CS 457/557: Functional Languages

Leveraging Laziness

Mark P Jones
Portland State University

Lazy Evaluation:

With a **lazy** evaluation strategy:

- Don't evaluate until you have to
- When you do evaluate, save the result so that you can use it again next time …

Why use lazy evaluation?

- Avoids redundant computation
- Eliminates special cases (e.g., && and ||)
- Facilitates reasoning

Lazy evaluation encourages:

- Programming in a compositional style
- Working with "infinite data structures"
- Computing with "circular programs"

Compositional Style:

Separate aspects of program behavior separated into independent components

```
fact n          = product [1..n]

sumSqrs n       = sum (map (\x -> x*x) [1..n])

minimum         = head . sort
```

“Infinite” Data Structures:

Data structures are evaluated lazily, so we can specify “infinite” data structures in which only the parts that are actually needed are evaluated:

```
powersOfTwo = iterate (2*) 1
twoPow n        = powersOfTwo !! n

fibs      = 0 : 1 : zipWith (+) fibs (tail fibs)
fib n     = fibs !! n
```

Circular Programs:

An example due to Richard Bird ("Using circular programs to eliminate multiple traversals of data"):

Consider a tree datatype:

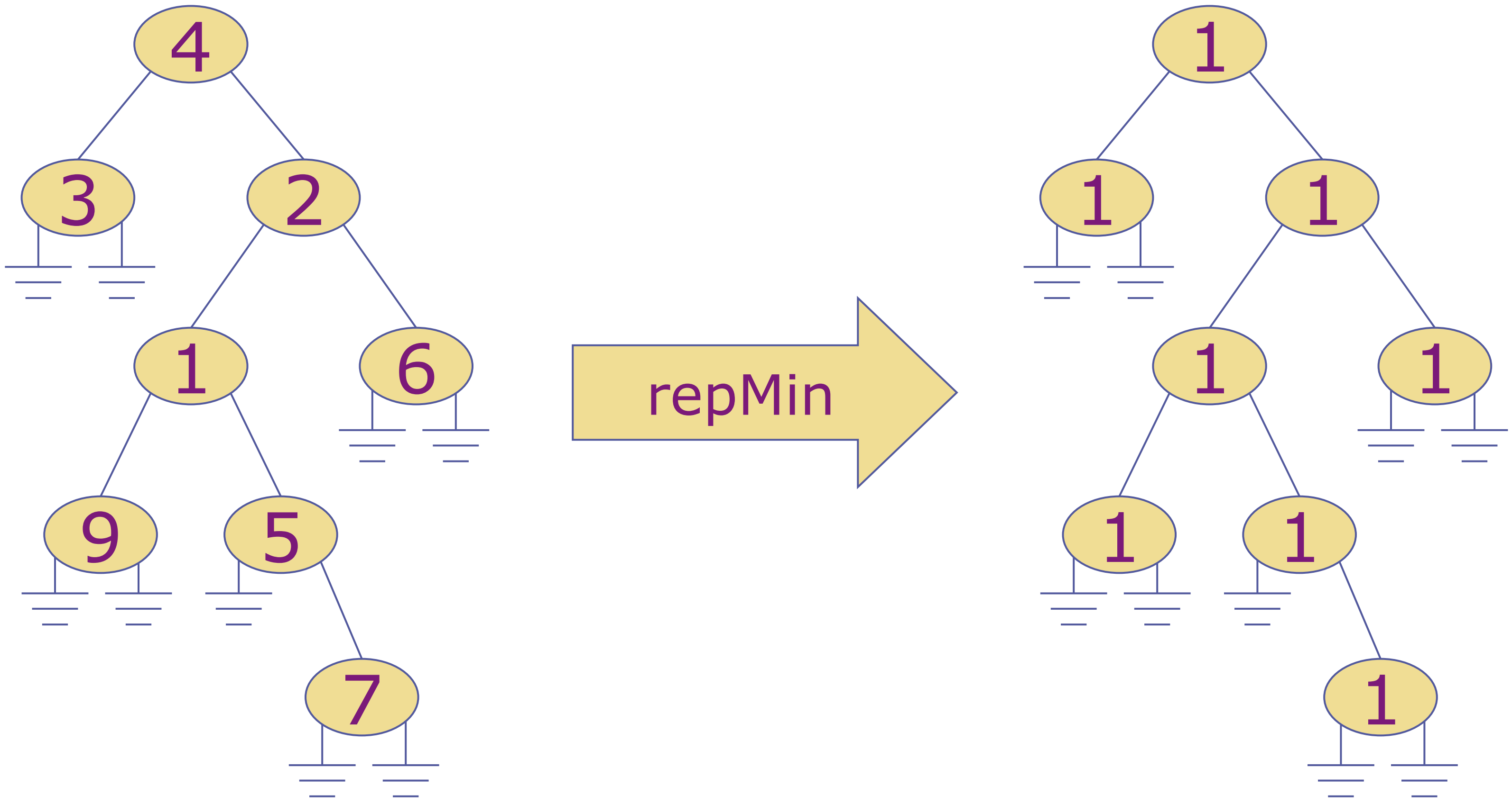
```
data Tree = Leaf | Fork Int Tree Tree
```

