Dollars, dots and parameters

LISP/Python:

$$wordList\ docs = sort\ (nub\ (concat\ (map\ words\ docs)))$$

A little better with dollars:

$$wordList\ docs = sort\ \$\ nub\ \$\ concat\ \$\ map\ words\ docs$$

Even better:

$$wordList\ docs = sort \cdot nub \cdot concat \cdot map\ words\ \$\ docs$$

Equivalently:

$$wordList\ docs = (sort \cdot nub \cdot concat \cdot map\ words)\ docs$$

Best:

$$wordList = sort \cdot nub \cdot concat \cdot map\ words$$

How to do input and output with pure functions?

I/O is not a side effect!

- IO *a* is a “container” for values of type *a* which describes what actions to do to obtain the value.
- We have tools to combine IO descriptions (simpler programs) to larger ones (more complex programs).
- *main* :: IO () describes actions of the whole program.
- To run the program, GHC runtime ‘follows’ the description generated by *main*.

IO — example

```
main =
  do putStr "Give me a number: "
     a ← getLine
     putStrLn "Numbers starting from zero: "
     printNums 0 (read a)

printNums :: Int → Int → IO ()
printNums a b
  | a ⩾ b       = return ()
  | otherwise = do print a
                   printNums (a + 1) b
```

do-syntax “glues” the individual IO descriptions together.

A very imperfect allegory, just for illustration

Haskell:

```
main = do
  putStrLn "hello"
  x ← getLine
  putStrLn x
```

Very rough and bad pseudo-C++ work-alike:

```
template struct Seq<A,B> {
  A a; B b;
  auto get() { a.get(); b.get(); }
};
template struct Glue<A,B> {
  A a; B b;
  auto get() { return b.get(a.get()); }
};
const auto program =
  Seq(PutStrLnConst("hello"),
      Glue(GetLine,
           PutStrLn));
void main () { program.get() }
```

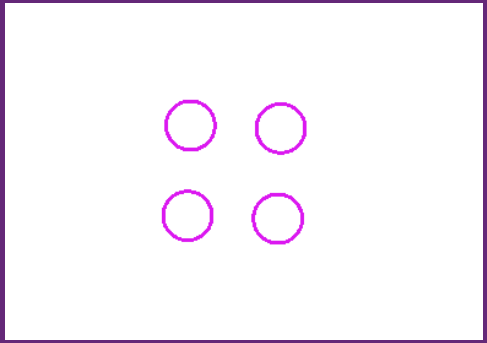
But wait!

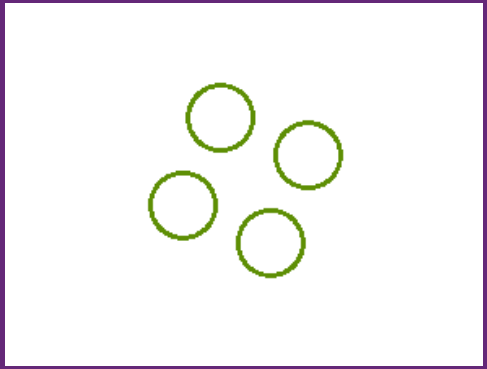
Is this getting too theoretical?

Get something cool on screen in 10 minutes!

```
import Graphics.Gloss

main = animate FullScreen white $ λt →
  let blink ph sp = (1 + sin (ph + t * sp)) / 2
  in Pictures $ flip map [1..4] $ λx →
       Rotate (90 * x + 20 * t) $
       Translate (80 + 40 * cos t) 0 $
       Color (makeColor (blink 0 1)
                        (blink 3 1)
                        (blink 0 π)
                        1) $
       ThickCircle 20 3
```





Practicals

```
  - quick sort
qsort [ ]       = [ ]
qsort (x : xs) = qsort a ++ x : qsort b
  where a = filter (<x) xs
        b = filter (>= x) xs

  - run-length encoding
rle = map (λl → (head l, length l)) . group

  - make lists with equivalent elements
group [ ]       = [ ]
group l@(x: _) = takeWhile (== x) l
                 : group (dropWhile (== x) l)
```

Can you guess the types?

Practicals

```
  - quick sort
qsort [ ]       = [ ]
qsort (x : xs) = qsort a ++ x : qsort b
  where a = filter (<x) xs
        b = filter (>= x) xs

  - run-length encoding
rle = map (λl → (head l, length l)) . group

  - make lists with equivalent elements
group [ ]       = [ ]
group l@(x: _) = takeWhile (== x) l
                 : group (dropWhile (== x) l)


sort :: Ord a => [a] → [a]
rle :: Eq a => [a] → [(a, Int)]
group :: Eq a => [a] → [[a]]
```

