Haskell

Prof. dr. ir. Tom Schrijvers
tom.schrijvers@kuleuven.be

Functional Programming

The λ-calculus

- 1936: untyped variant, Alonzo Church
- 1940: simply typed variant
- basis for study of "computation"
- basis for Functional Programming languages

λ-calculus: 3 forms of expressions

	e.g.
$E ::= \lambda x.E$	$\lambda x.x$
$\mid\ x$	x
$\mid\ E_1\ E_2$	$(\lambda x.x)\ y$

λ-calculus: often extended

	e.g.
E ::= λx.E	λx.x*x
| x	x
| E_1 E_2	(λx.x*x) 7
| n	42
| E_1 * E_2	x*y

λ-calculus: 3 calculation rules

α conversion

$\lambda x.E == \lambda y.[y/x]E$

bv. $\lambda x.x*x == \lambda y.y*y$

β reduction

$(\lambda x.E_1)\ E_2 == [E_2/x]E_1$

bv. $(\lambda x.x*x)\ 7 == 7*7$

η conversion

$\lambda x.(E\ x) == E$

bv. $\lambda x.((\lambda y.y)\ x) == \lambda y.y$

λ-calculus: 3 calculation rules

α conversion

$\lambda x.E == \lambda y.[y/x]E$

bv. $\lambda x.x*x == \lambda y.y*y$

β reduction evaluation

$(\lambda x.E_1)\ E_2 == [E_2/x]E_1$

bv. $(\lambda x.x*x)\ 7 == 7*7$

η conversion

$\lambda x.(E\ x) == E$

bv. $\lambda x.((\lambda y.y)\ x) == \lambda y.y$

Evolution

1936
λ calculus

Alonzo Church

Evolution

Functional Languages

1936
λ calculus

Alonzo Church

Evolution

Functional Languages

1936
λ calculus

Alonzo Church

1958
Lisp

John McCarthy

Evolution

Functional Languages

1936
λ calculus

Alonzo Church

1958
Lisp

John McCarthy

Scheme

Racket

Evolution

Functional Languages

1936
λ calculus

Alonzo Church

1958
Lisp

John McCarthy

Scheme

Racket

1973
ML

Robin Milner

Evolution

Functional Languages

1936
λ calculus

Alonzo Church

1958
Lisp

John McCarthy

Scheme

Racket

1973
ML

Robin Milner

Ocaml

F#

Evolution

Functional Languages

1936
λ calculus

Alonzo Church

1958
Lisp

John McCarthy

Scheme

Racket

1973
ML

Robin Milner

Ocaml

1987
Haskell

Haskell Committee

F#

Evolution

Functional Languages

1936
λ calculus

Alonzo Church

1958
Lisp

John McCarthy

Scheme

Racket

1973
ML

Robin Milner

Ocaml

F#

1987
Haskell

Haskell Committee

TODAY

most influential FP language

Evolution

Functional Languages

1936
λ calculus

Alonzo Church

1958
Lisp

John McCarthy

Scheme

Racket

1973
ML

Robin Milner

Ocaml

F#

1987
Haskell

Haskell Committee

Scala

TODAY

most influential FP language

Evolution

Functional Languages

Mainstream

1936
λ calculus

Alonzo Church

1958
Lisp

John McCarthy

Scheme

Racket

1973
ML

Robin Milner

Ocaml

F#

1987
Haskell

Haskell Committee

Scala

Evolution

Functional Languages

Mainstream

1936
λ calculus

Alonzo Church

1958
Lisp

John McCarthy

Scheme

Racket

1973
ML

Robin Milner

Ocaml

F#

1987
Haskell

Haskell Committee

Scala

2007
C#

Microsoft

Evolution

Functional Languages

Mainstream

1936
λ calculus

Alonzo Church

1958
Lisp

John McCarthy

Scheme

Racket

1973
ML

Robin Milner

Ocaml

F#

1987
Haskell

Haskell Committee

Scala

2007
C#

Microsoft

2011
C++11

ISO International Organization for Standardization

Evolution

Functional Languages

Mainstream

1936
λ calculus

Alonzo Church

1958
Lisp

John McCarthy

Scheme

Racket

1973
ML

Robin Milner

Ocaml

F#

1987
Haskell

Haskell Committee

Scala

2007
C#

Microsoft

2011
C++11

ISO International Organization for Standardization

2014
Java 8

ORACLE

Swift

5 Reasons **You** Should Learn Functional Programming

47 Degrees

1. Easy to maintain
2. Become a better programmer
3. It’s a fun way to code
4. FP has become very popular
5. Much more than a trend

https://bit.ly/389zdMq

Haskell

Haskell B. Curry

- studied combinators
- inspired names of
  3 programming languages
- Curry-Howard isomorphism
- inspired the term
  currying (aka schönfinkeling)

American logician
1900-1982

History

1985 Miranda, predecessor of Haskell

1987 Haskell committee: standardisation of lazy functional programming language

1990 Haskell 1.0

1998 Haskell '98, main milestone

2006 Haskell Prime: Haskell 2010, ...

Only a few of

Haskell's Advantages

- ★ Very high level of abstraction
- ★ Strongly statically typed
- ★ Domain Specific Languages (DSLs)
  - ✦ custom language for problem domain
  - ✦ embedded in Haskell as a library (EDSL)

Haskell's Unofficial Motto

Simon Peyton Jones
Microsoft Research

Haskell's Unofficial Motto

Simon Peyton Jones
Microsoft Research

Haskell's Unofficial Motto

Simon Peyton Jones
Microsoft Research

Haskell's Unofficial Motto

Simon Peyton Jones
Microsoft Research

Haskell's Unofficial Motto

Simon Peyton Jones
Microsoft Research

Haskell's Unofficial Motto

Simon Peyton Jones
Microsoft Research

Java	C	PHP	Ruby	Haskell	as seen by...
					Java fans
					C fans
					PHP fans
					Ruby fans
					Haskell fans

TESLA

TESLA

TESLA MODEL S MODEL 3 MODEL X MODEL Y SOLAR ROOF SOLAR PANELS SHOP TESLA ACCOUNT

Careers

About Careers Contact Legal Investors Suppliers

Haskell Engineer, Data Tools

Job Category	Engineering & Information Technology
Location	Palo Alto, California
Req. ID	50998
Job Type	Full-time

BUILD THE FUTURE

APPLY

Tesla participates in the E-Verify Program

The Role

This team builds design infrastructure and tooling that enables us to rapidly develop software with precision and agility. As a member of this team you will shape the landscape for how Tesla designs future systems. Our vehicles contain many systems, distributed across a large network architecture, working together to accomplish a multitude of vehicle functions. This team's core technology is an infrastructure that describes, manages and analyzes how these systems communicate with each other. This is accomplished through three main functions:

