Before we begin

```
cd 00_setup && make
# Windows users: see setup.md
```

(setup)

g2g

Haskell 101

pravnar@, javran@, mihaimaruseac@, ibobyr@, nicuveo@

November 6, 2024

g2g

- 101
  - Concepts and generalities
  - Syntax overview
  - Data structures
  - Declaring functions

Not today

- Project environment
  - Cabal? Stack?
  - Hackage? Stackage?
  - Haskell at Google?
- Advanced stuff
  - Functors?
  - Monads?
  - Monad Transformers?
  - ...

Prerequisites

- Programming knowledge
  - **Imperative programming is enough**
  - **Functional programming is a plus**

What is Haskell?

A programming language

- General purpose

What is Haskell?

A programming language

- General purpose
- Purely functional

What is Haskell?

A programming language

- General purpose
- Purely functional
- Lazily evaluated

g2g

What is Haskell?

A programming language

- General purpose
- Purely functional
- Lazily evaluated
- Strongly statically typed

Haskell is NOT...

- A silver bullet
- Only for category theorists

Haskell is NOT...

- A silver bullet
- Only for category theorists
- Hard!

Haskell is NOT...

- A silver bullet
- Only for category theorists
- Hard! Just different...

- Computation method is function application

```haskell
sum [1 .. 10]
```
(Haskell)

- Computation method is function application
- Contrast with variable assignment

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

(Haskell)

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

(C++)

- Computation method is function application
- Contrast with variable assignment
- Lambda calculus → . . . → Haskell

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

(Haskell)

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

(C++)

- Similar to math

```haskell
f :: Int -> Int
f x = x + 1
```
(Haskell)

$$f : \mathbb{Z} \to \mathbb{Z}$$
$$f(x) = x + 1$$

- Similar to math

Math	Haskell
$f(x)$	`f x`
$f(x, y)$	`f x y`
$f(g(x))$	`f (g x)`
$f(g(x), h(y))$	`f (g x) (h y)`
$f(x)g(y)$	`f x * g y`

g2g

- Similar to math
- most common thing $\Longrightarrow$ easiest to type

Math	Haskell
$f(x)$	`f x`
$f(x, y)$	`f x y`
$f(g(x))$	`f (g x)`
$f(g(x), h(y))$	`f (g x) (h y)`
$f(x)g(y)$	`f x * g y`

- No more statement vs expression differences

```
int a = 3;
int b = 5;
int c = f(a + b, a + 2 * b);
```

(C++)

Everything is an expression

- No more statement vs expression differences

```
1 + 2
```

(Haskell)

Everything is an expression

- No more statement vs expression differences

```haskell
let a = 1
in  a + 2
```

(Haskell)

Everything is an expression

- No more statement vs expression differences
- Textual substitution as model for evaluation (reduction)

```
let a = 1
in  a + 2
```
(Haskell)

```
1 + 2
```
(eval)

g2g

Everything is an expression

- No more statement vs expression differences
- Textual substitution as model for evaluation (reduction)

```haskell
let a = 1
in  a + 2
```
(Haskell)

```
3
```
(eval)

Everything is an expression

- No more statement vs expression differences
- Textual substitution as model for evaluation (reduction)

```
let a = if someBool then 1 else 0
in  a + 2
```

(Haskell)

Everything is an expression

- No more statement vs expression differences
- Textual substitution as model for evaluation (reduction)

```haskell
let a = if someBool then 1 else 0
in  a + 2
```

(Haskell)

```
(if someBool then 1 else 0) + 2
```

(eval)

Everything is an expression

- No more statement vs expression differences
- Textual substitution as model for evaluation (reduction)

```haskell
let a = if someBool then 1 else 0
in  a + (let b = 2 in b)
```