Practicals — laziness

$$findMin = head \cdot quicksort$$

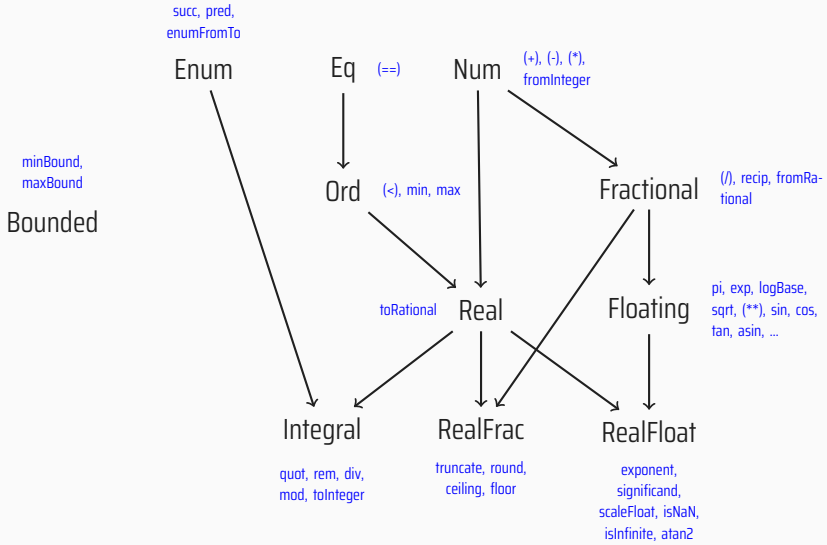
What is the time complexity of *findMin*?

Type classes, functors and IO

Simple type classes

type class	operations
Eq	$(\equiv), (\not\equiv)$
Ord	$(<), (\leqslant)$, *compare*, ...
Enum	[1 . . 10], ['a' . . 'z'], [False . .]
Num	$(+), (-), (*)$, *abs*, ...
Integral	*div*, *mod*, ...
Fractional	$(/)$, ...
Floating	π, *sin*, *cos*, *exp*, *log*, *logBase*, *sqrt*, ...
Show	*show*
Read	*read*

Number hierarchy



Side note: The number hierarchy versus computers

toRational 0.1
  ⇝ 3602879701896397 % 36028797018963968

ceiling (1.0 / 0.0)
  ⇝ 179769313486231590772930519078902473361 79 …

(0.1 + 0.2 – 0.3) / (0.2 – 0.3 + 0.1)
  ⇝ 2.0

(This is not a Haskell-specific problem. Don't use floats for exact stuff.)

Overloading

Type classes enable function overloading in the presence of full type inference.

C++-style ("ad-hoc") overloading is only possible with unidirectional type inference; in Haskell it would be NP-complete to resolve.

```
class MyClass a where
  myfun :: a → a → a

instance MyClass Int where
  myfun = (+)

instance MyClass String where
  myfun = (++)

myfun 1 3          ⇝ 4
myfun "wo" "rd"    ⇝ "word"
```

That's it!

At this point you have already seen all important Haskell syntax.

(I.e., all syntax required to pass this course.)

Overloading — example

```
instance Num Bool where
  (+)         = (∨)
  (*)         = (∧)
  negate      = ¬
  abs         = id
  signum      = id
  fromInteger = (≠ 0)
```

```
True + False                ⇝ True
sum [0, 1, 2] :: Bool       ⇝ True
product [0, 1, 2] :: Bool   ⇝ False
```

Overloading — example 2

```
newtype I7 = I7 Int
instance Num I7 where
  (+)           = i7lift2 (+)
  (−)           = i7lift2 (−)
  (∗)           = i7lift2 (∗)
  negate        = i7lift negate
  abs           = id
  signum (I7 0) = I7 0
  signum _      = I7 1
  fromInteger i = I7 $ i `mod` 7

i7lift f  (I7 a)        = I7 (f a `mod` 7)
i7lift2 f (I7 a) (I7 b) = I7 (f a b `mod` 7)
```

$(1 + 2 + 3 + 4) :: I7 \quad \rightsquigarrow \; ?$

Overloading — example 3

```
data Infinite a = MinusInf | Finite a | PlusInf

instance Ord a ⇒ Ord (Infinite a) where
  compare (Finite a) (Finite b) = compare a b
  compare MinusInf   MinusInf   = EQ
  compare MinusInf   _          = LT
  compare _          MinusInf   = GT
  compare PlusInf    PlusInf    = EQ
  compare PlusInf    _          = GT
  compare _          PlusInf    = LT

MinusInf < Finite 5   ⇝ True
```

Overloading — example 4

$$
\begin{aligned}
&\textbf{instance}\ \mathrm{Num}\ a \Rightarrow \mathrm{Num}\ [a]\ \textbf{where}\\
&\quad (+) \quad\ = \mathit{zipWith}\ (+)\\
&\quad (-) \quad\ = \mathit{zipWith}\ (-)\\
&\quad (*) \quad\ = \mathit{zipWith}\ (*)\\
&\quad \mathit{negate} \quad = \mathit{map}\ \mathit{negate}\\
&\quad \mathit{abs} \quad\ = \mathit{map}\ \mathit{abs}\\
&\quad \mathit{signum} \quad = \mathit{map}\ \mathit{signum}\\
&\quad \mathit{fromInteger} = \mathit{repeat} \cdot \mathit{fromInteger}
\end{aligned}
$$

$$3 * [1,2,3] + 2 \quad \leadsto [5,8,11]$$

$$\mathit{fibs} = 0 : 1 : \mathit{fibs} + \mathit{tail}\ \mathit{fibs}$$

Typeclasses implement extensible open unions

Closed union (with ADTs):

```
data Shape
  = HalfSquareTriangle {side :: Float}
  | Circle {radius :: Float}

area HalfSquareTriangle {side}
  = side^2 / 2
area Circle {radius}
  = π * radius^2
```

...Shape can be extended only by redefinition.

Open union (with TCs):

```
class Shape s where area :: s → Float

data HalfSquareTriangle =
  HalfSquareTriangle {side :: Float}

instance Shape DiagonalTriangle where
  area t = side t^2 / 2

data Circle = Circle {radius :: Float}

instance Shape Circle where
  area c = π * radius c^2
```

...anyone can add as many implementations as desirable, even in other modules.

