

Haskell

1 / 234

Haskell features (from module 01):

- ▶ Purity
- ▶ Laziness
- ▶ Type classes
- ▶ Monads

2 / 234

Some historical milestones:

David Turner's SASL (1976),
KRC (1982), Miranda (1985)

Many other lazy functional
languages

3 / 234

First committee meeting 1987
(last one 1999)

Haskell 1.0 Report 1 April 1990

Glasgow Haskell Compiler (GHC)
1992

Early 1999: Haskell 98 report

4 / 234

Haskell initially resembles OCaml
with fewer keywords.

It avoids some by making
whitespace significant.

5 / 234

Value definitions:

$\langle id \rangle = \langle expr \rangle$

$x = 3$

$y = x * x + 3 * x + 4$

6 / 234

Function definitions:

$\langle id \rangle \langle param \rangle \langle param \rangle \dots = \langle expr \rangle$

sqr x y = x*x + y*y

7/234

Multipart function definitions:

$\langle id \rangle \langle pattern \rangle \langle pattern \rangle \dots = \langle expr \rangle$

pred 0 = 0
pred n = n-1

8/234

Offside rule:

All definitions (in a group) and all parts of a multipart definition start in the same column, and anything starting to the right of that continues the definition or part.

Curly braces and semicolons can be used to override.

9/234

Some primitive types:

Int, Real, Char, Bool.

Type variables are in lower case.

Type constructors: \rightarrow , $[]$, $(,)$.

Lambda-expressions: $\backslash x \rightarrow x * x$.

10/234

`::` means “has type”
and `:` means `cons`.

`String` is just a synonym for
`[Char]`.

Rest-of-line comments:

```
-- like this
```

Nested multi-line comments:

```
{- like this -}
```

11 / 234

Many functions are predefined in the
Prelude.

Many others are available in library
modules.

`import` is like OCaml’s `open`.

12 / 234

Computing permutations

13/234

```
perms1 :: [a] -> [[a]]
perms1 [] = [[]]
perms1 (x:xs) = addToAll x (perms1 xs)

addToAll x [] = []
addToAll x (p:ps) = addToOne x p ++ addToAll x ps

addToOne x [] = [[x]]
addToOne x (y:ys) =
    (x:y:ys) : consOnEach y (addToOne x ys)

consOnEach y [] = []
consOnEach y (p:ps) = (y:p) : consOnEach y ps
```

14/234

Running Haskell

15 / 234

The interpreter `ghci` resembles
`ocaml`.

The compiler `ghc` resembles `gcc`.

(It formerly used `gcc` for linking, and
as a back end.)

16 / 234

ghc expects main to be defined:

```
main :: IO()
```

```
main = print (perms [1,2,3,4])
```

We won't do this (or explain it) for a while.

17 / 234

Any two-parameter curried function
can be used as an operator:

```
5 'div' 2.
```

Any operator can be used as a
function: `(*) 3 4`.

One argument can be supplied:
`(3:), (: [7])`.

18 / 234

To avoid parentheses, the function application operator \$ (with lowest precedence) is used:

```
main = print $ perms [1,2,3,4]
```

What is the type of (\$) ?

19/234

```
perms1 = foldr addToAll [[]]

addToAll x = concat . map (addToOne x)

addToOne x [] = [[x]]
addToOne x (y:ys) =
  (x:y:ys) : map (y:) (addToOne x ys)
```

20/234

Haskell has many overloaded operators, and users can define their own.

21 / 234

```
sqr x = x*x  
:type sqr  
> sqr :: (Num a) => a -> a
```

:type is a command to `ghci`.
`Num` is a **type class** (details soon).

22 / 234

Haskell has if-then-else, with Boolean literals `True` and `False`, and logical connectives `&&`, `||`, and `not`.

23 / 234

Guards are a convenient alternative in definitions.

```
abs x | x >= 0    = x
      | otherwise = -x
```

24 / 234

Haskell has `let` expressions similar to OCaml:

```
let
    sqr1 = x*x
    sqr2 = y*y
in
    sqr1 + sqr2
```

25 / 234

A case expression resembles OCaml's `match`:

```
case x of
    [] -> False
    (x:xs) -> lookup x y
```

26 / 234

The use of `where` is more restricted, but it can scope across guards.

```
taxPayable s | amt < 100 = amt
              | otherwise = amt + 50
where
  amt = s * 0.07
```

27/234

List comprehensions:

```
myMap f xs = [f x | x <- xs]

myFilter p xs = [x | x <- xs, p x]

cross xs ys = [(x,y) | x <- xs, y <- ys]
```

28/234

```
perms2 [] = [[]]
perms2 xs
  = [y:p | (y,ys) <- sels xs,
        p <- perms2 ys]

sels [] = []
sels (x:xs)
  = (x,xs)
    : [(y,x:ys) | (y,ys) <- sels xs]
```

29/234

List comprehensions are actually syntactic sugar for a concept explored later in this lecture module.