Define a function

```
repMin :: Tree -> Tree
```

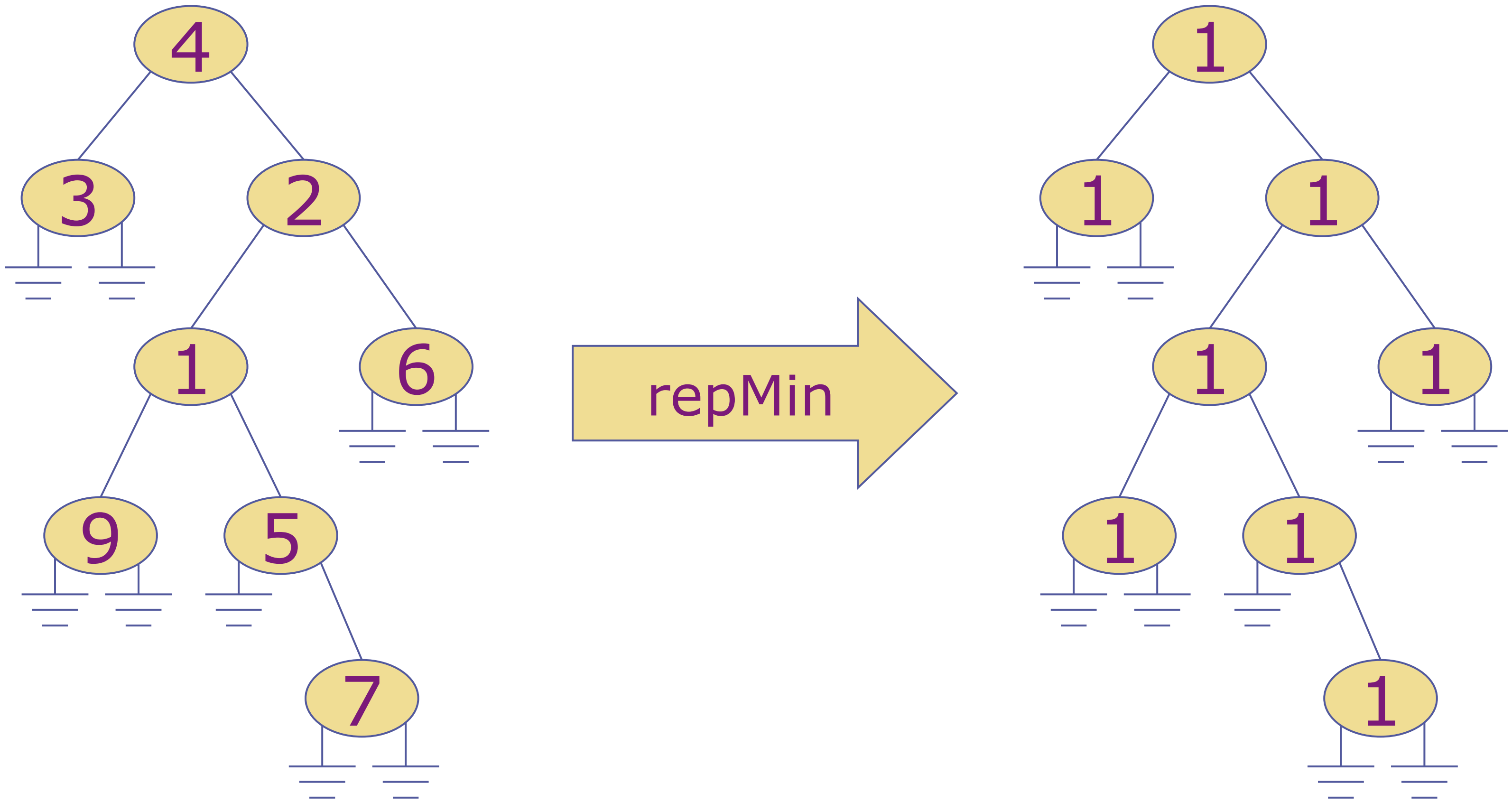
that will produce an output tree with the same shape as the input but replacing each integer with the minimum value in the original tree.

Example:



Same shape, values replaced with minimum

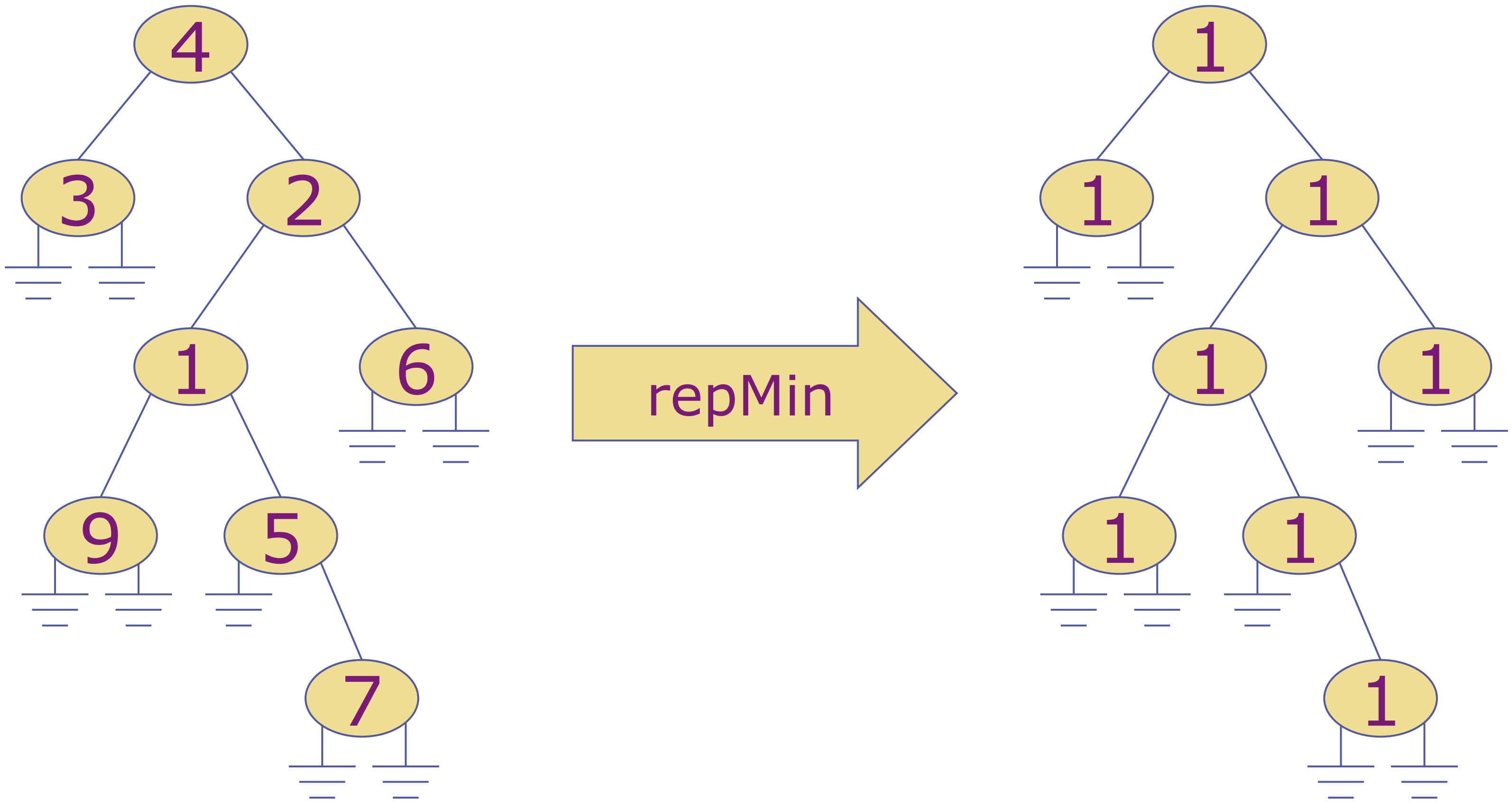
Example:



