Putting Laziness to Work
Why use laziness

• Laziness has lots of interesting uses
  – Build cyclic structures. Finite representations of infinite data.
  – Do less work, compute only those values demanded by the final result.
  – Build infinite intermediate data structures and actually materialize only those parts of the structure of interest.
    • Search based solutions using enumerate then test.
  – Memoize or remember past results so that they don’t need to be recomputed
Cyclic structures

• cycles :: [Int]
• cycles = 1 : 2 : 3 : cycles
Cyclic Trees

- \texttt{data Tree a = Tip a | Fork (Tree a) (Tree a)}

- \texttt{t2 = Fork (Fork (Tip 3) (Tip 4)) (Fork (Tip 9) t2)}
Mutually Cyclic

\[(t_3, t_4) = (\text{Fork}(\text{Fork}(\text{Tip 11}) t_3) t_4, \text{Fork}(\text{Tip 21}) (\text{Fork}(\text{Tip 33}) t_3))\]
Prime numbers and infinite lists

primes :: [Integer]
primes = sieve [2..]
    where sieve (p:xs) =
        p : sieve [x | x<-xs,
                      x `mod` p /= 0]
Dynamic Programming

• Consider the function

fib :: Integer -> Integer
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

LazyDemos> :set +s
LazyDemos> fib 30
1346269
(48072847 reductions, 78644372 cells, 1 garbage collection)

• takes about 9 seconds on my machine!
Why does it take so long?

• Consider \((\text{fib} \ 6)\)
What if we could remember past results?

• **Strategy**
  – Create a data structure
  – Store the result for every \((\text{fib } n)\) only if \((\text{fib } n)\) is demanded.
  – If it is ever demanded again return the result in the data structure rather than re-compute it

• **Laziness is crucial**

• **Constant time access is also crucial**
  – Use of functional arrays
Lazy Arrays

```haskell
import Data.Array

table = array (1,5) [(1,'a'),(2,'b'),(3,'c'),(5,'e'),(4,'d')]
```

- The array is created once
- Any size array can be created
- Slots cannot be over written
- Slots are initialized by the list
- Constant access time to value stored in every slot
Taming the duplication

```haskell
fib2 :: Integer -> Integer
fib2 z = f z
    where table = array (0,z) [ (i, f i) | i <- range (0,z) ]
          f 0 = 1
          f 1 = 1
          f n = (table ! (n-1)) + (table ! (n-2))
```

LazyDemos> fib2 30
1346269
(4055 reductions, 5602 cells)

Result is instantaneous on my machine
Can we abstract over this pattern?

• Can we write a memo function that memoizes another function.
• Allocates an array
• Initializes the array with calls to the function
• But, We need a way to intercept recursive calls
A fixpoint operator does the trick

• \texttt{fix f = f (fix f)}

• \texttt{g fib 0 = 1}
• \texttt{g fib 1 = 1}
• \texttt{g fib n = fib (n-1) + fib (n-2)}

• \texttt{fib1 = fix g}
Generalizing

memo :: Ix a => (a,a) -> ((a -> b) -> a -> b) -> a -> b

memo bounds g = f
    where arrayF = array bounds
        [ (n, g f n) | n <- range bounds ]
        f x = arrayF ! x

fib3 n = memo (0,n) g n

fact = memo (0,100) g
    where g fact n =
        if n==0 then 1 else n * fact (n-1)
Representing Graphs

import ST
import qualified Data.Array as A

type Vertex = Int

-- Representing graphs:

type Table a = A.Array Vertex a

type Graph = Table [Vertex]

-- Array Int [Int]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[2,3]</td>
</tr>
<tr>
<td>2</td>
<td>[7,4]</td>
</tr>
<tr>
<td>3</td>
<td>[5]</td>
</tr>
<tr>
<td>4</td>
<td>[6,9,7]</td>
</tr>
<tr>
<td>5</td>
<td>[8]</td>
</tr>
<tr>
<td>6</td>
<td>[9]</td>
</tr>
<tr>
<td>7</td>
<td>[9]</td>
</tr>
<tr>
<td>8</td>
<td>[10]</td>
</tr>
<tr>
<td>9</td>
<td>[10]</td>
</tr>
<tr>
<td>10</td>
<td>[]</td>
</tr>
</tbody>
</table>

