CS 457/557: Functional Languages

Lazy Evaluation

Mark P Jones and Andrew Tolmach

Portland State University

What is "Lazy Evaluation"?

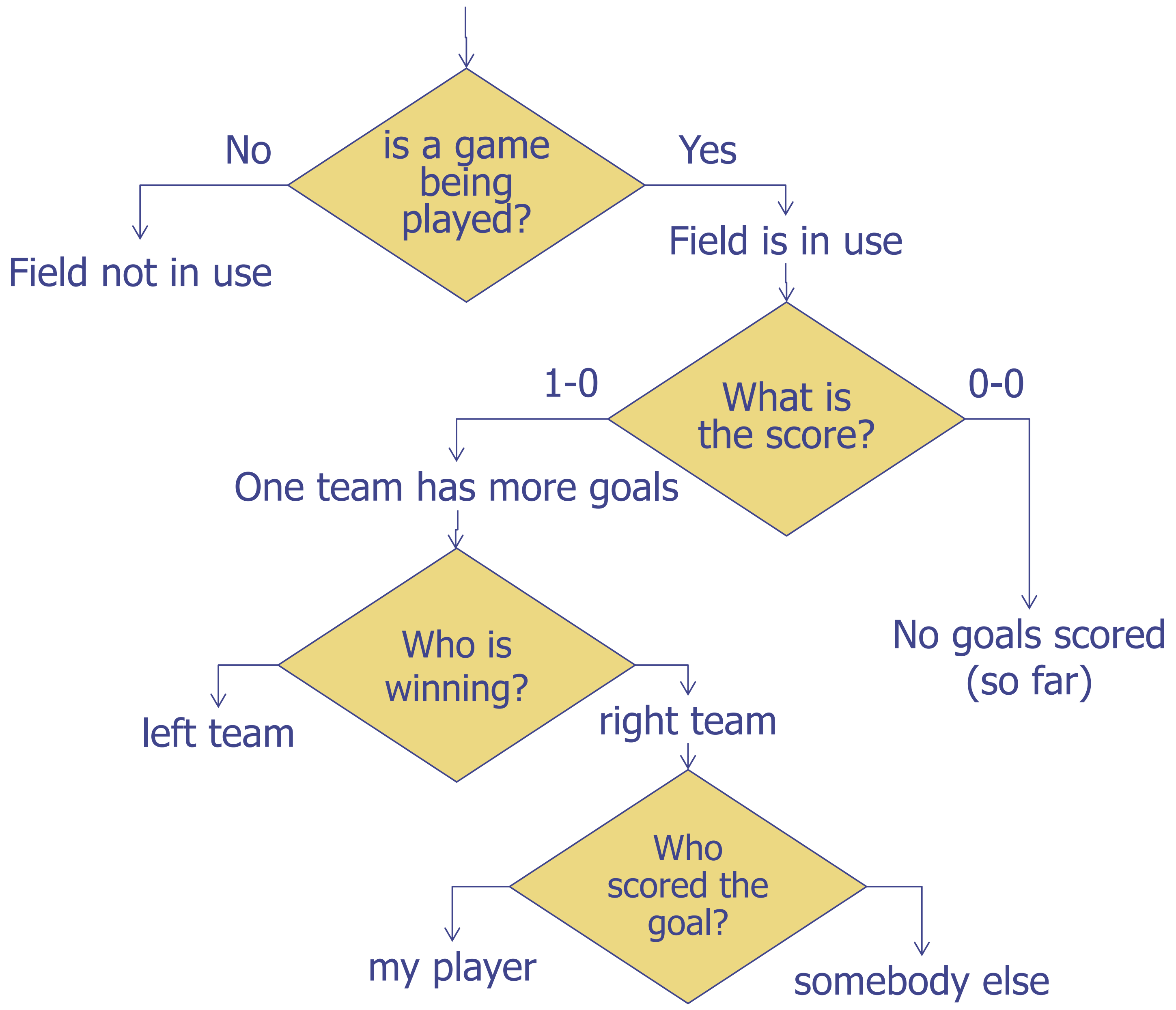
With a **lazy** evaluation strategy:

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

Also called **non-strict** evaluation, **call-by-need** evaluation, or **demand-driven** evaluation

In some sense, an opposite to **eager** / **strict** / **call-by-value** evaluation strategies

The Soccer Field in the Park



Evaluation on Demand

- You have to ask a series of simple questions to learn about the result of a computation
- Every answer gives us a little more information
- We only get answers to questions that we ask
- You don't have to ask the same question twice
- Initially, we have "no information", ☐
- You might not want to know everything about the result

Lazy Evaluation in Practice

Do not evaluate any part of an expression until its value is needed

```
(\x -> 42) (head []) == 42
head [1..] == 1
foldr (&&) True (repeat False) == False
```

but

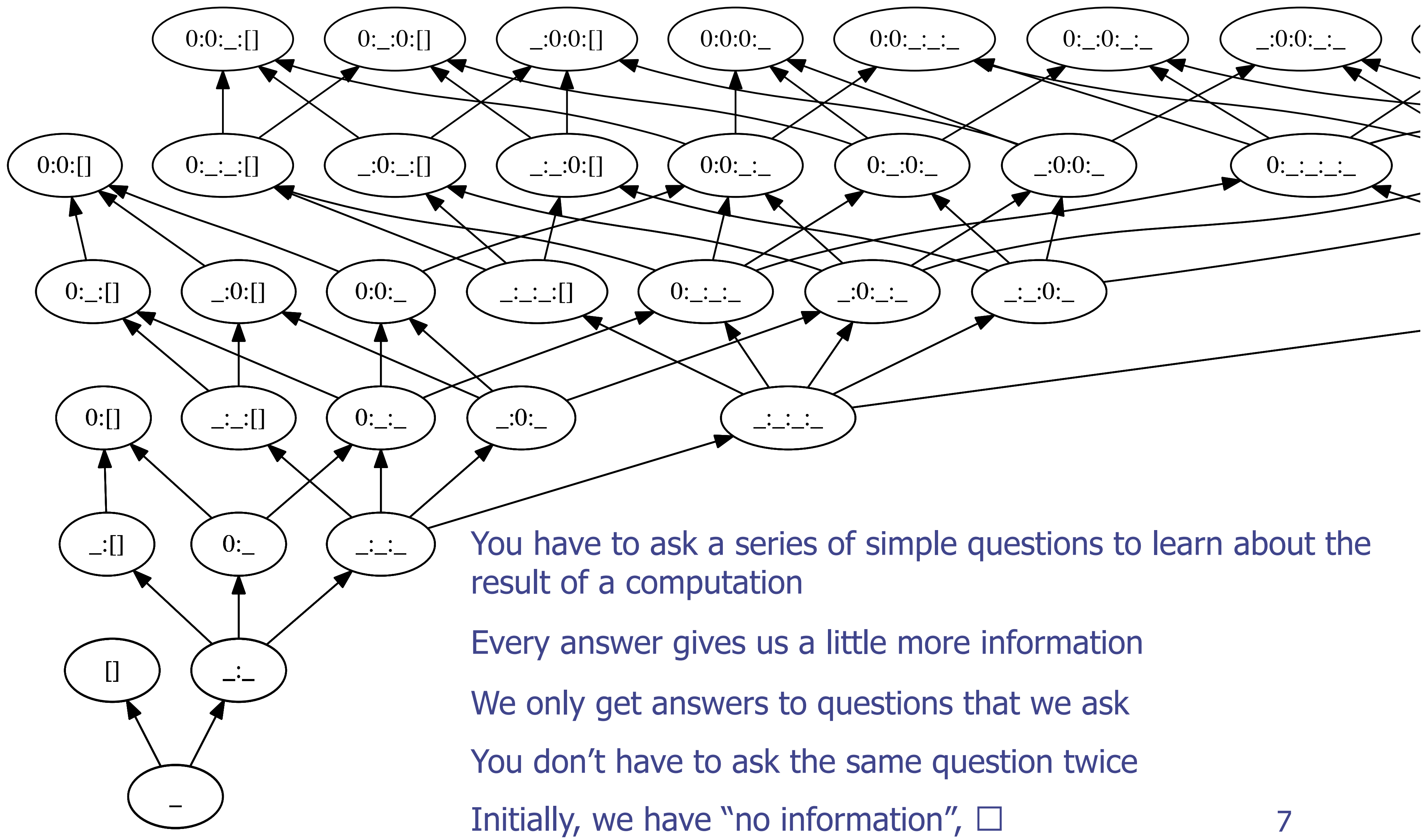
```
foldr (&&) True (repeat True) == ⊥
```

