

Programming Languages and Techniques (CIS120)

Lecture 14

October 2, 2017

ASM & Equality

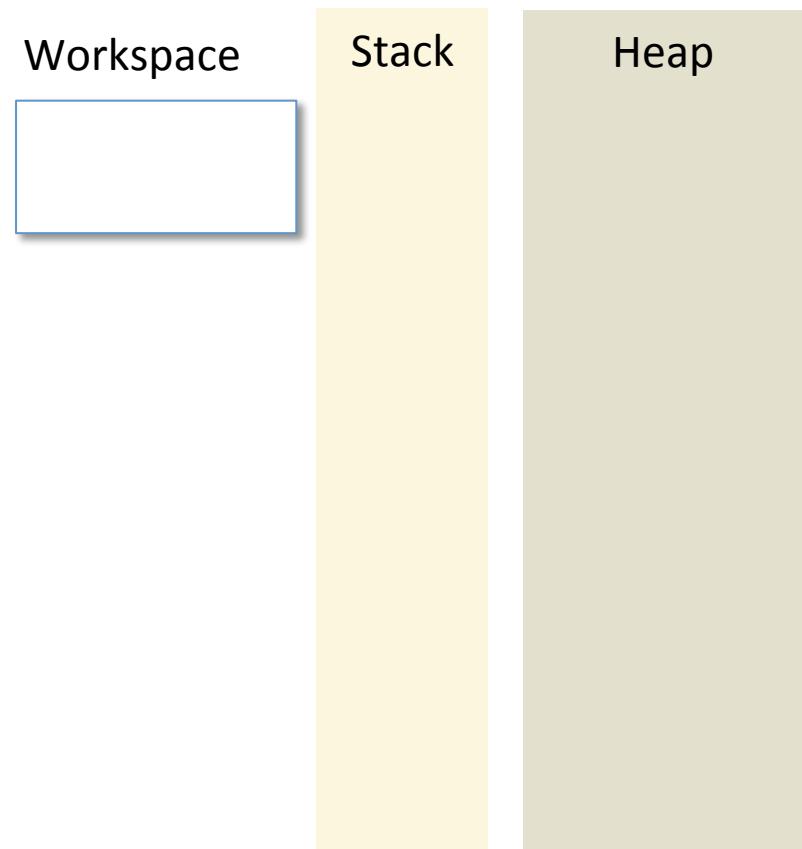
Lecture notes: Chapter 16

Announcements

- No recitation sections this week (Fall Break!)
- Dr. Zdancewic will cover the noon lecture on Weds.
- Homework 4
 - due on October 10th
- Midterm 1
 - *October 13th in Class*
 - Where? Last Names:
 - A – M Leidy Labs 10 (Here)
 - N – Z Meyerson Hall B1
 - Covers lecture material through Chapter 13
 - Review materials (old exams) on course website
- **Review Session:**
 - Wednesday, Oct. 11th 6:00-8:00pm, Towne 100

Abstract Stack Machine

- Three “spaces”
 - workspace
 - the expression the computer is currently working on simplifying
 - stack
 - temporary storage for `let` bindings and partially simplified expressions
 - heap
 - storage area for large data structures
- Initial state:
 - workspace contains whole program
 - stack and heap are empty
- Machine operation:
 - In each step, choose next part of the workspace expression and simplify it
 - (Sometimes this will also involve changes to the stack and/or heap)
 - Stop when there are no more simplifications to be done



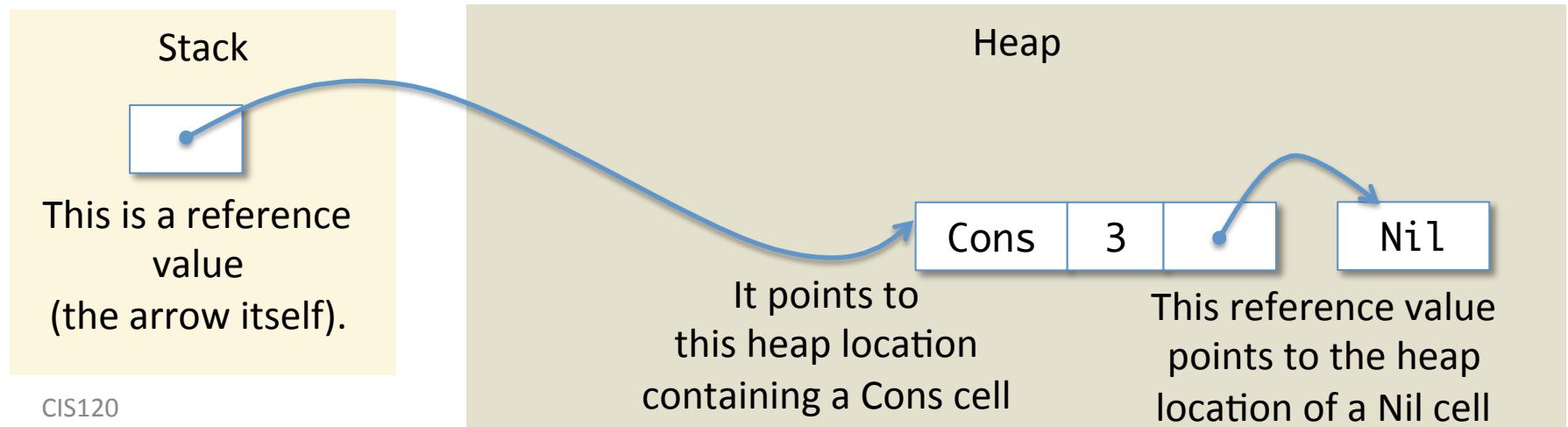
Values and References

A *value* is either:

- a *primitive value* like an integer, or,
- a *reference* to a location in the heap

A reference is the *address* of a piece of data in the heap. We draw a reference value as an “arrow”:

- The start of the arrow is the reference itself (i.e. the address).
- The arrow “points” to the value located at the reference’s address.



What answer does the following function produce when called?

```
let f (p1:point) (p2:point) : int =
  p1.x <- 17;
  let z = p1.x in
  p2.x <- 42;
  z
```

1. 17
2. 42
3. sometimes 17 and sometimes 42
4. f is ill typed

Answer: 17

Ah... Refs!

OCaml provides syntax for working with updatable *references*:

`type 'a ref = {mutable contents:'a}`

`ref e` \equiv `{contents = e}` has type `t ref` when $(e : t)$

`e1 := e2` \equiv `(e1).contents <- e2` has type `unit` when
 $(e1 : t \text{ ref})$ and $(e2 : t)$

`!e` \equiv `(e).contents` has type `t` when $(e : t \text{ ref})$

"is defined to be"
(not Ocaml syntax)

Ocaml
"syntactic sugar"

equivalent expressions

type constraints

Comparison To Java (or C, C++, ...)

Java

```
int f() {  
    int x = 3;  
    x = x + 1;  
    return x;  
}
```

OCaml

```
let f () : int =  
    let x = ref 3 in  
    x := !x + 1;  
    !x
```

- x has type int
- meaning on left of $=$ different than on right
- *implicit* dereference

- x has type (int ref)
- use $:=$ for update
- *explicit* dereference

Simplifying lists and datatypes using the heap

Simplification

Workspace

```
1::2::3::[]
```

Stack

Heap

For uniformity, we'll
pretend lists are declared
like this:

```
type 'a list =  
| Nil  
| Cons of 'a * 'a list
```

Simplification

Workspace

```
Cons (1,Cons (2,Cons (3,Nil)))
```

Stack

Heap

For uniformity, we'll
pretend lists are declared
like this:

```
type 'a list =  
| Nil  
| Cons of 'a * 'a list
```

Simplification

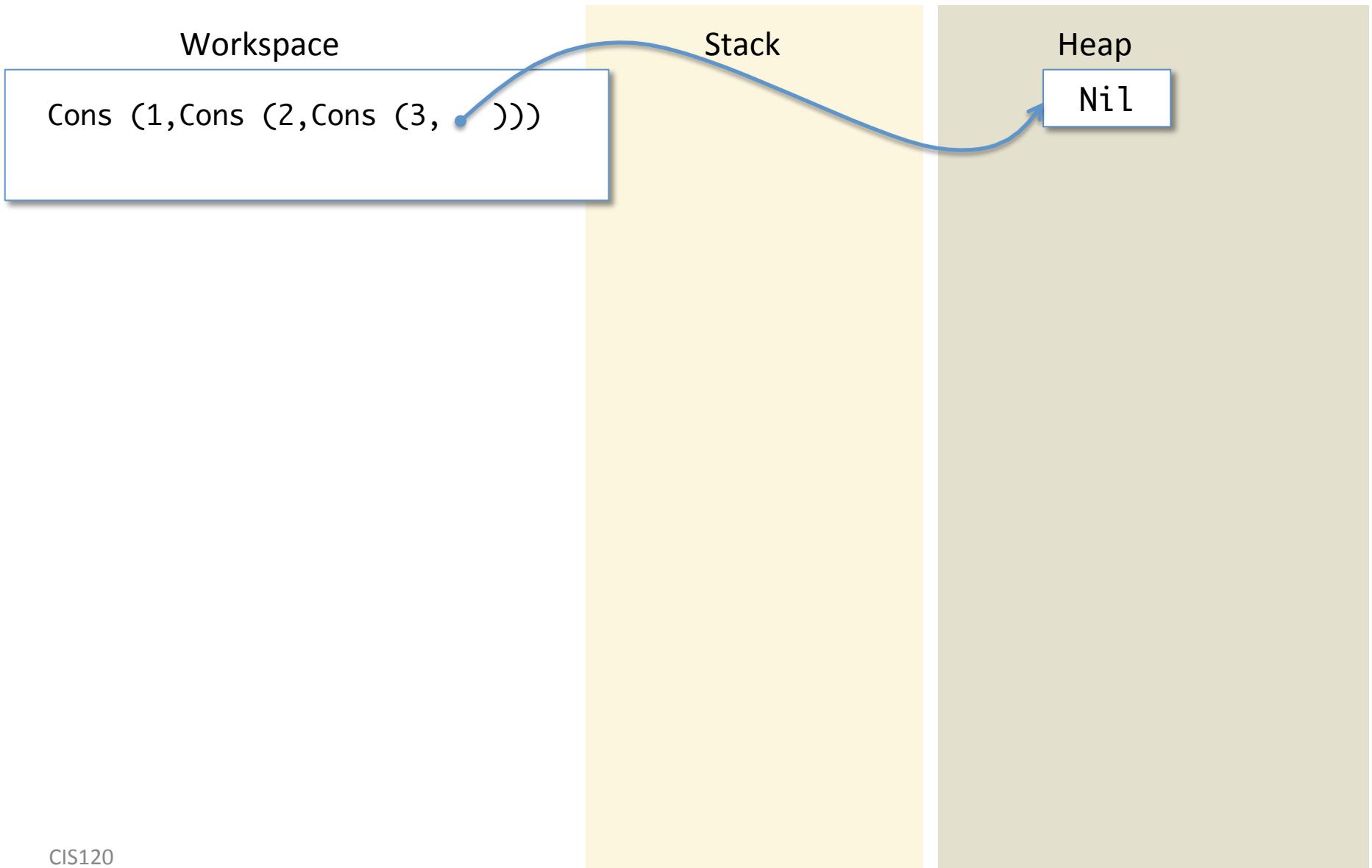
Workspace

```
Cons (1,Cons (2,Cons (3,Nil)))
```

