A Functional Space Model

COS 326
Andrew W. Appel
Princeton University

Space

Understanding the space complexity of functional programs

- At least two interesting components:
  - the amount of *live space* at any instant in time
  - the *rate of allocation*
    - a function call may not change the amount of live space by much but may allocate at a substantial rate
    - because functional programs act by generating new data structures and discarding old ones, they often allocate a lot
      - » OCaml garbage collector is optimized with this in mind
      - » interesting fact: at the assembly level, the number of writes by a functional program is roughly the same as the number of writes by an imperative program

Space

Understanding the space complexity of functional programs

- At least two interesting components:
  - the amount of *live space* at any instant in time
  - the *rate of allocation*
    - a function call may not change the amount of live space by much but may allocate at a substantial rate
    - because functional programs act by generating new data structures and discarding old ones, they often allocate a lot
      - » OCaml garbage collector is optimized with this in mind
      - » interesting fact: at the assembly level, the number of writes by a functional program is roughly the same as the number of writes by an imperative program
- *What takes up space*?
  - conventional first-order data: tuples, lists, strings, datatypes
  - function representations (closures)
  - the call stack

CONVENTIONAL DATA

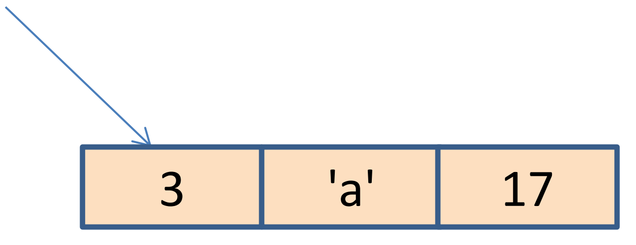
OCaml Representations for Data Structures

Type:

```
type triple = int * char * int
```

Representation:

(3, 'a', 17)



OCaml Representations for Data Structures

Type:

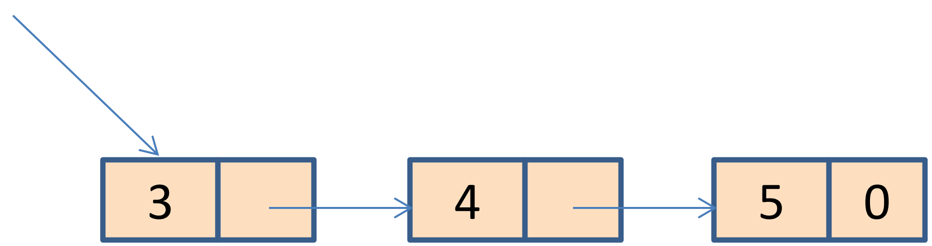
```
type mylist = int list
```

Representation:

[ ]

[3; 4; 5]

0



Space Model

Type:

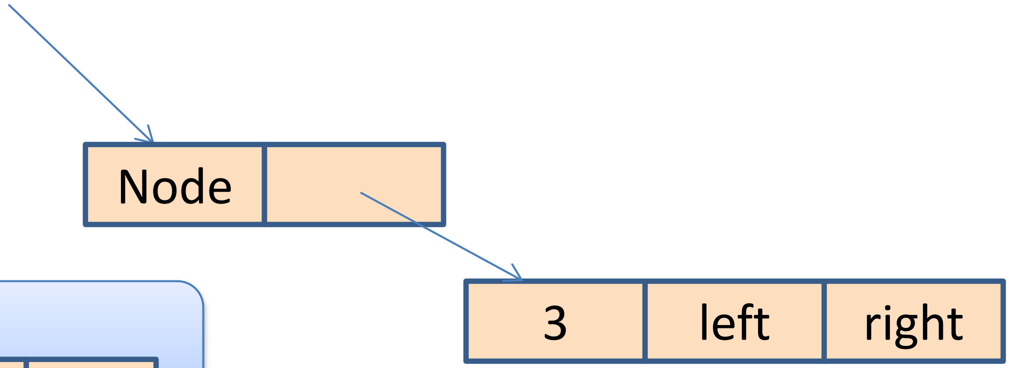
```
type tree = Leaf | Node of int * tree * tree
```

Representation:

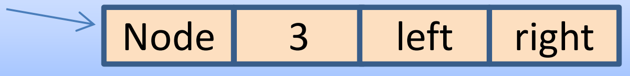
Leaf

Node(3, left, right)

0



Actually like this in Ocaml:



Allocating space

In C, you allocate when you call "malloc"

In Java, you allocate when you call "new"

What about ML?

Allocating space

Whenever you *use a constructor*, space is allocated:

```
let rec insert (t:tree) (i:int) =
  match t with
    Leaf -> Node (i, Leaf, Leaf)
  | Node (j, left, right) ->
      if i <= j then
        Node (j, insert left i, right)
      else
        Node (j, left, insert right i)
```

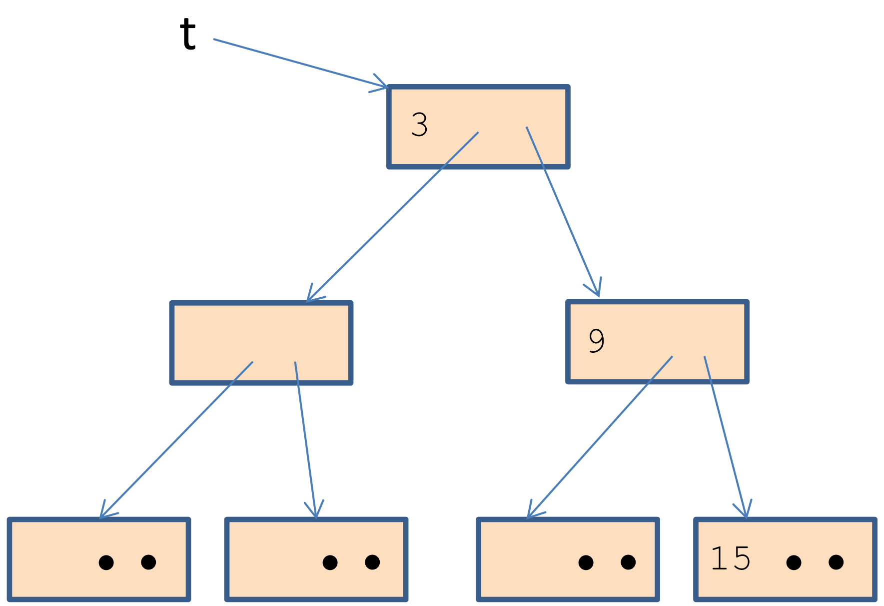
Allocating space

Whenever you use a constructor, space is allocated:

```
let rec insert (t:tree) (i:int) =
  match t with
    Leaf -> Node (i, Leaf, Leaf)
  | Node (j, left, right) ->
      if i <= j then
        Node (j, insert left i, right)
      else
        Node (j, left, insert right i)
```

Consider:

insert t 21



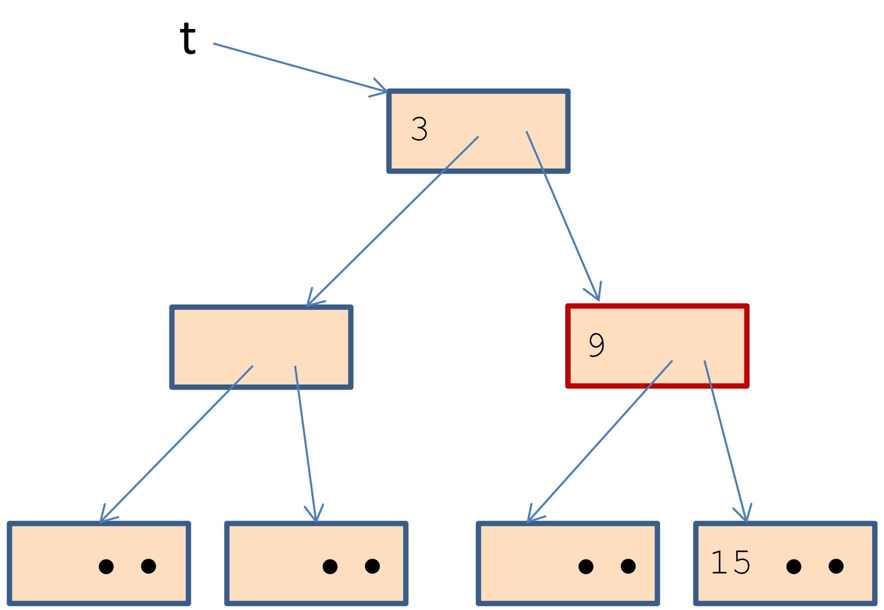
Allocating space

Whenever you use a constructor, space is allocated:

```
let rec insert (t:tree) (i:int) =
  match t with
    Leaf -> Node (i, Leaf, Leaf)
  | Node (j, left, right) ->
      if i <= j then
        Node (j, insert left i, right)
      else
        Node (j, left, insert right i)
```