Haskell in Industry

Finance

ABN·AMRO

BARCLAYS CAPITAL

Deutsche Bank

CREDIT SUISSE

Telecom

QUALCOMM

at&t

Alcatel·Lucent

Many Others

Google

intel

The New York Times

Blockchain/
Smart Contracts

Erlang

Haskell

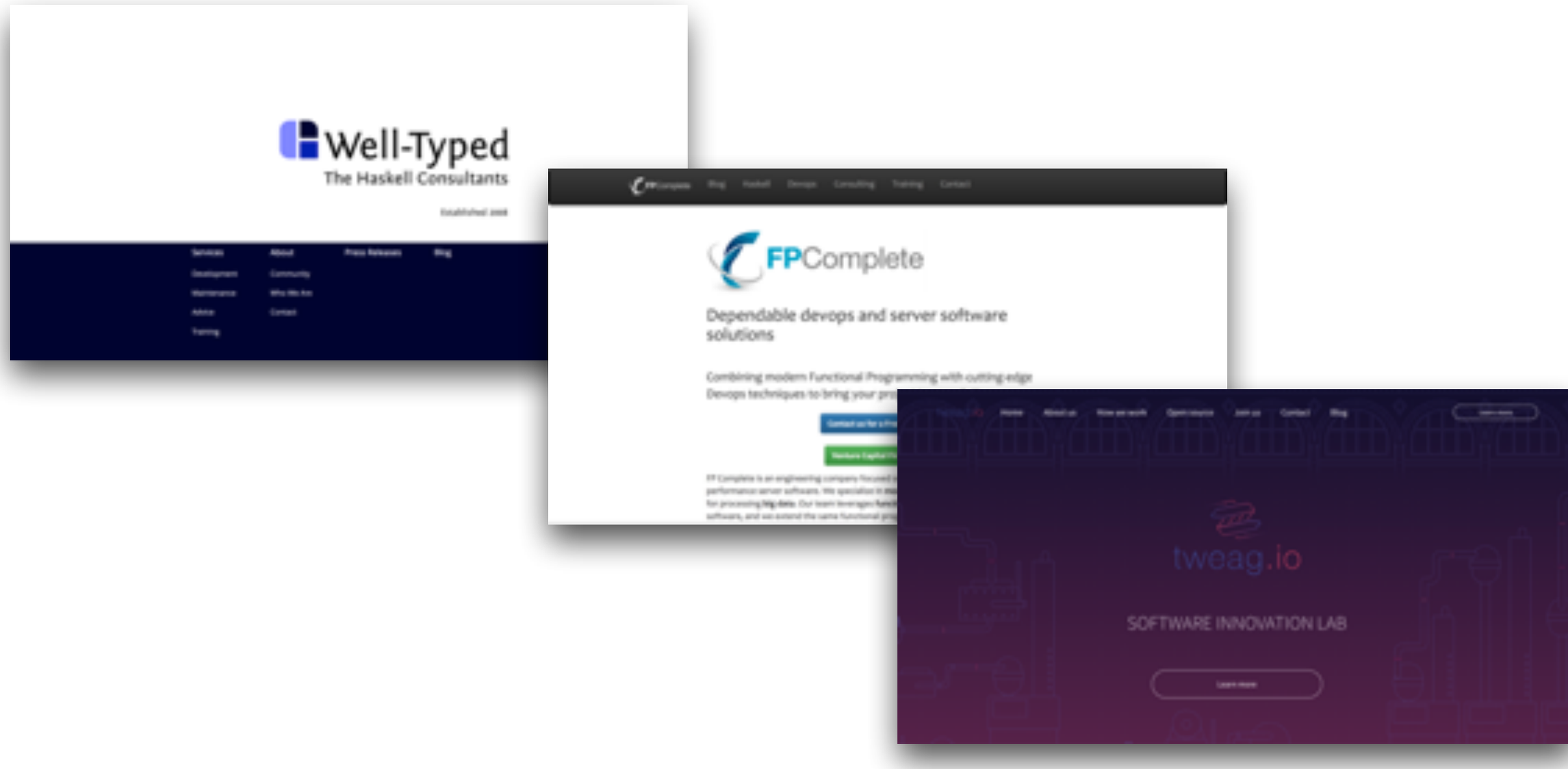
Elm

Haskell

OCaml

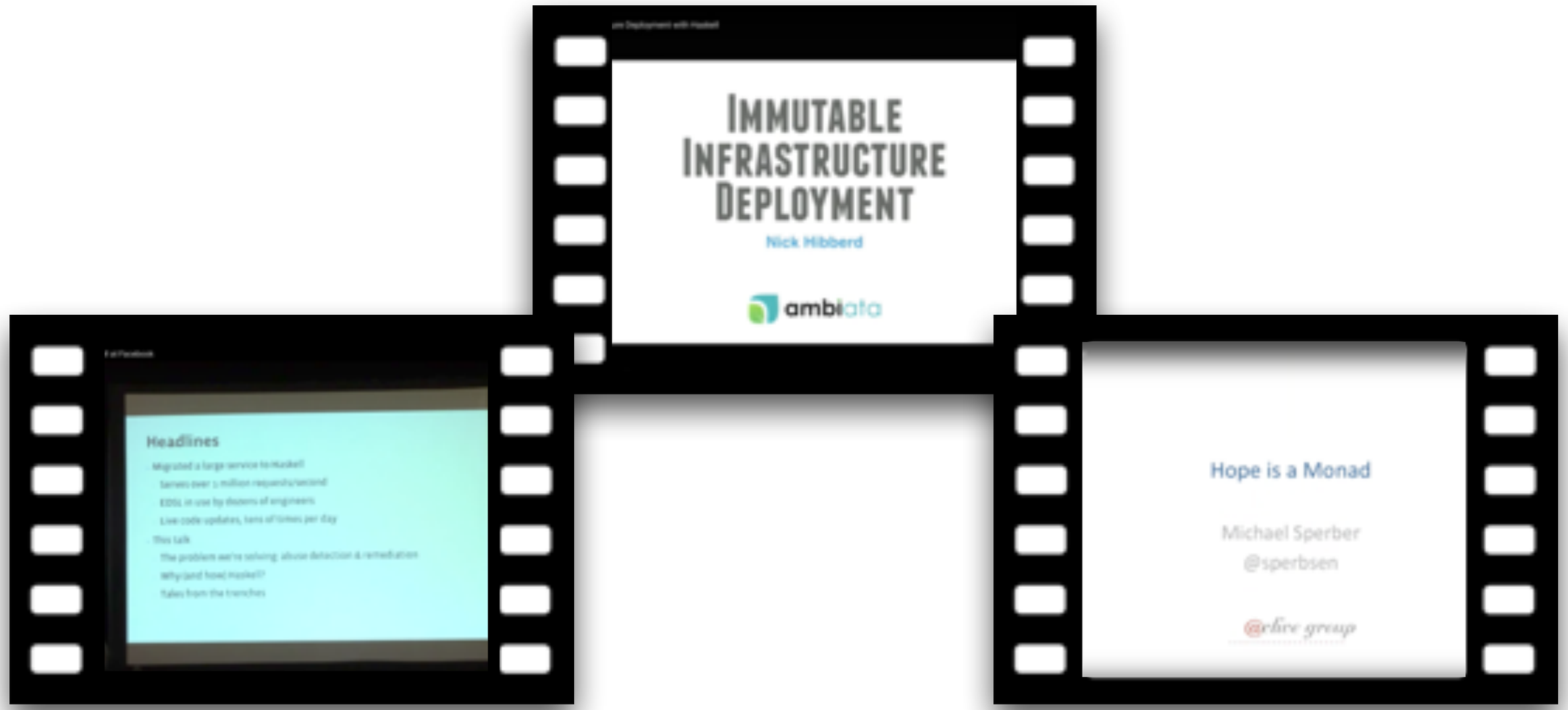
Scala

Haskell Support / Consultants



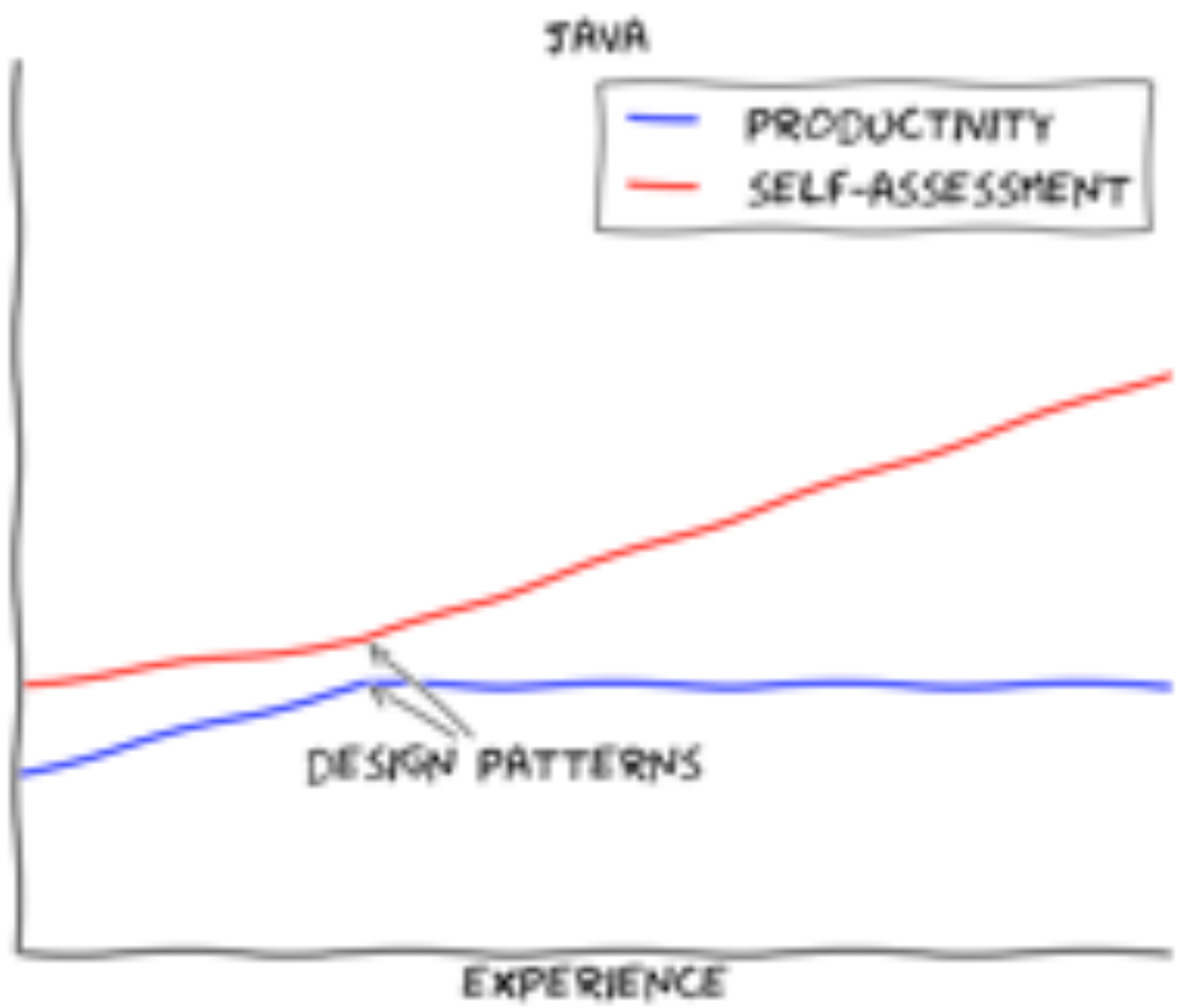
Annual CUFP Workshop

http://cufp.org



Lots of Testimonial Videos

Learning Curve



https://github.com/Dobiasd/articles/blob/master/programming_language_learning_curves.md

