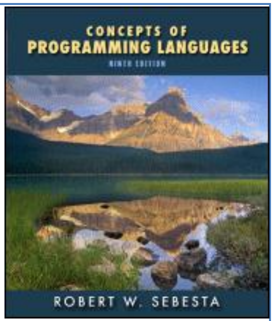
Haskell



1 Ch15 – Functional Programming Languages
   - 15.8 Haskell

2 Haskell, named after Haskell Curry
   - Download Winhugs

3 Reference
   - A Gentle Introduction to Haskell, Paul Hudak, et.al.

Functions

- Function and λ expression

```
f n = if n==0 then 1 else n*f(n-1)
f = \n->if n==0 then 1 else n*f(n-1)  -- spaces between = and \
```

- Pattern matching

```
f 0 = 1
f n = n*f(n-1)
f = \n->case n of 0->1
                  n->n*f(n-1)
```

- Guard

```
f n | n==0 = 1
    | otherwise = n*f(n-1)
```

Cf. Math definition

f(n) = 1, if n=0

= n*f(n-1), otherwise

Functions

- Let expression

```
f n = let f 0 a = a
          f n a = f (n-1) (n*a)
      in f n 1
```

- where clause

```
f n = f n 1
      where f 0 a = a
            f n a = f (n-1) (n*a)
```

- layout

  Line up in columns after let, where, and of.

```
module Main where
f n = let f 0 a
            = a          -- ok
            ; f n a =    -- ok
f n = let f 0 a
          = a            -- no
        f n a =          -- no
       ; f n a =         -- no
```

Functions

- Curried function

```
c f g x = f (g x)
c = \f g x->f (g x)
c = \f-> \g-> \x->f (g x)          -- spaces between -> and \
c :: (a -> b) -> (c -> a) -> c -> b    -- polymorphic function

c db sq 5 where db x=x+x; sq x=x*x
(db `c` sq) 5 where db x=x+x; sq x=x*x

(db . sq) 5 where db x=x+x; sq x=x*x
(.) db sq 5 where db x=x+x; sq x=x*x
(.) :: (a -> b) -> (c -> a) -> c -> b
```

Functions

- Section (partial application)

(+) ≡ \x y->x+y — (+) 2 3 ⇒ 5 :: Integer

(+2) ≡ \x->x+2 — (+2) 3 ⇒ 5 :: Integer

(2+) ≡ \y->2+y — (2+) 3 ⇒ 5 :: Integer

map (^2) [1,2,3,4,5] ⇒ [1,4,9,16,25] :: [Integer]

map (2^) [1,2,3,4,5] ⇒ [2,4,8,16,32] :: [Integer]

filter (>3) [1,2,3,4,5] ⇒ [4,5] :: [Integer]

filter (3>) [1,2,3,4,5] ⇒ [1,2] :: [Integer]

Exception

(2-) is a section;

(-2) isn't a section.

Lists

- Constructor and selector

```
xs = [2,3,4,5]
1 : xs       ⇒ [1,2,3,4,5] :: [Integer]
head xs      ⇒ 2 :: Integer
tail xs      ⇒ [3,4,5] :: [Integer]
null xs      ⇒ False :: Bool
xs++xs       ⇒ [2,3,4,5,2,3,4,5] :: [Integer]
```

- List-processing functions

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

```
filter f [] = []
filter f (x:xs) | f x = x : filter f xs
                | otherwise = filter f xs
```

Lists

- List-processing functions

```
product [] = 1
product (x:xs) = x * product xs
enumFromTo m n | m > n = []
               | otherwise = m : enumFromTo (m+1) n
```

- Arithmetic sequence

```
[1..9]      ⇒ [1,2,3,4,5,6,7,8,9] :: [Integer]
[1,3..9]    ⇒ [1,3,5,7,9] :: [Integer]
[1..9] ≡ enumFromTo 1 9              -- syntactic sugar
[1,3..9] ≡ enumFromThenTo 1 3 9
factorial = product . enumFromTo 1
                      \n -> if 1 > n then [] else 1 : enumFromTo 2 n
```

Lists

- List comprehension

[x*x| x<-[1..9]] ⇒ [1,4,9,16,25,36,49,64,81]

$$[x*x \mid \underbrace{x\texttt{<-}[1..9]}_{\text{generator}},\ \underbrace{\text{even } x}_{\text{guard}}] \Rightarrow [4,16,36,64] :: [\text{Integer}]$$

≡ map (^2) (filter even (enumFromTo 1 9))

N.B. $\{\ x^2 \mid 1 \leq x \leq 9,\ x \text{ is even}\ \}$ is called a set comprehension.

[(x,y)| x<-[1..2],y<-[1..2]]

⇒ [(1,1),(1,2),(2,1),(2,2)] :: [(Integer,Integer)]

qsort [] = []

qsort (x:xs) = qsort [y|y<-xs,y<x] ++ [x] ++ qsort [y|y<-xs,y>=x]

Lazy evaluation

- Non-strict function

  f ⊥ = ⊥ f is a strict function

  f ⊥ ≠ ⊥ f is a non-strict function

  f x = x+x f is strict; f (1`div` 0) = ⊥

  f x = 7 f is non-strict; f (1`div` 0) = 7

  +, -, *, /, etc are strict in both arguments; e.g. ⊥ + ⊥ = ⊥.