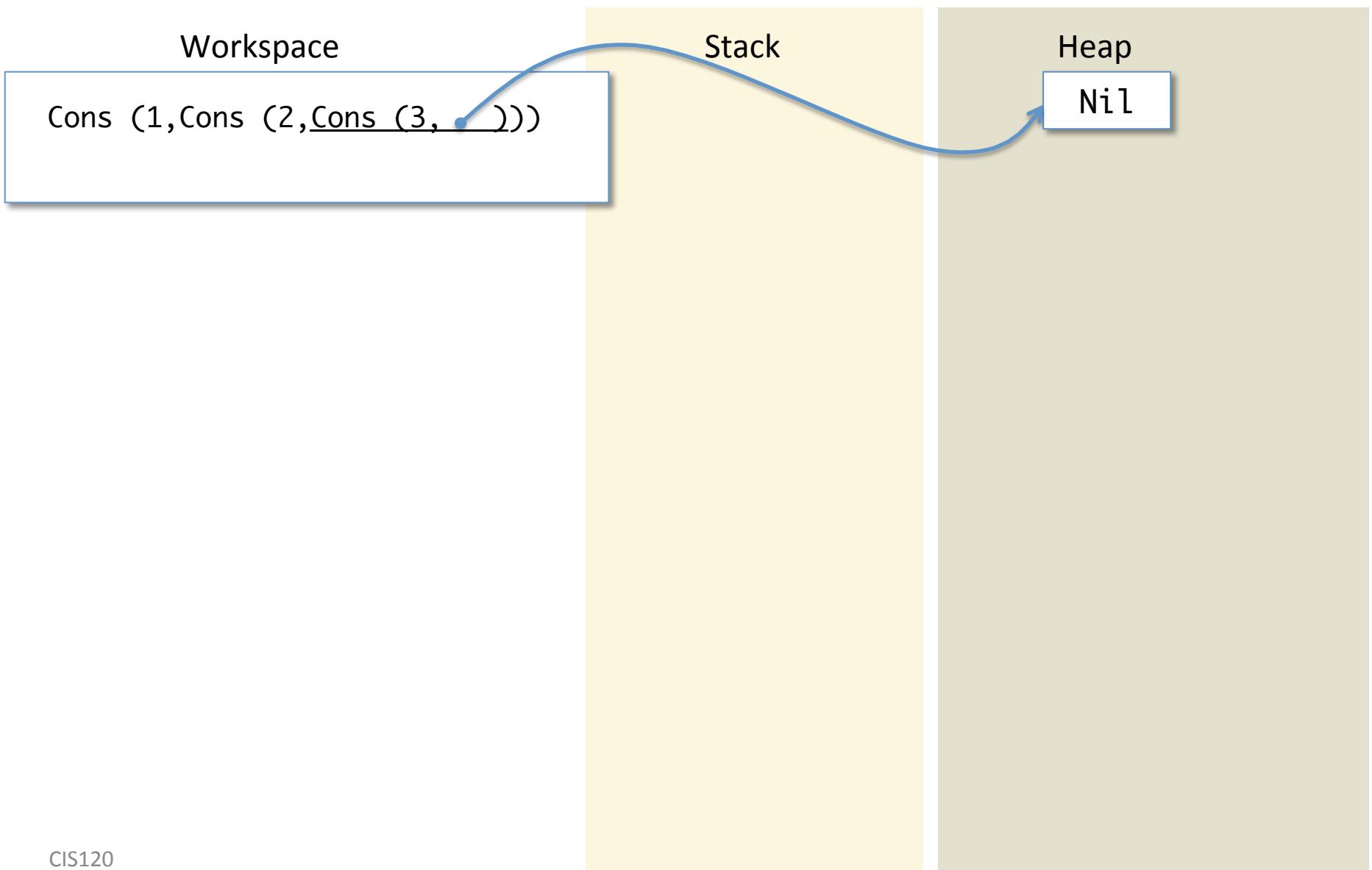
Stack

Heap

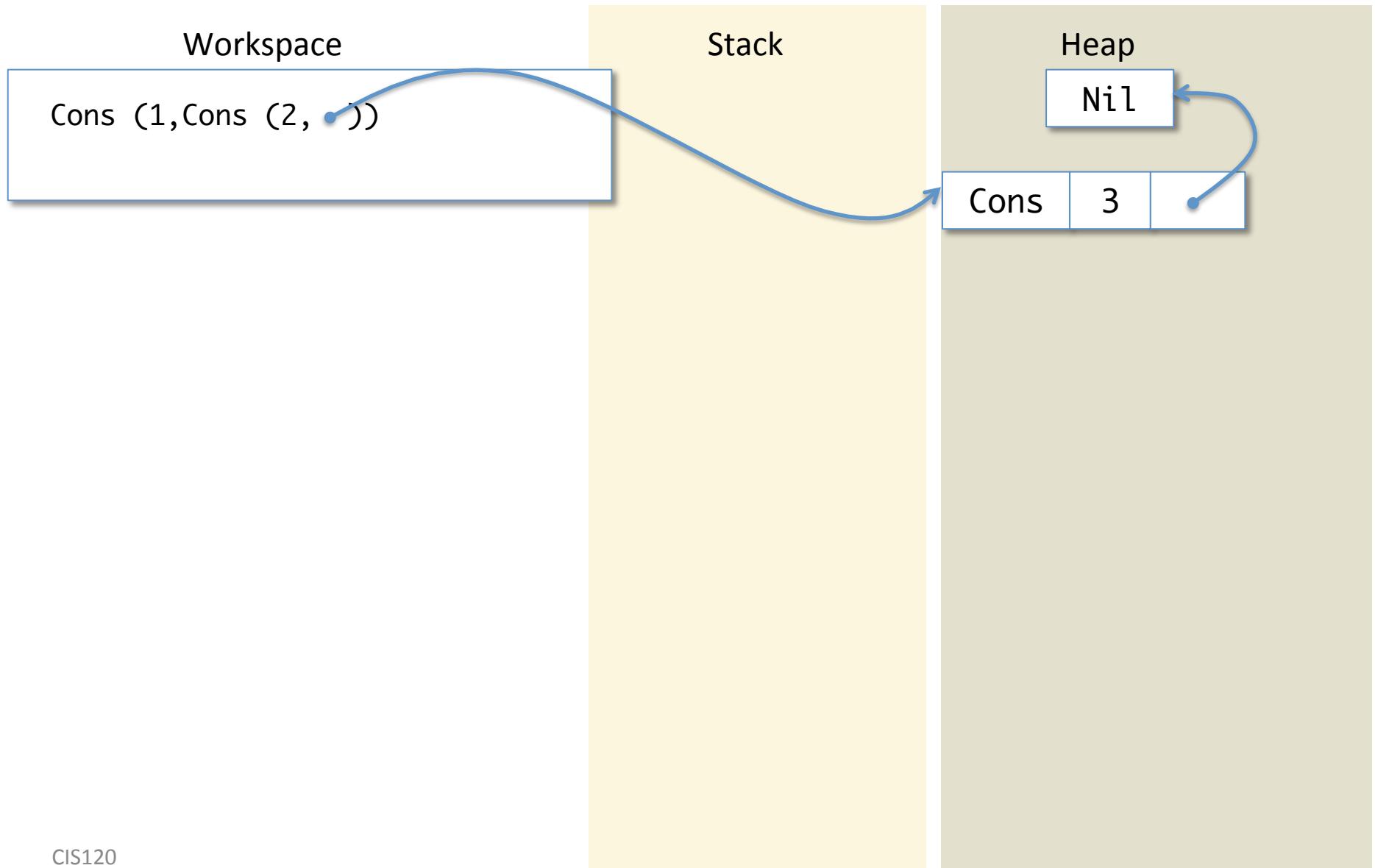
Simplification



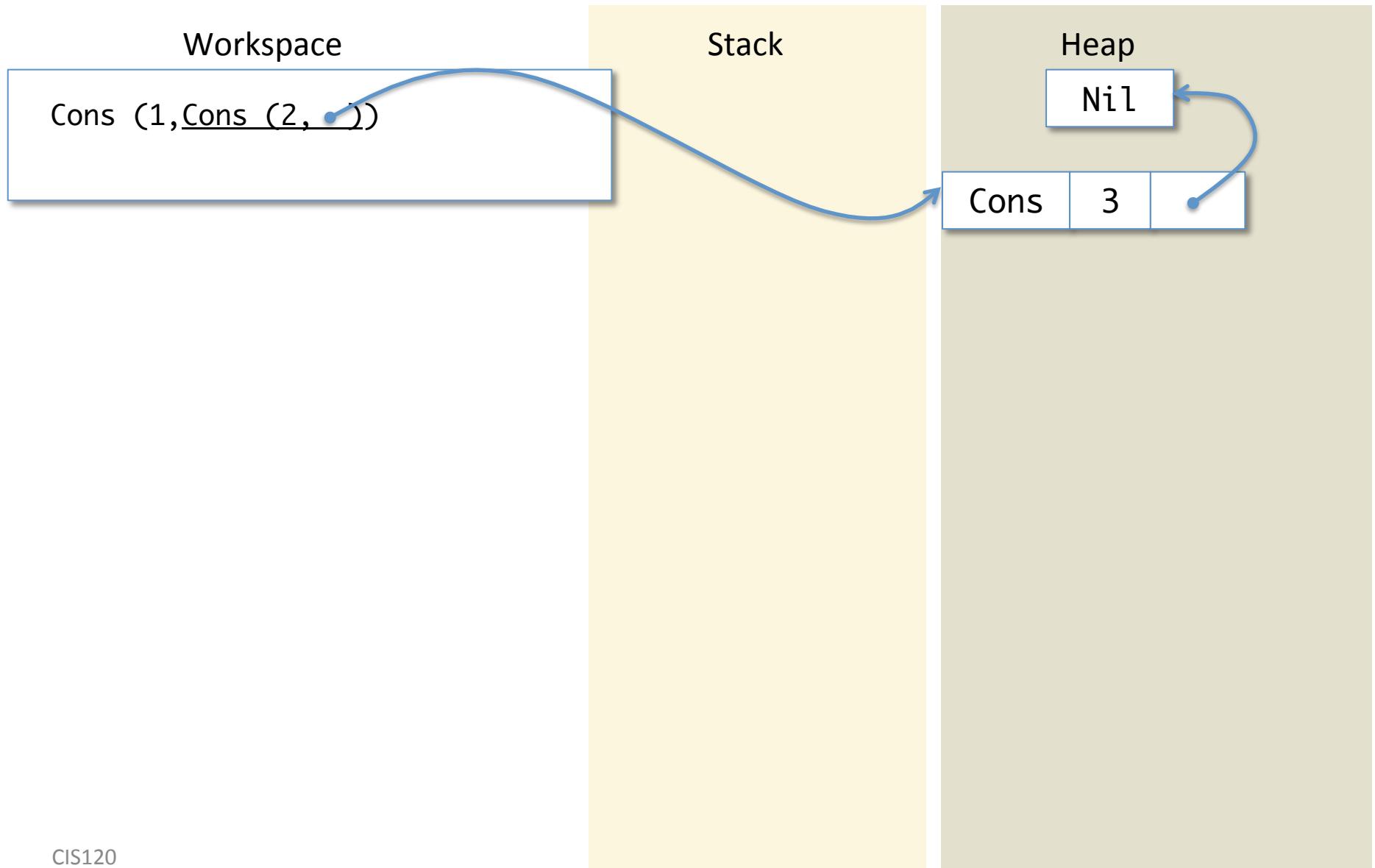
Simplification



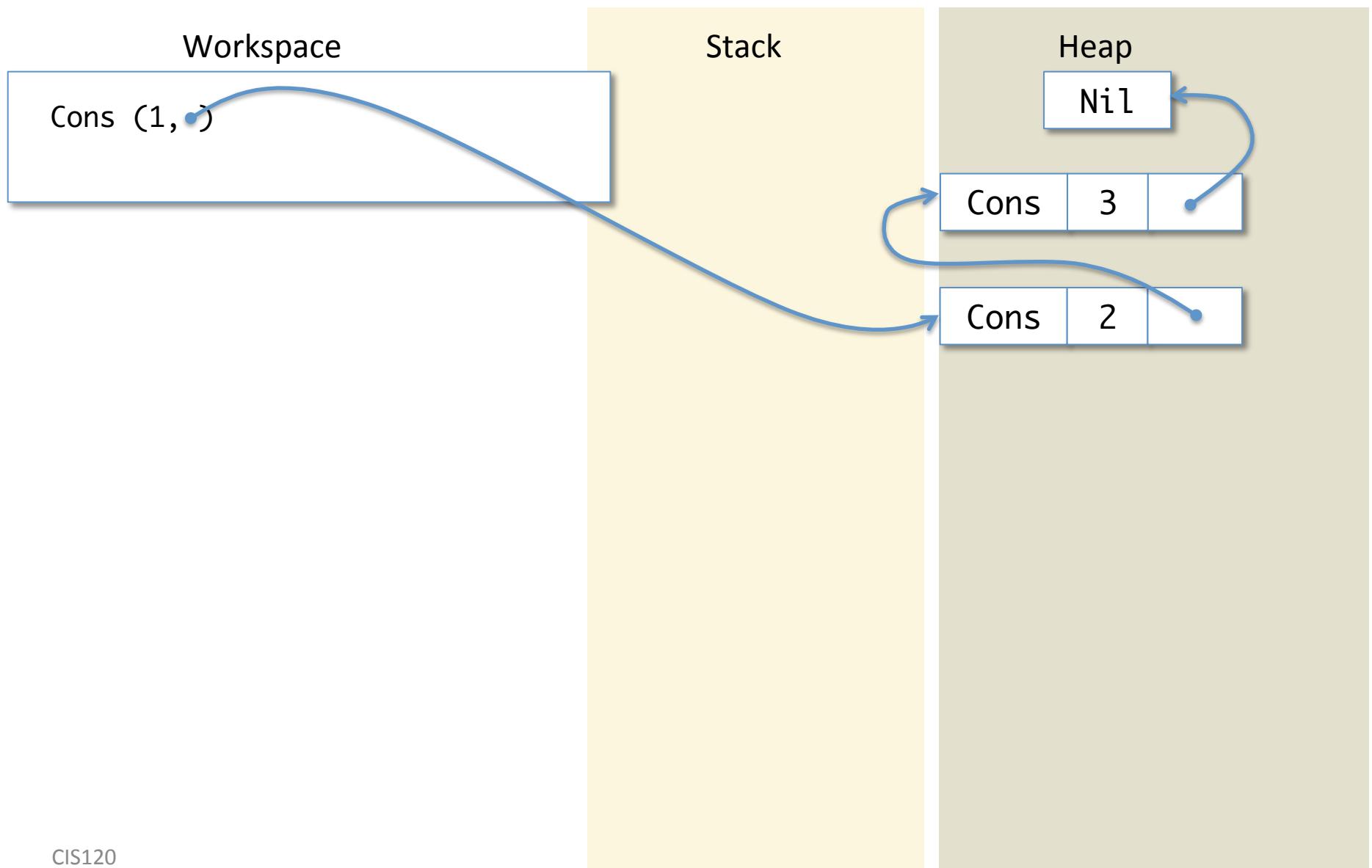
Simplification



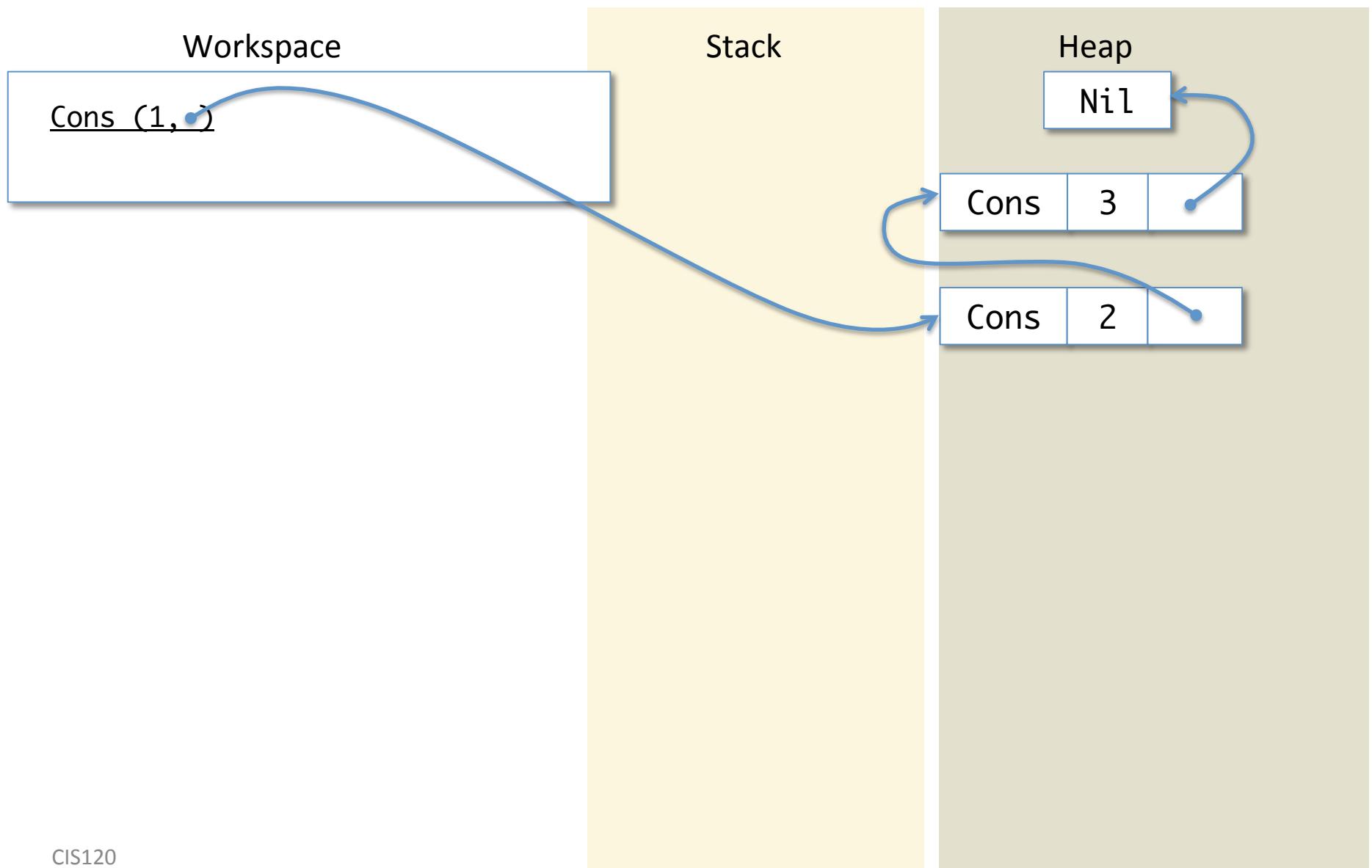
Simplification



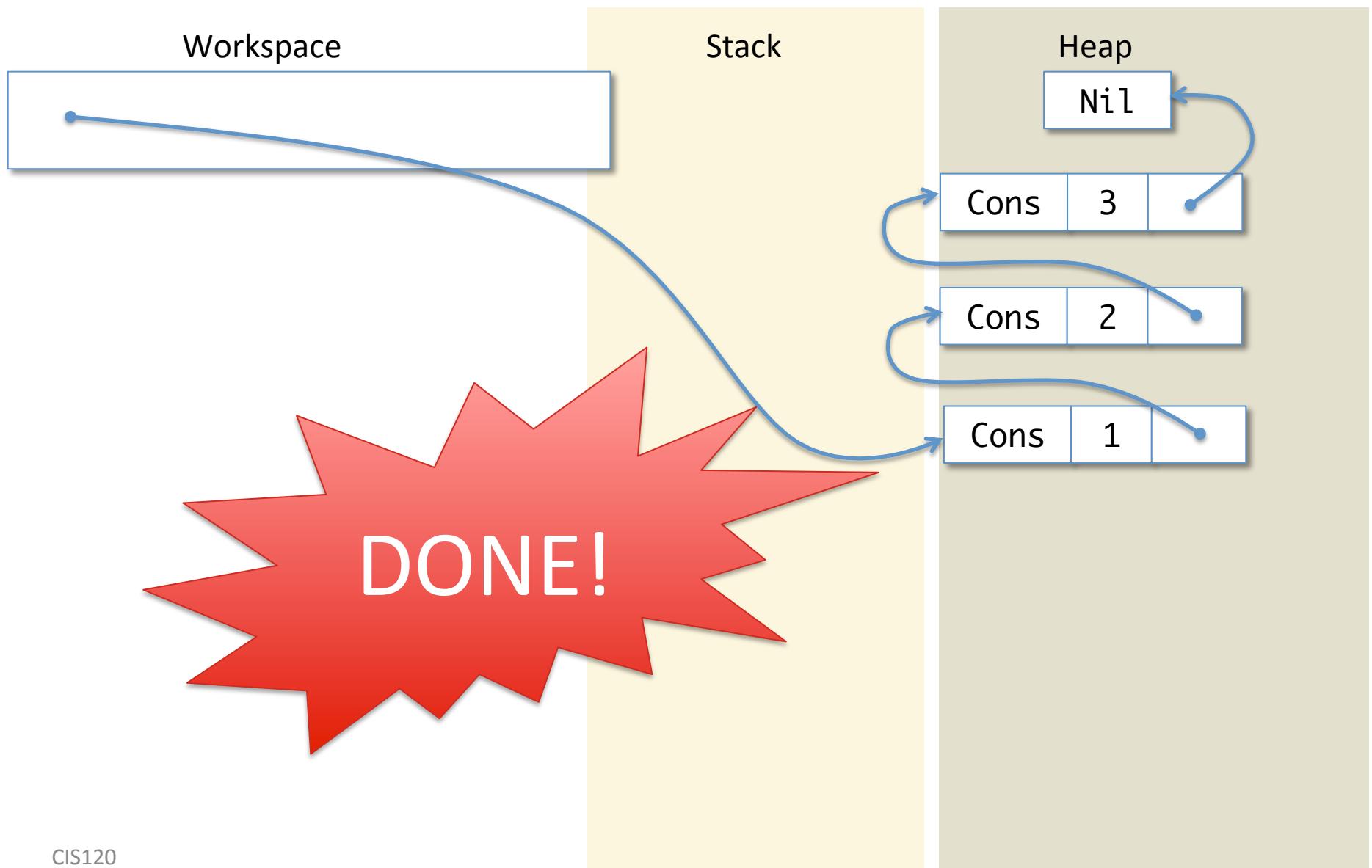
Simplification



Simplification



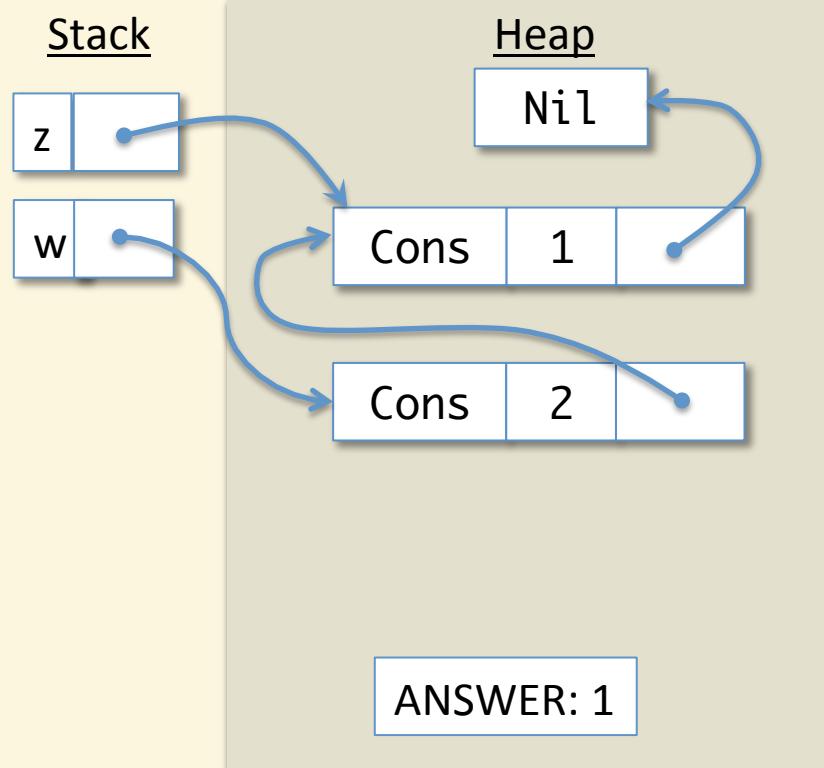
Simplification



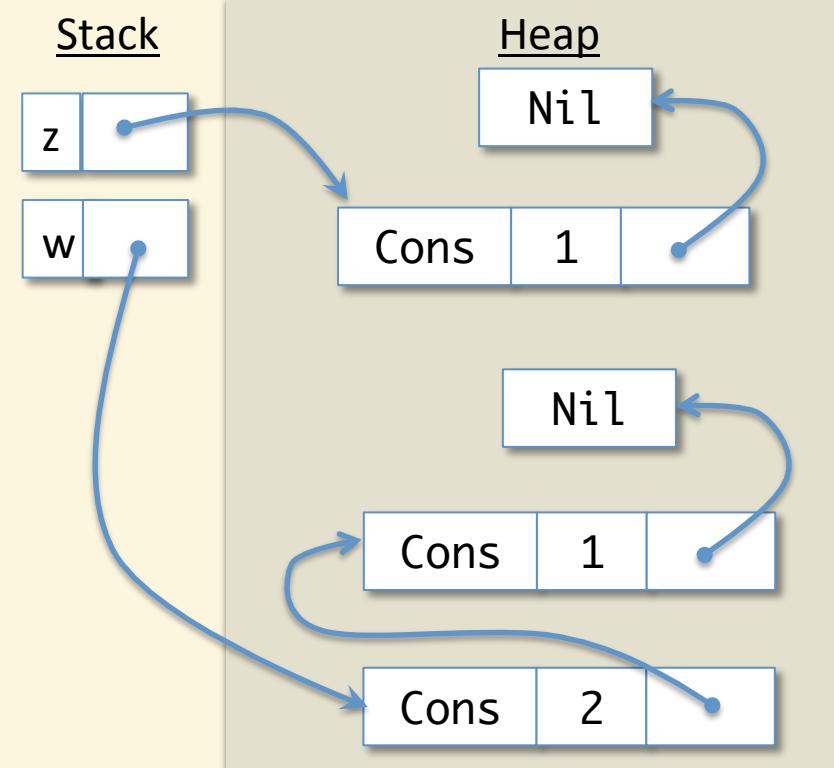
What do the Stack and Heap look like after simplifying the following code on the workspace?

```
let z = Cons (1, Nil) in  
let w = Cons (2, z) in  
    w
```

1.



2.



Simplifying functions

Function Simplification

Workspace

```
let add1 (x : int) : int =  
    x + 1 in  
add1 (add1 0)
```

Stack

Heap

Function Simplification

Workspace

```
let add1 (x : int) : int =
  x + 1 in
add1 (add1 0)
```

Stack

Heap

Function Simplification

Workspace

```
let add1 : int -> int =  
  fun (x:int) -> x + 1 in  
add1 (add1 0)
```

Stack

Heap

Function Simplification

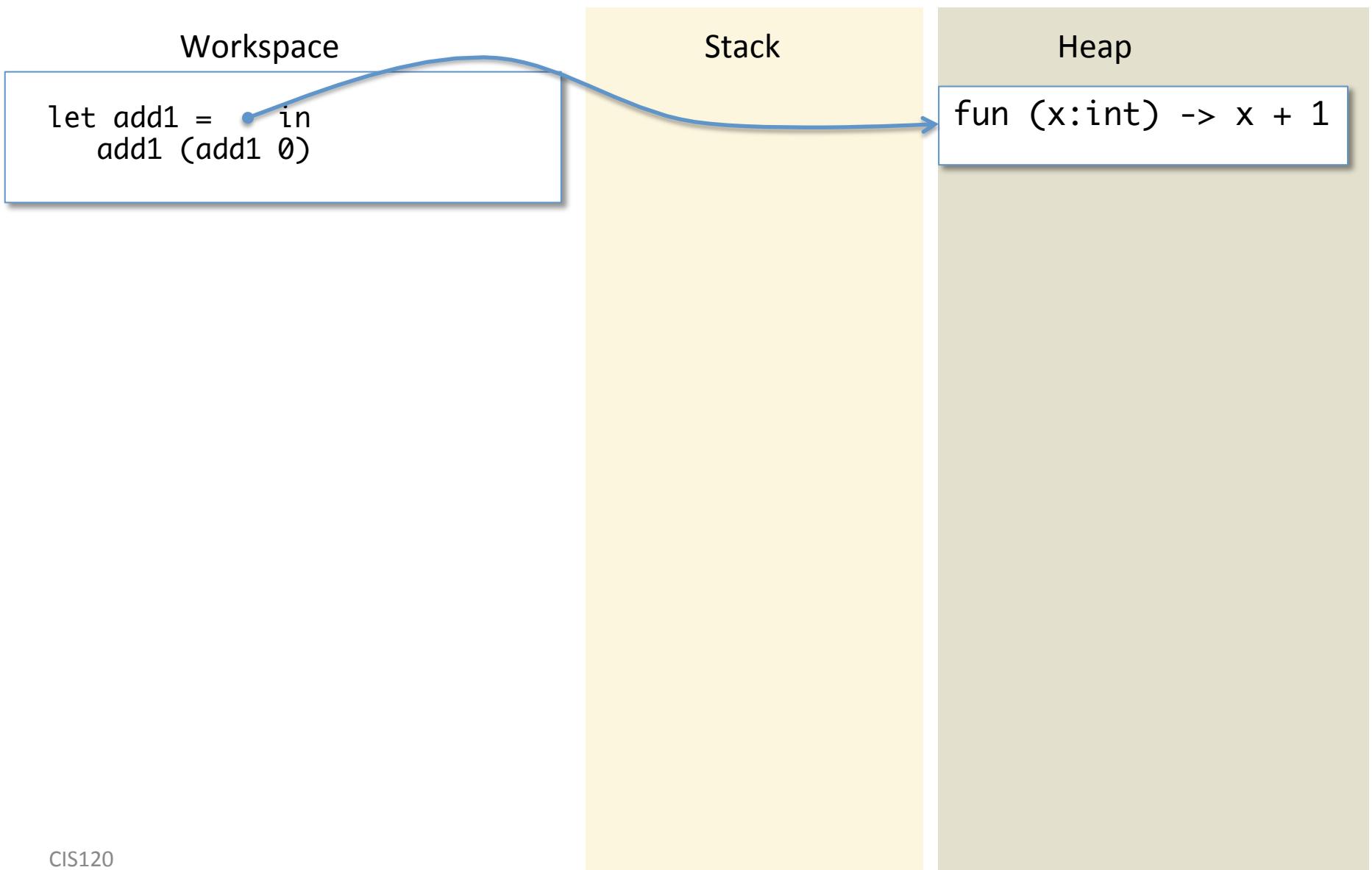
Workspace

```
let add1 : int -> int =  
  fun (x:int) -> x + 1 in  
add1 (add1 0)
```