30/234

Haskell allows type synonyms using `type`, and algebraic data types declared using `data`.

Unlike OCaml, data constructors are first-class functions and may be curried (typical).

31 / 234

```
data BTree a =  
    Empty  
    | Node a (Btree a) (Btree a)
```

32 / 234

Haskell's equivalent of OCaml's option type is Maybe.

```
data Maybe a
  = Nothing | Just a
```

This is more important in Haskell because of restrictions on exceptions.

33 / 234

Either extends the idea of Maybe.

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

34 / 234

Haskell's record syntax is useful for data types with many components.

```
data Student
  = S {name::String,
        uwID::Int,
        uwUserid::String}
```

35 / 234

A newtype declaration declares a distinct type with no run-time overhead, but it must have a single data constructor with a single field.

```
newtype Student'
  = Student {getStudent::Student}
```

This is useful for type abstraction.

36 / 234

So far, Haskell just looks like OCaml with more syntactic sugar.

The first key difference is purity.

37 / 234

We've seen that `print` can be used in `main` in a complete program.

It's not clear yet where else it can be used or what its type might be.

38 / 234

Purity means we can't expect to do printf-style debugging as easily as with Racket or OCaml.

`trace` (from `Debug.Trace`) comes close.

```
trace :: String -> a -> a
```

39/234

```
fact 0 = 1
fact n
  = trace (show n)
        (n * fact (n-1))
```

```
fact 5
> 5
...
1
120
```

40/234

The second key difference is laziness.

41 / 234

Laziness

42 / 234

Haskell uses an evaluation order similar to NOR (with no reduction inside abstractions).

This permits short-circuiting operators to be functions.

43 / 234

```
myAnd :: Bool -> Bool -> Bool
myAnd False _ = False
myAnd _ x = x
```

44 / 234

Typing `myAnd False undefined` into GHCi produces `False`, as expected.

In fact, `undefined` is defined in the Prelude as:

```
undefined
  = error "Prelude.undefined"
```

45 / 234

```
ones = 1 : ones
ones = [1,1..]
```

```
nats = 0 : map (+1) nats
nats = [0,1..]
```

```
odds = filter odd nats
odds = [1,3..]
```

46 / 234

```

fibs =
  0 : 1 :
    zipWith (+) fibs
              (tail fibs)

```

47/234

```

primes1 = sieve [2..]
sieve (p:ns)
  = p : sieve [n | n <- ns,
                  n `mod` p /= 0]

primes2 = 2 : oprimes
  where
    oprimes = 3 : filter isPrime [5,7..]
    possDivs n = takeWhile (\p-> p*p <= n)
                      oprimes
    notDiv n p = n `mod` p /= 0
    isPrime n = all (notDiv n) (possDivs n)

```

48/234

Laziness permits a more declarative style of programming.

49 / 234

`iterate` is in the Prelude.

```
iterate: (a -> a) -> a -> [a]
```

```
iterate f x
```

```
  = x : iterate f (f x)
```

```
nats = iterate (+1) 0
```

50 / 234

```
next n | odd n = 3*n+1
      | otherwise = n `div` 2

collatz n = takeWhile (/=1)
           $ iterate next n
```

51 / 234

Immutable arrays

52 / 234

Here's an example of creating a one-dimensional immutable array from a list of (index, value) pairs.

```
sqrs = array (1,100)
           [(i, i*i)
            | i <- [1..100]]
```

The function `assocs` reverses this process.

53 / 234

As with arrays in other languages, the chief advantage is constant-time access to any element, via the accessor operator `!`.

Just as laziness permitted us to put the name of a list on both the left-hand and right-hand sides of a definition, we can do so with arrays.

54 / 234

```

-- farray!n is nth Fibonacci number

farray = array (0,100)
           $ (0,0):(1,1)
           :[(n,fib n) | n<-[2..100]]
where
  fib n = farray!(n-1) + farray!(n-2)

```

55/234

```

-- colarray!n is #iters of collatz to reach 1

colarray = array (1,100)
           $ (1,0):[(n, coll n) | n<-[2..100]]
where
  coll n | odd n = 1 + check(3*n+1)
         | otherwise = 1 + check(n 'div' 2)
  check n | n > 100 = coll n
         | otherwise = colarray!n

```

56/234

Laziness lets us write very simple complete Haskell programs that interact with, say, the shell that runs them.

57 / 234

```
main :: IO ()  
interact :: (String -> String) -> IO ()
```

Intuitively, the first argument of `interact` is applied to the input, and the result is the output.

