n-1]
         where
            f x = x*2 + 1
```

can be simplified to:

```
odds n = map (\x → x*2 + 1) [0..n-1]
```

Set Comprehensions

In mathematics, the comprehension notation can be used to construct new sets from old sets

$$\{ x^2 \mid x \in \{1...5\} \}$$

the set {1,4,9,16,25} of all numbers x^2 such that x is an element of the set {1…5}

Set Comprehensions

In Haskell, a similar comprehension notation can be used to construct new lists from old lists

```
[x^2 | x ← [1..5]]
```

the set {1,4,9,16,25} of all numbers x^2 such that x is an element of the set {1…5}

Set Comprehensions

```
[x^2 | x ← [1..5]]
```

The expression `x ← [1..5]` is called a **generator**,
as it states how to generate values for `x`

Comprehensions can have multiple generators, separated by commas:

```
> [(x,y) | x ← [1,2,3], y ← [4,5]]

[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

Set Comprehensions

Changing the **order** of the generators changes the order of the elements in the final list:

```
> [(x,y) | y ← [4,5], x ← [1,2,3]]

[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

Multiple generators are like **nested loops**, with later generators as more deeply nested loops whose variables change value more frequently.

Set Comprehensions

```
> [(x,y) | y ← [4,5], x ← [1,2,3]]

[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

x ← [1,2,3] is the last generator, so the value of the x component of each pair changes most frequently.

Dependant Generators

Later generators can depend on the variables that are introduced by earlier generators

`[(x,y) | x ← [1..3], y ← [x..3]]`

The list `[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]` **of all pairs of numbers** `(x,y)` **such that** `x,y` **are elements of the list** `[1..3]` **and** `y ≥ x`

Dependant Generators

Using a dependant generator we can define the library function that **concatenates** a list of lists:

```
concat :: [[a]] → [a]
concat xss = [x | xs ← xss, x ← xs]
```

```
> concat [[1,2,3],[4,5],[6]]

[1,2,3,4,5,6]
```

Guards

List comprehensions can use guards to **restrict** the values produced by earlier generators

```
[x | x ← [1..10], even x]
```

The list [2,4,6,8,10] of all numbers x such that x is an element of the list [1..10] and x is even

Guards

Using a guard we can define a function that maps a positive integer to its list of factors:

```
factors :: Int → [Int]
factors n = [x | x ← [1..n], n `mod` x == 0]
```

```
> factors 15

[1,3,5,15]
```

Guards

A positive integer is **prime** if its only factors are 1 and itself.
Using `factors` we can define a function that decides if a number is *prime*:

```
prime :: Int → Bool
prime n = factors n == [1,n]
```

```
> prime 15
False

> prime 7
True
```

Guards

Using a guard we can now define a function that returns the list of **all primes** up to a given limit:

```
primes :: Int → [Int]
primes n = [x | x ← [2..n], prime x]
```

```
> primes 40

[2,3,5,7,11,13,17,19,23,29,31,37]
```

Zip Function

A useful library function is **zip**, which maps two lists to a list of pairs of their corresponding elements

```
zip :: [a] → [b] → [(a,b)]
```

```
> zip ['a','b','c'] [1,2,3,4]

[('a',1),('b',2),('c',3)]
```

Zip Function

Using zip we can define a function returns the list of all **pairs of adjacent elements** from a list:

```
pairs :: [a] → [(a,a)]
pairs xs = zip xs (tail xs)
```

```
> pairs [1,2,3,4]

[(1,2),(2,3),(3,4)]
```

Zip Function

Using pairs we can define a function that decides if the elements in a list are **sorted**:

```
sorted :: Ord a ⇒ [a] → Bool
sorted xs = and [x ≤ y | (x,y) ← pairs xs]
```

```
> sorted [1,2,3,4]
True

> sorted [1,3,2,4]
False
```

String Comprehensions

A string is a sequence of characters enclosed in double quotes.
Internally, however, strings are represented as *lists of characters*.

```
"abc" :: String
```

means ['a', 'b', 'c'] :: [Char]

String Comprehensions

Because strings are just special kinds of **lists**,
any polymorphic function that operates on lists can also be applied to strings.

```
> length "abcde"
5

> take 3 "abcde"
"abc"

> zip "abc" [1,2,3,4]
[('a',1),('b',2),('c',3)]
```

String Comprehensions

List comprehensions can also be used to define functions on strings, such counting how many times a character **occurs** in a string:

```
count :: Char → String → Int
count x xs = length [x’ | x’ ← xs, x == x’]
```

```
> count ’e’ "Open4Tech Summer School"
3
```

Recursive Functions

```
fac 0 = 1
fac n = n * fac (n-1)
```

```
fac 3
3 * fac 2
3 * (2 * fac 1)
3 * (2 * (1 * fac 0))
3 * (2 * (1 * 1))
3 * (2 * 1)
3 * 2
6
```

Recursive Functions

```
product :: Num a ⇒ [a] → a

product []     = 1
product (n:ns) = n * product ns
```

```
product [2,3,4]
2 * product [3,4]
2 * (3 * product [4])
2 * (3 * (4 * product []))
2 * (3 * (4 * 1))
24
```