Typeclasses implement dependency injection

```
data Expr
  = Add Expr Expr
  | Negate Expr
  | Mult Expr Expr
  | Literal Integer
  | Abs Expr
  | Signum Expr
  deriving (Show)
```

```
instance Num Expr where
  (+)         = Add
  (*)         = Mult
  negate      = Negate
  abs         = Abs
  signum      = Signum
  fromInteger = Literal

sum [1, 2, 3] :: Expr
  ⇝ Add (Add (Add (Literal 0) (Literal 1))
             (Literal 2))
        (Literal 3)
```

Typeclasses are not an interface!

OOP: `INumeric doSomething(INumeric a)`

vs.

Haskell: $doSomething :: \text{Num } a \Rightarrow a \rightarrow a$

What's the difference?

Typeclasses are not subtyping!

C-like: `(long)(double)(int)(float)5` is OK, integers can be converted automatically.

Haskell: `((5 :: Float) :: Int)` is an error!

(...but wait, (5::Float) works, so how does the integer 5 get converted to a Float?)

Typeclasses are not subtyping!

C-like: `(long)(double)(int)(float)5` is OK, integers can be converted automatically.

Haskell: ((5 :: Float) :: Int) is an error!

(...but wait, (5::Float) works, so how does the integer 5 get converted to a Float?)

Desugaring of literals: 5 is rewritten to $fromInteger\ 5$, where 5 is a fixed Integer value.

$fromInteger :: \text{Num}\ a \Rightarrow \text{Integer} \rightarrow a$

Outcome: $5 :: \text{Num}\ a \Rightarrow a$

(the conversion is typically optimized out)

Typeclasses do not implement run-time polymorphism!

Without additional effort, all type choices *are static*.

To work that around, we need to add the typeclass to data definition manually.

```
data SomeShape = ∀s.Shape s ⇒ SomeShape s

test :: [SomeShape]
test = [SomeShape $ Star 5 10,
        SomeShape $ Circle 10,
        SomeShape $ BlobbySquare 1 2 3 4 5]
```

- SomeShape is usually called an “existential type”.
- The construction is opaque (inner type information is *lost*) and should be avoided.

Semigroups and Monoids (Universal Glue)

class Semigroup *a* **where**
  $(\diamond) :: a \to a \to a$

class Semigroup *a* $\Rightarrow$ Monoid *a* **where**
  mempty :: *a*

Semigroups give a generic operation for gluing stuff together (written as <>):

$[1,2] \diamond [4,5] \rightsquigarrow [1,2,4,5]$
`"some"` $\diamond$ `"thing"` $\rightsquigarrow$ `"something"` - also works for Text and ByteString
Just $[1,2] \diamond$ Nothing $\diamond$ Just $[3] \rightsquigarrow$ Just $[1,2,3]$
EQ $\diamond$ LT $\diamond$ GT $\rightsquigarrow$ LT
mempty :: Maybe *a* $\rightsquigarrow$ Nothing

There are multiple monoids over numeric types:

Sum 5 $\diamond$ Sum 3 $\rightsquigarrow$ Sum 8
Product 5 $\diamond$ Product 3 $\rightsquigarrow$ Product 15
(*mempty* :: Product Int) $\rightsquigarrow$ Product 1

Container-ness

data Tree a = Nil | Branch (Tree a) a (Tree a)

map (+1) [1, 2, 3] $\leadsto$ [2, 3, 4]

map (+1) (Branch Nil 1 (Branch Nil 2 Nil)) – error

Can we overload *map*?

Container-ness

data Tree a = Nil | Branch (Tree a) a (Tree a)

map (+1) [1, 2, 3] $\leadsto$ [2, 3, 4]

map (+1) (Branch Nil 1 (Branch Nil 2 Nil)) - error

Can we overload *map*?

class Functor f **where** *fmap* :: $(a \to b) \to (f\ a \to f\ b)$

Caution: the kind of f is $* \to *$.

instance Functor Tree **where**

fmap _ Nil = Nil

fmap f (Branch *l a r*) = Branch (*fmap f l*) (*f a*) (*fmap f r*)

fmap (+1) [1, 2, 3] $\leadsto$ [2, 3, 4]

fmap (+1) (Branch Nil 1 (Branch Nil 2 Nil)) $\leadsto$ Branch Nil 2 (Branch Nil 3 Nil)

Container-ness

$(\langle\$\rangle) = \mathit{fmap}$

$(*2)$	$\langle\$\rangle\ [2,3,4]$	$\rightsquigarrow [4,6,8]$
$(+1) \cdot (+2)$	$\langle\$\rangle$ Just 5	$\rightsquigarrow$ Just 8
$(+1) \cdot (+2)$	$\langle\$\rangle$ Nothing	$\rightsquigarrow$ Nothing
uncurry gcd	$\langle\$\rangle$ Just (1024, 768)	$\rightsquigarrow$ Just 256
fmap show	$\langle\$\rangle\ [$Nothing, Just 3, Just 4$]$	$\rightsquigarrow [$Nothing, Just `"3"`, Just `"4"`$]$


Vorsicht
Funktor
2 m Abstand halten

Reasonable functors

$$fmap\ id \equiv id$$

$$fmap\ f \cdot fmap\ g \equiv fmap\ (f \cdot g)$$

Functions as data: Functions glue together!