Consider:

insert t 21



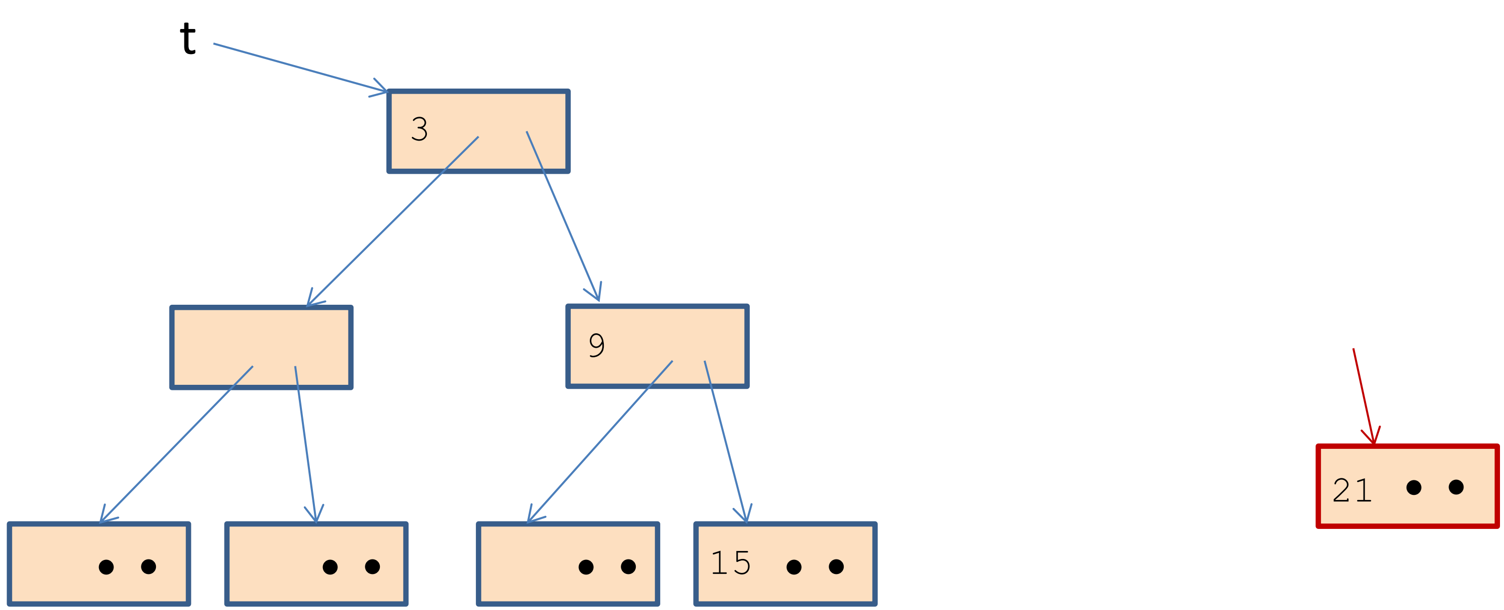
Allocating space

Whenever you use a constructor, space is allocated:

```
let rec insert (t:tree) (i:int) =
  match t with
    Leaf -> Node (i, Leaf, Leaf)
  | Node (j, left, right) ->
      if i <= j then
        Node (j, insert left i, right)
      else
        Node (j, left, insert right i)
```

Consider:

insert t 21



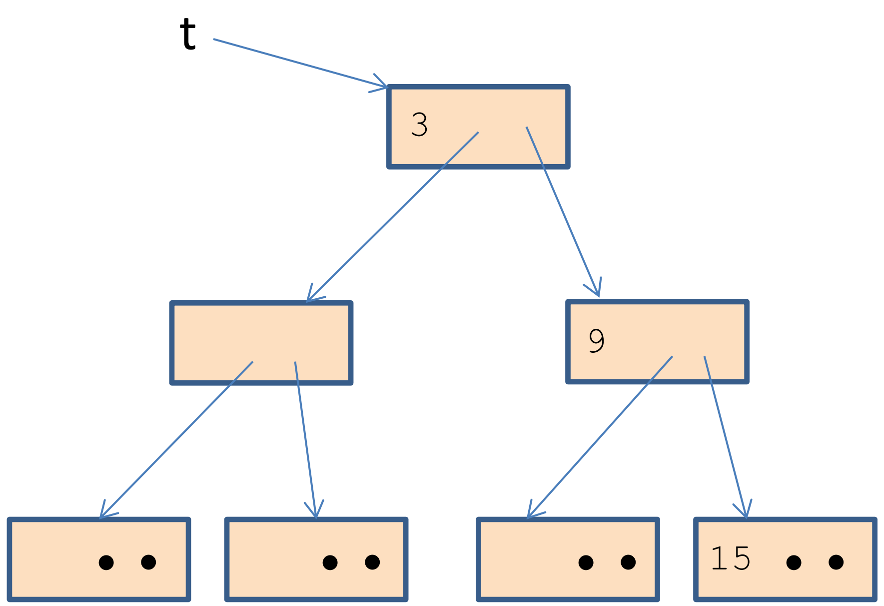
Allocating space

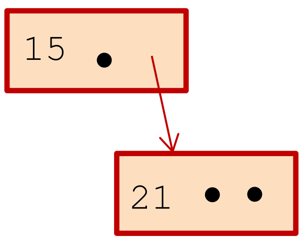
Whenever you use a constructor, space is allocated:

```
let rec insert (t:tree) (i:int) =
  match t with
    Leaf -> Node (i, Leaf, Leaf)
  | Node (j, left, right) ->
      if i <= j then
        Node (j, insert left i, right)
      else
        Node (j, left, insert right i)
```

Consider:

insert t 21





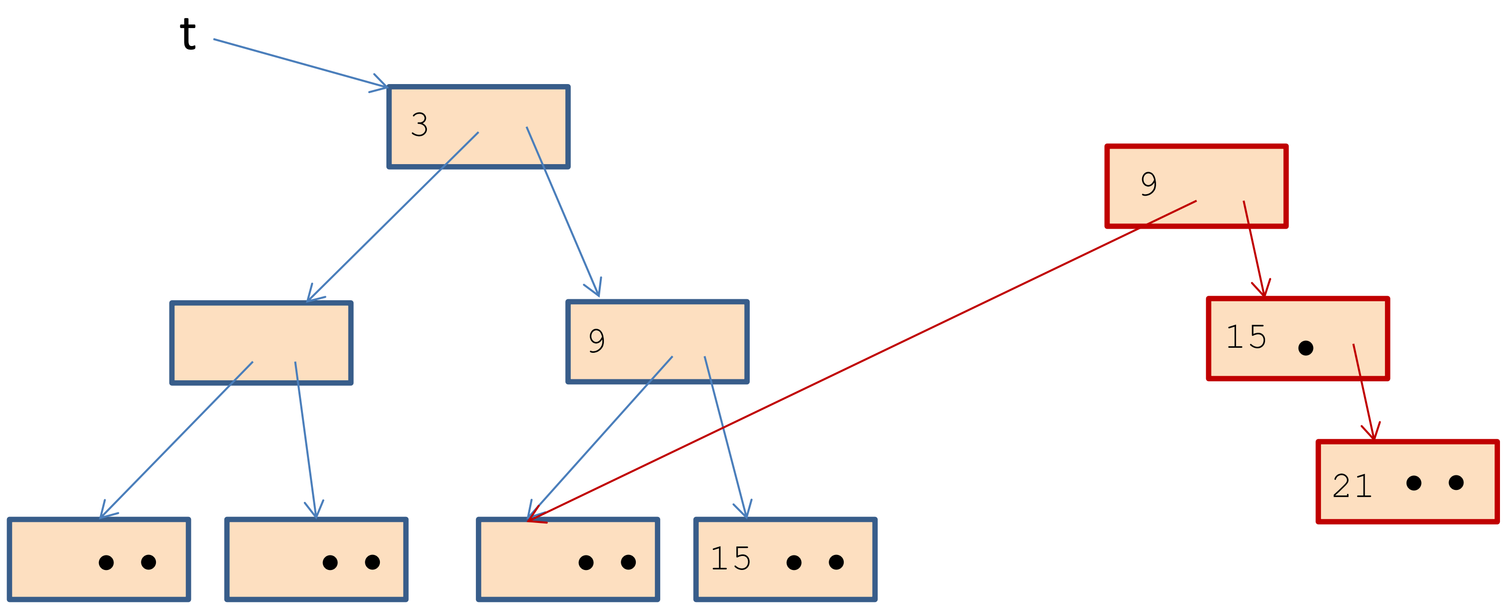
Allocating space

Whenever you use a constructor, space is allocated:

```
let rec insert (t:tree) (i:int) =
  match t with
    Leaf -> Node (i, Leaf, Leaf)
  | Node (j, left, right) ->
      if i <= j then
        Node (j, insert left i, right)
      else
        Node (j, left, insert right i)
```

Consider:

insert t 21



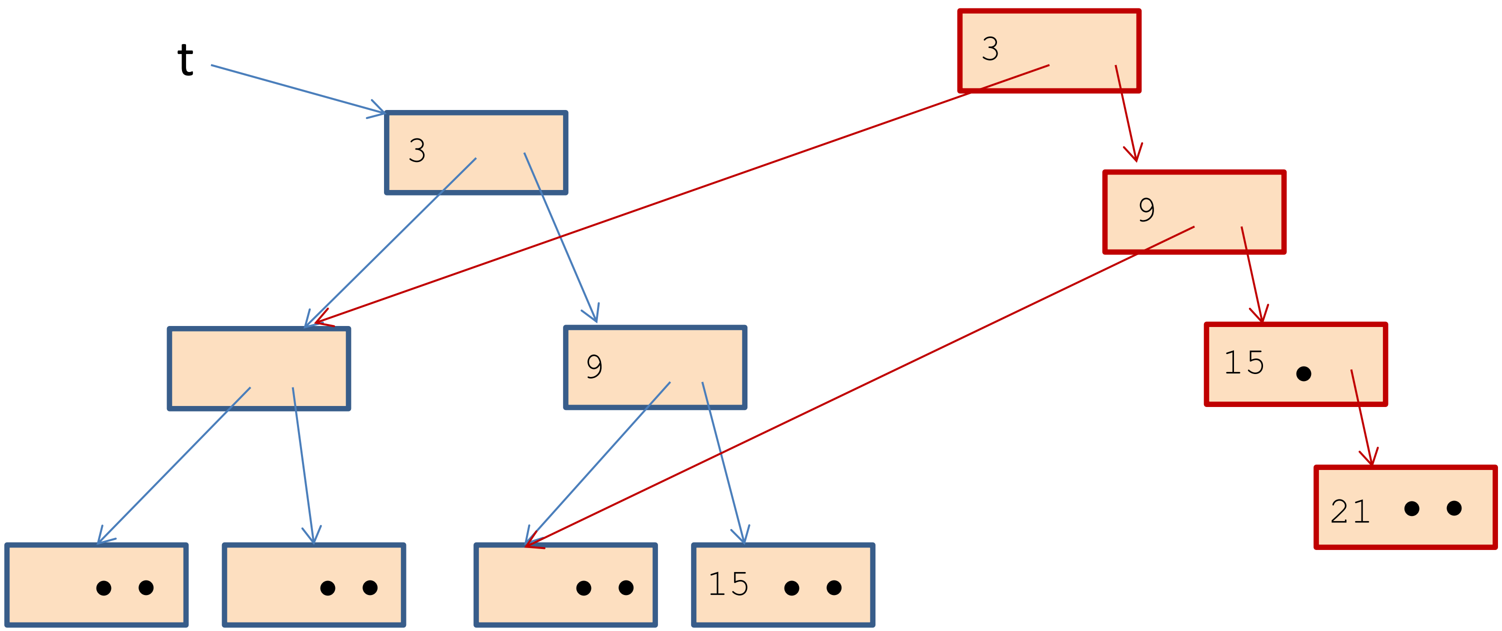
Allocating space

Whenever you use a constructor, space is allocated:

```
let rec insert (t:tree) (i:int) =
  match t with
    Leaf -> Node (i, Leaf, Leaf)
  | Node (j, left, right) ->
      if i <= j then
        Node (j, insert left i, right)
      else
        Node (j, left, insert right i)
```

Consider:

insert t 21



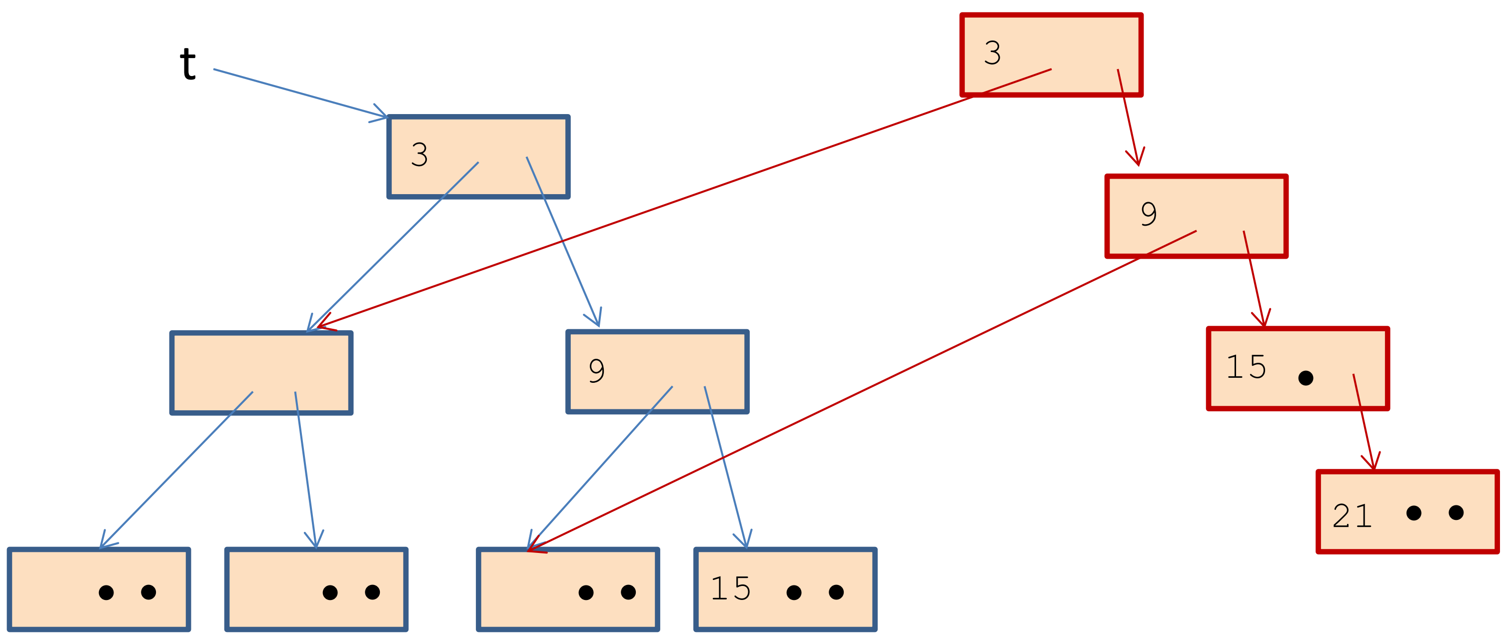
Allocating space

Whenever you use a constructor, space is allocated:

```
let rec insert (t:tree) (i:int) =
  match t with
    Leaf -> Node (i, Leaf, Leaf)
  | Node (j, left, right) ->
      if i <= j then
        Node (j, insert left i, right)
      else
        Node (j, left, insert right i)
```

Total space allocated is proportional to the height of the tree.

