CS 320: Concepts of Programming Languages

Wayne Snyder
Computer Science Department
Boston University

Lecture 11: More Monads!

- Review: Ok Monad, an improved Maybe Monad
- The List Monad (a.k.a. "Map and Flatten")
- List Comprehensions

Next time (after break): The State monad

The Ok Monad

Let's review the last lecture by creating an improved version of the Maybe Monad, called the Ok Monad:

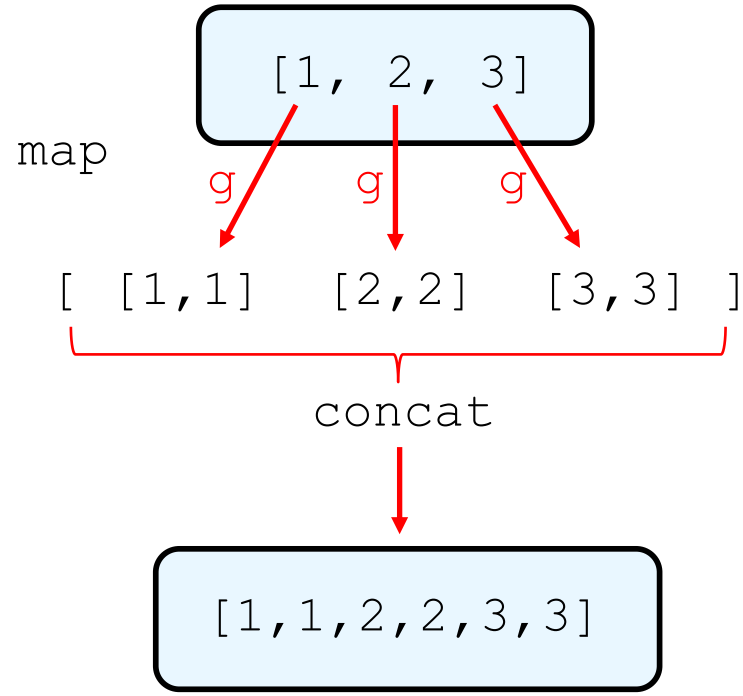
```
data Checked a = Ok a
               | Warning a String
               | Error String deriving (Show,Eq)
```

A review of the code (posted on the web as MonadLectureCode3.hs) is better than Powerpoint for this one....

List Monad

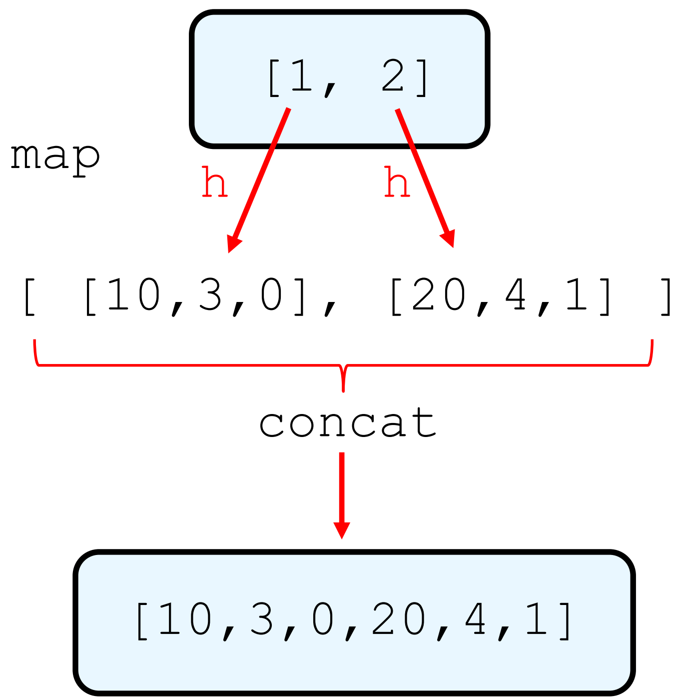
Another very useful monad is the List Monad, which is defined in the Prelude. The key to any monad is the definition of bind, so let's look at it right away and see what it does:

```
instance Monad [] where
    -- return :: a -> [a]
    return x = [x]

    --  (>>=) :: [a] -> (a -> [b]) -> [b]
    xs >>= f = concat (map f xs)

g :: a -> [a]
g x = [x,x]

[1,2,3] >>= g
     => concat ( map g [1,2,3] )
     => concat  [(g 1),(g 2),(g 3)]
     => concat  [[1,1],[2,2],[3,3]]
     => [1,1,2,2,3,3]
```