Index for each node

Edges (out of) that index
Functions on graphs

type Vertex = Int

type Edge = (Vertex,Vertex)

vertices :: Graph -> [Vertex]

indices :: Graph -> [Int]

edges :: Graph -> [Edge]
Building Graphs

buildG :: Bounds -> [Edge] -> Graph

graph = buildG (1,10)

[ (1, 2), (1, 6), (2, 3),
  (2, 5), (3, 1), (3, 4),
  (5, 4), (7, 8), (7, 10),
  (8, 6), (8, 9), (8, 10) ]
data Tree a = Node a (Forest a)
type Forest a = [Tree a]

nodesTree (Node a f) ans =
    nodesForest f (a:ans)

nodesForest [] ans = ans
nodesForest (t : f) ans =
    nodesTree t (nodesForest f ans)

• Note how any tree can be spanned
• by a Forest. The Forest is not always
• unique.
The DFS algorithm finds a spanning forest for a graph, from a set of roots.

\[
\text{dfs} :: \text{Graph} \to [\text{Vertex}] \to \text{Forest Vertex}
\]

\[
dfs g vs = \text{prune} (A.\text{bounds} g) (\text{map} (\text{generate} g) vs)
\]

\[
\text{generate} :: \text{Graph} \to \text{Vertex} \to \text{Tree Vertex}
\]

\[
generate g v = \text{Node} v (\text{map} (\text{generate} g) (g `\text{aat`} v))
\]
import qualified Data.Array.ST as B

type Set s = B.STArray s Vertex Bool

mkEmpty :: Bounds -> ST s (Set s)
mkEmpty bnds = newSTArray bnds False

contains :: Set s -> Vertex -> ST s Bool
contains m v = readSTArray m v

include :: Set s -> Vertex -> ST s ()
include m v = writeSTArray m v True
Pruning already visited paths

```haskell
prune :: Bounds -> Forest Vertex -> Forest Vertex
prune bnds ts =
  runST (do { m <- mkEmpty bnds; chop m ts })

chop :: Set s -> Forest Vertex -> ST s (Forest Vertex)
chop m [] = return []
chop m (Node v ts : us)
  do { visited <- contains m v
       ; if visited
         then chop m us
       else do { include m v
                    ; as <- chop m ts
                    ; bs <- chop m us
                    ; return(Node v as : bs)
                 }
  }
```
Topological Sort

postorder :: Tree a -> [a]
postorder (Node a ts) = postorderF ts ++ [a]

postorderF :: Forest a -> [a]
postorderF ts = concat (map postorder ts)

postOrd :: Graph -> [Vertex]
postOrd = postorderF . Dff

dff :: Graph -> Forest Vertex
dff g = dfs g (vertices g)
A. Control Flow Graph
\[ a :: \text{Graph Char } a \]

B. DFS Labeled Graph
\[ b :: \text{Graph Char } (\text{Int},[\text{Char}]) \]
\[ \text{dfsnum} :: \text{v }\rightarrow\text{Int} \]
\[ \text{dfsnum } v = \text{fst}\left(\text{apply } b \ v\right) \]
\[ \text{dfspath} :: \text{v }\rightarrow\text{[v]} \]
\[ \text{dfspath } v = \text{snd}\left(\text{apply } b \ v\right) \]

C. Semi-Dominator Labeled Graph
\[ c :: \text{Graph Char Char} \]
\[ \text{semi} :: \text{v }\rightarrow\text{v} \]
\[ \text{semi } v = \text{apply } c \]

D. Semi-Dominator Graph
\[ d :: \text{Graph Char } a \]

E. Dominator Graph
\[ e :: \text{Graph Char } a \]