58 / 234

```
-- Unix 'cat'
main = interact id
```

```
-- Unix 'wc -l'
showln = (++ "\n") . show
main = interact $ showln . length . lines
```

59/234

```
linemap f = interact $ unlines . f . lines
```

```
-- Unix 'head -10'
main = linemap $ take 10
```

```
-- Unix 'grep a'
main = linemap $ filter $ elem 'a'
```

```
-- Unix 'grep help'
main = linemap $ filter
      $ Data.List.isInfixOf "help"
```

60/234

```

countChars s = [count c s | c <- ['a'..'z']]
  where
    count c s = (c, length [c' | c' <- s, c' == c])

main = interact (show . countChars)

-- alternate implementation

countChars s = assocs counts
  where
    counts = accumArray (+) 0 ('a','z')
              [(c,1) | c <- s, c >= 'a', c <= 'z']

main = interact (show . countChars)

```

61 / 234

Determining the time and space complexity of lazy code can be challenging.

Consider two familiar sorting algorithms.

62 / 234

```

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

```

Suppose xs has length n .
What is the time complexity of
`take k (qsort xs)`?

63/234

```

merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys)
  | x <= y    = x : merge xs (y:ys)
  | otherwise = y : merge (x:xs) ys

```

```

mpairs [] = []
mpairs [xss] = [xss]
mpairs (xs:ys:xss)
  = merge xs ys : mpairs xss

```

64/234


```
mergeall [] = []  
mergeall [xs] = xs  
mergeall xss = mergeall (mpairs xss)  
  
msort = mergeall . (map (:[]))
```

What is the time complexity of
`take k (msort xs)` ?

65/234

```
foldr c b [] = b  
foldr c b (x:xs) = c x (foldr c b xs)
```

What is the space complexity of
`foldr (+) 0 xs` ?

66/234

```
foldl c b [] = b
foldl c b (x:xs) = foldl c (c b x) xs
```

What is the space complexity of
`foldl (+) 0 xs` ?

67/234

```
foldl' c b [] = b
foldl' c !b (x:xs) = foldl' c (c b x) xs
```

What is the space complexity of `foldl' (+) 0 xs` ?

68/234

Type classes

69 / 234

Type classes offer a controlled approach to overloading.

There are a number of predefined classes: `Eq`, `Ord`, `Show`, `Read`, `Num`, `Ix`, and more.

70 / 234

You can create instances of these classes.

You can also create your own classes and instantiate them.

(These are **not** OO classes.)

71 / 234

Types in the Eq class provide == and /=.

```
member _ [] = False
member y (x:xs)
  = (x==y) || (member y xs)
```

This will be typed as:

```
member :: Eq a => a -> [a] -> Bool
```

72 / 234

All the base types (`Int`, `Bool`, etc.) are members of `Eq`, as are lists and tuples of members.

The simple way to create a member of `Eq` is to append deriving `Eq` to a datatype.

73/234

```
data Btree a =  
    Empty  
  | Node (Btree a) a (Btree a)  
    deriving Eq
```

```
(Node Empty 4 Empty)  
  /= (Node Empty 5 Empty)  
> True
```

74/234

We may wish to provide a non-derived equality method.

```
data First = Pair Int Int
```

```
instance Eq First where  
    (Pair x _) == (Pair y _) = (x==y)
```

```
(Pair 1 3) == (Pair 2 3)
```

```
> False
```

```
(Pair 1 3) == (Pair 1 4)
```

```
> True
```

75/234

Here is the actual definition of Eq:

```
class Eq a where  
    (==), (/=) :: a -> a -> Bool  
    x /= y = not (x == y)  
    x == y = not (x /= y)
```

Either of these default definitions may be overridden.

76/234

The Ord class

77 / 234

Ord inherits from Eq and specifies the four comparison operators `<`, `<=`, `>`, `>=`. It gives default definitions for `min` and `max` in terms of these.

There is also a three-way compare function.

78 / 234

Our sorting routines are typed in terms of `Ord`:

```
msort :: (Ord a) => [a] -> [a]
```

Most basic datatypes are instances of `Ord`, and user-defined datatypes can derive `Ord` (lexicographic ordering).

79 / 234

Other predefined typeclasses

80 / 234

Show specifies the method

`show :: a -> String.`

Read specifies the method

`read :: String -> a`, and can be used for simple parsing.

81 / 234

Num inherits from Eq, and specifies

`+`, `-`, `*`, `negate`, `abs`, and `signum`.

Division is handled by `Integral` and

`Fractional`, which inherit from `Num`.

82 / 234

Implementation of type classes

83 / 234

For simplicity, consider the class Num
to only specify +, *, and negate.

84 / 234

In program:

```
class Num a where
  (+), (*) :: a -> a -> a
  negate   :: a -> a
```

85/234

Translation:

```
data NumDict a
  = MkND (a -> a -> a)
          (a -> a -> a)
          (a -> a)
plus  (MkND p _ _) = p
times (MkND _ t _) = t
neg   (MkND _ _ n) = n
```

86/234

Now suppose we've defined a `Matrix` type, and want to overload our numeric operators to apply to it.