  : is non-strict in both its arguments

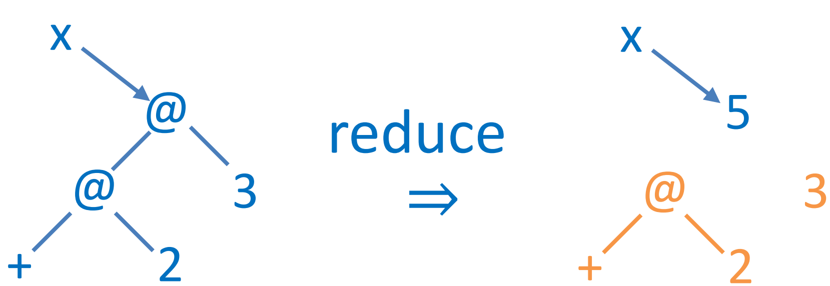
```
length [] = 0
length (_:xs) = 1+length xs        -- : is a lazy constructor.
length [div 1 0,mod 1 0] ⇒ 2       -- Lists are lazy data structures
-- or, length (div 1 0 : mod 1 0 : [])
```

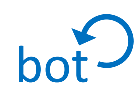
Lazy evaluation

- Lazy evaluation

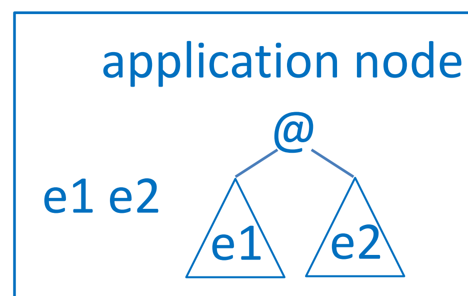
```
x = 2 + 3      -- Do we need to evaluate 2+3 when defining x?
Hugs>  x       -- Do we need to evaluate 2+3 now?
Hugs>  x       -- Do we need to evaluate 2+3 again?
bot = bot      -- Do we need to evaluate bot when defining bot?
Hugs>  bot  -- Do we need to evaluate bot now?  Infinite …
```

- Graph reduction





Orange-colored nodes become garbage.



Graph reduction

- Graph reduction algorithm

While the expression is still reducible do

1 Select the next redex (reducible expression) by unwinding the left spine of the graph to the first non-@ node

2 Reduce it

3 Update the root of the redex with the result

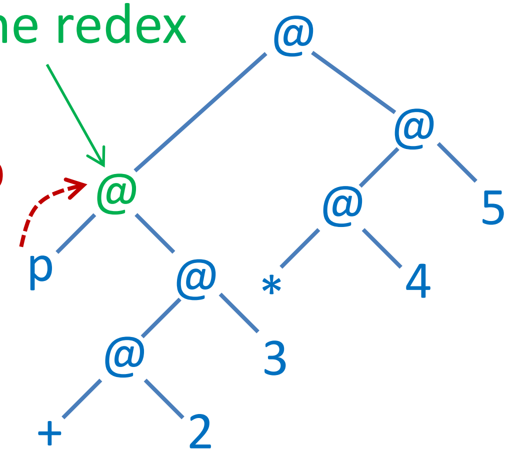
p x y =x+y

p (2+3) (4*5)

redex to reduce

root of the redex

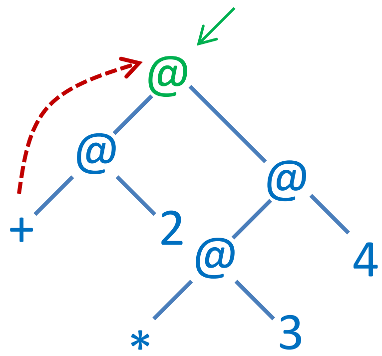
move 1-step up for the root

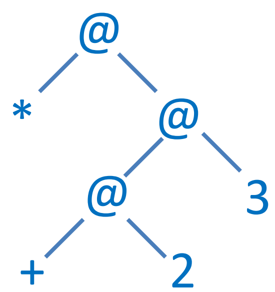


Graph reduction

- Graph reduction algorithm

2+3*4 root of the redex (*) (2+3)





In Haskell, (*)(2+3) is irreducible.

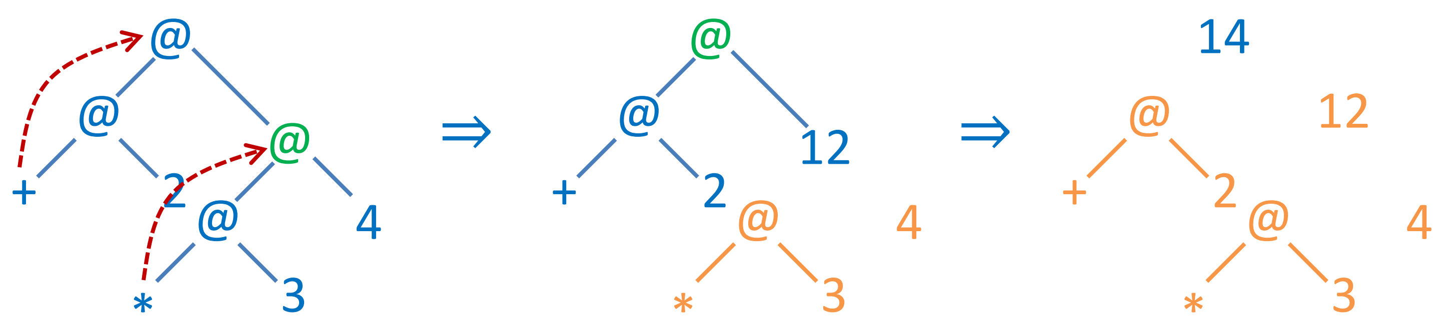
In technical term, it is an expression in WHNF (Weak Head Normal Form), and hence irreducible.