(Haskell)

g2g

Everything is an expression

- No more statement vs expression differences
- Textual substitution as model for evaluation (reduction)

```
let a = if someBool then 1 else 0
in  a + (let b = 2 in b)
```
(Haskell)

```
(if someBool then 1 else 0) + (let b = 2 in b)
```
(eval)

- No more statement vs expression differences
- Textual substitution as model for evaluation (reduction)

```haskell
let a = if someBool then 1 else 0
in  a + (let b = 2 in b)
```
(Haskell)

```
(if someBool then 1 else 0) + 2
```
(eval)

- No more statement vs expression differences
- Textual substitution as model for evaluation (reduction)

```haskell
let offset = case color of
                Red   ->  0
                Green ->  8
                Blue  -> 16
in baseValue + offset
```

(Haskell)

- Do we have variables?

```
let a = 3
in  a = a + 1
```

(Haskell)

- Do we have variables?

```
let a = 3
in  a = a + 1
```

(Haskell)

```
3 = 3 + 1
```

(eval)

- Do we have variables?

```
let a = 3
in  a = a + 1
```
(Haskell)

```
3 = 4 -- this is bad
```
(eval)

Purely functional

- Do we have variables?
- No: Everything is immutable

```
let a = 3
in  a = a + 1 -- compile error
```

(Haskell)

```
3 = 4 -- compile error
```

(eval)

- Do we have variables?
- No: Everything is immutable

```haskell
let a = 3
in  a = a + 1 -- compile error
```

(Haskell)

```haskell
let a = 3
    b = a + 1
in b -- this is ok
```

(Haskell)

Purely functional

- Do we have variables?
- No: Everything is immutable
- No side effects!

```haskell
numberToWords :: Int -> String
```

(Haskell)

g2g

- Do we have variables?
- No: Everything is immutable
- No side effects unless explicitly stated

```
numberToWords :: Int -> String
readFile :: String -> IO String
```

(Haskell)

Purity...

```haskell
numberToWords :: Int -> String  -- pure function
readFile :: String -> IO String -- impure function
```

(Haskell)

g2g

...and corruption

- $\exists$ `f :: a -> IO a` **(from pure to impure)**
- $\nexists$ `f :: IO a -> a` **(from impure to pure)**

- $\exists$ `f :: a -> IO a` **(from pure to impure)**
- $\nexists$ `f :: IO a -> a` **(from impure to pure)**

IO corrupts.

- Which order for textual substitution?

```haskell
add x y = x + y
```
(Haskell)

```
add (12 + 8) (20 + 2)
```
(eval)

- Which order for textual substitution?

```haskell
add x y = x + y
```
(Haskell)

```
-- arguments first          | -- function body first
add (12 + 8) (20 + 2)       | add (12 + 8) (20 + 2)
                            |
                            |
                            |
```
(eval)

g2g

- Which order for textual substitution?

```haskell
add x y = x + y
```

(Haskell)

```
-- arguments first        | -- function body first
add (12 + 8) (20 + 2)     | add (12 + 8) (20 + 2)
add 20 22                 | (12 + 8) + (20 + 2)
20 + 22                   | 20 + 22
42                        | 42
```

(eval)

Let's talk about evaluation order

- Which order for textual substitution?
- Are they always equivalent?

```haskell
add x y = x + y
```

(Haskell)

```
-- arguments first        | -- function body first
add (12 + 8) (20 + 2)     | add (12 + 8) (20 + 2)
add 20 22                 | (12 + 8) + (20 + 2)
20 + 22                   | 20 + 22
42                        | 42
```

(eval)

- Which order for textual substitution?
- Are they always equivalent?

```
const x y = x
```

(Haskell)

```
-- arguments first                | -- function body first
const expr1 expr2                 | const expr1 expr2
                                  |
                                  |
                                  |
```

(eval)

g2g

Let's talk about evaluation order

- Which order for textual substitution?
- Are they always equivalent?

```haskell
const x y = x
```