Obvious implementation:

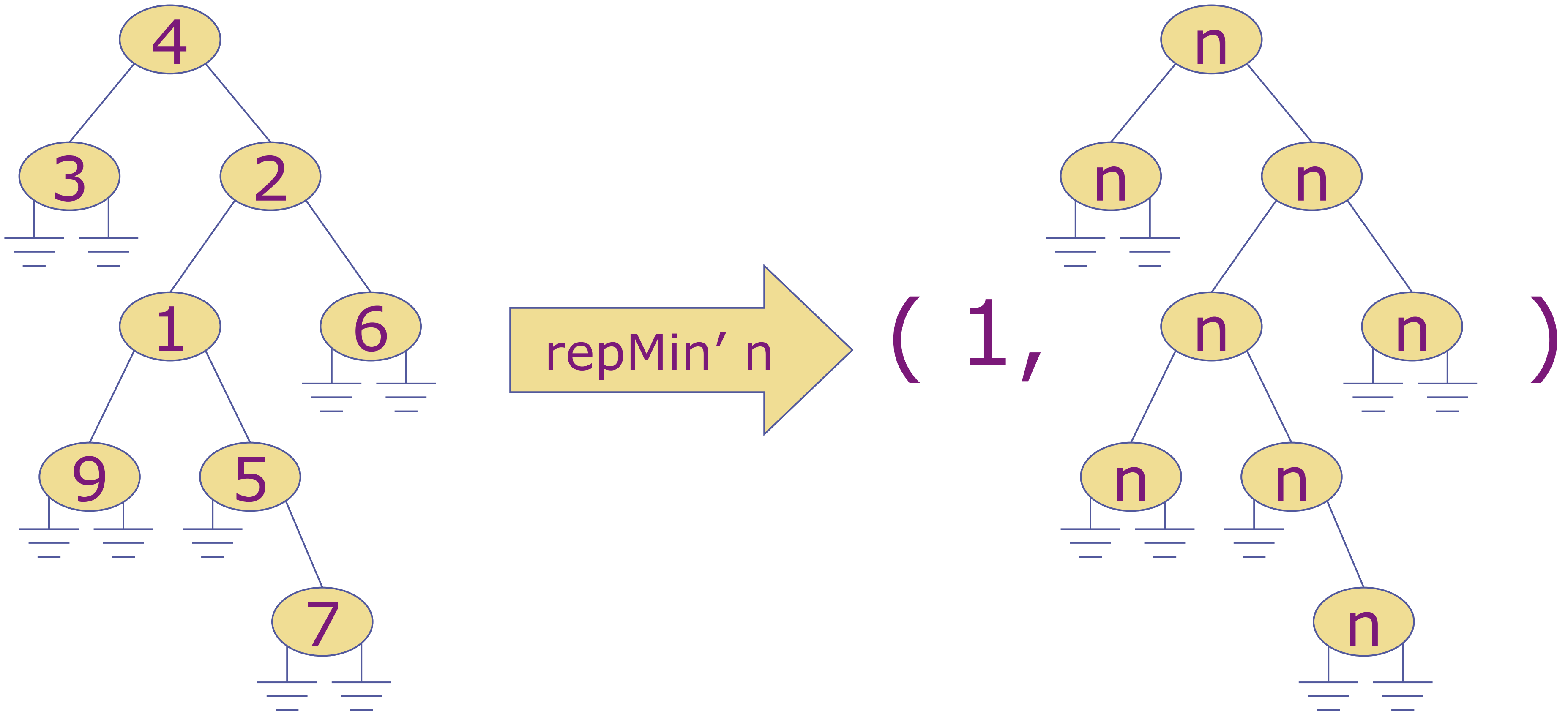
```
repMin t = mapTree (\n -> m) t
           where m = minTree t
```

Example:



Can we do this with only one traversal?