Hugs> (*) (2+3)

primMulInteger (2 + 3) :: Integer -> Integer

Graph reduction

- Reduce primitive function applications
  1. Reduce strict arguments, if any
  2. Execute the primitive function
  3. Update the root of the redex with the result
- Example – Reduction of 2+3*4



Green-colored nodes are the roots of the next redexes.

Graph reduction

- Reduce λ applications
  1. Copy the λ body
  2. Substitute a pointer to the argument for each occurrence of the formal parameter
  3. Update the root of the redex with the result

Normal order reduction to WHNF
+
Substitute pointers (Sharing)
+
Update redex root with result

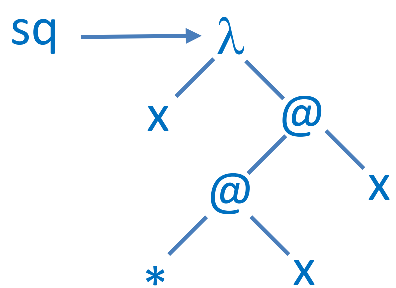
= Lazy evaluation

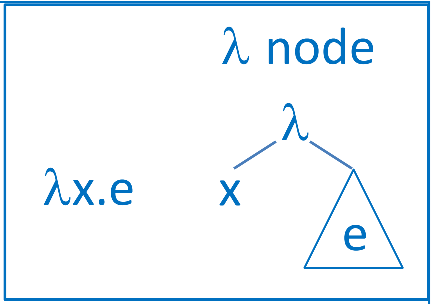
Graph reduction

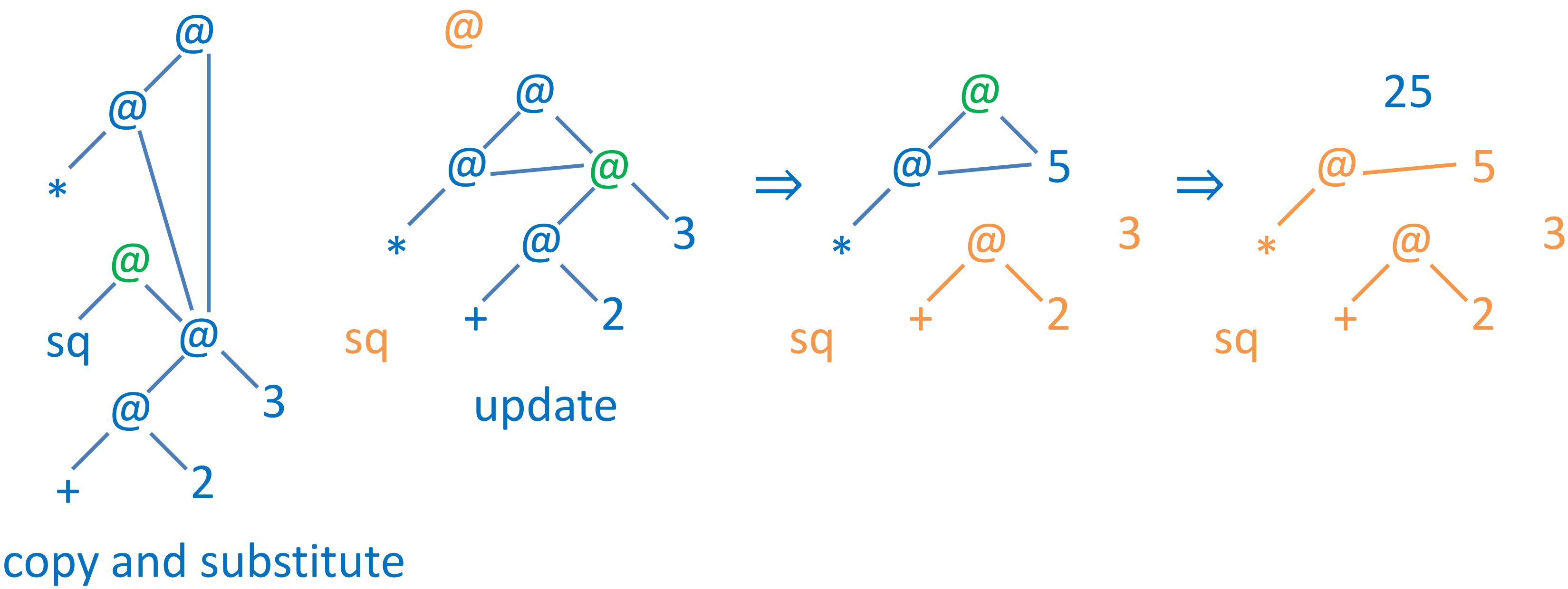
- Example

  sq x = x*x

  sq (2+3)