Learning Curve

https://github.com/Dobiasd/articles/blob/master/programming_language_learning_curves.md

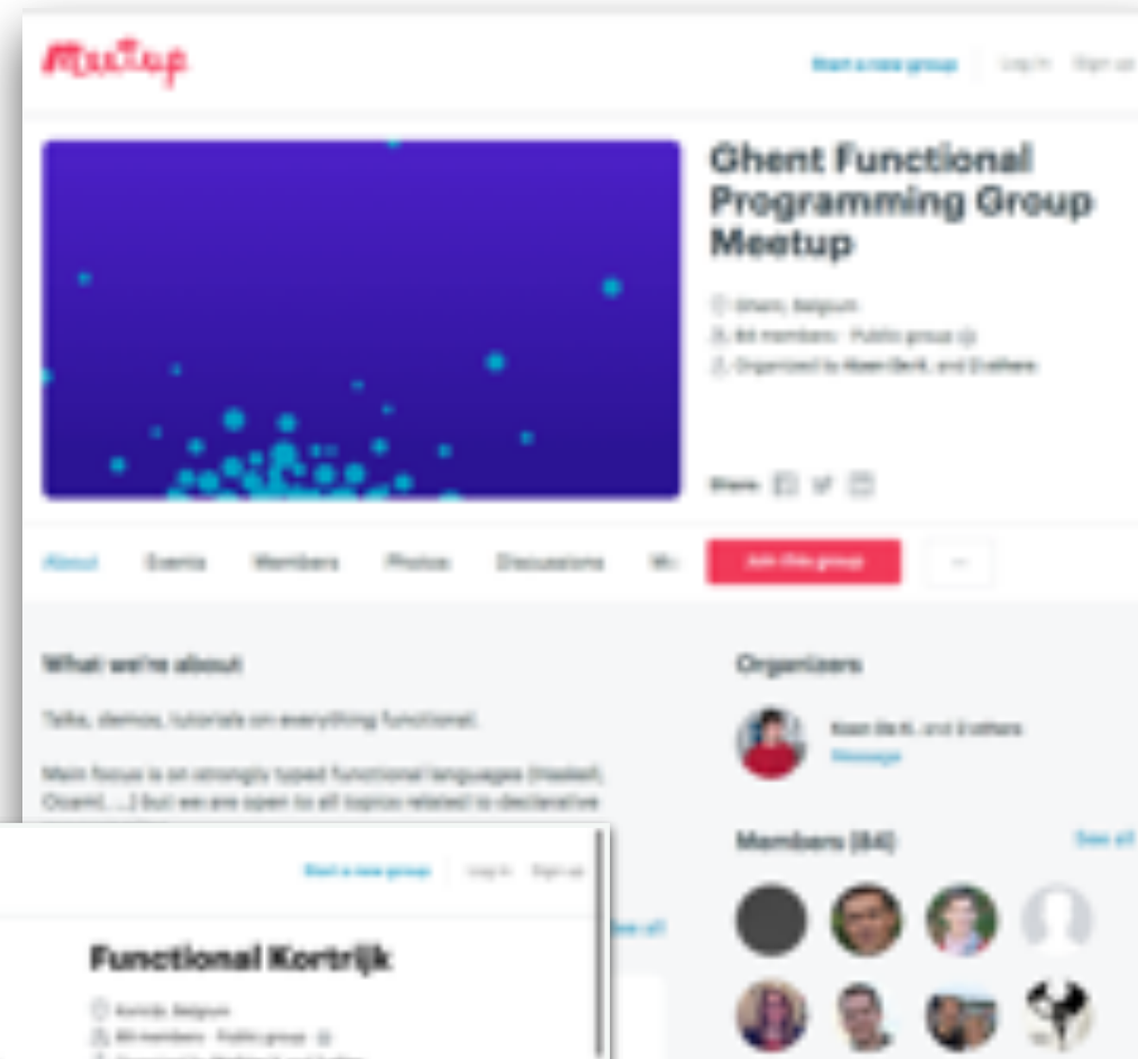
Haskell Tools

- ★ Glasgow Haskell Compiler (GHC), S. Peyton Jones et al., Microsoft Research
- ★ Hackage.haskell.org: 1500+ Haskell libraries & tools, with installation manager cabal
- ★ HPC: program coverage tool
- ★ QuickCheck: property-based testing
- ★ ...

haskell.org/hoogle

FP Meetups

Leuven Functional Programming Meetup
What we're about
Organizers
Past events (3)


Ghent Functional Programming Group Meetup
What we're about
Organizers
Members (84)

Functional Kortrijk
functional kortrijk
What we're about
Organizers

Haskell Jobs

CentralApp

Filled by former master student Brecht Serckx

All jobs → Senior Back-end Engineer

Senior Back-end Engineer

Brussels, Bruxelles-Capitale, Région de Bruxelles Hoofdstedelijk Gewest, Belgium · Engineering

DESCRIPTION

About CentralApp

CentralApp is a successful international Belgian startup. After raising 500k€ from the founders of Keytrade and Skynet, CentralApp has developed a platform that digitally transforms the way restaurants, shops and beauty salons do marketing. This tool is now used by over 1000 businesses in Belgium, France, and other countries.This last year our websites have have been used by over 10 million people, a 400% growth since last year.

We work daily with industry leaders working with local businesses such as Google, BNP Paribas Fortis, Deliveroo, Schwarzkopf, or with Petit Futé.

- Writing Scala/Haskell

piesync

← Job Openings

Platform Engineer (Haskell)

Development · Gent, Oost-Vlaanderen

PieSync (https://www.piesync.com) is a growing startup with the sole purpose of increasing productivity of companies worldwide by connecting all major SaaS applica

As Platform Engineer you are working on our platform, which powers a seamless, re time 2-way sync solution. You will implement complex new features, improve the robustness and scalability of our platform and work on analysis and monitoring tools cooperation with dashboard engineers, you are working on the dashboard/platform interface and API's.

Things you might work on

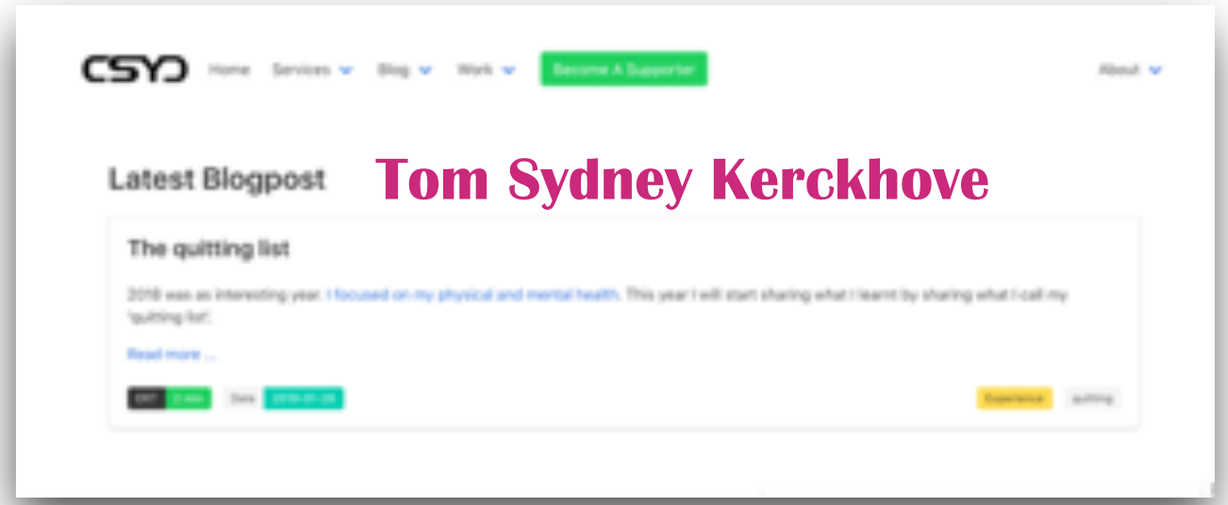
The beating heart of PieSync is a fleet of workers that fetch modified data from our customer's cloud apps, matching and aligning it with data in other apps. This synchronization process is configured using a flexible, composable DSL which is interpreted by our core sync engine.