Lazy Evaluation in Practice

```
cheap      = length expensive
expensive= [ fib 23 | i <- [1..5] ]
midrange = [ last expensive | i <- [1..5] ]
```

```
Hugs> :set +s
Main> cheap
5
(156 reductions, 241 cells)

Main> midrange
[28657,28657,28657,28657,28657]
(1655884 reductions, 2638042 cells, 2 garbage collections)

Main> expensive
[28657,28657,28657,28657,28657]
(6622851 reductions, 10551205 cells, 10 garbage collections)

Main>
```

Evaluation on Demand



You have to ask a series of simple questions to learn about the result of a computation

Every answer gives us a little more information

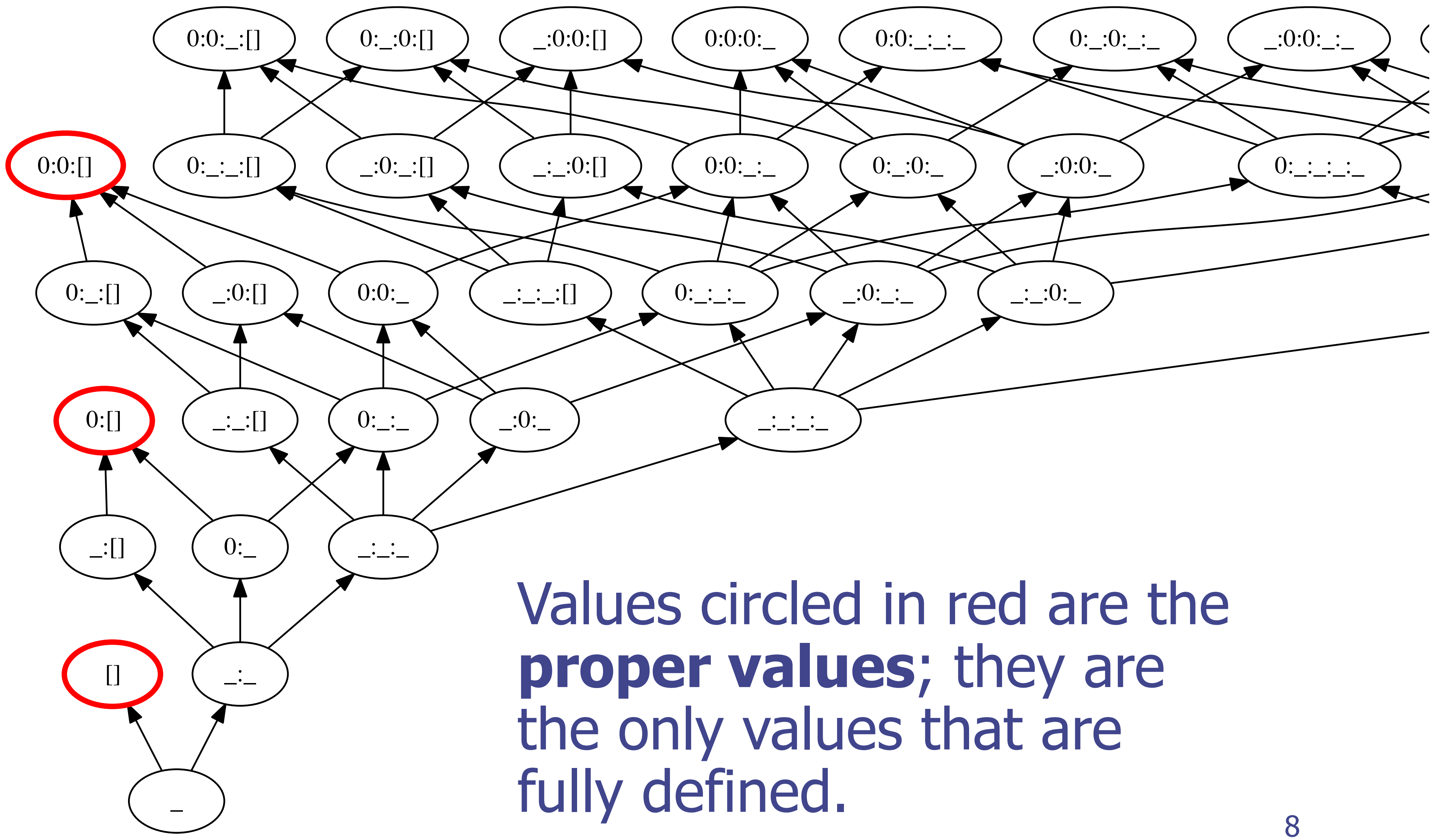
We only get answers to questions that we ask