~ log n, if tree with n nodes is balanced



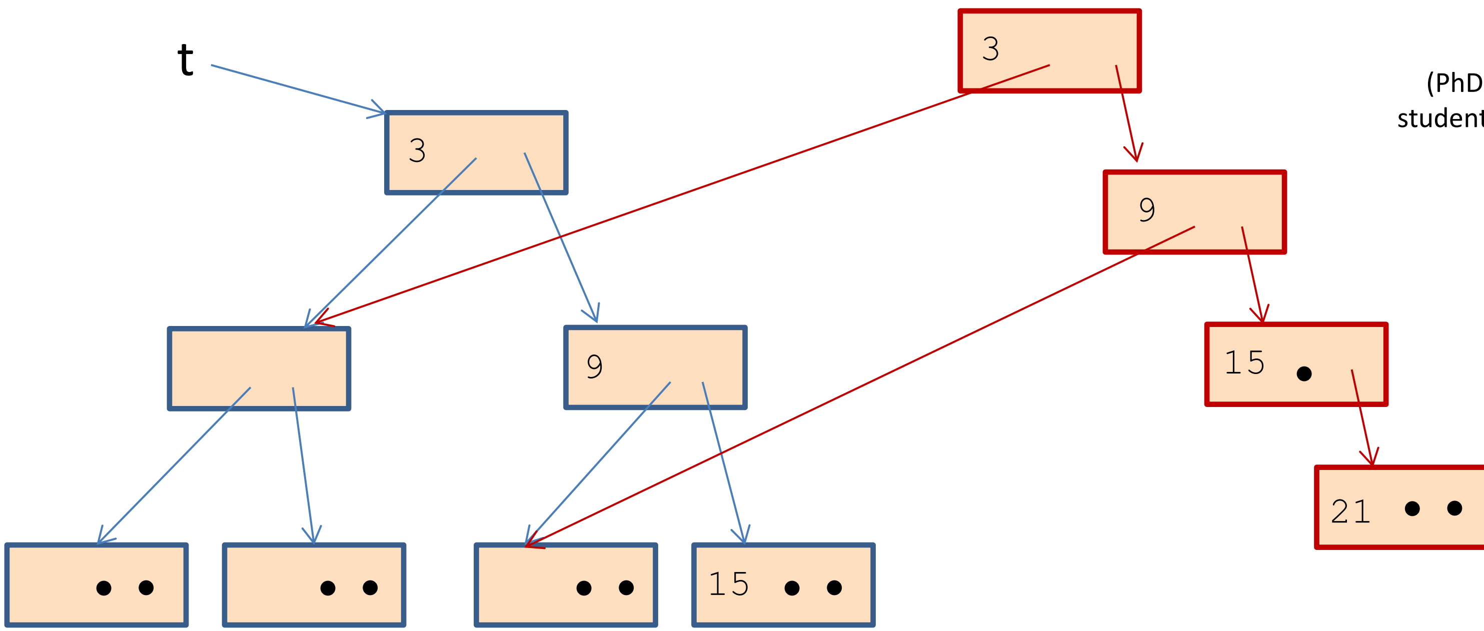
Net space allocated

The garbage collector reclaims unreachable data structures on the heap.

```
let fiddle (t: tree) =
  insert t 21
```

John McCarthy invented GC 1960

(PhD Princeton 1951, student of Alonzo Church)

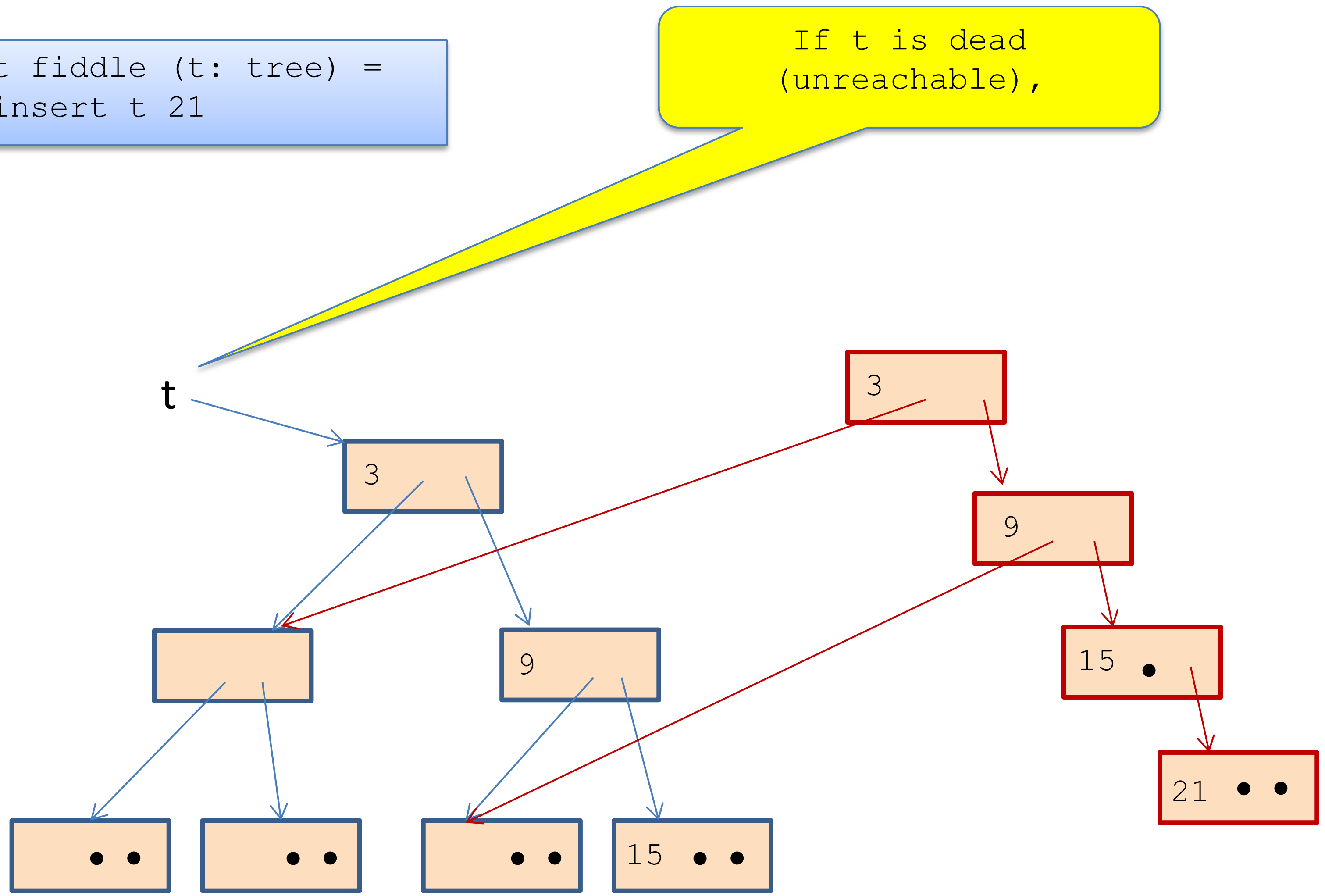


Net space allocated

The garbage collector reclaims

unreachable data structures on the heap.

```
let fiddle (t: tree) =
  insert t 21
```