Writing this in Haskell provides us with huge benefits. The combination of type safety an extensive automated test suite allows us to release sooner and with more confide Using a hexl-like structure we can completely isolate synchronization logic from oper execution, so our sync engine is essentially pure code.

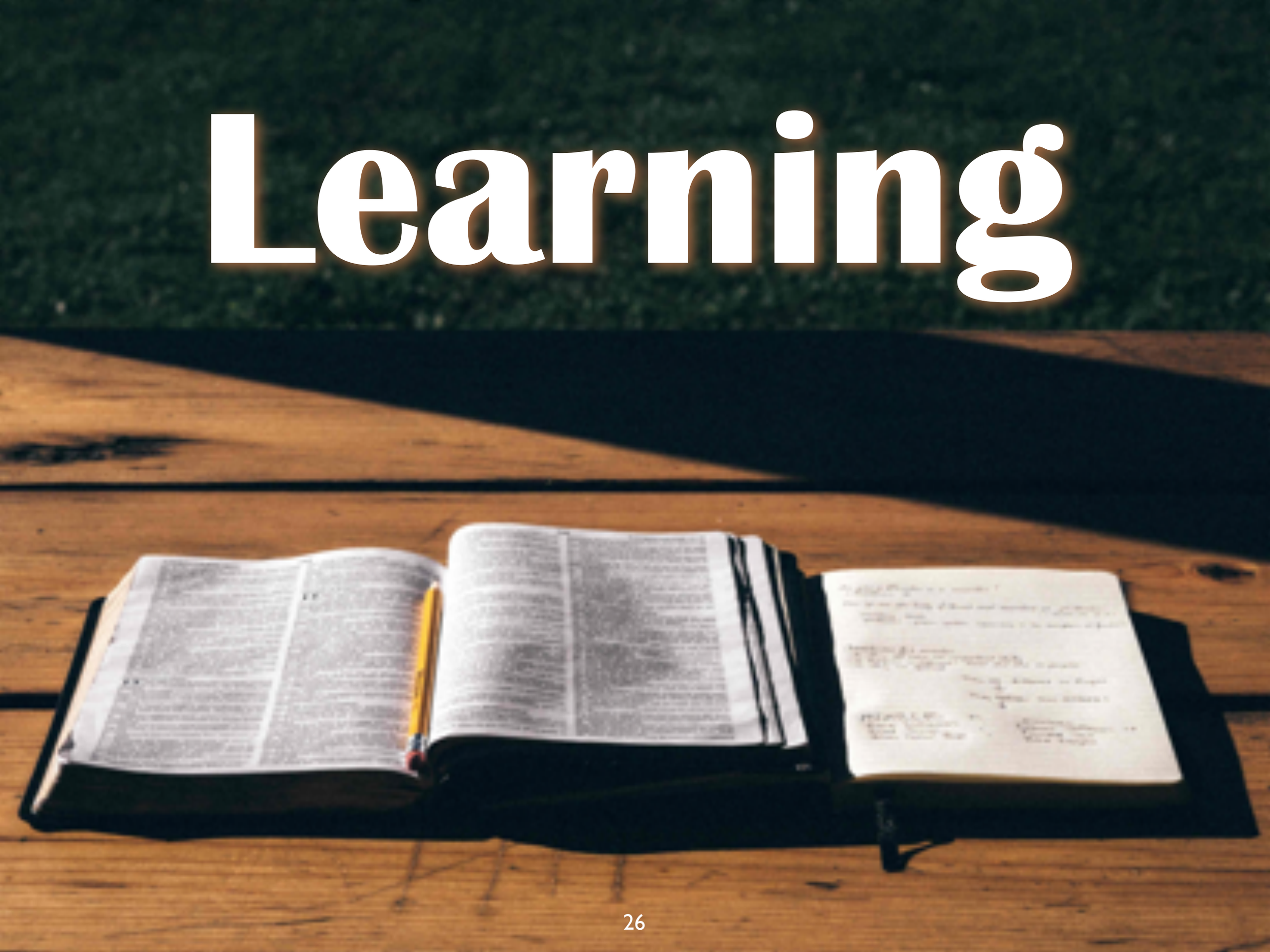
But we're not done yet! Some of the things we want (you) to work on:

- Improve the way resources are indexed and matched.
- Offer our customers more clever ways to sync data by extending our DSL and h think about how these new platform powers can be controlled in the user interfa
- Extend the engine to allow the integrations team to connect a broader range of to the platform, by inventing generic solutions to common API limitations.

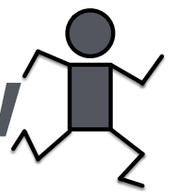
Professional Haskellers



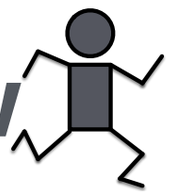
Learning



SDT 2019-20: January

	Pass	Fail	No Show 
Prolog	63 %	37 %	
Haskell	80 %	20 %	
Total	76 %	24 %	14%

DL 2019-20: January

	Pass	Fail	No Show 
Prolog	**40 %**	**60 %**	
Haskell	**60 %**	**40 %**	
Total	**45 %**	**55 %**	**13%**

How to Learn Programming?

How to Learn Programming?

1. Practice

How to Learn Programming?

1. Practice

2. Practice

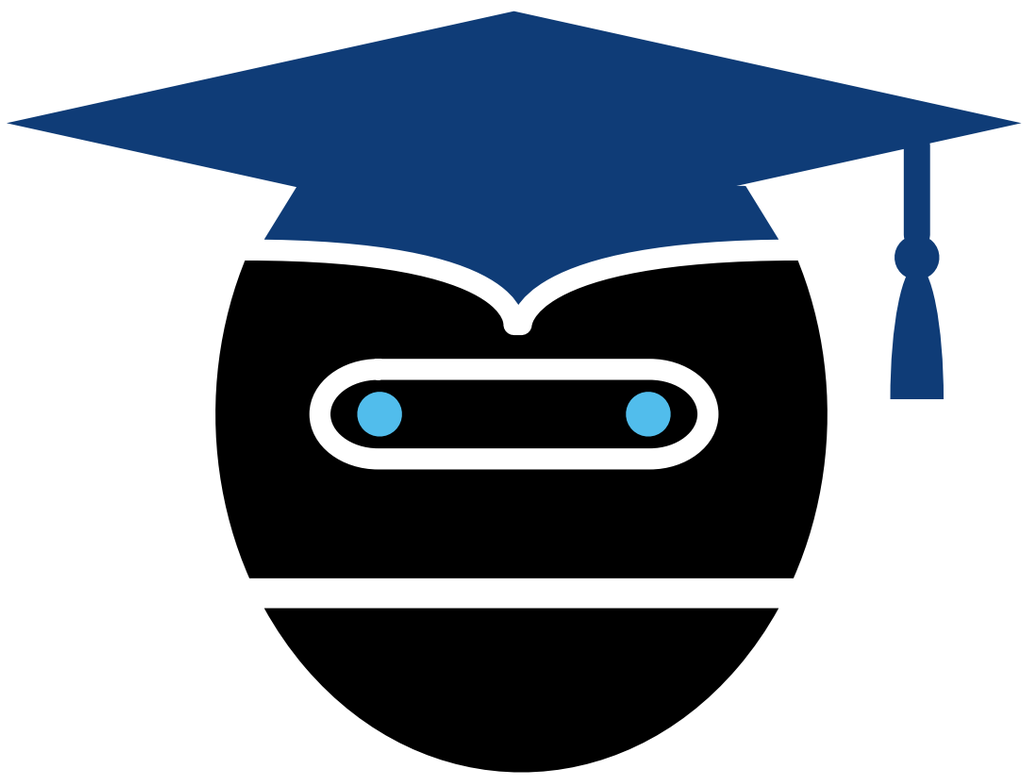
How to Learn Programming?