List Monad

```
instance Monad [] where
    -- return :: a -> [a]
    return x = [x]

    --  (>>=) :: [a] -> (a -> [b]) -> [b]
    xs >>= f = concat (map f xs)

h :: a -> [a]
h x = [x*10,x+2,x-1]

     [1,2] >>= h
 => concat ( map h [1,2] )
 => concat  [(h 1),(h 2)]
 => concat  [[10,2,0],[20,4,1]]
 => [10,2,0,20,4,1]
```



List Monad

```
instance Monad [] where
   -- return :: a -> [a]
   return x = [x]

   --  (>>=) :: [a] -> (a -> [b]) -> [b]
   xs >>= f = concat (map f xs)
```

Since what `concat` does is usually called "flattening a list," my motto for the List Monad is MAF = Map And Flatten!

Quick Quiz: What is the result of the following?

```
k :: a -> [a]
k x = [x,x,x]
```

```
                    [1,2] >>= k

            map:    ???

   and flatten:    ???
```

List Monad

```
instance Monad [] where
   -- return :: a -> [a]
   return x = [x]

   --  (>>=) :: [a] -> (a -> [b]) -> [b]
   xs >>= f = concat (map f xs)
```

Since what `concat` does is usually called "flattening a list," my motto for the List Monad is MAF = Map And Flatten!

Quick Quiz: What is the result of the following?

```
k :: a -> [a]
k x = [x,x,x]
```

```
                [1,2] >>= k

          map:  [[1,1,1],[2,2,2]]

  and flatten:  [1,1,1,2,2,2]
```

List Monad

```
instance Monad [] where
   -- return :: a -> [a]
   return x = [x]

   --  (>>=) :: [a] -> (a -> [b]) -> [b]
   xs >>= f = concat (map f xs)
```

Since what `concat` does is usually called "flattening a list," my motto for the List Monad is MAF = Map And Flatten!

Quick Quiz: What is the result of the following?

```
m :: a -> [a]
m x = [x]
```

```
[1,2] >>= m
```

List Monad

```
instance Monad [] where
    -- return :: a -> [a]
    return x = [x]

    --   (>>=) :: [a] -> (a -> [b]) -> [b]
    xs >>= f = concat (map f xs)
```

Since what `concat` does is usually called "flattening a list," my motto for the List Monad is MAF = Map And Flatten!

Quick Quiz: What is the result of the following?

```
m :: a -> [a]                  -- m is the same as return
m x = [x]
```

`[1,2] >>= m`

map: **`[[1],[2]]`**

and flatten: **`[1,2]`**

-- remember that `return`
-- is like the identity
-- for monads

List Monad

```
instance Monad [] where
   -- return :: a -> [a]
   return x = [x]

   --  (>>=) :: [a] -> (a -> [b]) -> [b]
   xs >>= f = concat (map f xs)
```

Since what `concat` does is usually called "flattening a list," my motto for the List Monad is MAF = Map And Flatten!

Quick Quiz: What is the result of the following?

```
z :: a -> [a]
z x = []
```

```
[1,2] >>= z
```

List Monad

```
instance Monad [] where
   -- return :: a -> [a]
   return x = [x]

   --   (>>=) :: [a] -> (a -> [b]) -> [b]
   xs >>= f = concat (map f xs)
```

Since what `concat` does is usually called "flattening a list," my motto for the List Monad is MAF = Map And Flatten!

Quick Quiz: What is the result of the following?

```
z :: a -> [a]
z x = []
```

```
                 [1,2] >>= z

            map: [[],[]]

   and flatten: []
```

List Monad

```
instance Monad [] where
   -- return :: a -> [a]
   return x = [x]

   --   (>>=) :: [a] -> (a -> [b]) -> [b]
   xs >>= f = concat (map f xs)
```

Since what `concat` does is usually called "flattening a list," my motto for the List Monad is MAF = Map And Flatten!