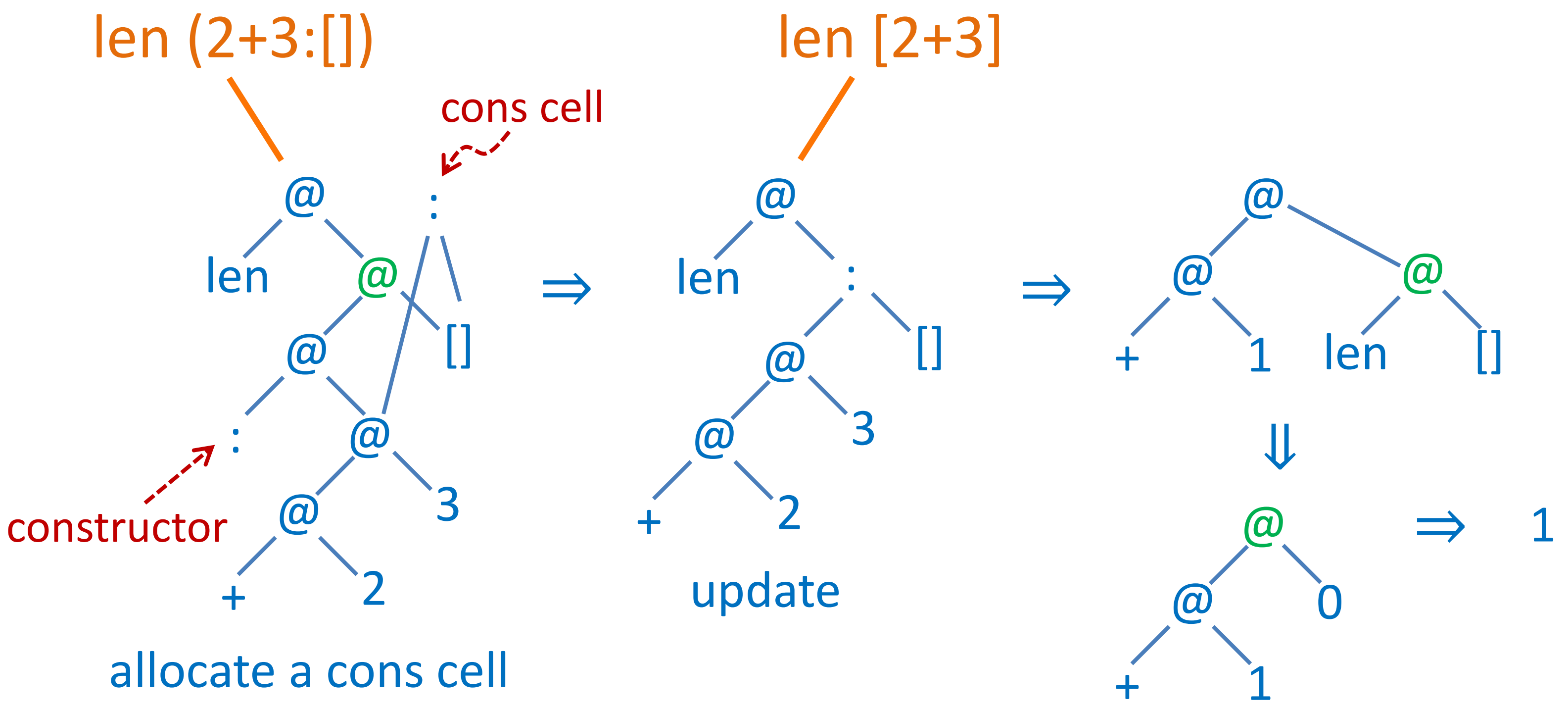


Graph reduction

- Reduce argument for pattern matching

  len [] = 0

  len (_:xs) = 1+len xs

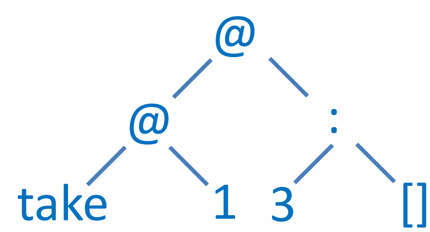


Graph reduction

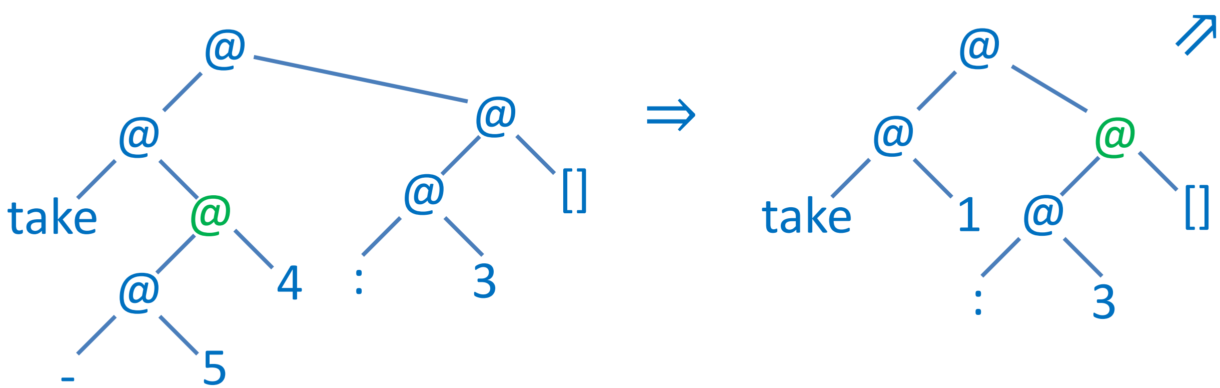
- Reduce argument for pattern matching

Pattern matching semantics: left-to-right, top-to-down

```
take 0 _ = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs
```



take (5-4) (3:[])