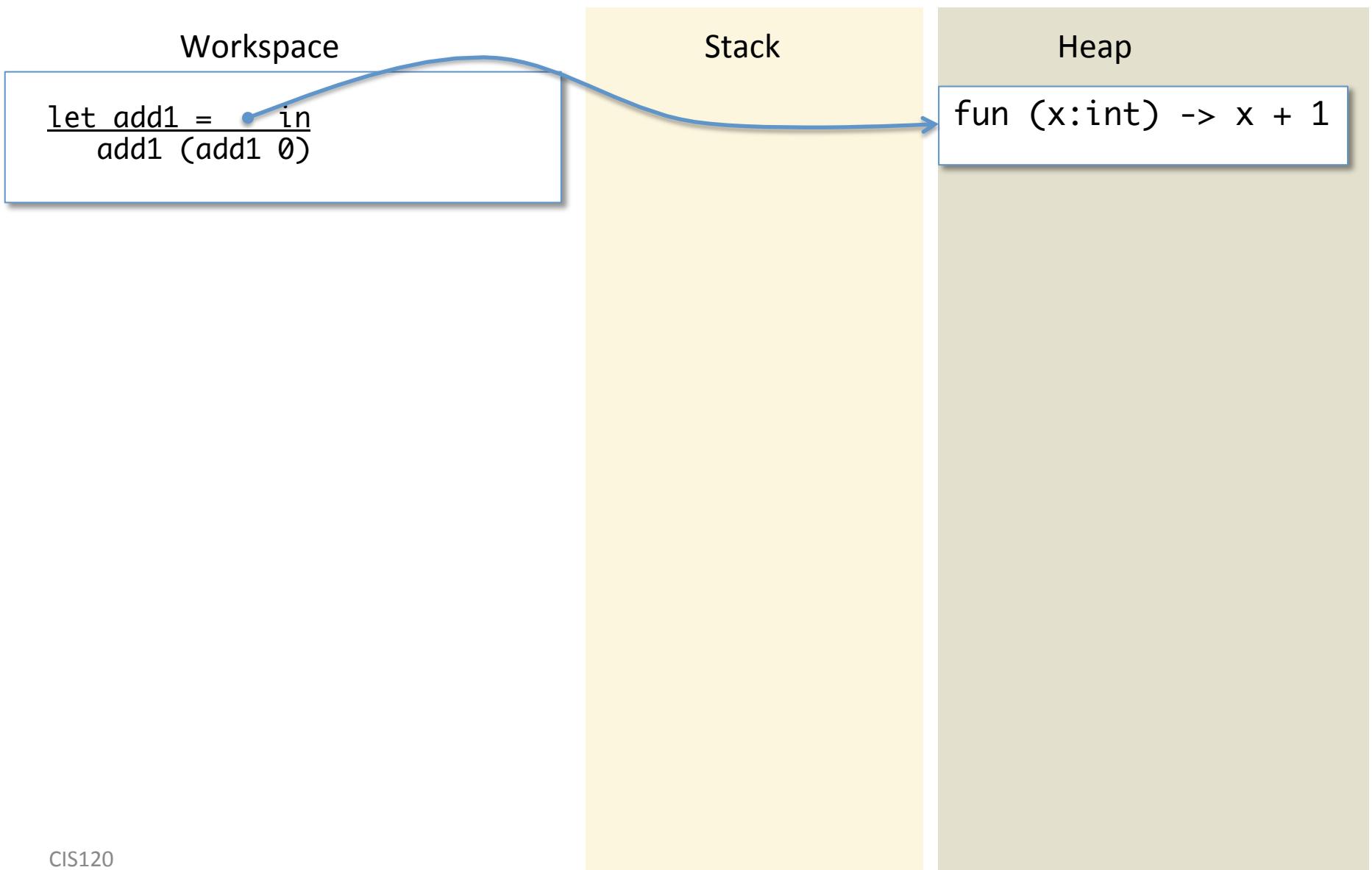
Stack

Heap

Function Simplification



Function Simplification



Function Simplification

Workspace

```
add1 (add1 0)
```

Stack

```
add1
```

Heap

```
fun (x:int) -> x + 1
```

Function Simplification

Workspace

```
add1 (add1 0)
```

Stack

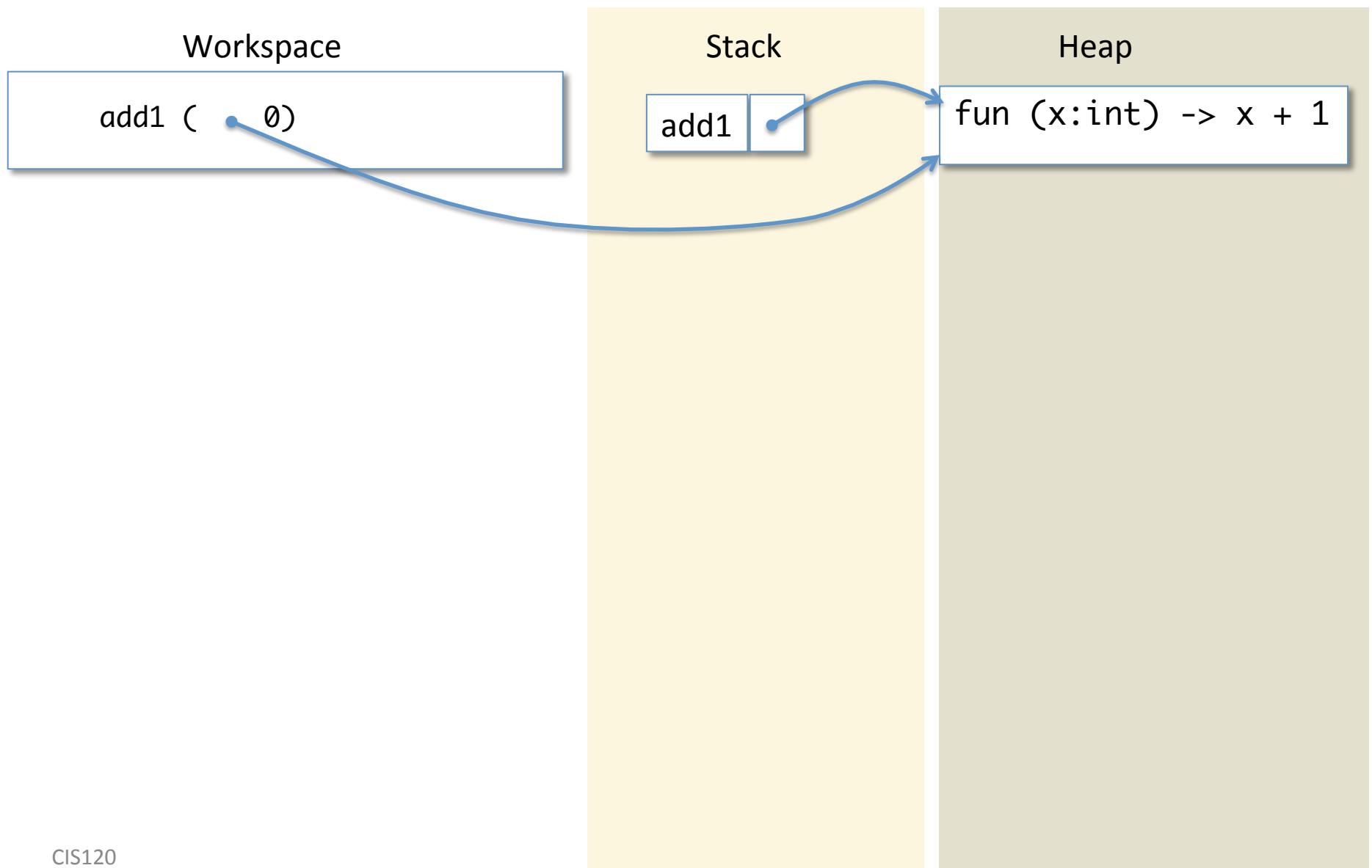
```
add1
```

Heap

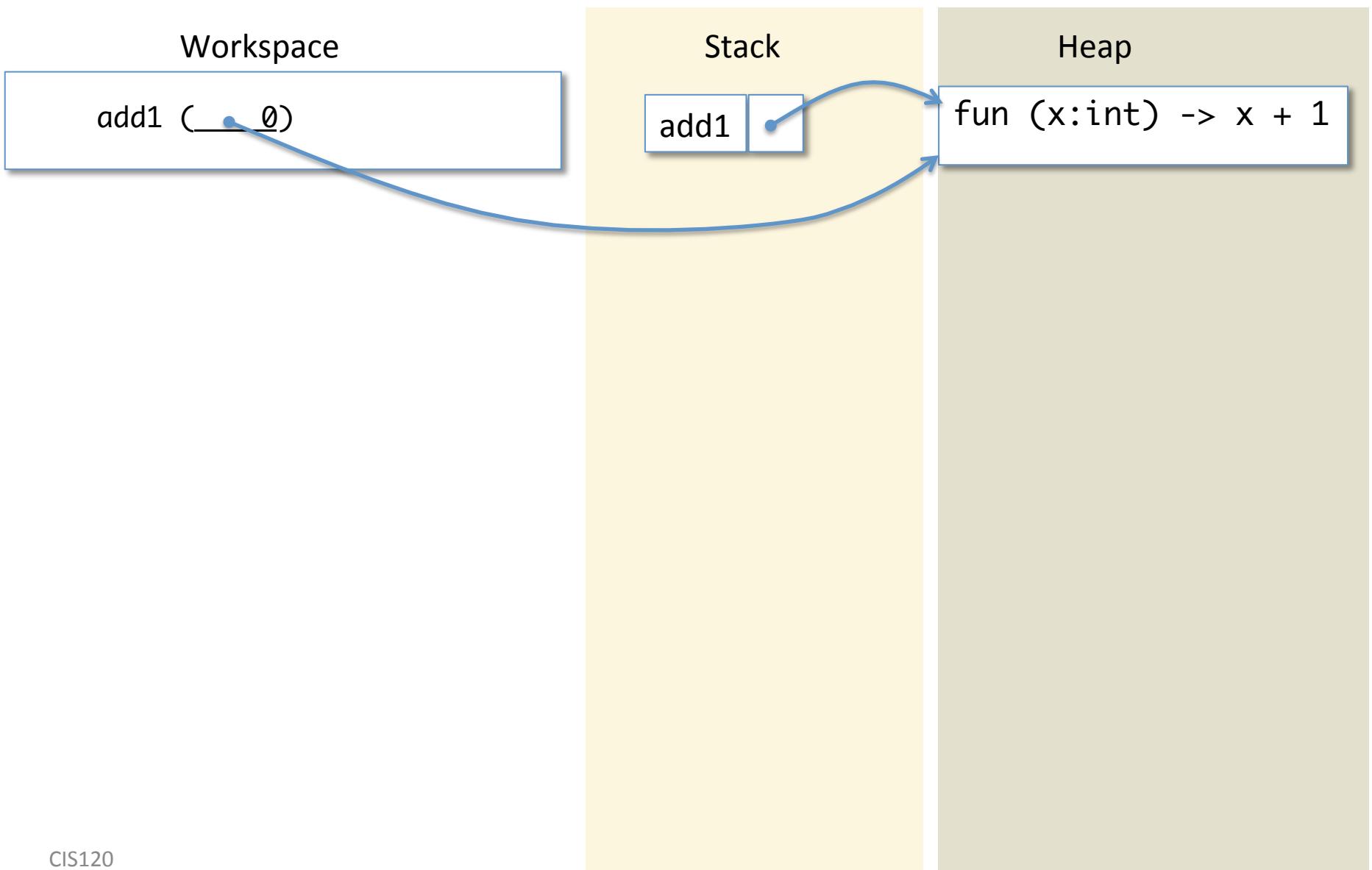
```
fun (x:int) -> x + 1
```



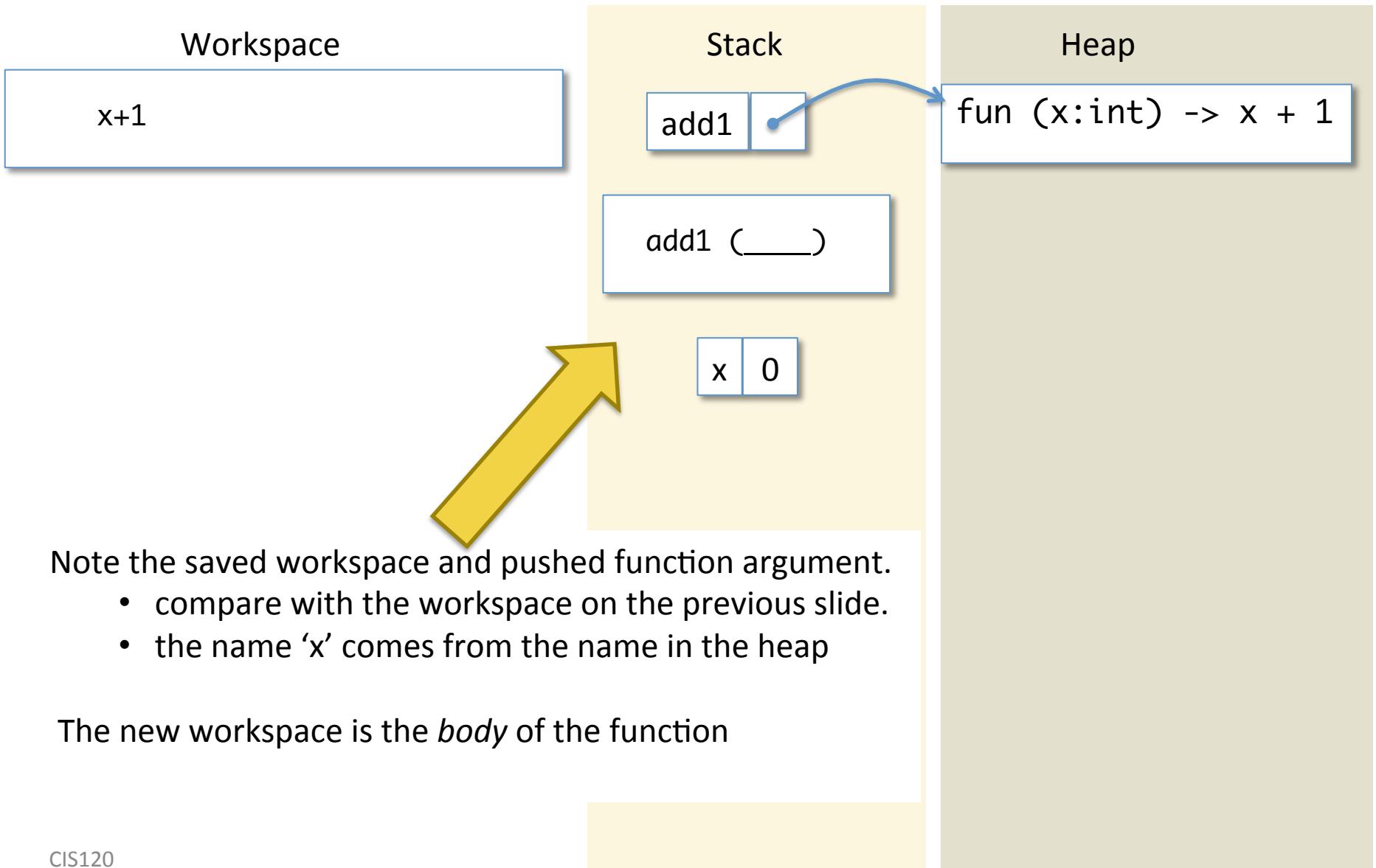
Function Simplification



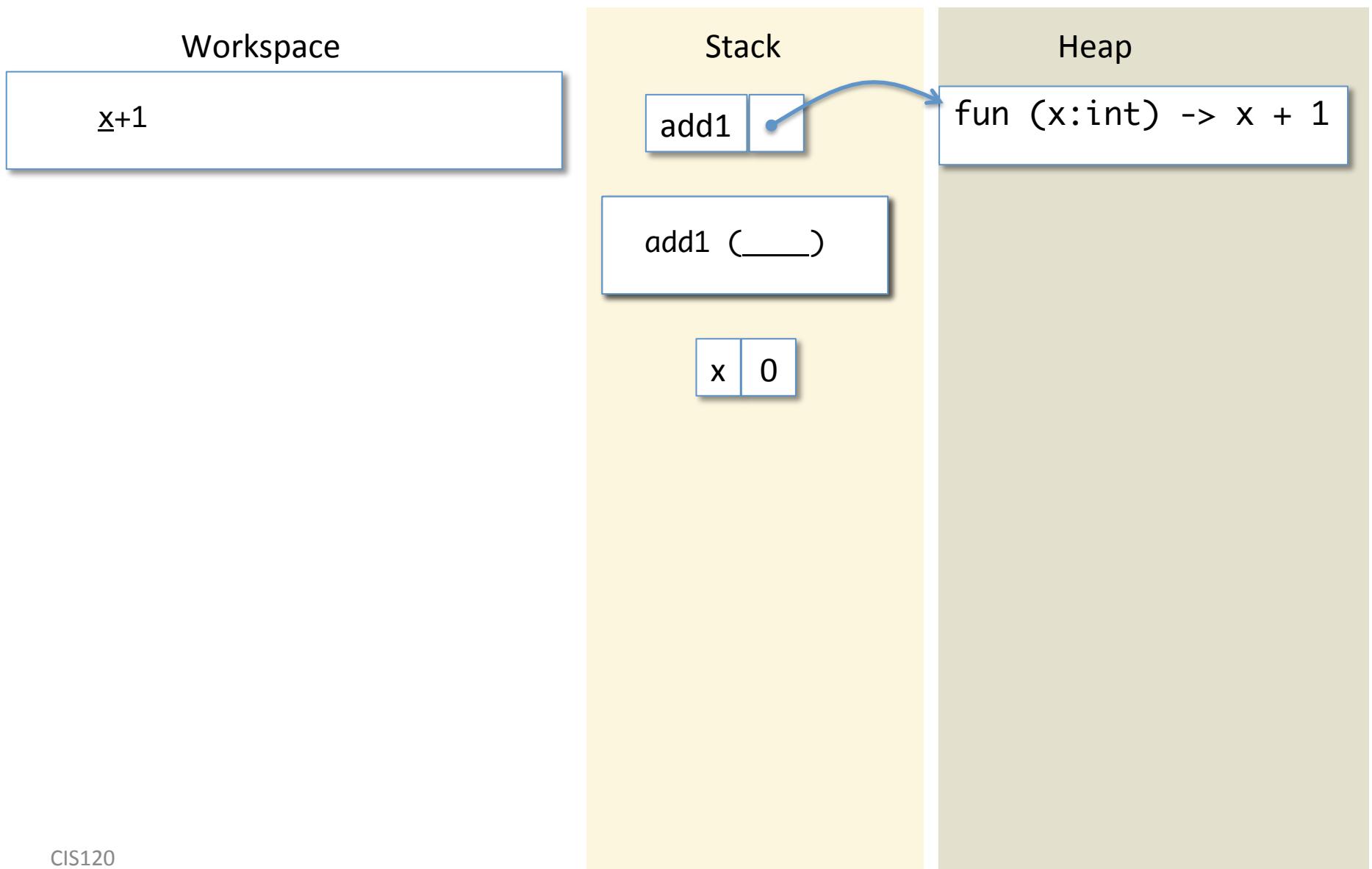
Function Simplification



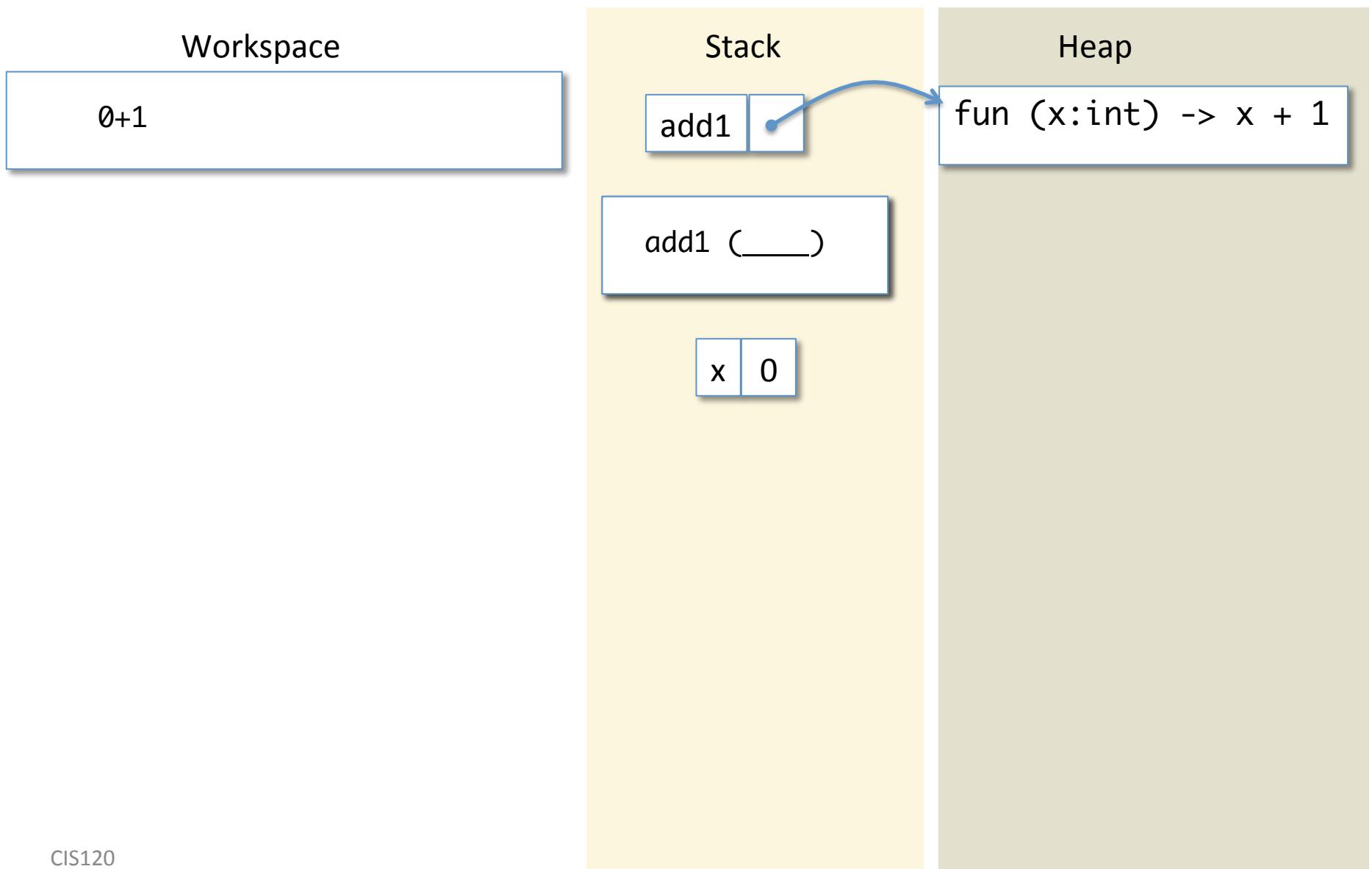
Do the Call, Saving the Workspace



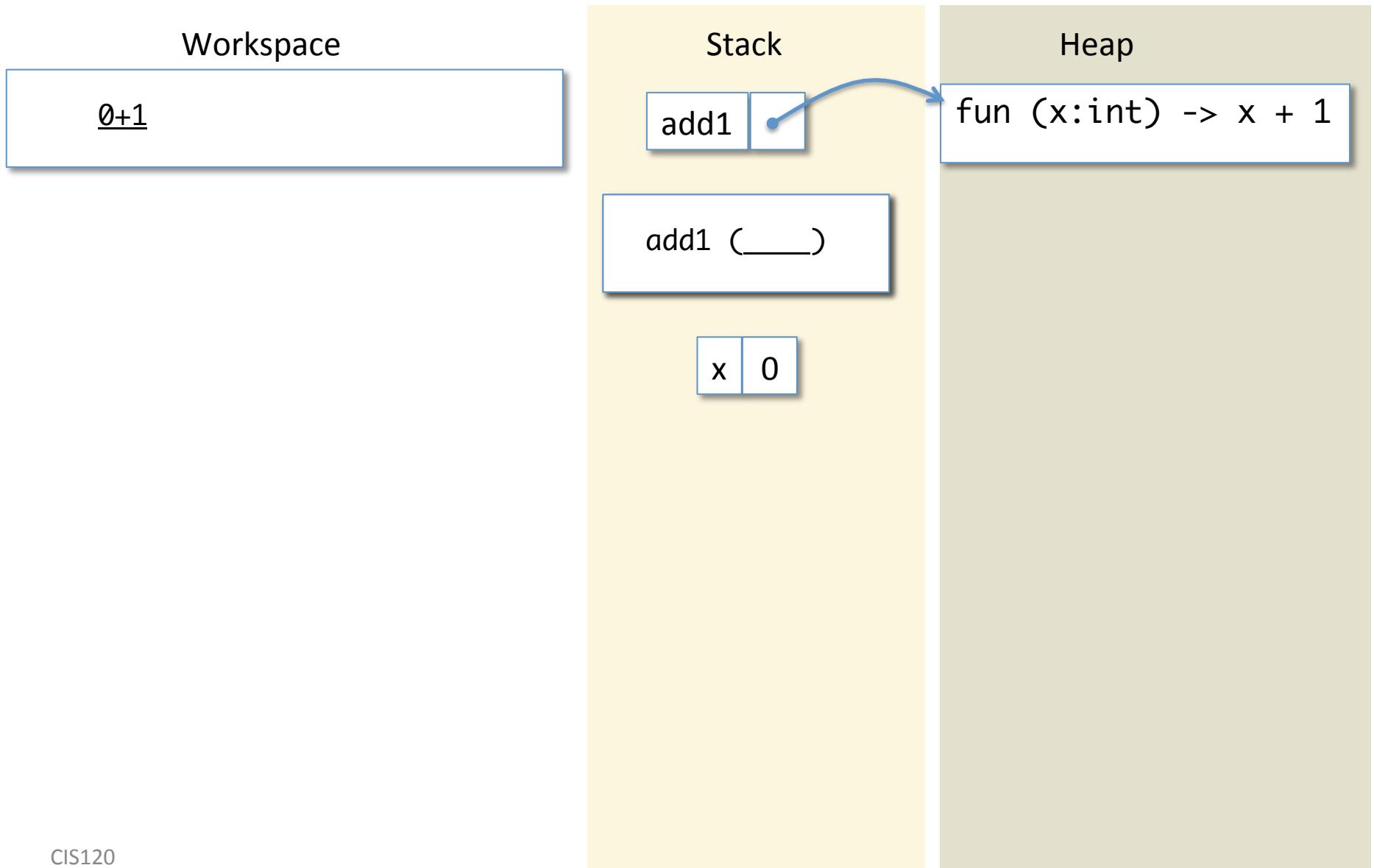
Function Simplification



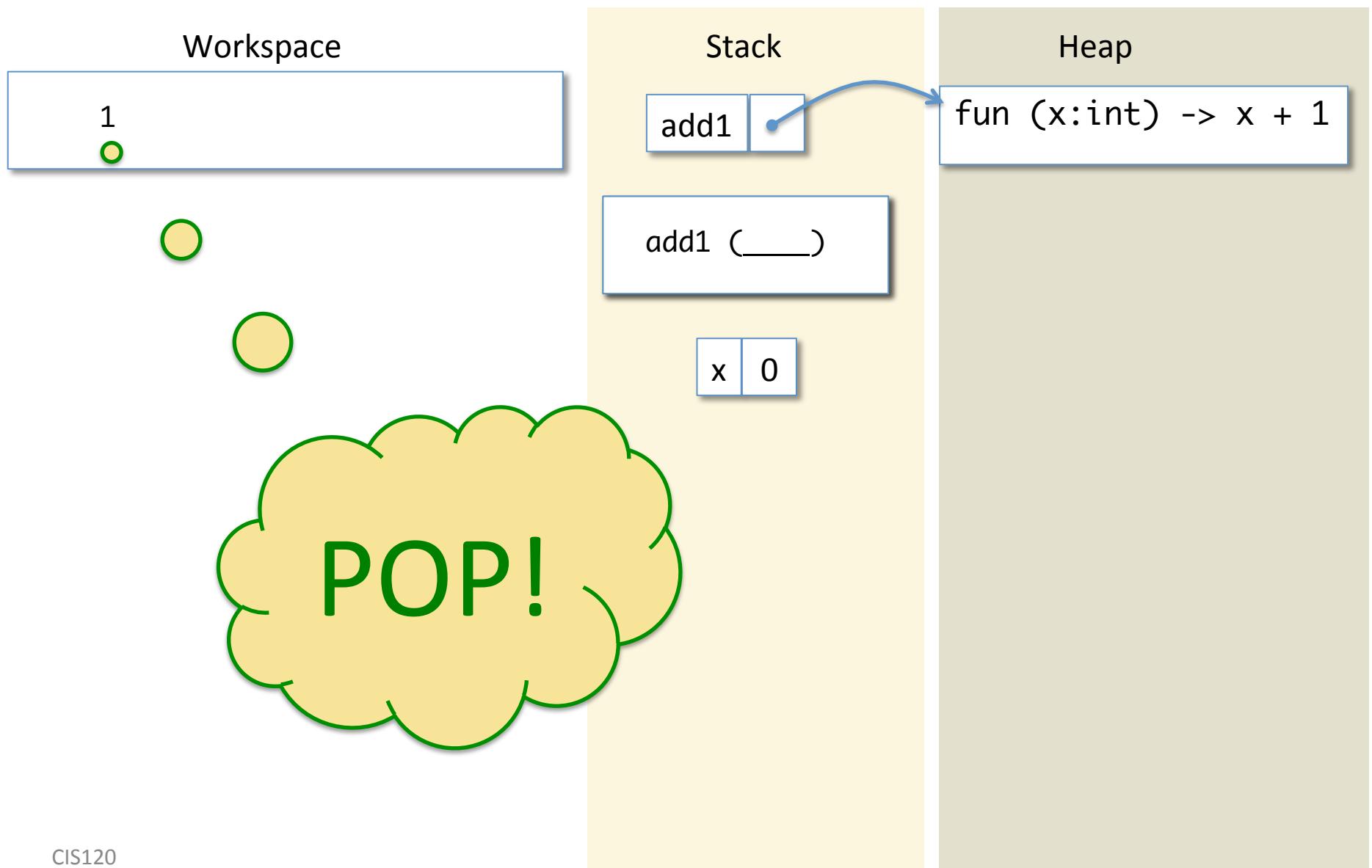
Function Simplification



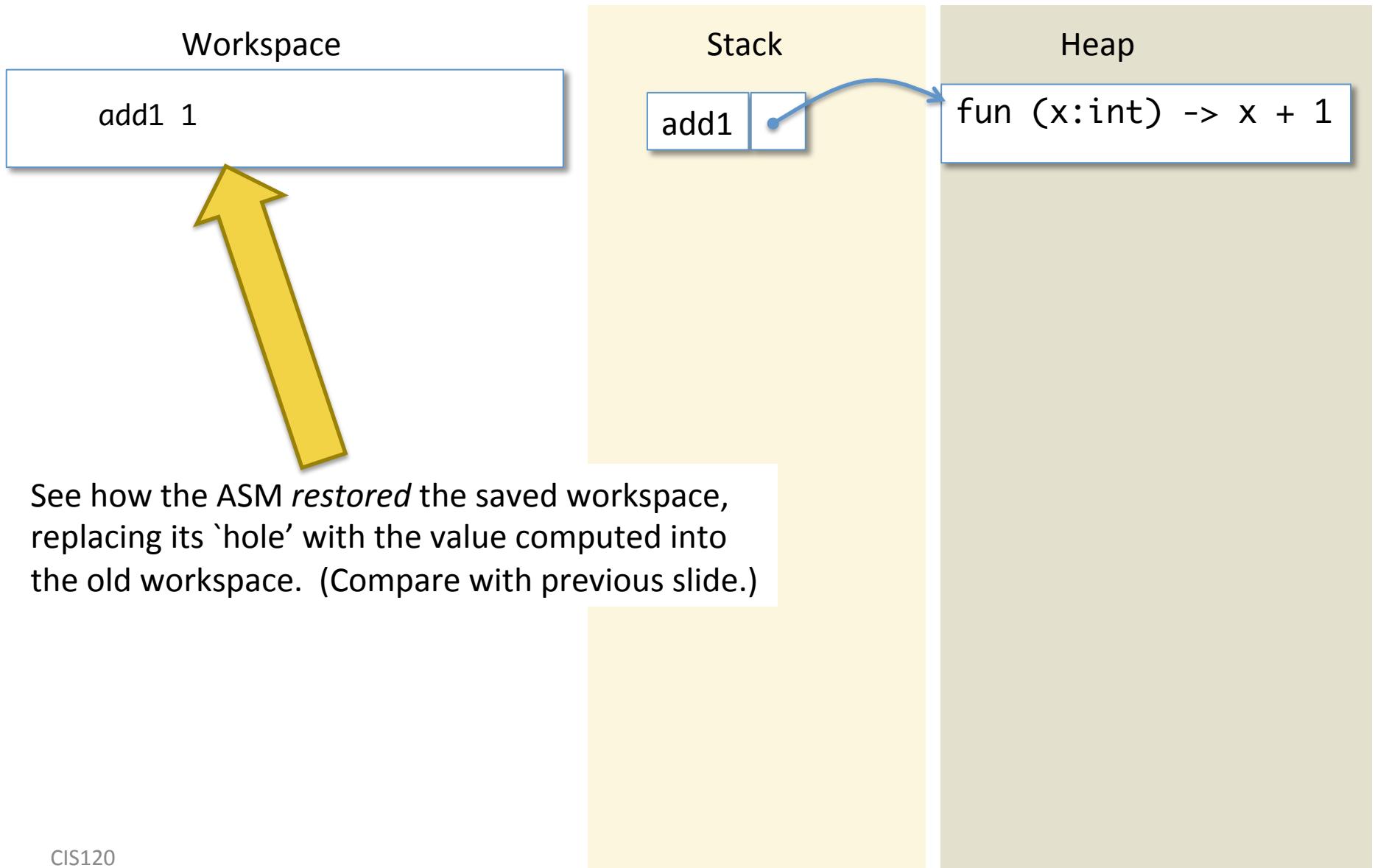
Function Simplification



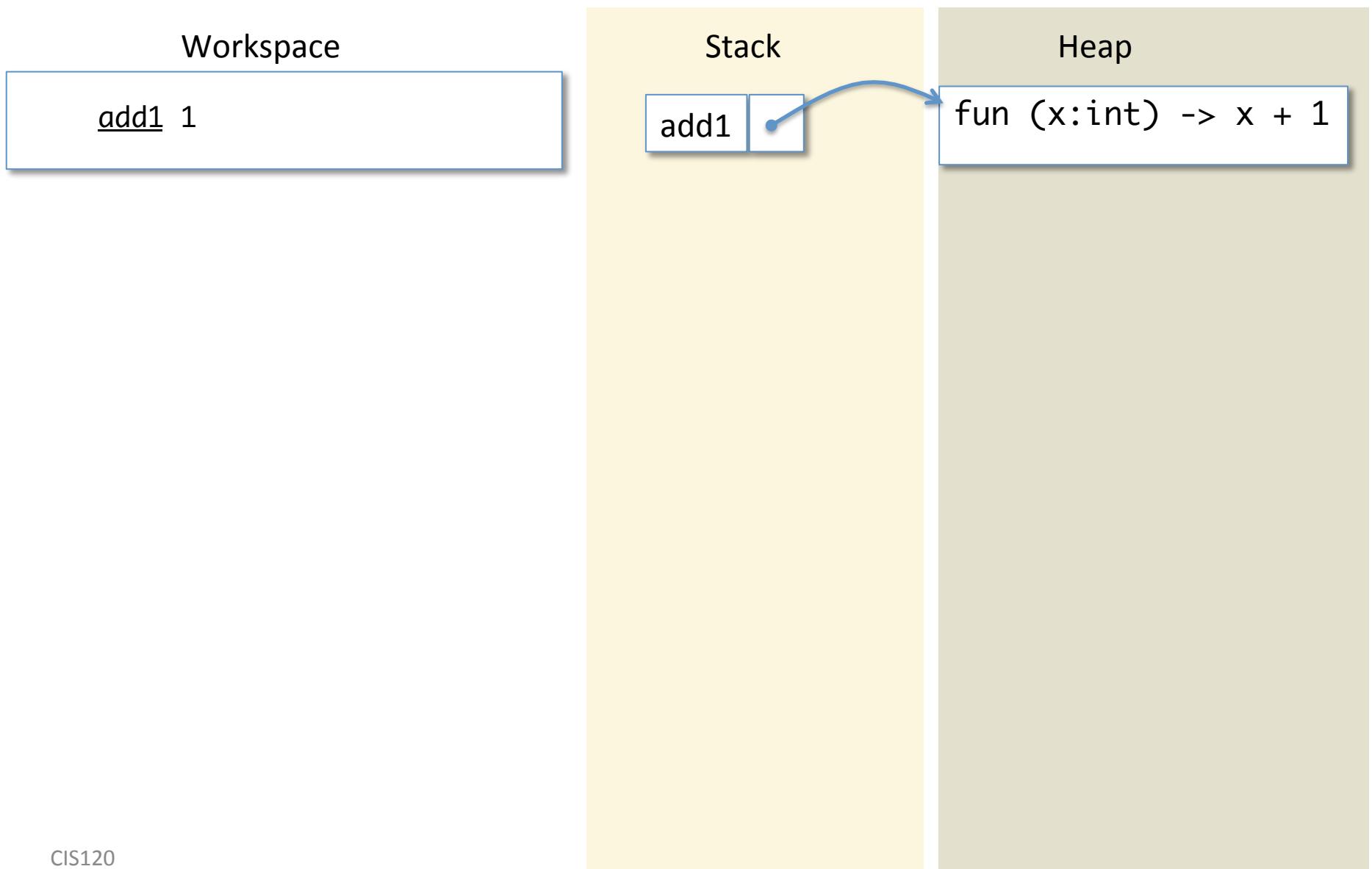
Function Simplification



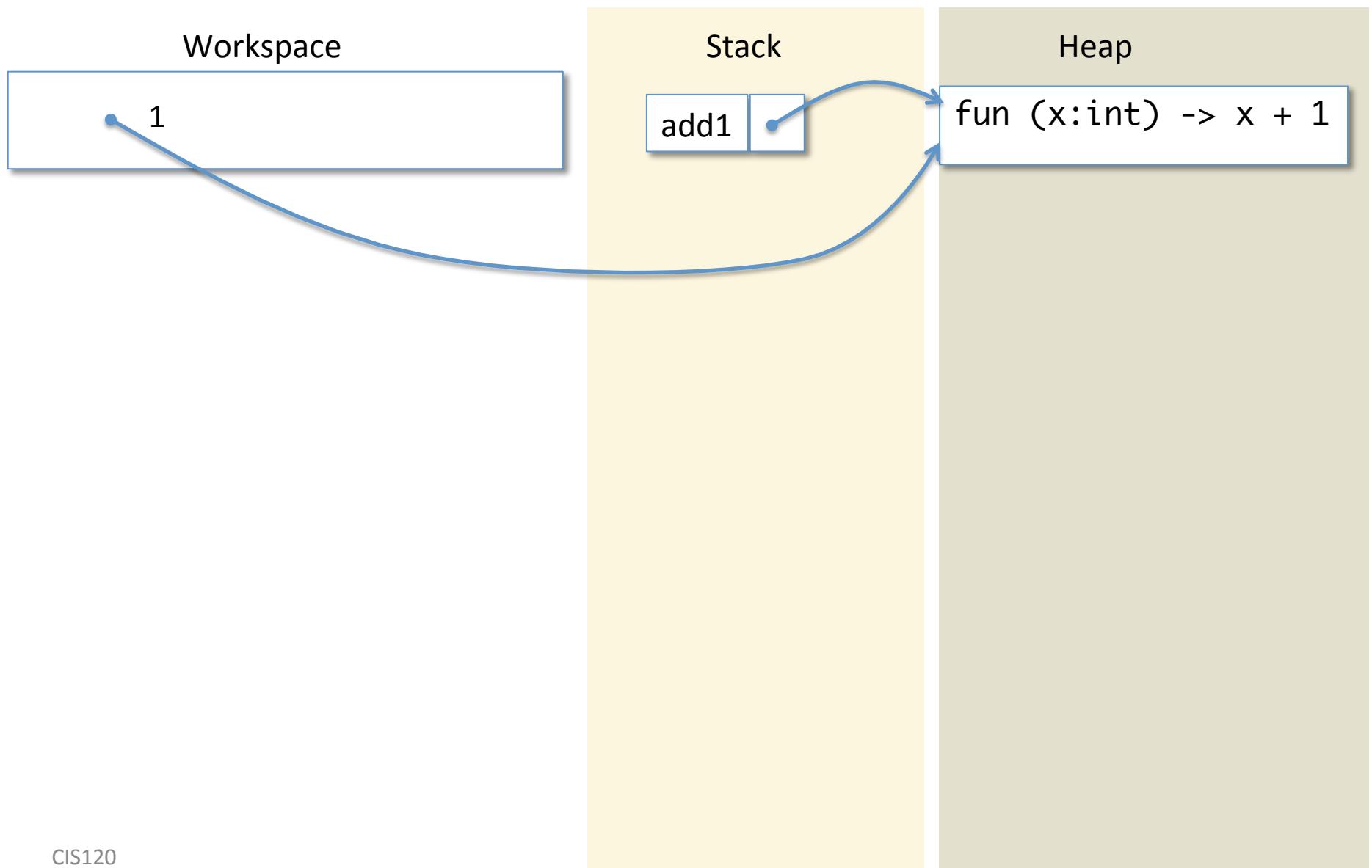
Function Simplification



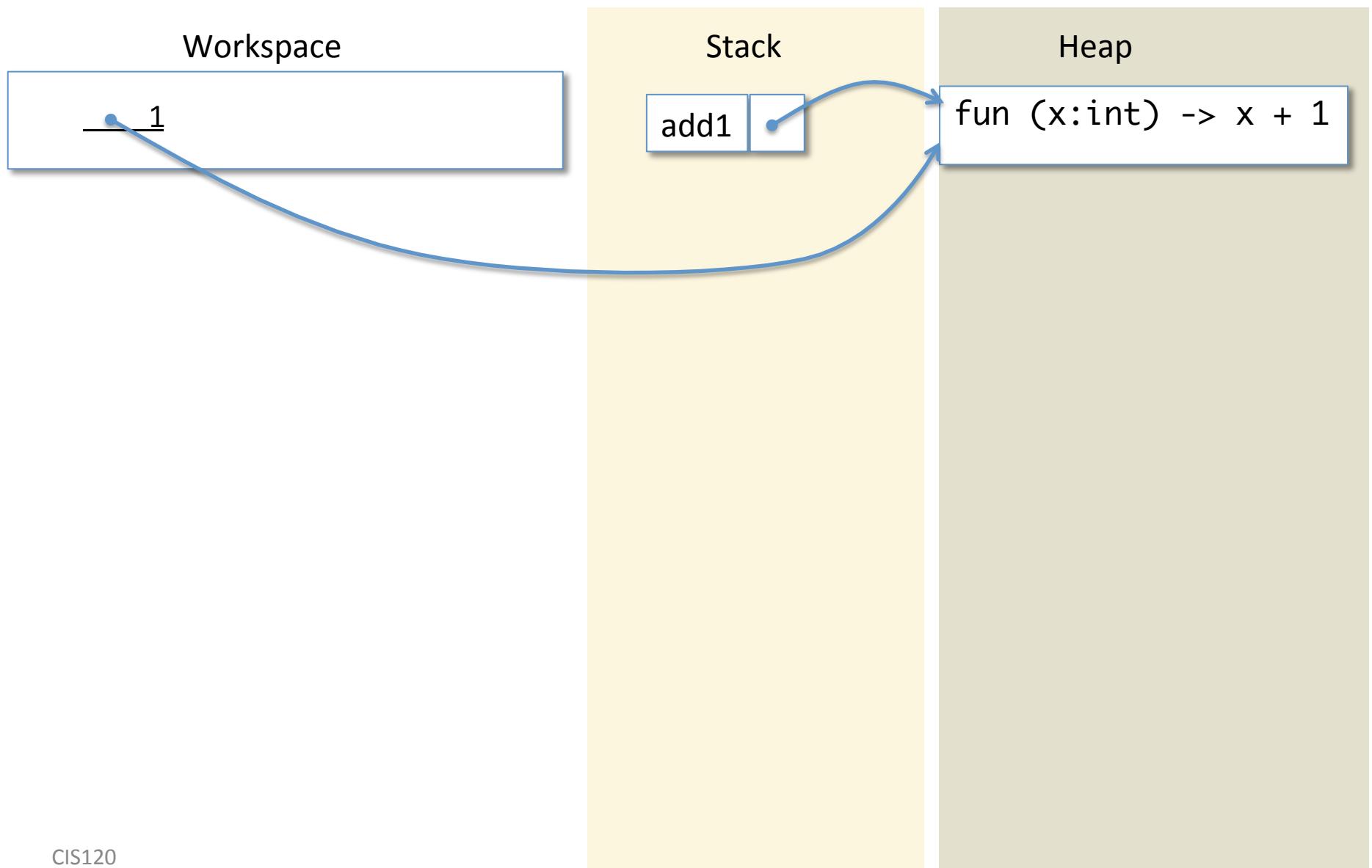
Function Simplification



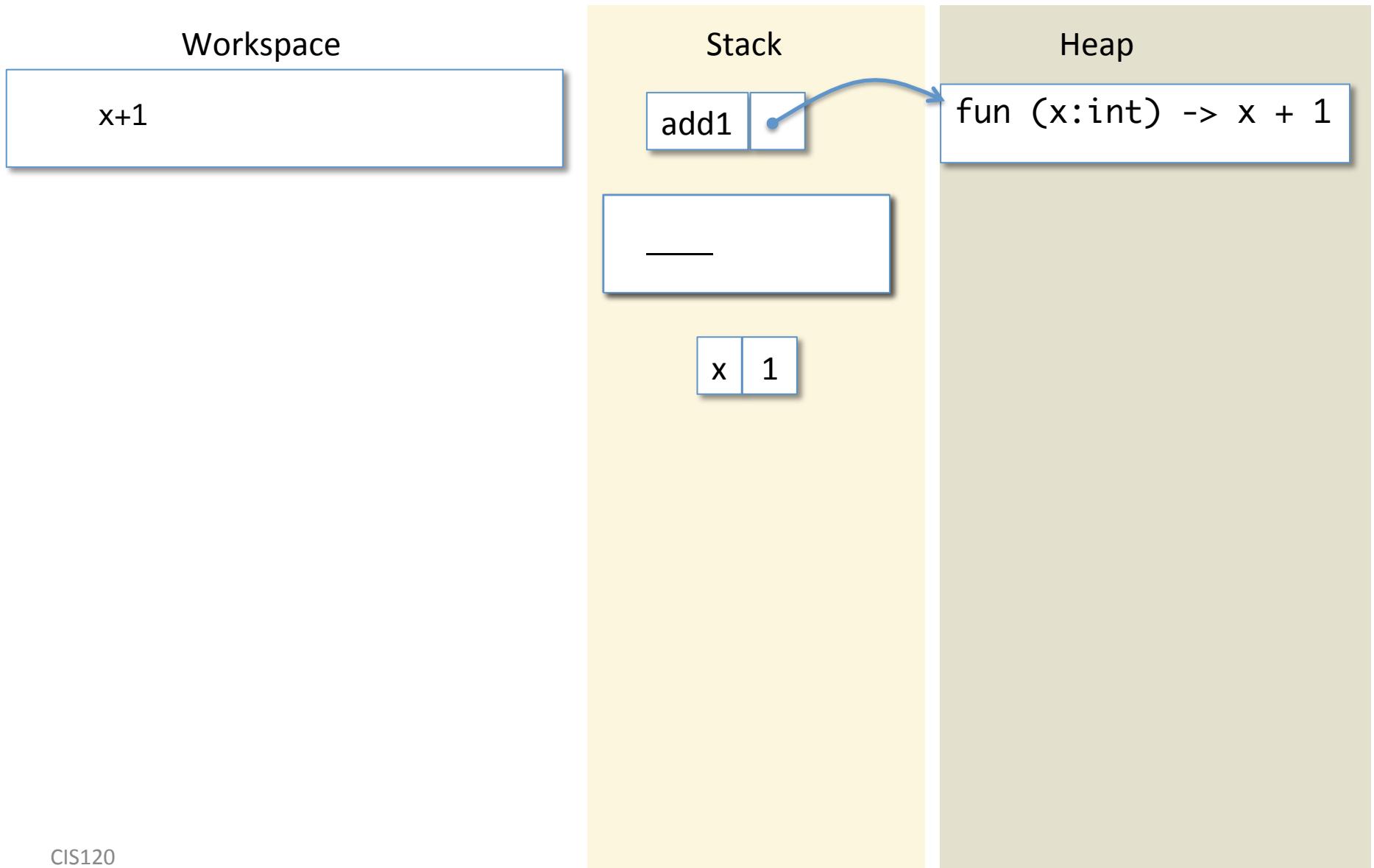
Function Simplification



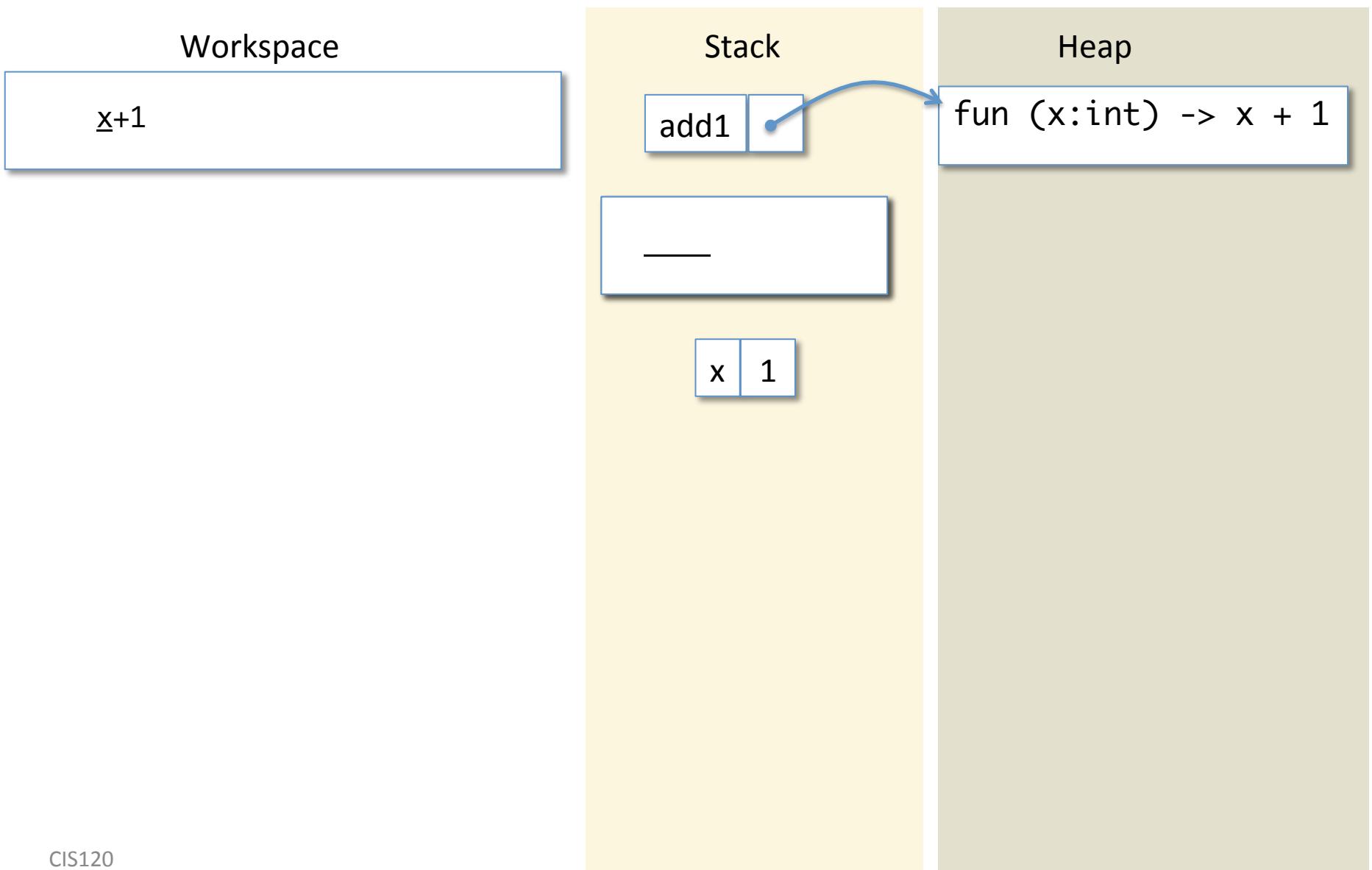
Function Simplification



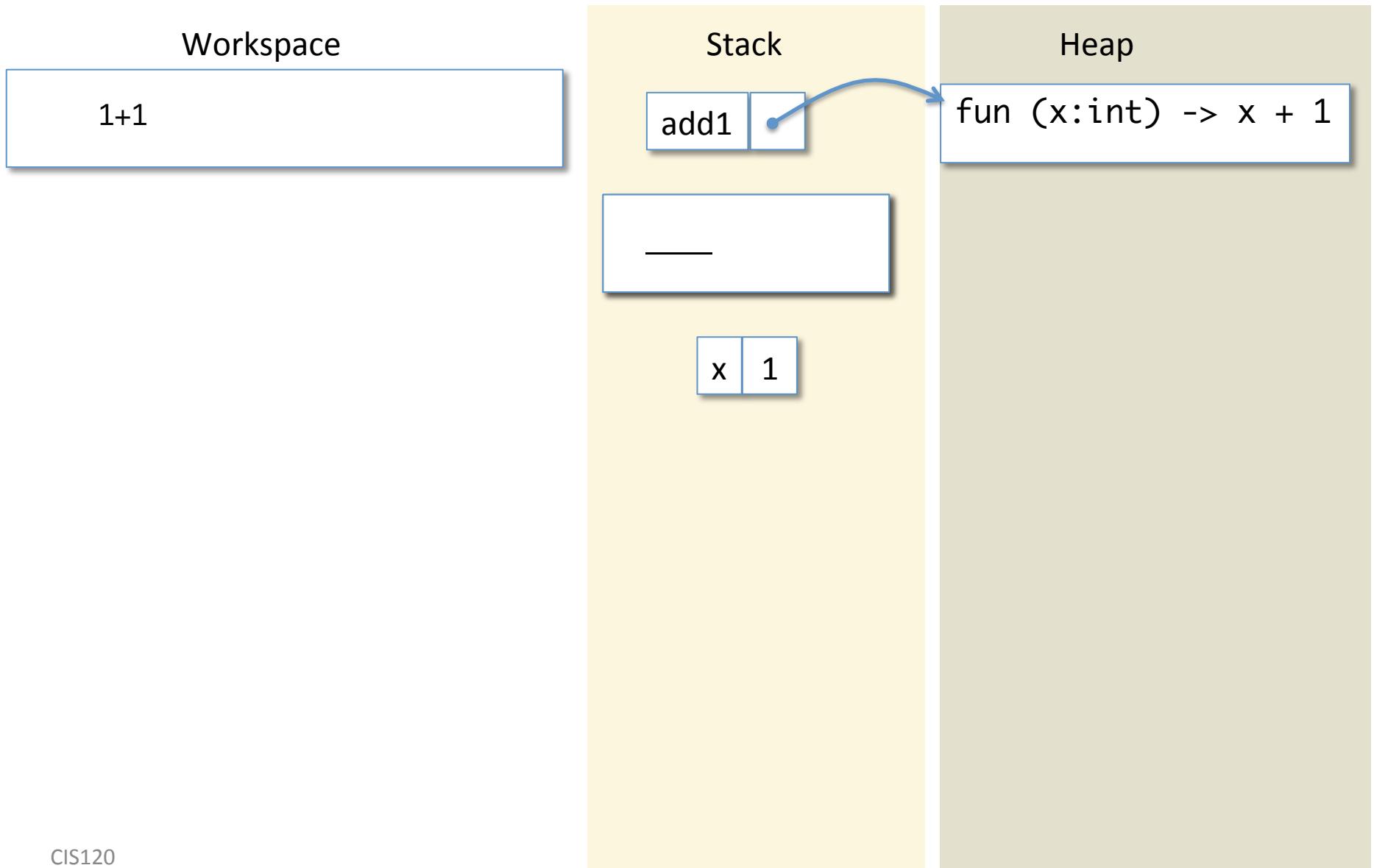
Function Simplification



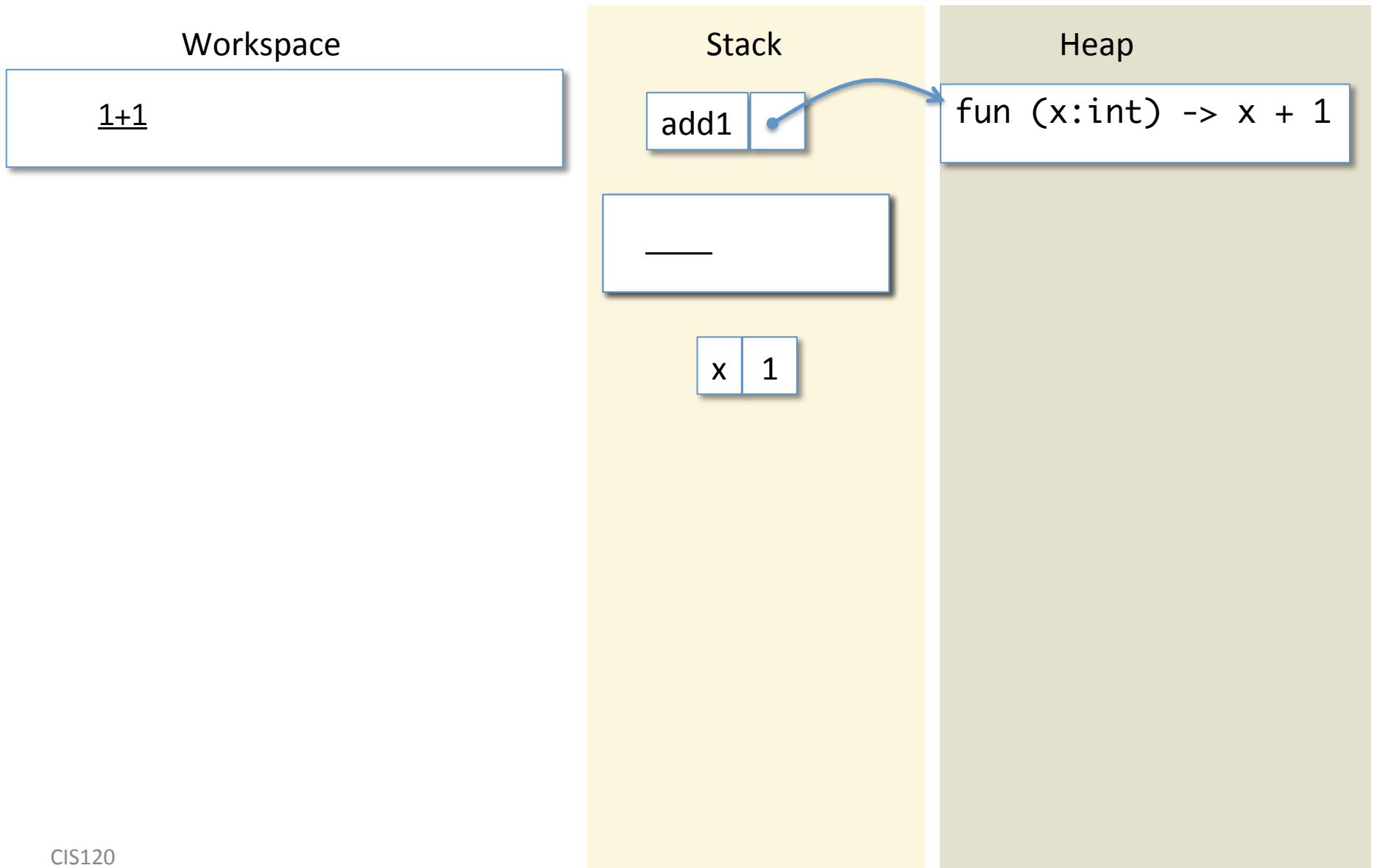
Function Simplification



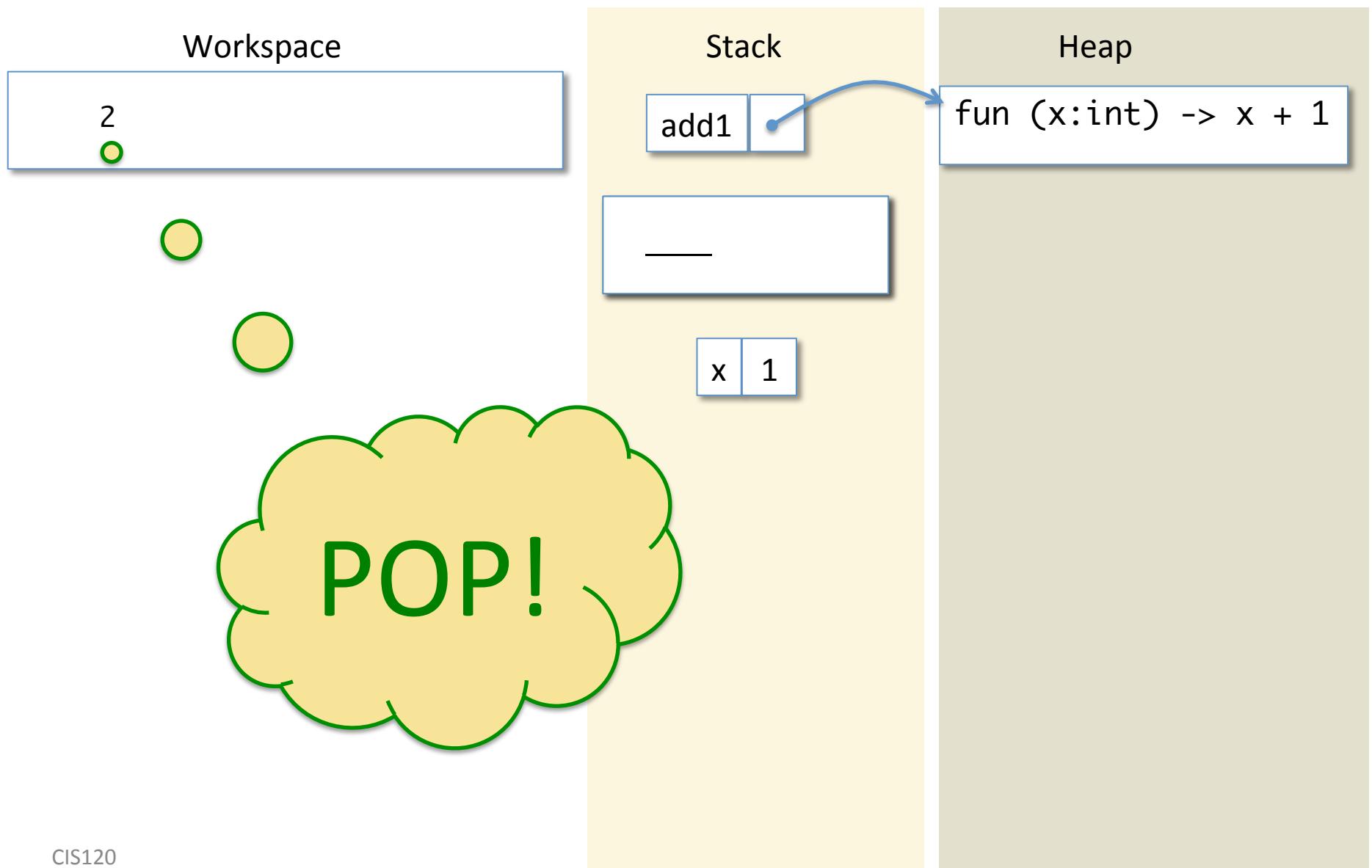
Function Simplification



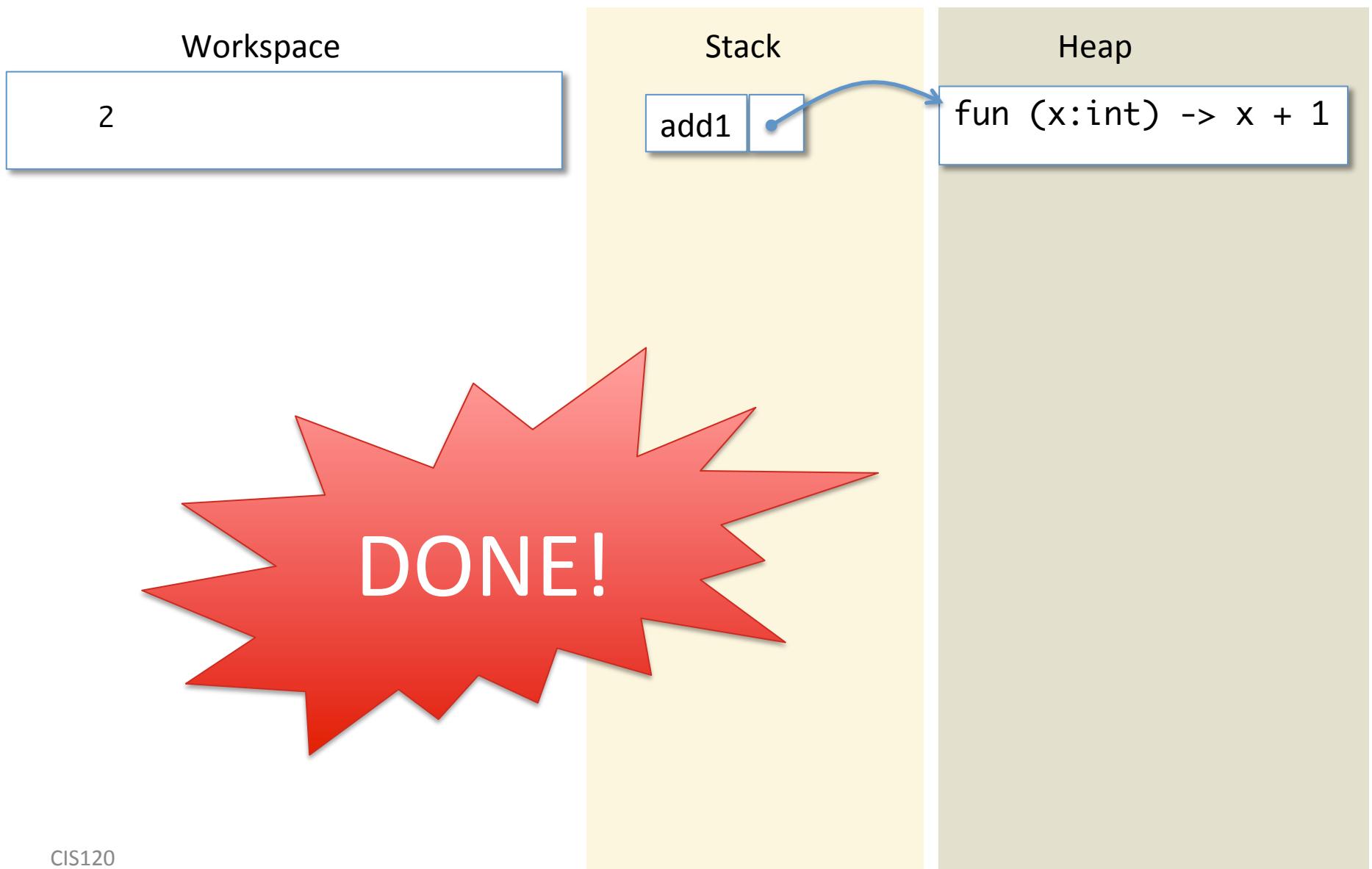
Function Simplification



Function Simplification



Function Simplification



Simplifying Functions

- A function definition “`let rec f (x1:t1)...(xn:tn) = e in body`” is always ready.