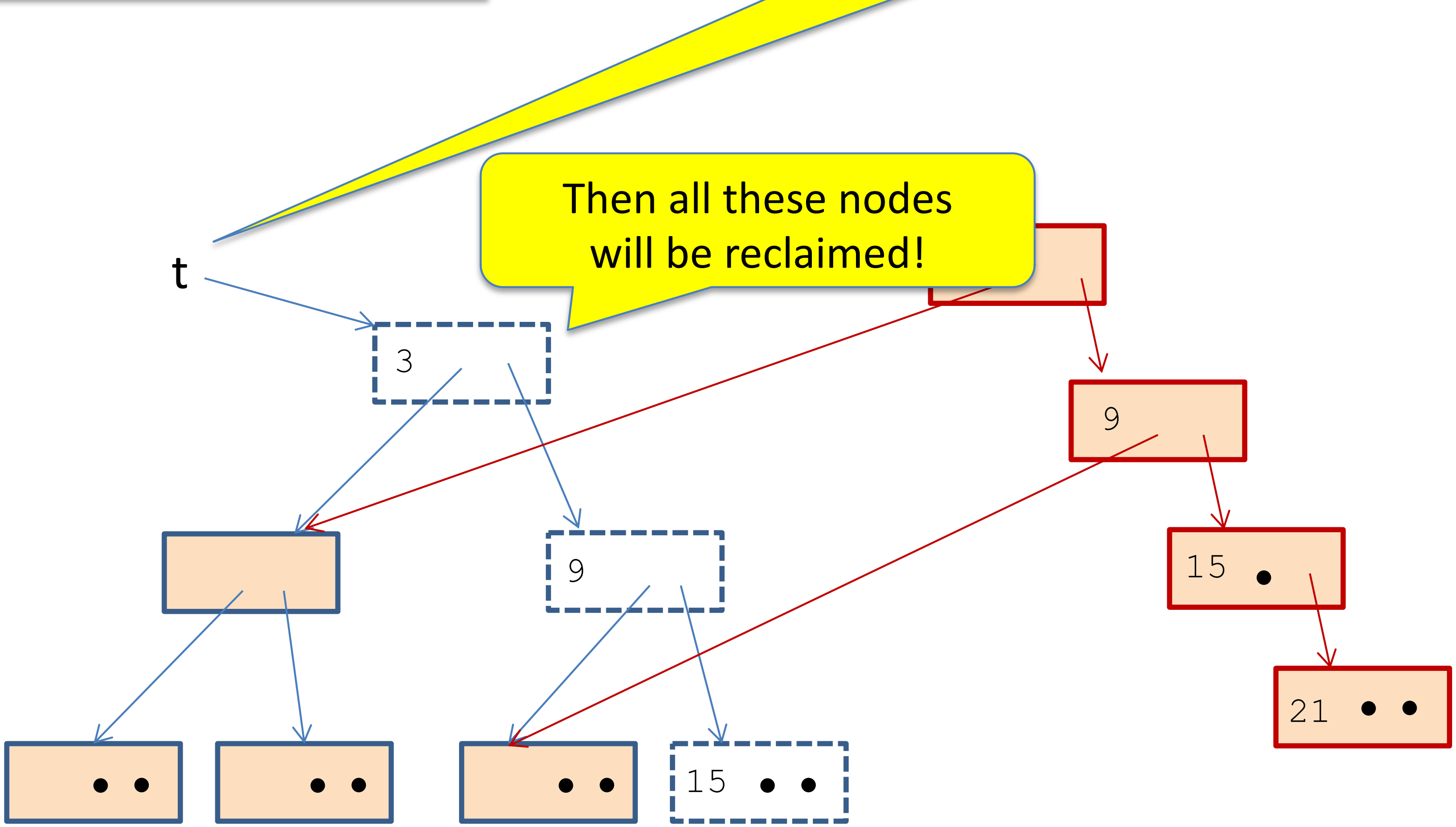
Net space allocated

The garbage collector reclaims

unreachable data structures on the heap.

```
let fiddle (t: tree) =
  insert t 21
```

If t is dead (unreachable),



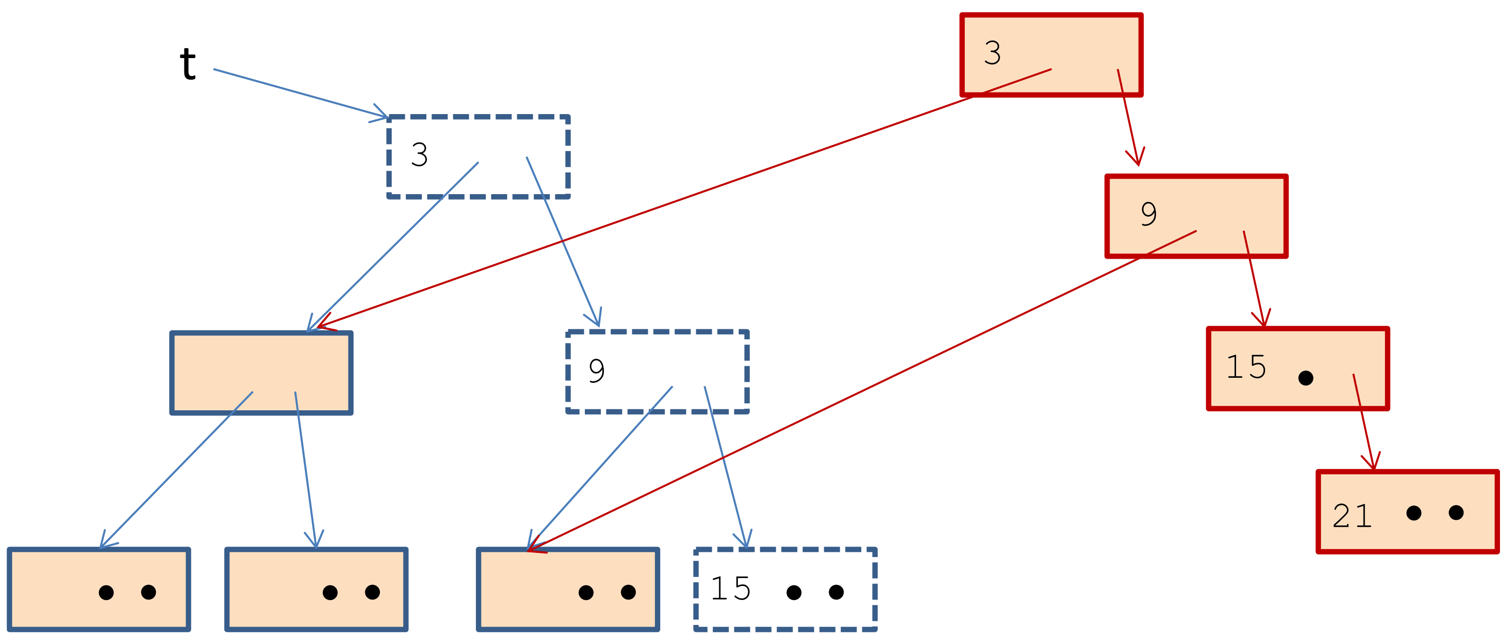
Net space allocated

The garbage collector reclaims unreachable data structures on the heap.

```
let fiddle (t: tree) =
  insert t 21
```