Pattern matching semantics

- Pattern matching semantics

```
-- Built-in take
take 0 _ = []
take _ [] = []                    take 0 bot = []
take n (x:xs) = x : take (n-1) xs take bot [] = bot

-- take: another version
take _ [] = []
take 0 _ = []                     take 0 bot = bot
take n (x:xs) = x : take (n-1) xs take bot [] = []
```

Lazy data structure

- Lazy data structure (Infinite data structure)

  take 5 [1..] ⇒ [1,2,3,4,5] :: [Integer]

  take 5 [1,3..] ⇒ [1,3,5,7,9] :: [Integer]

  [1..] = enumFrom 1

  [1,3..] = enumFromThen 1 3

  enumFrom n = n : enumFrom (n+1)

  enumFromThen n m = n : enumFromThen m (m+m-n)

  N.B. These recursive functions have no boundaries.

Lazy data structure

- Lazy data structure (Infinite data structure)

ints = enumFrom 1 -- define by generation

Here is the reduction of "take 2 ints" :

```
take 2 ints
= take 2 (enumFrom 1)
= take 2 (1 : enumFrom 2)
= 1 : take 1 (enumFrom 2)
= 1 : take 1 (2 : enumFrom 3)
= 1 : 2 : take 0 (enumFrom 3)
= 1 : 2 : []
= [1,2]
```

1+1

+1

2-1

```
take 0 _ = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs
enumFrom n = n : enumFrom (n+1)
```

3 becomes 2+1

2-1 is reduced here for pattern matching

1+1 is reduced here for output

Lazy data structure

- Remark

Output, take and enumFrom cooperate as coroutines.

output $\underset{\text{respond with an element}}{\overset{\text{request for an element}}{\rightleftarrows}}$ take $\underset{\text{respond with an element}}{\overset{\text{request for an element}}{\rightleftarrows}}$ enumFrom

Haskell is output-driven.

Hugs> enumFrom 1

[1,2,3,4,5,6,7,8,9,10,...... go infinitely

Were it not,

Hugs> enumFrom 1

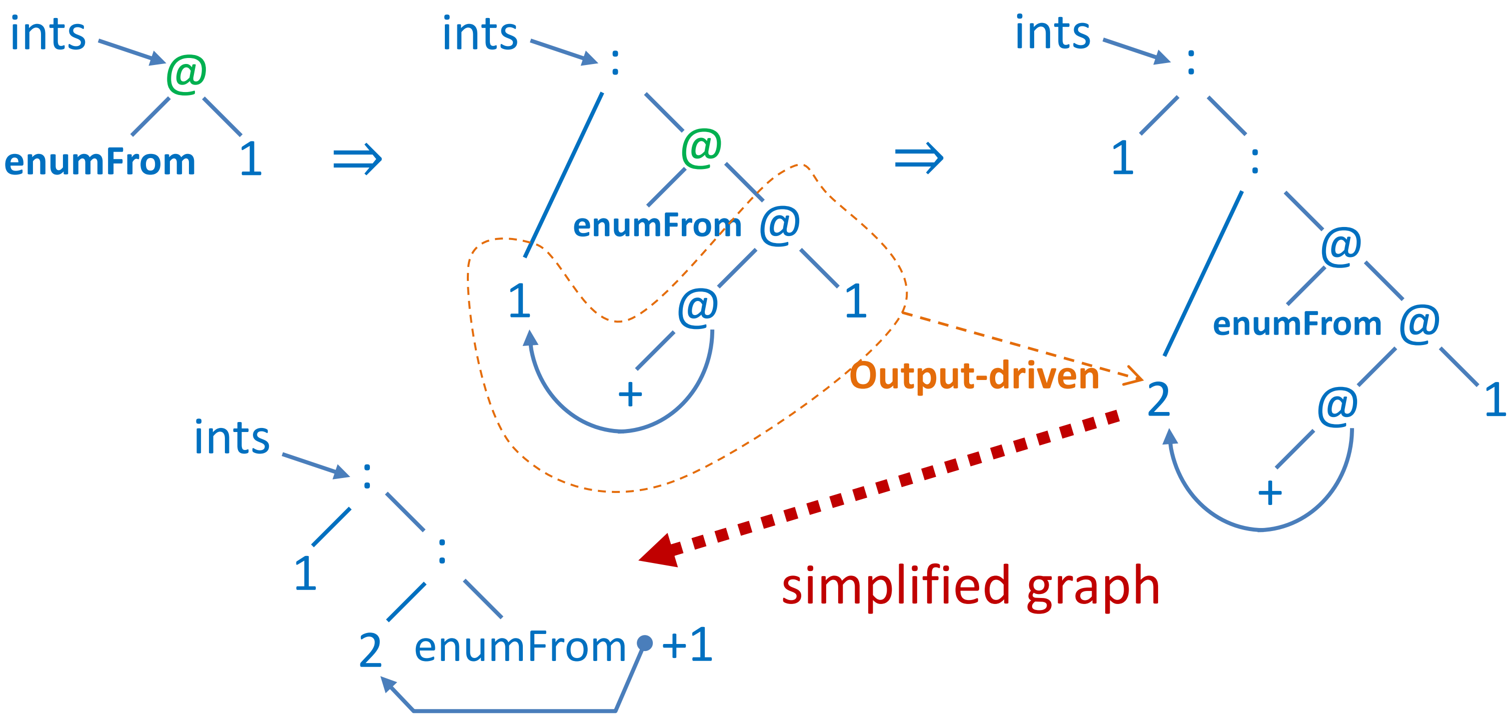
would evaluate to 1 : enumFrom 2 and terminate.