 - It is simplified by replacing it with “`let f = fun (x:t1)...(x:tn) = e in body`”
- A function “`fun (x1:t1)...(xn:tn) = e`” is always ready.
 - It is simplified by moving the function to the heap and replacing the function expression with a pointer to that heap data.
- A function *call* is ready if the function and its arguments are all values
 - it is simplified by
 - saving the current workspace contents on the stack
 - adding bindings for the function’s parameter variables (to the actual argument values) to the end of the stack
 - copying the function’s body to the workspace

Function Completion

When the workspace contains just a single value, we *pop the stack* by removing everything back to (and including) the last saved workspace contents.

The value currently in the workspace is substituted for the function application expression in the saved workspace contents, which are put back into the workspace.

If there aren't any saved workspace contents in the stack, the whole computation is finished and the value in the workspace is its final result.

Simplifying pattern matching & recursion

Example

```
let rec append (l1: 'a list) (l2: 'a list) : 'a list =
begin match l1 with
| Nil -> l2
| Cons(h, t) -> Cons(h, append t l2)
end in

let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil)) in

append a b
```

Simplification

Workspace

```
let rec append (l1: 'a list)
  (l2: 'a list) : 'a list =
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
  Cons(h, append t l2)
end in
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil)) in
append a b
```

Stack

Heap

Function Definition