87 / 234

In program:

```
instance Num Matrix where
  (+)      = matAdd
  (*)      = matMult
  negate   = matNeg
```

88 / 234

Translation:

```
NDMatrix :: NumDict Matrix
NDMatrix = MkND matAdd
           matMult
           matNeg
```

89/234

In program:

```
square :: (Num a) => a -> a
square x = x * x
```

90/234

Translation:

```
square :: NumDict a -> a -> a  
square nd x = times nd x x
```

91 / 234

In program:

```
m :: Matrix  
m = identityMatrix 10  
msq = square m
```

92 / 234

Translation:

```
m :: Matrix  
m = identityMatrix 10  
msq = square NDMatrix m
```

93 / 234

For more details, see Philip Wadler and Stephen Blott, “How to make ad-hoc polymorphism less ad-hoc”.

94 / 234

More useful type classes

95 / 234

The Monoid class

96 / 234

In mathematics, a monoid is a set with an identity and an associative binary operation.

The integers with 0 and + are a monoid.

Lists with [] and ++ are a monoid.

97 / 234

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a

  mconcat :: [a] -> a
  mconcat = foldr mappend mempty
```

98 / 234

The Functor class

99 / 234

Using Maybe can get awkward.

Consider adding two values of
Maybe type.

100 / 234

```
case x of
  Nothing -> Nothing
  Just v1 ->
    case y of
      Nothing -> Nothing
      Just v2 -> Just (v1+v2)
```

101 / 234

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Here f is a type constructor with exactly one argument.

102 / 234

Kinds are a useful concept at this point.

The kind of a concrete type (e.g. `Bool`) is `*`.

The kind of `Maybe` and `[]` is `* \Rightarrow *`.

103 / 234

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

`f` must have kind `* \Rightarrow *`.

104 / 234

The type constructor $(,)$ has kind $* \Rightarrow * \Rightarrow *$, so it can't be an instance of `Functor`.

The same is true of the type constructor $(->)$.

105 / 234

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

How are `Maybe` and `[]` instances?

106 / 234

```
instance Functor Maybe where
  fmap _ Nothing  = Nothing
  fmap g (Just a) = Just (g a)
```

107/234

```
fmap (3*) (Just 2)
> Just 6
fmap (3*) Nothing
> Nothing
```

108/234

```
instance Functor [] where  
    fmap = map
```

109/234

```
fmap (3*) [1,2,3]  
> [3,6,9]
```

110/234

`(->)` and `(,)` have the wrong kind to be an instance of `Functor`.

But `((->) r)` and `((,) w)` have the right kind.

111 / 234

```
instance Functor ((->) r) where
    fmap = (.)
```

This is mildly interesting, but gets more interesting later.

112 / 234

More useful is:

```
instance Functor ((,) w) where  
    fmap f (w,a) = (w, f a)
```

```
fmap (*3) ("result", 4)  
> ("result", 12)
```

113/234

What if we want the value of type w to be on the other side of the pair?

We can use a `newtype` declaration to create the right kind of type constructor.

114/234

```
newtype With w a
  = W {getPair::(a,w)}
```

```
instance Functor (With w) where
  fmap f (W (a,w)) = W (f a, w)
```

115/234

The Functor laws:

```
fmap id = id
fmap (g . h) = fmap g . fmap h
```

116/234

```
fmap (+) (Just 2) (Just 3)
```

This does not work. Why not?

117/234

The Applicative class

118/234

Defined in Control.Applicative:

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Recall (\$) :: (a -> b) -> a -> b.

119/234

Maybe, [], ((->) r), and ((,) w)
are all instances of Applicative.

120/234

```
instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  _ <*> Nothing = Nothing
  (Just f) <*> (Just y) = Just (f y)
```

121 / 234

```
fmap (+) (Just 2) <*> (Just 3)
> Just 5
(+) <$> (Just 2) <*> (Just 3)
> Just 5
pure (+) <*> (Just 2) <*> (Just 3)
> Just 5
```

122 / 234

There are five `Applicative` laws, of which the most important is:

$$\text{fmap } g \ x = \text{pure } g \ \langle * \rangle \ x$$

123 / 234

The laws can be used to prove that there is a canonical form for any `Applicative` expression:

$$\text{pure } f \ \langle * \rangle \ x_1 \ \langle * \rangle \ x_2 \ \dots \ \langle * \rangle \ x_n$$
$$f \ \langle \$ \rangle \ x_1 \ \langle * \rangle \ x_2 \ \dots \ \langle * \rangle \ x_n$$

124 / 234

```
instance Applicative [] where
  pure = \x -> [x]
  fs <*> xs = [f x | f <- fs, x <- xs]
```

125/234

```
instance Applicative [] where
  pure = \x -> [x]
  fs <*> xs =
    concat (map (\f->map f xs) fs)
```

126/234

```
instance Applicative ((->) r) where
  pure = const
  f <*> x = \r -> f r (x r)
```

Does this look familiar?

127 / 234

This can be used to provide a value as an extra parameter throughout a computation.