1. Practice

2. Practice

3. Practice some more

How to Practice?



http://esystant.be

NEW

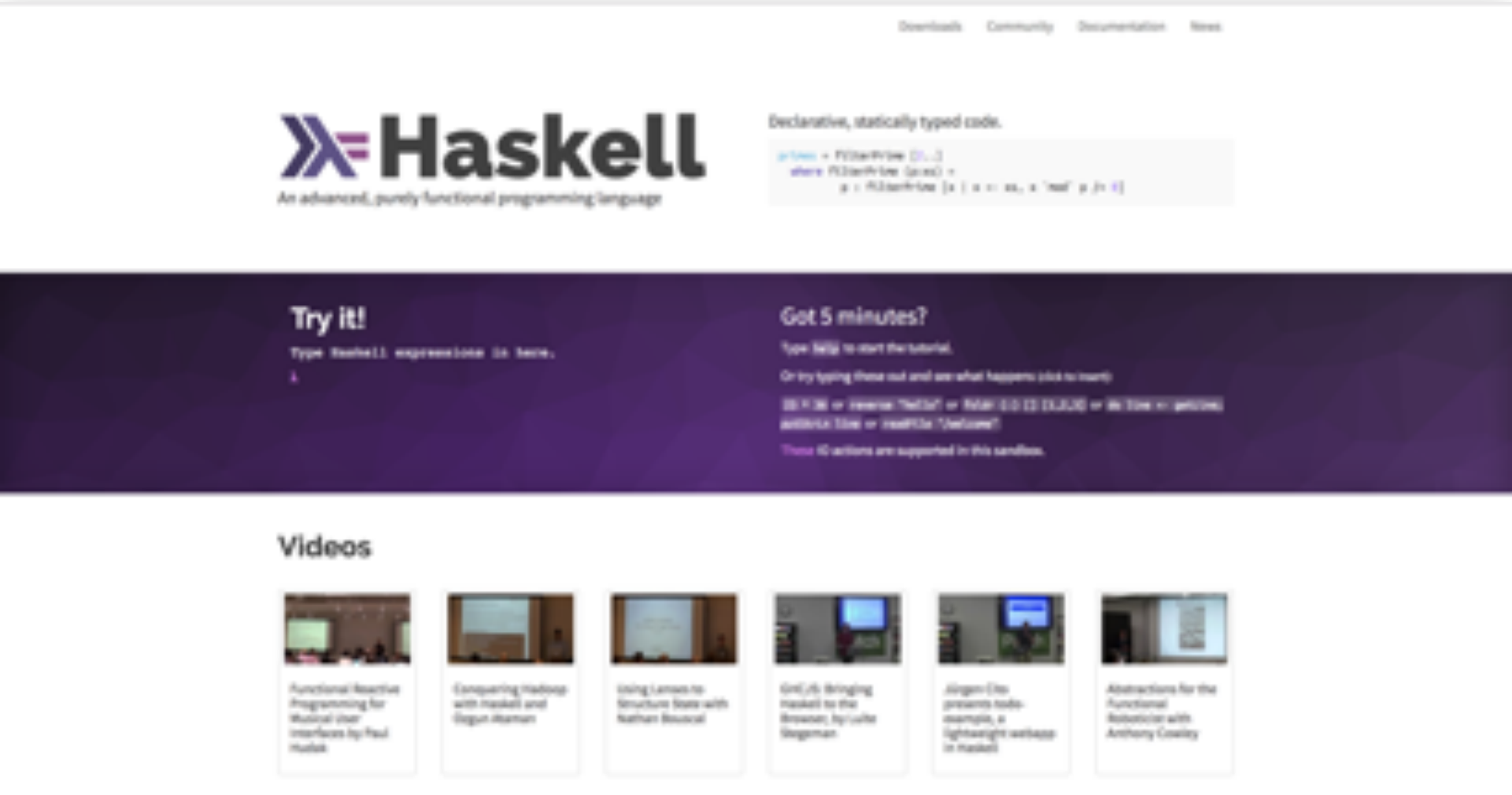
Coaching

Use science to pass this course

"…**class attendance** [is] a better predictor of college grades than any other known predictor of academic performance…"

Class Attendance in College: A Meta-Analytic Review of the Relationship of Class Attendance With Grades and Student Characteristics. Marcus Credé, Sylvia G. Roch, and Urszula M. Kieszczynka. Review of Educational Research, June 2010, Vol. 80, No. 2, pp. 272–295

haskell.org


Downloads
Community
Documentation
News
Haskell
An advanced, purely functional programming language
Declarative, statically typed code.
Try it!
Type Haskell expressions in here.
Got 5 minutes?
Videos

Free Online Book

Free Online Book

Real World Haskell

by Bryan O'Sullivan, Don Stewart, and John Goerzen

Welcome to Real World Haskell

This is the online home of the book "Real World Haskell". It is published by O'Reilly Media. The first edition was released in November 2008.

We make the content freely available online. If you like it, please buy a copy.

For news updates, please visit our blog.

Buy online

For your convenience, we have included links to the book through sellers in several countries.

- O'Reilly (USA, UK)
- Powell's Books (USA)
- Amazon (USA)
- Amazon (Deutschland)
- Amazon (Canada)
- Amazon (UK)

What's that creature on the cover?

Bonus Slides

Summary of Basic Language Features

In the begining there was only...

In the begining there was only...

the function

My 1st function

Definition of function f

Math

$$f(x) = x + 1$$

My 1st function

Definition of function f

Math ⟷ Haskell

$f(x) = x + 1$

My 1st function

Definition of function f

Math		Haskell
$f(x) = x + 1$	$\longleftrightarrow$	`f x = x + 1`

Apply the Function

```
> f 0
1
```

Apply the Function

```
> f 0
1
```

(versus f(0) in math)

Apply the Function

```
> f 0
1

> f 2
3
```

(versus f(0) in math)

Repeated Application

```
> f 0
1
```

Repeated Application

```
> f 0
1
> f (f 0)
2
```

Repeated Application

```
> f 0
1
> f (f 0)
2
> f (f (f 0))
3
```

Multiple Parameters

```
g x y = x + 2 * y
```

Multiple Parameters

```
g x y = x + 2 * y

> g 1 2

5
```

Multiple Parameters

```
g x y = x + 2 * y
```

```
> g 1 2
5

> g 1 (f 2)
7
```

Int

Int

Values

```
0, -1, 42, 137
```

Int

Values

```
0, -1, 42, 137
```

```
> 1 + 2 * 3 - 4

3
```

Int

Values

`0, -1, 42, 137`

```
> 1 + 2 * 3 - 4
3
```

Predefined functions

`mod, div, ...`

Int

Values

`0, -1, 42, 137`

```
> 1 + 2 * 3 - 4
3
```

Predefined functions

`mod, div, ...`

```
> div 7 3
2
> mod 7 3
1
```

Float

Float

Values

```
0.0, -1.0, 3.14159, ...
```

Float

Values

```
0.0, -1.0, 3.14159, ...
```

Predefined functions

```
+, -, *, /, pi, …
```

Bool

Bool

Values

`True, False`

Bool

Values

```
True, False
```

Predefined functions

```
&&, ||, not
>, <, …
```

Bool

Values

```
True, False
```

Predefined functions

```
&&, ||, not
>, <, …
```

```
> 1 > 2
False

> 2 > 1
True
```

Bool

User-defined function

```
passed p = p > 9
```

Bool

User-defined function

```
passed p = p > 9

> passed 10

True
```

Bool

User-defined function

```
passed p = p > 9

> passed 10
True

> passed 8
False
```

If-then-else

```
rate a =
  if a > 2500000
    then 1.00
    else 1.40
```

If-then-else

```
rate a =
  if a > 2500000
     then 1.00
     else if a > 500000
             then 1.40
             else 1.90
```

Guards

```
rate a
    | a > 2500000   = 1.00
    | a >  500000   = 1.40
    | otherwise     = 1.90
```

Lists

Lists

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

Lists

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

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

Lists

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

> head [1,2,3]
1

> null [1]
False
```

Lists

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

> head [1,2,3]
1

> null [1]
False

> null []
True
```

Primitive List Syntax

```
[ ]          -- empty list
x : xs       -- stick element in front
```

Primitive List Syntax

```
[ ]          -- empty list
x : xs       -- stick element in front
```

```
[1,2,3] == 1 : 2 : 3 : []
```

Producing Lists

```
fromto l u
  | l > u
  = []
  | otherwise
  = l : fromto (l+1) u
```

Producing Lists

```
fromto l u
  | l > u
  = []
  | otherwise
  = l : fromto (l+1) u
```

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

Consuming Lists

```
add l
  | null l
  = 0
  | otherwise
  = head l + add (tail l)
```

```
> add [1,2,3]
6
```

Pattern Matching

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

Pattern Matching

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

typical pattern:
structural recursion

Types

Types

Static typing

Types

Static typing for free

Types

Static typing for free

Powerful type inference

- Hindley-Milner's algorithm
- writing type signatures is optional

(Type) Signatures

```
f :: Int → Int
f x = x + 1
```

(Type) Signatures

```
f :: Int → Int
f x = x + 1

g :: Int → Int → Int
g x y = ...
```

(Type) Signatures

```
f :: Int → Int
f x = x + 1

g :: Int → Int → Int
g x y = ...

passed :: Int → Bool
passed p = ...
```

User-Defined Types

Algebraic DataTypes (ADTs)

```
data Light = Red
           | Orange
           | Green
```

User-Defined Types

Algebraic DataTypes (ADTs)

```
data Light = Red
           | Orange
           | Green
```

type constructor

User-Defined Types

Algebraic DataTypes (ADTs)

```
data Light = Red
           | Orange
           | Green
```

type constructor

data constructors

User-Defined Types

Algebraic DataTypes (ADTs)

aka disjoint unions

aka sum of product types

```
data Light = Red
           | Orange
           | Green
```

type constructor

data constructors

Light ADT

```
data Light
  = Red | Orange | Green
```

Light ADT

```
data Light
  = Red | Orange | Green
```

sum type

Light ADT

enum type

sum type

```
data Light
  = Red | Orange | Green
```

Light ADT

enum type

sum type

```
data Light
  = Red | Orange | Green
```

```
next :: Light → Light
next Red    = Green
next Green  = Orange
next Orange = Red
```