You don’t have to ask the same question twice

Initially, we have “no information”, ☐

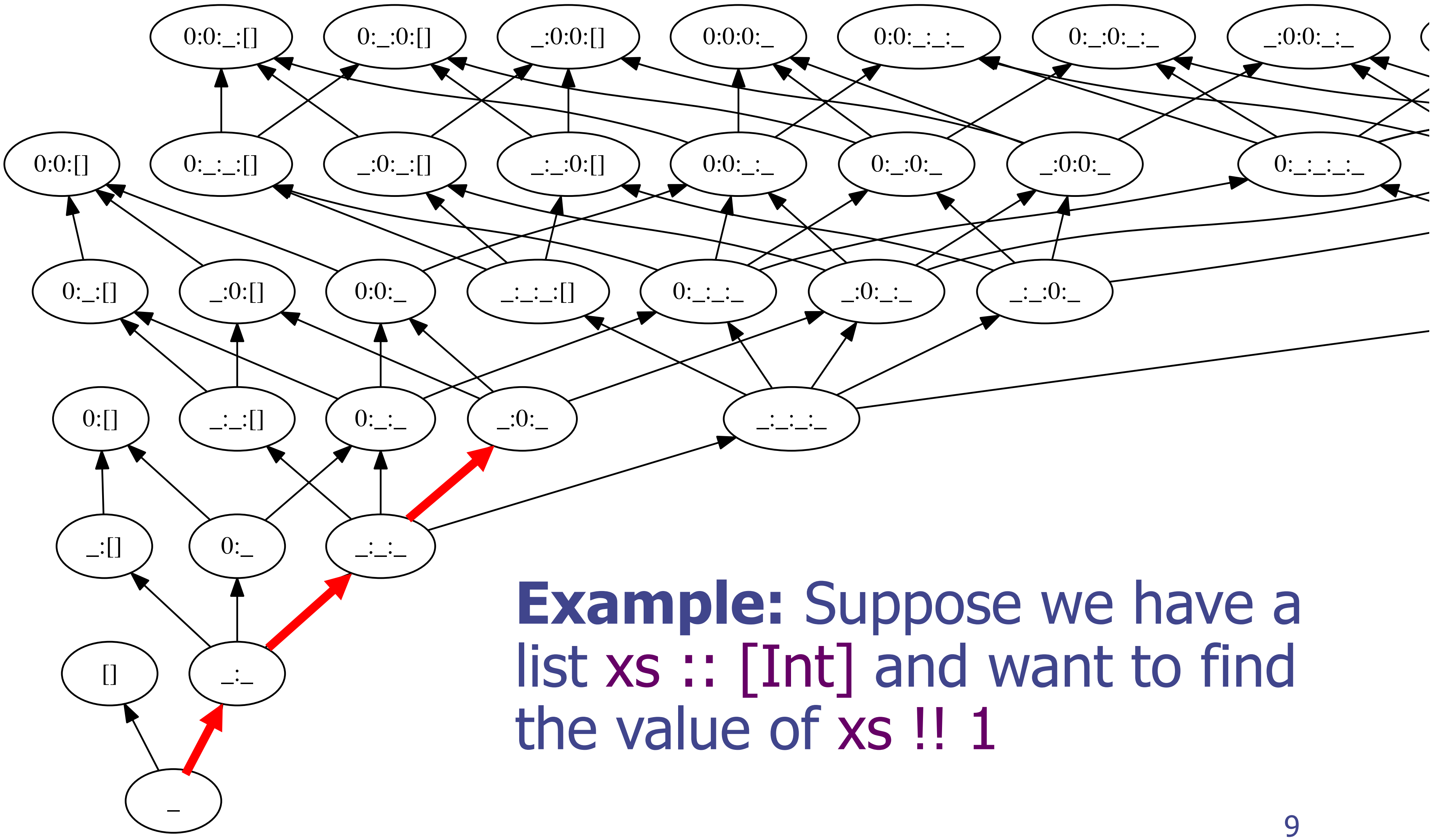
You might not need to know everything about the result

Evaluation on Demand



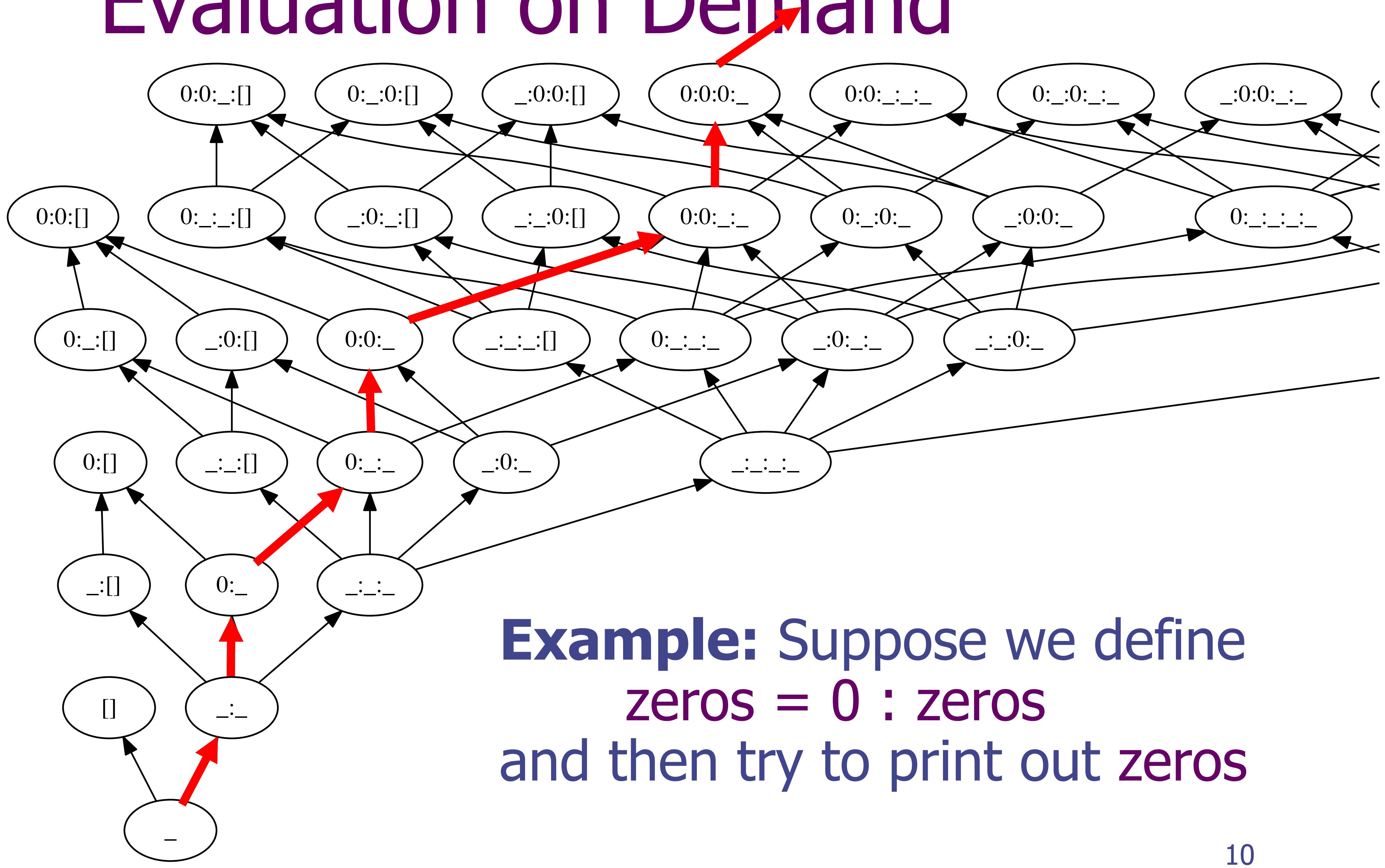
Values circled in red are the **proper values**; they are the only values that are fully defined.

Evaluation on Demand



Example: Suppose we have a list xs :: [Int] and want to find the value of xs !! 1

Evaluation on Demand



Example: Suppose we define
zeros = 0 : zeros
and then try to print out zeros

Foundational Ideas

- We're edging towards some very important ideas in the foundations of programming language semantics. (Not just functional languages!)
- Every value, even the "infinite" ones, can be described by a sequence of approximations, starting with □ and with each subsequent element being more well-defined than its predecessor
- The basic idea is not so unfamiliar:

π = 3. 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3 . . .

Why use Lazy Evaluation?

- To avoid redundant computation
- To eliminate special cases (e.g., && and ||) can be defined as regular functions:

```
True  && x = x
False && x = False
```

- To facilitate reasoning (e.g., we can be sure that (\x -> e) e′ = [e′/x] e)

Why use Lazy Evaluation?

