Efficiency

Andreas Lochbihler
Department of Computer Science
ETH Zurich

Efficiency

- Up to now: focus on elegant, abstract code
- Now: An introduction to programming with efficiency in mind.
- How do we measure efficiency?

Time

- Number of evaluation steps $\mathcal{O}(n^2 \cdot m)$
- Execution time of compiled program 0.34 ms on computer X

Space

- Size of the largest expression during execution
- Amount of memory the computation requires

Strictness

- A function `f :: a -> b` is **strict** if it evaluates its argument. I.e., `f` propagates nontermination and exceptions.

```
neverFalse x = (x == x)   strict
```

```
? neverFalse divZero
*** Exception: divide by zero
```

```
alwaysTrue _ = True   lazy
```

```
? alwaysTrue divZero
True
```

- **Strict function application** `$!` evaluates the argument first.

```
? alwaysTrue $! divZero
*** Exception: divide by zero
```

```
-- divZero = 1 `div` 0
```

- `$!` evalutes the argument only to the **first constructor** or abstraction `\x -> _`. Sub-terms may still be unevaluated.

```
? alwaysTrue $! [divZero]
True
```

```
-- $! [divZero] ⇝ (divZero : [])
```

Strict left folds

```
foldl f z [] = z
foldl f z (x:xs) =
  foldl f (f z x) xs

maximum (x:xs) = foldl max x xs

maximum [1,2,3]
 = foldl max 1 [2,3]
 = foldl max (max 1 2) [3]
 = foldl max (max (max 1 2) 3) []
 = max (max 1 2) 3
 = max 2 3
 = 3

? maximum [0..20000000]
*** Exception: stack overflow
```

- Builds up huge unevaluated term.
- `max :: Int->Int->Int` is strict, so should be evaluated eagerly.

```
foldl' f z [] = z      -- defined in
foldl' f z (x:xs) =    -- Data.List
  (foldl' f $! f z x) xs

maximum' (x:xs) = foldl' max x xs

maximum' [1,2,3]
 = foldl' max 1 [2,3]
 = (foldl' max $! max 1 2) [3]
 = foldl' max 2 [3]
 = (foldl' max $! max 2 3) []
 = foldl' max 3 []
 = 3

? maximum' [0..20000000]
20000000
```

- Evaluates intermediate terms immediat.
- May fail if `f` is not strict enough. Examples?

Left folds vs. right folds: strict

```
foldl' f z [] = z
foldl' f z (x:xs) =
  (foldl' f $! f z x) xs
```

```
foldr f z [] = z
foldr f z (x:xs) =
  f x (foldr f z xs)
```

Rule of thumb: If `f` is strict, prefer `foldl'` if possible.

```
length = foldl' g 0
  where g l _ = l + 1
```

```
length [4,5,6]
 = foldl' g 0 [4,5,6]
 = (foldl' g $! g 0 4) [5,6]
 = foldl' g 1 [5,6]
 = (foldl' g $! g 1 5) [6]
 = foldl' g 2 [6]
 = (foldl' g $! g 2 6) []
 = foldl' g 3 []
 = 3
```

```
length = foldr h 0
  where h _ l -> 1 + l
```

```
length [4,5,6]
 = foldr h 0 [4,5,6]
 = 1 + (foldr h 0 [5,6])
 = 1 + (1 + (foldr h 0 [6]))
 = 1 + (1 + (1 + (foldr h 0 [])))
 = 1 + (1 + (1 + 0))
 = 1 + (1 + 1)
 = 1 + 2
 = 3
```

danger of stack overflow

Left folds vs. right folds: lazy

```
foldl' f z [] = z
foldl' f z (x:xs) =
  (foldl' f $! f z x) xs
```

```
foldr f z [] = z
foldr f z (x:xs) =
  f x (foldr f z xs)
```

Rule of thumb: If `f` is lazy in the second argument, prefer `foldr`.

```
and = foldl' (&&) True

and [True,False,True]
 = foldl' (&&) True [True,False,True]
 = (foldl' (&&) $! (True && True)) [False,True]
 = foldl' (&&) True [False,True]
 = (foldl' (&&) $! (True && False)) [True]
 = foldl' (&&) False [True]
 = (foldl' (&&) $! (False && True)) []
 = False
```

```
and = foldr (&&) True

and [True,False,True]
 = foldr (&&) True [True,False,True]
 = True && foldr (&&) True [False,True]
 = foldr (&&) True [False,True]
 = False && foldr (&&) True [True]
 = False
```

Laziness allows to immediately execute &&.

Left folds always process the whole list; right folds may stop early.

Left folds vs. right folds: infinite lists

- Strict left folds produce complete result, but only at the end.
  They are **strict in the list structure.**
- Right folds produce result **incrementally**.
  They can work with laziness and infinite lists.

```
aBs = "AB" : aBs                                -- aBs = ["AB", "AB", ...]

take 3 (concat aBs)                             -- concat = foldl (++) []
 = take 3 (foldl (++) [] aBs)
 = take 3 (foldl (++) ([] ++ "AB") aBs)
 = take 3 (foldl (++) (([] ++ "AB") ++ "AB") aBs)
 = ...   nontermination

take 3 (concat aBs)                             -- concat = foldr (++) []
 = take 3 (foldr (++) [] aBs)
 = take 3 ("AB" ++ foldr (++) [] aBs)
 = 'A' : 'B' : take 1 (foldr (++) [] aBs)
 = 'A' : 'B' : take 1 ("AB" ++ foldr (++) [] aBs)
 = 'A' : 'B' : 'A' : []
```

Measuring time

What is wrong with the following?

```
import System.CPUTime                          -- getCPUTime :: IO Integer

main = do
  start <- getCPUTime

  let bs = sum [1..2000000]                    -- bs is never used
  _ <- return $! sum [1..2000000]              -- $! works for primitive values
  stop <- getCPUTime
  putStrLn (show ((stop - start) `div` 10^9))  -- in milliseconds

? main
584
```

Always compile with ghc. Do not measure in ghci.

```
> ghc -O timeSum.hs
> ./timeSum
48
```

-O tells ghc to optimize.

Measuring space

- Garbage collector reclaims memory only when program runs out of memory.
- Limit heap size with runtime option `+RTS -Msize` to force GC.
- Find threshold for which program no longer runs out of memory.

```
maximum (x:xs) = foldl max x xs
main = putStrLn (show (maximum [1..20000]))
```

```
> ghc -rtsopts space.hs
> ./space +RTS -M1255k
Heap exhausted;
> ./space +RTS -M1256k
20000
```

```
maximum' (x:xs) = foldl' max x xs
main = putStrLn (show (maximum' [1..20000]))
```

```
> ghc -rtsopts space'.hs
> ./space' +RTS -M575k
Heap exhausted;
> ./space' +RTS -M576k
20000
```

Tail-recursion

A function `f` is **tail-recursive**
if the recursive call is the last action in every recursive equation.

Examples:

```
foldl' f z [] = []
foldl' f z (x:xs) = (foldl' f $! f z x) xs   -- tail-recursive

foldr f z [] = []
foldr f z (x:xs) = f x (foldr f z xs)        -- not tail-recursive

gcd x y
  | x == y    = x
  | x > y     = gcd (x - y) y                -- tail-recursive
  | otherwise = gcd x (y - x)                -- tail-recursive
```

Tail-recursive functions can be compiled into **loops**. At recursive call,

- overwrite old parameters with new ones,
- then jump back to start of function code.

prevents stack overflows

Accumulators

Accumulators are additional function parameters that store intermediate results. Accumulators are usually strict (`$!`).

They can be used to make a function tail-recursive.
Order of composition and association is reversed.

```
fac 0 = 1
fac n = n * fac (n - 1)
-- n*((n-1)*(...*1*1)...)
```

```
length []     = 0
length (x:xs) = 1+length xs
-- 1+(1+...(1+(1+0))...)
```

```
rev []     = []
rev (x:xs) = rev xs ++ [x]
```

```
fac2 = aux 1  -- (...((1*n)*(n-1))*...)*1
  where
    aux acc 0 = acc
    aux acc n = (aux $! acc * n) (n - 1)
```

```
length = aux 0  -- (...((0+1)+1)+...)+1
  where
    aux acc []     = acc
    aux acc (x:xs) = (aux $! acc + 1) xs
```

```
qrev = aux []
  where
    aux acc []     = acc
    aux acc (x:xs) = aux (x:acc) xs
```

No need for ($!). Why?

Accumulators (cont.)

Use accumulator to avoid **repeated concatenation**:

```
data Tree t = Leaf | Node t (Tree t) (Tree t)

inorder :: Tree t -> [t]
inorder Leaf          = []
inorder (Node x l r) = inorder l ++ ([x] ++ inorder r)

inorder t = go t []
  where go :: Tree t -> [t] -> [t]
        go Leaf          acc = acc
        go (Node x l r) acc = go l ([x] ++ go r acc)
```

We have seen this before: difference lists `type DList t = [t] -> [t]`

```
                                  -- toList xs   = xs []
inorder t = toList (go t)         -- empty       = \xs -> xs
  where                           -- sngl x      = \xs -> x : xs
    go :: Tree t -> DList t       -- ys `app` zs = \xs -> ys (zs xs)
    go Leaf          = empty
    go (Node x l r) = go l `app` (sngl x `app` go r)
```

Tail recursion is not lazy

- Tail recursive functions return complete result upon termination, i.e., no intermediate results and no laziness (cf. `foldl`).
- Use tail recursion for strict functions, not everywhere!
- Example: **Don't do `map` tail-recursively!**

```
map f []     = []
map f (x:xs) = f x : map f xs
```

not tail-recursive, but lazy and $\mathcal{O}(n)$

```
map f = go []
  where
    go acc []     = acc
    go acc (x:xs) = go (acc ++ [f x]) xs
```

tail-recursive, but not lazy and $\mathcal{O}(n^2)$ repeated concatenation

```
map f = go []
  where
    go acc []     = reverse acc
    go acc (x:xs) = go (f x : acc) xs
```

tail-recursive and $\mathcal{O}(n)$, but not lazy and traverses the list twice

Sharing

```
isPrime p = trace "p" ([x | x <- [1 .. p], p `mod` x == 0] == [1,p])
```

```
silly m n
  | isPrime m && m < n  = n * m
  | isPrime m && even n = n + m
  | otherwise           = n - m
```

```
silly' m n
  | primeM && m < n  = n * m
  | primeM && even n = n + m
  | otherwise        = n - m
  where primeM = isPrime m
```

How often is `isPrime` called?

```
? silly 101 10
p
p
111
```

twice

```
? silly' 101 10
p
111
```

once

- GHC does **not** eliminate redundant recomputations, just in some cases if run with `-O`.
- Manually factor out common subexpressions with `let` and `where`.