Workspace

```
let rec append (l1: 'a list)
  (l2: 'a list) : 'a list =
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
  Cons(h, append t l2)
end in
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil)) in
append a b
```

Stack

Heap

Rewrite to a “fun”

Workspace

```
let append =
  fun (l1: 'a list)
    (l2: 'a list) ->
  begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
    Cons(h, append t l2)
  end in
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil)) in
append a b
```

Stack

Heap

Function Expression

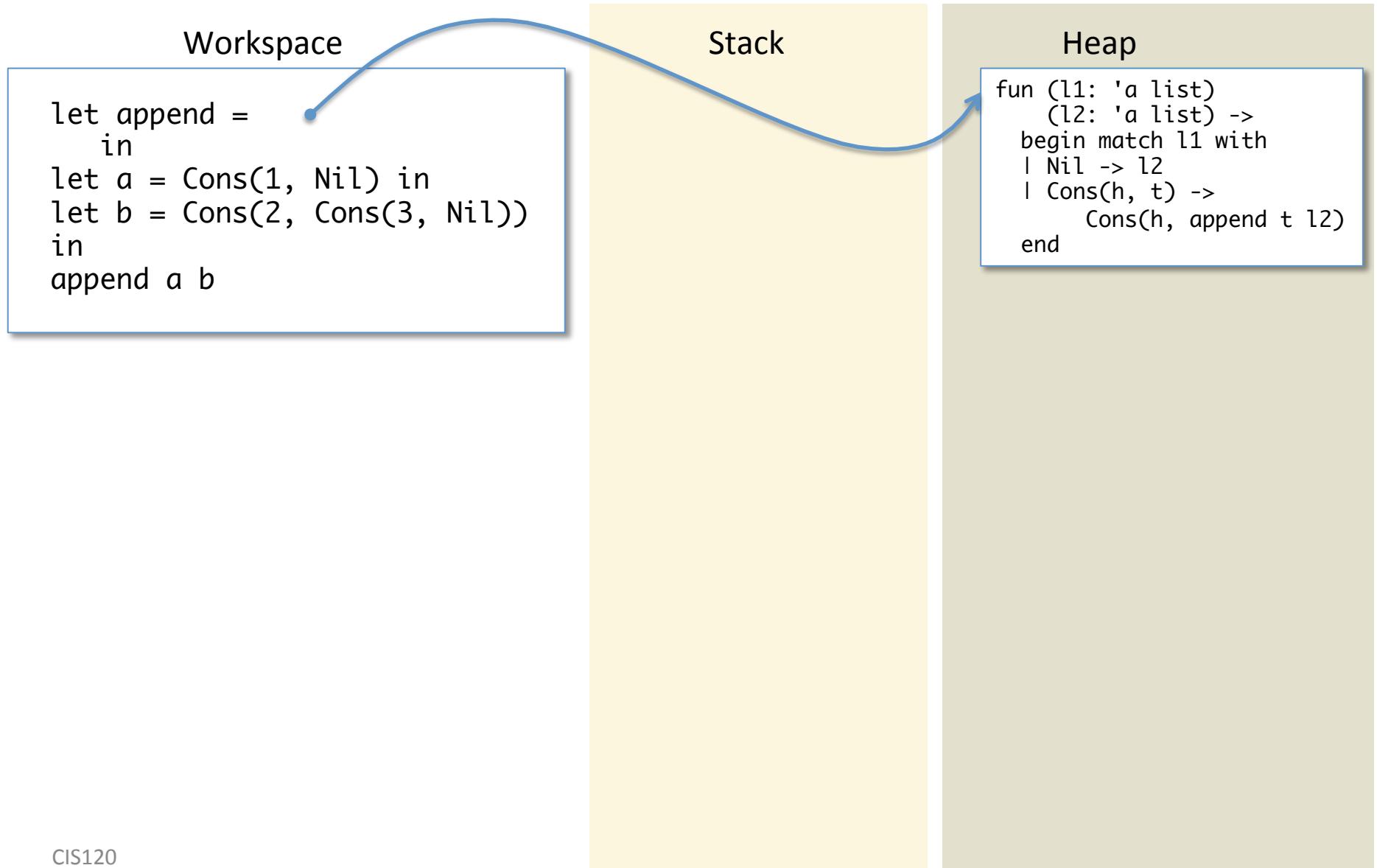
Workspace

```
let append =
  fun (l1: 'a list)
    (l2: 'a list) ->
  begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
    Cons(h, append t l2)
  end in
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil)) in
append a b
```

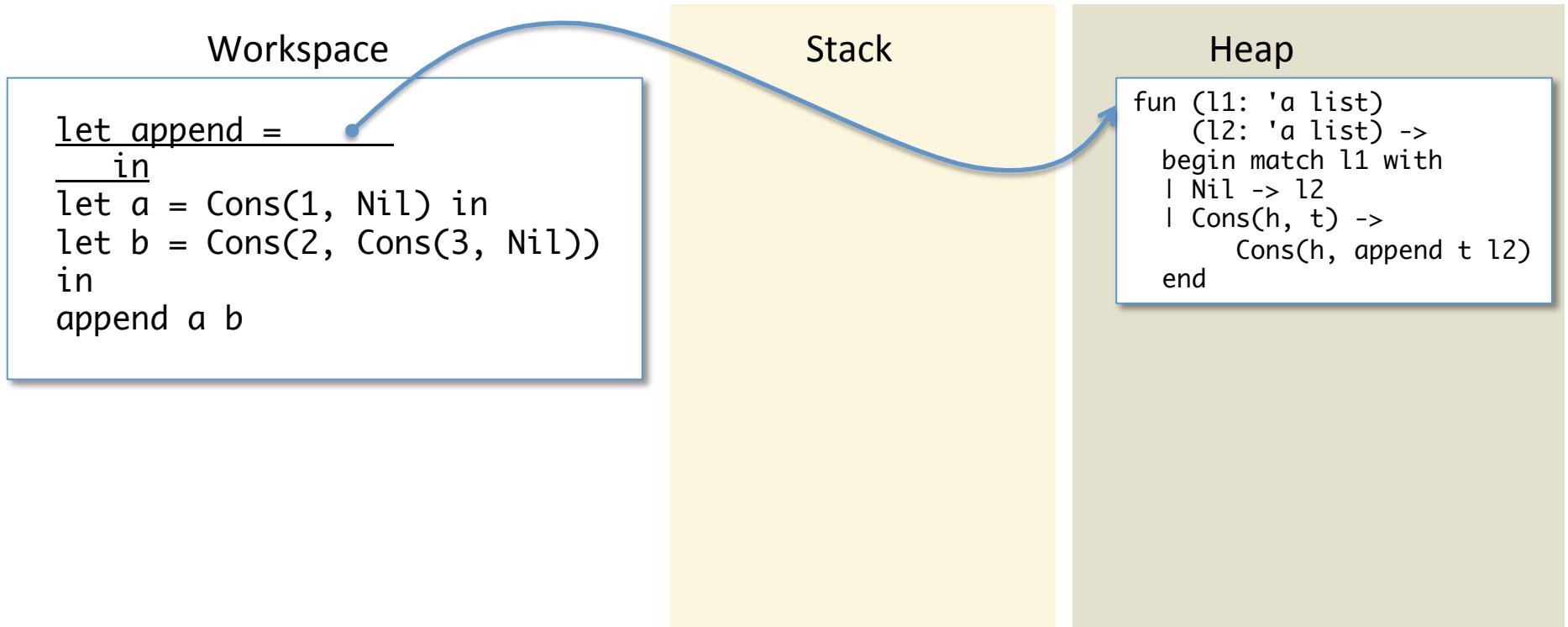
Stack

Heap

Copy to the Heap, Replace w/Reference



Let Expression



Note that the reference to a function in the heap is a value.

Create a Stack Binding

Workspace

```
let a = Cons(1, Nil) in  
let b = Cons(2, Cons(3, Nil))  
in  
append a b
```

Stack

append

Heap

```
fun (l1: 'a list)  
    (l2: 'a list) ->  
begin match l1 with  
| Nil -> l2  
| Cons(h, t) ->  
    Cons(h, append t l2)  
end
```

Allocate a Nil cell

Workspace

```
let a = Cons(1, Nil) in  
let b = Cons(2, Cons(3, Nil))  
in  
append a b
```

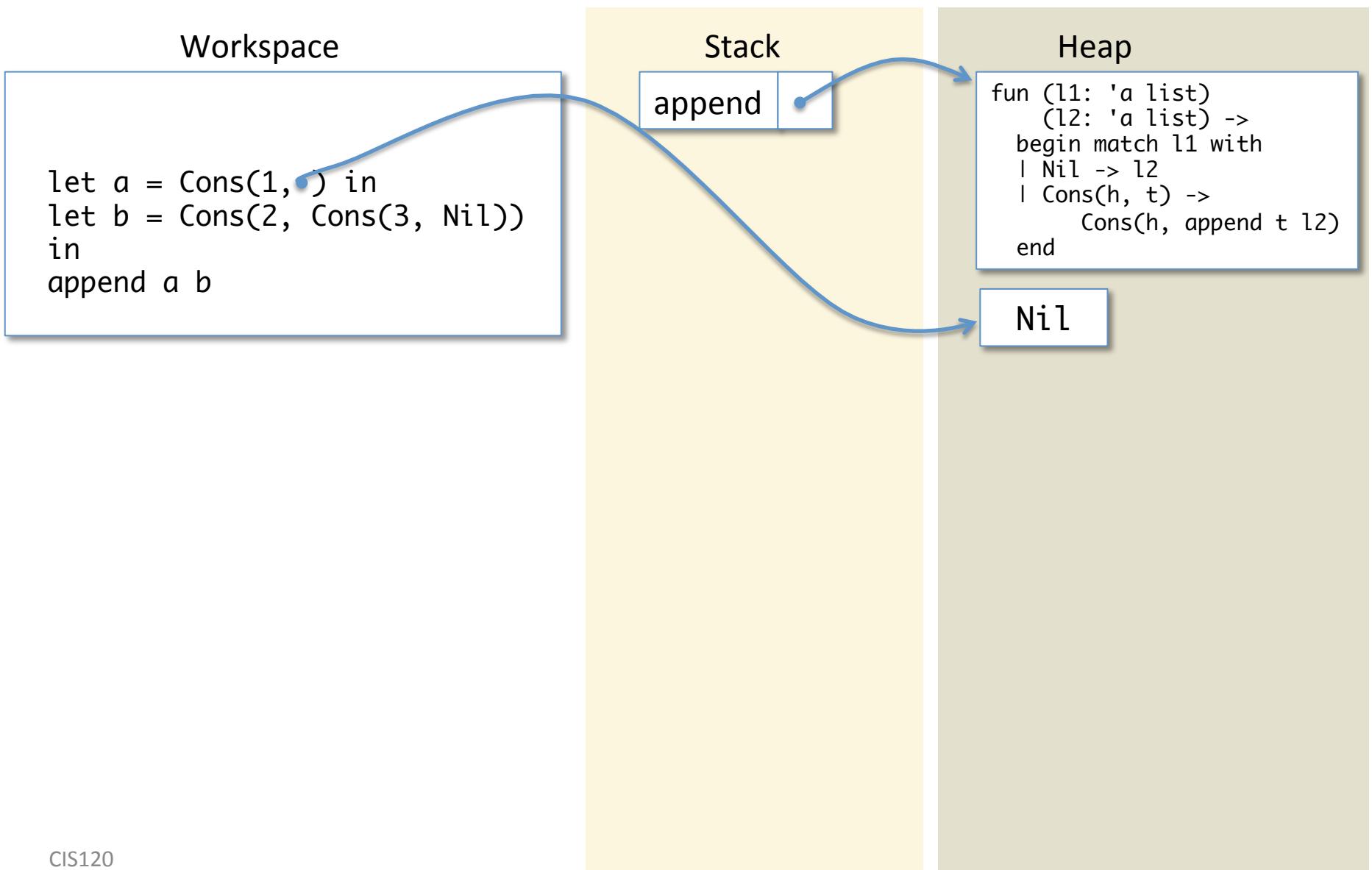
Stack

append

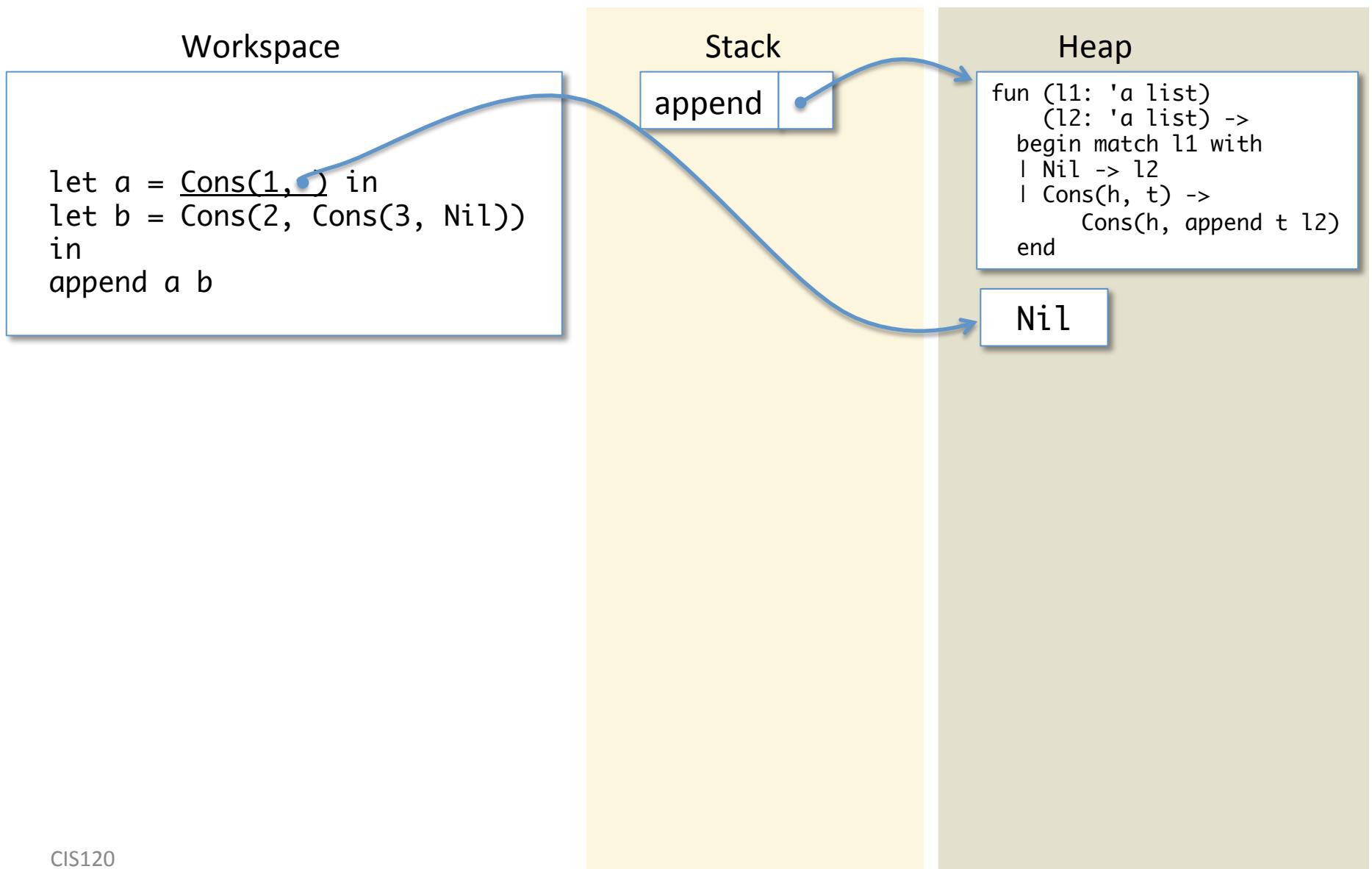
Heap

```
fun (l1: 'a list)  
  (l2: 'a list) ->  
begin match l1 with  
| Nil -> l2  
| Cons(h, t) ->  
  Cons(h, append t l2)  
end
```