Functions are **containers for results**.

```
instance Semigroup b ⇒ Semigroup (a → b) where
  f ⋄ g = λa → f a ⋄ g a
instance Monoid b ⇒ Monoid (a → b) where
  mempty = λ_ → mempty
```

$$
\begin{aligned}
&(\mathit{drop}\ 3\ [1,2,3,4,5] \diamond \mathit{take}\ 3\ [1,2,3,4,5]) && \rightsquigarrow [4,5,1,2,3]\\
&(\mathit{drop}\ 3 \diamond \mathit{take}\ 3)\ [1,2,3,4,5] && \rightsquigarrow [4,5,1,2,3]\\
&(\mathit{drop} \diamond \mathit{take})\ 2\ [1,2,3,4,5] && \rightsquigarrow [3,4,5,1,2]
\end{aligned}
$$

Functions as data: Functions are containers!

Similarly:

```
instance Functor ((→) p) where
  fmap = (∘)

fmap :: (a → b) → ((p → a) → (p → b))    - specialized to functions

          (∗3)       :: Int → Int
show ⟨$⟩ (∗3)       :: Int → String
show ⟨$⟩ (∗3) $ 3   ⇝ "9"
```

(Intuition: You “open” the “function container” by supplying a parameter.)

Functions as data: Functions are numbers!

```
instance Num b ⇒ Num (a → b) where
  (f + g) x          = f x + g x
  (f * g) x          = f x * g x
  negate f x         = negate (f x)
  signum f x         = signum (f x)
  abs f x            = abs (f x)
  fromInteger n x = fromInteger n
```

```
(sin^2 + cos^2) 4.64234   ⇝ 1.0
(sin + 1) 1               ⇝ 1.8414709848078965
(sin + 1) π               ⇝ 1.0000000000000002
(negate negate) 5         ⇝ 5
3 7                       ⇝ 3
((+) * (−)) 7 5           ⇝ 24
```

Nomenclature: Point-free style

If you do not move the arguments around yourself, you can't mess it up.

Danger: Various funny people call this a 'point-less style'.

Functions vs. containers

$$
\begin{aligned}
(\$) &:: (a \to b) \to (\ a \ \to \ b\ ) \\
map &:: (a \to b) \to ([a] \to [b]) \\
fmap &:: (a \to b) \to (f\ a \ \to f\ b\ ) \\
\_ &:: f\ (a \to b) \to (f\ a \ \to f\ b\ )
\end{aligned}
$$

...so what if the function $(a \to b)$ is also in some container?

Functions vs. containers

Example problem:

$$\begin{aligned} \mathit{solveProblem} = \textbf{let}\ & a = \mathit{getProgram} :: d \to r \\ & b = \mathit{getData} :: d \\ \textbf{in}\ & a\ b \end{aligned}$$

Errors might cause either program or data to be missing.

Example problem:

$$
\begin{aligned}
\mathit{solveProblem} = \textbf{let}\ & a = \mathit{getProgram} :: d \to r \\
& \quad b = \mathit{getData} :: d \\
\textbf{in}\ & a\ b
\end{aligned}
$$

Errors might cause either program or data to be missing. We can use `Maybe` to manage the error:

$$
\begin{aligned}
& \mathit{ap}\ (\text{Just}\ l)\ (\text{Just}\ r) = \text{Just}\ (l\ r) \\
& \mathit{ap}\ \_\quad\ \_\quad = \text{Nothing}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{solveProblem} = \textbf{let}\ & a = \mathit{getProgram} :: \text{Maybe}\ (d \to r) \\
& \quad b = \mathit{getData} :: \text{Maybe}\ d \\
\textbf{in}\ & a\ `\mathit{ap}`\ b
\end{aligned}
$$

We did not need to write the `if` in the actual code! (Danger avoided!)

Functions vs. containers

Another problem:

$$myNumbers\ x = [x + 1, x - 1, x * 1, x + 2, x - 2, x * 2, \ldots]$$

We can make this nicer again by tearing out the repeating pattern.

Functions vs. containers

Another problem:

$$myNumbers\ x = [x + 1, x - 1, x * 1, x + 2, x - 2, x * 2, \ldots]$$

We can make this nicer again by tearing out the repeating pattern.

```
funs `ap` args = concat $ map (λfun → map fun args) funs
myNumbers x = [(+) x, (-) x, (*) x] `ap` [1..]
```

We did not need to write the loop! (Danger avoided!)

Applicative

The concept of applying “wrapped” functions is captured as such:

$$\textbf{class}\ \text{Functor}\ f \Rightarrow \text{Applicative}\ f\ \textbf{where}$$
$$(\langle * \rangle) :: f\ (a \to b) \to f\ a \to f\ b$$
$$pure :: a \to f\ a$$

Applied to the examples:

$$solveProblem = getProgram\ \langle * \rangle\ getData$$
$$myNumbers = [\,(+)\ x, (-)\ x, (*)\ x\,]\ \langle * \rangle\ [1..]$$

More arguments: Why does this work?

$$myNumbers' = [\,(+), (-), (*)\,]\ \langle * \rangle\ [x] \qquad \langle * \rangle\ [1..]$$
$$myNumbers'' = [\,(+), (-), (*)\,]\ \langle * \rangle\ pure\ x\ \langle * \rangle\ [1..]$$

(we read ‘pure’ as ‘this is innocent and won’t do any tricks’)