(Haskell)

```
-- arguments first         | -- function body first
const (40 + 2) (20 + 2)    | const (40 + 2) (20 + 2)
const 42 22                | (40 + 2)
42                         | 42
                           |
```

(eval)

Let's talk about evaluation order

- Which order for textual substitution?
- Are they always equivalent?

```haskell
const x y = x
```
(Haskell)

```
-- arguments first              | -- function body first
const (4+2) (error "Crash")     | const (4+2) (error "Crash")
                                |
                                |
                                |
```
(eval)

Let's talk about evaluation order

- Which order for textual substitution?
- Are they always equivalent?

```haskell
const x y = x
```

(Haskell)

```
-- arguments first               | -- function body first
const (4+2) (error "Crash")      | const (4+2) (error "Crash")
error: Crash                     | (4 + 2)
                                 | 6
                                 |
```

(eval)

Lazy evaluation

- Deferred expression evaluation
- Not used $\Longrightarrow$ not computed

- Deferred expression evaluation
- Not used $\Longrightarrow$ not computed

QUIZ

Lazy evaluation

- Deferred expression evaluation
- Not used $\Longrightarrow$ not computed

```cpp
if (obj != nullptr && obj->value > 0)
```

(C++)

g2g

Laziness pros and cons

- Memory pitfalls

Delayed computations (but escape hatches)

Laziness pros and cons

- Memory pitfalls
- IO and parallelism pitfalls

Delayed computations (but escape hatches)

Laziness pros and cons

- \- Memory pitfalls
- \- IO and parallelism pitfalls
- \+ Huge optimizations

Equation reduction and short-circuiting

Laziness pros and cons

- \- Memory pitfalls
- \- IO and parallelism pitfalls
- \+ Huge optimizations
- \+ Greater expressivity (e.g. infinite structures)

```haskell
> naturalNumbers = [0,1..]
> squaredNumbers = map (^2) naturalNumbers
> take 5 squaredNumbers
[0,1,4,9,16]
```

(Haskell)

- Untyped

```asm
mov rax, r8
mov rbx, r9
add rax, rbx
```

(asm)

- Untyped
- Dynamically typed

```python
def add(a, b):
  return a + b
```

(Python)

- Untyped
- Dynamically typed
- Statically typed

```cpp
int add(int a, int b) {
  return a + b;
}
```

(C++)

- Untyped
- Dynamically typed
- Statically typed
- Haskell: strongly statically typed with type inference

```haskell
-- add :: Int -> Int -> Int
add x y = x + y
```

(Haskell)

- Read-Eval-Print-Loop interpreter
- Your new best friend
- Fast iteration time

- Read-Eval-Print-Loop interpreter
- Your new best friend
- Fast iteration time
- Desktop calculator

```haskell
Prelude> 2 + 3
5
Prelude> sqrt (3 ^ 2 + 4 ^ 2)
5.0
```

(Haskell)

- Read-Eval-Print-Loop interpreter
- Your new best friend
- Fast iteration time
- Desktop calculator
- Type inference using `:type` (`:t`)
- ...

Basic Haskell types

```
Prelude> :t anInt
anInt :: Int
Prelude> :t aDouble
aDouble :: Double
Prelude> :t 'a'
'a' :: Char
Prelude> :t "this is a string"
"this is a string" :: String -- or [Char]
Prelude> :t [anInt, anInt + 2, 2 * anInt]
[anInt, anInt + 2, 2 * anInt] :: [Int]
Prelude> :t (anInt, aDouble)
(anInt, aDouble) :: (Int, Double)
```

(Haskell)

Function types

```
f        ::  Int -> Int -> Int
```

g2g

Function types

```
f       ::  Int -> Int -> Int
f x y = x + y
```

Function types

```
f       ::  Int -> Int -> Int
f x y = x + y
f 1 2  ::                 Int
```

Function types

```
f       ::  Int -> Int -> Int
f x y = x + y
f 1 2  ::                Int
3
```

g2g

- What is the type of `length` below?

```
Prelude> length [1 .. 5]
5
Prelude> length "this is some string"
19
```

