Haskell

Haskell features (from module 01):

- Purity
- Laziness
- Type classes
- Monads

Some historical milestones:

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

Many other lazy functional languages

First committee meeting 1987 (last one 1999)

Haskell 1.0 Report 1 April 1990

Glasgow Haskell Compiler (GHC) 1992

Early 1999: Haskell 98 report

Haskell initially resembles OCaml with fewer keywords.

It avoids some by making whitespace significant.

Value definitions:
$\langle id \rangle$ = $\langle expr \rangle$

```
x = 3
y = x*x + 3*x + 4
```


Function definitions:
$\langle id \rangle$ $\langle param \rangle$ $\langle param \rangle$ ... = $\langle expr \rangle$

```
sqr x y = x*x + y*y
```


Multipart function definitions:
$\langle id \rangle$ $\langle pattern \rangle$ $\langle pattern \rangle$ ... = $\langle expr \rangle$

```
pred 0 = 0
pred n = n-1
```

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.

Some primitive types:
`Int`, `Real`, `Char`, `Bool`.

Type variables are in lower case.

Type constructors: `->`, `[]`, `(,)`.

Lambda-expressions: `\x -> x*x`.

`::` 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 -}`

Many functions are predefined in the Prelude.

Many others are available in library modules.

`import` is like OCaml's `open`.

Computing permutations


```
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
```


Running Haskell

The interpreter `ghci` resembles `ocaml`.

The compiler `ghc` resembles `gcc`.

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

`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.

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])`.

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

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

What is the type of `($)`?


```
perms1 = foldr addToAll [[]]

addToAll x = concat . map (addToOne x)

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

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


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

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

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

Guards are a convenient alternative in definitions.

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

Haskell has `let` expressions similar to OCaml:

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


A `case` expression resembles OCaml's `match`:

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


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
```


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]
```

```
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]
```


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

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).


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

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.

`Either` extends the idea of `Maybe`.

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


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

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


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.

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

The first key difference is purity.

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.

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
```



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

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

The second key difference is laziness.

Laziness

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

This permits short-circuiting operators to be functions.


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

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

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

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



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

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

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



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



```
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)
```

Laziness permits a more declarative style of programming.

`iterate` is in the Prelude.

```
iterate: (a -> a) -> a -> [a]
iterate f x
 = x : iterate f (f x)

nats = iterate (+1) 0
```



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

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


Immutable arrays

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.

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.


```
-- 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)
```



```
-- 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
```

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


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

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


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

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



```
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"
```

```haskell
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)
```


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

Consider two familiar sorting algorithms.


```haskell
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)`?


```haskell
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
```

```
mergeall [] = []
mergeall [xs] = xs
mergeall xss = mergeall (mpairs xss)

msort = mergeall . (map (:[]))
```

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


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

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


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

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


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

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

Type classes

Type classes offer a controlled approach to overloading.

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

You can create instances of these classes.

You can also create your own classes and instantiate them.

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

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
```

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.


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

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


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
```


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.

The `Ord` class

`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.

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).

Other predefined typeclasses

`Show` specifies the method `show :: a -> String`.

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

`Num` inherits from `Eq`, and specifies `+`, `-`, `*`, `negate`, `abs`, and `signum`.

Division is handled by `Integral` and `Fractional`, which inherit from `Num`.

Implementation of type classes

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

In program:

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


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
```


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

In program:

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

Translation:

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


In program:

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


Translation:

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


In program:

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

Translation:

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


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

More useful type classes

The Monoid class

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.


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

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


The Functor class

Using `Maybe` can get awkward.

Consider adding two values of `Maybe` type.

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



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

Here `f` is a type constructor with exactly one argument.

Kinds are a useful concept at this point.

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

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


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

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

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

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


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

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


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



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

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



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


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

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


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

This is mildly interesting, but gets more interesting later.

More useful is:

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

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


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.


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

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


The `Functor` laws:

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

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

This does not work. Why not?

The Applicative class

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`.

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

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



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


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

```
fmap g x = pure g <*> x
```


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

```
pure f <*> x1 <*> x2 ... <*> xn
```

```
f <$> x1 <*> x2 ... <*> xn
```

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



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



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

Does this look familiar?

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`.

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

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


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

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



```
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)
```


The Monad class

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

(Valid, but not real definition.)


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

(Still valid and not real definition.)


```
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.)


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

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

ids = [("Mary", 20123456), ...]
marks = [(20123456, 99), ...]

lookup "Mary" ids >>= \id ->
 lookup id marks
> Just 99
```



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


`do expr` desugars to `expr`.

```
do expr
   block
```

desugars to

```
expr >>
do block
```



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

desugars to

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

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

```
import Control.Monad
```



```
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)
```



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

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

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

The Reader monad

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



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

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

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



```haskell
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
```



```haskell
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
```

```
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
```


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


```
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)
```


The Writer monad

The Reader monad was a `newtype` wrapper around the `((->) r)` instance of `Applicative`.

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

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

For example, logging.

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

The State monad

Monads hide plumbing.

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

What plumbing is involved in manipulating state?

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).

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.


```
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
```

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).


```
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
```



```
-- 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
```


These helper functions are useful.

```
evalState m s = fst $ runState m s

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

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

```
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
```


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.

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

The `IO` monad

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”.

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.


```
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`?

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.

The monad laws

The term “monad” comes from category theory.

Monads are supposed to satisfy three laws.

The three monad laws are:

```
return x >>= f  = f x

m >>= return    = m

m >>= (\x -> f x >>= g)
  = (m >>= f) >>= g
```

These laws are not enforced by GHC.

The first two laws in `do` notation:

```
do {y<-return x; f y} = f x

do {y<-m; return y}   = m
```

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

```
m >>= (\x -> f x >>= g)
  = (m >>= f) >>= g

do y <- do x <- m
           f x
   g y
=
do x <- m
   do y <- f x
      g y
```


Both of these are equal to:

```
do x <- m
   y <- f x
   g y
```


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


```
<=< :: (b -> m c)
       -> (a -> m b)
       -> (a -> m c)

f <=< g x = do y <- g x
               f y

f <=< g = \x -> (g x >>= f)
```

The three monad laws:

```
return <=< f  = f

f <=< return  = f

f <=< (g <=< h) = (f <=< g) <=< h
```


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


```
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
```


Instances of `MonadPlus` should satisfy these laws:

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

The `List` monad


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

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



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

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



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

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

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

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



```
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.


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

pyth = do
  z <- [1..]
  x <- [1..z]
  y <- [x..z]
  guard (x^2+y^2 == z^2)
  guard (gcd x (gcd y z) == 1)
  return (x,y,z)
```


Monad transformers

`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?

We could try composing monads.

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

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

(Works for Applicative.)

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.


```
-- 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?
```

```haskell
-- 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 = ?
```



```haskell
-- 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)
```



```haskell
-- 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
```



```haskell
-- 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'
```

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)
```


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

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


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

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


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.

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


```
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)
```


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.

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

```
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.)

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

One example is simple creation of flexible parsers.

What should the type of a parser be?


```
String -> Bool
```

```
[t] -> Bool
```



```
[t] -> ([t],a)
```



```
[t] -> Maybe ([t],a)
```



```
[t] -> [([t],a)]
```

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


```
[t] -> [([t],a)]
```

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


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



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

```
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)
```



```
-- 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)
```



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


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.

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



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



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



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

What if we only want a complete parse?

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

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

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


Briefly: some other features

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.

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

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

Next: some of the type theory behind all this.