Quick Quiz: What is the result of the following?

```
n :: Integer -> [Integer]
n x = if even x then [x,x] else [x]
```

```
[1,2,3,4] >>= n
```

List Monad

```
instance Monad [] where
   -- return :: a -> [a]
   return x = [x]

   --  (>>=) :: [a] -> (a -> [b]) -> [b]
   xs >>= f = concat (map f xs)
```

Since what `concat` does is usually called "flattening a list," my motto for the List Monad is MAF = Map And Flatten!

Quick Quiz: What is the result of the following?

```
n :: Integer -> [Integer]
n x = if even x then [x,x] else [x]

                 [1,2,3,4] >>= n

           map:  [[1],[2,2],[3],[4,4]]

   and flatten:  [1,2,2,3,4,4]
```

List Monad

```
instance Monad [] where
   -- return :: a -> [a]
   return x = [x]

   --   (>>=) :: [a] -> (a -> [b]) -> [b]
   xs >>= f = concat (map f xs)
```

Since what `concat` does is usually called "flattening a list," my motto for the List Monad is MAF = Map And Flatten!

Quick Quiz: What is the result of the following?

```
q :: Integer -> [Integer]
q x = if even x then [x] else []
```

```
[1,2,3,4] >>= q
```

List Monad

```
instance Monad [] where
   -- return :: a -> [a]
   return x = [x]

   --   (>>=) :: [a] -> (a -> [b]) -> [b]
   xs >>= f = concat (map f xs)
```

Since what `concat` does is usually called "flattening a list," my motto for the List Monad is MAF = Map And Flatten!

Quick Quiz: What is the result of the following?

```
r :: Integer -> [Integer]
r x = if even x then [x] else []
```

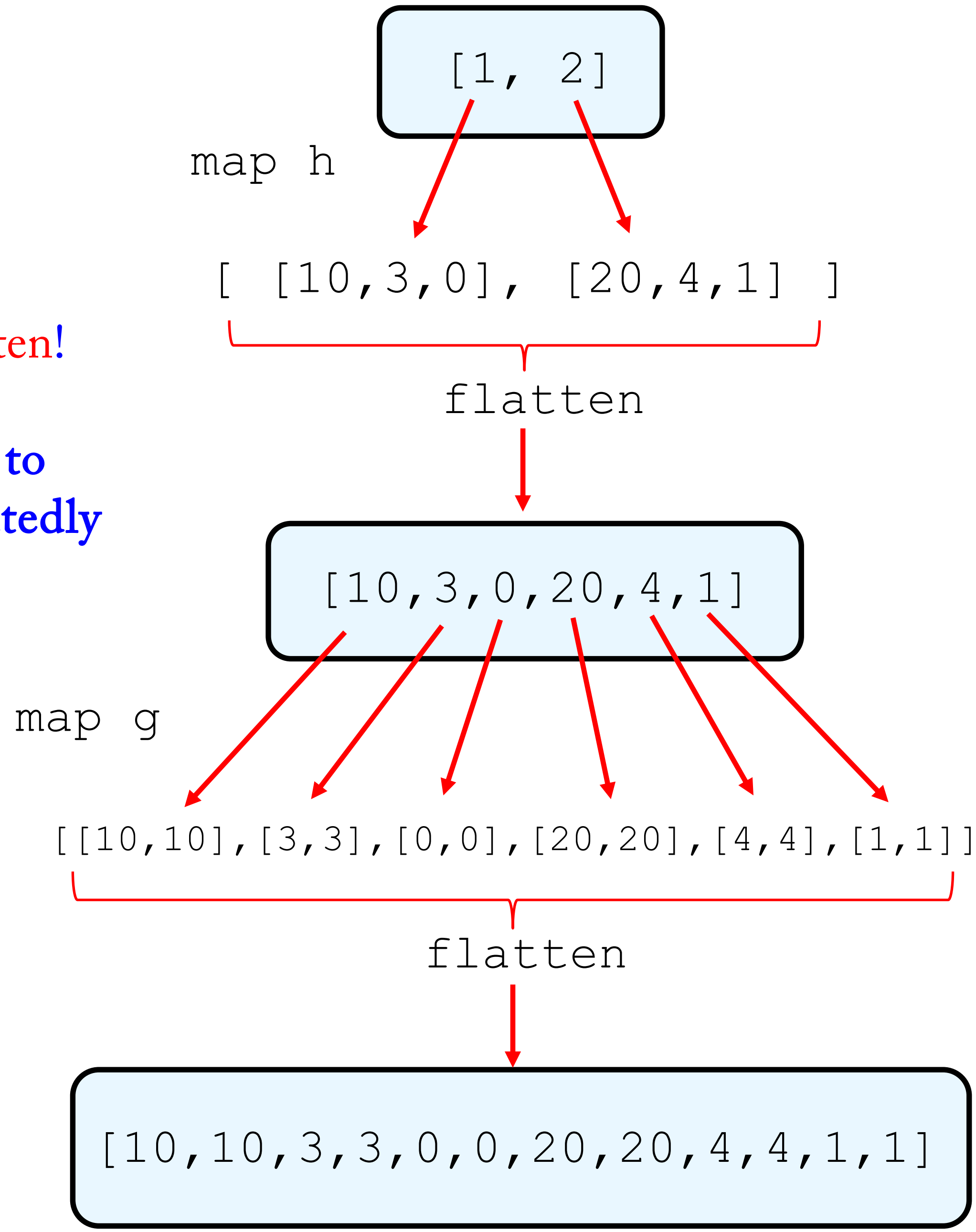
```
                  [1,2,3,4] >>= r

           map:   [[],[2],[],[4]]

   and flatten:   [2,4]
```