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
```

Memoization

A more general facility that takes advantage of laziness is **memoization**

```
import Data.Vector((!),generate)

fib n = fibs ! n
    where fibs = generate (n+1) f
              f 0 = 0
              f 1 = 1
              f n = (fibs ! (n-1)) + (fibs ! (n-2))
```

Circular Programs

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

Consider a tree datatype:

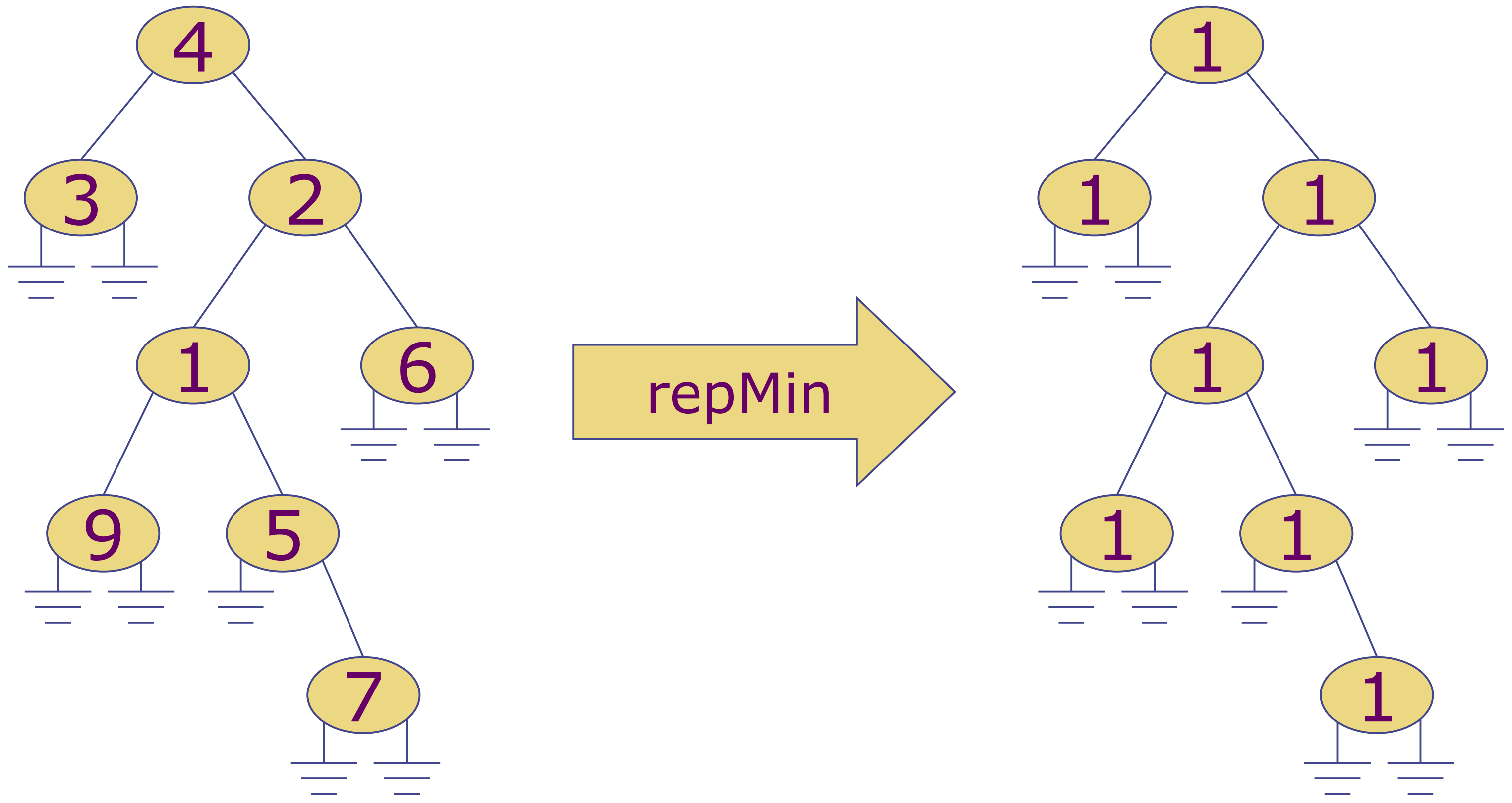
```
data Tree = Leaf | Fork Int Tree Tree
```

Define a function

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

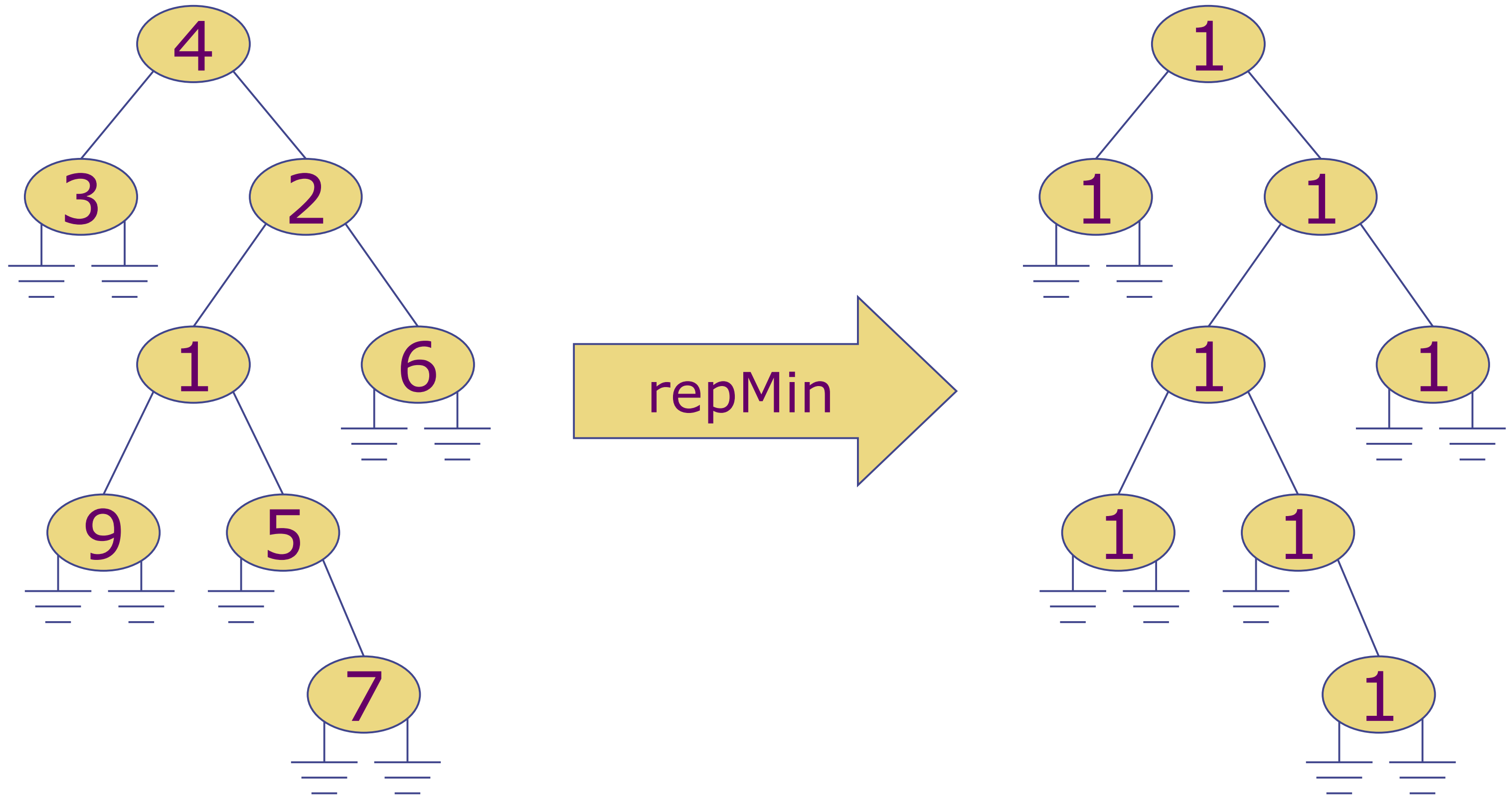