Person ADT

```
data Person = P String Int
```

Person ADT

product type

```
data Person = P String Int
```

Person ADT

product type

```
data Person = P String Int

tom :: Person
tom = P “Tom” 38
```

Person ADT

product type

```
data Person = P String Int

tom :: Person
tom = P “Tom” 38

mkPerson
  :: String → Int → Person
mkPerson name age
  = P name age
```

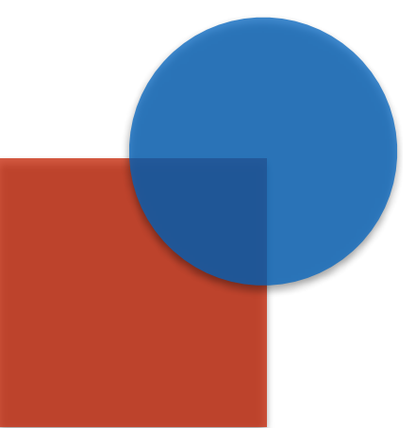
Person ADT

product type

```
data Person = P String Int

tom :: Person
tom = P “Tom” 38

mkPerson
  :: String → Int → Person
mkPerson name age
  = P name age

getName (P n a) = n
```

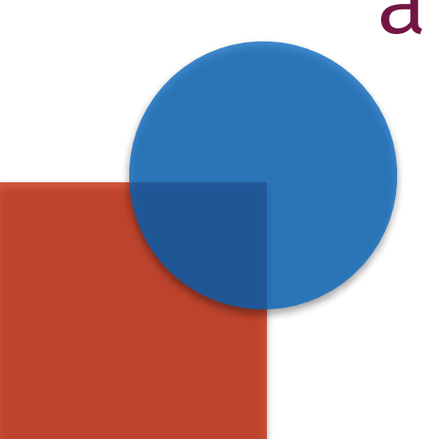
Sum of Products

```
data Shape
  = Circle Float
  | Rectangle Float Float
```



Sum of Products

```
data Shape
  = Circle Float
  | Rectangle Float Float

area :: Shape → Float
area (Circle radius)
  = radius * radius * 3.14
area (Rectangle width height)
  = width * height
```



Recursive Datatypes

```
data Tree
  = Empty
  | Branch Int Tree Tree
```

Recursive Datatypes

```
data Tree
  = Empty
  | Branch Int Tree Tree

treesum :: Tree → Int
treesum Empty
  = 0
treesum (Branch x l r)
  = x + treesum l + treesum r
```

What's its type?

```
f x = x
```

What's its type?

```
f x = x

> f 7

7
```

What's its type?

```
f x = x

> f 7
7

> f True
True
```

Parametric Polymorphism

```
f :: a → a
f x = x
```

Parametric Polymorphism

type-variabel

```
f :: a → a

f x = x
```

Parametric Polymorphism

forall a.

type-variabel

```
f :: a → a

f x = x
```

Parametric Polymorphism

forall a.

type-variabel

```
f :: a → a

f x = x
```

most general = principal type

Type Instantiation

```
f x = x

> f 7
7
> f True
True
```

Type Instantiation

```
f x = x
> f 7
7
> f True
True
```

[Int/a]

Type Instantiation

```
f x = x
> f 7
7
> f True
True
```

[Int/a]

[Bool/a]

What is the principal type?

```
const x y = x
```

What is the principal type?

```
const :: a → b → a
const x y = x
```

Monomorphic Datatypes

```
data Tree
  = Empty
  | Branch Int Tree Tree

treesum :: Tree → Int
treesum Empty
  = 0
treesum (Branch x l r)
  = x + treesum l + treesum r
```

Polymorphic Datatypes

```
data Tree a
  = Empty
  | Branch a (Tree a) (Tree a)
```

Polymorphic Datatypes

```
data Tree a
  = Empty
  | Branch a (Tree a) (Tree a)

treesum :: Tree Int → Int
treesum Empty
  = 0
treesum (Branch x l r)
  = x + treesum l + treesum r
```

Polymorphic Datatypes

```
data Tree a
  = Empty
  | Branch a (Tree a) (Tree a)

treesum :: Tree Int → Int
treesum Empty
  = 0
treesum (Branch x l r)
  = x + treesum l + treesum r

height :: Tree a → Int
height Empty
  = 0
height (Branch x l r)
  = 1 + (height l `max` height r)
```

Polymorphic Lists

```
data [] a
  = []
  | (:) a ([] a)
```

pseudo-defintion

Polymorphic Lists

```
data [a]
   = []
   | a : [a]
```

pseudo-defintion

Basic Types

Unit type

```
data () = ()
```

aka
void

Tuple Type

```
data (a,b) = (a,b)

fst :: (a,b) → a
snd :: (a,b) → b
```

projection

basic product type: a × b

Multiple Outputs

```
> 7 `quotrem` 3
(2,1)
```

n-ary Tuples

```
("Tom", 33, 1.94)
  :: (String, Int, Float)
```

Either

```
data Either a b
  = Left a
  | Right b
```

basis sum type: a + b

List Comprehension

Set Comprehensions (from math)

$$S = \{1,2,3,4,5,6,7,8,9,10\}$$

$$R = \{ n * n \mid n \in S \wedge n < 5 \}$$
$$= \{1,4,9,16\}$$

List Comprehensions in Haskell

```
s :: [Int]
s = [1,2,3,4,5,6,7,8,9,10]

r :: [Int]
r = [ n*n | n ← s, n < 5]
```

List Comprehensions in Haskell

```
s :: [Int]
s = [1,2,3,4,5,6,7,8,9,10]

r :: [Int]
r = [ n*n | n ← s, n < 5]
```

generator

List Comprehensions in Haskell

```
s :: [Int]
s = [1,2,3,4,5,6,7,8,9,10]

r :: [Int]
r = [ n*n | n ← s, n < 5]
```

generator

guard

Famous Haskell Quicksort

```
qs :: [Int] → [Int]
qs [] = []
qs (pivot:xs) =
    qs [l | l ← xs, l ≤ pivot]
    ++
    [pivot]
    ++
    qs [h | h ← xs, h > pivot]
```

Cartesian Product

```
cp :: [a] → [b] → [(a,b)]
cp l r = [(a,b)| a ← l, b ← r]
```

Cartesian Product

```
cp :: [a] → [b] → [(a,b)]
cp l r = [(a,b)| a ← l, b ← r]
```

```
> cp [1,2] ['a','b']
```

Cartesian Product

```
cp :: [a] → [b] → [(a,b)]
cp l r = [(a,b)| a ← l, b ← r]
```

```
> cp [1,2] ['a','b']
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

Cartesian Product

```
cp :: [a] → [b] → [(a,b)]
cp l r = [(a,b)| a ← l, b ← r]
```

```
> cp [1,2] ['a','b']
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

order!

Useful Notation

```
s :: [Int]
s = [1..10]
```

Useful Notation

```
s :: [Int]
s = [1..10]
```

```
> [(a,b)| a ← [1..2]
        , b ← [1..a] ]
```

Useful Notation

```
s :: [Int]
s = [1..10]
```

```
> [(a,b)| a ← [1..2]
        , b ← [1..a] ]
[(1,1),(2,1),(2,2)]
```

The Secret of String

Type Alias

```
data Distance = D Int
```

Type Alias

```
data Distance = D Int

type Distance = Int
```

Type Alias

creates new type

```
data Distance = D Int
```

```
type Distance = Int
```

Type Alias

creates new type

```
data Distance = D Int
```

```
type Distance = Int
```

defines additional name
for existing type

String Exposed

```
type String = [Char]
```

String Exposed

```
type String = [Char]

> head “Tom”
```

String Exposed

```
type String = [Char]

> head “Tom”
‘T’
```

String Exposed

```
type String = [Char]

> head “Tom”
‘T’
> “Tom” == [‘T’,’o’,’m’]
```

String Exposed

```
type String = [Char]

> head “Tom”
‘T’
> “Tom” == [‘T’,’o’,’m’]
True
```

CIJFERS & LETTERS

The Countdown Problem

The Problem

Given

source numbers	`[1,3,7,10,25,50]`
target number	765

The Problem

Given

source numbers `[1,3,7,10,25,50]`

target number 765

Wanted

expression =

The Problem

Given

source numbers `[1,3,7,10,25,50]`

target number `765`

Wanted

expression `=`

`(1+50)*(25-10)`

User-Defined Datatype

```haskell
data Op = Add | Sub | Mul | Div
```

User-Defined Datatype

```
data Op = Add | Sub | Mul | Div
```

“enum”-type with 4 alternatives

User-Defined Datatype

```
data Op = Add | Sub | Mul | Div
```

"enum"-type with 4 alternatives

values

User-Defined Datatype

type name

```
data Op = Add | Sub | Mul | Div
```

values

"enum"-type with 4 alternatives

User-Defined Function

```
apply :: Op → Int → Int → Int
```

User-Defined Function

type signature

```
apply :: Op → Int → Int → Int
```

User-Defined Function

type signature

function name

```
apply :: Op → Int → Int → Int
```

User-Defined Function

type signature

function name

parameter types

```
apply :: Op → Int → Int → Int
```

User-Defined Function

type signature

function name

parameter types

return type

```
apply :: Op → Int → Int → Int
```

User-Defined Function

type signature

function name

parameter types

return type

```
apply :: Op → Int → Int → Int
apply Add x y = x + y
apply Sub x y = x - y
apply Mul x y = x * y
apply Div x y = x `div` y
```