Recursive Functions

```haskell
length :: [a] → Int
length []     = 0
length (_:xs) = 1 + length xs
```

```haskell
length [1,2,3]
1 + length [2,3]
1 + (1 + length [3])
1 + (1 + (1 + length []))
1 + (1 + (1 + 0))
3
```

Recursive Functions

```
reverse :: [a] → [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

```
reverse [1,2,3]
reverse [2,3] ++ [1]
(reverse [3] ++ [2]) ++ [1]
((reverse [] ++ [3]) ++ [2]) ++ [1]
(([] ++ [3]) ++ [2]) ++ [1]
[3,2,1]
```

Recursive & Multiple Args

```
zip :: [a] → [b] → [(a,b)]
zip []       _       = []
zip _        []      = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

Recursive & Multiple Args

```
drop :: Int → [a] → [a]
drop 0 xs       = xs
drop _ []       = []
drop n (_:xs) = drop (n-1) xs
```

Recursive & Multiple Args

```
(++) :: [a] → [a] → [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Quick Sort

Rules:

1. The *empty* list is already sorted.
2. Non-empty lists can be sorted by sorting the **tail values** ≤ the **head**, sorting the **tail values** > the **head**, and then appending the resulting lists on either side of the head value.

Quick Sort

```haskell
qsort :: Ord a ⇒ [a] → [a]

qsort []       = []
qsort (x:xs) =
   qsort smaller ++ [x] ++ qsort larger
   where
      smaller = [a | a ← xs, a ≤ x]
      larger  = [b | b ← xs, b > x]
```

Quick Sort

```
q [3,2,4,1,5]

q [2,1]  ++ [3] ++  q [4,5]

q [1] ++ [2] ++ q []    q [] ++ [4] ++ q [5]

[1]              []     []              [5]
```

Higher-Order Functions

A function is called ***higher-order*** if it takes a function as an argument or returns a function as a result.

```
twice :: (a → a) → a → a
twice f x = f (f x)
```

Higher-Order Functions

Common programming idioms can be encoded as functions within the language itself.

Domain specific languages can be defined as collections of higher-order functions.

Algebraic properties of higher-order functions can be used to reason about programs.

Higher-Order Functions

Give me examples from your favorite programming language/library

Higher-Order Functions

Map

```
map :: (a → b) → [a] → [b]
```

```
> map (+1) [1,3,5,7]

[2,4,6,8]
```

Map Function

The map function can be defined in a simple manner using a ***list comprehension***:

```
map f xs = [f x | x ← xs]
```

Alternatively, the map function can also be defined using ***recursion***:

```
map f []     = []
map f (x:xs) = f x : map f xs
```

Filter Function

The higher-order function `filter` selects every element from a list that satisfies a predicate

```
filter :: (a → Bool) → [a] → [a]
```

```
> filter even [1..10]

[2,4,6,8,10]
```

Filter Function

Filter can be defined using a ***list comprehension***:

```
filter p xs = [x | x ← xs, p x]
```

Alternatively, it can be defined using ***recursion***:

```
filter p [] = []
filter p (x:xs)
   | p x       = x : filter p xs
   | otherwise = filter p xs
```

Foldr Function

A number of functions on lists can be defined using the following simple **pattern of recursion**:

```
f []     = v
f (x:xs) = x ⊕ f xs
```

f maps the empty list to some value **v**, and any non-empty list to some function ⊕ applied to its **head** and **f** of its **tail**

Foldr Function

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

product []     = 1
product (x:xs) = x * product xs

and []       = True
and (x:xs)   = x && and xs
```

```
v = 0
⊕ = +

v = 1
⊕ = *

v = True
⊕ = &&
```

Foldr Function

The higher-order library function **foldr** (*fold right*) encapsulates this simple ***pattern of recursion***, with the function ⊕ and the value v as arguments

```
sum = foldr (+) 0

product = foldr (*) 1

or = foldr (||) False

and = foldr (&&) True
```

Foldr Function

It is best to think of **foldr** as simultaneously replacing each `(:)` in a list by a given *function*, and `[]` by a given *value*

```
  sum [1,2,3]
=
  foldr (+) 0 [1,2,3]
=
  foldr (+) 0 (1:(2:(3:[])))
=
  1+(2+(3+0))
=
  6
```

replace each (:) by (+) and [] by 0

Foldr Function

It is best to think of **foldr** as simultaneously replacing each `(:)` in a list
by a given *function*, and `[]` by a given *value*

```
  product [1,2,3]
= foldr (*) 1 [1,2,3]
= foldr (*) 1 (1:(2:(3:[])))
= 1*(2*(3*1))
= 6
```

replace each (:)
by (*) and [] by 1

Foldr Function

```haskell
length :: [a] → Int
length []      = 0
length (_:xs) = 1 + length xs
```

```haskell
length = foldr (\ _ n → 1+n) 0
```

Foldr Function

```
reverse :: [a] → [a]
reverse []       = []
reverse (x:xs) = reverse xs ++ [x]
```

```
reverse = foldr (\x xs → xs ++ [x]) []
```

Foldr Function

Some recursive functions on lists, such as `sum`, are simpler to define using `foldr`.