```
pure (*) <*> (+1) <*> (*4) $ 2
> 24
```

We can peek at the parameter value using `id`.

128 / 234


```

interp' :: Expr -> Env -> Value
...
interp' (BinOp OpAdd x y)
  = pure (*)
    <*> (interp x)
    <*> (interp y)

interp :: Expr -> Value
interp t = interp' t emptyEnv

```

129/234

How do we make $((,) w)$ an instance of Applicative?

```

pure  :: a -> (w, a)
(<*>) :: (w, a -> b)
      -> (w, a)
      -> (w, b)

```

130/234

```
instance ((,) w) of
  Monoid w => Applicative w where
    pure x = (mempty, x)
    (w, f) <*> (w', x) =
      (mappend w w', f x)

("3 times ", (*3)) <*> ("2", 2)
> ("3 times 2", 6)
```

131 / 234

The Monad class

132 / 234

```
class Applicative m => Monad m where
  (=<<) :: (a -> m b) -> m a -> m b
```

(Valid, but not real definition.)

133/234

```
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
```

(Still valid and not real definition.)

134/234

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> mb) -> mb
  (>>) :: m a -> m b -> m b
  fail :: String -> m a

  m >> n = m >>= \_ -> n
  fail s = error s
```

(Real definition.)

135/234

```
instance Monad Maybe where
  (Just x) >>= f = f x
  Nothing >>= _ = Nothing
  return = Just
  fail _ = Nothing
```

136/234

```
lookup :: a -> [(a, b)] -> Maybe b
```

```
ids = [("Mary", 20123456), ...]
```

```
marks = [(20123456, 99), ...]
```

```
lookup "Mary" ids >=> \id ->
```

```
  lookup id marks
```

```
> Just 99
```

137/234

```
do id <- lookup "Mary" ids
```

```
  lookup id marks
```

138/234

do expr desugars to expr.

```
do expr  
  block
```

desugars to

```
expr >>  
do block
```

139/234

```
do pat <- expr  
  block
```

desugars to

```
expr >>= \pat ->  
do block
```

140/234

Control.Monad defines a number of general functions, which can be accessed with:

```
import Control.Monad
```

141 / 234

```
liftM2 :: (Monad m) =>
  (a -> b -> c) -> m a -> m b -> m c

liftM2 f mv1 mv2 = do x <- mv1
                      y <- mv2
                      return (f x y)
```

142 / 234

```
liftM2 (+) (Just 2) (Just 3)  
> Just 5
```

```
liftM2 (+) (Just 2) Nothing  
> Nothing
```

There are “lifted” versions `mapM`, `filterM`, `ap`,
and so on.

143 / 234

The Reader monad

144 / 234


```
instance Monad ((->) r) where
  return a = \_ -> a
  m >>= f = \r -> f (m r) r
```

145/234

```
pure (*) <*> (+1) <*> (*4) $ 2
> 24
```

```
(do x <- (+1)
    y <- (*4)
    return (x*y)) $ 2
> 24
```

```
liftM2 (*) (+1) (*4) $ 2
> 24
```

146/234

```
interp' :: Expr -> Env -> Value
```

```
...
```

```
interp' (BinOp OpAdd x y)
```

```
  = do xv <- interp' x
```

```
      yv <- interp' y
```

```
      return (xv+yv)
```

```
interp :: Expr -> Value
```

```
interp t = interp' t emptyEnv
```

147/234

```
newtype Reader r a =
```

```
  R {runReader :: r -> a}
```

```
instance Monad (Reader r) where
```

```
  return a = R $ \_ -> a
```

```
  m >>= f =
```

```
    R $
```

```
      \r -> runReader (f (runReader m r)) r
```

148/234

```
interp' :: Expr -> Reader Env Value
...
interp' (BinOp OpAdd x y)
  = do xv <- interp' x
      yv <- interp' y
      return (xv+yv)

interp t =
  runReader (interp' t) $ emptyEnv
```

149/234

To complete the interpreter, we need to look at the environment (variable case) and to run with a modified environment (application case).

150/234

```
ask :: Reader r r
ask = R id

local :: (r -> r) -> Reader r a -> Reader r a
local f m = R $ \r -> runReader m (f r)
```

151 / 234

The Writer monad

152 / 234

The Reader monad was a newtype wrapper around the $((\rightarrow) r)$ instance of `Applicative`.

