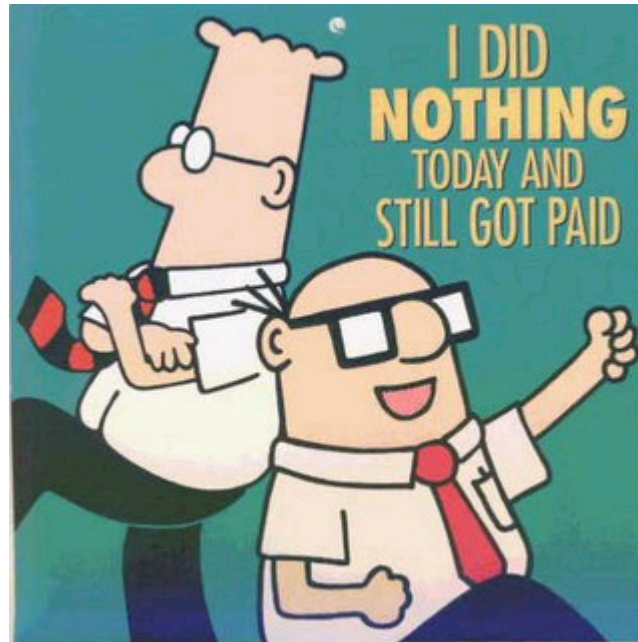


# Laziness



# Another Function

```
expensive :: Integer -> Integer
expensive n
  | n <= 1    = 1
  | otherwise = expensive (n-1)
                + expensive (n-2)

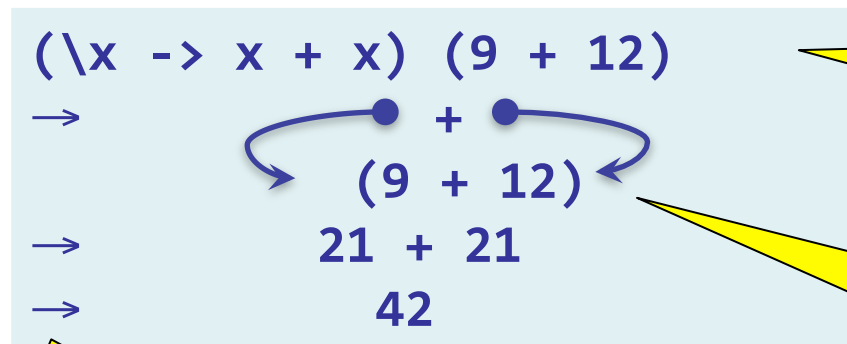
choice :: Bool -> a -> a -> a
choice False x y = x
choice True  x y = y
```

```
Main> choice False 17 (expensive 99999)
17
```

Without delay...

# Laziness

- Haskell is a *lazy* language
  - A particular function argument is only evaluated when it is **needed**, and
  - if it is needed then it is evaluated just once



“apply” needs  
the function

(+) needs  
its  
arguments

A computation model  
called **graph reduction**

# When is a Value "Needed"?

```
strange :: Bool -> Integer  
strange False = 17  
strange True  = 17
```

```
Main> strange undefined  
Program error: undefined
```

An argument  
is evaluated  
when a  
*pattern match*  
occurs

use undefined or error  
to test if something *is*  
evaluated

But also *primitive*  
*functions* evaluate  
their arguments

# Lazy Programming Style

- Separate
  - Where the computation of a value is defined
  - Where the computation of a value happens



Modularity!

# Backtracking

- E.g. the Suduko lab
- Write an expression which represents all valid solutions to a problem and pick the first one.
- Laziness ensures that we do not generate more than we need

# At Most Once?

```
apa :: Integer -> Integer  
apa x = (f x)^2 + f x + 1
```

```
Main> apa (6^2)
```

6<sup>2</sup> evaluated once but  
f (36) is evaluated twice

```
bepa :: Integer -> Integer -> Integer  
bepa x y = f 17 + x + y
```

```
Main> bepa 1 2 + bepa 3 4
```

...