Ha! So you can use the List Monad to filter a list!

List Monad

Since what `concat` does is usually called "flattening a list," my motto for the List Monad is MAF = Map And Flatten!

The key to the List Monad, however, is to understand what it does when you repeatedly apply MAF:




```
g :: a -> [a]
g x = [x,x]

h :: a -> [a]
h x = [x*10,x+2,x-1]
```

`[1,2] >>= h >>= g`

List Monad

Repeated applications of MAF

```
g :: a -> [a]
g x = [x,x]

h :: a -> [a]
h x = [x*10,x+2,x-1]

r :: Integer -> [Integer]
r x = if even x then [x] else []
```

`[1,2] >>= h >>= g >>= r`

```
[1, 2]

map h and flatten

[10,3,0,20,4,1]

map g and flatten

[10,10,3,3,0,0,20,20,4,4,1,1]

map r and flatten

[10,10,0,0,20,20,4,4]
```

List Monad

Repeated applications of MAF

```
g :: a -> [a]
g x = [x,x]

h :: a -> [a]
h x = [x*10,x+2,x-1]

r :: Integer -> [Integer]
r x = if even x then [x] else []
```

`[1,2] >>= r >>= h >>= g`

`[1,2]`

map r and flatten

`[2]`

map h and flatten

`[20,4,1]`

map g and flatten

`[20,20,4,4,1,1]`

List Monad: Do Notation

```
g :: a -> [a]
g x = [x,x]

h :: a -> [a]
h x = [x*10,x+2,x-1]

r :: Integer -> [Integer]
r x = if even x then [x] else []
```

But of course we want to use **do** notation!

Let's translate this last example from bind to **do**, by way of lambda expressions:

```
[1,2] >>= r >>= h >>= g

[1,2] >>= (\x -> r x >>= (\y -> h y >>= (\z -> g z)))

[1,2] >>= \x -> r x >>= \y -> h y >>= \z -> g z

[1,2] >>= \x ->
r x   >>= \y ->
h y   >>= \z ->
g z

do x <- [1,2]
   y <- r x
   z <- h y
   g z
```

List Monad: Do Notation

```
g :: a -> [a]
g x = [x,x]

h :: a -> [a]
h x = [x*10,x+2,x-1]

r :: Integer -> [Integer]
r x = if even x then [x] else []
```

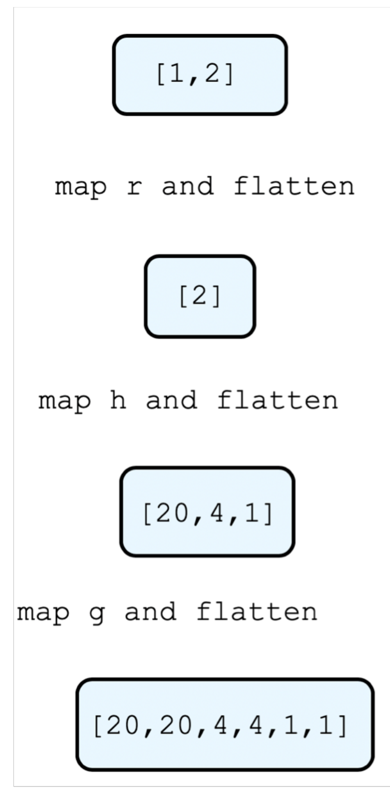
But of course we want to use do notation!