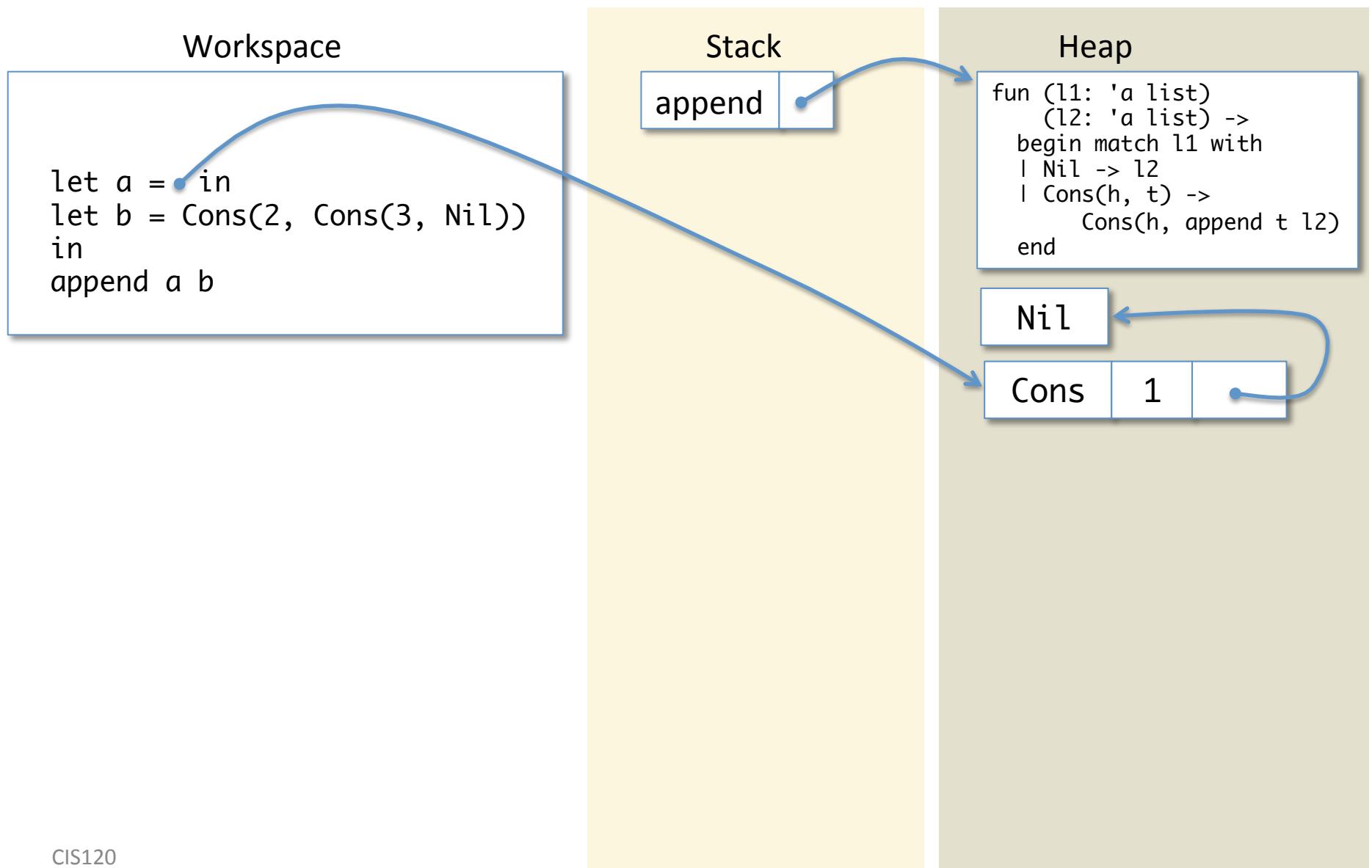
Allocate a Nil cell



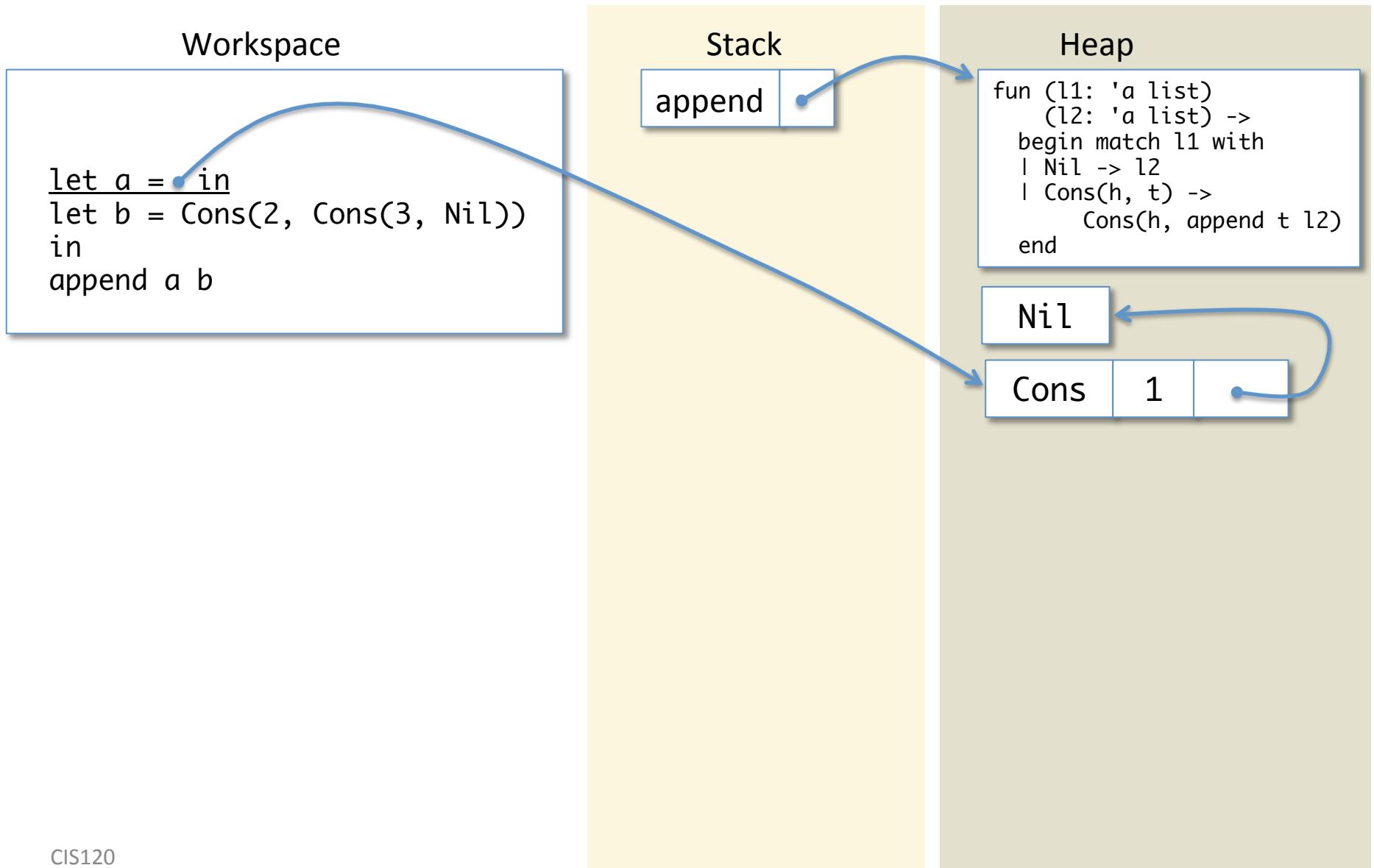
Allocate a Cons cell



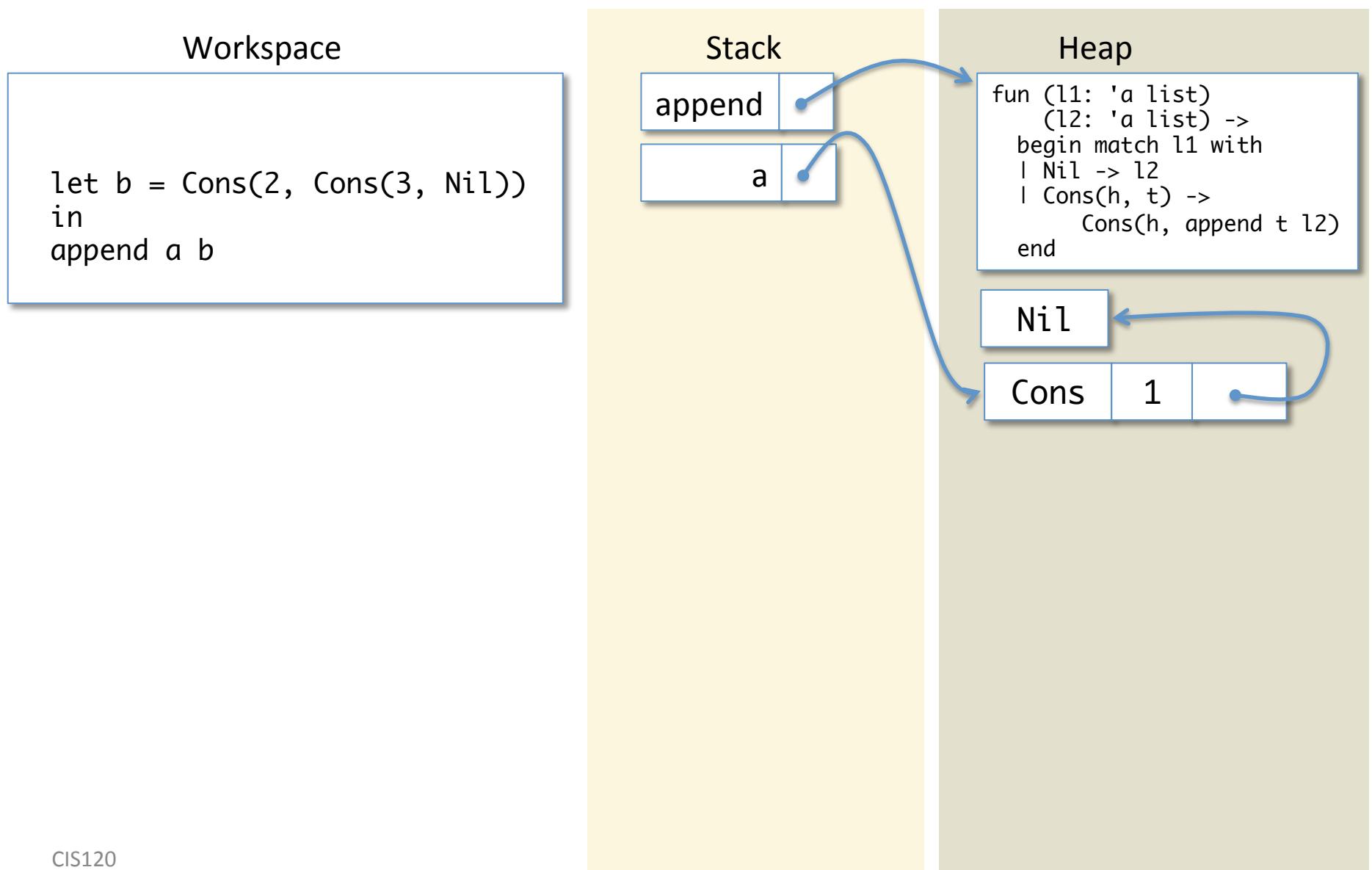
Allocate a Cons cell



Let Expression



Create a Stack Binding



Allocate a Nil cell

Workspace

```
let b = Cons(2, Cons(3, Nil))  
in  
append a b
```

Stack

| | |
|--------|---|
| append | • |
| a | • |

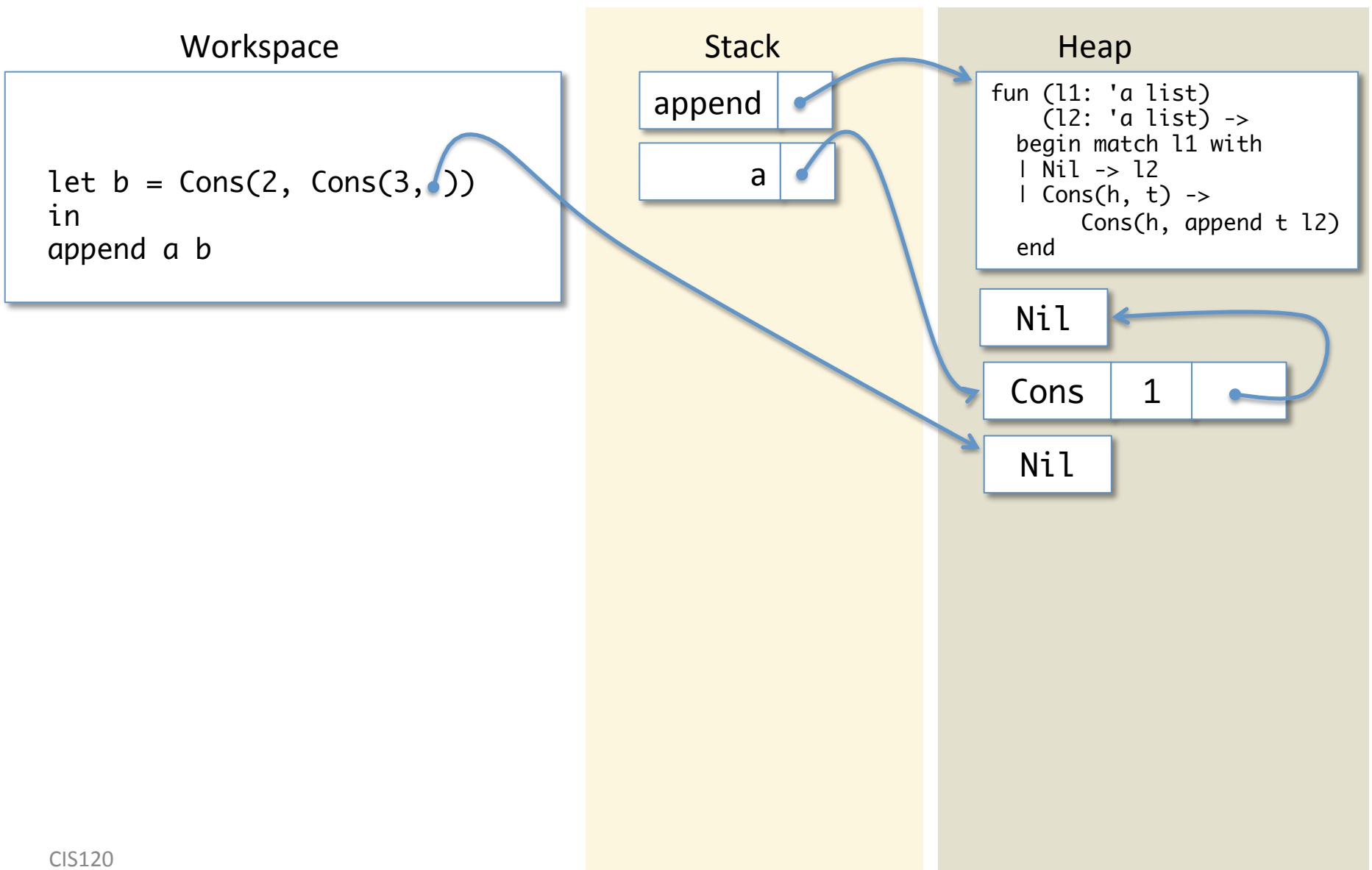
Heap

```
fun (l1: 'a list)  
  (l2: 'a list) ->  
begin match l1 with  
| Nil -> l2  
| Cons(h, t) ->  
  Cons(h, append t l2)  
end
```