Quiz: How to  
avoid  
recomputation?

f 17 is  
evaluated  
twice

# At Most Once!

```
apa :: Integer -> Integer
apa x = v^2 + v + 1
      where v = f x
```

```
bepa :: Integer -> Integer -> Integer
bepa x y = f17 + x + y
```

```
f17 :: Integer
f17 = f 17
```

The compiler might  
also perform these  
optimisations

ghc -O or  
ghc -ffull-laziness

# Infinite Lists

- Because of laziness, values in Haskell can be *infinite*
- Do not compute them completely!
  - Instead, only use parts of them

```
take n [3..]
```

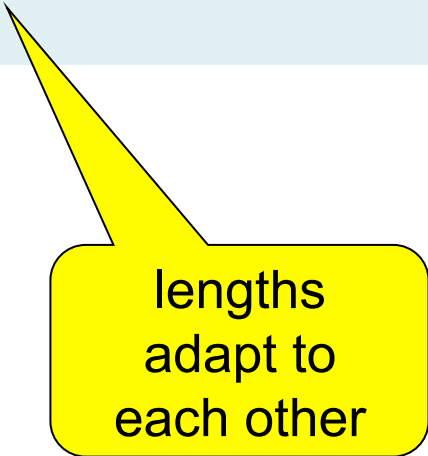
```
xs `zip` [1..]
```

# Examples

- Uses of infinite lists

# Example: PrintTable

```
printTable :: [String] -> IO ()
printTable xs =
  sequence_ [ putStrLn (show i ++ ":" ++ x)
            | (x,i) <- xs `zip` [1..]
            ]
```



lengths  
adapt to  
each other

# Iterate

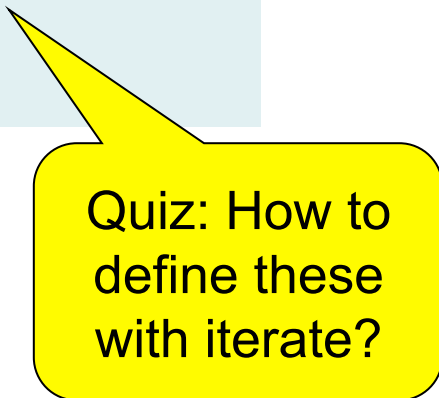
```
iterate :: (a -> a) -> a -> [a]  
iterate f x = x : iterate f (f x)
```

```
Main> iterate (*2) 1  
[1,2,4,8,16,32,64,128,256,512,1024,...]
```

# Other Handy Functions

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

cycle :: [a] -> [a]
cycle xs = xs ++ cycle xs
```



Quiz: How to  
define these  
with iterate?

# Alternative Definitions

```
repeat :: a -> [a]  
repeat x = iterate id x
```

```
cycle :: [a] -> [a]  
cycle xs = concat (repeat xs)
```

# Replicate

```
replicate :: Int -> a -> [a]  
replicate n x = take n (repeat x)
```

```
Main> replicate 5 'a'  
"aaaaa"
```

# Problem: Grouping List Elements

```
group :: Int -> [a] -> [[a]]  
group = ?
```

```
Main> group 3 "apabepacepa!"  
["apa", "bep", "ace", "pa!"]
```

# Problem: Grouping List Elements

```
group :: Int -> [a] -> [[a]]
group n = takeWhile (not . null)
         . map (take n)
         . iterate (drop n)
```

. connects "stages"  
--- like Unix pipe  
symbol |

# Problem: Prime Numbers

```
primes :: [Integer]
primes = ?
```

```
Main> take 4 primes
[2,3,5,7]
```

# Problem: Prime Numbers

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

*Eratosthenes' sieve* – cf.  
Exercise in Week 4

# Infinite Datastructures

```
data Labyrinth
  = Crossroad
  { what    :: String
  , left   :: Labyrinth
  , right  :: Labyrinth
  }
```



How to make an interesting labyrinth?

# Infinite Datastructures

```
labyrinth :: Labyrinth
labyrinth = start
  where
    start  = Crossroad "start"  forest town
    town   = Crossroad "town"   start  forest
    forest = Crossroad "forest" town   exit
    exit   = Crossroad "exit"   exit   exit
```

What happens  
when we print  
this structure?

# Lazy IO