Let's translate this last example from bind to do, by way of lambda expressions:

```
[1,2] >>= r >>= h >>= g
```

```
[1,2] >>= (\x -> r x >>= (\y -> h y >>= (\z -> g z)))
```

```
do x <- [1,2]
   y <- r x
   z <- h y
   g z
```



Q: What do the variables x, y, z represent in the computation? What values do they take on?

List Monad: Do Notation

```
g :: a -> [a]
g x = [x,x]

h :: a -> [a]
h x = [x*10,x+2,x-1]

r :: Integer -> [Integer]
r x = if even x then [x] else []
```

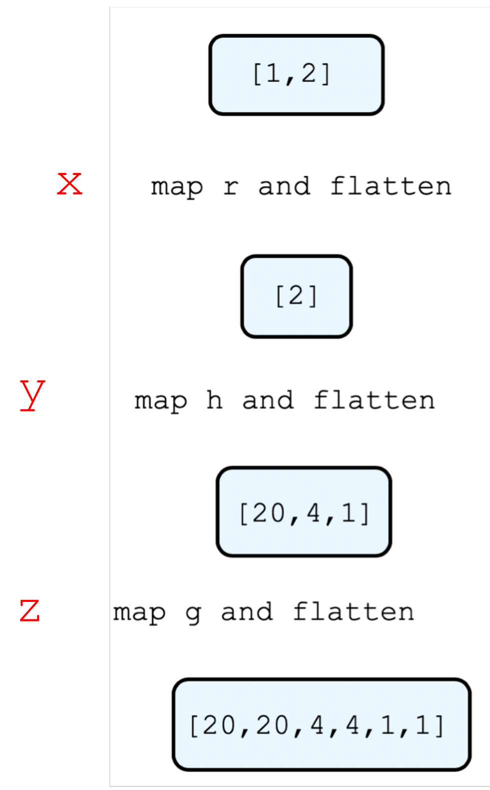
Let's translate this last example from `bind` to `do`, by way of lambda expressions:

```
[1,2] >>= r >>= h >>= g

[1,2] >>= (\x -> r x >>= (\y -> h y >>= (\z -> g z)))

do x <- [1,2]
   y <- r x
   z <- h y
   g z
```

Q: What do the variables x, y, z represent in the computation? What values do they take on?

A: They "iterate" through the list as the function is mapped onto the list:

x takes on the values 1, 2
y takes on the value 2
z takes on the values 20, 4, 1



List Monad: Do Notation

```
g :: a -> [a]
g x = [x,x]

h :: a -> [a]
h x = [x*10,x+2,x-1]

r :: Integer -> [Integer]
r x = if even x then [x] else []
```

Let's translate this last example from `bind` to `do`, by way of lambda expressions:

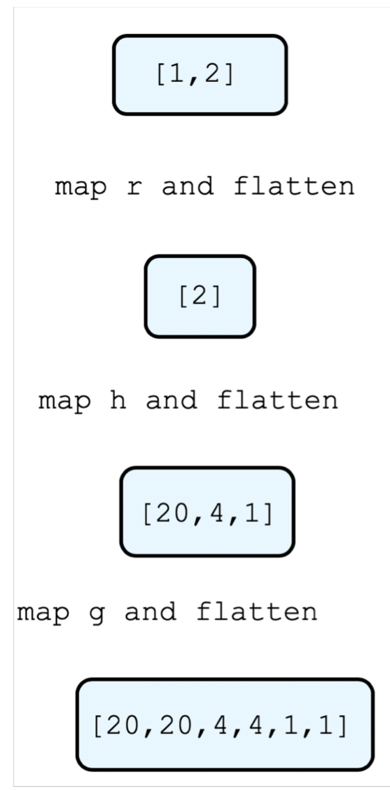
```
[1,2] >>= r >>= h >>= g
```

```
[1,2] >>= (\x -> r x >>= (\y -> h y >>= (\z -> g z)))
```