Applicative

```
Just show   ⟨*⟩ Just 3                          ⇝ Just "3"
(pure show ⟨*⟩ pure 3) :: Maybe String          ⇝ Just "3"
(pure show ⟨*⟩ pure 3) :: [String]              ⇝ ["3"]
Nothing     ⟨*⟩ Just 3                          ⇝ Nothing
Just show   ⟨*⟩ Nothing                         ⇝ Nothing
Just gcd    ⟨*⟩ Just 1024 ⟨*⟩ Just 768          ⇝ Just 256
```

Applicative instance semantics

instance	semantics
Maybe	failing
Either I	failing with an error of type I
[ ]	nondeterministic computation (multiple results)
$(,)\ a$	computation with annotated values
$(\rightarrow)\ p$	computation with a global parameter
⋮	⋮

Applicative examples

Just (,) ⟨∗⟩ Just 3 ⟨∗⟩ *pure* 5 ⇝ Just (3, 5)

(,) ⟨$⟩ *pure* 3 ⟨∗⟩ Just 5 ⇝ Just (3, 5)

((+) ⟨$⟩ *pure* 3 ⟨∗⟩ *pure* 7) :: Either String Int

⇝ Right 10

(+) ⟨$⟩ Right 3 ⟨∗⟩ *pure* 7

⇝ Right 10

(+) ⟨$⟩ *pure* 3 ⟨∗⟩ Left `"sorry error"`

⇝ Left `"sorry error"`

[(+1), (+2), *negate*] ⟨∗⟩ [2, 3, 4]

⇝ [3, 4, 5, 4, 5, 6, −2, −3, −4]

rle = *map* ((,) ⟨$⟩ *head* ⟨∗⟩ *length*) ∘ *group*

rle' = *map* (*liftA2* (,) *head* *length*) ∘ *group*

Reasonable applicative functors

$$\mathit{pure}\ \mathit{id} \langle * \rangle\ a \equiv a$$

$$\mathit{pure}\ (a\ b) \equiv \mathit{pure}\ a \langle * \rangle\ \mathit{pure}\ b$$

$$a \langle * \rangle\ \mathit{pure}\ b \equiv \mathit{pure}\ (\$b) \langle * \rangle\ a$$

$$a \langle * \rangle\ (b \langle * \rangle\ c) \equiv \mathit{pure}\ (\cdot) \langle * \rangle\ a \langle * \rangle\ b \langle * \rangle\ c$$

Applicatives vs. Monoids

Like monoids glue values together, applicatives glue containers together.

Haskell tuple $(,)$ serves as a gluable annotation for containers, combining the concepts.

$$\begin{aligned}&\textbf{instance}\ \text{Monoid}\ a \Rightarrow \text{Applicative}\ ((,)\ a)\ \textbf{where}\\ &\quad \mathit{pure}\ x \qquad\quad = (\mathit{mempty}, x)\\ &\quad (u, f)\ \langle * \rangle\ (v, x) = (u \diamond v, \quad f\ x)\end{aligned}$$

$$\texttt{("hello ",(+))}\ \langle * \rangle\ \mathit{pure}\ 20\ \langle * \rangle\ \texttt{("world!",2006)}$$
$$\rightsquigarrow\ \texttt{("hello world!",2026)}$$

(This is useful for keeping a “log” of what was computed.)

Functions vs. containers, final level!

$$
\begin{aligned}
(\$) &:: (a \to b) \to (a \to b) \\
fmap &:: (a \to b) \to (f\ a \to f\ b) \\
(\langle * \rangle) &:: f\ (a \to b) \to (f\ a \to f\ b) \\
\_ &:: (a \to f\ b) \to (f\ a \to f\ b)
\end{aligned}
$$

Functions typically *induce new semantics* (e.g., they fail or produce more solutions).

Can we somehow convert these functions to ones that can continue working with some pre-existing semantics?

Alternative view of the same problem (we could have tried with $fmap$ above):

$$join :: f\ (f\ a) \to f\ a$$

Joining Applicatives

Example with Maybe: We have some functions that may fail, and we want to safely glue them to one function that manages the failure properly.

```
lookup  :: Eq a ⇒ a → [(a, b)] → Maybe b

lookup4 :: Eq a ⇒ a → [(a, a)] → Maybe a
lookup4 a l = case lookup a l of
  Nothing → Nothing
  Just a' → case lookup a' l of
    Nothing → Nothing
    Just a'' → case lookup a'' l of
      Nothing → Nothing
      Just a''' → lookup a''' l
```

SW Engineering: Eyes hurt! Let's tear out the repeating code piece!

andThen

```
andThen :: Maybe a → (a → Maybe b) → Maybe b
andThen Nothing _ = Nothing
andThen (Just a) f = f a

lookup4 a l = lookup a l `andThen` λa →
              lookup a l `andThen` λa →
              lookup a l `andThen` λa →
              lookup a l
```

Slightly better!

andThen (on steroids)

```
infixr 1 `andThen`

lookup4 a l = go a `andThen` go
                   `andThen` go
                   `andThen` go
  where go a = lookup a l
```

What about nondeterminism?