User-Defined Function

type signature

function name

parameter types

return type

```
apply :: Op → Int → Int → Int
apply Add x y = x + y
apply Sub x y = x - y
apply Mul x y = x * y
apply Div x y = x `div` y
```

parameters

User-Defined Function

type signature

function name

parameter types

return type

```
apply :: Op → Int → Int → Int
apply Add x y = x + y
apply Sub x y = x - y
apply Mul x y = x * y
apply Div x y = x `div` y
```

parameters

body

User-Defined Function

```
apply :: Op → Int → Int → Int
apply Add x y = x + y
apply Sub x y = x - y
apply Mul x y = x * y
apply Div x y = x `div` y
```

case analysis
by
pattern matching

GHCi REPL

```
> apply Add 1 2
```

GHCi REPL

```
> apply Add 1 2
3
```

GHCi REPL

```
> apply Add 1 2
3
> apply Mul 3 2
```

GHCi REPL

```
> apply Add 1 2
3
> apply Mul 3 2
6
```

GHCi REPL

```
> apply Add 1 2
3
> apply Mul 3 2
6
> apply Sub 4 1
```

GHCi REPL

```
> apply Add 1 2
3
> apply Mul 3 2
6
> apply Sub 4 1
3
```

GHCi REPL

```
> apply Add 1 2
3
> apply Mul 3 2
6
> apply Sub 4 1
3
> apply Div 3 0
```

GHCi REPL

```
> apply Add 1 2
3
> apply Mul 3 2
6
> apply Sub 4 1
3
> apply Div 3 0
*** Exception: divide by zero
>_
```

Another Function

```
valid :: Op → Int → Int → Bool
valid Add x y = True
valid Sub x y = x > y
valid Mul x y = True
valid Div x y = y /= 0 && x `mod` y == 0
```

Recursive Datatype

```haskell
data Expr = Val Int | App Op Expr Expr
```

Recursive Datatype

type name

```
data Expr = Val Int | App Op Expr Expr
```

Recursive Datatype

type name

```
data Expr = Val Int | App Op Expr Expr
```

data constructors

Recursive Datatype

type name constructor field

```
data Expr = Val Int | App Op Expr Expr
```

data constructors

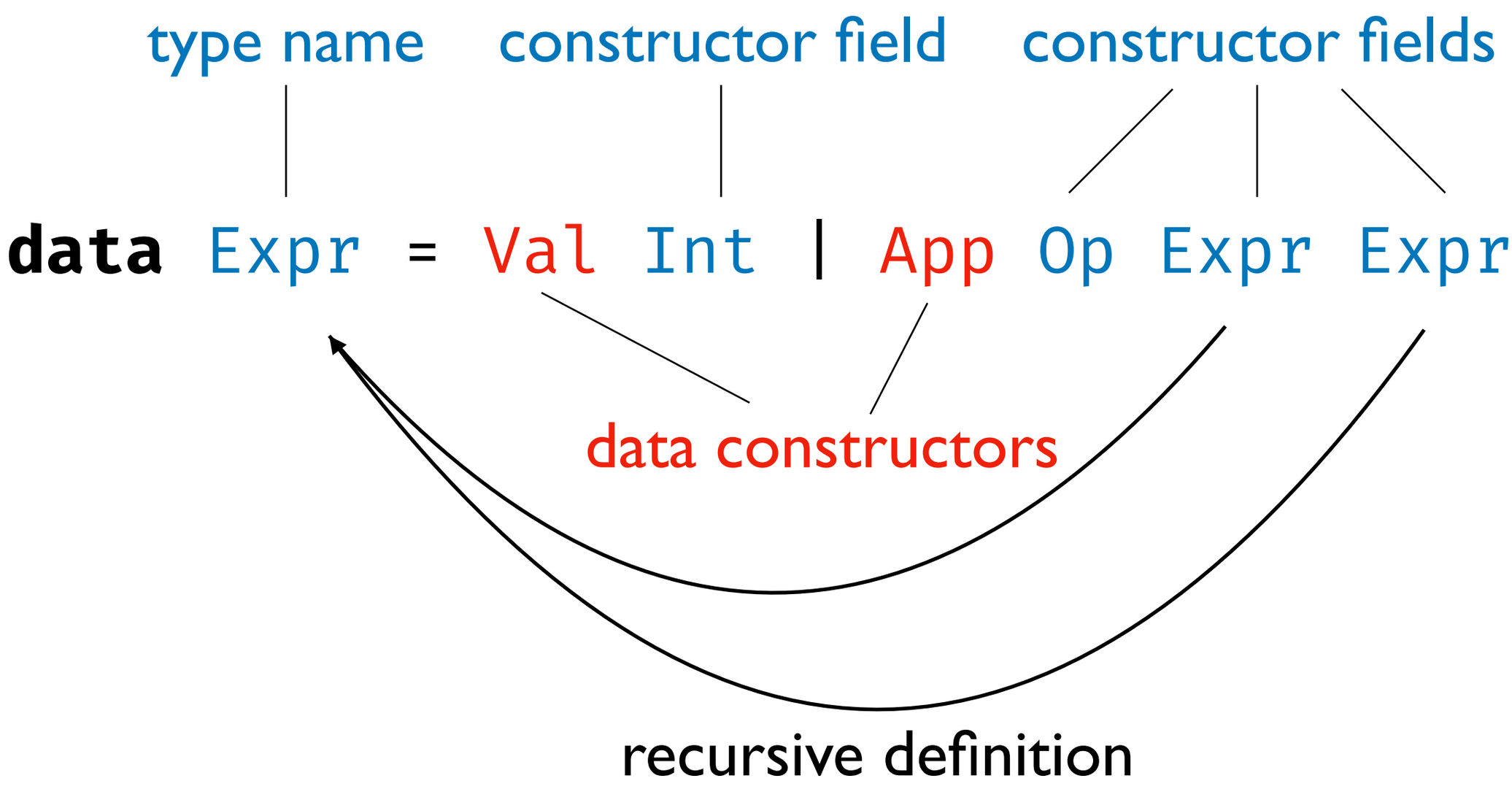
Recursive Datatype

type name constructor field constructor fields

```
data Expr = Val Int | App Op Expr Expr
```