Net new space allocated:
1 node

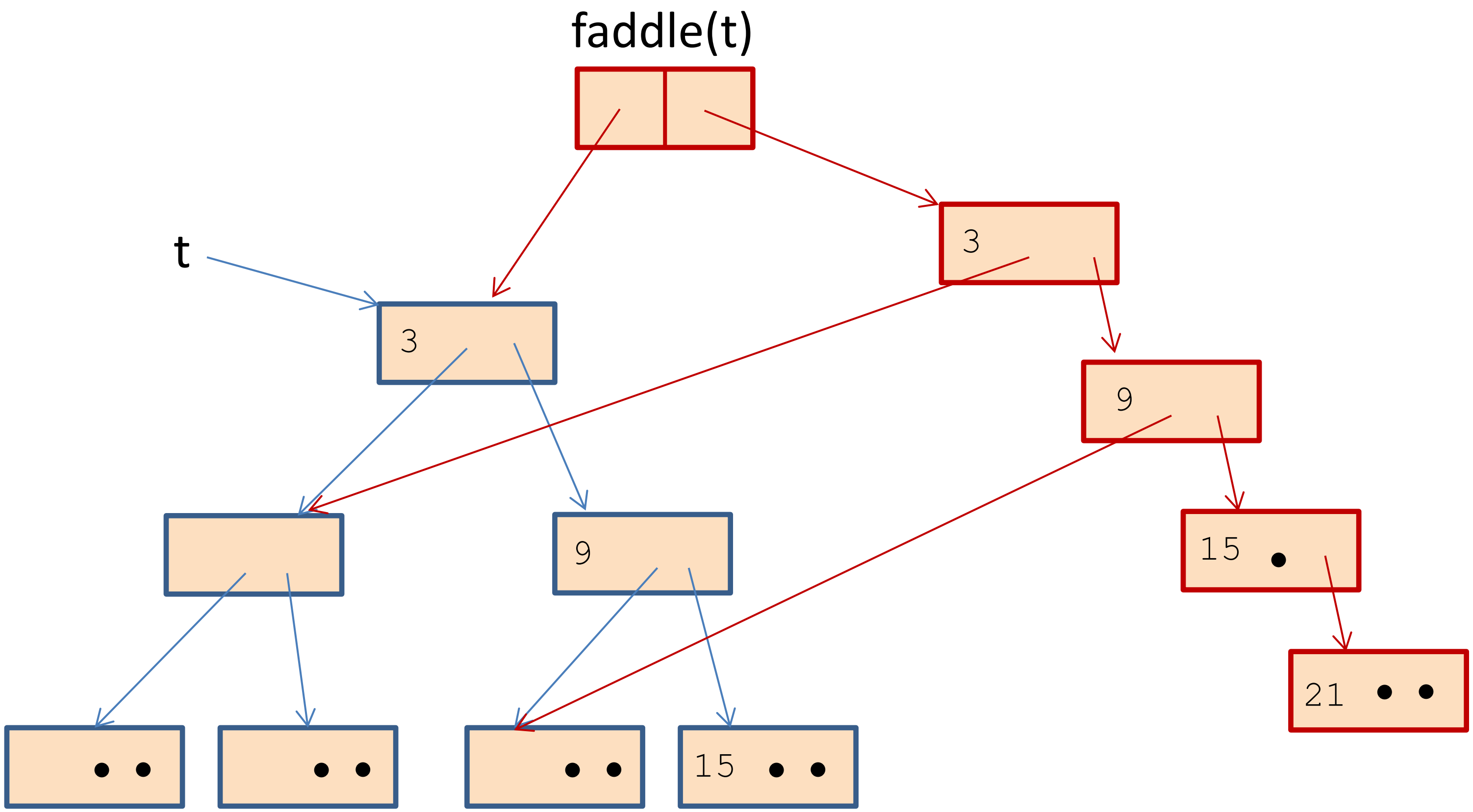
(just like “imperative” version of binary search trees)



Net space allocated

But what if you want to keep the old tree?

```
let faddle (t: tree) =
  (t, insert t 21)
```



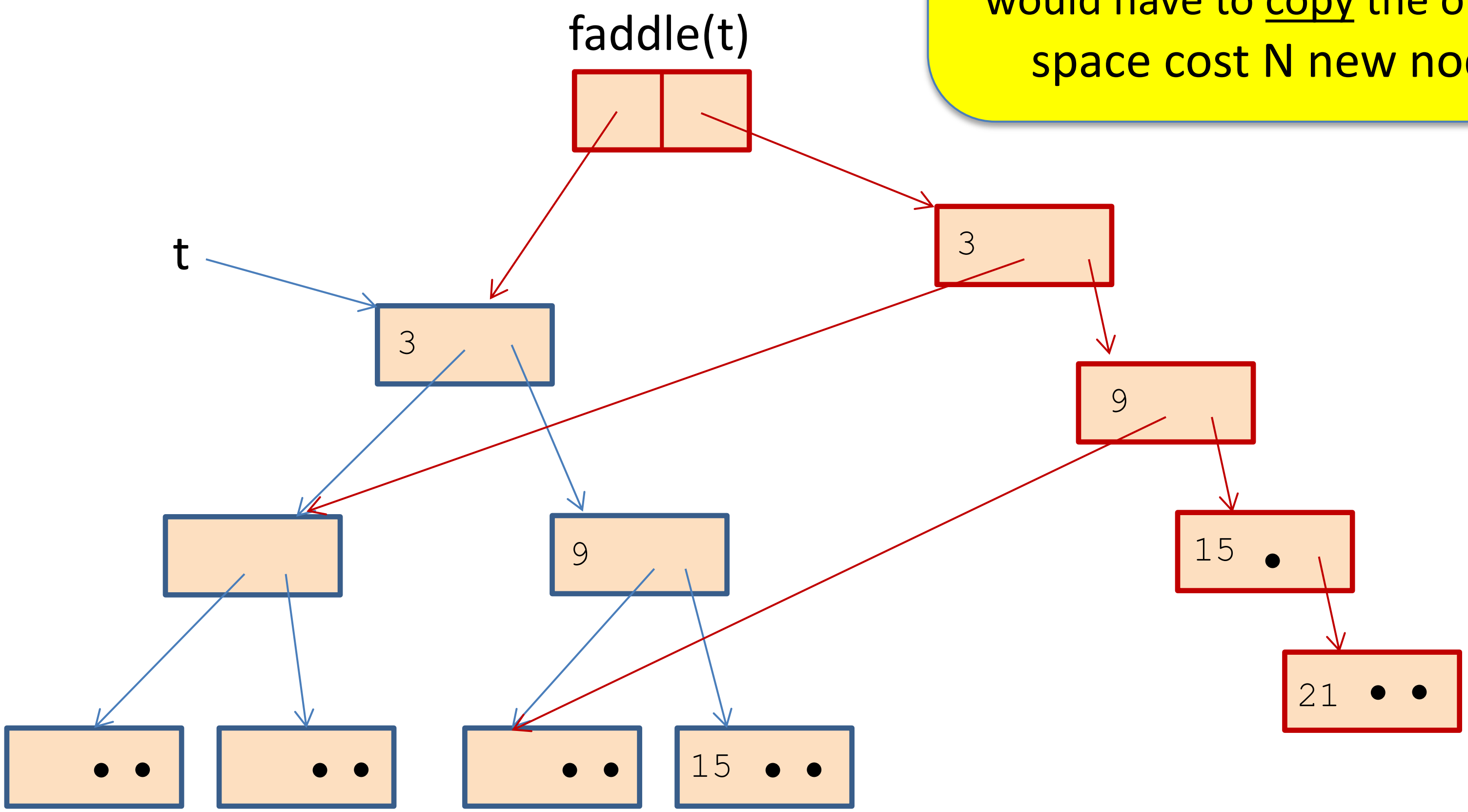
Net space allocated

But what if you want to keep the old tree?

```
let faddle (t: tree) =
  (t, insert t 21)
```

Net new space allocated:
log(N) nodes

but note: "imperative" version would have to copy the old tree, space cost N new nodes!



Compare


```
let check_option (o:int option) : int option =
  match o with
    Some _ -> o
  | None -> failwith “found none”
```

```
let check_option (o:int option) : int option =
  match o with
    Some j -> Some j
  | None -> failwith “found none”
```

Compare


```
let check_option (o:int option) : int option =
  match o with
    Some _ -> o
  | None -> failwith "found none"
```

allocates nothing when arg is Some i

```
let check_option (o:int option) : int option =
  match o with
    Some j -> Some j
  | None -> failwith "found none"
```

allocates an option when arg is Some i

Compare


```
let cadd (c1:int*int) (c2:int*int) : int*int =
  let (x1,y1) = c1 in
  let (x2,y2) = c2 in
  (x1+x2, y1+y2)
```

```
let double (c1:int*int) : int*int =
  let c2 = c1 in
  cadd c1 c2
```

```
let double (c1:int*int) : int*int =
  cadd c1 c1
```

```
let double (c1:int*int) : int*int =
  let (x1,y1) = c1 in
  cadd (x1,y1) (x1,y1)
```

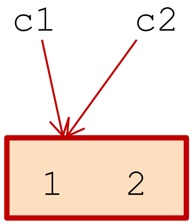
Compare

```
let cadd (c1:int*int) (c2:int*int) : int*int =
  let (x1,y1) = c1 in
  let (x2,y2) = c2 in
  (x1+x2, y1+y2)
```

```
let double (c1:int*int) : int*int =
  let c2 = c1 in
  cadd c1 c2
```

```
let double (c1:int*int) : int*int =
  cadd c1 c1
```

```
let double (c1:int*int) : int*int =
  let (x1,y1) = c1 in
  cadd (x1,y1) (x1,y1)
```