Task:

- A robot can move in some directions (depending on where it stands now),
- there it can do some actions (depending on the new position),
- and in turn this gives a lot of possibilities of where it can stand in the end.
- Find all these possibilities.

```
directionsAt :: Location                → [Direction]
go           :: Direction → Location → Location
actionsAt    :: Location                → [Action]
act          :: Action → Location       → Location
```

```
directionsAt :: Location                → [Direction]
go           :: Direction → Location    → Location
actionsAt    :: Location                → [Action]
act          :: Action → Location       → Location
```

```
possibleResults :: Location → [Location]
possibleResults loc = concat $
  map (λdir → let newloc = go dir loc
              in map (λaction → act action newloc)
                     (actionsAt newloc)
      )
      (directionsAt loc)
```

...this code does spark zero joy.

```
withTheseCases :: [a] → (a → [b]) → [b]
withTheseCases a f = concat (map f a)

infixr 2 `withTheseCases`
```

```
withTheseCases :: [a] → (a → [b]) → [b]
withTheseCases a f = concat (map f a)

infixr 2 `withTheseCases`


possibleResults loc =
  directionsAt loc
   `withTheseCases` λdir →
  let newloc = go dir loc
  in actionsAt newloc
       `withTheseCases` λaction →
      [act action newloc]
```

(the last line returns a single possibility)

Generic andThen

$$
\begin{array}{lll}
\mathit{andThen} & :: \text{Maybe } a \to (a \to \text{Maybe } b) & \to \text{Maybe } b \\
\mathit{withTheseCases} & :: [a] \to (a \to [b]) & \to [b] \\
(\gg\!=) & :: m\ a \to (a \to m\ b) & \to m\ b
\end{array}
$$

Generic andThen

$$
\begin{array}{lll}
\mathit{andThen} & :: \text{Maybe } a \to (a \to \text{Maybe } b) & \to \text{Maybe } b \\
\mathit{withTheseCases} & :: [a] \to (a \to [b]) & \to [b] \\
(\gg\!=) & :: m\ a \to (a \to m\ b) & \to m\ b
\end{array}
$$

$$
\begin{array}{lll}
(\$) & :: & (a \to b) \to (a \to b) \\
\mathit{fmap} & :: \text{Functor } f \Rightarrow & (a \to b) \to (f\ a \to f\ b) \\
(\langle * \rangle) & :: \text{Applicative } f \Rightarrow & f\ (a \to b) \to (f\ a \to f\ b) \\
\mathit{flip}\ (\gg\!=) & :: \text{Monad } m \Rightarrow & (a \to m\ b) \to (m\ a \to m\ b)
\end{array}
$$

Generic *andThen* and *withTheseCases* are called $(\gg\!=)$, read “bind”.

Monads

Name: applicative monoid (almost).

$$
\begin{aligned}
&\textbf{class}\ \text{Applicative}\ m \Rightarrow \text{Monad}\ m\ \textbf{where}\\
&\quad (\gg\!=) \quad :: m\ a \to (a \to m\ b) \to m\ b\\
&\quad (\gg) \quad :: m\ a \to \quad m\ b \to m\ b\\
&\quad \mathit{return} :: a \qquad\qquad \to m\ a
\end{aligned}
$$

$(\gg)$ throws away the 'result', only retaining the changes in the 'wrapping'.

Alternatively, one could also define monads with:

$$
\begin{aligned}
&\mathit{join} :: \text{Monad}\ m \Rightarrow m\ (m\ a) \to m\ a\\
&(a \gg\!= f) = \mathit{join}\ (f \langle\$\rangle\ a)
\end{aligned}
$$

Reasonable monads

$$return\ a \gg\!= f \equiv f\ a$$

$$a \gg\!= return \equiv a$$

$$a \gg\!= (\lambda x \rightarrow b\ x \gg\!= c) \equiv (a \gg\!= b) \gg\!= c$$


MONADS!
@impurepics
NO! PLEASE, STOP!
THEY'RE STILL CHILDREN

Imperative syntax

Because writing $\gg=$ and $\gg$ is a nuisance, Haskell has the imperative **do**-syntax.

$$
\begin{aligned}
fn = \textbf{do}\ & a \leftarrow f \\
& b \leftarrow g \\
& h\ a\ b \\
& h\ b\ a
\end{aligned}
$$

...gets rewritten to:

$$
\begin{aligned}
fn = f \gg= \lambda a \rightarrow\ & \\
g \gg= \lambda b \rightarrow\ & \\
h\ a\ b \gg\ & \\
h\ b\ a &
\end{aligned}
$$

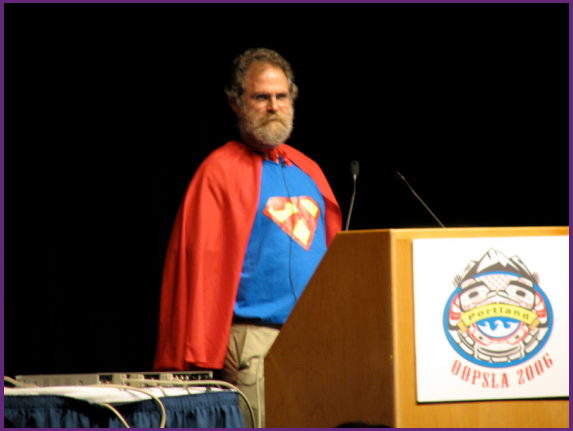
Imperative syntax

```
lookup4 a l =
  do a ← lookup a l
     a ← lookup a l
     a ← lookup a l
     a ← lookup a l
     return a
```

```
possibleResults loc =
  do dir ← directionsAt loc
     let newloc = go dir loc
     action ← actionsAt newloc
     return $ act action newloc
```