(Haskell)

- What is the type of `length` below?
- `length :: [a] -> Int`

```
Prelude> length [1 .. 5]
5
Prelude> length "this is some string"
19
```
(Haskell)

- What is the type of `length` below?
- `length :: [a] -> Int`
- `a` is a type variable: stand-in for any type

```
Prelude> length [1 .. 5]
5
Prelude> length "this is some string"
19
```

(Haskell)

First letter rule

Context	Capital letter	Lowercase letter
Type	Fixed type (`Int`, ...)	Type variable (`a`, ...)
Value	????	Functions, arguments, names, ...

- Type of a function determine what the function can do?
- There can be only one `a -> b -> a` function
- (modulo assumptions)

Searching for functions

- Type of a function determine what the function can do?
- There can be only one `a -> b -> a` function
- (modulo assumptions)
- Can we get to the function given the type signature?

g2g

Searching for functions

- Type of a function determine what the function can do?
- There can be only one `a -> b -> a` function
- (modulo assumptions)
- Can we get to the function given the type signature?
- `djinn`: Generate Haskell code from a type

- Type of a function determine what the function can do?
- There can be only one `a -> b -> a` function
- (modulo assumptions)
- Can we get to the function given the type signature?
- `djinn`: Generate Haskell code from a type
- Hoogle, Hayoo

- Haskell code is immutable
- But our code needs variables

```
sumList :: [Int] -> Int
sumList list = ???
```
(Haskell)

- Haskell code is immutable
- But our code needs variables
- Recursion is our friend

```haskell
sumList :: [Int] -> Int
sumList list =
  if null list
  then 0
  else head list + sumList (tail list)
```

(Haskell)

directory : `01_functions`

change : `src/Codelab.hs`
(replace `codelab` with implementation)

check with : `make`

Type synonyms

```haskell
type Point    = (Int, Int)  -- tuple
```

(Haskell)

g2g

Type synonyms

```haskell
type Point   = (Int, Int)  -- tuple
type Polygon = [Point]      -- list
```

(Haskell)

Type synonyms

```haskell
type Point   = (Int, Int)  -- tuple
type Polygon = [Point]      -- list
type Map k v = [(k, v)]     -- type parameters
```

(Haskell)

g2g

What do they consist of?

What do they consist of?

- No private members...

What do they consist of?

- No private members...
- No modifiers...

What do they consist of?

- No private members...
- No modifiers...
- No methods...

What do they consist of?

- No private members...
- No modifiers...
- No methods...
- Only constructors

C++ data types

```
class C {
 public:
  C(string name, int age);
  virtual ~C ();
  string getName();
  int getAge();
  int getAgeIn(int n);
  void setName(string name);
  void setAge(int age);
 private:
  string name;
  int age;
};
```

(C++)

name
age
C()
~C()
getName()
getAge()
getAgeIn()
setName()
setAge()

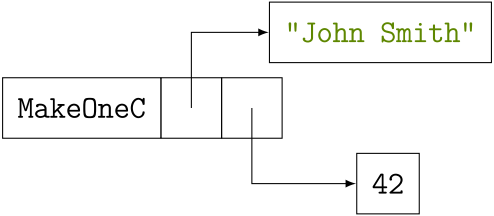
g2g

Haskell data types

```haskell
data C = MakeOneC String Int

person :: C
person = MakeOneC "John Smith" 42
```

(Haskell)



The almighty "data" keyword

```haskell
data Bool     = False | True
```

(Haskell)

```haskell
True  :: Bool
False :: Bool
```

(types)

The almighty "data" keyword

```
data Bool     = False | True
data None     = None
```
(Haskell)

```
None :: None
```
(types)

The almighty ”data” keyword

```haskell
data Bool    = False | True
data None    = None
data Minutes = Minutes Int
```

(Haskell)

```haskell
Minutes    :: Int -> Minutes
Minutes 42 ::        Minutes
```

(types)

g2g

First letter rule

Context	Capital letter	Lowercase letter
Type	Fixed type (`Int`, ...)	Type variable (`a`, ...)
Value	Constructors (True, False)	Functions, arguments, names, ...