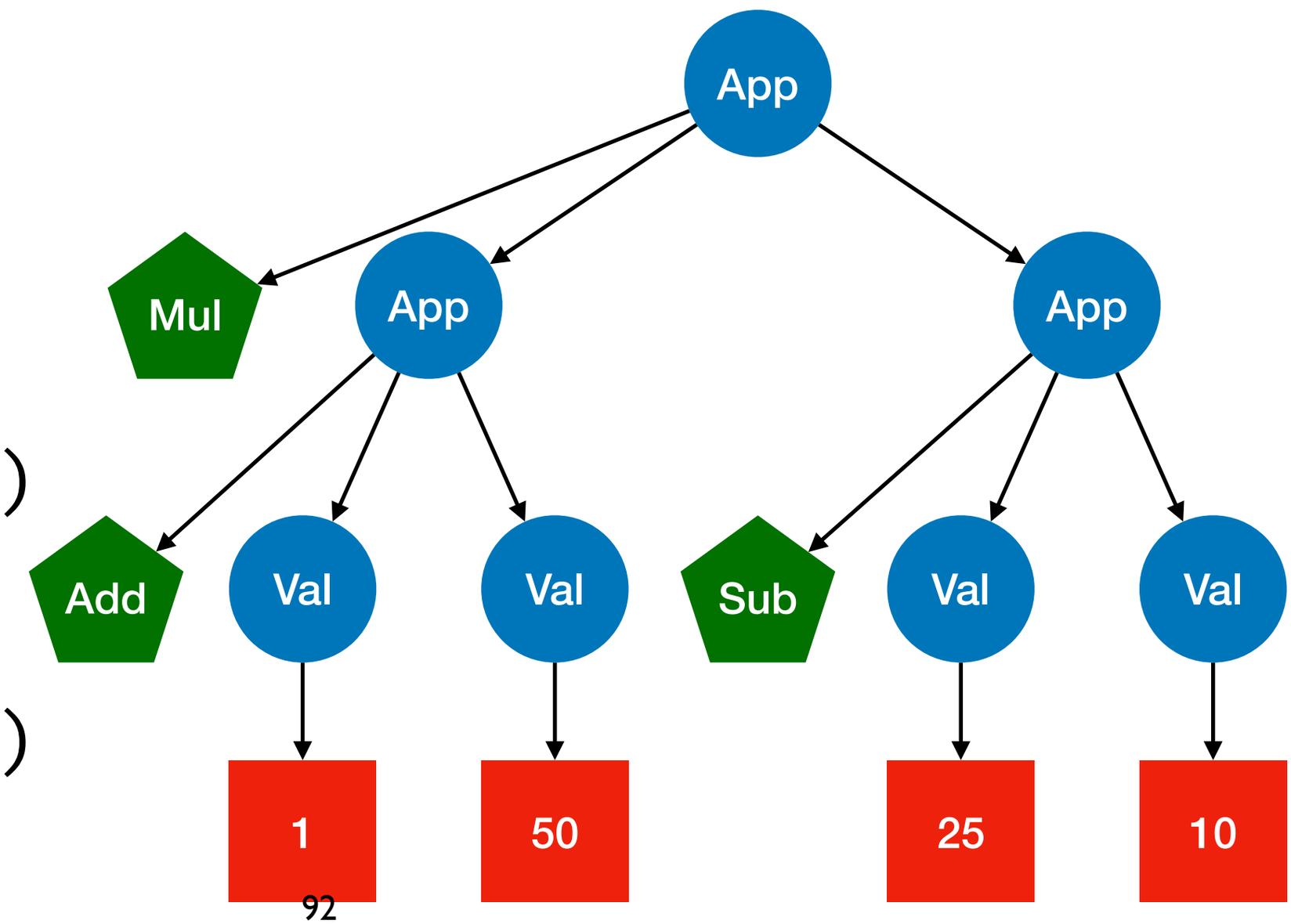
data constructors

Recursive Datatype



Recursive Datatype

```
example :: Expr
example =
  App Mul
    (App Add
        (Val 1)
        (Val 50))
    (App Sub
        (Val 25)
        (Val 10))
```



Recursive Fuction

```
data Expr = Val Int | App Op Expr Expr
```

```
values :: Expr → [Int]
```

Recursive Fuction

```
data Expr = Val Int | App Op Expr Expr
```

list of Int

```
values :: Expr → [Int]
```

Recursive Fuction

```
data Expr = Val Int | App Op Expr Expr
```

list of Int

```
values :: Expr → [Int]
values (Val n)      = [n]
values (App _ l r) = values l ++ values r
```

Recursive Fuction

```
data Expr = Val Int | App Op Expr Expr
```

list of Int

singleton list

```
values :: Expr → [Int]
values (Val n)       = [n]
values (App _ l r) = values l ++ values r
```

Recursive Fuction

```
data Expr = Val Int | App Op Expr Expr
```

list of Int

singleton list

list concatenation

```
values :: Expr → [Int]
values (Val n)       = [n]
values (App _ l r) = values l ++ values r
```

Recursive Fuction

```
data Expr = Val Int | App Op Expr Expr
```

list of Int

singleton list

list concatenation

```
values :: Expr → [Int]
values (Val n)       = [n]
values (App _ l r) = values l ++ values r
```

recursive definition

Recursive Fuction

```
data Expr = Val Int | App Op Expr Expr
```

list of Int

singleton list

list concatenation

```
values :: Expr → [Int]
values (Val n)       = [n]
values (App _ l r) = values l ++ values r
```

recursive definition

function follows recursion of datatype

Recursive Fuction

```
data Expr = Val Int | App Op Expr Expr
```

```
values :: Expr → [Int]
values (Val n)       = [n]
values (App _ l r) = values l ++ values r
```

function follows recursion of datatype

```
> values example
[1,50,25,10]
```

List Comprehension

```
eval :: Expr → [Int]
eval (Val n)      =
  [n | n > 0]
eval (App o l r) =
  [apply o x y
  | x ← eval l
  , y ← eval r
  , valid o x y
  ]
```

List Comprehension

```
eval :: Expr → [Int]
eval (Val n)      =
  [n | n > 0]
eval (App o l r) =
  [apply o x y
  | x ← eval l
  , y ← eval r
  , valid o x y
  ]
```

generators

List Comprehension

```
eval :: Expr → [Int]
eval (Val n)      =
  [n | n > 0]
eval (App o l r) =
  [apply o x y
  | x ← eval l
  , y ← eval r
  , valid o x y
  ]
```

guards

generators

List Comprehension

```
eval :: Expr → [Int]
eval (Val n)      =
  [n | n > 0]
eval (App o l r) =
  [apply o x y
  | x ← eval l
  , y ← eval r
  , valid o x y
  ]
```

guards

generators

```
> eval example
[765]
```

Parametric Polymorphism

```
subbags :: [a] → [[a]]
subbags xs = [zs | ys ← subs xs
                 , zs ← perms ys ]
```

Parametric Polymorphism

type variable

```
subbags :: [a] → [[a]]
subbags xs = [zs | ys ← subs xs
                 , zs ← perms ys ]
```

Parametric Polymorphism

aka
Generics

type variable

```
subbags :: [a] → [[a]]
subbags xs = [zs | ys ← subs xs
                 , zs ← perms ys ]
```

Parametric Polymorphism

aka
Generics

type variable

```
subbags :: [a] → [[a]]
subbags xs = [zs | ys ← subs xs
                 , zs ← perms ys ]
```

```
> subbags [1,2]
[[],[1],[2],[1,2],[2,1]]
> subbags [True,False]
[[],[True],[False],[True,False],[False,True]]
```

Check Solution

```
solution :: Expr → [Int] → Int → Bool
solution e ns n =
      elem (values e) (subbags ns)
   && eval e == [n]
```

Check Solution

```
solution :: Expr → [Int] → Int → Bool
solution e ns n =
      elem (values e) (subbags ns)
   && eval e == [n]
```

```
> solution example [1,3,7,10,25,50] 765
True
```

Generate Solution

```
> split [1,2,3]
[([],[1,2,3]),([1],[2,3]),([1,2],[3])
,([1,2,3],[])]
```

Generate Solution

```
> split [1,2,3]
[([],[1,2,3]),([1],[2,3]),([1,2],[3])
,([1,2,3],[])]
```

```
split :: [a] → [([a],[a])]
```

Generate Solution

```
> split [1,2,3]
[([],[1,2,3]),([1],[2,3]),([1,2],[3])
,([1,2,3],[])]
```

```
split :: [a] → [([a],[a])]
split []     = [([],[])]
```

Generate Solution

```
> split [1,2,3]
[([],[1,2,3]),([1],[2,3]),([1,2],[3])
,([1,2,3],[])]
```

```
split :: [a] → [([a],[a])]
split []     = [([],[])]
split (x:xs) =
  ([],x:xs) : [(x:ls,rs)
               |(ls,rs)←split xs]
```

Generate Solution

```
> nesplit [1,2,3]
[([1],[2,3]),([1,2],[3])]
```

Generate Solution

```
> nesplit [1,2,3]
[([1],[2,3]),([1,2],[3])]
```

```
nesplit :: [a] → [([a],[a])]
```

Generate Solution

```
> nesplit [1,2,3]
[([1],[2,3]),([1,2],[3])]
```

```
ne (xs,ys) = not (null xs || null ys)

nesplit :: [a] → [([a],[a])]
```

Generate Solution

```
> nesplit [1,2,3]
[([1],[2,3]),([1,2],[3])]
```

```
ne :: ([a],[b]) → Bool
ne (xs,ys) = not (null xs || null ys)

nesplit :: [a] → [([a],[a])]
```

Generate Solution

```
> nesplit [1,2,3]
[([1],[2,3]),([1,2],[3])]
```

```
ne :: ([a],[b]) → Bool
ne (xs,ys) = not (null xs || null ys)

nesplit :: [a] → [([a],[a])]
nesplit l = [t | t ← split l, ne t]
```

Generate Solution

```
combine :: Expr → Expr → [Expr]
combine l r = [App o l r | o ← ops]

ops :: [Op]
ops = [Add,Sub,Mul,Div]
```

Generate Solution

```
exprs :: [Int] → [Expr]
exprs []  = []
exprs [n] = [Val n]
exprs ns  = [e | (ls,rs) ← nesplit ns
               , l ← exprs ls
               , r ← exprs rs
               , e ← combine l r
               ]
```

Solutions

```
solutions :: [Int] → Int → [Expr]
solutions ns n
  = [e | ns' ← subbags ns
       , e   ← exprs ns'
       , eval e == [n]
       ]
```

Functional Pearl: The countdown problem.
Graham Hutton. Journal of Functional Programming.
2002.

Solutions

```
solutions :: [Int] → Int → [Expr]
solutions ns n
  = [e | ns' ← subbags ns
       , e   ← exprs ns'
       , eval e == [n]
       ]
```

naive
generate & test
strategy

Functional Pearl: The countdown problem.
Graham Hutton. Journal of Functional Programming.
2002.