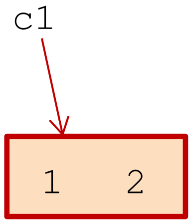


Compare


```
let cadd (c1:int*int) (c2:int*int) : int*int =
  let (x1,y1) = c1 in
  let (x2,y2) = c2 in
  (x1+x2, y1+y2)
```

```
let double (c1:int*int) : int*int =
  let c2 = c1 in
  cadd c1 c2
```

```
let double (c1:int*int) : int*int =
  cadd c1 c1
```




```
let double (c1:int*int) : int*int =
  let (x1,y1) = c1 in
  cadd (x1,y1) (x1,y1)
```

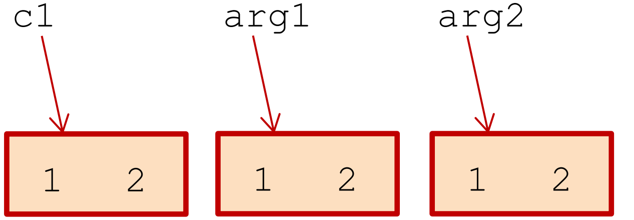
Compare


```
let cadd (c1:int*int) (c2:int*int) : int*int =
  let (x1,y1) = c1 in
  let (x2,y2) = c2 in
  (x1+x2, y1+y2)
```

```
let double (c1:int*int) : int*int =
  let c2 = c1 in
  cadd c1 c2
```

```
let double (c1:int*int) : int*int =
  cadd c1 c1
```

```
let double (c1:int*int) : int*int =
  let (x1,y1) = c1 in
  cadd (x1,y1) (x1,y1)
```



Compare


```
let cadd (c1:int*int) (c2:int*int) : int*int =
  let (x1,y1) = c1 in
  let (x2,y2) = c2 in
  (x1+x2, y1+y2)
```

```
let double (c1:int*int) : int*int =
  let c2 = c1 in
  cadd c1 c2
```

no allocation here
(1 pair allocated in cadd)

```
let double (c1:int*int) : int*int =
  cadd c1 c1
```

no allocation here
(1 pair allocated in cadd)

```
let double (c1:int*int) : int*int =
  let (x1,y1) = c1 in
  cadd (x1,y1) (x1,y1)
```

allocates 2 pairs here
(unless the compiler happens to optimize…)

Compare


```
let cadd (c1:int*int) (c2:int*int) : int*int =
  let (x1,y1) = c1 in
  let (x2,y2) = c2 in
  (x1+x2, y1+y2)
```

```
let double (c1:int*int) : int*int =
  let (x1,y1) = c1 in
  cadd c1 c1
```

double does not allocate

extracts components: it is a read

FUNCTION CLOSURES

Closures (A reminder)

Nested functions like bar often contain free variables:

```
let foo y =
  let bar x = x + y in
  bar
```

Here's bar on its own:

```
let bar x = x + y
```

y is *free* in the definition of bar

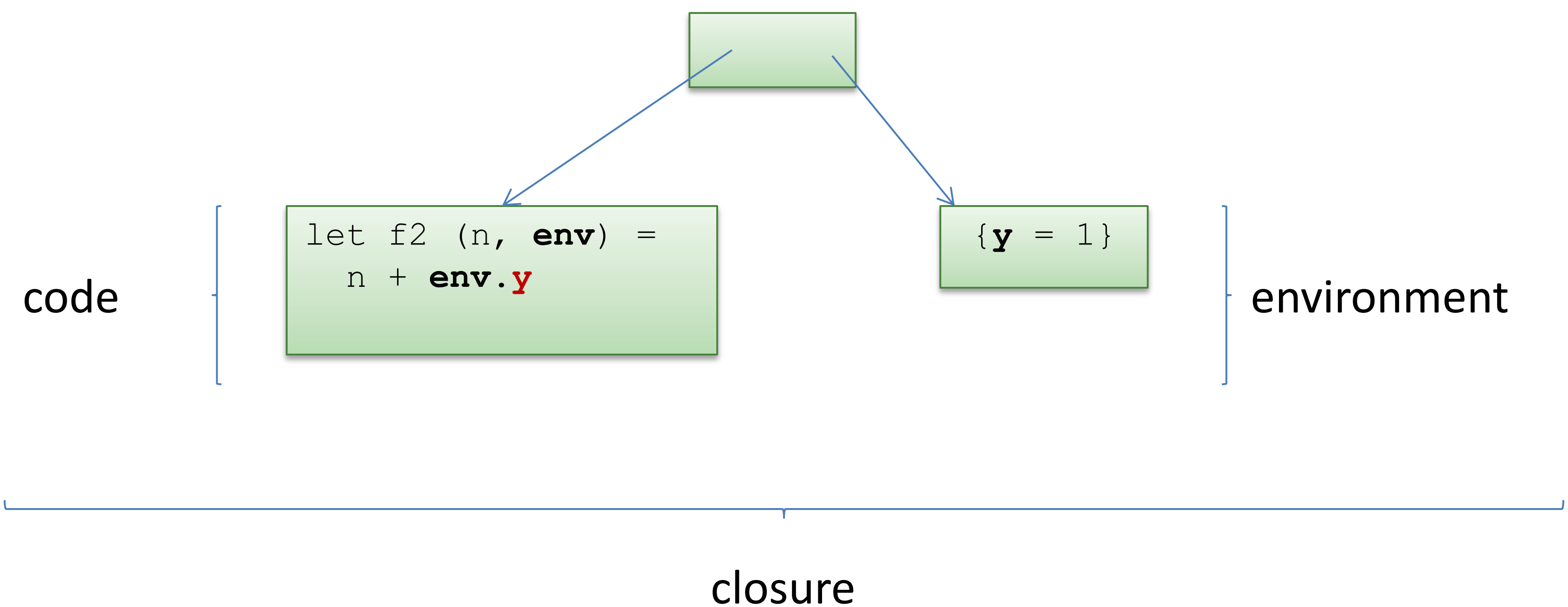
To implement bar, the compiler creates a *closure*, which is a pair of code for the function plus an environment holding the free variables.

But what about nested, higher-order functions?

bar again:

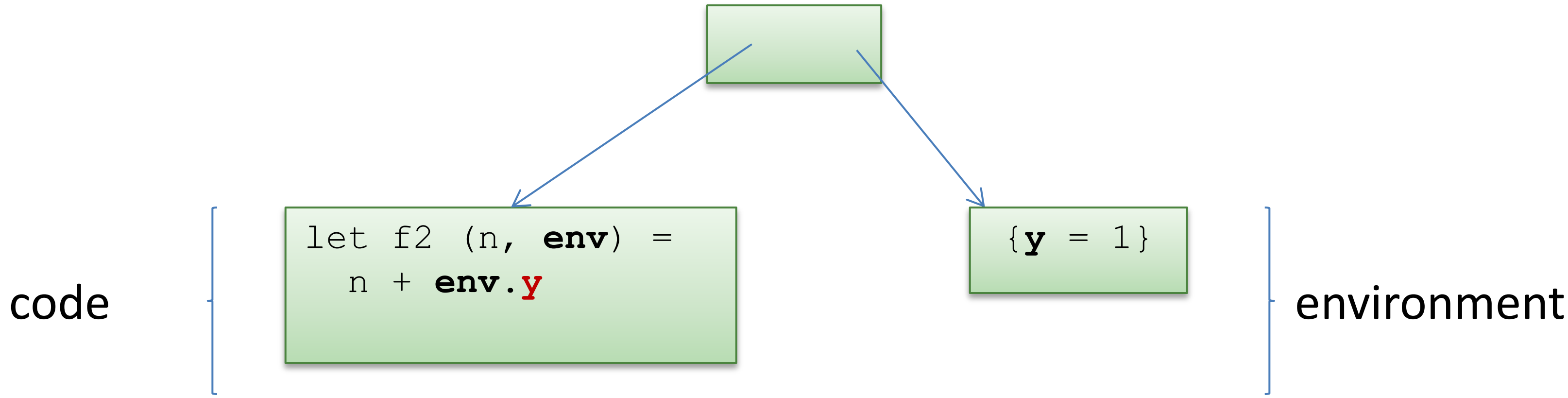
```
let bar x = x + y
```

bar's representation:



But what about nested, higher-order functions?

To estimate the (heap) space used by a program, we often need to estimate the (heap) space used by its closures.