Not

```haskell
data Bool = False | True

not :: Bool -> Bool
not x = ???
```

(Haskell)

g2g

Not

```haskell
data Bool = False | True

not :: Bool -> Bool
not x = if x then False else True
```

(Haskell)

g2g

Not

```haskell
data Bool = False | True

not :: Bool -> Bool
not x = if x then False else True
```

(Haskell)

Only for built-in type(s)

Not

```
data Bool = False | True

not :: Bool -> Bool
not True  = False
not False = True
```

(Haskell)

#PatternMatching

And

```haskell
data Bool = False | True

(&&) :: Bool -> Bool -> Bool
x && y = ???
```

(Haskell)

And

```haskell
data Bool = False | True

(&&) :: Bool -> Bool -> Bool
x && y = if x
  then (if y then True else False)
  else False
```

(Haskell)

g2g

And

```haskell
data Bool = False | True

(&&) :: Bool -> Bool -> Bool
x && y = if x
  then (if y then True else False)
  else False
```

(Haskell)

Only for built-in type(s)

```haskell
data Bool = False | True

(&&) :: Bool -> Bool -> Bool
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

(Haskell)

```haskell
data Bool = False | True

(&&) :: Bool -> Bool -> Bool
True && True = True
x    && y    = False
```

(Haskell)

And

```haskell
data Bool = False | True

(&&) :: Bool -> Bool -> Bool
True && True = True
x    && y    = False
```

(Haskell)

Can you spot a problem?

g2g

And

```
data Bool = False | True

(&&) :: Bool -> Bool -> Bool
True && True = True
x    && y    = False
```

(Haskell)

Eager in the second argument :(

And

```haskell
data Bool = False | True

(&&) :: Bool -> Bool -> Bool
True && y = y
x    && y = False
```

(Haskell)

g2g

And

```haskell
data Bool = False | True

(&&) :: Bool -> Bool -> Bool
True && y = y
_    && _ = False
```

(Haskell)

Deconstructors?

```haskell
data Minutes = Minutes Int

add :: Minutes -> Minutes -> Minutes
add mx my = ???
```

(Haskell)

g2g

Deconstructors?

```haskell
data Minutes = Minutes Int

add :: Minutes -> Minutes -> Minutes
add mx my = mx + my
```

(Haskell)

Deconstructors?

```haskell
data Minutes = Minutes Int

add :: Minutes -> Minutes -> Minutes
add mx my = mx + my
```

(Haskell)

Needs overload for `+` to work on `Minutes` directly!

g2g

Deconstructors?

```haskell
data Minutes = Minutes Int

add :: Minutes -> Minutes -> Minutes
add (Minutes x) (Minutes y) = ???
```

(Haskell)

Deconstructors?

```haskell
data Minutes = Minutes Int

add :: Minutes -> Minutes -> Minutes
add (Minutes x) (Minutes y) = Minutes (x + y)
```

(Haskell)

Deconstructors?

```haskell
data Minutes = Minutes Int

add :: Minutes -> Minutes -> Minutes
add (Minutes x) (Minutes y) = Minutes $ x + y
```

(Haskell)

Record syntax

```haskell
data User = User String String Int
```

(Haskell)

```haskell
User                :: String -> String -> Int -> User
```

(types)

Record syntax

```haskell
data User = User {
  userFirstName :: String,
  userLastName  :: String,
  userAge       :: Int
}
```
(Haskell)

```haskell
User              :: String -> String -> Int -> User
```
(types)

g2g

Record syntax