Drawbacks of sharing

```
sublists [] = [[]]
sublists (x:xs) = sublists xs ++ map (x:) (sublists xs)

sublists' [] = [[]]
sublists' (x:xs) = ys ++ map (x:) ys
  where ys = sublists' xs
```

Memory consumption of `length (sublists [1..n])` in kB:

n	10	11	12	13	14	15	20
sublists	164k	164k	172k	172k	172k	228k	228k
sublists'	164k	172k	220k	352k	568k	1048k	29172k

but `sublists'` is twice as fast as `sublists`.

`ys` must be kept in memory while `length` traverses ys.

Visualizing sharing

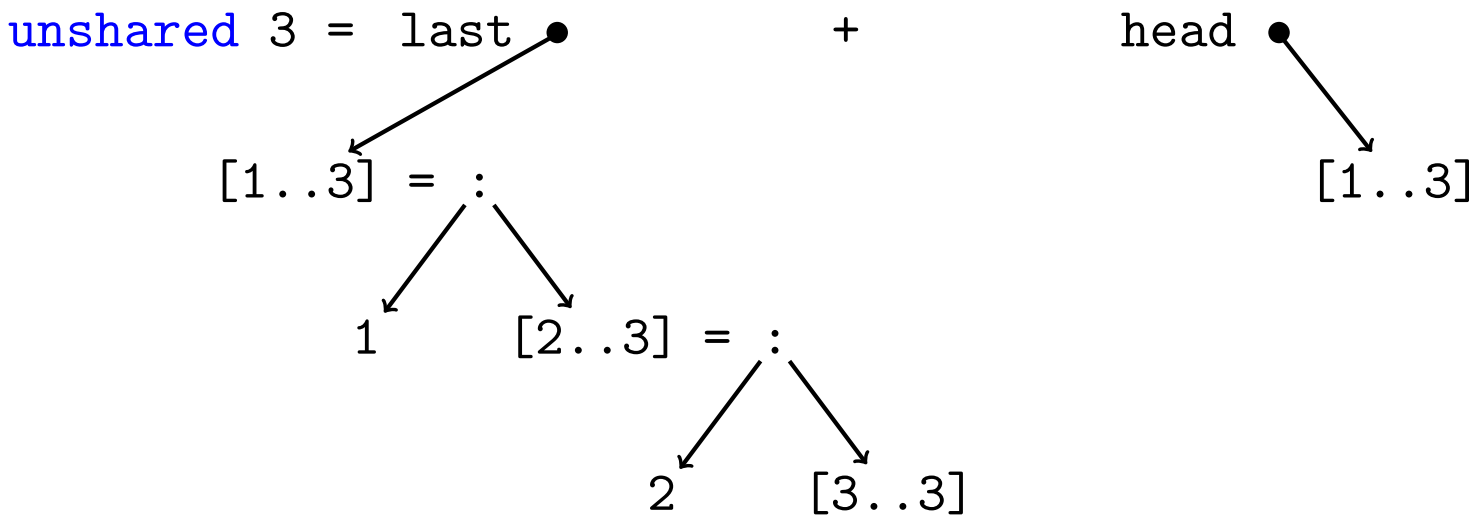
```
unshared n = last [1..n] + head [1..n]
shared   n = let xs = [1..n] in last xs + head xs

last (x:[]) = x                          head (x:xs) = x
last (x:xs) = last xs
```

memory freed by
the garbage collector

Memory consumption of `unshared`: $\mathcal{O}(1)$

Visualizing sharing

```
unshared n = last [1..n] + head [1..n]
shared   n = let xs = [1..n] in last xs + head xs

last (x:[]) = x                          head (x:xs) = x
last (x:xs) = last xs
```

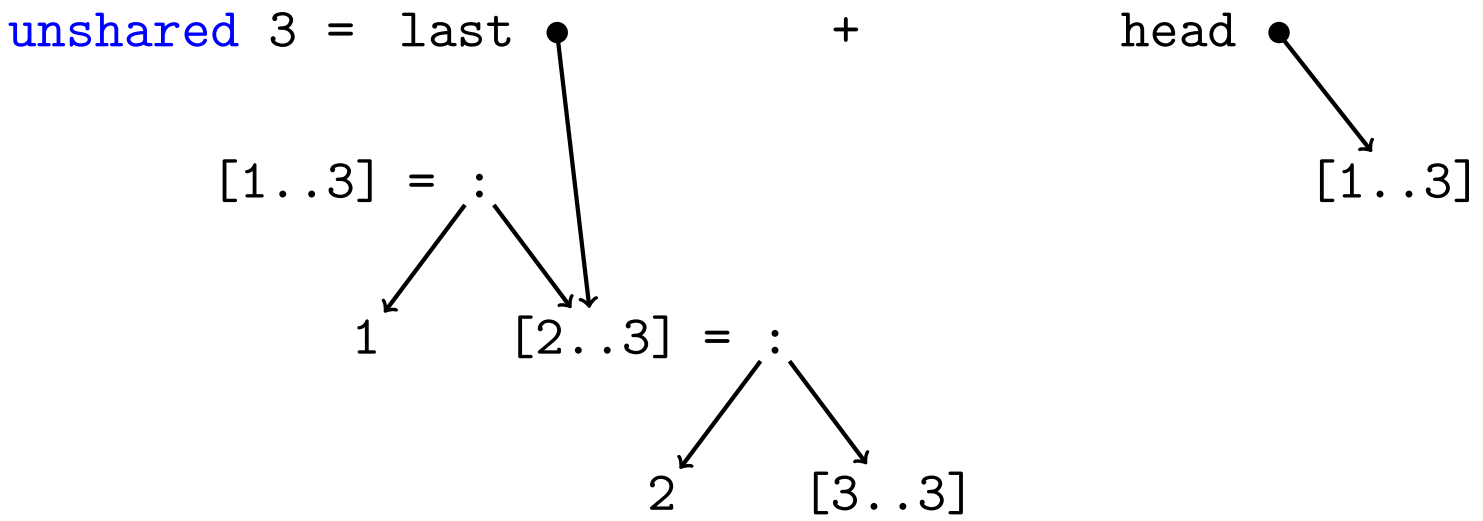


memory freed by
the garbage collector

Memory consumption of `unshared`: $\mathcal{O}(1)$

Visualizing sharing

```
unshared n = last [1..n] + head [1..n]
shared   n = let xs = [1..n] in last xs + head xs

last (x:[]) = x                          head (x:xs) = x
last (x:xs) = last xs
```

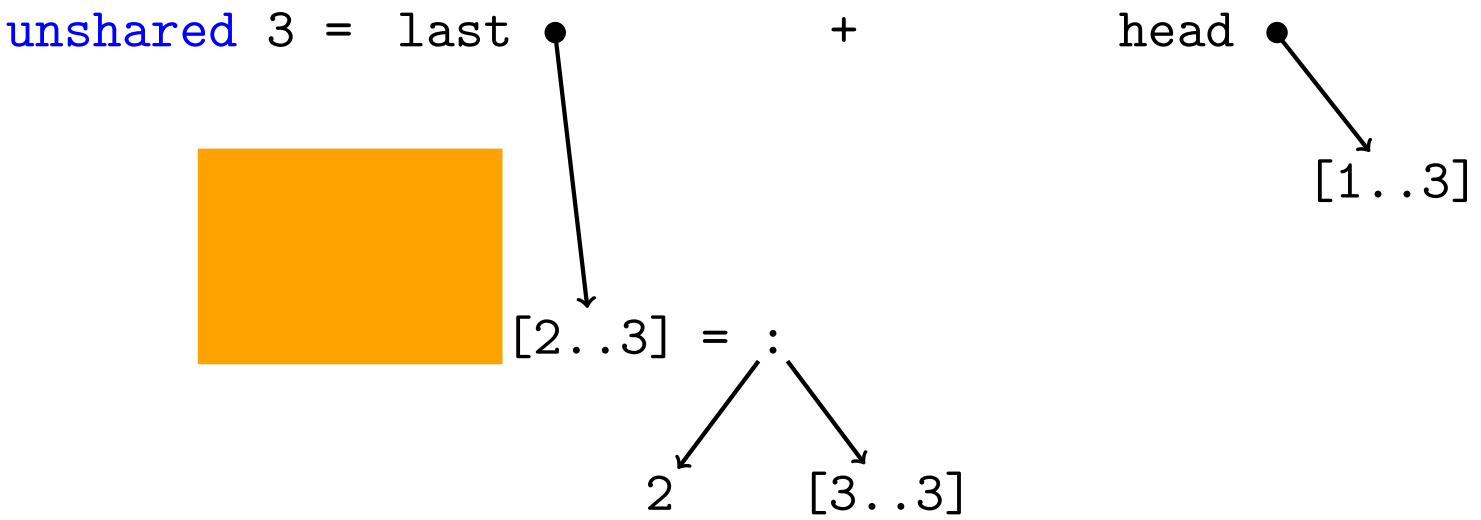


memory freed by
the garbage collector

Memory consumption of `unshared`: $\mathcal{O}(1)$

Visualizing sharing

```
unshared n = last [1..n] + head [1..n]
shared   n = let xs = [1..n] in last xs + head xs

last (x:[]) = x                          head (x:xs) = x
last (x:xs) = last xs
```

