# Lecture 8: Haskell

CSC 131
Fall, 2014

Kim Bruce

# Common Type Classes

- Eq, Ord, Enum, Bounded, Show, Read
  - See http://www.haskell.org/tutorial/stdclasses.html
- data defs pick up default if add to class:
  - data ... deriving (Show, Eq)
- Can redefine:
  - instance Show TrafficLight where
    show Red = "Red light"
    show Yellow = "Yellow light"
    show Green = "Green light"

# More Type Classes

- class (Eq a) => Num a where ...
  - instance of Num a must be Eq a
- Polymorphic function types can be prefixed w/ type classes
  - test x y = x < y *has type* (Ord a) => a -> a -> Bool
  - *Can be used w/ x, y of any Ord type.*
- *More later ...*
  - *Error messages often refer to actual parameter needing to be instance of a class -- to have an operation.*

# Higher-Order Functions

- Functions that take function as parameter
  - Ex: map:: (a → b) → ([a] → [b])
- Build new control structures
  - listify oper identity [] = identity
    listify oper identity (fst:rest) =
    oper fst (listify oper identity rest)
  - sum' = listify (+) 0
    mult' = listify (*) 1
    and' = listify (&&) True
    or' = listify (||) False

# Quicksort

```
partition (pivot, []) = ([],[])
partition (pivot, first : others) =
   let
      (smalls, bigs) = partition(pivot, others)
   in
      if first < pivot
         then (first:smalls, bigs)
         else (smalls, first:bigs)
```

Type is:

```
partition :: (Ord a) => (a, [a]) -> ([a], [a])
```

# Quicksort

```
qsort [] = []
qsort [singleton] = [singleton]
qsort (first:rest) =
   let
      (smalls, bigs) = partition(first,rest)
   in
      qsort(smalls) ++ [first] ++ qsort(bigs)
```

Type is:

```
qsort :: (Ord t) => [t] -> [t]
```

# Quicksort - parametrically

```
partition (pivot, []) lThan = ([],[])
partition (pivot, first : others) lThan =
  let
    (smalls, bigs) = partition(pivot, others) lThan
  in
    if (lThan first pivot)
      then (first:smalls, bigs)
      else (smalls, first:bigs)

partition ::
        (t, [a]) -> (a -> t -> Bool) -> ([a], [a])

*Main> partition(6,[8,4,6,3])(>)
```

# Quicksort

```
qsort [] lt = []
qsort [singleton] lt = [singleton]
qsort (first:rest) lt =
   let
      (smalls, bigs) = partition (first,rest) lt
   in
      qsort smalls lt ++ [first]
                      ++ qsort bigs lt

qsort :: [a] -> (a -> a -> Bool) -> [a]

*Main> qsort [33,66,32,87,999,2](>)
[999,87,66,33,32,2]
```

# Recursive Datatype Examples

- data IntTree = Leaf Integer |
  Interior (IntTree,IntTree)
  deriving Show
  - Example values: Leaf 3, Interior(Leaf 4,Leaf -5), ...
- data Tree a = Niltree |
  Maketree (a, Tree a, Tree a)

# Binary Search Using Trees

```
insert new Niltree = Maketree(new,Niltree,Niltree)
insert new (Maketree (root,l,r)) =
  if new < root
     then Maketree (root,(insert new l),r)
     else Maketree (root,l,(insert new r))

buildtree [] = Niltree
buildtree (fst : rest) =
                 insert fst (buildtree rest)
```

# Binary Search Tree

```
find elt Niltree = False
find elt (Maketree (root,left,right)) =
  if elt == root
     then True
     else if elt < root then find elt left
     else find elt right      -- elt > root

bsearch elt list = find elt (buildtree list)
```

# Haskell is Lazy!

# Lazy vs. Eager Evaluation

- Eager: Evaluate operand, substitute operand value in for formal parameter, and evaluate.
- Lazy: Substitute operand for formal parameter and evaluate body, evaluating operand only when needed.
  - Each actual parameter evaluated either not at all or only once! (Essentially cache answer once computed)
  - Like left-most outermost, but more efficient

# Lazy evaluation

- Compute f(1/0,17) where f(x,y) = y
- Computing head(qsort[5000,4999..1]) is faster than qsort[5000,4999..1]
- Compare time of computations of:
  - fib 32
  - dble (fib 32) where dble x = x + x
- Computations based on *graph reduction*
  - *like tree rewriting, except w/computation graphs - sharing*

# Lazy Lists

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

*complexity O(fib n) ~ O($2^n$)*

```
fibList = f 1 1
  where f a b = a : f b (a+b)
fastFib n = fibList!!n
```

*complexity O(n)*

```
fibs = 1:1:[ a+b | (a,b) <- zip fibs (tail fibs)]

primes = sieve [ 2.. ]
       where
         sieve (p:x) = p :
              sieve [ n | n <- x, n `mod` p > 0]
```

# Call-by-need

- Efficient implementation of call-by-name (Algol 60)
- If purely functional language then may evaluate expression at most once, because can never change.
- Hence graph instead of tree works!
  - dble(fib 32)

# Monads

The ontological essence of a **monad** is its ***irreducible simplicity***. Unlike atoms, monads possess no material or spatial character. They also differ from atoms by their complete mutual independence, so that interactions among monads are only apparent. Instead, by virtue of the principle of pre-established harmony, **each monad follows a preprogrammed set of "instructions" peculiar to itself, so that a monad "knows" what to do at each moment**.