A Slightly Easier Problem:



In a single traversal:

- Calculate the minimum value in the tree
- Replace each entry with some given n

A Single Traversal:

We can code this algorithm fairly easily:

```
repMin'              :: Int -> Tree -> (Int, Tree)
repMin' n Leaf       = (maxInt, Leaf)
repMin' n (Fork m l r)
                     = (min nl nr, Fork n l' r')
                       where
                         (nl, l')  = repMin' n l
                         (nr, r') = repMin' n r
```

"Tying the knot"

- Now a call repMin′ m t will produce a pair (n, t′) where
  - n is the minimum value of all the integers in t
  - t′ is a tree with the same shape as t but with each integer replaced by m.
- We can implement repMin by creating a cyclic structure that passes the minimum value that is returned by repMin′ as its first argument:

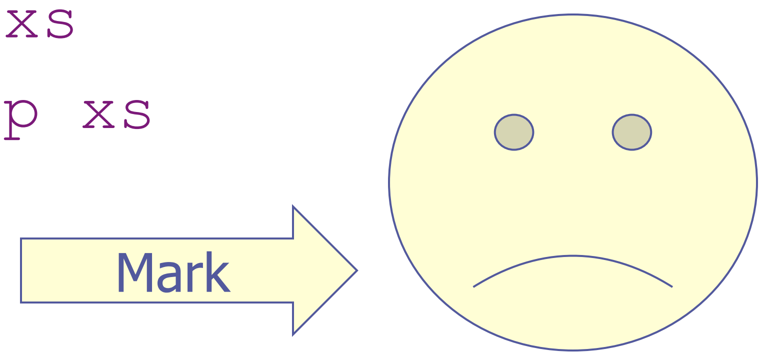
repMin t = t′ **where** (n, t′) = repMin′ n t

Aligning Separators: a more realistic example

Mark is Fussy about Layout:

Have you noticed how I get fussy about code like:

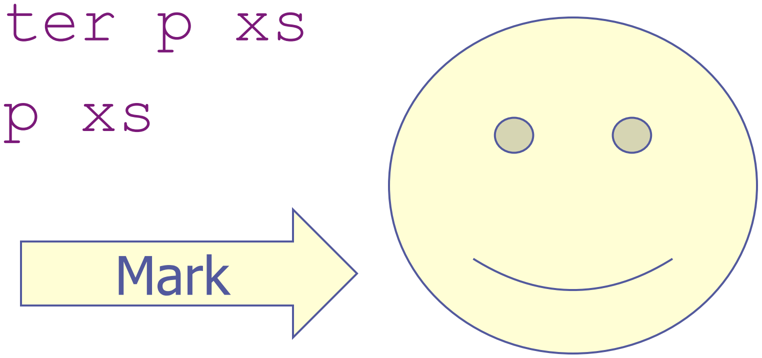
```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs

filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
    | p x = x : filter p xs
    | otherwise = filter p xs
```



Mark is Fussy about Layout:

… and try to line up the separators like this:

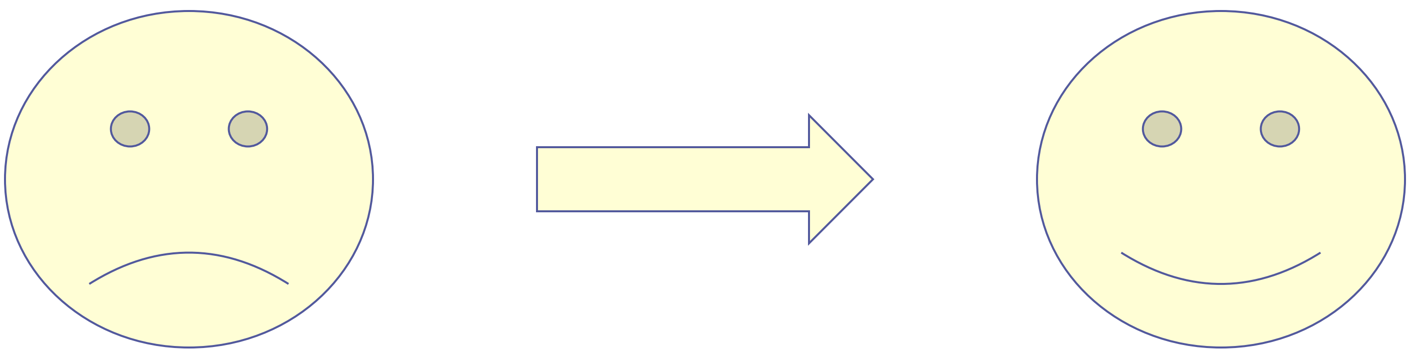
```
map           :: (a -> b) -> [a] -> [b]
map f []       = []
map f (x:xs)   = f x : map f xs

filter        :: (a -> Bool) -> [a] -> [a]
filter p []    = []
filter p (x:xs)
   | p x       = x : filter p xs
   | otherwise = filter p xs
```



Can we do this Automatically?

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

filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
   | p x = x : filter p xs
   | otherwise = filter p xs
```

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

filter        :: (a -> Bool) -> [a] -> [a]
filter p []    = []
filter p (x:xs)
   | p x       = x : filter p xs
   | otherwise = filter p xs
```



Thinking about an Algorithm:

Let’s look at this line by line:

```
6    map ::  (a -> b) -> [a] -> [b]
10   map f [] = []
14   map f (x:xs) = f x : map f xs

9    filter :: (a -> Bool) -> [a] -> [a]
13   filter p [] = []
     filter p (x:xs)
10       | p x = x : filter p xs
16       | otherwise = filter p xs
```

Maximum

Total # chars up to and including first separator

Thinking about an Algorithm:

Let’s look at this line by line:

<table>
<tr><td>10</td><td>6</td><td><code>map :: (a -> b) -> [a] -> [b]</code></td></tr>
<tr><td>6</td><td>10</td><td><code>map f [] = []</code></td></tr>
<tr><td>2</td><td>14</td><td><code>map f (x:xs) = f x : map f xs</code></td></tr>
<tr><td>0</td><td></td><td></td></tr>
<tr><td>7</td><td>9</td><td><code>filter :: (a -> Bool) -> [a] -> [a]</code></td></tr>
<tr><td>3</td><td>13</td><td><code>filter p [] = []</code></td></tr>
<tr><td>0</td><td></td><td><code>filter p (x:xs)</code></td></tr>
<tr><td>6</td><td>10</td><td><code> | p x = x : filter p xs</code></td></tr>
<tr><td>0</td><td>16</td><td><code> | otherwise = filter p xs</code></td></tr>
</table>

extra chars to insert before first separator

Some Preliminaries:

```
separators        :: [String]
separators         = [ "=", "::" ]

pad               :: Int -> String -> String
pad n s            = take n (s ++ repeat ' ')
```