that will produce an output tree with the same shape as the input but replacing each integer with the minimum value in the original tree.

Example



Same shape, values replaced with minimum

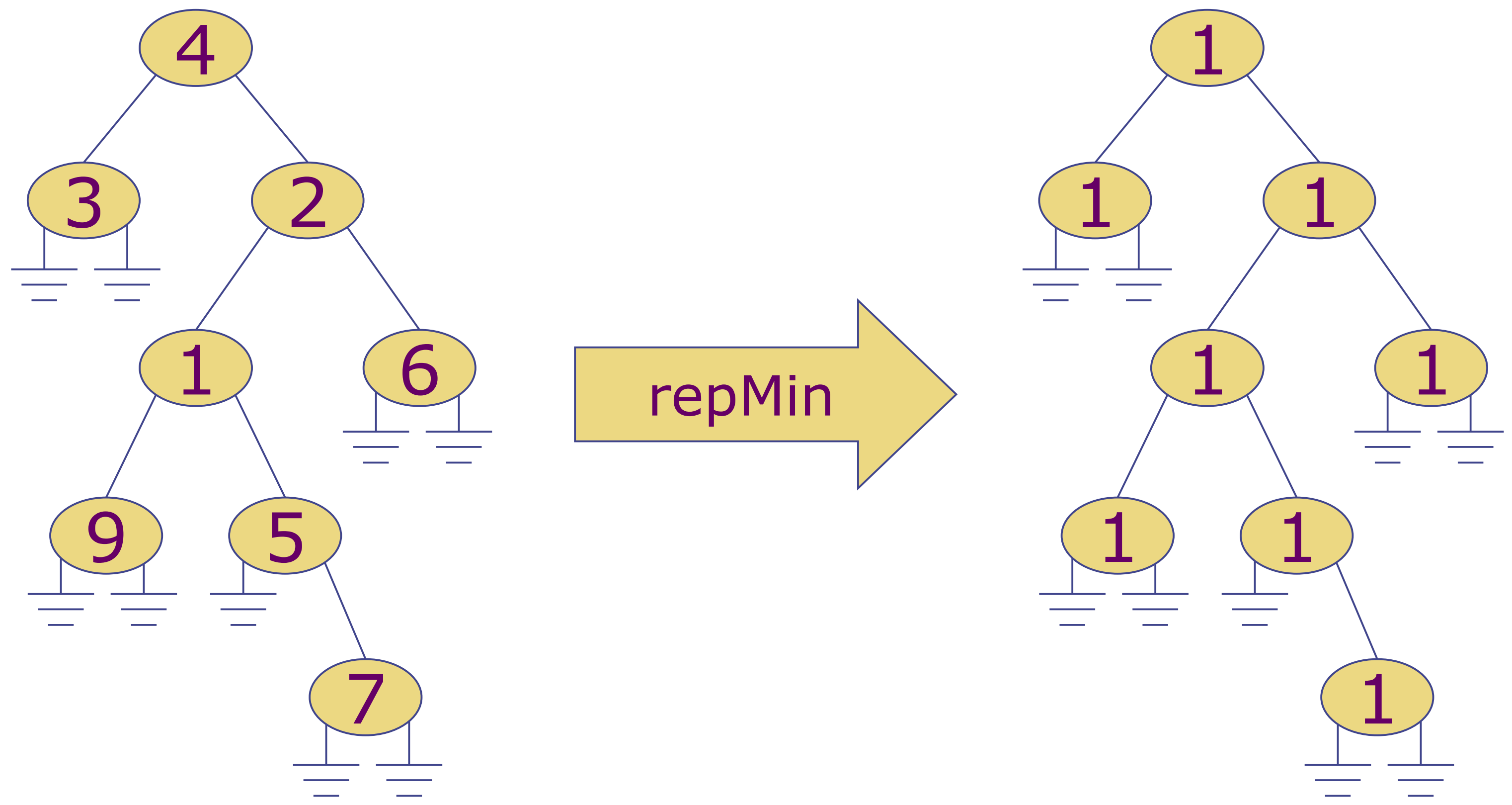
Example



Obvious implementation:

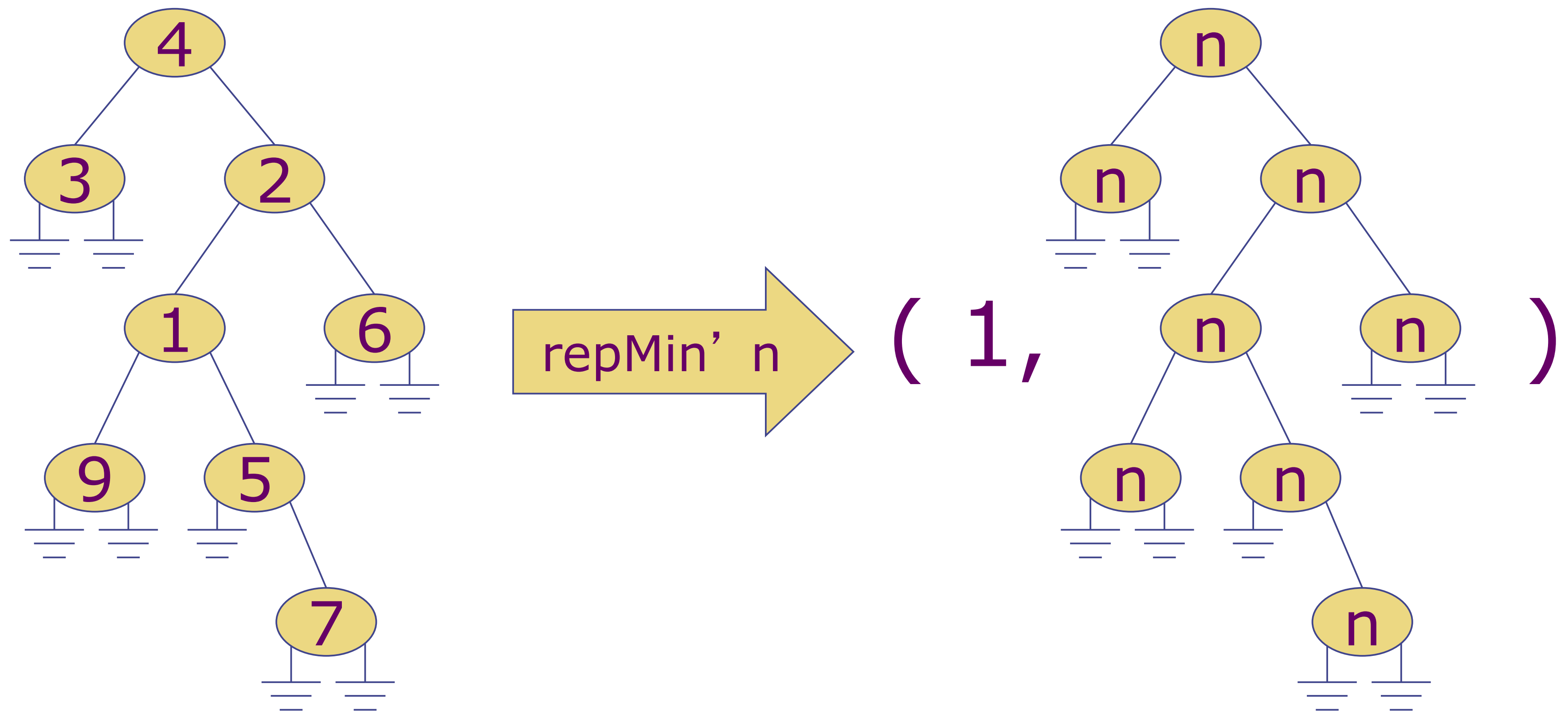
```
repMin t = mapTree (\n -> m) t
              where m = minTree t
```

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 m (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.

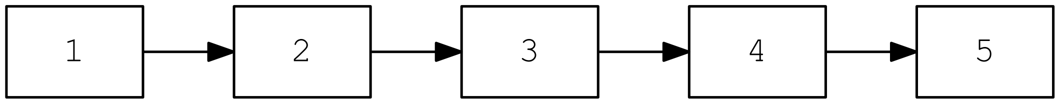
- We can implement repMin by creating a cyclic structure that passes the minimum value that is returned by repMin' as its first argument:

```
repMin t = t' where (n, t') = repMin' n t
```

Building Cyclic Data Structures

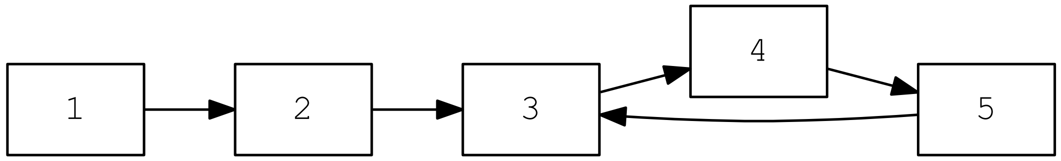
Cyclic Structures

- Haskell makes it easy to define linked structures:



```
nums = 1 : 2 : 3 : 4 : 5 : []
```

- We can even define structures with loops:

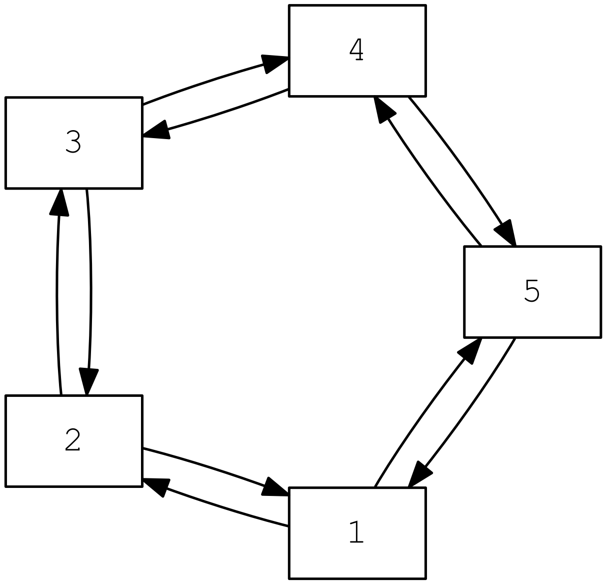


```
loopy = 1 : 2 : loop
    where loop = 3 : 4 : 5 : loop
```

- How far can we go?

Doubly Linked Structures

- Can we build a doubly linked structure?



```
ring = r1
 where
  r1 = Node r5 1 r2
  r2 = Node r1 2 r3
  r3 = Node r2 3 r4
  r4 = Node r3 4 r5
  r5 = Node r4 5 r1
```

```
data Ring a = Node (Ring a) a (Ring a)
```

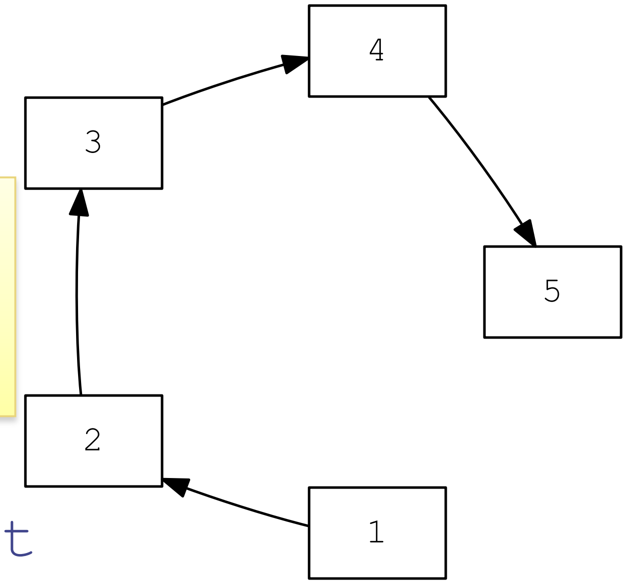
- Can we build a ring from an arbitrary list?

```
makeRing :: [a] -> Ring a
```

Making Rings, First Attempt

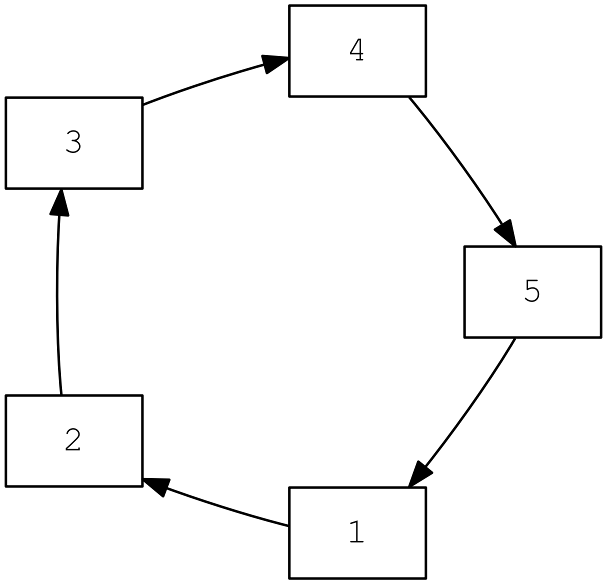
```
makeRing    :: [a] -> Ring a
makeRing xs = loop xs
 where
   loop []      = ???
   loop (x:xs) = this
     where this = Node ??? x next
           next = loop xs
```

loop back to the start…



Making Rings, Attempt II