```haskell
data User = User {
  userFirstName :: String,
  userLastName  :: String,
  userAge       :: Int
}
```

(Haskell)

```haskell
User          :: String -> String -> Int -> User
userFirstName :: User -> String
userLastName  :: User -> String
userAge       :: User -> Int
```

(types)

directory : `02_datatypes`

change : `src/Codelab.hs`
(replace `codelab` with implementation)

check with : `make`

The almighty "data" keyword. Continued

```haskell
data Bool    = False | True
data None    = None
data Minutes = Minutes Int
```

(Haskell)

The almighty "data" keyword. Continued

```haskell
data Bool    = False | True
data None    = None
data Minutes = Minutes Int
data Maybe a = Nothing | Just a
```

(Haskell)

```haskell
Nothing ::      Maybe a
Just    :: a -> Maybe a
Just 42 ::      Maybe Int
```

(types)

g2g

The almighty "data" keyword. Continued

```haskell
data Bool    = False | True
data None    = None
data Minutes = Minutes Int
data Maybe a = Nothing | Just a
data List a  = Nil | Cell a (List a)
```
(Haskell)

```haskell
                 Nil ::      List a
                Cell :: a -> List a -> List a
Cell 0 (Cell 1 (Nil)) :: List Int
```
(types)

g2g

The almighty "data" keyword. Continued

```haskell
data Bool      = False | True
data None      = None
data Minutes = Minutes Int
data Maybe a = Nothing | Just a
data List a  = Nil | Cell a (List a)
```

(Haskell)

```haskell
                 Nil ::      List a
                Cell :: a -> List a -> List a
Cell 0 $ Cell 1 $ Nil :: List Int
```

(types)

The almighty "data" keyword. Continued

```haskell
data Bool      = False | True
data None      = None
data Minutes = Minutes Int
data Maybe a = Nothing | Just a
data List a  = Nil | Cell a (List a)
data [a]       = [] | (a:[a])
```

(Haskell)

```haskell
        [] :: [a]
       (:) :: a -> [a] -> [a]
0 : 1 : [] :: [Int]
```

(types)

g2g

The almighty "data" keyword. Continued

```haskell
data Bool      = False | True
data None      = None
data Minutes = Minutes Int
data Maybe a = Nothing | Just a
data List a  = Nil | Cell a (List a)
data [a]       = [] | (a:[a])
```
(Haskell)

```haskell
    [] :: [a]
   (:) :: a -> [a] -> [a]
[0, 1] :: [Int]
```
(types)

Length

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

length :: [a] -> Int
length l = ???
```

(Haskell)

Length

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

length :: [a] -> Int
length []       = ???
length (x:xs) = ???
```

(Haskell)

g2g

Length

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

length :: [a] -> Int
length []       = 0
length (x:xs) = ???
```

(Haskell)

g2g

Length

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

length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

(Haskell)

#Recursion

g2g

Codelab - `03_lists`

directory : `03_lists`

change : `src/Codelab.hs`
(replace `codelab` with implementation)

check with : `make`

Curried functions, partial application

```haskell
f       :: Int ->    Int -> Int
```

(Haskell)

Curried functions, partial application

```
f       :: Int -> ( Int -> Int )
f       :: Int ->   Int -> Int
```

(Haskell)

Curried functions, partial application

```haskell
f       :: Int -> ( Int -> Int )
f       :: Int ->   Int -> Int
f 1     ::          Int -> Int
```

(Haskell)

g2g

Curried functions, partial application

```
 f       :: Int -> ( Int -> Int )
 f       :: Int ->   Int -> Int
 f 1     ::          Int -> Int

(f 1) 2 ::                 Int
```

(Haskell)

Curried functions, partial application

```
 f       :: Int -> ( Int -> Int )
 f       :: Int ->   Int -> Int
 f 1     ::          Int -> Int
 f 1  2  ::                 Int
(f 1) 2  ::                 Int
```

(Haskell)