Patching Lines:

Target length to end of first separator

Input string

Actual length to end of first separator

Output string

```
patchLine      :: Int -> String -> (Int, String)
patchLine n cs = head (matches ++ [(0, cs)])
 where
```

Find first match

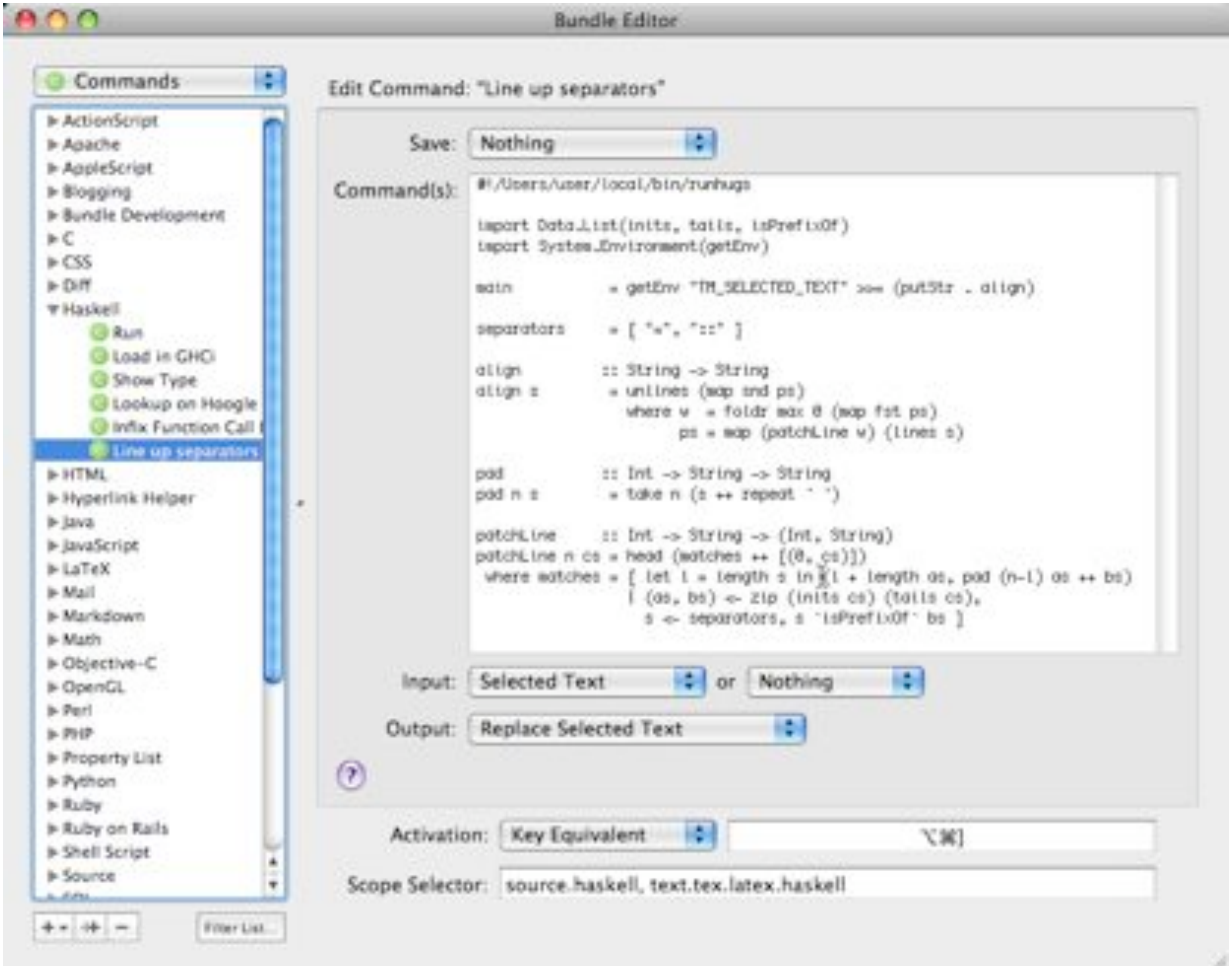
Default case

```
   matches = [ let l = length s
               in (l + length as,
                   pad (n-l) as ++ bs)
             | (as, bs) <- zip (inits cs)
                               (tails cs),
               s <- separators,
               s `isPrefixOf` bs ]
```

Tying the Knot (again):