```
makeRing    :: [a] -> Ring a
makeRing xs = start
 where
   start         = loop xs

   loop []       = start
   loop (x:xs)   = this
     where this = Node ??? x next
           next = loop xs
```

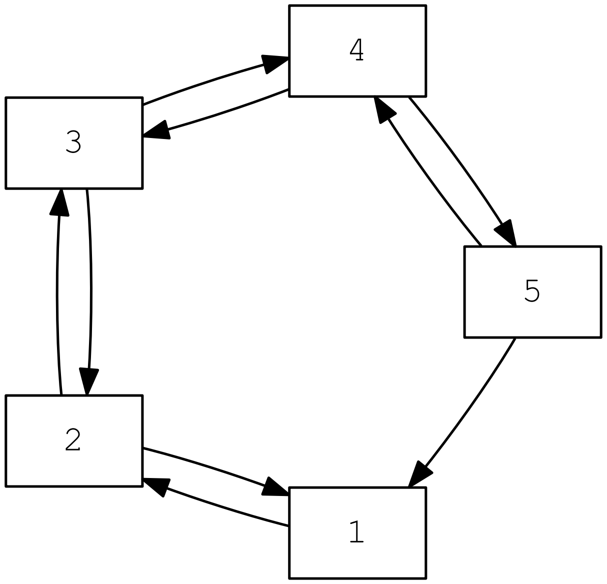


We don’t know what the predecessor should be; so ask for it to be supplied as a parameter …

Making Rings, Attempt III

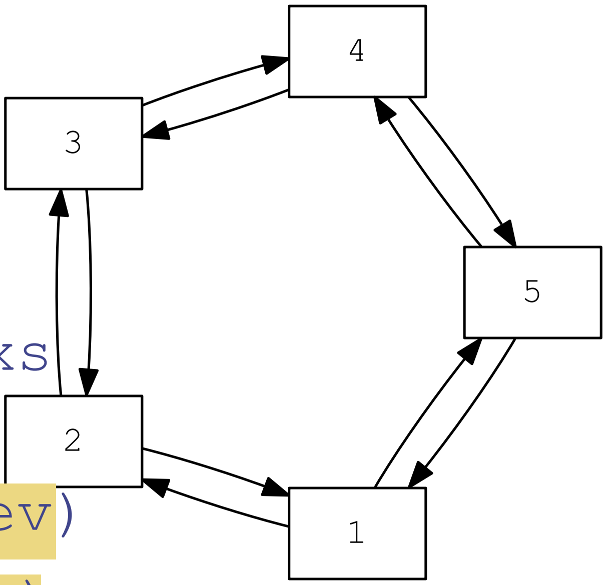
```
makeRing    :: [a] -> Ring a
makeRing xs = start
 where
   start         = loop ??? xs

   loop prev []     = start
   loop prev (x:xs) = this
     where this = Node prev x next
           next = loop this xs
```

need last node …



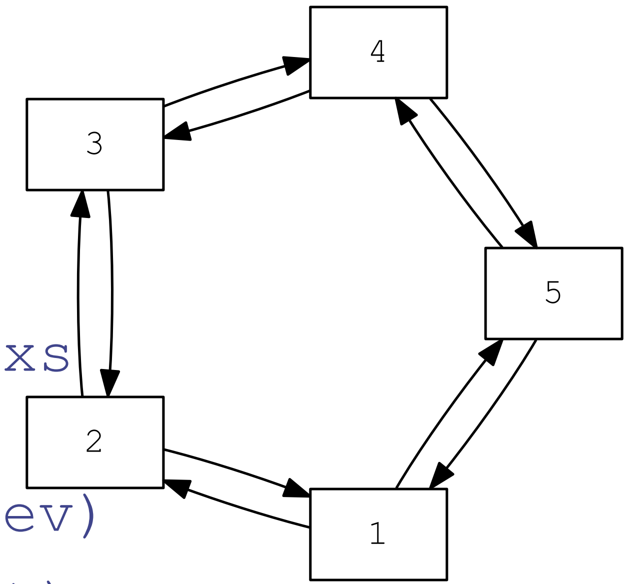
Making Rings, at last!

```
makeRing    :: [a] -> Ring a
makeRing xs = start
 where
   (start, last)     = loop last xs

   loop prev []      = (start, prev)
   loop prev (x:xs)  = (this, last)
     where this           = Node prev x next
           (next, last)   = loop this xs
```

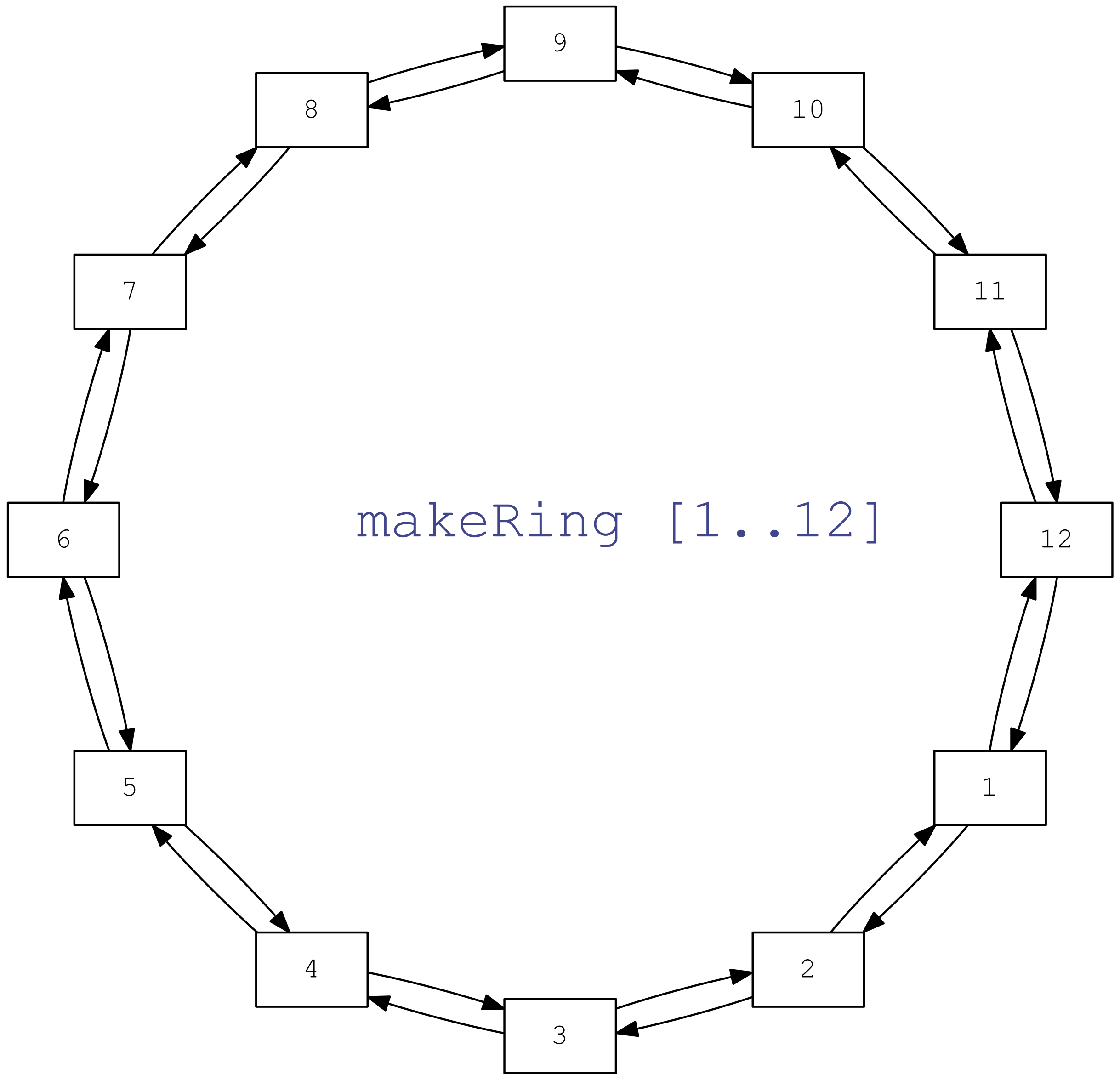


Making Rings, at last!