Philip Lee Wadler

Philip Lee Wadler



The Lambdaman

Side note: list comprehension

The “special set-like comprehension syntax” for lists is actually a monad syntax:

```
possibleResults loc =
  [act action newloc | dir ← directionsAt loc,
                       let newloc = go dir loc,
                       action ← actionsAt newloc]
```

More typical uses:

```
map f l = [f a | a ← l]
filter f l = [a | a ← l, f a]
pythagoreanTriples = [(a, b, c) | c ← [1..], b ← [1..c], a ← [1..b],
                                  c^2 ≡ a^2 + b^2]
```

(Note: the “conditions” are desugared using *guard* from Control.Monad)

For completeness: Maybe monad

```
instance Functor Maybe where
  fmap f (Just x) = Just (f x)
  fmap _ Nothing = Nothing
instance Applicative Maybe where
  pure = Just
  Just f ⟨*⟩ Just a = Just (f a)
  _ ⟨*⟩ _ = Nothing
instance Monad Maybe where
  return = pure
  Just a >>= f = f a
  _ >>= _ = Nothing
```

For completeness: List monad

```
instance Functor [ ] where
  fmap = map
instance Applicative [ ] where
  pure x = [x]
  fs ⟨*⟩ as = concatMap (flip map as) fs
instance Monad [ ] where
  return = pure
  as >>= f = concatMap f as
```

Extra example: tuple is a “comments” monad

```
type WithLog a = ([String], a)

log :: String → WithLog ()
log x = ([x], ())

solveQuad :: Floating a ⇒ a → a → a → WithLog [a]
solveQuad a b c = do
  let d = b^2 – 4 * a * c
  log $ "D = " ++ show d
  if d < 0 then do log "no solutions"
                   pure []
  else if d ≡ 0 then do log "single solution"
                        pure [–b / (2 * a)]
                else do log "two solutions"
                        pure [(–b + sqrt d) / (2 * a)
                              ,(–b – sqrt d) / (2 * a)]
```

```
putStrLn :: String → IO ()
getLine  :: IO String

main :: IO ()
main = do putStrLn "give a name!"
          name ← getLine
          putStrLn $
            "well hello this is " ++ name
```

IO is no magic!

```
putStrLn :: String → IO ()
getLine  :: IO String

main :: IO ()
main = putStrLn "give a name!" ≫
       getLine ≫= λname →
       putStrLn ("well hello this is " ++ name)
```

Teaser: Because IO is a monad...

```
main :: IO ()
main = something ← length ⟨$⟩ getLine
       putStrLn (show something)
```

OK so how do we create the IO?

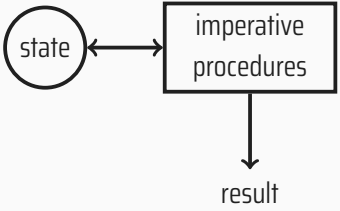
Idea: IO action must manipuate some global state! Let's make a monad that can save the state first, and we will be able to build on top of that later.

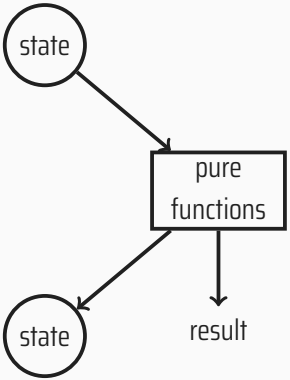
Requirements:

- *put* :: *s* → StateMonad ()
- *get* :: StateMonad *s*
- this should return True:

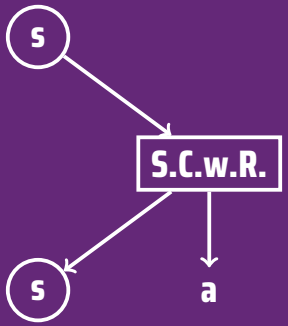
```
do put a
   b ← get
   return $ a ≡ b
```

Stateful computation in pure functions





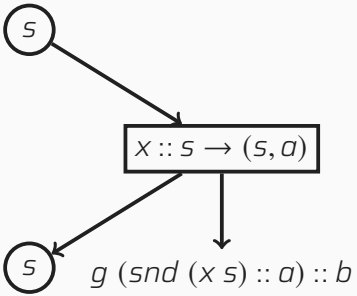
```
newtype State s a =
  StateChangeWithResult (s → (s, a))
```



Stateful computation is container-ish (like other functions!)

instance Functor (State *s*) **where**

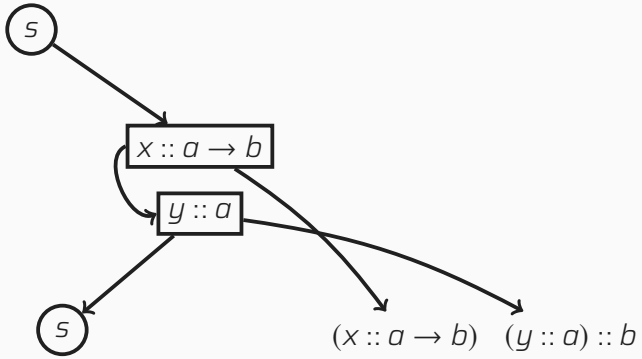
fmap g (SCwR *x*) =



Stateful computation is applicative!