In other words, it is essentially the same as nested for loops in Python:

```
do x <- [1,2]
   y <- r x
   z <- h y
   g z
```

```
result = []
for x in [1,2]:
    for y in r(x):
        for z in h(y):
            result += g(z)
```



List Monad: Do Notation

```
g :: a -> [a]
g x = [x,x]

h :: a -> [a]
h x = [x*10,x+2,x-1]

r :: Integer -> [Integer]
r x = if even x then [x] else []
```

```
do x <- [1,2]
   y <- r x
   z <- h y
   g z
```

```
result = []
for x in [1,2]:
    for y in r(x):
        for z in h(y):
            result += g(z)
```

Where the test implemented by function r would be better expressed as:

```
result = []
    for x in [1,2]:
        if (x % 2 == 0):
            for z in h(x):
                result += g(z)
```

List Monad: Do Notation

```
g :: a -> [a]
g x = [x,x]

h :: a -> [a]
h x = [x*10,x+2,x-1]

r :: Integer -> [Integer]
r x = if even x then [x] else []
```

```
do x <- [1,2]
   y <- r x
   z <- h y
   g z
```

But rather than write this out with nested for loops in Python:

```
result = []
for x in [1,2]:
   if (x % 2 == 0):
      for z in h(x):
         result += g(z)
```

you could do it with a list comprehension:

```
[ w for w in g(z) for z in h(x) for x in [1,2] where (x % 2 == 0) ]
```

Wouldn't it be nice if we could do the same thing in Haskell? ....

List Monad: Do Notation and List Comprehensions

Haskell provides list comprehensions as "syntactic sugar" for **`do`** expressions with the List Monad:

Example:

```
do x <- [1,2]
   x <- r x
   y <- h x
   z <- g y
   return z
```

can be written as

```
[ z | x <- [1,2], even x, y <- h x, z <- g y ]
```

which you can read as: "For every x in [1,2], where x is even, for every y in (h x), and for every z in (g y), collect together all the z's into a list," or write in standard mathematical "set builder" notation as:

{ **z** | **x** ∈ **[1,2]** with **x** even ∧ **y** ∈ **h(x)** ∧ **z** ∈ **g(y)** }

List Monad

Summary: All three of these return the same list:

```
lst1 = [(x,y) | x <- [1,2,3] , y <- [1,2,3], x /= y]

lst2 = do x <- [1,2,3]
          y <- [1,2,3]
          if x == y then [] else [(x,y)]


lst3 = [1,2,3] >>= (\x -> [1,2,3] >>= (\y -> if x == y then [] else return (x,y)))
```

```
Main> lst1
[(1,2),(1,3),(2,1),(2,3),(3,1),(3,2)]

Main> lst2
[(1,2),(1,3),(2,1),(2,3),(3,1),(3,2)]

Main> lst3
[(1,2),(1,3),(2,1),(2,3),(3,1),(3,2)]
```

But I know which one I prefer!   Use List Comprehensions whenever possible!

List Monad

List comprehensions lead to all sorts of clever and elegant solutions to programming problems:

```
factors :: Integer -> [Integer]
factors n = [ x | x <- [1..n], n `mod` x == 0 ]

isPrime :: Integer -> Bool
isPrime n = factors n == [1,n]

primesLessThan :: Integer -> [Integer]
primesLessThan n = [ x | x <- [2 .. n], isPrime x ]


Main> primesLessThan 100
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,
71,73,79,83,89,97]
```

List Monad

To conclude, **quicksort** is a nice example of the power of list comprehensions:

```
qsort :: [Integer] -> [Integer]

qsort [] = []

qsort (x:xs) = (qsort small) ++ mid ++ (qsort large)
    where small = [ y | y <- xs, y < x]
          mid   = [ y | y <- xs, y == x] ++ [x]
          large = [ y | y <- xs, y > x]

Main> qsort [2,5,2,23,7,5,3,-90,5,6,4,213,74,56,-8]

[-90,-8,2,2,3,4,5,5,5,6,7,23,56,74,213]
```