```
makeRing   :: [a] -> Ring a
makeRing xs = start
 where
   (start, last)    = loop last xs

   loop prev []     = (start, prev)
   loop prev (x:xs) = (this, last)
     where this             = Node prev x next
           (next, last) = loop this xs
```



Making Rings, at last!



Operations on Rings

```
next, prev          :: Ring a -> Ring a
next (Node p v n)    = n
prev (Node p v n)    = p

curr                :: Ring a -> a
curr (Node p v n)    = v

forward             :: Ring a -> [a]
forward              = map curr . iterate next

backward             :: Ring a -> [a]
backward              = map curr . iterate prev
```

In practice …

```
Main> take 10 (forward (makeRing [1..7]))
[1,2,3,4,5,6,7,1,2,3]
Main> take 10 (backward (makeRing [1..7]))
[1,7,6,5,4,3,2,1,7,6]
Main>
```

For these examples, we could have used modulo arithmetic …

But Rings are more general …

and more mindbending too! ☺

Pragmatic Aspects of Lazy Evaluation

Laziness and Performance

- Laziness delays the evaluation of expressions until their values are needed
  - In theory, this should mean that computations only do the minimum amount of work that is necessary
  - But delaying work has costs too ..
- Performance can be impacted by laziness
- … but there are tools we can use to deal with that!

Summing a list of numbers

```
mySum []      = 0
mySum (x:xs)  = x + mySum xs
```

Simple recursive definition

```
mySum [1..4]
   = 1 + mySum [2..4]
   = 1 + (2 + mySum [3..4])
   = 1 + (2 + (3 + mySum [4..4]))
   = 1 + (2 + (3 + (4 + mySum [])))
   = 1 + (2 + (3 + (4 + 0)))
   = 1 + (2 + (3 + 4))
   = 1 + (2 + 7)
   = 1 + 9
   = 10
```

Computation grows ("on the stack") until we can begin reducing the expression

How can we make this run in constant space?

In practice ...

```
Main> mySum [1..]

ERROR - Control stack overflow
Main> :set +g
Main> mySum [1..]

{{Gc:921075}}ERROR - Control stack overflow
Main>
```

Displays memory recovered after each garbage collection

Using Tail Recursion

An accumulating parameter

```
mySum1 xs             = sumLoop1 0 xs
sumLoop1 n []         = n
sumLoop1 n (x:xs)     = sumLoop1 (n+x) xs
```

Tail recursive definition

```
mySum1 [1..4]
   = sumLoop1 0 [1..4]
   = sumLoop1 1 [2..4]
   = sumLoop1 3 [3..4]
   = sumLoop1 6 [4..4]
   = sumLoop1 10 []
   = 10
```

Partial sums are collected in the accumulating parameter!

Too good to be true?

In practice ...

```
Main> :set +g
Main> mySum [1..]

{{Gc:921075}}ERROR - Control stack overflow
Main> mySum1 [1..]
{{GcSegmentation fault
ada:~/fun%
```

Laziness kicks in ☹

Here's what really happens …

```
mySum1 [1..4]
  = sumLoop1 0 [1..4]
  = sumLoop1 (0 + 1) [2..4]
  = sumLoop1 ((0 + 1) + 2) [3..4]
  = sumLoop1 (((0 + 1) + 2) + 3) [4..4]
  = sumLoop1 ((((0 + 1) + 2) + 3) + 4) []
  = ((((0 + 1) + 2) + 3) + 4)
  = (((1 + 2) + 3) + 4)
  = ((3 + 3) + 4)
  = (6 + 4)
  = 10
```

Laziness tells us: don't evaluate the argument until it is needed

Still builds a large expression before summing starts …

The expression for `mySum1 [1..]` is so large, it crashes the hugs garbage collector!

Strictness Analysis

- This example runs fine in GHC; how is that possible?
- GHC includes:
  - An advanced program analysis called “strictness analysis” that is able to determine that `sumLoop1` is strict in both arguments.
  - An advanced optimizer that is able to use this information to generate equivalent code for `sumLoop1` that evaluates the accumulating parameter as computation proceeds.
- Can we get this behavior without relying on a “sufficiently smart” compiler?

The seq operator

- Haskell includes a special primitive:

```
seq :: a -> b -> b
```

- Intuitively, `` x `seq` y `` evaluates `x` and then returns the value of `y`

```
□ `seq` y = □

x `seq` y = y,    if x≠□
```

- Technically, we cannot actually match against □ (that amounts to solving the halting problem), but we can still implement `seq` as a primitive …

Using seq to sum a list

```
mySum2 xs          = sumLoop2 0 xs
sumLoop2 n []      = n
sumLoop2 n (x:xs)  = n `seq` sumLoop2 (n+x) xs
```

Force evaluation of n before recursive call

```
mySum2 [1..4]
  = sumLoop2 0 [1..4]
  = sumLoop2 (0+1) [2..4]
  = sumLoop2 (1+2) [3..4]
  = sumLoop2 (3+3) [4..4]
  = sumLoop2 (6+4) []
  = 6+4
  = 10
```

Runs in constant space, even without strictness analysis!

In practice …

```
Main> :set +g
Main> mySum [1..]

{{Gc:921075}}ERROR - Control stack overflow
Main> mySum2 [1..]
{{Gc:986551}}{{Gc:986553}}{{Gc:986552}}{{Gc:9
86552}}{{Gc:986555}}{{Gc:986549}}{{Gc:986554}
}{{Gc:986558}}{{Gc:986555}}{{Gc:986558}}{{Gc:
986549}}{{Gc:986555}}{{Gc:986558}}{{Gc:986553
}}{{Gc:986556}}{{Gc:986551}}
{{Gc:986553}}{{Gc^C:986556}}{Interrupted!}


{{Gc:986556}}Main>
```

Confirms that we are running in "constant space"

Laziness and IO Action Quiz

```
prog1 :: IO ()
prog1  = do putStr "Type quit to stop: "
            l <- getLine
            if l=="quit"
              then putStrLn "We are done!"
              else do putStrLn l
                      prog1
```

Will this program run in constant space?

Tail recursion

Yes, assuming bounded input on each line …

Laziness and IO Action Quiz

```
prog2 :: IO Int
prog2  = do putStr "Type quit to stop: "
            l <- getLine
            if l=="quit"
              then do putStrLn "We are done!"
                      return 0
              else do putStrLn l
                      n <- prog2
                      return (n+1)
```

Returns number of lines read

What about this version?

No tail recursion: each call to prog2 will create deeper nesting

Laziness and IO Action Quiz

```
prog3  :: Int -> IO Int
prog3 n = do putStr "Type quit to stop: "
             l <- getLine
             if l=="quit"
               then do putStrLn "We are done!"
                       return n
               else do putStrLn l
                       prog3 (n+1)
```

Accumulating parameter

Tail recursion

Will this program run in constant space?

Depends on the compiler …

Laziness and IO Action Quiz

```
prog4  :: Int -> IO Int
prog4 n = do putStr "Type quit to stop: "
             l <- getLine
             if l=="quit"
               then do putStrLn "We are done!"
                       return n
               else do putStrLn l
                       n `seq` prog4 (n+1)
```

Will this program run in constant space?

Yes!

Forces evaluation of accumulating parameter

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 …
- Subtle interactions with performance …
- Further Reading:
  - Programming in Haskell, Graham Hutton, Chapter 15
  - 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