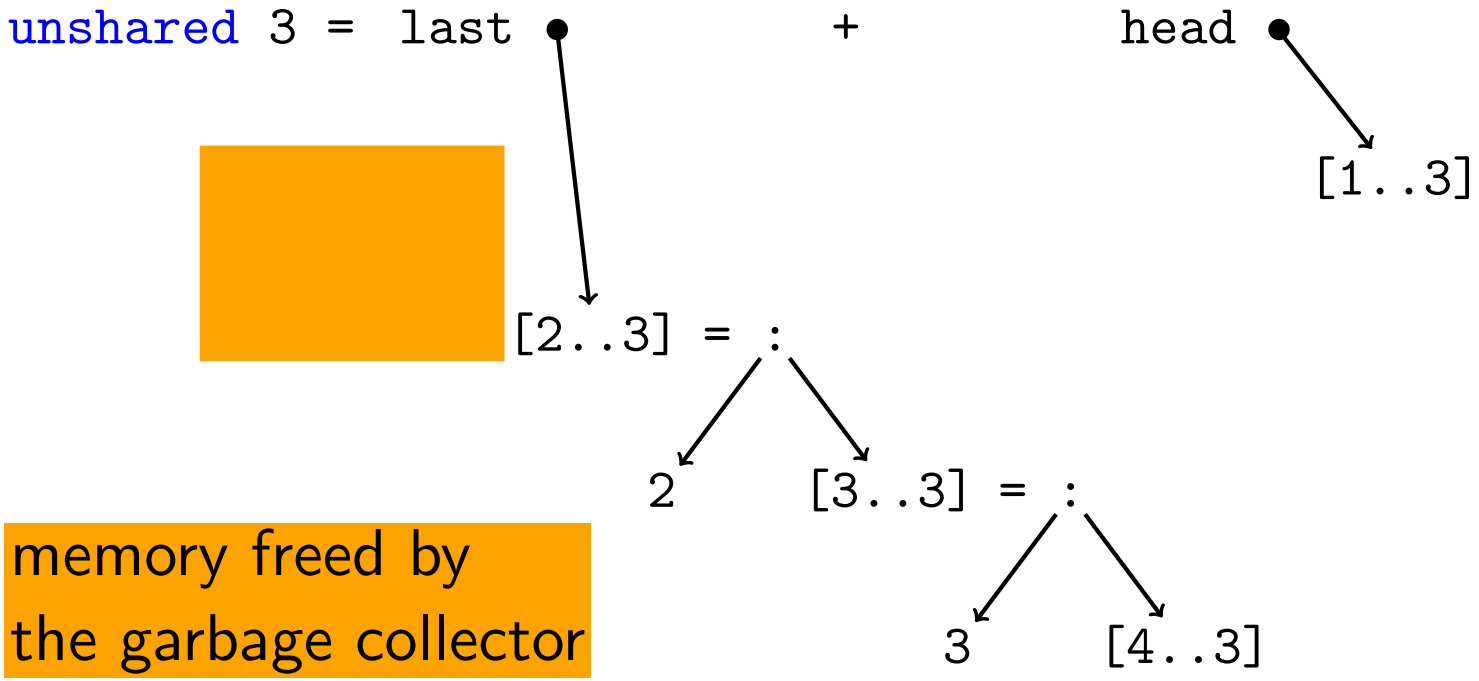


memory freed by
the garbage collector

Memory consumption of `unshared`: $\mathcal{O}(1)$

Visualizing sharing

```
unshared n = last [1..n] + head [1..n]
shared   n = let xs = [1..n] in last xs + head xs

last (x:[]) = x                          head (x:xs) = x
last (x:xs) = last xs
```

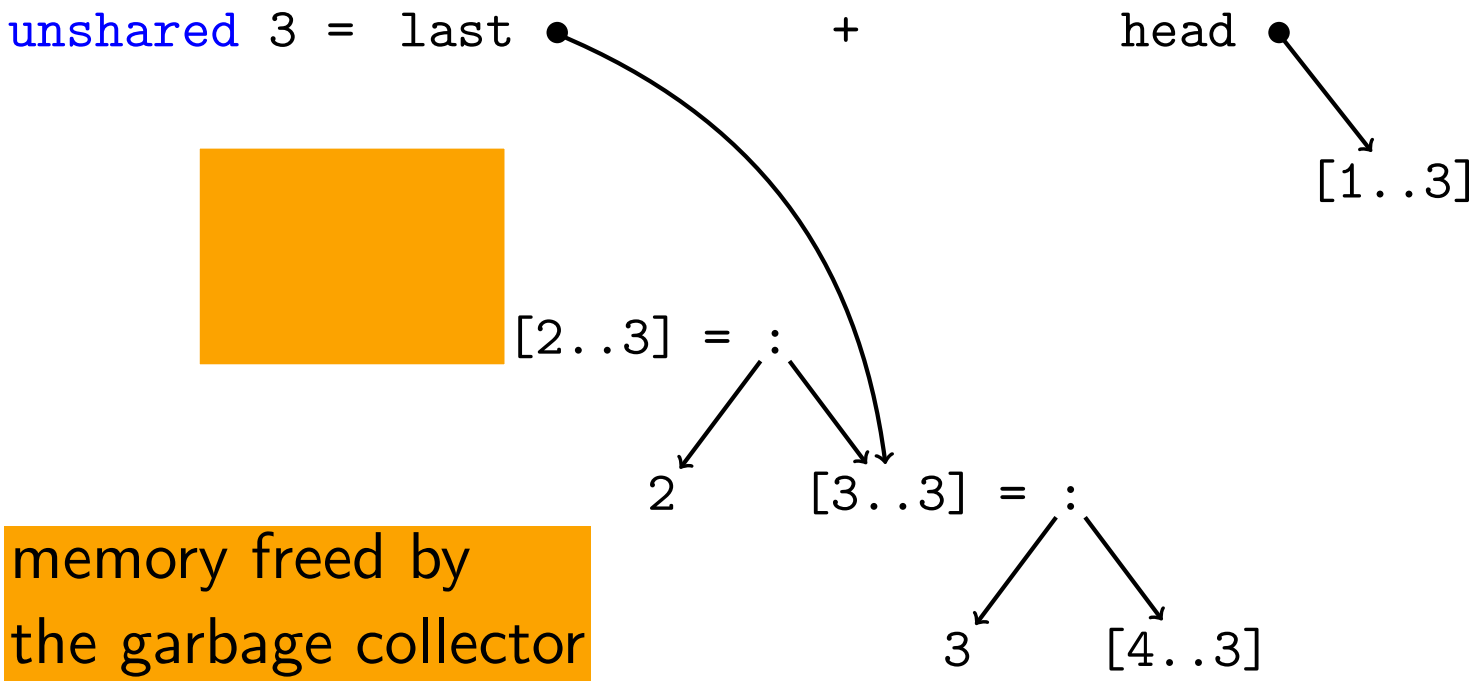


memory freed by
the garbage collector

Memory consumption of `unshared`: $\mathcal{O}(1)$

Visualizing sharing

```
unshared n = last [1..n] + head [1..n]
shared   n = let xs = [1..n] in last xs + head xs

last (x:[]) = x                          head (x:xs) = x
last (x:xs) = last xs
```

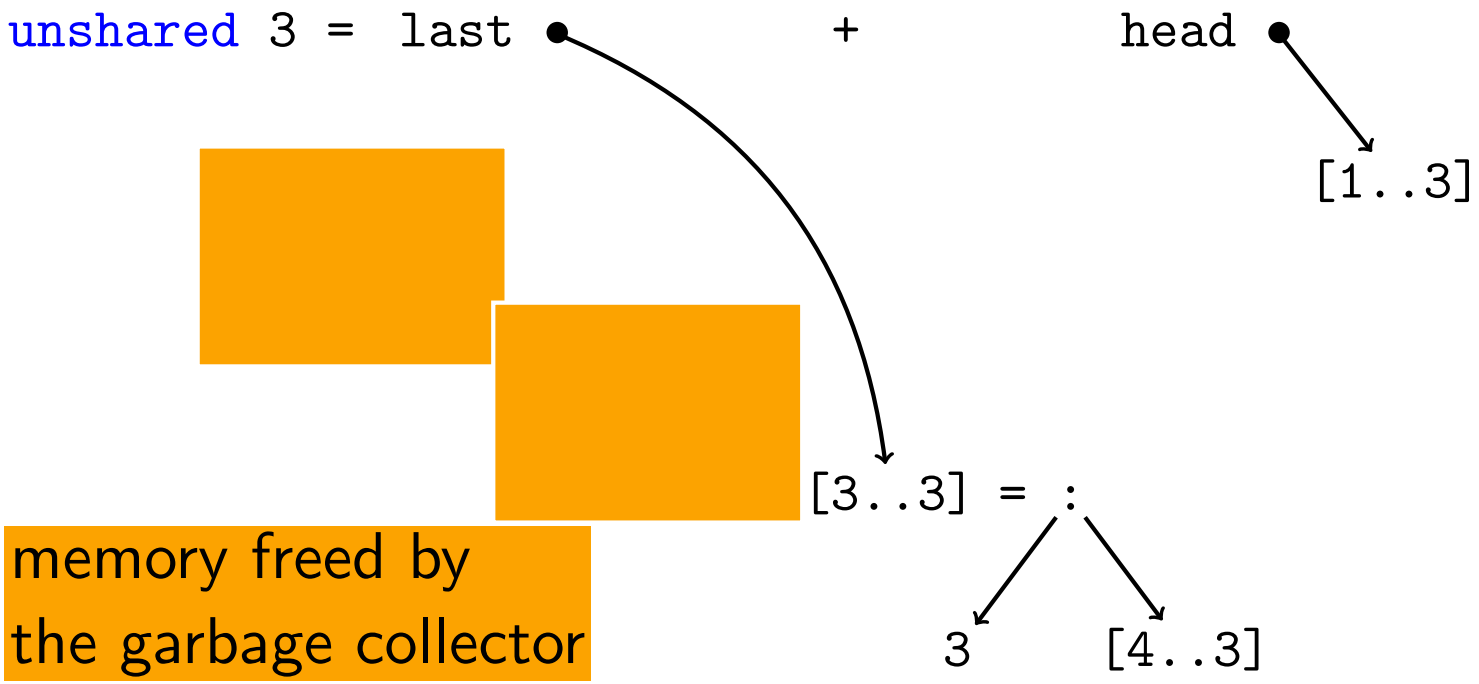


memory freed by
the garbage collector

Memory consumption of `unshared`: $\mathcal{O}(1)$

Visualizing sharing

```
unshared n = last [1..n] + head [1..n]
shared   n = let xs = [1..n] in last xs + head xs

last (x:[]) = x                          head (x:xs) = x
last (x:xs) = last xs
```