Properties of functions defined using `foldr` can be proved using algebraic properties of `foldr`

Advanced program optimizations can be simpler if `foldr` is used in place of explicit recursion

Function Composition

The library function `(.)` returns the composition of two functions as a single function

```
(.) :: (b → c) → (a → b) → (a → c)
f . g = λx → f (g x)
```

Eg.

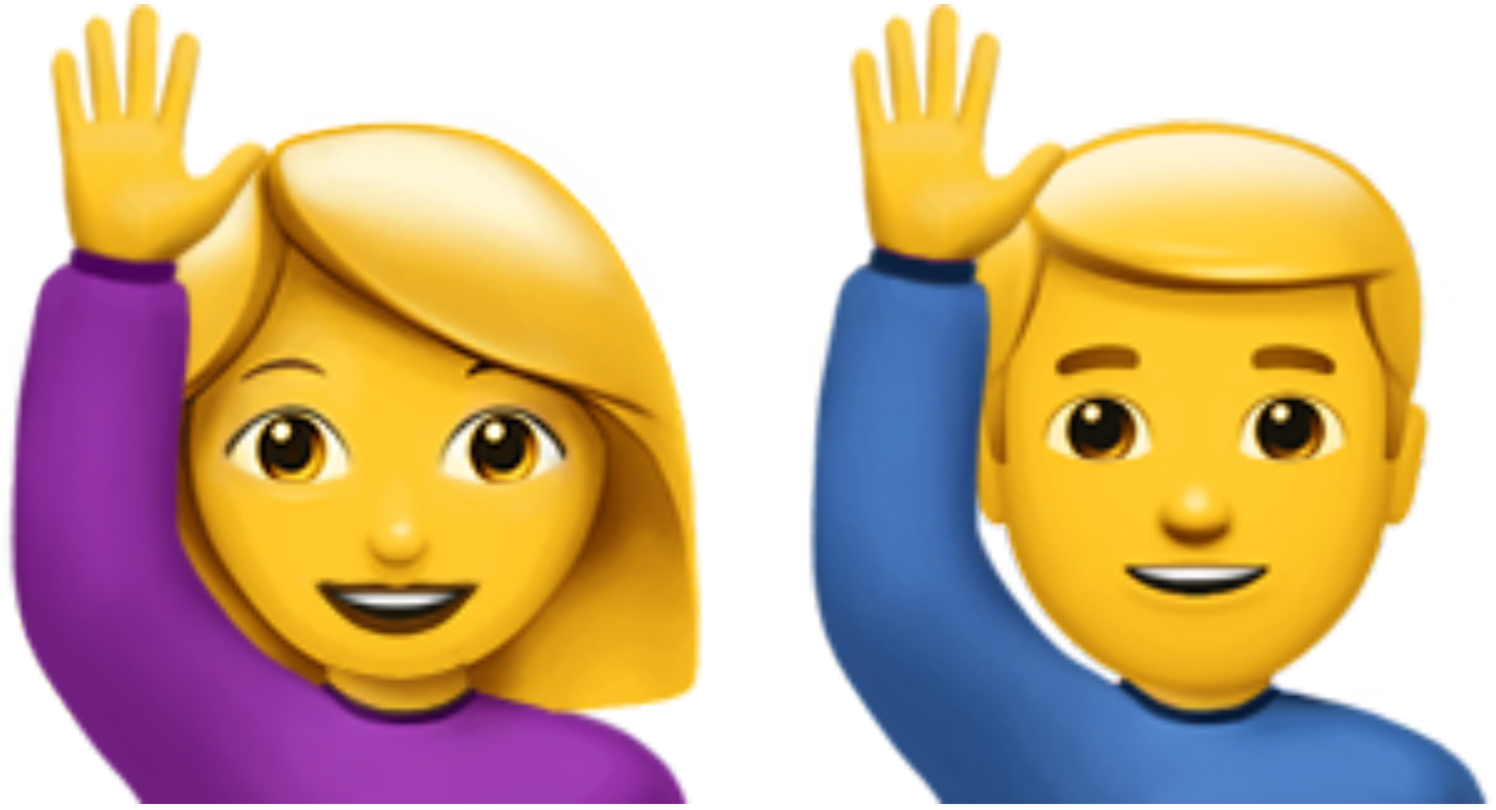
```
filter::(a -> Bool) -> [a] -> [a]
length::[a] -> Int
=>
let e = length . filter (\x -> odd x) xs
e::Int
```

Functional Patterns in C++

Problem:

Counting adjacent repeated values in a sequence.