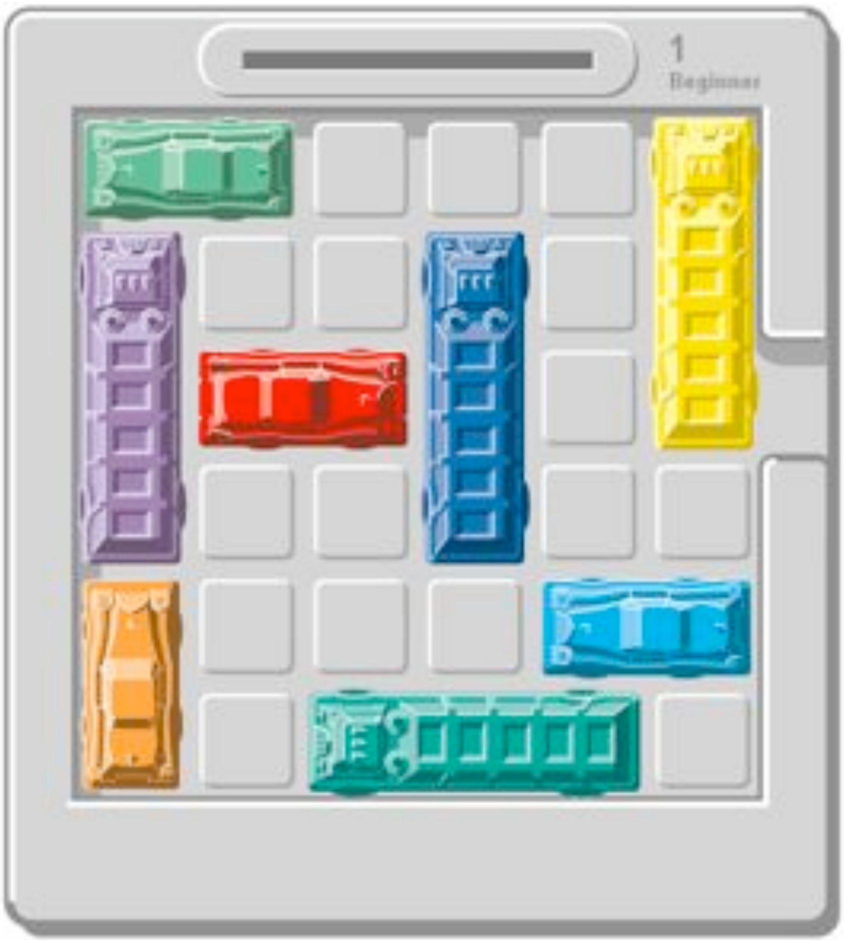
```
main     :: IO ()
main      = getEnv "TM_SELECTED_TEXT"
            >>= (putStr . align)

align    :: String -> String
align s   = unlines (map snd ps)
     where w  = foldr max 0 (map fst ps)
           ps = map (patchLine w) (lines s)
```

An Editor Plugin:



Combining Techniques of Lazy Programming

RUSH HOUR
TRAFFIC JAM PUZZLE



"Escape! That's the goal.

Rush Hour is a premier sliding block puzzle designed to challenge your sequential-thinking skills (and perhaps your traffic-officer aspirations as well)."


BINARY ARTS®


BINARY ARTS®


BINARY ARTS®


BINARY ARTS


BINARY ARTS™


BINARY ARTS®


BINARY ARTS®


BINARY ARTS®


BINARY ARTS®

A Rush Hour Solver:

Uses lazy evaluation in three important ways:

- Written in compositional style
- Natural use of an infinite data structure (a search tree that is subsequently pruned to a finite tree that eliminates duplicate puzzle positions)
- Cyclic programming techniques used to implement breadth-first pruning of the search tree.

Representing the Board:

```
type Position  = (Coord, Coord)
type Coord     = Int

maxw, maxh     :: Coord
maxw           = 6
maxh           = 6
```

Representing the Pieces:

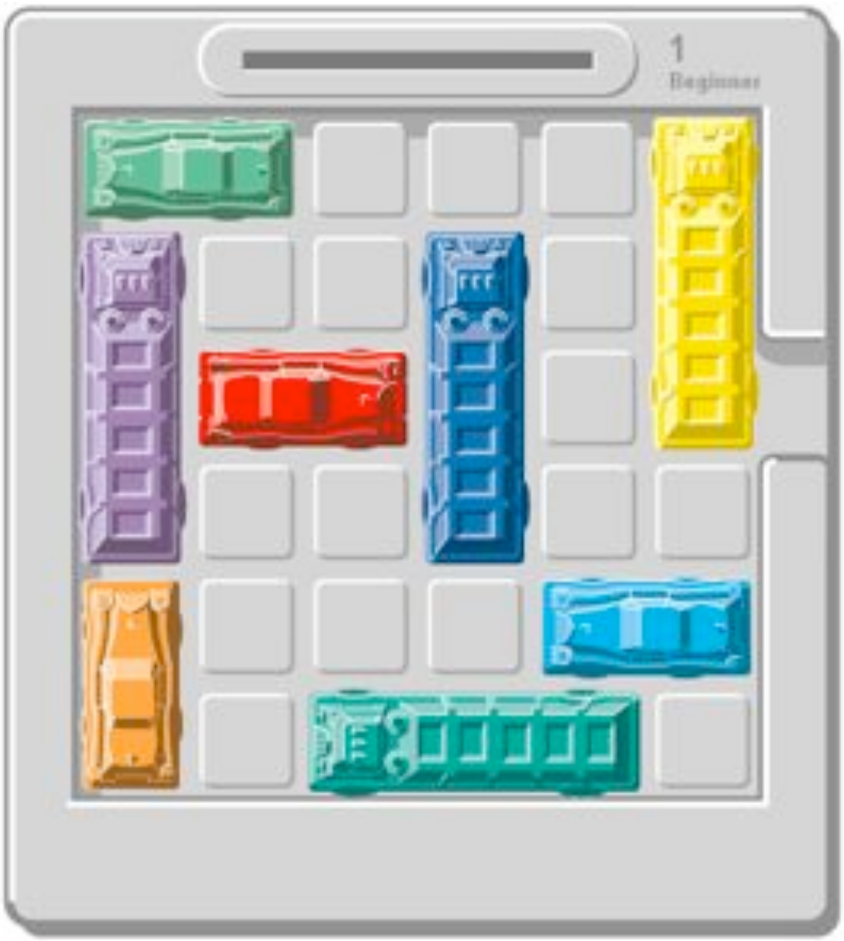
```
type Vehicle = (Color, Type)

data Color    = Red | … | Emerald
                deriving (Eq, Show)

data Type     = Car | Truck
                deriving (Eq, Show)

len          :: Type -> Int
len Car       = 2
len Truck     = 3
```

Representing Puzzles:

```
type Puzzle           = [Piece]
type Piece            = (Vehicle, Position, Orientation)

data Orientation      = W | H

vehicle              :: Piece -> Vehicle
vehicle (v, p, o)     = v

solved               :: Piece -> Bool
solved p              = p == ((Red, Car), (4,3), W)
```