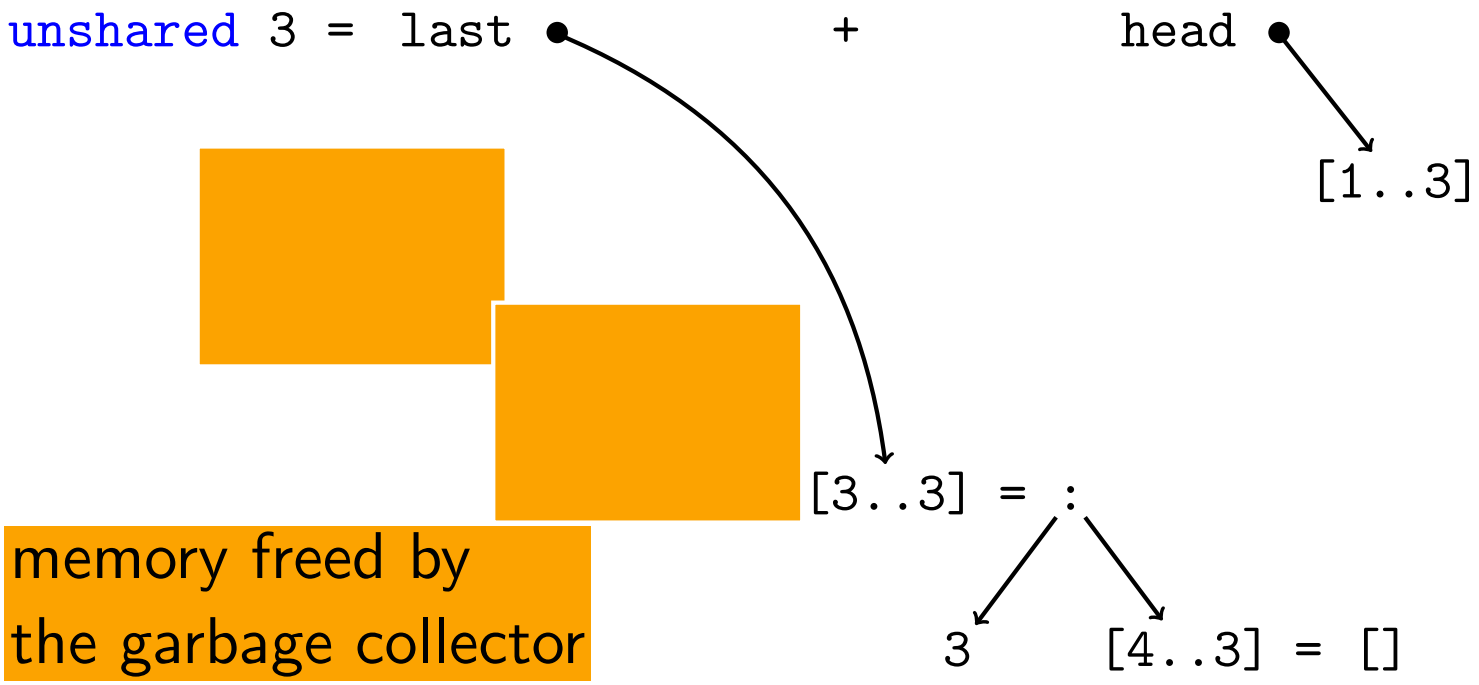


memory freed by
the garbage collector

Memory consumption of `unshared`: $\mathcal{O}(1)$

Visualizing sharing

```
unshared n = last [1..n] + head [1..n]
shared   n = let xs = [1..n] in last xs + head xs

last (x:[]) = x                          head (x:xs) = x
last (x:xs) = last xs
```

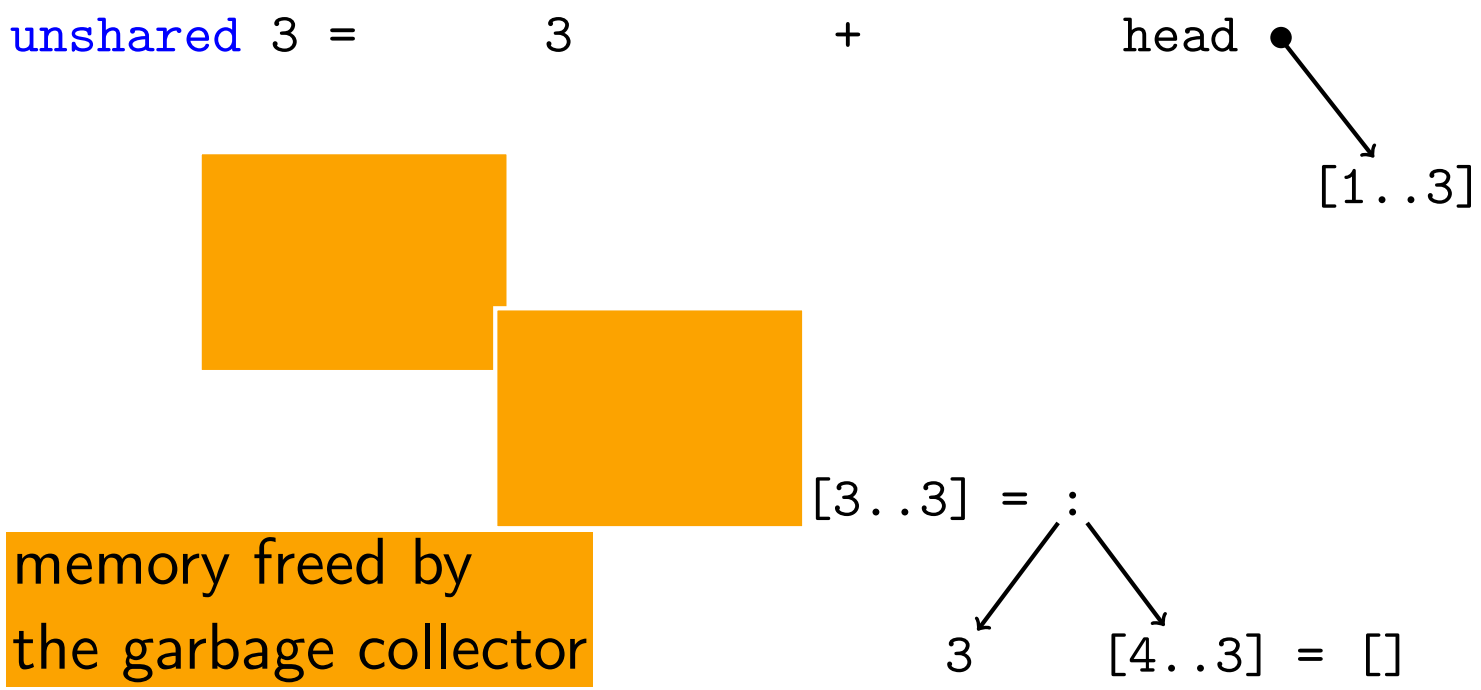


memory freed by
the garbage collector

Memory consumption of `unshared`: $\mathcal{O}(1)$

Visualizing sharing

```
unshared n = last [1..n] + head [1..n]
shared   n = let xs = [1..n] in last xs + head xs

last (x:[]) = x                          head (x:xs) = x
last (x:xs) = last xs
```



memory freed by
the garbage collector

Memory consumption of `unshared`: $\mathcal{O}(1)$

Visualizing sharing

```
unshared n = last [1..n] + head [1..n]
shared   n = let xs = [1..n] in last xs + head xs

last (x:[]) = x                          head (x:xs) = x
last (x:xs) = last xs
```

```
unshared 3 =       3          +          head ●
                                           [1..3]
```

memory freed by
the garbage collector

Memory consumption of `unshared`: $\mathcal{O}(1)$

Visualizing sharing

```
unshared n = last [1..n] + head [1..n]
shared   n = let xs = [1..n] in last xs + head xs

last (x:[]) = x                    head (x:xs) = x
last (x:xs) = last xs
```

```
unshared 3 =      3          +          head
```

evaluate [1..3] once more

```
[1..3] = :
        1   [2..3]
```

memory freed by the garbage collector

Memory consumption of `unshared`: $\mathcal{O}(1)$

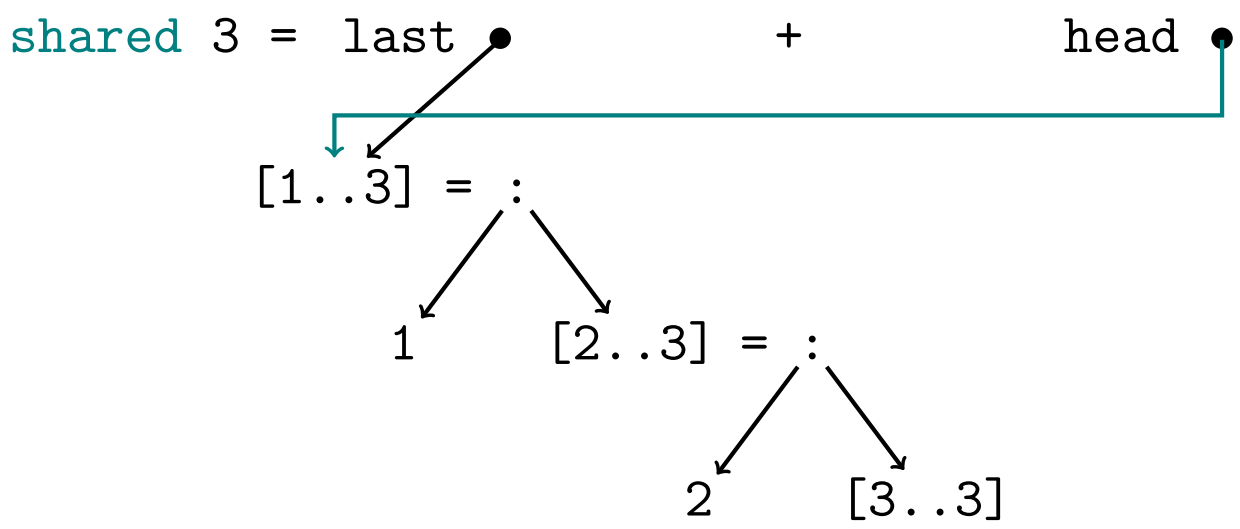
Visualizing sharing

```
unshared n = last [1..n] + head [1..n]
shared   n = let xs = [1..n] in last xs + head xs

last (x:[]) = x                          head (x:xs) = x
last (x:xs) = last xs
```

Memory consumption of `unshared`: $\mathcal{O}(1)$ shared: $\mathcal{O}(n)$

Visualizing sharing

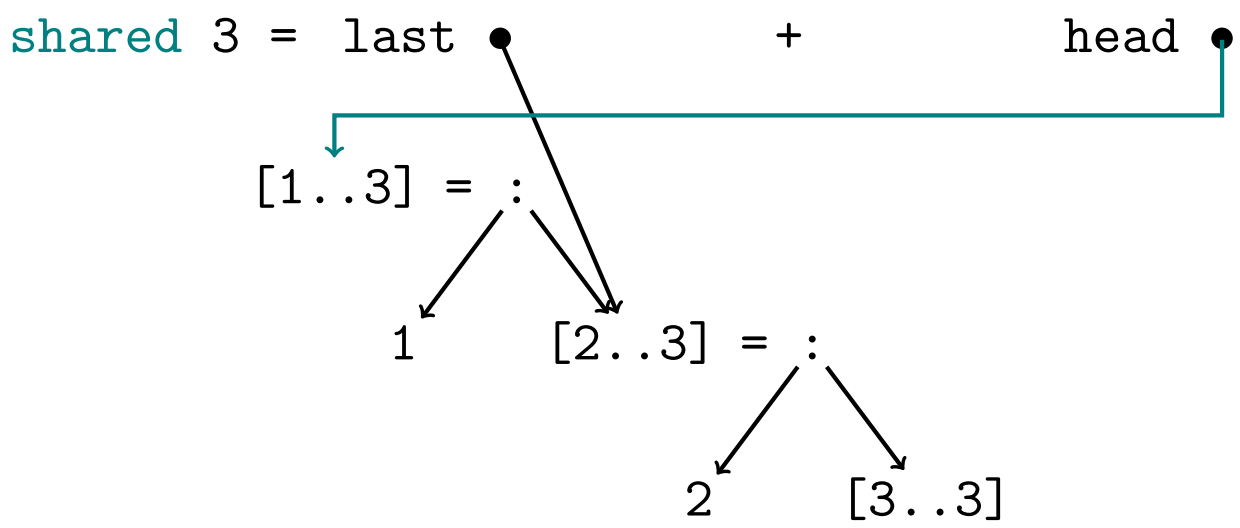
```
unshared n = last [1..n] + head [1..n]
shared   n = let xs = [1..n] in last xs + head xs

last (x:[]) = x                          head (x:xs) = x
last (x:xs) = last xs
```