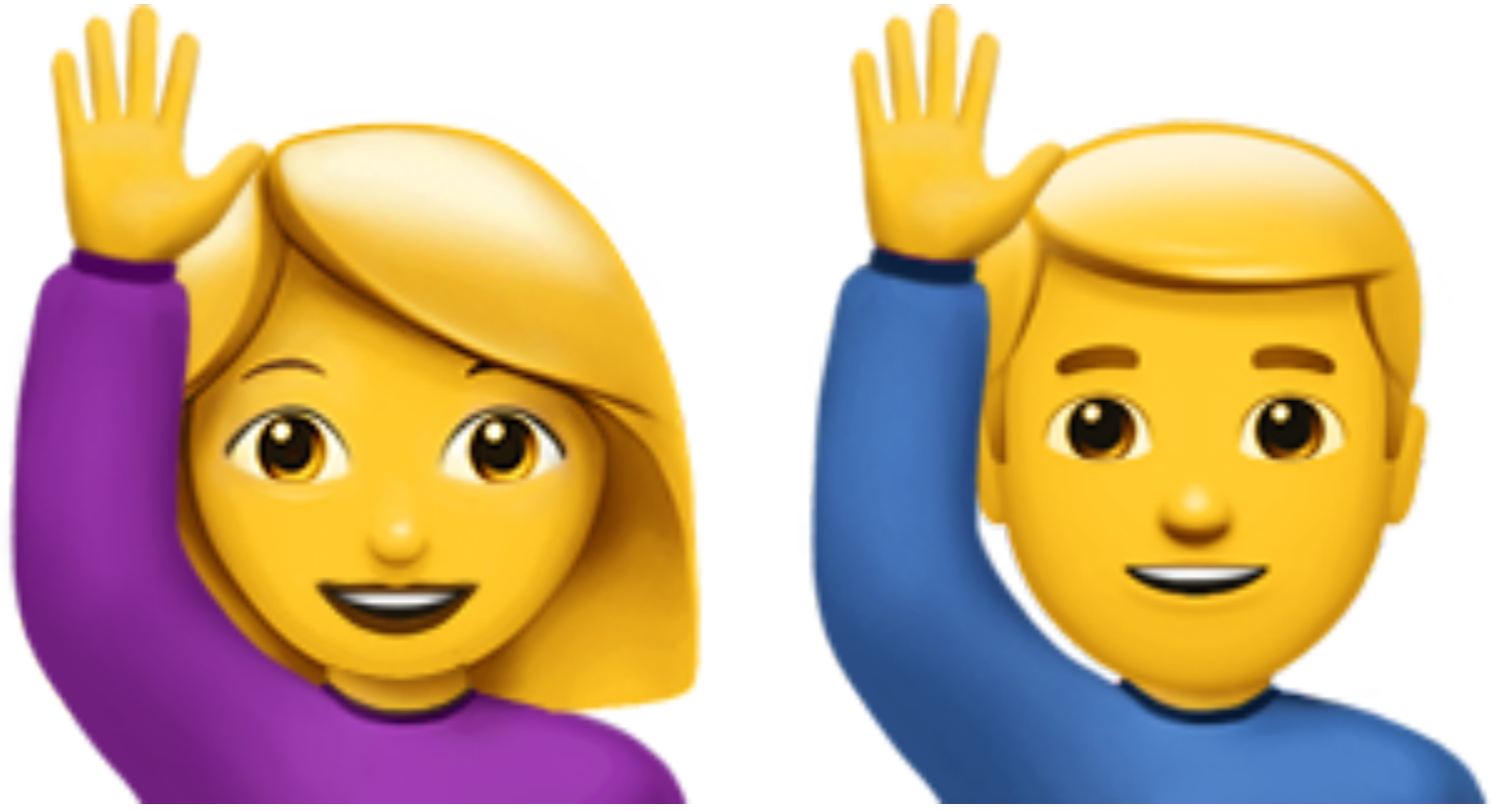
How many of you solved this textbook exercise before ?
(in any programming language)



C++ Counting adjacent repeated values in a sequence

`{ 5, 8, 8, 2, 1, 1, 9, 4, 4, 7 }`

Who wants to try it now ?



C++ **Counting adjacent repeated values in a sequence**

Visual hint:

```
  { 5, 8, 8, 2, 1, 1, 9, 4, 4, [7] }

{ 5, 8, 8, 2, 1, 1, 9, 4, 4, 7 }

(==)

  { 0, 1, 0, 0, 1, 0, 0, 1, 0 }      (+) ➡️ 3
```

C++ Counting adjacent repeated values in a sequence

Let me guess... a bunch of `for` loops, right ?

How about something shorter ?

An STL `algorithm` maybe ?