Lazy data structure

- Remark

  ints memoizes the already-computed elements.

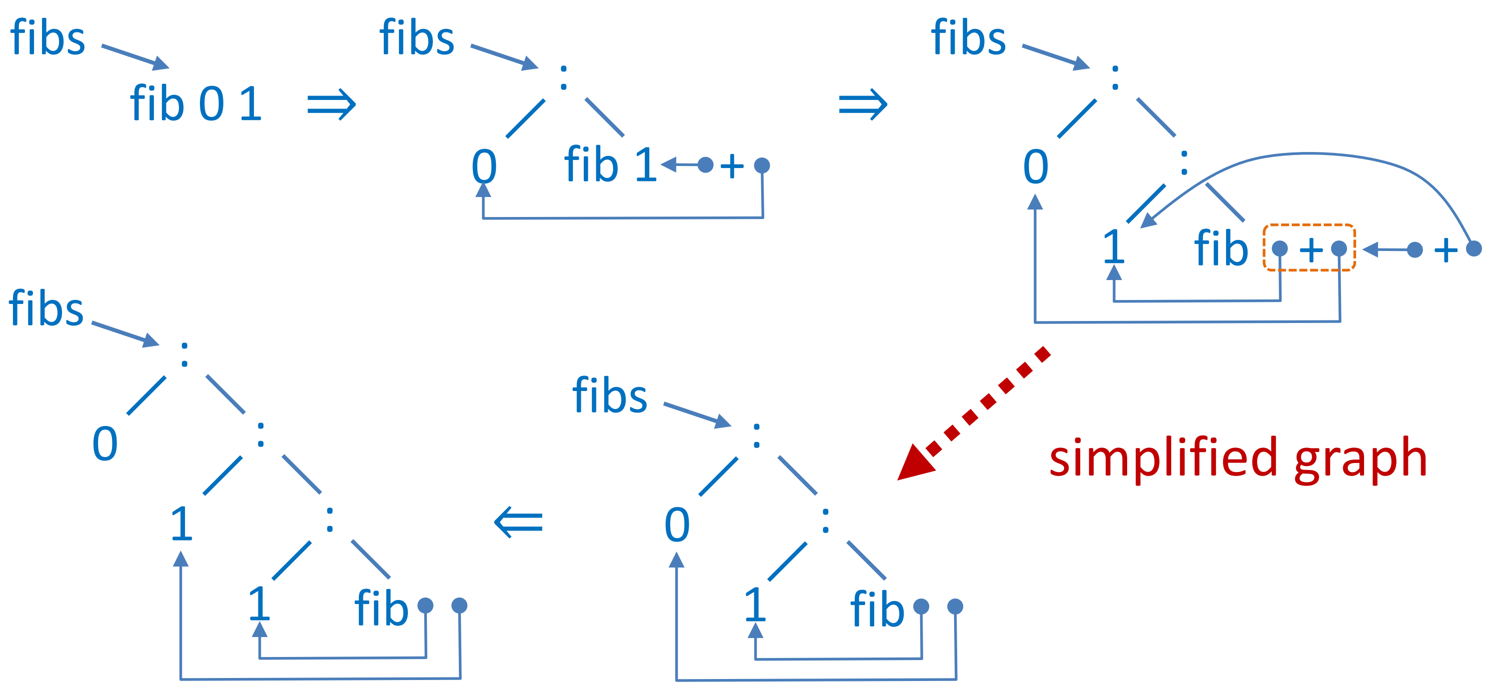
  enumFrom n = n : enumFrom (n+1)