Memory consumption of `unshared`: $\mathcal{O}(1)$ shared: $\mathcal{O}(n)$

Visualizing sharing

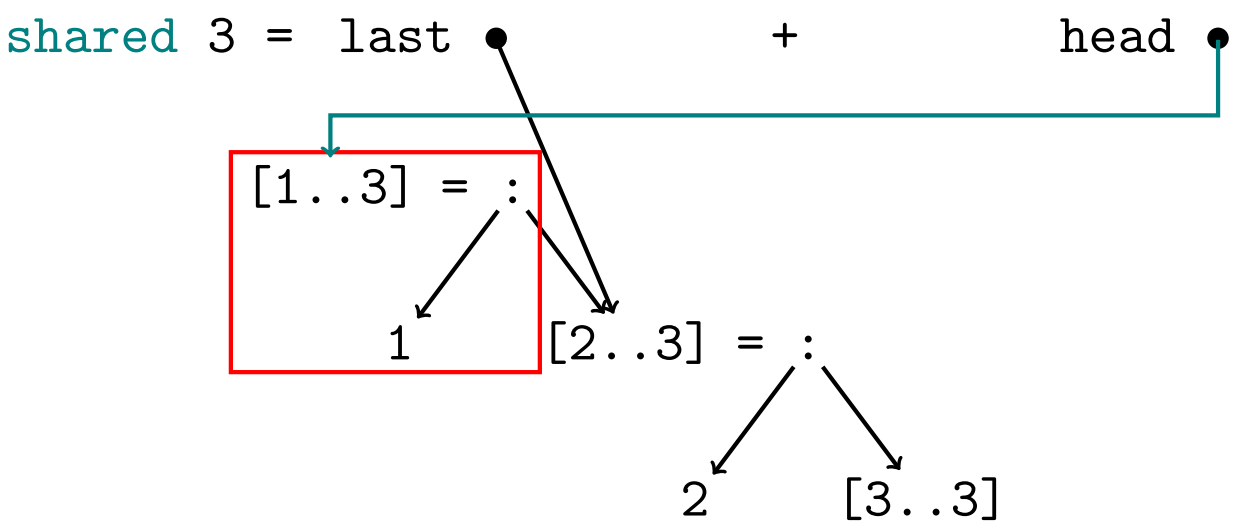
```
unshared n = last [1..n] + head [1..n]
shared   n = let xs = [1..n] in last xs + head xs

last (x:[]) = x                          head (x:xs) = x
last (x:xs) = last xs
```



Memory consumption of `unshared`: $\mathcal{O}(1)$ shared: $\mathcal{O}(n)$

Visualizing sharing

```
unshared n = last [1..n] + head [1..n]
shared   n = let xs = [1..n] in last xs + head xs

last (x:[]) = x                          head (x:xs) = x
last (x:xs) = last xs
```

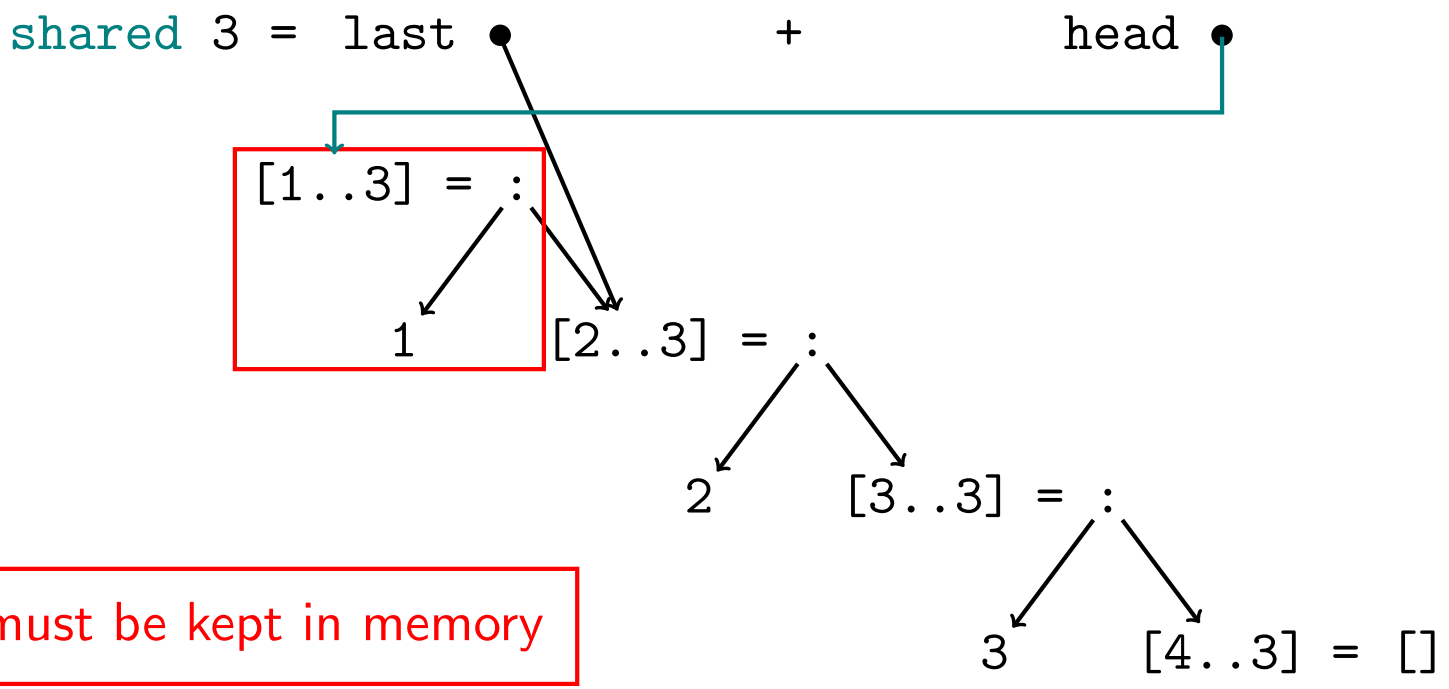


must be kept in memory

Memory consumption of `unshared`: $\mathcal{O}(1)$ shared: $\mathcal{O}(n)$

Visualizing sharing

```
unshared n = last [1..n] + head [1..n]
shared   n = let xs = [1..n] in last xs + head xs

last (x:[]) = x                          head (x:xs) = x
last (x:xs) = last xs
```

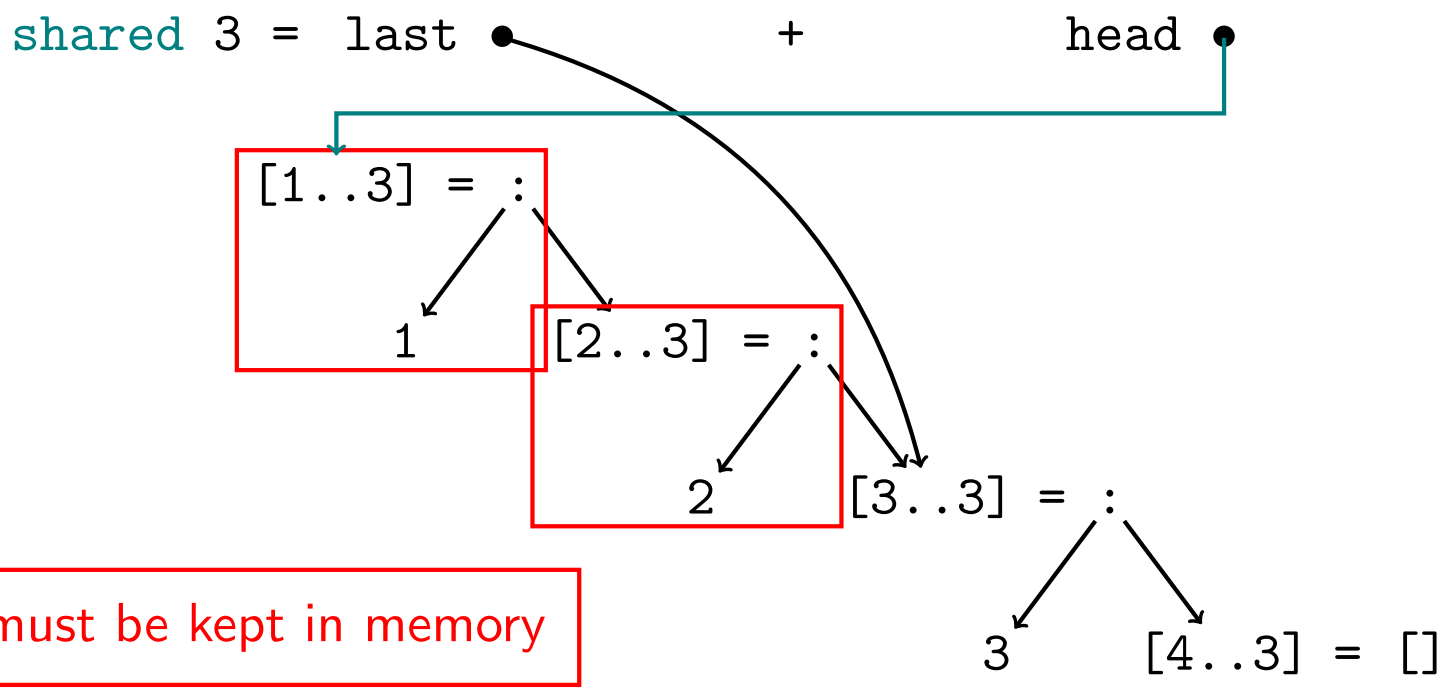


must be kept in memory

Memory consumption of `unshared`: $\mathcal{O}(1)$ shared: $\mathcal{O}(n)$

Visualizing sharing

```
unshared n = last [1..n] + head [1..n]
shared   n = let xs = [1..n] in last xs + head xs

last (x:[]) = x                          head (x:xs) = x
last (x:xs) = last xs
```