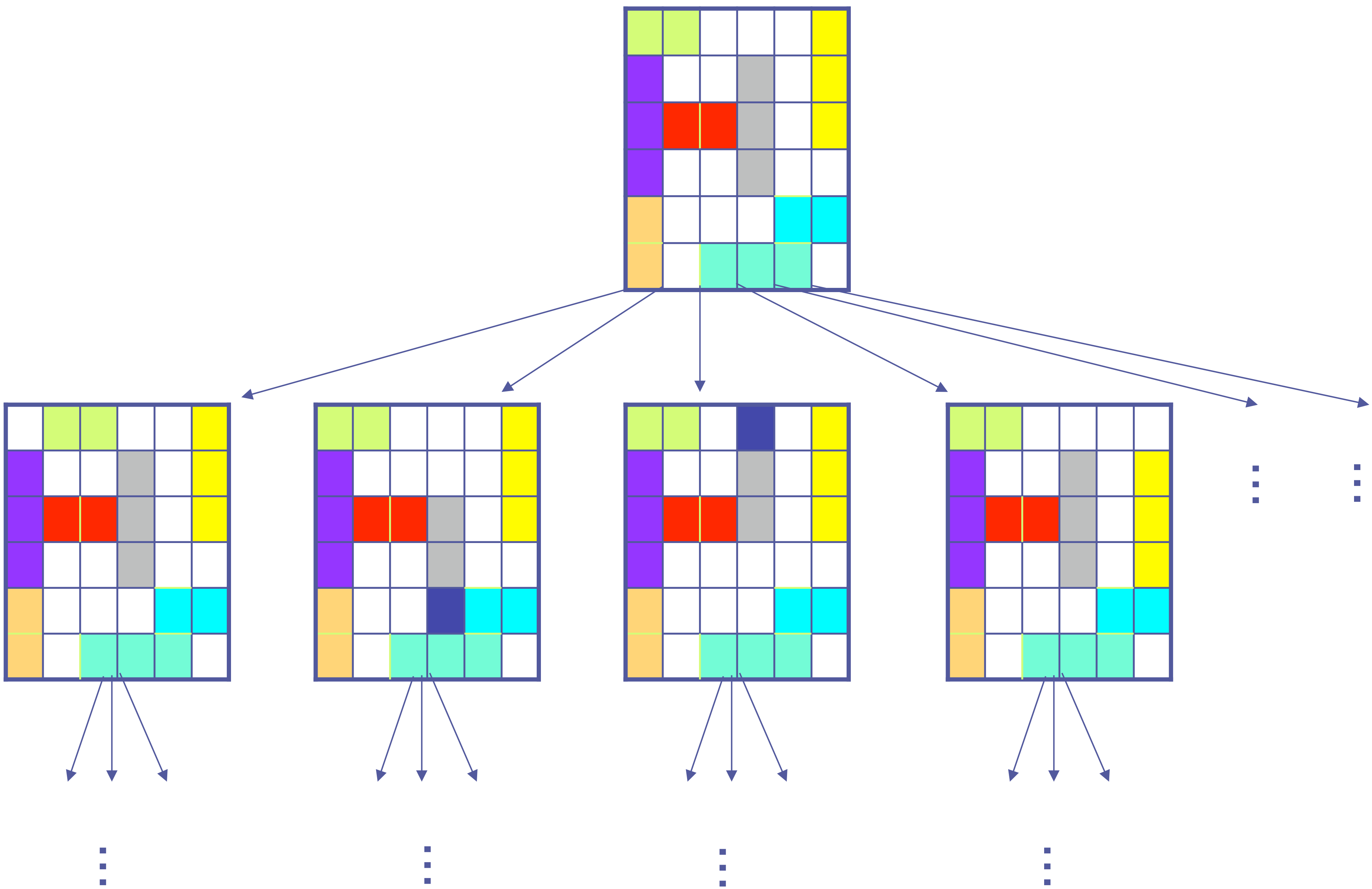
RUSH HOUR

TRAFFIC JAM PUZZLE




```
puzzle1 :: Puzzle
puzzle1 =
 [  ((LtGreen, Car),    (0,5), W),
    ((Yellow, Truck),   (5,3), H),
    ((Violet, Truck),   (0,2), H),
    ((Blue, Truck),     (3,2), H),
    ((Red, Car),        (1,3), W),
    ((Orange, Car),     (0,0), H),
    ((LtBlue, Car),     (4,1), W),
    ((Emerald, Truck),  (2,0), W) ]
```

From Moves to Trees:



Checking for Obstructions:

```
puzzleObstructs :: Puzzle -> Position -> Bool
puzzleObstructs puzzle pos
                 = or [ pieceObstructs p pos | p<-puzzle ]

pieceObstructs  :: Piece -> Position -> Bool
pieceObstructs ((c,t), (x,y), W) (u,v)
                 = (y==v) && (x<=u) && (u<x+len t)
pieceObstructs ((c,t), (x,y), H) (u,v)
                 = (x==u) && (y<=v) && (v<y+len t)
```

Calculating Moves:

```
moves                  :: Puzzle -> Piece -> [Piece]
moves puzzle piece = step back piece ++ step forw piece
 where
  back                  :: Piece -> Maybe Piece
  back (v, (x,y), W)
      | x>0 && free p = Just (v, p, W)
                          where p = (x-1, y)
  ...
  free                   = not . puzzleObstructs puzzle

  step                  :: (a -> Maybe a) -> a -> [a]
  step dir p             = case dir p of
                             Nothing -> []
                             Just p' -> p' : step dir p'
```

Forests and Trees:

```
type Forest a     = [Tree a]
data Tree a       = Node a [Tree a]

mapTree          :: (a -> b) -> Tree a -> Tree b
mapTree f (Node x cs)
                  = Node (f x) (map (mapTree f) cs)

pathsTree :: Tree a -> Tree [a]
pathsTree  = descend []
  where descend xs (Node x cs)
             = Node xs' (map (descend xs') cs)
               where xs' = x:xs
```