We can do the same for the $((,) w)$ instance of `Applicative`, giving us the Writer monad.

153 / 234

The Writer monad is useful for accumulating information in a monadic fashion during a computation.

For example, logging.

154 / 234

Naturally, we might want to combine the ideas of the Reader and Writer monad.

155 / 234

The State monad

156 / 234

Monads hide plumbing.

The plumbing hidden by the Maybe monad is the wrapping/unwrapping of the value.

What plumbing is involved in manipulating state?

157 / 234

If the computation is tail-recursive, we can put state in an extra parameter (an accumulator).

It's harder with a more general computation (e.g. on trees).

158 / 234

We can pass the state as an extra parameter to a function.

But if the function affects the state, that has to be returned along with the value the function computed.

Example: renumbering a binary tree in prefix order.

159/234

```
data BTree a = Empty
  | Node (BTree a) a (BTree a)
    deriving Show

numTree :: BTree a -> Int
        -> (BTree Int, Int)

numTree Empty s = (Empty, s)
numTree (Node l v r) s =
  let (l',s') = numTree l (s+1)
      (r',s'') = numTree r s'
  in (Node l' s r', s'')

run t = numTree t 0
```

160/234

To hide the state plumbing,
expressions such as `numTree r s'`
must become `numTree r`.

In other words, a monadic value
must be a function consuming a
state and producing a tuple
(value, new state).

161 / 234

```
newtype State s a
  = State {runState :: s -> (a, s)}

instance Monad (State s) where
  -- return :: a -> State s a

      return x = State $ \s -> (x, s)

  -- >>= :: State s a
  --      -> (a -> State s b)
  --      -> State s b
```

162 / 234

```

-- newtype State s a
-- = State {runState :: s -> (a, s)}
-- >>= :: State s a
--      -> (a -> State s b)
--      -> State s b

(State h) >>= f = State $ \s ->
  let (a, ns)    = h s
      (State g) = f a
  in g ns

```

163/234

These helper functions are useful.

```

evalState m s = fst $ runState m s

get = State $ \s -> (s, s)

put s = State $ \_ -> ((), s)

```

164/234

```
numTree2 Empty = return Empty
numTree2 (Node l v r) =
    do num <- get
       put (num+1)
       l' <- numTree2 l
       r' <- numTree2 r
       return (Node l' num r')

run2 t = evalState (numTree2 t) 0
```

165/234

The code on the last slide (plus the Btree definition) runs when Control.Monad.State.Lazy is imported.

The comments at the bottom of the source code include a more complex example: renumbering a tree where nodes with the same label get the same number.

166/234

Exercise:

Desugar the `do` notation to see that the monadic code is essentially doing the same thing as the code we wrote from scratch.

167 / 234

The IO monad

168 / 234

We saw earlier that `main` had type `IO ()`.

Conceptually, the `IO` monad is a state monad, where the state is the state of the “world”.

169 / 234

Unlike with the monads we saw earlier, we cannot pattern-match an `IO` value, and the only “run” function is `main`.

This encapsulates the impure parts of an interactive program in a way that keeps the bulk of one’s code pure.

170 / 234

```
main = mainloop

mainloop =
  do putStrLn "Ready!"
    typed <- getLine
    if (typed == "Done")
      then putStrLn "Bye!"
      else do putStrLn (map toUpper typed)
            mainloop
```

What is the type of `putStrLn`?

What is the type of `getLine`?

171 / 234

The IO monad has become a way of handling computational situations that are awkward in a pure language.

For example, exceptions can be thrown anywhere, but can only be caught within the IO monad.

172 / 234

The monad laws

173 / 234

The term “monad” comes from category theory.

Monads are supposed to satisfy three laws.

174 / 234

The three monad laws are:

$$\text{return } x \gg= f = f \ x$$
$$m \gg= \text{return} = m$$
$$\begin{aligned} m \gg= (\lambda x \rightarrow f \ x \gg= g) \\ = (m \gg= f) \gg= g \end{aligned}$$

These laws are not enforced by GHC.

175/234

The first two laws in do notation:

$$\text{do } \{y \leftarrow \text{return } x; f \ y\} = f \ x$$
$$\text{do } \{y \leftarrow m; \text{return } y\} = m$$

176/234

The third law repeated, and in `do` notation:

$$\begin{aligned} m >>= (\backslash x \rightarrow f\ x >>= g) \\ &= (m >>= f) >>= g \end{aligned}$$
$$\begin{aligned} &\text{do } y \leftarrow \text{do } x \leftarrow m \\ &\quad f\ x \\ &\quad g\ y \\ &= \\ &\text{do } x \leftarrow m \\ &\quad \text{do } y \leftarrow f\ x \\ &\quad g\ y \end{aligned}$$

177/234

Both of these are equal to:

$$\begin{aligned} &\text{do } x \leftarrow m \\ &\quad y \leftarrow f\ x \\ &\quad g\ y \end{aligned}$$

178/234

The laws become a little clearer if we phrase them in terms of \leq , which is monadic function composition.