*-wikipedia*

# Monads

In category theory, a branch of mathematics, a *monad*, or triple is an (endo-)functor, together with two natural transformations. Monads are used in the theory of pairs of adjoint functors, and they generalize closure operators on partially ordered sets to arbitrary categories.

*-wikipedia*

# Toward Monads

- *data* Maybe a = Nothing | Just a *deriving* (Eq, Show)
  - Useful for computations that may not have a result
  - Part of "standard prelude" imported by all Haskell modules
  - Look up a phone number for a person.
  - Maybe Integer includes Nothing, Just 7, ...
- Hard to work with type as values "wrapped"

# Maybe Monad

- dormRooms =[("Jack",10),("Jill",20),("Ann",20)]
- phonesForRooms = [(10,23434),(20,23435),(30,23438)]
- getDormFor name [] = Nothing
  *-- 2nd arg is name-room pairs*
  getDormFor name ((nm,rm):rest) = if nm == name
  then Just rm
  else getDormFor name rest
- getPhoneForRoom rm [] = Nothing
  getPhoneForRoom rm ((rmnum,phone):rest) =
  if rm == rmnum then Just phone
  else getPhoneForRoom rm rest

# Awkward to Compose

- getPhoneForName name rooms phones =
  case getDormFor name rooms of
  Nothing -> Nothing
  Just rm -> getPhoneForRoom rm phones

- Must unwrap values to use and then rewrap
- Easier if could write:
  - getPFN name rooms phones =
    do rm <- getDormFor name rooms
    num <- getPhoneForRoom rm phones
    return num
  - *and not have to worry about error cases!*

# Defining Monads

- class Monad m where ← *part of Standard Prelude*
  (>>=) :: m a → (a → m b) → m b
  return :: a → m a
  fail:: string → m a
  - >>= allows a kind of composition of wrapped values or computations -- called *bind*
  - return wraps an unwrapped value.
  - fail takes error string & aborts program
  - a >> b *abbreviates* a >> _ -> b *(constant fcn)*

# Maybe Monad

- instance Monad Maybe where ← *part of Standard Prelude*
  (>>=) Nothing f = Nothing
  (>>=) (Just x) f = f x
  return x = Just x
  fail s = Nothing
- >>= preserves "Nothing",
- >>= unwraps argument to compute w/ a Just'ed value
- Second arg of >>= is function applied to unwrapped value
- *Abbreviate* compu >>= \x → exp *as*
  do x <- compu
  exp

# Back to Example

- Expression
  - getPFN name rooms phones =
    do rm <- getDormFor name rooms
    num <- getPhoneForRoom rm phones
    return num
  - *abbreviates*
  - getPFN name rooms phones =
    getDormFor name rooms >>=
    (\rm -> getPhoneForRoom rm phones)

# Monads

- Provide operations to compose wrapped values
- Operations obey laws:
  - return x >>= f == f x *left identity*
  - c >>= return == c *right identity*
  - c >>= (\x -> f x >>= g) == (c >>= f) >>= g *associativity*

# In "do" notation

- Left identity:

```
do { x' <- return x;
     f x'
   }
```

≡

```
do { f x }
```

- Right identity:

```
do { x <- m;
     return x
   }
```

≡

```
do { m }
```

- Associativity:

```
do { y <- do { x <- m;
               f x
             }
     g y
}
```

≡

```
do { x <- m;
     do { y <- f x;
          g y
        }
}
```

# Application of Laws

- Program:

```
skip_and_get = do
        unused <- getLine
        line <- getLine
        return line
```

- is equivalent to:

```
skip_and_get = do
        unused <- getLine
        getLine
```

*by right identity*

*See http://www.haskell.org/haskellwiki/Monad_laws for more info*

# Other Monad Examples

- Error handling M(a) = a ∪ {error}
  - Add a special "error value" to a type
  - Define bind operator ">>=" to propagate error
- Information-flow tracking M(a) = a × Labels
  - Add information flow label to each value
  - Define bind to check and propagate labels
- State M(a) = a × States
  - Computation produces value and new state
  - Define bind to make output state of first go to input state of second