Making Trees:

```
forest       :: Puzzle -> Forest (Piece, Puzzle)
forest ps     = [ Node (m, qs) (forest qs)
                | (as, p, bs) <- splits ps,
                  m <- moves (as++bs) p,
                  let qs = as ++ [m] ++ bs ]

splits       :: [a] -> [([a], a, [a])]
splits xs     = … exercise to the reader …
```

```
(e.g., splits "dog"
          = [("",'d',"og"),("d",'o',"g"),("do",'g',"")])
```

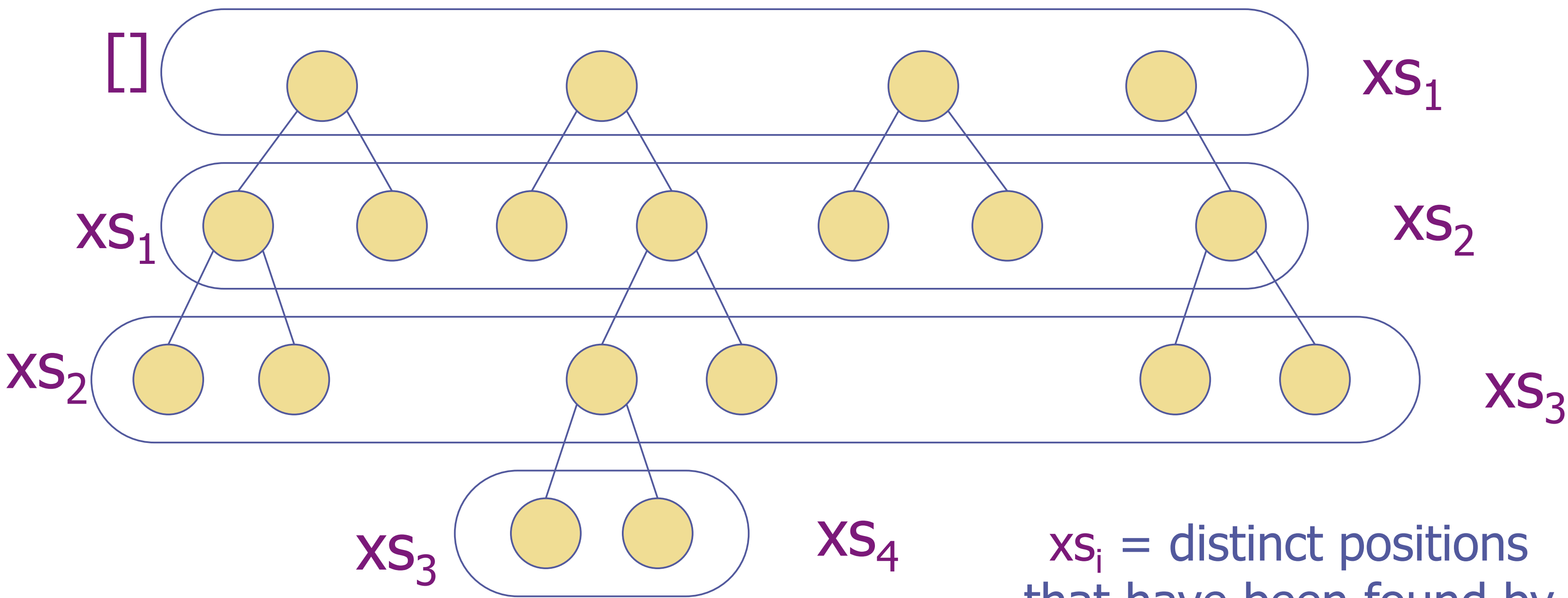
Pruning the Tree:

- We want to avoid puzzle solutions in which the same piece is moved in two successive turns
- The generated tree may contain many instances of this pattern
- We can prune away repetition using:

```
trimRel   :: (a -> a -> Bool) -> Tree a -> Tree a
trimRel rel (Node x cs)
           = Node x (filter (\(Node y _) -> rel x y) cs)
```

Eliminating Duplicate Puzzles:

- We don't want to explore any single puzzle configuration more than once
- We want to find shortest possible solutions (requires breadth-first search of the forest)



xs_i = distinct positions that have been found by the end of the i^{th} level

```haskell
trimDups :: Eq b => (a -> b) -> Forest a -> Forest a
trimDups val f = f'
 where
  (f', xss)= prune f ([]:xss)

  prune [] xss = ([], xss)
  prune (Node v cs : ts) xss
    = let x = val v in
      if x `elem` head xss
       then prune ts xss
       else let (cs', xss1) = prune cs (tail xss)
                (ts', xss2)
                      = prune ts ((x:head xss):xss1)
            in (Node v cs' : ts', xss2)
```

knot tying

infinite list

Breadth-First Search:

```
bfs :: Tree t -> [t]
bfs  = concat . bft

bft (Node x cs) = [x] : bff cs
bff             = foldr (combine (++)) [] . map bft

combine :: (a -> a -> a) -> [a] -> [a] -> [a]
combine f (x:xs) (y:ys) = f x y : combine f xs ys
combine f []     ys     = ys
combine f xs     []     = xs
```

The Main Solver:

Written in a fully compositional style

```
solve :: Puzzle -> IO ()
solve  = putStrLn
       . unlines
       . map show
       . reverse
       . head
       . filter (solved . head)
       . concat
       . bff
       . map (pathsTree . mapTree fst)
       . trimDups (\(p,ps) -> ps)
       . map (trimRel (\(v,ps) (w,qs) -> vehicle v /= vehicle w))
       . forest
```

Summary:

- Laziness provides new ways (with respect to other paradigms) for us to think about and express algorithms
- Enhanced modularity from compositional style, infinite data structures, etc…
- Novel programming techniques like knot tying/circular programs …
- Further Reading:
  - Why Functional Programming Matters, John Hughes
  - The Semantic Elegance of Applicative Languages, D. A. Turner
  - Using Circular Programs to Eliminate Multiple Traversals of Data Structures, Richard Bird