Lazy data structure

- Lazy data structure (Infinite data structure)
  - fib a b = a : fib b (b+a)

    fibs = fib 0 1 -- define by generation



Lazy data structure

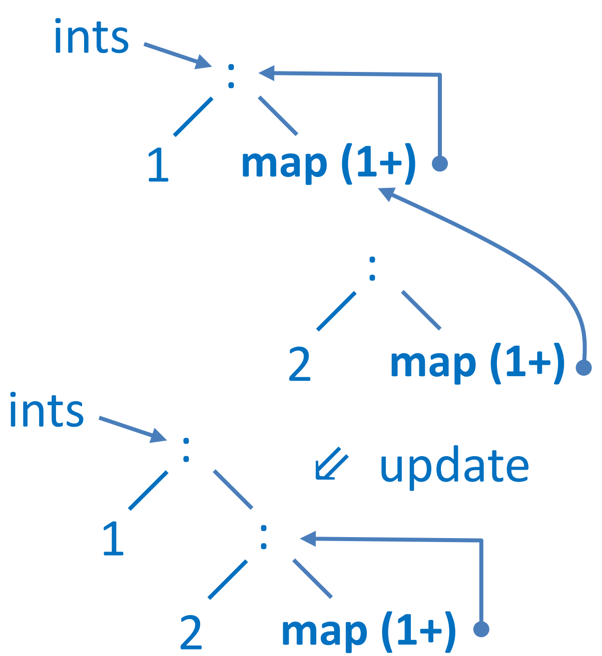
- Cyclic data structure

```
ints = 1 : [ x+1 | x <- ints ]
ints = 1 : map (+1) ints
```

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

Here is the reduction of "take 2 ints" :

```
take 2 ints
= take 2 (1 : map (1+) ints)
= 1 : take 1 (map (1+) ints)
= 1 : take 1 (2 : map (1+) (tail ints))
= 1 : 2 : take 0 (map (1+) (tail ints))
= 1 : 2 : []
```



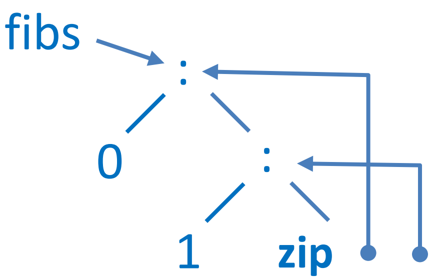
Lazy data structure

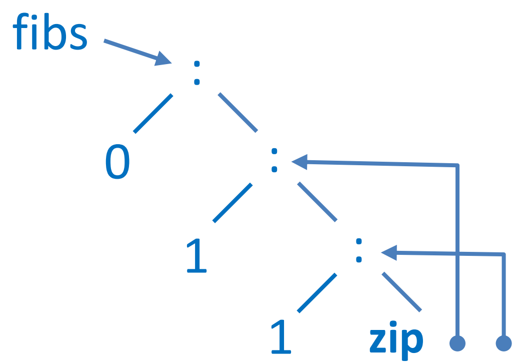
- Cyclic data structure

```
fibs = 0 : 1 : [ x+y | (x,y) <- zip fibs (tail fibs) ]
```

```
zip [] _ = []
zip _ [] = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

fibs	0 1 1 2 3 5 …
tail fibs +	1 1 2 3 5 8 …
	1 2 3 5 8 13…





Strictness

- Big unevaluated graph

sum [] = 0

sum (x:xs) = x+sum xs

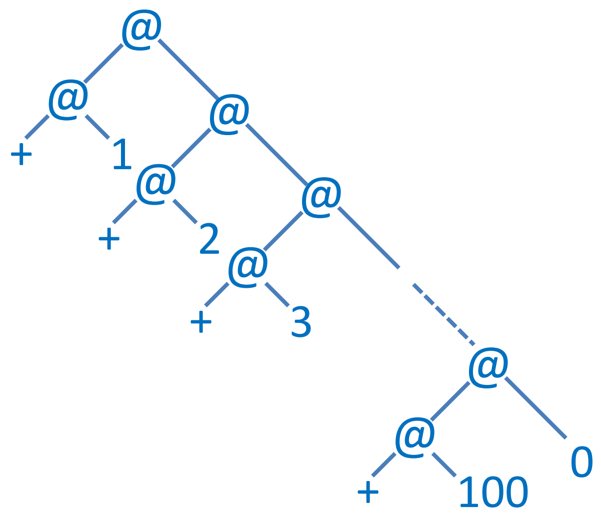
sum [1..100]

= 1+sum [2..100]

= 1+2+sum [3..100]

= ...

= 1+2+...+100+0



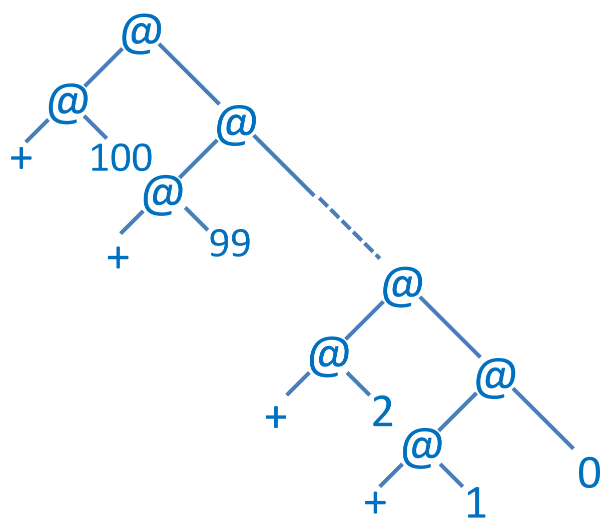
This takes O(n) time and O(n) space.

Strictness

- Big unevaluated graph

```
sum xs = sum xs 0
            where sum [] a = a
                  sum (x:xs) a= sum xs (x+a)
```