```
headFile f = do
  c <- readFile f
  let c' = unlines . take 5 . lines $ c
  putStrLn c'
```

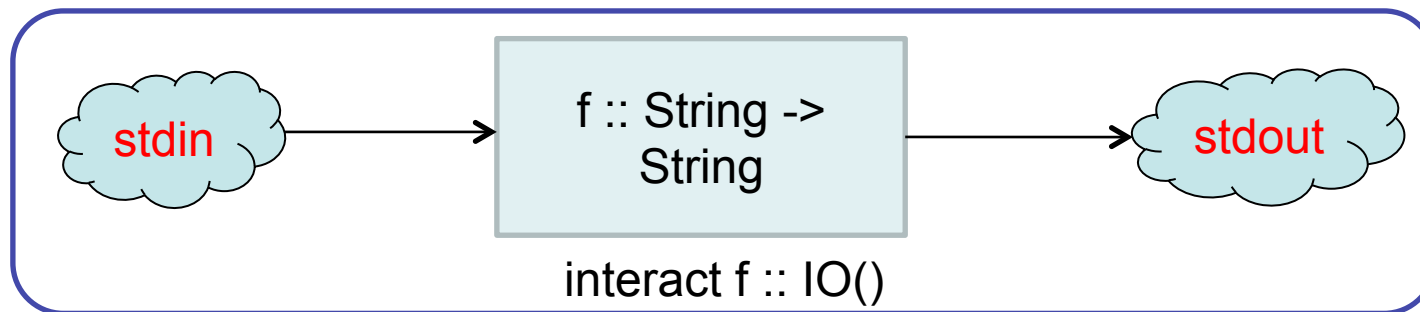
*Does not actually read in  
the whole file*

*Need to print  
causes just 5  
lines to be read*

# Lazy IO

- Common pattern: take a function form String to String, connect **stdin** to the input and **stdout** to the output

```
interact :: (String -> String) -> IO()
```



# Lazy IO

```
import Network.HTTP.Base(urlEncode)

encodeLines = interact $
  unlines . map urlEncode . lines
```

**Main>** *encodeLines*

*hello world*

hello%20world

*2+3=5*

2%2B3%3D5

...

# Controlling Laziness

- Haskell includes some features to reduce the amount of laziness allowing us to decide *when* something gets evaluated
- Used for performance tuning, particularly for controlling space usage
- Not recommended that you mess with this unless you have to – hard to get right in general



# Example

- Sum of a list of numbers

```
million :: Integer  
million = 1000000
```

```
Main sum [1..million]  
** Exception: Stack overflow **
```

# Example

- sum of a list of numbers

```
sum' :: [Integer] -> Integer
sum' [] = 0
sum' (x:xs) = x + sum' xs
```

```
million = 1000000 :: Integer
```

```
Main sum' [1..million]
```

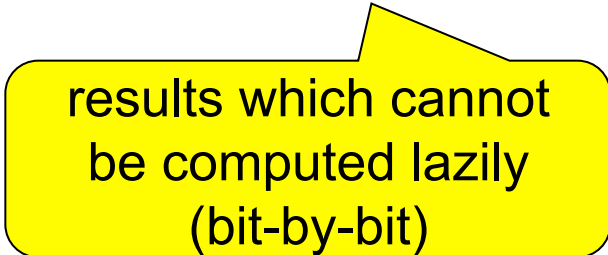
```
** Exception: Stack overflow **
```

Not a problem of  
lazy evaluation!  
All languages will  
have problems  
with this

1 + (2 + (3 + (4 + ...

# Tail Recursion

- Important concept in non-lazy functional programming for efficient recursion
- Also useful in Haskell for recursive functions which compute a basic typed result (Integer, Double, Int, ...)

A yellow callout box with a black border and a pointer pointing towards the text "(Integer, Double, Int, ...)".

results which cannot  
be computed lazily  
(bit-by-bit)

# Tail Recursion

- A function is tail recursive if the recursive call itself produces the result

Example

```
last :: [a] -> a
last [x]      = x
last (x:xs)   = last xs
```

The recursive call  
is the whole result

Tail recursion uses no stack space. Can be compiled to an unconditional jump

# Tail Recursive Sum

```
sum' :: [Integer] -> Integer
```

```
sum' = s 0
```

```
  where s acc [] = acc
```

```
        s acc (x:xs) = s (acc+x) xs
```

tail recursive  
helper function

- Not typically used with lazy data (e.g. lists) since it stops us producing any of the result list until the last step of the recursion

# The Tail Recursive Pattern: foldl

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

```
foldl f v ( a : b : c : ... )
gives (...(v `f` a) `f` b) `f` c ) `f` ...
```

```
sum = foldl (+) 0
```

# Problem solved?

- Lazy evaluation is too lazy!