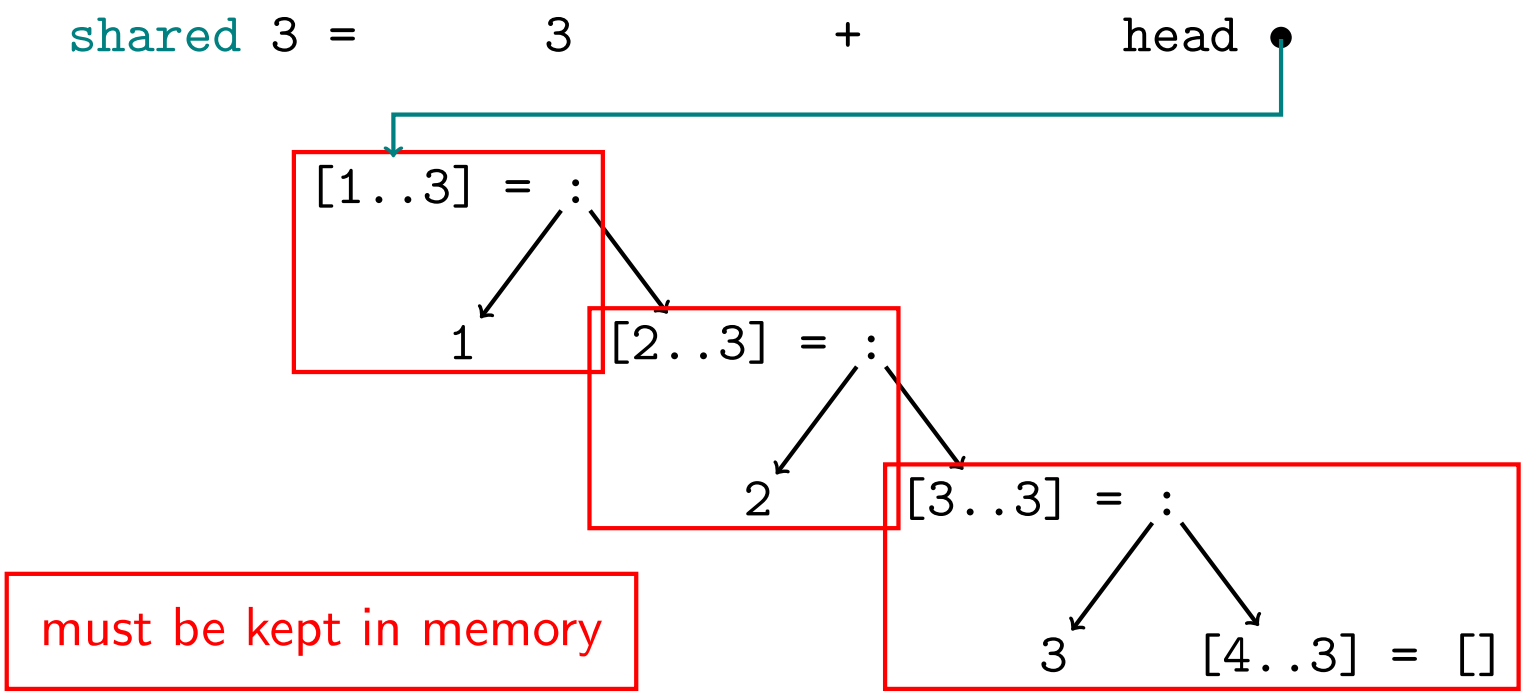


must be kept in memory

Memory consumption of `unshared`: $\mathcal{O}(1)$ shared: $\mathcal{O}(n)$

Visualizing sharing

```
unshared n = last [1..n] + head [1..n]
shared   n = let xs = [1..n] in last xs + head xs

last (x:[]) = x                          head (x:xs) = x
last (x:xs) = last xs
```



Memory consumption of `unshared`: $\mathcal{O}(1)$ shared: $\mathcal{O}(n)$

Avoid sharing with unit closures

- Function parameters are always shared. How can we avoid it?

```
shared :: [Int] -> Int
shared xs = last xs + head xs
```

```
> ghci +RTS -M10m
? shared [1..1000000]
Heap exhausted
```

- Idea: Function applications cannot be shared.
  Turn parameters into functions (**unit closures**)!

```
unshared :: (() -> [Int]) -> Int
unshared xs = last (xs ()) + head (xs ())
```

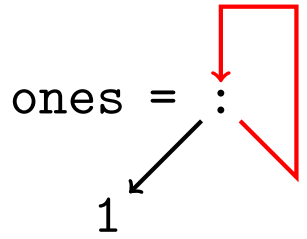
```
> ghci +RTS -M10m
? unshared (\_ -> [1..1000000])
1000001
```

- Disable sharing transformation with `-fno-full-laziness` in GHC mode `-O`.

Cyclic structures

- Recursion + sharing can lead to cycles in graph.

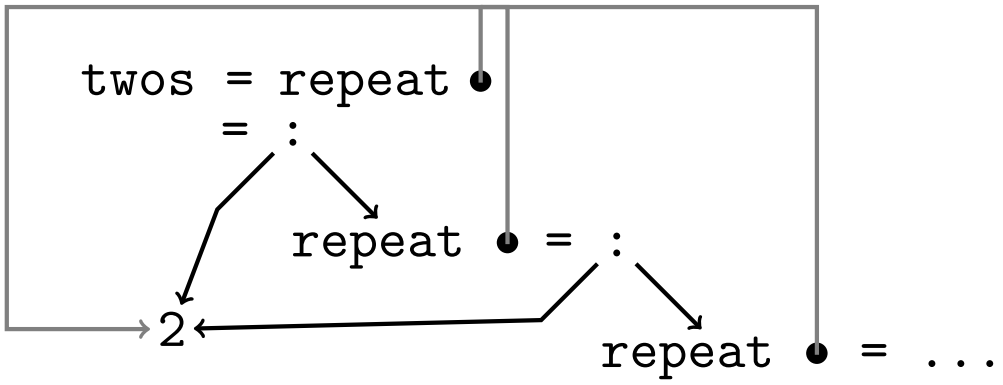
```
ones = 1 : ones
```



Generator for infinite list of 1s running in constant space.

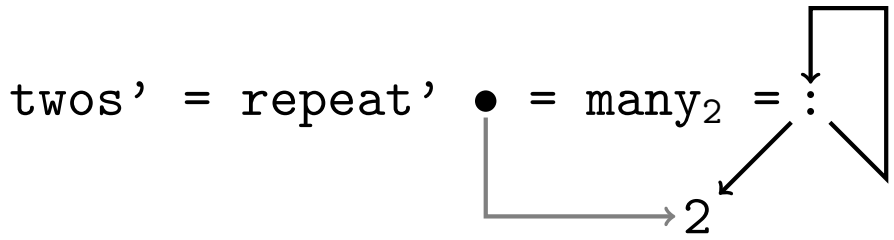
- How would you implement the infinite list of x?

```
repeat x = x:repeat x
twos     = repeat 2
```



Applications are not shared!

```
repeat' x = let many=x:many in many
twos'     = repeat' 2
```



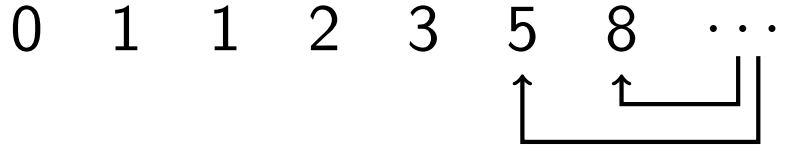
Constant memory no matter how deep `twos'` is evaluated.

Dynamic programming

- **Dynamic programming** computes solution bottom-up.
  Remember results and reuse them for larger instances.
- Example: Naive Fibonacci recomputes `fib(n-`k`)` many times.

```
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)          -- exponential runtime
```

- Fibonacci with dynamic programming: Build the list of `fib(i)`.



  To compute the next fib number, the last two values suffice.

```
fibDP n = go 0 1 n                         -- linear runtime
  where go a b 0 = a
        go a b i = (go b $! a + b) (i - 1)
```

  Prove: go $(\texttt{fib}\ (n-i))\ (\texttt{fib}\ ((n-i)+1))\ i = \texttt{fib}\ n$

Editing distance revisited

Goal: find "cheapest" sequence of editing steps using operations

Change a character	cost 1
Copy a character without change	cost 0
Delete a character	cost 1
Insert a character	cost 1

```
data Edit = Change Char | Copy | Delete | Insert Char

transform :: String -> String -> [Edit]
cost :: [Edit] -> Int

? transform "Hello" "help"
[Change 'h', Copy, Copy, Change 'p', Delete]

? cost $ transform "Hello" "help"
3
```

Transform performs redundant computations

- Implementation of `transform`:

```
transform [] [] = []
transform xs [] = map (\_ -> Delete) xs
transform [] ys = map Insert ys
transform (x:xs) (y:ys)
  | x == y       = Copy : transform xs ys
  | otherwise    = best [ Delete   : transform xs     (y:ys)
                        , Insert y : transform (x:xs) ys
                        , Change y : transform xs     ys     ]
```

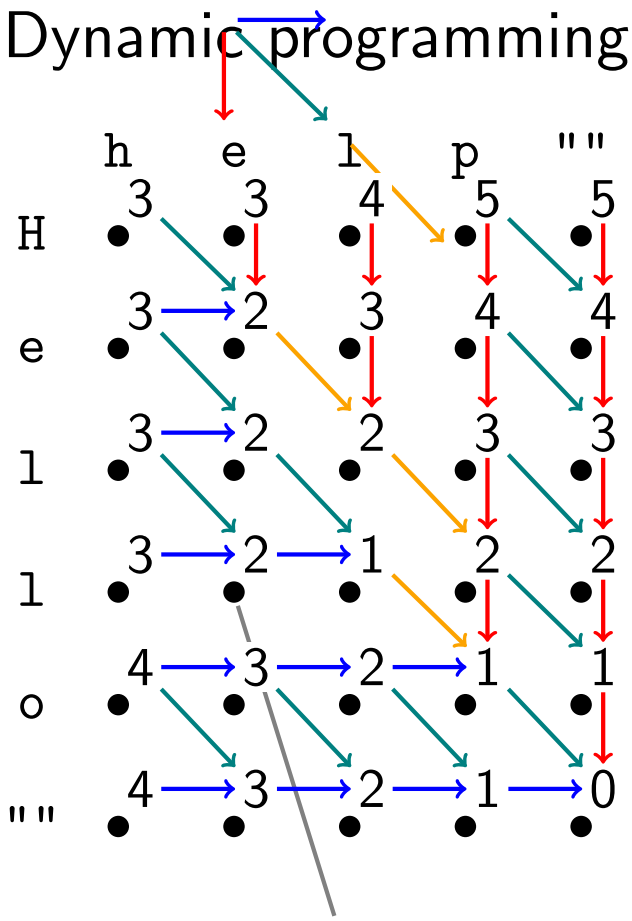
- `transform` repeatedly computed on the same subsequences:

```
transform "ac" "bd"
  Delete     -> transform "c" "bd"
                  Insert 'b' -> transform "c" "d"
  Insert 'b' -> transform "ac" "c"
                  Delete -> transform "c" "d"
  Change 'b' -> transform "c" "d"
```

Results in exponential run-time.

Editing distance with dynamic programming

Dynamic programming: Store sub-computations in a table.



transform `"lo"` into `"elp"`

To transform `"Hello"` into `"help"`, we can

delete `H` and transform `"ello"` into `"help"`,
insert `h` and transform `"Hello"` into `"elp"`, or
change `H` to `h` and transform `"ello"` into `"elp"`.

To transform `"ello"` into `"elp"`, we
copy `e` and transform `"llo"` into `"lp"`.