C++ Counting adjacent repeated values in a sequence

```
template<class InputIt1, class InputIt2,
         class T,
         class BinaryOperation1, class BinaryOperation2>
T inner_product(InputIt1 first1, InputIt1 last1,
                InputIt2 first2, T init,
                BinaryOperation1 op1  // "sum" function
                BinaryOperation2 op2) // "product" function
{
  while (first1 != last1)
  {
     init = op1(init, op2(*first1, *first2));
     ++first1;
     ++first2;
  }
  return init;
}
```

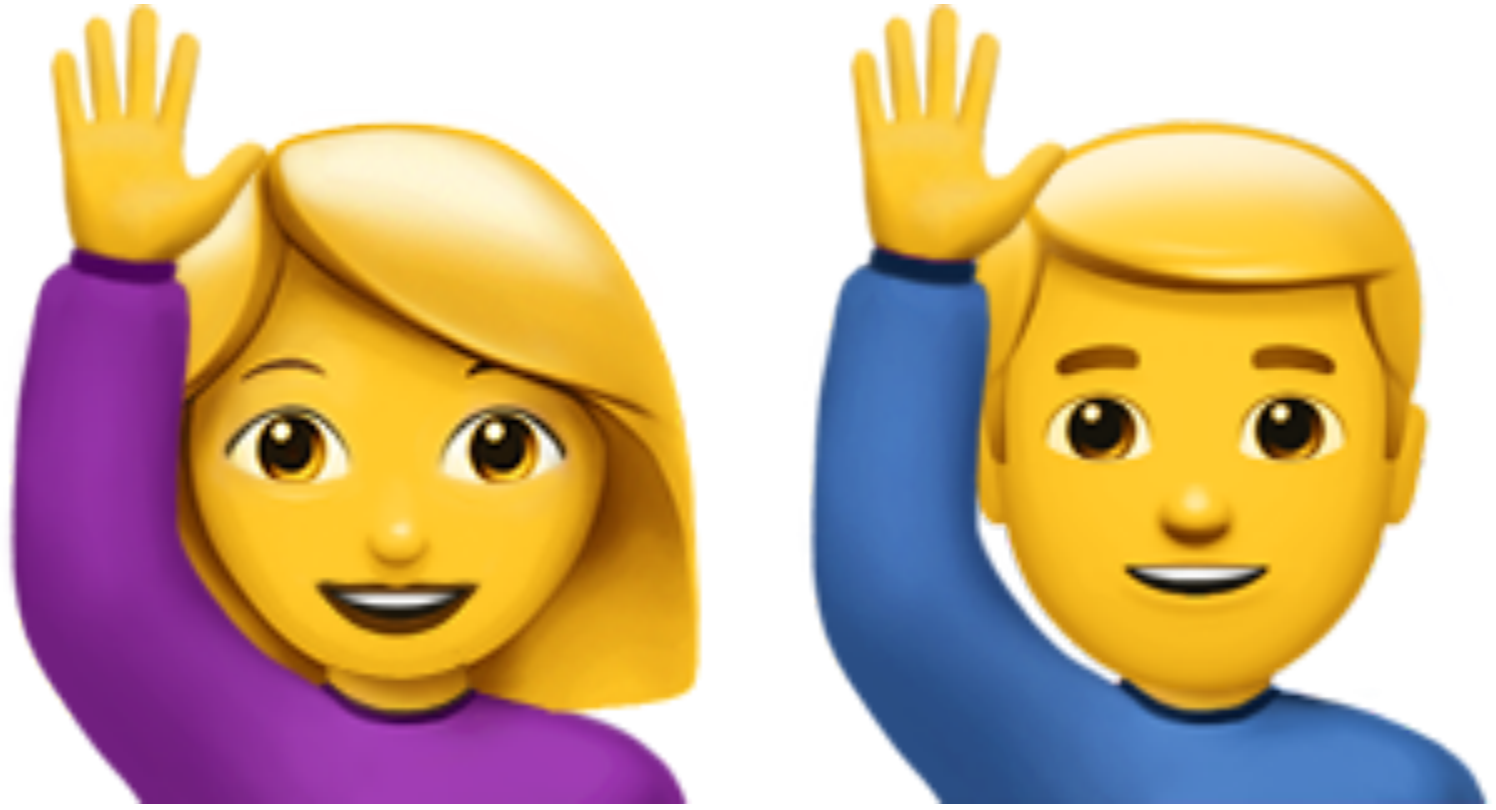
https://en.cppreference.com/w/cpp/algorithm/inner_product

C++ Counting adjacent repeated values in a sequence

```
template <typename T>
int count_adj_equals(const T & xs) // requires non-empty range
{
  return std::inner_product(
    std::cbegin(xs), std::cend(xs) - 1, // to penultimate elem
    std::cbegin(xs) + 1,                // collection tail
    0,
    std::plus{},
    std::equal_to{}); // yields boolean => 0 or 1
}
```

C++ Counting adjacent repeated values in a sequence

If you found that piece of code in a code-base,
would you **understand** what it does* ?



*** without my cool diagram & animation**

Counting adjacent repeated values in a sequence

Let's go back to Haskell for a few minutes...

Counting adjacent repeated values in a sequence

Visual hint:

```
[ 5, 8, 8, 2, 1, 1, 9, 4, 4, 7 ]

  [ 5, 8, 8, 2, 1, 1, 9, 4, 4, 7 ]

(-)

[ -3, 0, 6, 1, 0, -8, 5, 0, -3 ]  (==0) ➡️ 3
```

Counting adjacent repeated values in a sequence

```
let xs = [ 5, 8, 8, 2, 1, 1, 9, 4, 4, 7 ]

count_if f = length . filter f
adj_diff = mapAdjacent (-)
count_adj_equals = count_if (==0) . adj_diff

> count_adj_equals xs
3
```

That's it !