179/234

$$\begin{aligned} \leq & :: (b \rightarrow m\ c) \\ & \rightarrow (a \rightarrow m\ b) \\ & \rightarrow (a \rightarrow m\ c) \end{aligned}$$
$$f \leq g\ x = \text{do } y \leftarrow g\ x \\ f\ y$$
$$f \leq g = \backslash x \rightarrow (g\ x \gg= f)$$

180/234

The three monad laws:

$$\text{return} \leq\leq f = f$$
$$f \leq\leq \text{return} = f$$
$$f \leq\leq (g \leq\leq h) = (f \leq\leq g) \leq\leq h$$

181 / 234

It is convenient to add monoidal features:
an absorbing “zero” or “fail” monadic value,
and a “combining” or “choice” operation.

182 / 234

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a

instance MonadPlus Maybe where
  mzero = Nothing
  Nothing 'mplus' ys = ys
  xs      'mplus' _  = xs
```

183/234

Instances of MonadPlus should satisfy these laws:

```
mzero >>= f  = mzero
v >> mzero   = mzero
```

184/234

The List monad

185/234

```
instance Monad [] where
  return x = [x]
  xs >>= f
    = concat (map f xs)
```

```
instance MonadPlus [] where
  mzero = []
  mplus = (++)
```

186/234

```
do x <- [1,2]
   y <- [3,4]
   return (x,y)
```

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

187/234

```
[1,2] >>= \x =>
[3,4] >>= \y =>
return (x,y)
```

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

188/234

```
concat (map (\x ->
  concat (map (\y -> [(x,y)])
    [3,4]))
  [1,2])
```

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

189/234

```
do x <- [1,2]
  y <- [3,4]
  return (x,y)
=
[(x,y) | x <- [1,2], y <- [3,4]]
```

List comprehensions can be viewed as further sugaring of the List monad.

190/234

```
guard :: MonadPlus m => Bool -> m ()  
guard True = return ()  
guard False = mzero
```

```
pyth = do  
  z <- [1..]  
  x <- [1..z]  
  y <- [x..z]  
  guard (x2+y2 == z2)  
  guard (gcd x (gcd y z) == 1)  
  return (x,y,z)
```

191 / 234

Monad transformers

192 / 234

`Control.Monad.Trans.RWS.Lazy` gives us the RWS monad, which combines features of the Reader, Writer, and State monads.

How many of these do we have to create?

193 / 234

We could try composing monads.

```
newtype Compose m1 m2 a
  = C (m1 (m2 a))
```

Can't write `>>=` for `Compose m1 m2`.

(Works for Applicative.)

194 / 234

A monad transformer lets us create a new monad by adding features of one monad to another monad.

We will illustrate by writing `StateT`, the State monad transformer.

`StateT s Maybe` is a monad that adds failure to a stateful computation.

195 / 234

```
-- newtype State s a
-- = State {runState :: s -> (a, s)}

newtype StateT s m a
  = StateT {runStateT :: s -> m (a, s)}

-- contrast with m (State s a)
-- contrast with State s (m a)
-- what is the kind of StateT?
```

196 / 234

```

-- newtype State s a
-- = State {runState :: s -> (a, s)}

newtype StateT s m a
  = StateT {runStateT :: s -> m (a, s)}

instance Monad m => Monad (StateT s m)
  where
    -- return :: a -> StateT s m a
    return x = ?

```

197/234

```

-- newtype State s a
-- = State {runState :: s -> (a, s)}

newtype StateT s m a
  = StateT {runStateT :: s -> m (a, s)}

instance Monad m => Monad (StateT s m)
  where
    -- return :: a -> StateT s m a

return x = StateT $ \s -> return (x, s)

```

198/234

```

-- newtype State s a
-- = State {runState :: s -> (a, s)}

newtype StateT s m a
  = StateT {runStateT :: s -> m (a, s)}

instance Monad m => Monad (StateT s m)
where
  -- (>>=) :: StateT s m a
  --          -> (a -> StateT s m b)
  --          -> StateT m b

```

199/234

```

-- newtype State s a
-- = State {runState :: s -> (a, s)}

newtype StateT s m a
  = StateT { runStateT :: s -> m (a, s) }

instance Monad m => Monad (StateT s m)
where
  -- (>>=) :: StateT s m a
  --          -> (a -> StateT s m b)
  --          -> StateT m b

x >>= f =
  StateT $ \s ->
    do (a, s') <- runStateT x s
       runStateT (f a) s'

```

200/234

We can define `get` and `put` for `StateT s m`.