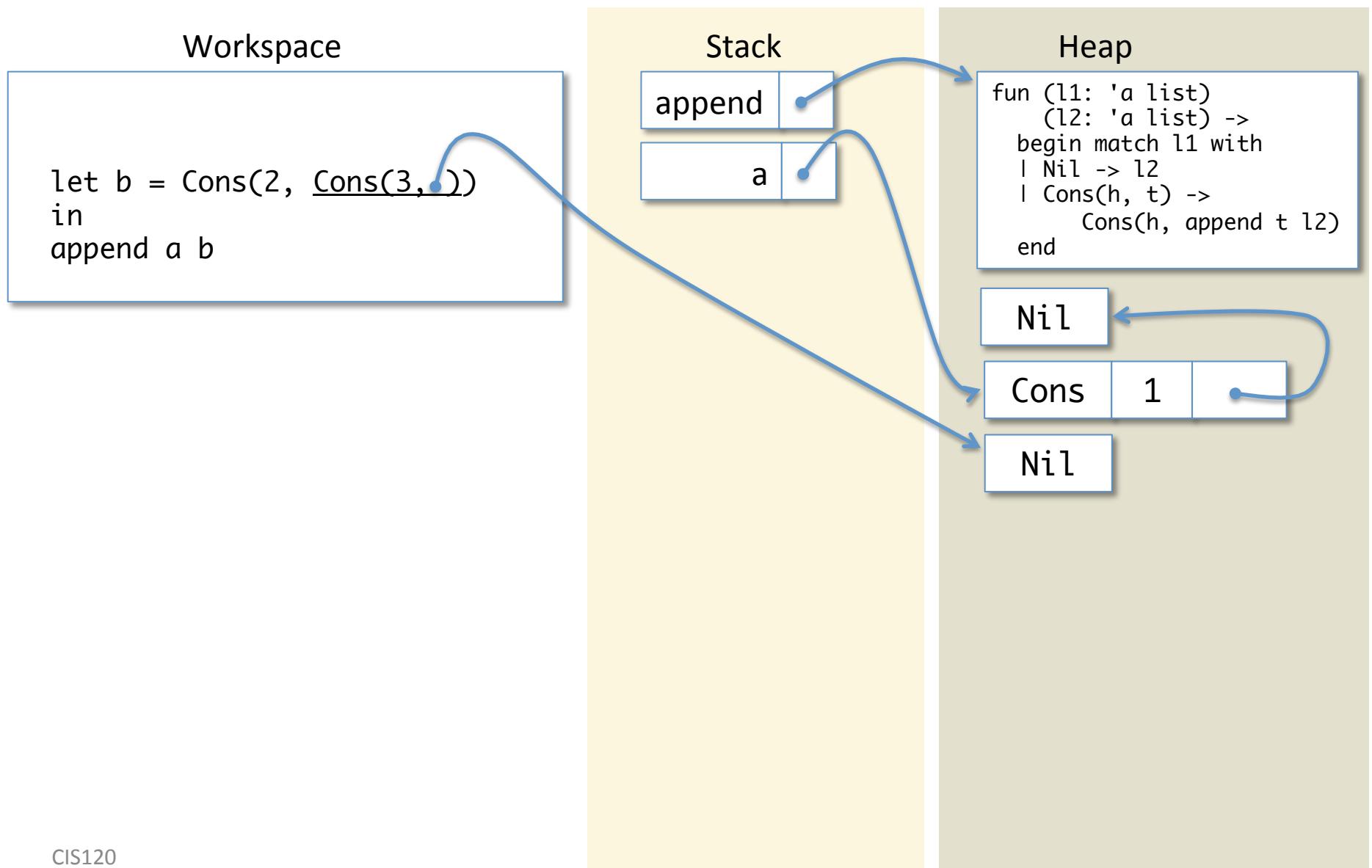
Nil

Cons 1 •

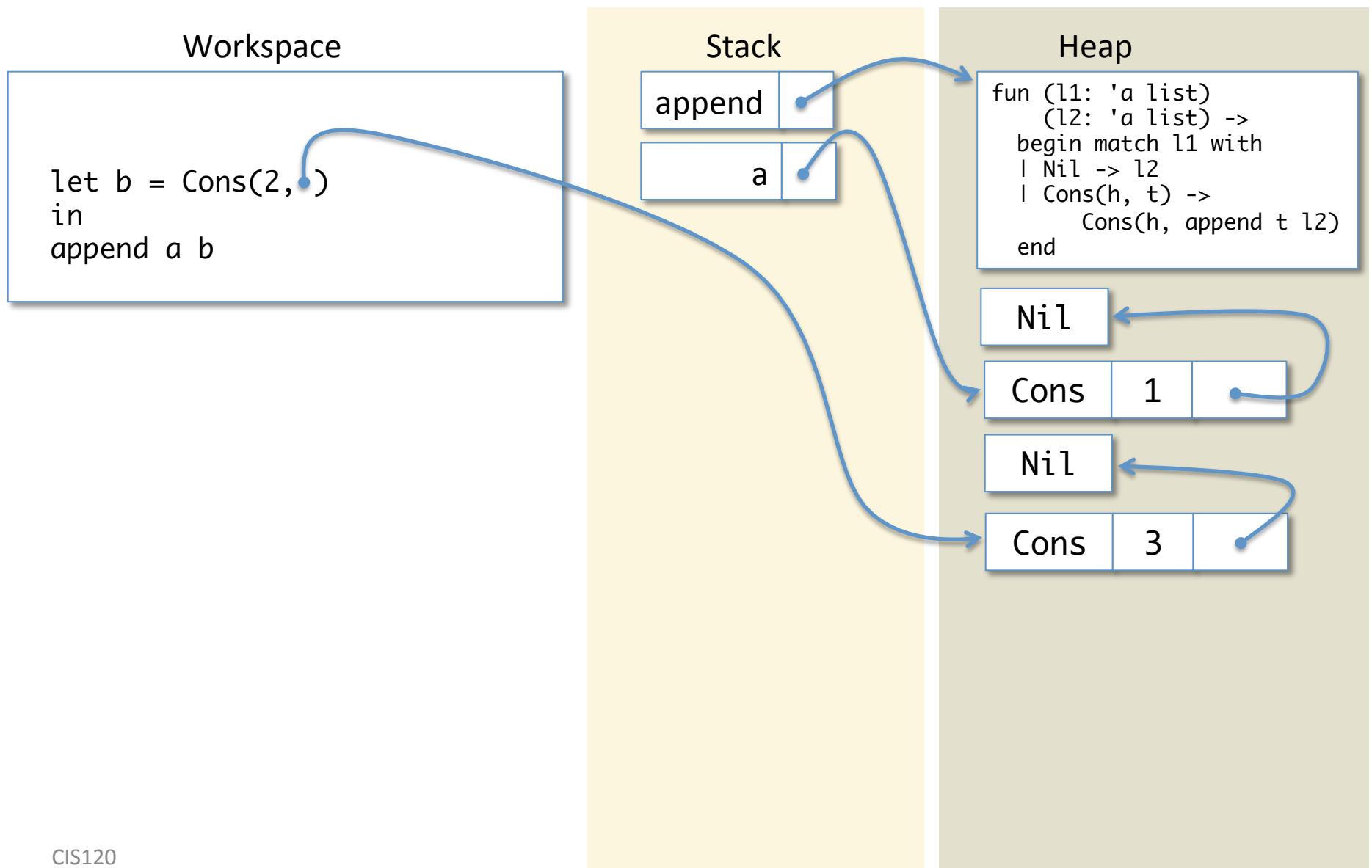
Allocate a Nil cell



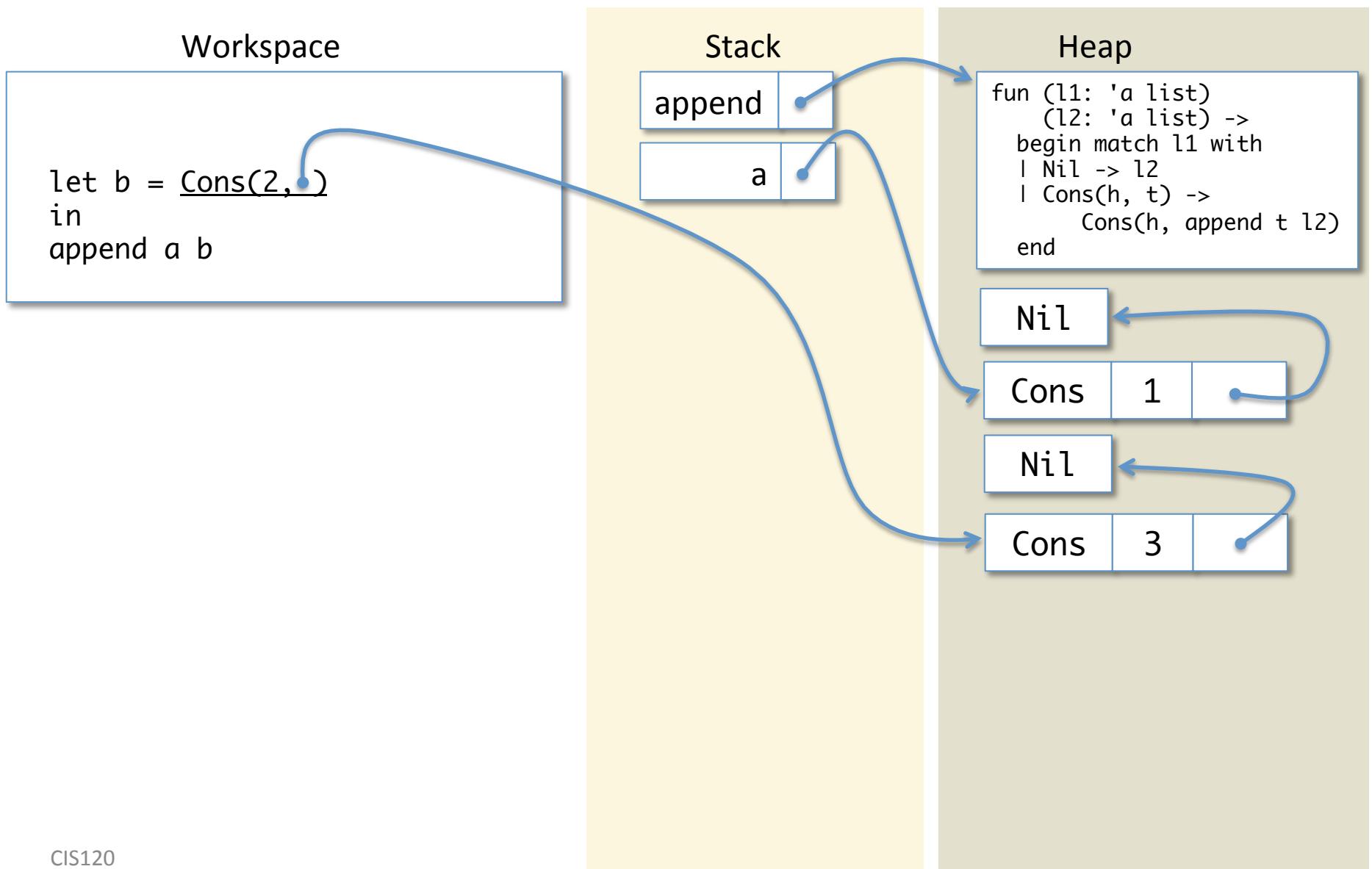
Allocate a Cons cell



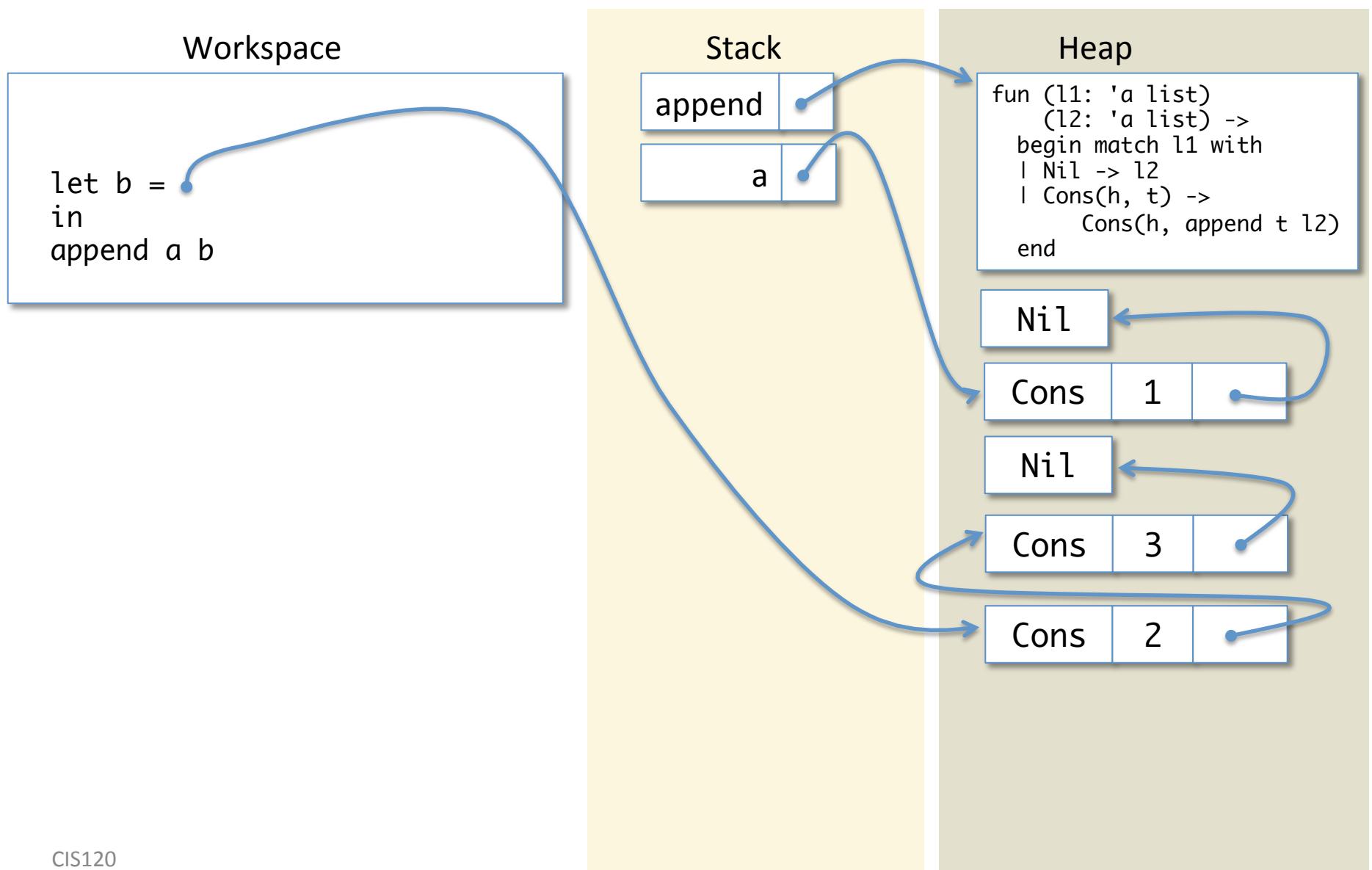
Allocate a Cons cell



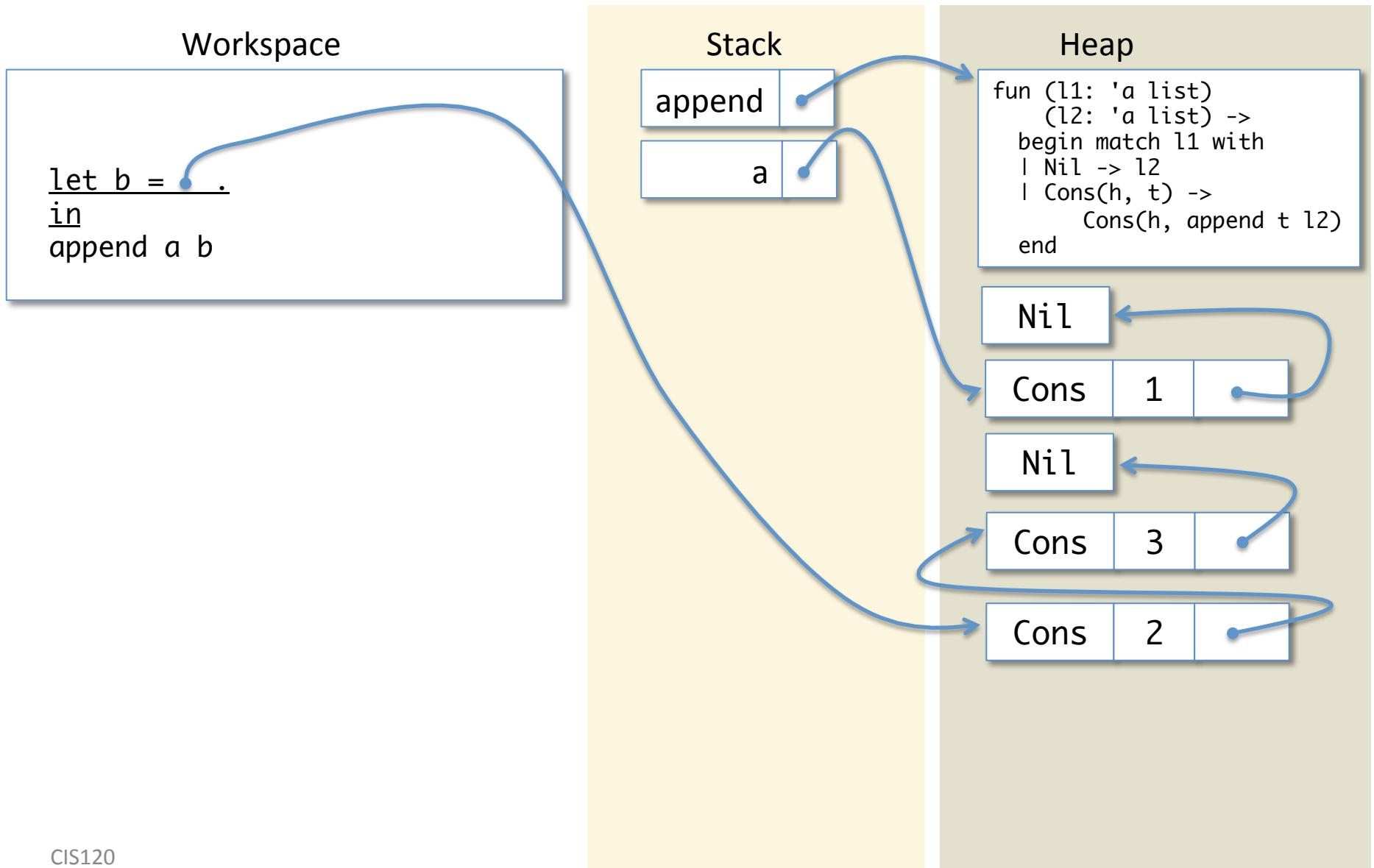
Allocate a Cons cell



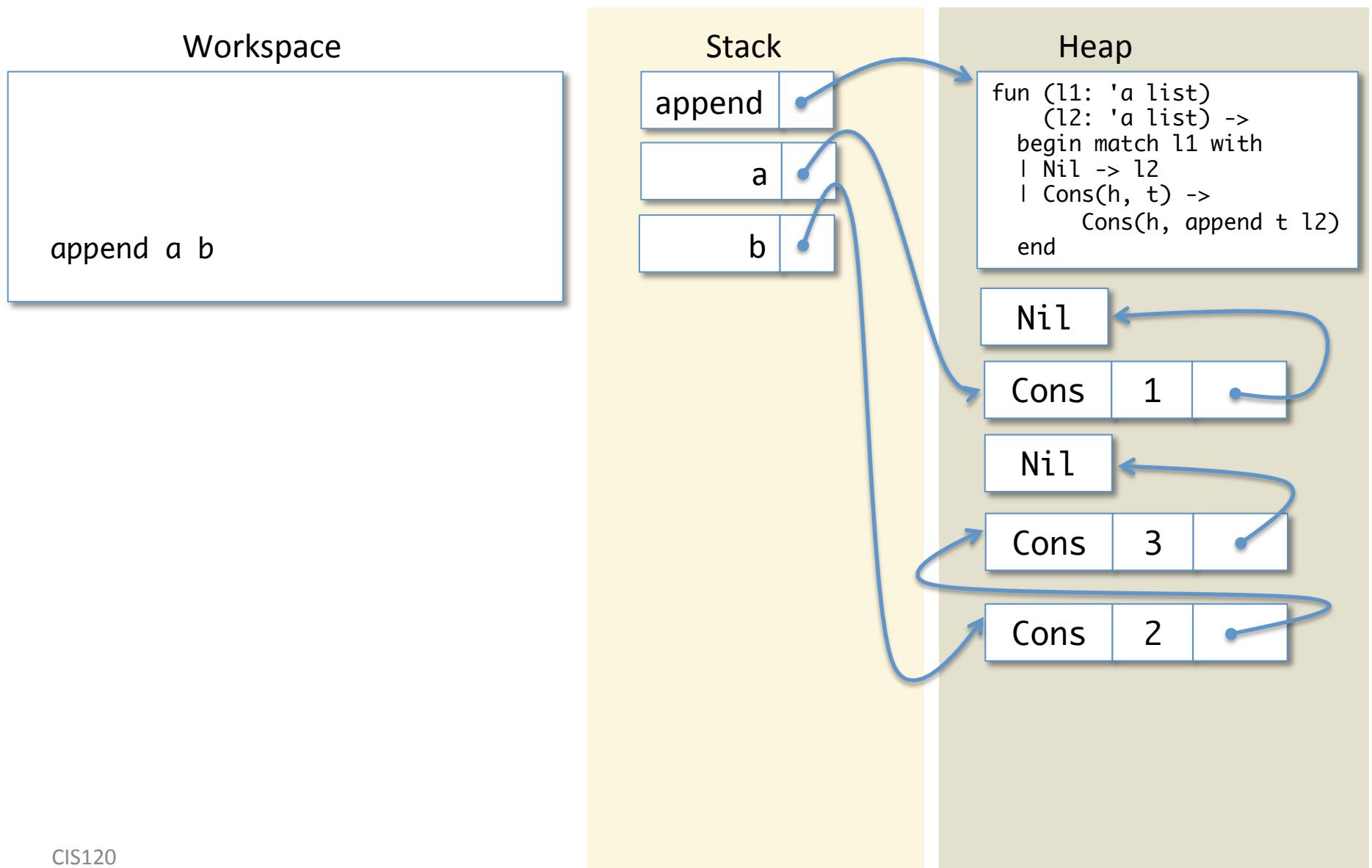
Allocate a Cons cell



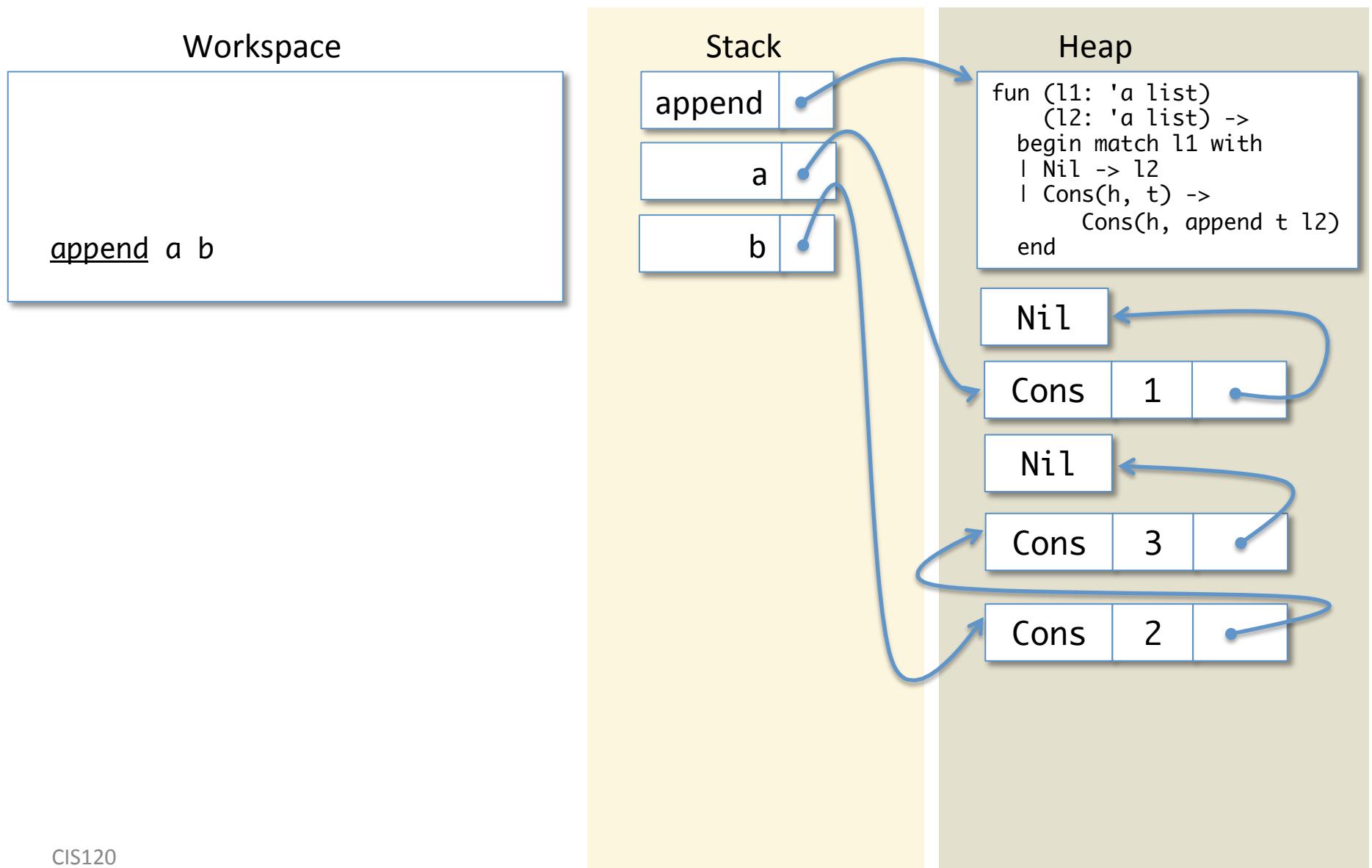
Let Expression



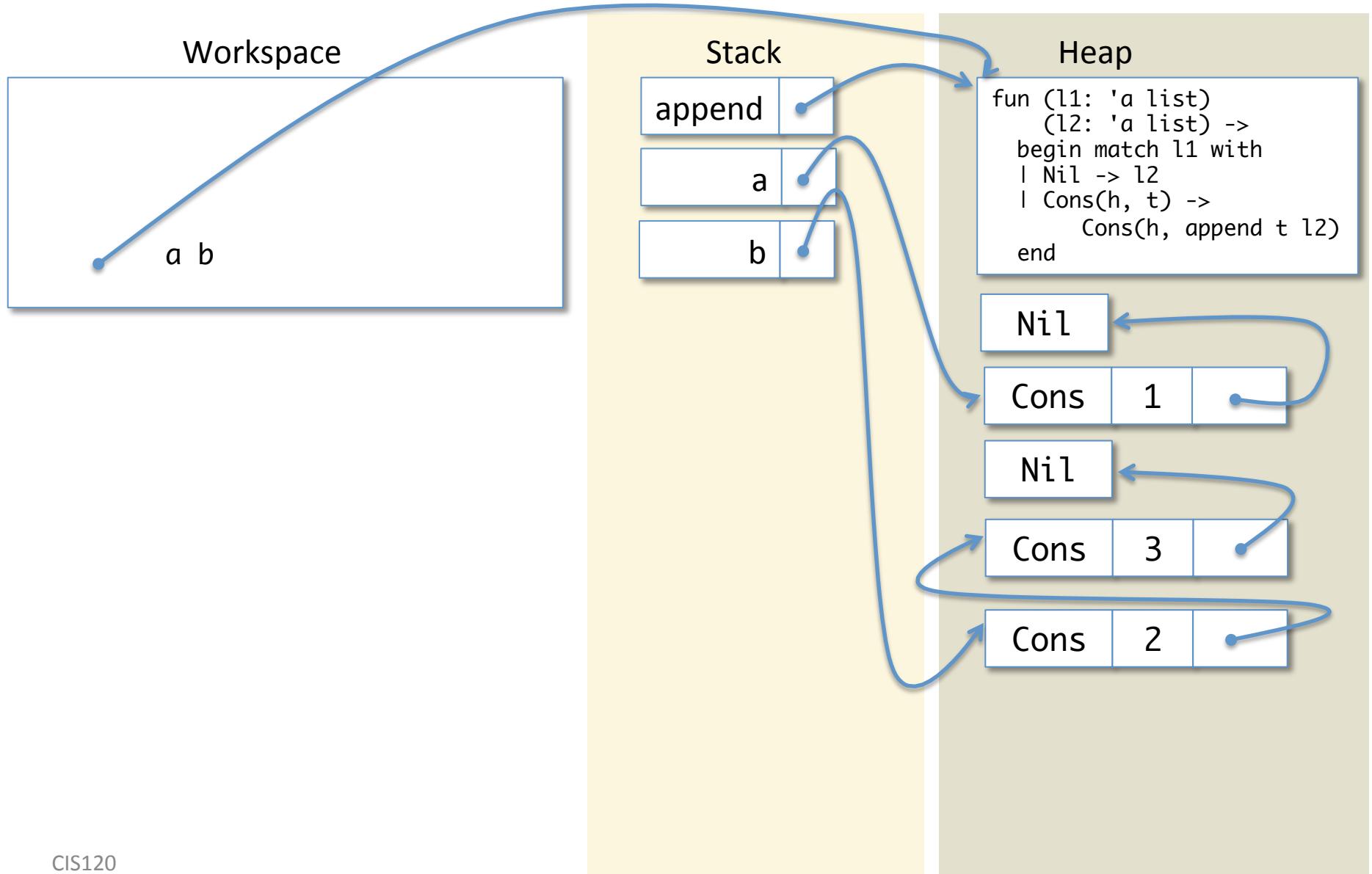
Create a Stack Binding



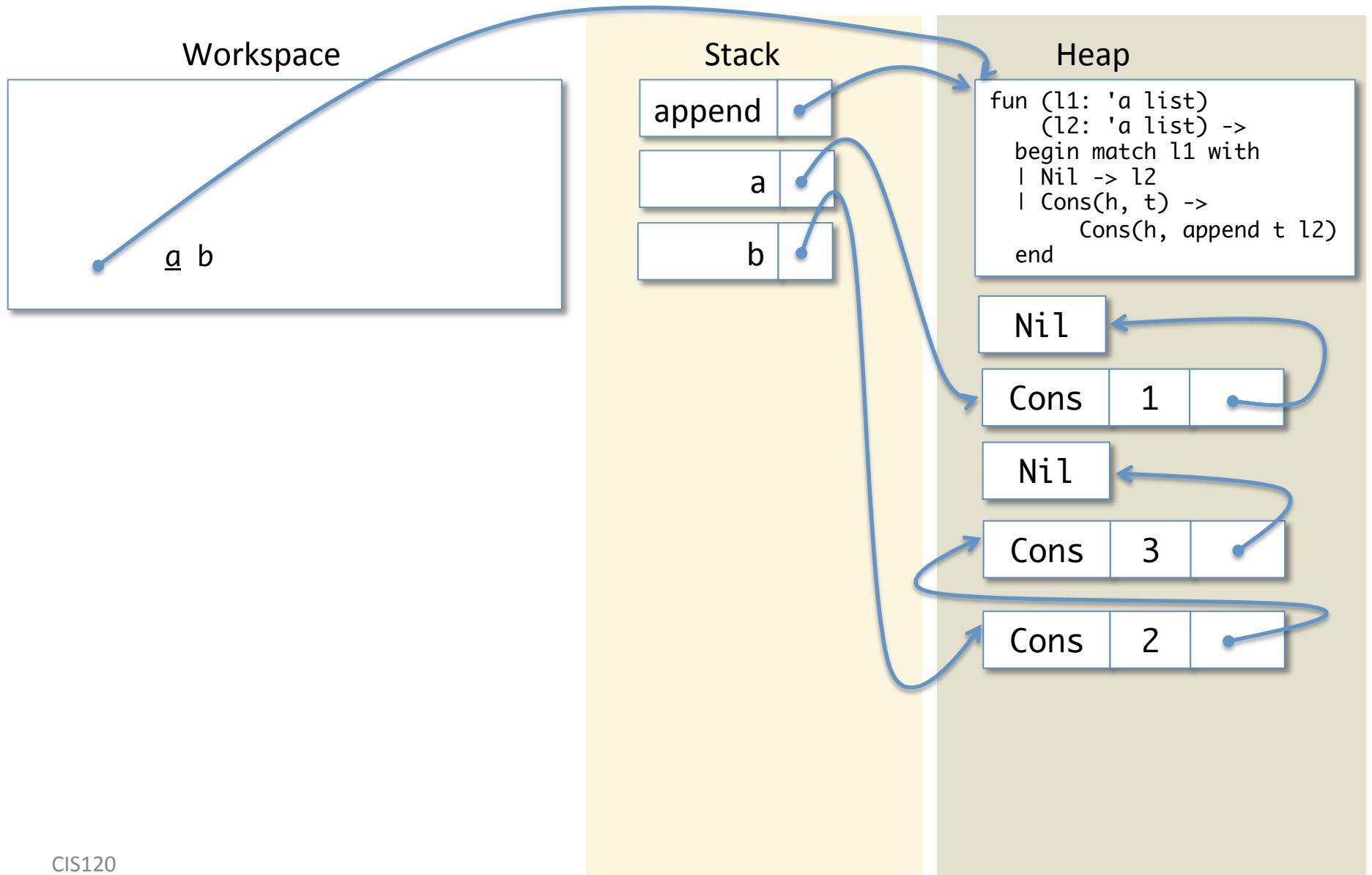
Lookup 'append'



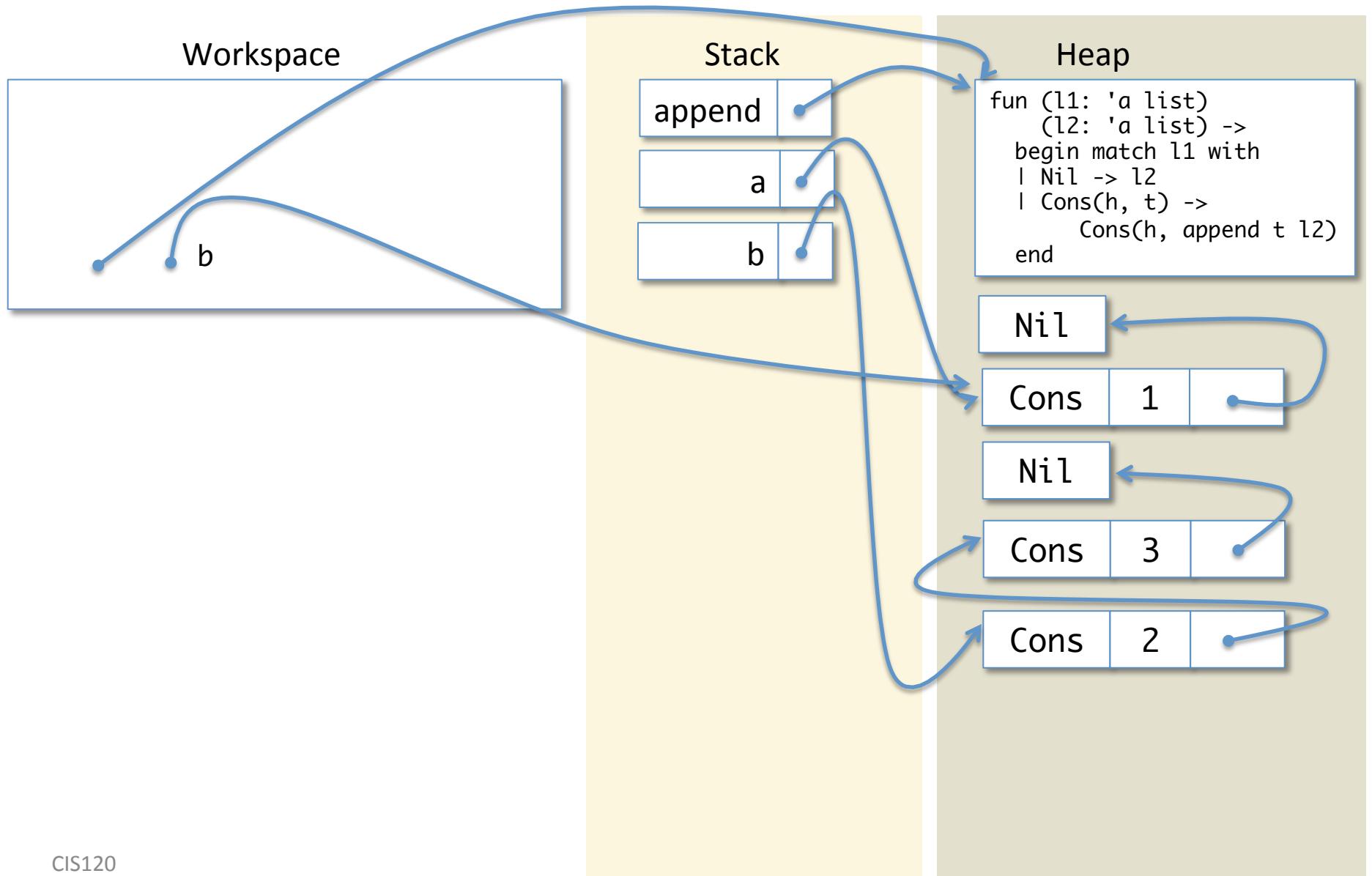
Lookup 'append'



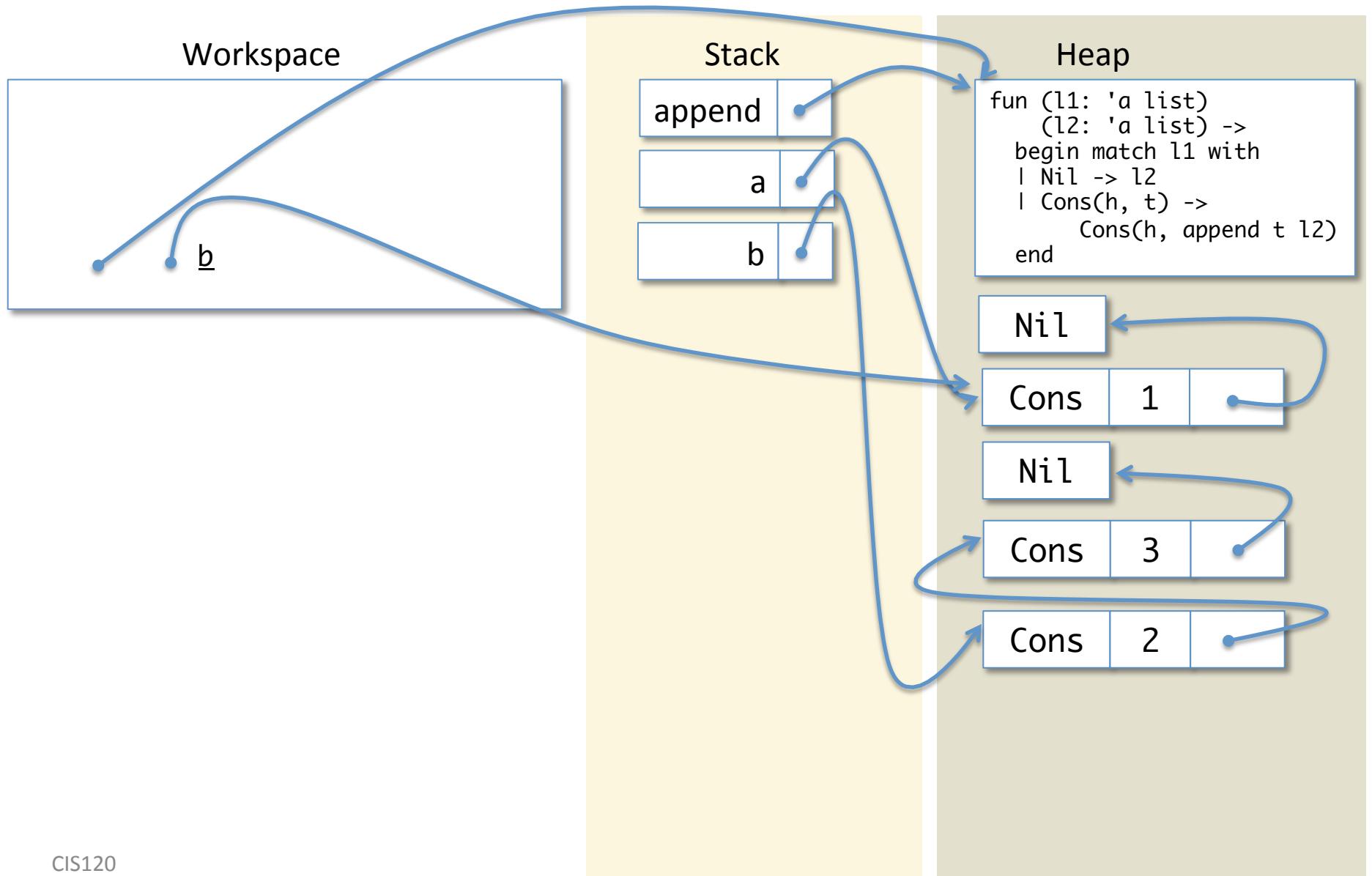
Lookup 'a'