Counting adjacent repeated values in a sequence

Let's break it down:

```
// C++
[](auto a, auto b) { return a + b; }
plus{}

[](auto e) ->bool { return e == 1; }
```

```
// Haskell
(\a b -> a + b)
(+)

(\e -> e == 1)
(==1)
```

Lambdas & sections

Counting adjacent repeated values in a sequence

Let's break it down:

```
length::[a] -> Int
filter::(a->Bool) -> [a] -> [a]

=>

count_if::(a->Bool) -> [a] -> Int
count_if f = length . filter f
```

Counting adjacent repeated values in a sequence

Let's break it down:

```
mapAdjacent::(a->a->b) -> [a] -> [b]
mapAdjacent _ [] = []
mapAdjacent f xs = zipWith f xs (tail xs)

      (-)::a -> a -> a
      adj_diff = mapAdjacent (-)

      =>

      adj_diff::[a] -> [a]
```

Counting adjacent repeated values in a sequence

Let's break it down:

```
(==0)::a -> Bool
count_if::(a->Bool) -> [a] -> Int
adj_diff::[a] -> [a]
```

```
count_adj_equals::[a] -> Int
count_adj_equals = count_if (==0) . adj_diff
```

Counting adjacent repeated values in a sequence

Let's break it down:

```
let xs = [ 5, 8, 8, 2, 1, 1, 9, 4, 4, 7 ]

> let ds = adj_diff xs
[ -3, 0, 6, 1, 0, -8, 5, 0, -3 ]

> count_if(==0) ds
3
```

Counting adjacent repeated values in a sequence

The algorithm

```
count_if f = length . filter f
adj_diff = mapAdjacent (-)
count_adj_equals = count_if (==0) . adj_diff
```

C++ Counting adjacent repeated values in a sequence

Back to modern C++

```
template <typename T>
int count_adj_equals(const T & xs)
{
  return accumulate(0,
                    zip(xs, tail(xs)) | transform(equal_to{}));
}
```

C++20 Ranges

Homework

1986:
Donald Knuth was asked to implement a program for the *"Programming pearls"* column in the **Communications of ACM** journal.

The task:
Read a file of text, determine the n most frequently used words, and print out a sorted list of those words along with their frequencies.

His solution written in Pascal was 10 pages long.

Homework

Response by Doug McIlroy was a 6-line shell script that did the same:

```
tr -cs A-Za-z '\n' |
    tr A-Z a-z |
    sort |
    uniq -c |
    sort -rn |
    sed ${1}q
```

Homework

Taking inspiration from Doug McIlroy's UNIX shell script,

write a C++ or Haskell algorithm, that solves the same problem: word frequencies

It's all about pipelines !

C++ 20 Ranges

Print only the **even** elements of a range in **reverse** order:

```
std::for_each(
   std::crbegin(v), std::crend(v),
   [](auto const i) {
      if(is_even(i))
         cout << i;
   });
```

```
for (auto const i : v
                  | rv::reverse
                  | rv::filter(is_even))
{
   cout << i;
}
```

C++ 20 Ranges

Skip the first **2** elements of the range and print only the **even** numbers of the **next 3** in the range:

```
auto it = std::cbegin(v);
std::advance(it, 2);
auto ix = 0;
while (it != cend(v) && ix++ < 3)
{
   if (is_even(*it))
      cout << (*it);
   it++;
}
```

```
for (auto const i : v
                  | rv::drop(2)
                  | rv::take(3)
                  | rv::filter(is_even))
{
   cout << i;
}
```

C++ 20 Ranges

Modify an *unsorted* range so that it retains only the **unique** values but in **reverse** order.

```
vector<int> v{ 21, 1, 3, 8, 13, 1, 5, 2 };

std::sort(std::begin(v), std::end(v));

v.erase(
   std::unique(std::begin(v), std::end(v)),
   std::end(v));

std::reverse(std::begin(v), std::end(v));
```

```
vector<int> v{ 21, 1, 3, 8, 13,
1, 5, 2 };

v = std::move(v) |
    ra::sort |
    ra::unique |
    ra::reverse;
```

C++ 20 Ranges

Create a range of ***strings*** containing the **last 3** numbers **divisible to 7** in the range **[101, 200]**, in **reverse** order.

```
vector<std::string> v;

for (int n = 200, count = 0;
     n >= 101 && count < 3; --n)
{
   if (n % 7 == 0)
   {
      v.push_back(to_string(n));
      count++;
   }
}
```

```
auto v = rs::iota_view(101, 201)
       | rv::reverse
       | rv::filter([](auto v) { return v%7==0; })
       | rv::transform(to_string)
       | rv::take(3)
       | rs::to_vector;
```

C++ 20 Ranges

Until the new ISO standard lands in a compiler near you…

Eric Niebler's implementation of the Ranges library is available here:
https://github.com/ericniebler/range-v3

It works will **Clang** 3.6.2 or later, **gcc** 5.2 or later, and **MSVC** 15.9 or later.