Fill table from bottom right to top left.

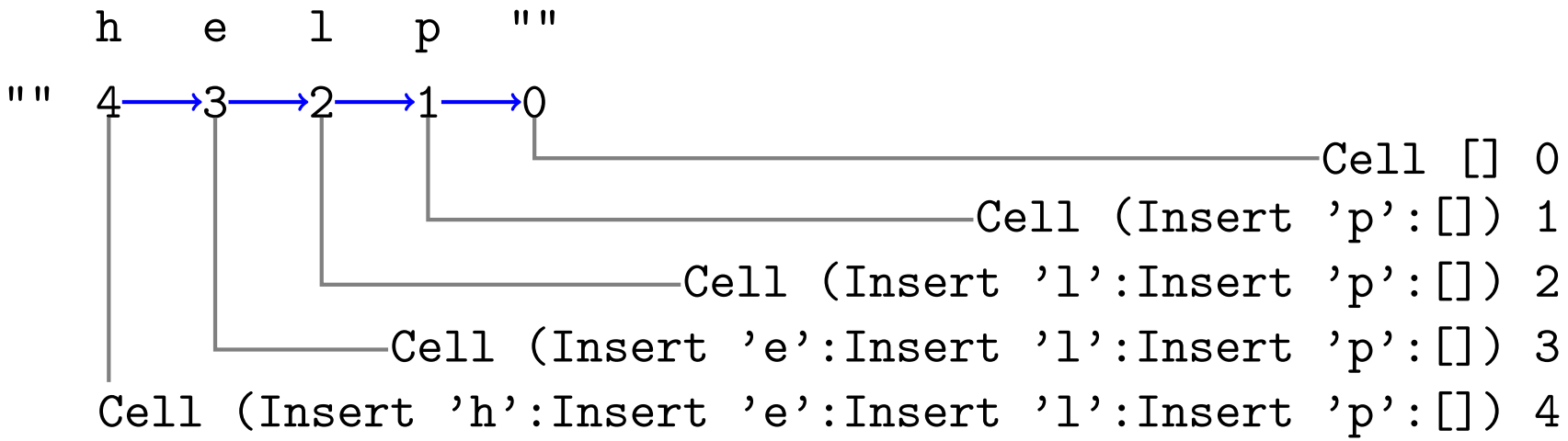
Table cells as an ADT

Every cell stores the sequence of edits and their costs.

```
module Cell (Edit(..), Cell, edits, cost,
             empty, change, copy, insert, delete) where

data Edit = Change Char | Copy | Delete | Insert Char
  deriving (Eq,Show)

data Cell = Cell [Edit] Int deriving (Show)

edits (Cell e _) = e
cost  (Cell _ c) = c

-- ADT ensures invariant cost c == length (filter (/= Copy) (edits c))

empty                          = Cell []                      0
change c (Cell edits cost) = Cell (Change c : edits) $! (cost + 1)
copy     (Cell edits cost) = Cell (Copy     : edits)    cost
delete   (Cell edits cost) = Cell (Delete   : edits) $! (cost + 1)
insert c (Cell edits cost) = Cell (Insert c : edits) $! (cost + 1)
```

The last row of the table I

In the last row, each cell is a `foldr` of `insert` over the suffix.

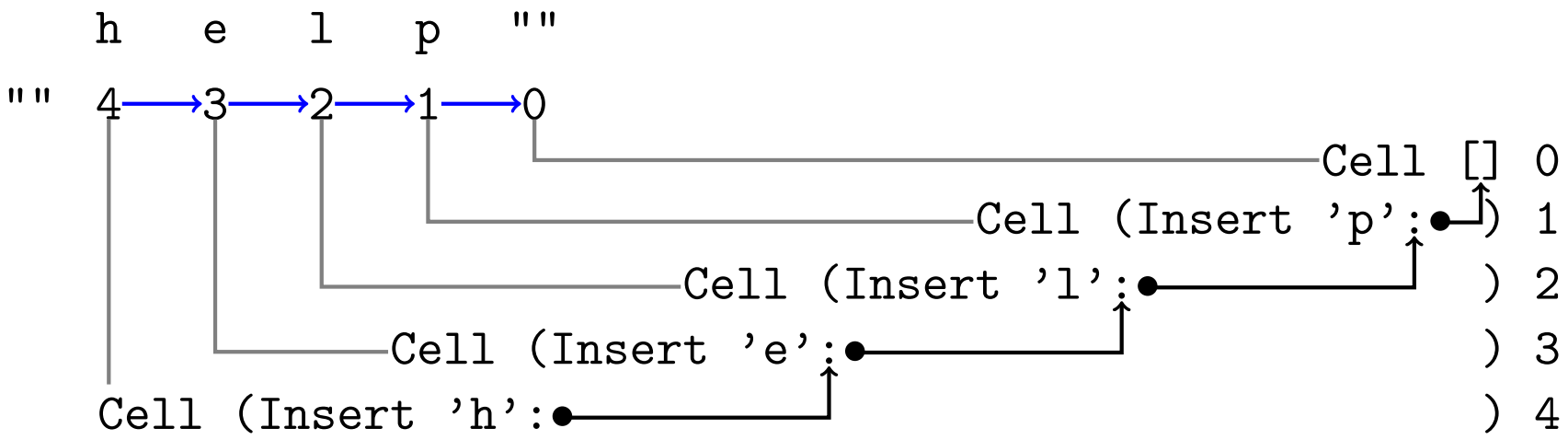



```
tails [] = [[]]
tails (x:xs) = (x:xs) : tails xs

lastRow ys = map (foldr insert empty) (tails ys)

? lastRow "lp"
[Cell [Insert 'l',Insert 'p'] 2,Cell [Insert 'p'] 1,Cell [] 0]
```

Suffixes of edits are not shared, i.e., time and space $\mathcal{O}(n^2)$.

The last row of the table II

In the last row, each cell is a `foldr` of `insert` over the suffix.



`scanr f` computes **list of right-folds** of `f` with sharing.

```
scanr :: (a -> b -> b) -> b -> [a] -> [b]
scanr f z []     = [z]
scanr f z (x:xs) = f x (head bs) : bs where bs = scanr f z xs

lastRow = scanr insert empty
```

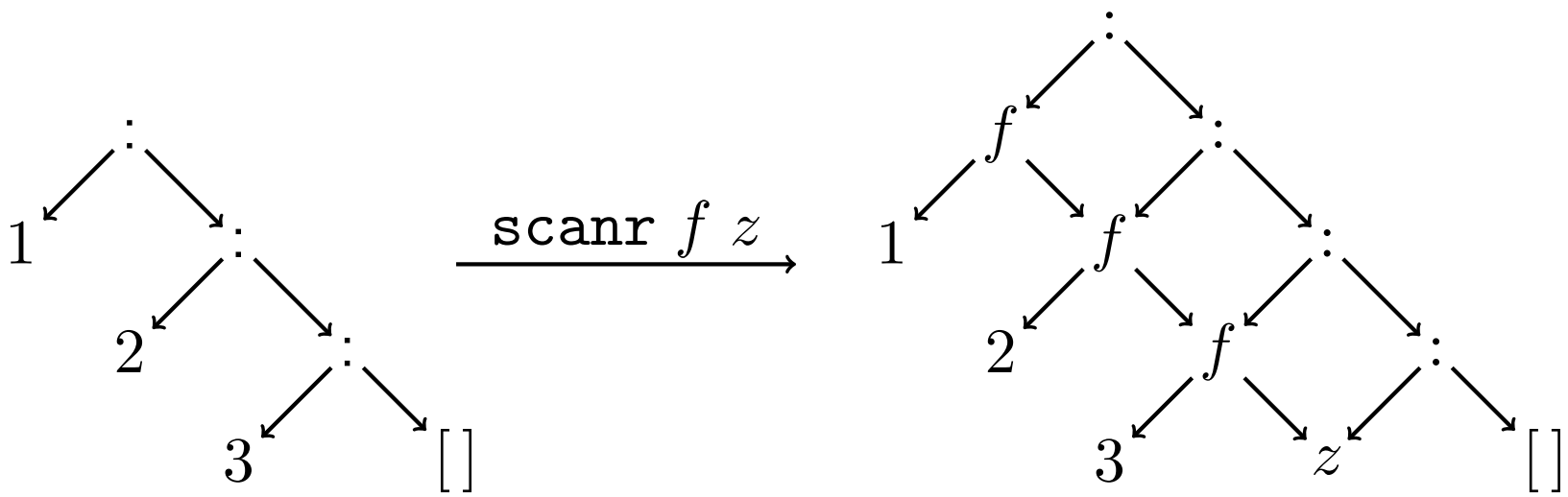
runs in space and time $\mathcal{O}(n)$

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

Visualizing `scanr`

```
scanr :: (a -> b -> b) -> b -> [a] -> [b]
scanr f z [] = [z]
scanr f z (x:xs) = f x (head bs) : bs where bs = scanr f z xs
```

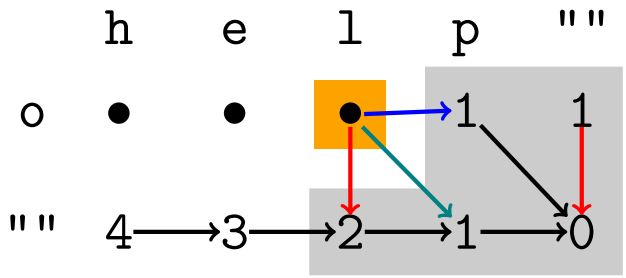
$$\texttt{scanr}\ f\ z\ [1,2,3] = [f\ 1\ (f\ 2\ (f\ 3\ z)), f\ 2\ (f\ 3\ z), f\ 3\ z, z]$$



$$\texttt{head}\ (\texttt{scanr}\ f\ z\ xs) = \texttt{foldr}\ f\ z\ xs$$

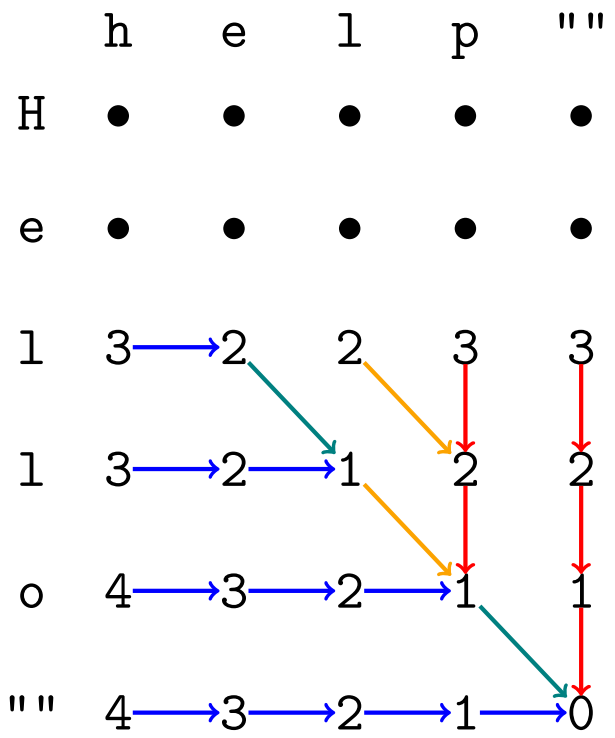
Filling the other rows

We only need the row below and the remainder of the current row.




```
type Row = [Cell]

fillRow :: String -> Char -> Row -> Row
fillRow ""     x (south: _                    ) = [delete south]
fillRow (y:ys) x (south:southEast:restBelow) = cell : east : currRest
  where
    (east : currRest) = fillRow ys x (southEast : restBelow)
    cell
      | x == y    = copy southEast
      | otherwise =
        best [ delete south, insert y east, change y southEast ]

best [c]    = c
best (c:cs) = if cost c <= cost c' then c else c'
              where c' = best cs
```

Filling the table



When one row is finished,

- the current row becomes the next row,
- and we start again filling the row with `fillRow ys`

This is again a fold.
The table is the **list of folds over the suffixes**, i.e., `scanr`.

```
fillTable :: String -> String -> [Row]
fillTable xs ys = scanr (fillRow ys) (lastRow ys) xs

transform :: String -> String -> [Edit]
transform xs ys = edits (head (head (fillTable xs ys)))
```

Exercise: How can this be simplified?