```
sum' [1..million]
  == s 0 [1..million]
  == s (0+1) [2..million]
  == s (0+1+2) [3..million]
  ...
```

Not computed until needed.  
i.e., at the millionth recursive  
call!

# Controlling laziness: seq

- Haskell includes a primitive function

$$\text{seq} :: a \rightarrow b \rightarrow b$$

which evaluates its first argument then returns its second.

The prelude also defines a strict application operation:

```
($!) :: (a -> b) -> (a -> b)
f $! x = x `seq` (f x)
```

"strict" is used to mean the opposite of "lazy"

# Strictness

- The compiler looks for arguments which will eventually be needed and will insert `seq` in appropriate places. E.g.

```
sum' :: [Integer] -> Integer
sum' = s 0
  where s acc []       = acc
        s acc (x:xs) = acc `seq` s (acc+x) xs
```

compile with  
optimisation:  
ghc -O

force acc to be  
simplified on each  
recursive call

# Strict Tail Recursion: foldl'

```
import Data.List(foldl')

foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' f v [] = v
foldl' f v (x:xs) = let a = f v x in
                    a `seq` foldl' f a xs
```

# Example

- Average of a list of numbers

```
average :: [Integer] -> Integer
average xs = sum' xs `div`
            fromIntegral (length xs)

million :: Integer
million = 1000000
```

```
Main> average (replicate million 1)
** Exception: Stack overflow **
```

making sum and  
length tail recursive  
and strict does not  
solve the problem

# Space Leak

- This problem is often called a **space leak**
  - sum forces us to build the whole of [1..million]
  - lazyness (“at most once”) requires us to **keep the list** in memory since it is going to be used by length
  - if we only computed sum then the garbage collector would collect it as we go along.

# Solution

- Make average use tail recursion by computing sum and length at the same time:

```
average xs = av 0 0 xs where
  av sm len []      = sm `div` fromIntegral len
  av sm len (x:xs) = sm `seq`
                    len `seq`
                    av (sm + x) (len + 1) xs
```

# Gotcha: seq is still quite lazy!

seq forces evaluation of first argument, but  
**only as far as the first constructor**

```
> undefined `seq` 3
```

```
*** Exception: Prelude.undefined
```

```
> (undefined,2) `seq` 3
```

```
3
```

# Example: sumlength

```
sumlength = foldl' f (0,0)
  where f (s,l) a = (s+a,l+1)
```

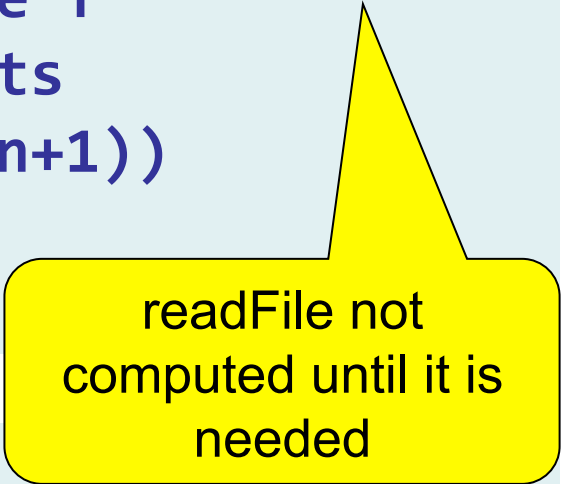
The pair is already "evaluated", so the seq has no effect

```
sumlength = foldl' f (0,0)
  where f (s,l) a = let (s',l') = (s+a,l+1)
                    in s' `seq` l' `seq` (s',l')
```

force the evaluation of the components before the pair is constructed

# Lazyness and IO

```
count :: String -> IO Int
count f = do contents <- readFile f
            let n = read contents
                writeFile f (show (n+1))
            return n
```



readFile not  
computed until it is  
needed

```
Main> count "testfile"
*** Exception "testfile": openFile resource
busy (file is locked)
```

# Lazyness and IO

```
count :: String -> IO Int
count f = do contents <- readFile f
             let n = read contents
                 n `seq` writeFile f (show (n+1))
             return n
```

- Often lazy IO is “just the right thing”
- Need to control it sometimes
  - Usually solve this by working at the level of file handles. See e.g. System.IO

# Conclusion

- Laziness
  - Evaluate "at most once"
  - programming style
- Do not have to use it
  - But powerful tool!
- Can make programs more "modular"
- Performance issues tricky
  - evaluation can be controlled using e.g. tail recursion and strictness. Best avoided unless necessary