Although the standard namespace for the Ranges library is **std::ranges**,
in this current implementation of the library it is **ranges::v3**

```
namespace rs = ranges::v3;
namespace rv = ranges::v3::view;
namespace ra = ranges::v3::action;
```

Higher-Order Functions

Higher Order Functions for Ordinary C++ Developers

Björn Fahller

```
compose([](auto const& s) { return s == "foo";},
        std::mem_fn(&foo::name))
```

Higher Order Functions – Meeting C++ 2018 © Björn Fahller @bjorn_fahller 1/93

https://github.com/rollbear/lift

https://www.youtube.com/watch?v=qL6zUn7iiLg

Higher-Order Functions

`boost::hof`

https://www.boost.org/doc/libs/develop/libs/hof/doc/html/doc/

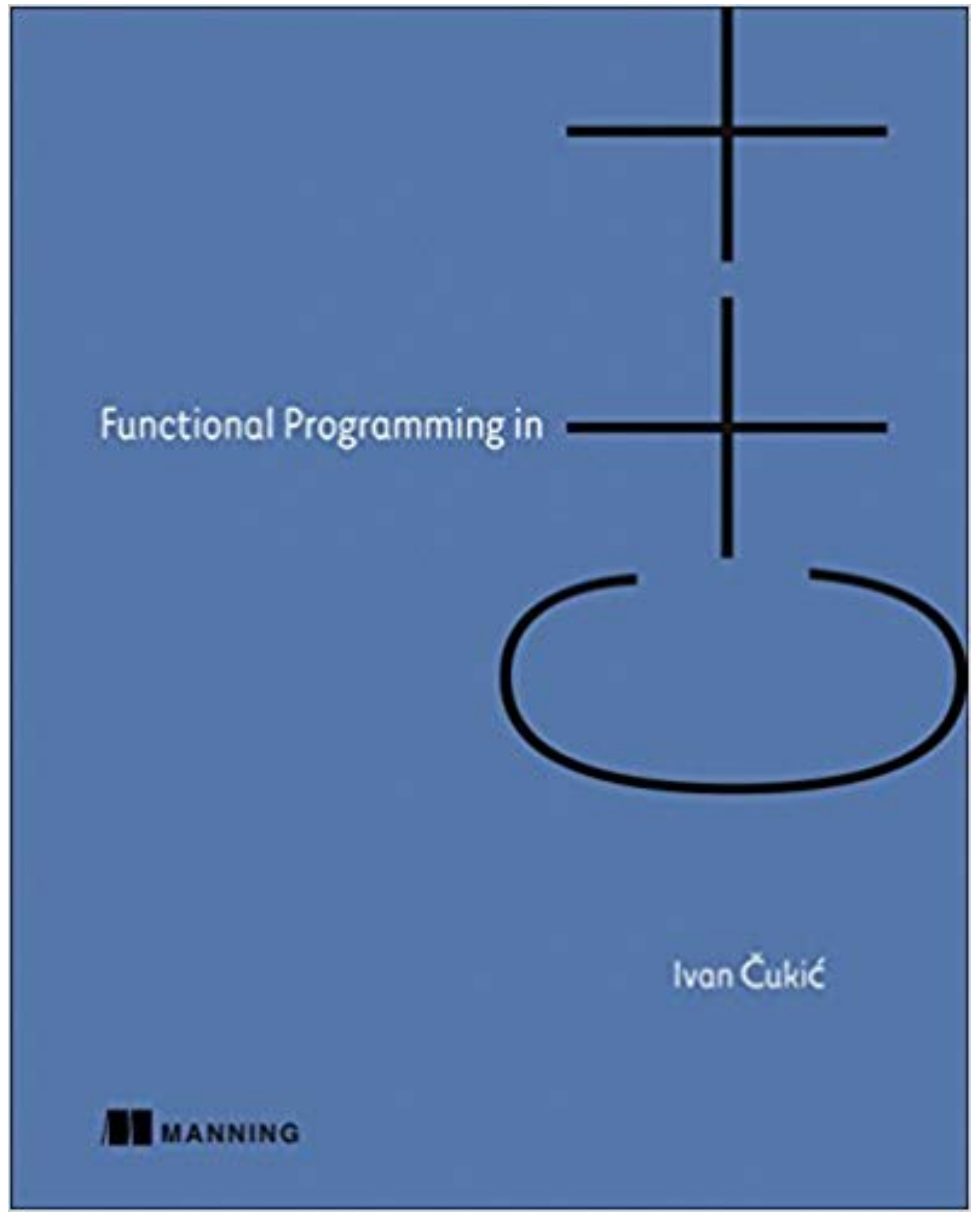
Further Study

“Ranges for distributed and asynchronous systems”
- Ivan Čukić [ACCU 2019]

https://www.youtube.com/watch?v=eelpmWo2fuU

“C++ Algorithms in Haskell and the Haskell Playground”
- Conor Hoekstra [C++Now 2019]

https://www.youtube.com/watch?v=dTO3-1C1-t0



“Functional Programming in C++” - Ivan Čukić

https://www.amazon.com/Functional-Programming-programs-functional-techniques/dp/1617293814