instance Applicative (State *s*) **where**

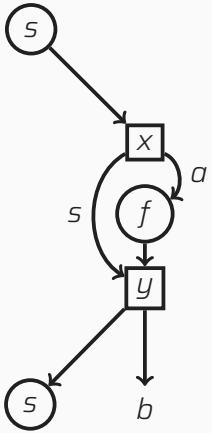
SCwR *x* ⟨∗⟩ SCwR *y* =



Stateful computation is monadic!

instance Monad (State *s*) **where**

SCwR $x \gg\!= f =$



State implementation

```
newtype State s a = SCwR (s → (s, a))

instance Functor (State s) where
  fmap func (SCwR f) =
    SCwR (fmap func . f)

instance Applicative (State s) where
  pure a = SCwR (λs → (s, a))
  SCwR f <*> SCwR v =
    SCwR (λs → let (s1, a1) = f s
                   (s2, a2) = v s1
               in (s2, a1 a2))
```

State implementation

```
instance Monad (State s) where
  return = pure
  SCwR f1 >>= f2 =
    SCwR (λs → let (s', a) = f1 s
                   SCwR f3 = f2 a
               in f3 s')
```

Usual state manipulation helpers

"Set" state to a new value:

$$put :: s \to \mathsf{State}\ s\ ()$$
$$put\ newVal = \mathsf{SCwR}\ (\lambda\_ \to (newVal, ()))$$

"Fetch" the current stored value and return it:

$$get :: \mathsf{State}\ s\ s$$
$$get = \mathsf{SCwR}\ (\lambda s \to (s, s))$$

Apply a function to the stored value:

$$modify :: (s \to s) \to \mathsf{State}\ s\ ()$$
$$modify\ f = \mathsf{SCwR}\ (\lambda s \to (f\ s, ()))$$

"Execute" a stateful computation to only extract result:

$$execState\ (\mathsf{SCwR}\ f)\ s = snd\ (f\ s)$$

Imperative factorial!

```
fact n = execState (factS n) 1

factS n
  | n ⩽ 1 = do result ← get
               return result
  | otherwise = do modify (∗n)
                   factS (n – 1)
```

Randomness is impure because it updates the generator

```
getRand :: Int → State Int Int
getRand max =
  do newSeed ← (12345+) • (1103515245*) ⟨$⟩ get
     put newSeed
     return $ (newSeed `div` 65536) `mod` max
```

(implementation taken from `man 3 rand`)

Using State as an Applicative

```
get2BoringRand :: Int → State Int (Int, Int)
get2BoringRand max = do
  a ← getRand max
  b ← getRand max
  return (a, b)
```

Using State as an Applicative

```
get2BoringRand :: Int → State Int (Int, Int)
get2BoringRand max = do
  a ← getRand max
  b ← getRand max
  return (a, b)
```

```
get2Rand max = pure (,) ⟨*⟩ getRand max ⟨*⟩ getRand max
```

Using State as an Applicative

```
get2BoringRand :: Int → State Int (Int, Int)
get2BoringRand max = do
  a ← getRand max
  b ← getRand max
  return (a, b)
```

```
get2Rand max = pure (,) ⟨*⟩ getRand max ⟨*⟩ getRand max
```

```
get2Rand' max = (,) ⟨$⟩ getRand max ⟨*⟩ getRand max
```

Using State as an Applicative

```
getNRand max 0 = pure []
getNRand max n = (:) <$> getRand max
                     <*> getNRand max (n – 1)
```

Using State as an Applicative

$$
\begin{aligned}
&\mathit{getNRand}\ \mathit{max}\ 0 = \mathit{pure}\ [\,] \\
&\mathit{getNRand}\ \mathit{max}\ n = (:)\ \langle\$\rangle\ \mathit{getRand}\ \mathit{max} \\
&\qquad\qquad\qquad\qquad \langle*\rangle\ \mathit{getNRand}\ \mathit{max}\ (n-1)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{getNRand'}\ \mathit{max}\ n = \mathit{mapM}\ (\lambda\_ \rightarrow \mathit{getRand}\ \mathit{max})\ [1\,..\,n] \\
&\mathit{getNRand''}\ \mathit{max}\ n = \mathit{traverse}\ (\mathit{const}\ \$\ \mathit{getRand}\ \mathit{max})\ [1\,..\,n]
\end{aligned}
$$

Using State as an Applicative

```
getNRand max 0 = pure []
getNRand max n = (:) <$> getRand max
                     <*> getNRand max (n - 1)

getNRand' max n = mapM (λ_ → getRand max) [1..n]
getNRand'' max n = traverse (const $ getRand max) [1..n]

getNRand''' max n = replicateM n $ getRand max
```

Using State as an Applicative

```
getNRand max 0 = pure []
getNRand max n = (:) <$> getRand max
                     <*> getNRand max (n - 1)

getNRand' max n = mapM (λ_ → getRand max) [1..n]
getNRand'' max n = traverse (const $ getRand max) [1..n]

getNRand''' max n = replicateM n $ getRand max

getNRand'''' = flip replicateM . getRand
```

The truth about IO in a pure language

```
$ ghci
GHCi, version 8.2.2: http://www.haskell.org/ghc/
Prelude> :i IO
newtype IO a
  = GHC.Types.IO
      (GHC.Prim.State# GHC.Prim.RealWorld
         -> (# GHC.Prim.State# GHC.Prim.RealWorld, a #))
```

But how much memory do you need to store the RealWorld?

But how much memory do you need to store the `RealWorld`?

It is just a “token” of type `Void`.

So what is this monad thing again?

So what is this monad thing again?