Our estimate will include the cost of the pair:

- two pointers = 2 words (8 bytes each, or 4 bytes each on some machines)
- the cost of the environment (1 word in this case).
- but not: the cost of the code (because the same code is reused in every closure of this function)

Space Model Summary

Understanding space consumption in FP involves:

- understanding the difference between
  - live space
  - rate of allocation
- understanding where allocation occurs
  - any time a constructor is used
  - whenever closures are created
- understanding the costs of
  - data types (fairly similar to Java)
  - costs of closures (pair + environment)

A remark about homework 4

WHY IT'S IMPORTANT TO PRUNE CLOSURE ENVIRONMENTS

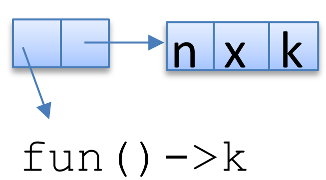
Pruning environments

```
let zeros i = if i=0 then [] else 0 :: s(i-1)

let h (n: int) : int =
  let f x =
    let k = List.length x in
    fun () -> k
  in
  let rec g i : (unit->int) list =
    if i=0 then [] else f (zeros n) :: g (i-1)
  in let bigdata = g n
  in List.fold_left (fun s u -> u()+s) 0 bigdata

let a = h 1000
```

Pruning environments

```
let zeros i = if i=0 then [] else 0 :: s(i-1)

let h (n: int) : int =
  let f x =
    let k = List.length x in
    fun () -> k
  in
  let rec g i : (unit->int) list =
    if i=0 then [] else f (zeros n) :: g (i-1)
  in let bigdata = g n
  in List.fold_left (fun s u -> u()+s) 0 bigdata

let a = h 1000
```

What variables are *in scope at this point* ?



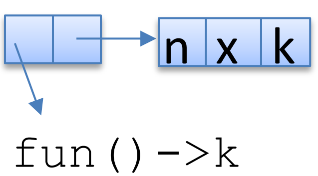
You *could* build a closure environment with all the variables currently in scope.

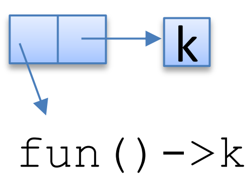
Pruning environments

```
let zeros i = if i=0 then [] else 0 :: s(i-1)

let h (n: int) : int =
  let f x =
    let k = List.length x in
    fun () -> k
  in
  let rec g i : (unit->int) list =
    if i=0 then [] else f (zeros n) :: g (i-1)
  in let bigdata = g n
  in List.fold_left (fun s u -> u()+s) 0 bigdata

let a = h 1000
```

What are the free variables of this function?

5 words of memory versus 3 words, what's the big deal?

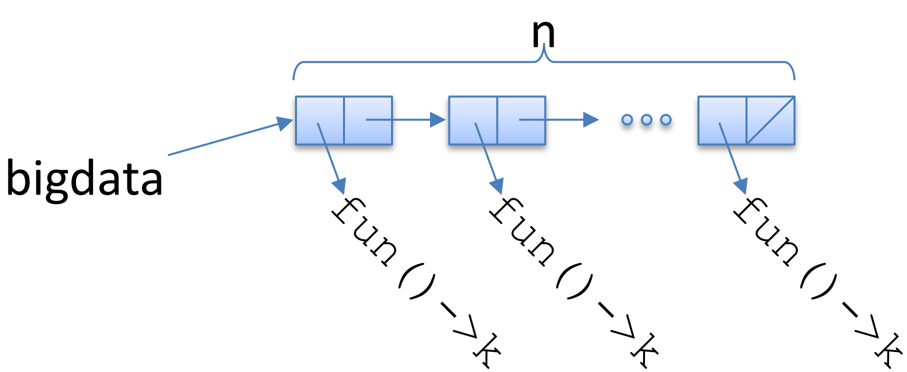




Pruning environments

```
let zeros i = if i=0 then [] else 0 :: s(i-1)

let h (n: int) : int =
  let f x =
    let k = List.length x in
    fun () -> k
  in
  let rec g i : (unit->int) list =
    if i=0 then [] else f (zeros n) :: g (i-1)
  in let bigdata = g n
  in List.fold_left (fun s u -> u()+s) 0 bigdata

let a = h 1000
```

Run the program to here, and what is in memory?

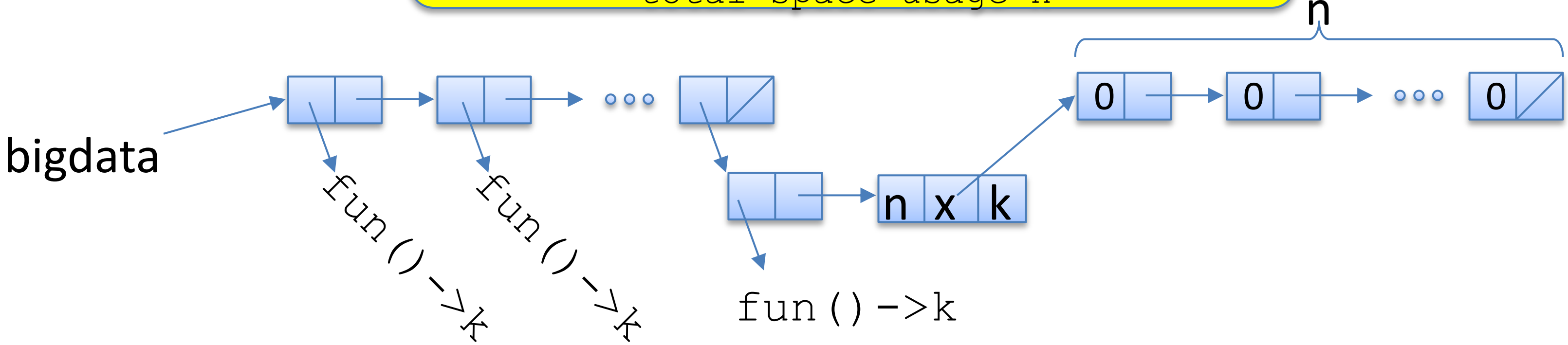


Pruning environments

```
let zeros i = if i=0 then [] else 0 :: s(i-1)

let h (n: int) : int =
  let f x =
    let k = List.length x in
    fun () -> k
  in
  let rec g i : (unit->int) list =
    if i=0 then [] else f (zeros n) :: g (i-1)
  in let bigdata = g n
  in List.fold_left (fun s u -> u()+s) 0 bigdata

let a = h 1000
```

What variables are *in scope at this point* ?

n closures for (fun()->k),
each is a list of length n,
total space usage n^2

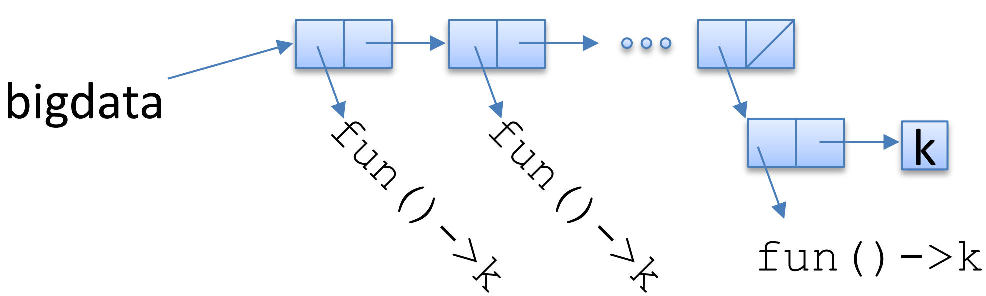


Pruning environments

```
let zeros i = if i=0 then [] else 0 :: s(i-1)

let h (n: int) : int =
  let f x =
    let k = List.length x in
    fun () -> k
  in
  let rec g i : (unit->int) list =
    if i=0 then [] else f (zeros n) :: g (i-1)
  in let bigdata = g n
  in List.fold_left (fun s u -> u()+s) 0 bigdata

let a = h 1000
```

What are the free variables of this function?

```
n closures for (fun()->k),
each is just a number k,
total space usage O(n)
```



Therefore

Closures should represent *only* the free variables of a function (not *all the variables currently in scope),*

otherwise the compiled program may use *asymptotically more space,*

such as $O(n^2)$ instead of $O(n)$