Haskell

ranges

std::optional

C++

algorithms

STL

Maybe | Just

lifting

monoids

lambdas & closures

fold

values types

lazy evaluation

declarative vs imperative

algebraic data types

monads

higher order functions

map

composition

pattern matching

FP

expressions vs statements

pure functions

currying

partial application

category theory

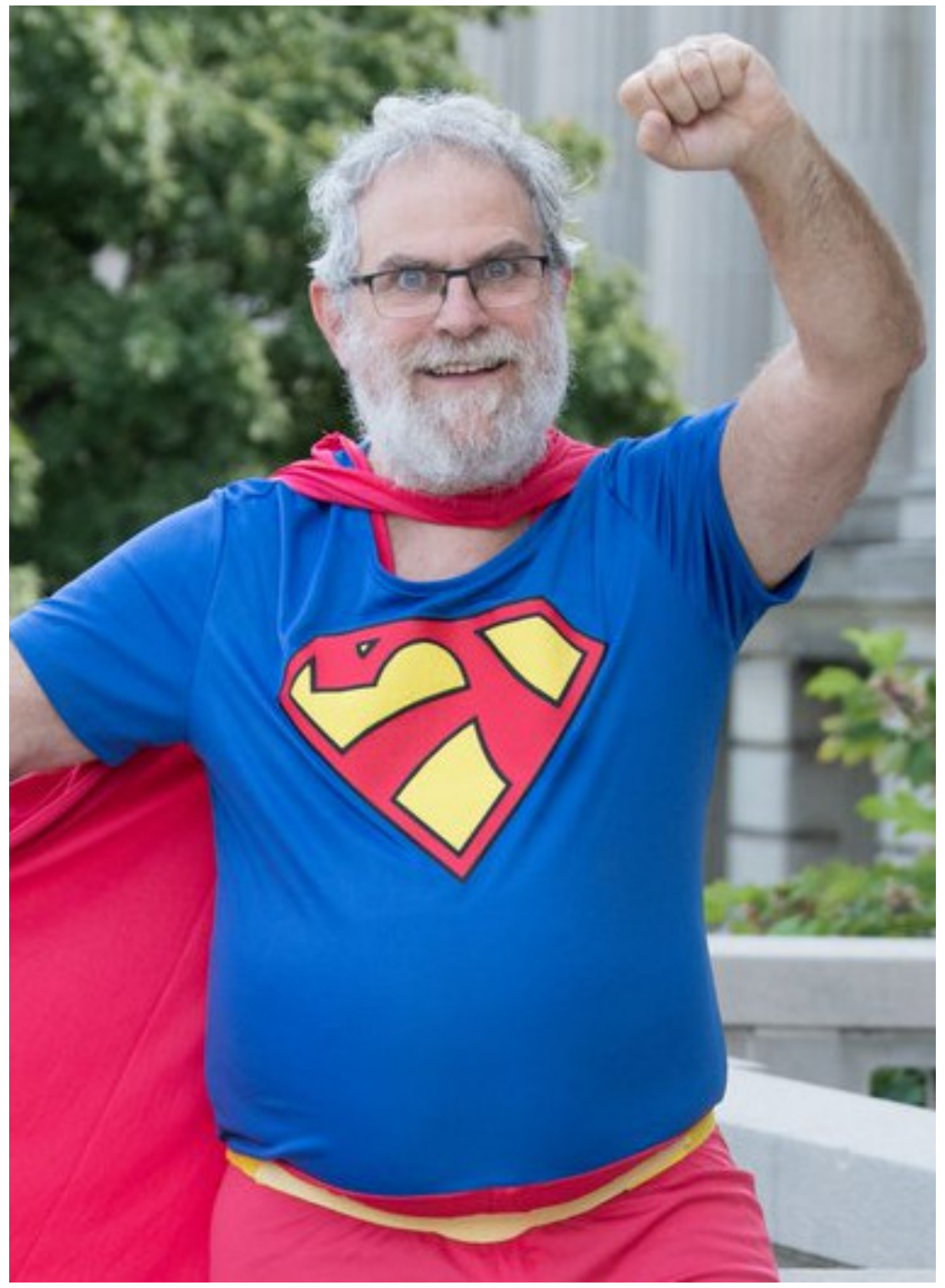
recursion

Historical Background

1990s

Phil Wadler and others develop type classes and monads,
two of the main innovations of Haskell

Takeaway



"Make your code readable.
Pretend the next person who looks at your code is a psychopath and they know where you live."

Phil Wadler

Open4Tech
Summer School 2019
24 iunie - 12 iulie

Curry On Functional Programming

July, 2019
Craiova

CAPHYON

@ciura_victor

Victor Ciura
Technical Lead, Caphyon
www.caphyon.ro