```
sum [1..100]
= sum [1..100] 0
= sum [2..100] (1+0)
= sum [3..100] (2+1+0)
= ...
= 100+...+2+1+0
```



This also takes O(n) time and O(n) space.

Strictness

- Sequence

  seq e1 e2

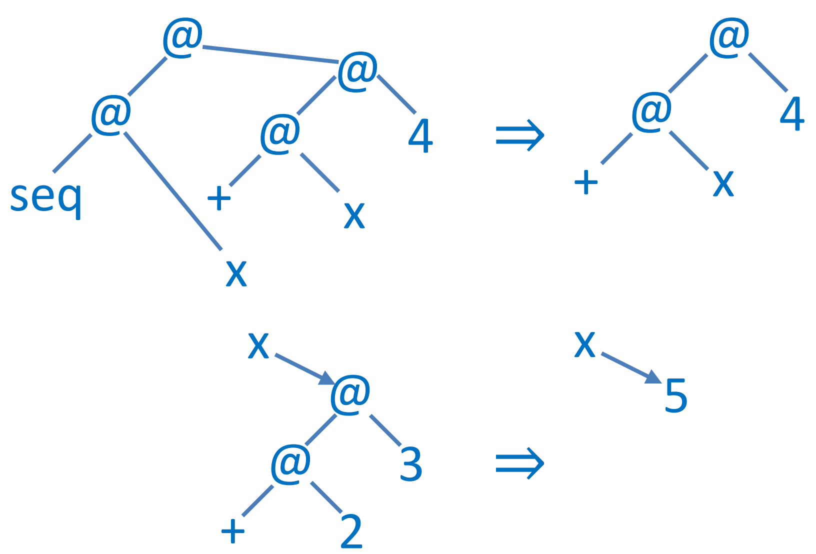
  seq is a built-in function that is strict in its first argument.

  seq $\perp$ e2 = $\perp$

  seq e1 e2 = e2, if e1 $\neq \perp$

  Hugs> seq x x+4 where x=2+3

  9 :: integer



  N.B. where and let are descriptions of graphs.

Strictness

- Strictness

```
f $! x = seq x (f x)
```

$! is a built-in infix operator that forces f to be a strict function

```
sum xs = sum xs 0
        where sum [] a = a
              sum (x:xs) a = sum xs $! x+a
                             -- ($!) (sum xs) (x+a)
```

N.B. $! has the lowest precedence

```
sum [1..100]
= sum [1..100] 0
= sum [2..100] 1
= sum [3..100] 3 = …
```

This takes O(n) time and O(1) space.

Full laziness

- Constant Application Form (CAF)

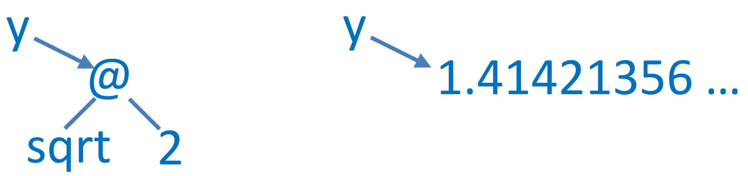
  f x = x + sqrt 2

  Without full laziness, the CAF sqrt 2 will be evaluated each time f is called.

  With full laziness, the function f will be compiled to

  f x = x + y where y = sqrt 2

  Thus, sqrt 2 will only be evaluated the first time f is called.



Full laziness

- Big reduction result

```
take 0 _ = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs
```

```
take = \n-> \xs -> if n==0 then []
                    else if xs==[] then []
                    else head xs : take (n-1) (tail xs)
```

```
take100 = take 100
```

```
Hugs> take100 [1..]
```

Firstly, this forces take 100 to be reduced, and take100 to be overwritten as, due to full laziness:

```
take100 = \xs -> if b100 then []
                 else if xs==[] then []
                 else head xs : take99 (tail xs)
                 where b100 = 100==0; take99 = take (100-1)
```

Full laziness

- Big reduction result

Secondly, the result is applied to [1..], forcing the CAF's to be reduced, and b100 and take99 to be overwritten as:

```
b100 = False
take99 = \xs -> if b99 then []
                else if xs==[] then []
                else head xs : take98 (tail xs)
                where b99 = 99==0; take98 = take (99-1)
```

Thus, as the reduction progresses, the graph will become bigger and bigger:

```
take100        take99        .....    take0
   ↓              ↓                      ↓
\xs -> if …    \xs -> if …    .....    \xs -> if …
```

Full laziness

- Big reduction result

  To solve this problem, we shall define take100 as

  ```
  take100 = takeA 100
            where takeA n xs = takeB xs n
                  takeB xs n = take n xs
  ```

  Hugs> take100 [1..]

  This forces takeA 100 to be reduced, and take100 to be overwritten as:

  take100 = \xs -> takeB xs 100

  Note that this is no CAF in the λ expression bound to take100

Space behavior of lazy functional programs

- Space behavior of lazy functional programs
  1) Not performing reductions, but holding the unevaluated graph, which is bigger than the result, e.g.

     sum [] = 0

     sum (x:xs) = x+sum xs
  2) Performing reductions and holding the result, which is bigger than the redex (space leak), e.g.

     take100 = take 100