The complete code for the table

```
import Cell

lastRow = scanr insert empty

fillRow ""     x (south: _                    ) = [delete south]
fillRow (y:ys) x (south:southEast:restBelow) = cell : east : currRest
  where
    (east : currRest) = fillRow x ys (southEast : restBelow)
    cell
      | x == y    = copy southEast
      | otherwise =
        best [ delete south, insert y east, change y southEast ]

best [c]    = c
best (c:cs) = if cost c <= cost c' then c else c'
              where c' = best cs

fillTable xs ys = scanr (fillRow ys) (lastRow ys) xs

transform xs ys = edits (head (head (fillTable xs ys)))
```

Transform examples

```
? :set +s

? transform "Hello" "help"
[Change 'h',Copy,Copy,Delete,Change 'p']
(0.00 secs, 552704 bytes)

? transform "123456" "654321"
[Delete,Change '6',Change '5',Copy,Insert '3',Change '2',Change '1']
(0.00 secs, 1061296 bytes)

? transform "12345678" "87654321"
[Delete,Change '8',Change '7',Change '6',Copy,Insert '4',Change '3',
 Change '2',Change '1']
(0.00 secs, 1134120 bytes)                    --former impl. took > 1 s

? transform "FMFP is challenging!" "Haskell is cool!"
[Insert 'H',Insert 'a',Insert 's',Change 'k',Change 'e',Change 'l',
 Change 'l',Copy,Copy,Copy,Copy,Copy,Delete,Change 'o',Change 'o',
 Copy,Delete,Delete,Delete,Delete,Delete,Delete,Copy]
(0.01 secs, 1580096 bytes)
```

Deforestation

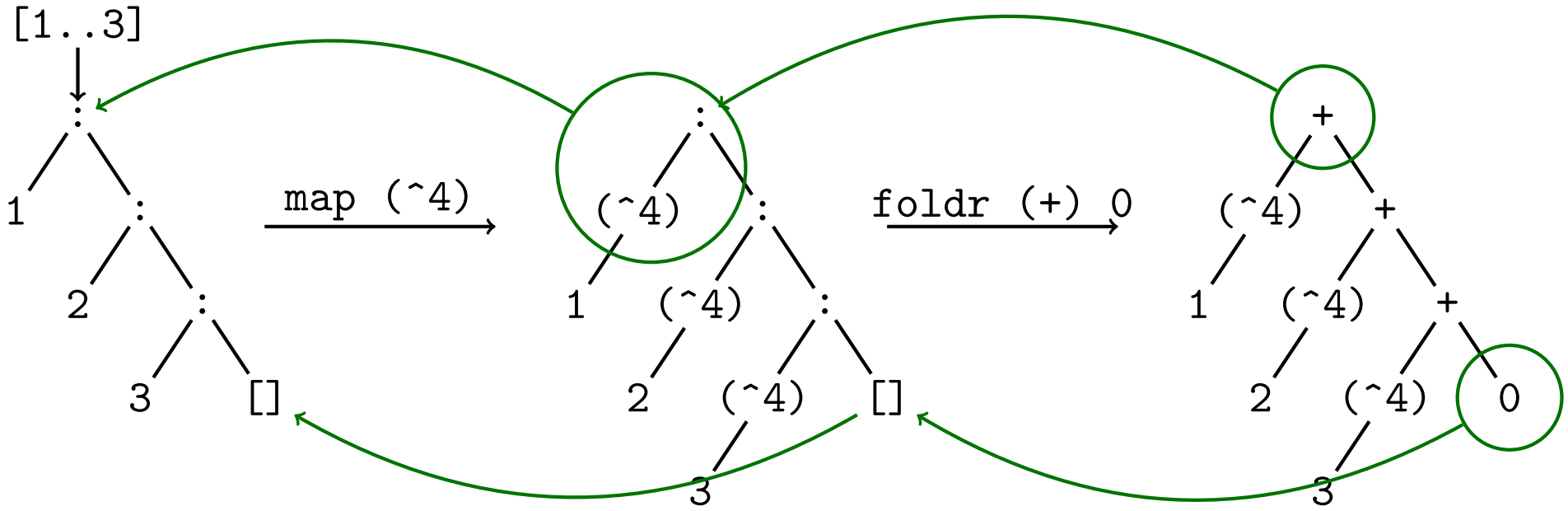
Recall the `sumFourthPowers` example and its execution:

```
foldr f z []     = z                    map f []     = []
foldr f z (x:xs) = f x (foldr f z xs)   map f (x:xs) = f x : map f xs

sumFourthPowers n = foldr (+) 0 (map (^4) [1..n])


sumFourthPowers n
 = foldr (+) 0 (map (^4) [1..n])
 = foldr (+) 0 (map (^4) (1 : [2..n]))
 = foldr (+) 0 (1^4 : map (^4) [2..n])
 = 1^4 + foldr (+) 0 (map (^4) [2..n])
 = ...
```

Intermediate lists are **not** constructed fully thanks to laziness,
but we still allocate the list nodes during the execution.

Deforestation avoids such intermediate allocations.

Foldr Build Fusion



- `[1..3]` builds list from constructors `(:)` and `[]`.
- `foldr f z` replaces `(:)` with `f` and `[]` with `z`.
- If we replace `(:)` and `[]` with `f` and `z` when we build the list, we avoid intermediate allocations.
- Let's parametrize `[1..3]` and `map` over `(:)` and `[]`.

Parametrizing over (:) and []

- Conventional implementation for [n .. m]:

```
enumFromTo ::                              Int -> Int -> [Int]
enumFromTo n m
  | n <= m    = n : enumFromTo (n + 1) m
  | otherwise = []
```

- Abstract over (:) and []

```
enumFromToFB :: (Int -> a -> a) -> a -> Int -> Int -> a
enumFromToFB cons nil n m
  | n <= m    = n `cons` enumFromToFB (n + 1) m
  | otherwise = nil
```

- We get back the old implementation by providing (:) and []:

```
build g = g (:) []

enumFromTo n m = build (\cons nil -> enumFromToFB cons nil n m)
```

The foldr-build fusion rule

```
foldr f z (build g) = g f z
```

Side condition: `g` is polymorphic in `z`'s type.

- GHC implements `enumFromTo` with `build enumFromToFB` when it occurs under a `foldr` and applies the fusion rule.

```
foldr (+) 0 (enumFromTo n m)                          -- at compile time
 = foldr (+) 0 (build (\cons nil -> enumFromToFB cons nil n m))
 = enumFromToFB (+) 0 n m
```

- No intermediate list at run-time any more!

```
enumFromToFB (+) 0 1 3                                -- at run time
 = 1 + enumFromToFB (+) 0 (1+1) 3
 = 1 + (2 + enumFromToFB (+) 0 (2+1) 3)
 = 1 + (2 + (3 + enumFromToFB (+) 0 (3+1) 3))
 = 1 + (2 + (3 + 0))
```

Fusion for `map`

- Write `map` as `foldr`:

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

- Abstract over `(:)` and `[]`

```
mapFB :: (b -> c -> c) -> c -> (a -> b) -> [a] -> c
mapFB cons nil f = foldr (\x xs -> f x `cons` xs) nil

map f xs = build (\cons nil -> mapFB cons nil f xs)
```

- GHC's optimizer eliminates the intermediate lists.

```
sumFourthPowers n
 = foldr (+) 0 (map (^4) [1..n])
 = foldr (+) 0 (build (\cons nil -> mapFB cons nil (^4) [1..n]))
 = mapFB (+) 0 (^4) [1..n]
 = foldr (\x xs -> x^4 + xs) 0
     (build (\cons nil -> enumFromToFB cons nil 1 n))
 = enumFromToFB (\x xs -> x^4 + xs) 0 1 n
```

Summary on deforestation

- Deforestation eliminates intermediate data structures
  - Allows to write programs at a **high level of abstraction** without losing efficiency.
  - `foldr-build` is just one example
- Optimized implementations must be provided manually.

  Library writers write them and tell GHC how to use them.
- Prefer combinators to manual recursive implementation.

  The compiler knows how to optimize the combinators,
  but not your own recursive function.

Summary on efficiency

- **Measure first!** Optimize only if necessary.
- Strictness annotations force evaluation of arguments
  - They cause extra work, so use sparingly.
  - `$!` evaluates only until the first constructor.
- Accumulators and tail recursion
  - Accumulators should be strict. Prefer `foldl'` over `foldl`.
  - Tail recursion avoids call stack overflows, but destroys laziness.
- Sharing
  - To avoid recomputations, extract them as `let`/`where` bindings.
  - Sharing can cause space leaks.
- Use list combinators! GHC knows them better than your functions.

A word of warning

Before you optimize, **double-check** that your algorithm is fast!

```
fib 0 = 0                                                    -- exponential
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)

fibDP n = go 0 1 n                                           -- linear with DP
  where go a b 0 = a
        go a b i = (go b $! a + b) (i - 1)
```

Exploit identities to get faster:

$$\texttt{fib}\,(2 \cdot n) = (2 \cdot \texttt{fib}\,(n+1) - \texttt{fib}\,n) \cdot \texttt{fib}\,n$$

```
fibFast n = fst (aux n)                                      -- logarithmic
  where
    aux 0 = (0, 1)           -- aux n = (fib n, fib (n+1))
    aux n | even n    = ((2*b-a)*a, b^2 + a^2)
          | otherwise = (b^2 + a^2, b*(2*a+b))
      where (a, b) = aux (n `div` 2)
```