```
get :: (Monad m) => StateT s m s
get = state $ \s -> (s, s)
```

```
put :: (Monad m) => s -> StateT s m ()
put s = state $ \_ -> ((), s)
```

201 / 234

In fact, `State s` is defined as `StateT s Identity`.

In the `Identity` monad, `return` does nothing, and `bind` is just function application.

202 / 234

```
newtype Identity a
  = Identity {runIdentity :: a}

instance Monad Identity where
  return = Identity
  m >>= f = f (runIdentity m)
```

203 / 234

There are many standard monad classes, some of which we've seen (Maybe, Reader, Writer, State).

Each is defined in terms of a monad transformer and the Identity monad.

204 / 234

Each monad transformer provides a `lift` operation to lift values of the underlying monad to the created monad.

205 / 234

```
class MonadTrans t where
  lift :: Monad m => m a -> t m a

instance MonadTrans (StateT s) where
  lift m =
    StateT $ \s ->
      do a <- m
      return (a,s)
```

206 / 234

Each standard monad transformer is made an instance of a class that provides lifted versions of the functions of another (such as `put` for `StateT`), thus avoiding some explicit lifting with stacked transformers.

207 / 234

We can also lift extended features (e.g. monoidal ones) of the underlying monad to the created monad.

208 / 234


```
instance MonadPlus m =>
  MonadPlus (StateT s m) where
    mzero = lift mzero
    mplus m1 m2 =
      StateT $ \s ->
        runStateT m1 s
        'mplus'
        runStateT m2 s
```

(Using same state facilitates backtracking.)

209 / 234

Using all this machinery we can do some complex things in a pretty simple fashion.

One example is simple creation of flexible parsers.

210 / 234

What should the type
of a parser be?

211 / 234

`String -> Bool`

212 / 234

$[t] \rightarrow \text{Bool}$

213/234

$[t] \rightarrow ([t], a)$

214/234

$[t] \rightarrow \text{Maybe } ([t], a)$

215/234

$[t] \rightarrow [([t], a)]$

216/234

(Idea from Philip Wadler, “How To Replace Failure By A List Of Successes”)

217 / 234

$$[t] \rightarrow [([t], a)]$$

This is the List monad (for results) combined with the State monad (for unconsumed input).

218 / 234

```
newtype Parser t a
  = Parser (StateT [t] [] a)
  deriving
    (Monad,
     MonadState [t],
     MonadPlus)
```

219/234

```
token :: Parser t t
token =
  do inp <- get
  case inp of
    [] -> mzero
    (t:ts) -> do put ts
                  return t
```

220/234

```
test :: (t -> Bool) -> Parser t t
test p =
    do t <- token
       guard (p t)
       return t
```

```
exactly :: t -> Parser t t
exactly t = test (==t)
```

221 / 234

```
-- zero or more
many :: Parser t a -> Parser t a
many p = mplus $ some p
        $ return []
```

```
-- one or more
some :: Parser t a -> Parser t a
some p = liftM2 (:) p (many p)
```

222 / 234

```
number :: Parser Char Integer
number =
    do ds <- some (test isDigit)
    return (read ds)
```

223 / 234

Recall the standard grammar for arithmetic expressions with operator precedence:

$$\begin{aligned} E &= T + E \mid T \\ T &= F * T \mid F \\ F &= n \mid (E) \end{aligned}$$

We could build a parse tree, or evaluate.

224 / 234


```
expr =  
  do a <- term  
    exactly '+'  
    b <- expr  
    return (a+b)  
'mplus'  
term
```

225 / 234

```
term =  
  do a <- factor  
    exactly '*'  
    b <- term  
    return (a*b)  
'mplus'  
factor
```

226 / 234

```
factor =  
  number  
  'mplus'  
do exactly '('  
  a <- expr  
  exactly ')'  
  return a
```

227 / 234

```
> runParser expr "2+4*3"  
[(14, ""), (6, "*3"), (2, "+4*3")]
```

What if we only want a complete parse?

```
(Just x) 'mplus' _ = Just x
```

228 / 234

```
newtype Parser t a
  = Parser (StateT [t] Maybe a)
  deriving
    (Monad,
     MonadState [t],
     MonadPlus)
```

```
> runParser expr "1+2*3"
[Just (14,"")]
```

229 / 234

Briefly:
some other
features

230 / 234

HUnit: unit testing (modelled on JUnit)

Specify assertions, combine them using monadic notation into test cases, combine them into named tests, group them into suites, run them.

231 / 234

QuickCheck: randomized testing of properties of code

- ▶ combinators for constructing properties
- ▶ altering random distributions for built-in data types (including infinite lists and functions)
- ▶ specifying distributions for user-defined data types

232 / 234

- Generalized algebraic data types
- Multi-parameter type classes
- Type families
- Concurrency and parallelism
- Foreign function interface

233 / 234

Next: some of the type theory behind all this.

234 / 234