**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]`
- `sumSqr 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:

```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
```
```haskell```
```text```
```
```text```
**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:**

```
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
```

```
In a single traversal:
  • Calculate the minimum value in the tree
  • Replace each entry with some given n
```

**“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:

```haskell
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
```

... and try to line up the separators like this:

```haskell
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?

Let's look at this line by line:

```haskell
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:

```haskell
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
```

Some Preliminaries:

```haskell
separators :: [String]
separators = [ "=" , "::" ]
pad :: Int -> String -> String
pad n s = take n (s ++ repeat ' ')
```
Patching Lines:

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

```haskell
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).”
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:

```haskell
type Position = (Coord, Coord)
type Coord = Int
maxw, maxh :: Coord
maxw = 6
maxh = 6
```

Representing the Pieces:

```haskell
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:

```haskell
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)
```
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)]

Checking for Obstructions:

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

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)
...

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:

  \[
  \text{trimRel} :: (a \to a \to \text{Bool}) \to \text{Tree } a \to \text{Tree } a \\
  \text{trimRel } \text{rel} (\text{Node } x \text{ cs}) = \text{Node } x (\text{filter } (\text{\lambda} (\text{Node } y \_ \to \text{rel } x \ y \) \text{ 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)

Breadth-First Search:

\[
\text{bfs} :: \text{Tree } t \to \text{[t]} \\
\text{bfs} = \text{concat . bft} \\
\text{bft } (\text{Node } x \text{ cs}) = [x] : \text{bff } cs \\
\text{bff} = \text{foldr (combine (++) \[] . map bft} \\
\text{combine } :: (a \to a \to a) \to \text{[a]} \to \text{[a]} \to \text{[a]} \\
\text{combine } f \left( (x:xs) \ (y:ys) \right) = f \ x \ y : \text{combine } f \ xs \ ys \\
\text{combine } f \ [] \ ys \ = \ ys \\
\text{combine } f \ xs \ [] \ = \ xs
\]

The Main Solver:

\[
\text{solve} :: \text{Puzzle } \to \text{IO } () \\
\text{solve} = \text{putStrLn . unlines . map show . reverse . head . filter (solved . head) . concat . bft . map (pathsTree . mapTree fst) . trimDups (\((p,p) \to ps\) . map (trimRel \((v,ps) \ (w,qs) \to \text{vehicle } v /= \text{vehicle } w)) . forest}
\]

Written in a fully compositional style

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