Quiz!

```
???  ::  (a -> b) -> [a] -> [b]
```

Quiz!

```
???  ::  (a -> b) -> [a] -> [b]
```

`(a -> b)`	function from type A to type B
`[a]`	list of values of type A
`[b]`	list of values of type B

g2g

Quiz!

```
map :: (a -> b) -> [a] -> [b]
```

`(a -> b)`	**function from type A to type B**
`[a]`	**list of values of type A**
`[b]`	**list of values of type B**

g2g

Quiz!

```haskell
map  ::  (a -> b) -> [a] -> [b]
```

```haskell
            add2 x = x + 2
        map add2 [1, 2, 3]
map (\x -> x + 2) [1, 2, 3]
```

`map add2 [1, 2, 3]` — **direct call, result** `[3, 4, 5]`

`map (\x -> x + 2) [1, 2, 3]` — **lambda function (anonymous)**

g2g

Quiz!

```
   map  ::  (a -> b) -> [a] -> [b]
?????? ::  (a -> Bool) -> [a] -> [a]
```

Quiz!

```haskell
   map  ::  (a -> b) -> [a] -> [b]
filter  ::  (a -> Bool) -> [a] -> [a]
```

Quiz!

```haskell
   map  ::  (a -> b) -> [a] -> [b]
filter  ::  (a -> Bool) -> [a] -> [a]
   (.)  ::  (b -> c) -> (a -> b) -> (a -> c)
```

Quiz!

```
   map  ::  (a -> b) -> [a] -> [b]
filter  ::  (a -> Bool) -> [a] -> [a]
   (.)  ::  (b -> c) -> (a -> b) -> (a -> c)
```

$$(f \circ g)(x) = f(g(x))$$

Quiz!

```
   map  ::  (a -> b) -> [a] -> [b]
filter  ::  (a -> Bool) -> [a] -> [a]
   (.)  ::  (b -> c) -> (a -> b) -> (a -> c)
```

```
           show :: Stuff -> String
length          ::          String -> Int
length   . show :: Stuff           -> Int
```

g2g

Quiz!

```
   map  ::  (a -> b) -> [a] -> [b]
filter  ::  (a -> Bool) -> [a] -> [a]
   (.)  ::  (b -> c) -> (a -> b) -> (a -> c)
```

```
cat input | grep token | sed stuff | tee output
```

g2g

Quiz with a vengeance!

```
?????  ::  (a -> b -> a) -> a -> [b] -> a
```

g2g

Quiz with a vengeance!

```
?????  ::  (a -> b -> a) -> a -> [b] -> a
```

`(a -> b -> a)`	combines accumulator and value
`a`	initial accumulator
`[b]`	list of values
`a`	result

Quiz with a vengeance!

```
foldl  ::  (a -> b -> a) -> a -> [b] -> a
```

`(a -> b -> a)`	combines accumulator and value
`a`	initial accumulator
`[b]`	list of values
`a`	result

Quiz with a vengeance!

```
foldl  ::  (a -> b -> a) -> a -> [b] -> a
```

`(a -> b -> a)`	combines accumulator and value
`a`	initial accumulator
`[b]`	list of values
`a`	result

"reduce"

Codelab - `04_abstractions`

directory : `04_abstractions`

change : `src/Codelab.hs`
(replace `codelab` with implementation)

check with : `make`

The end of the theoretical part!

Questions?

Links

- tryhaskell.org
- learnyouahaskell.com
- book.realworldhaskell.org
- haskellbook.com
- haskell.org/hoogle/
- Pragmatic types: types vs tests
- Why functional programming matters

directory : `05_maybe`

change : `src/Codelab.hs`
(replace `codelab` with implementation)

check with : `make`

Codelab - `06_rps` (bonus)

directory : `06_rps`

change : `src/Codelab.hs`
(replace `codelab` with implementation)

check with : `make`

g2g

The end!

Questions?