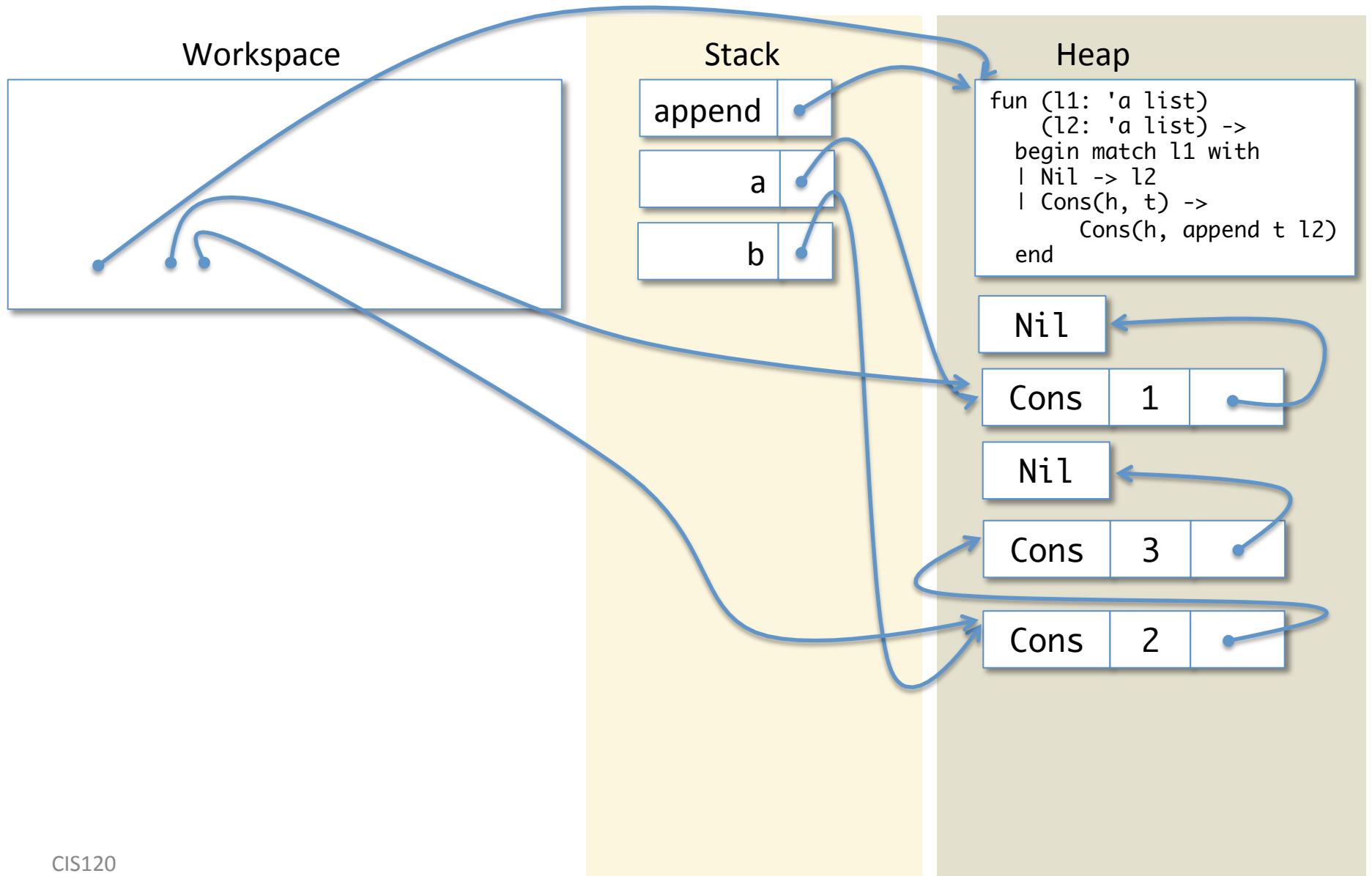
Lookup 'a'



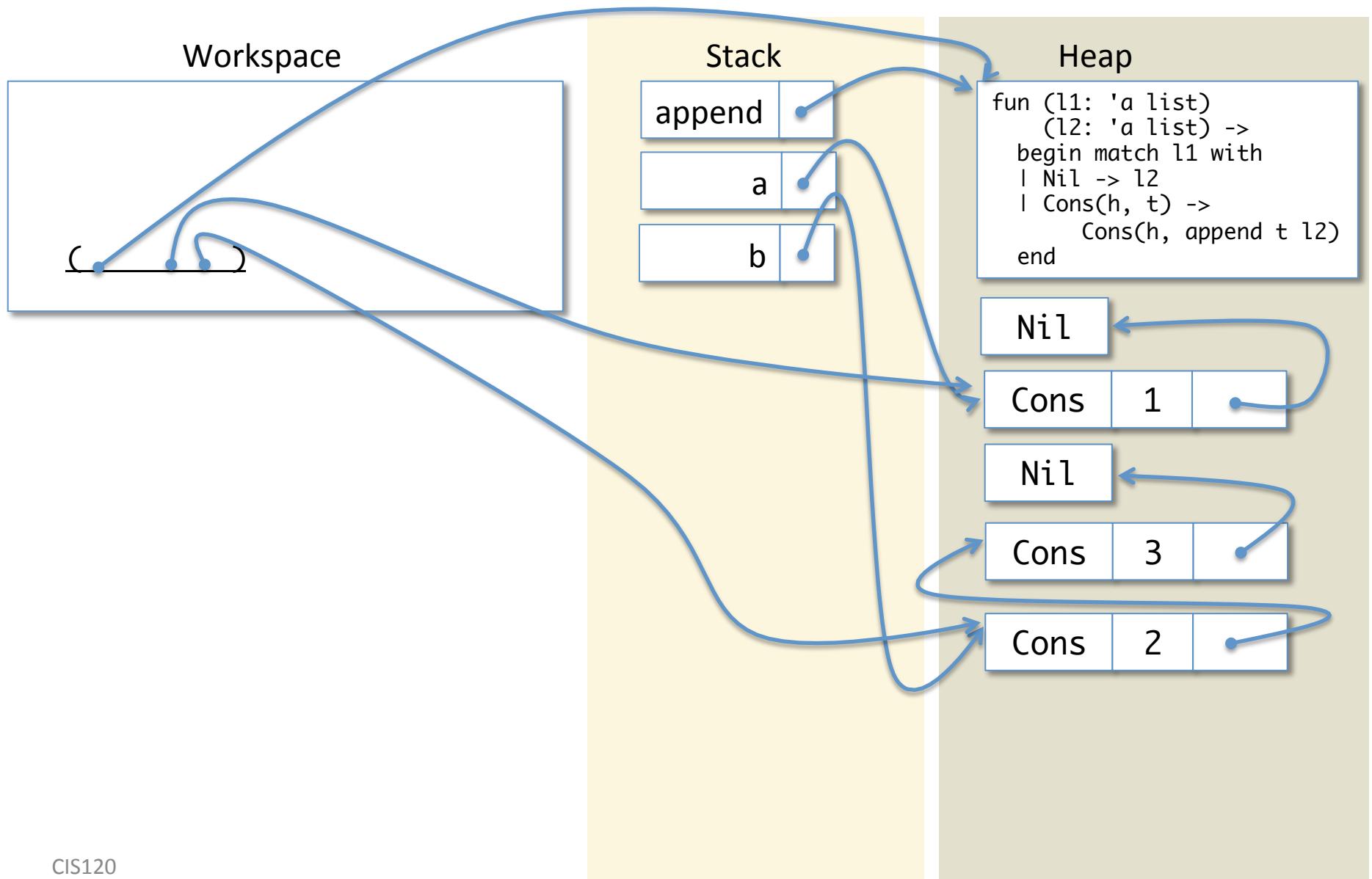
Lookup 'b'



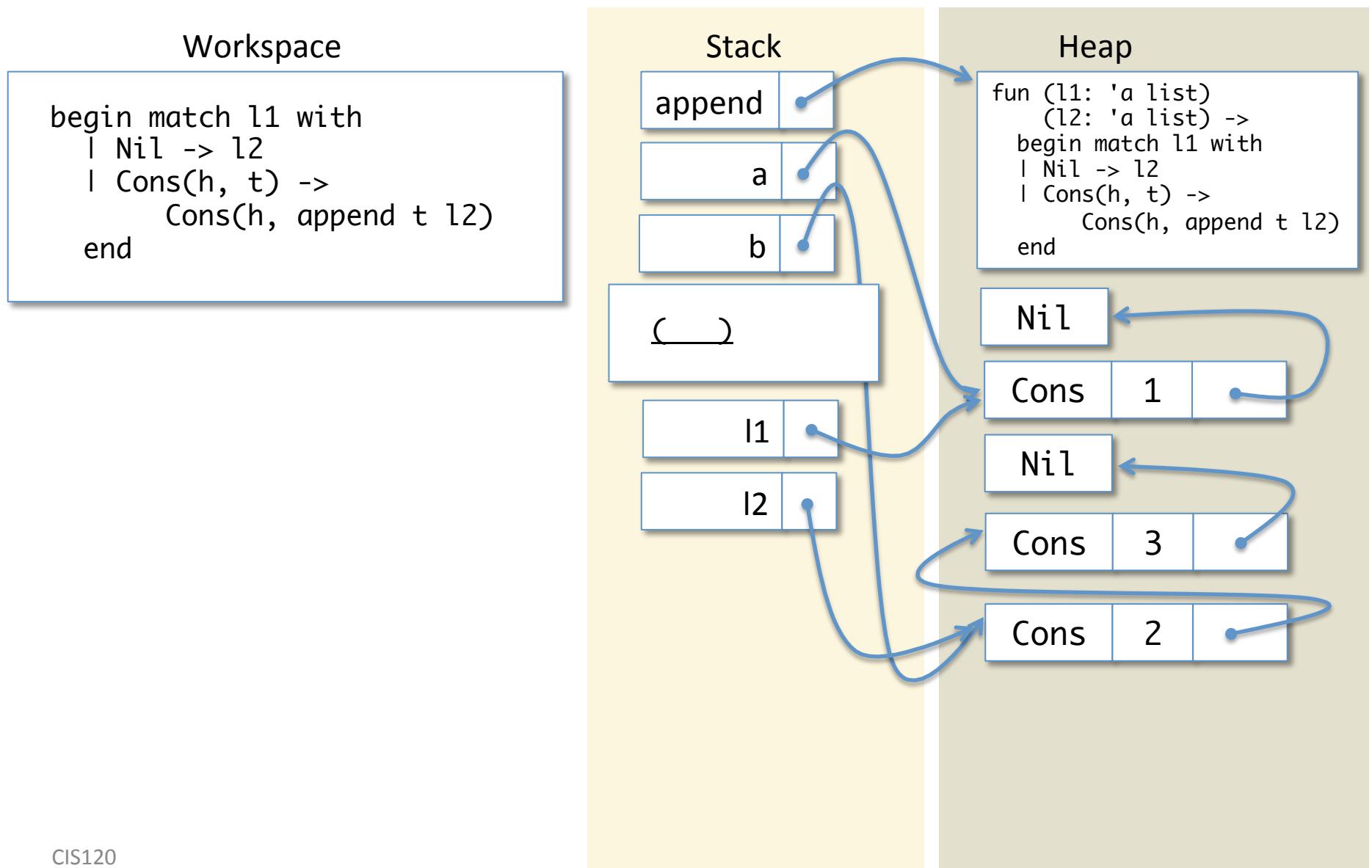
Lookup 'b'



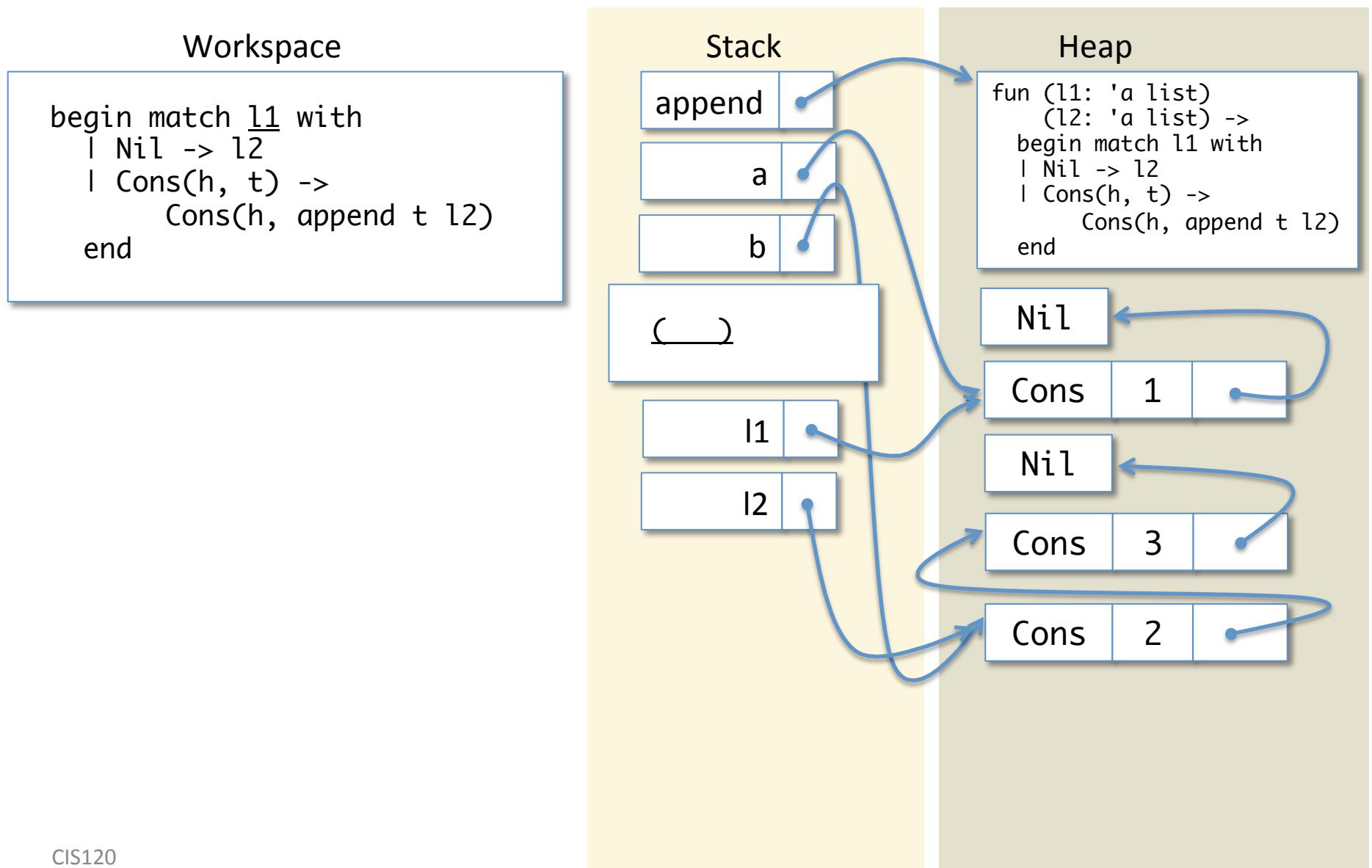
Do the Function call



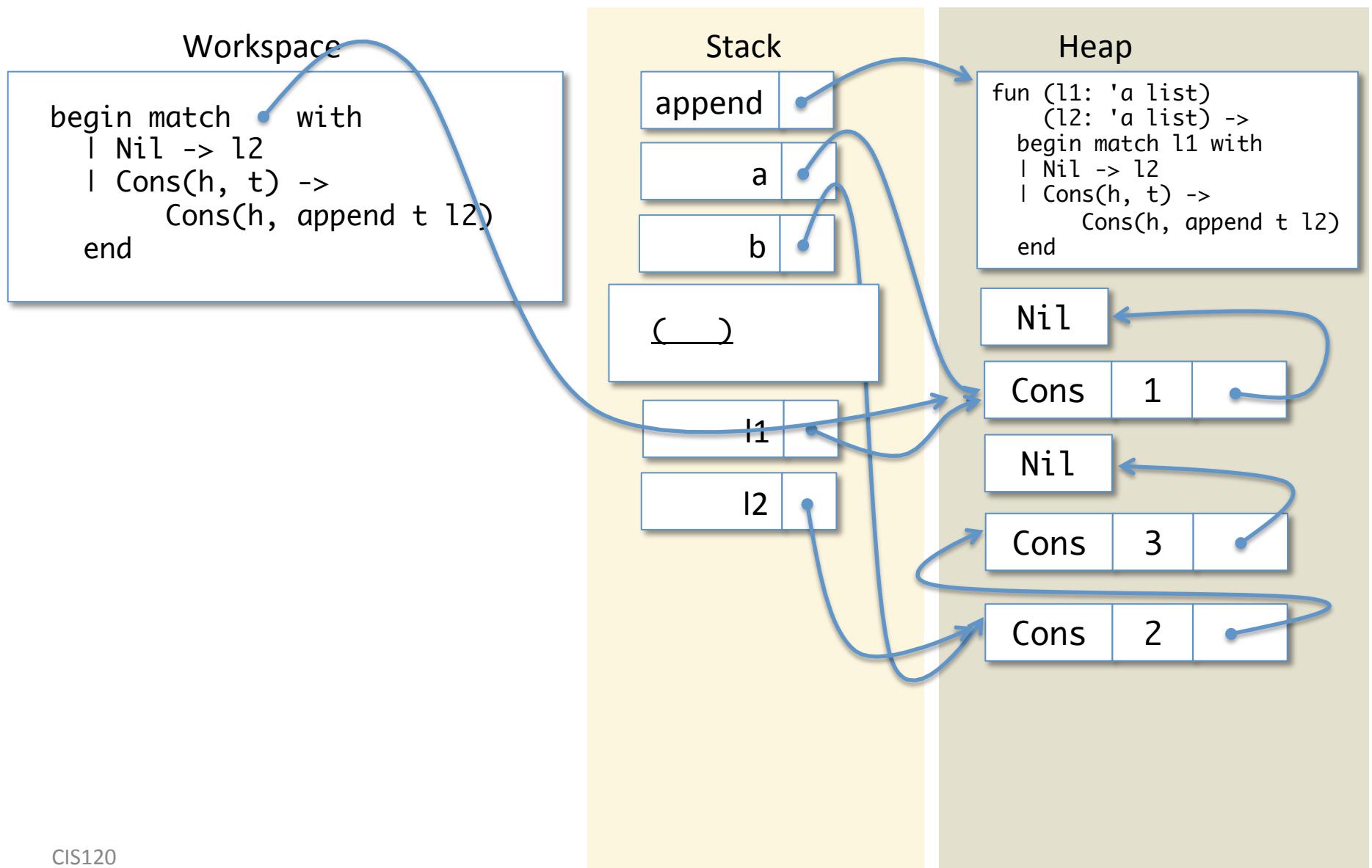
Save Workspace; push l1, l2



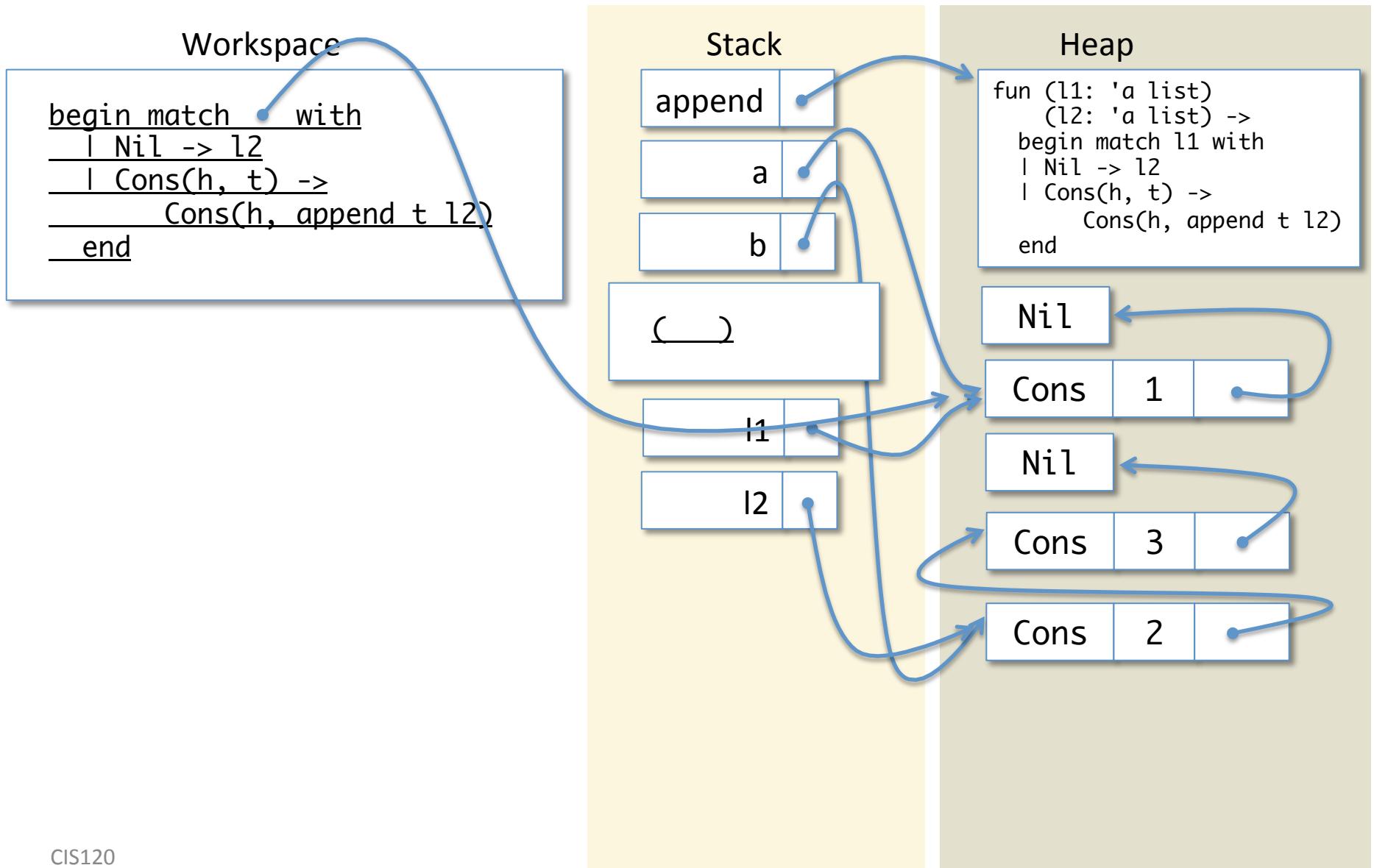
Lookup l1



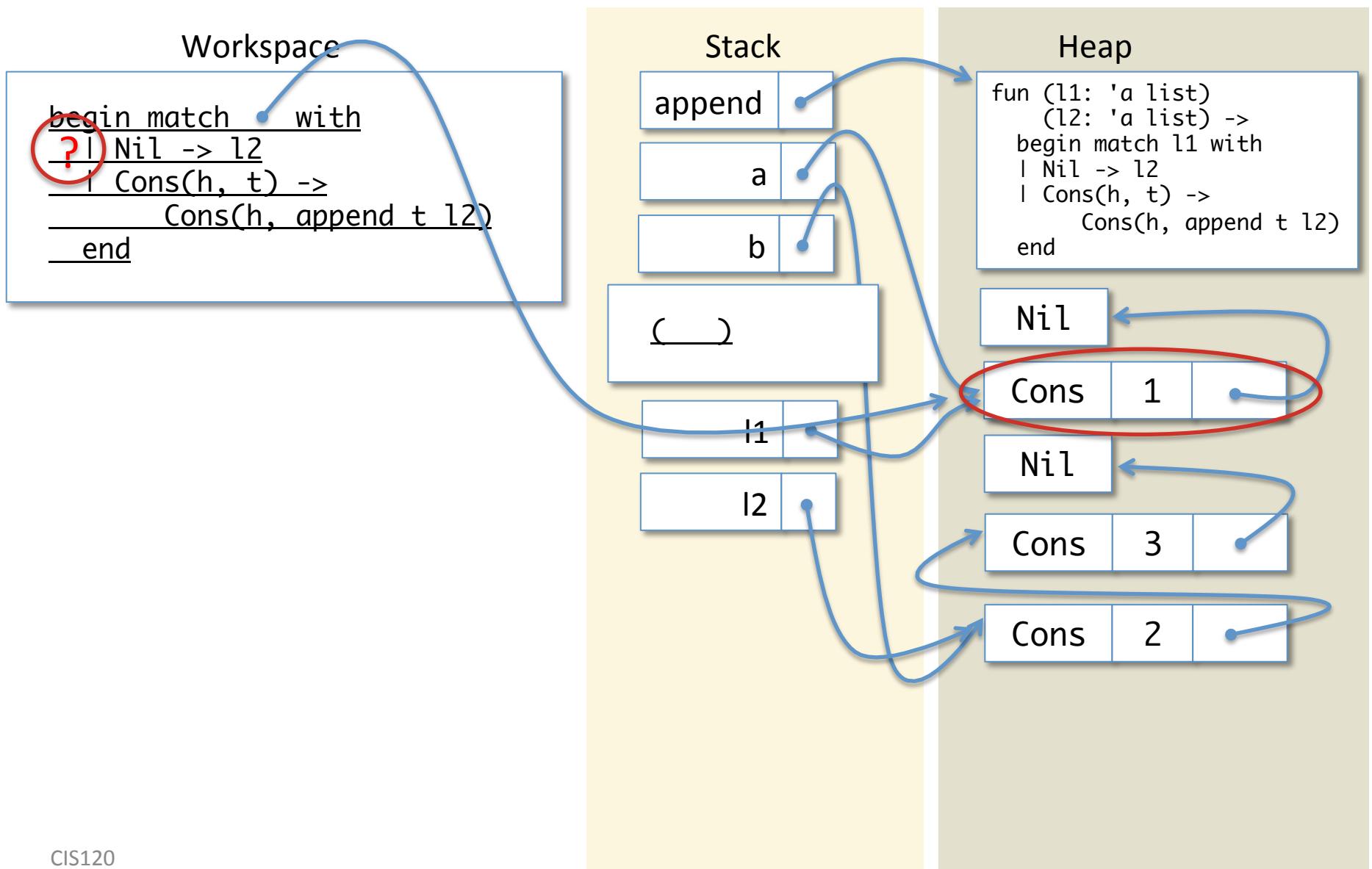
Lookup l1



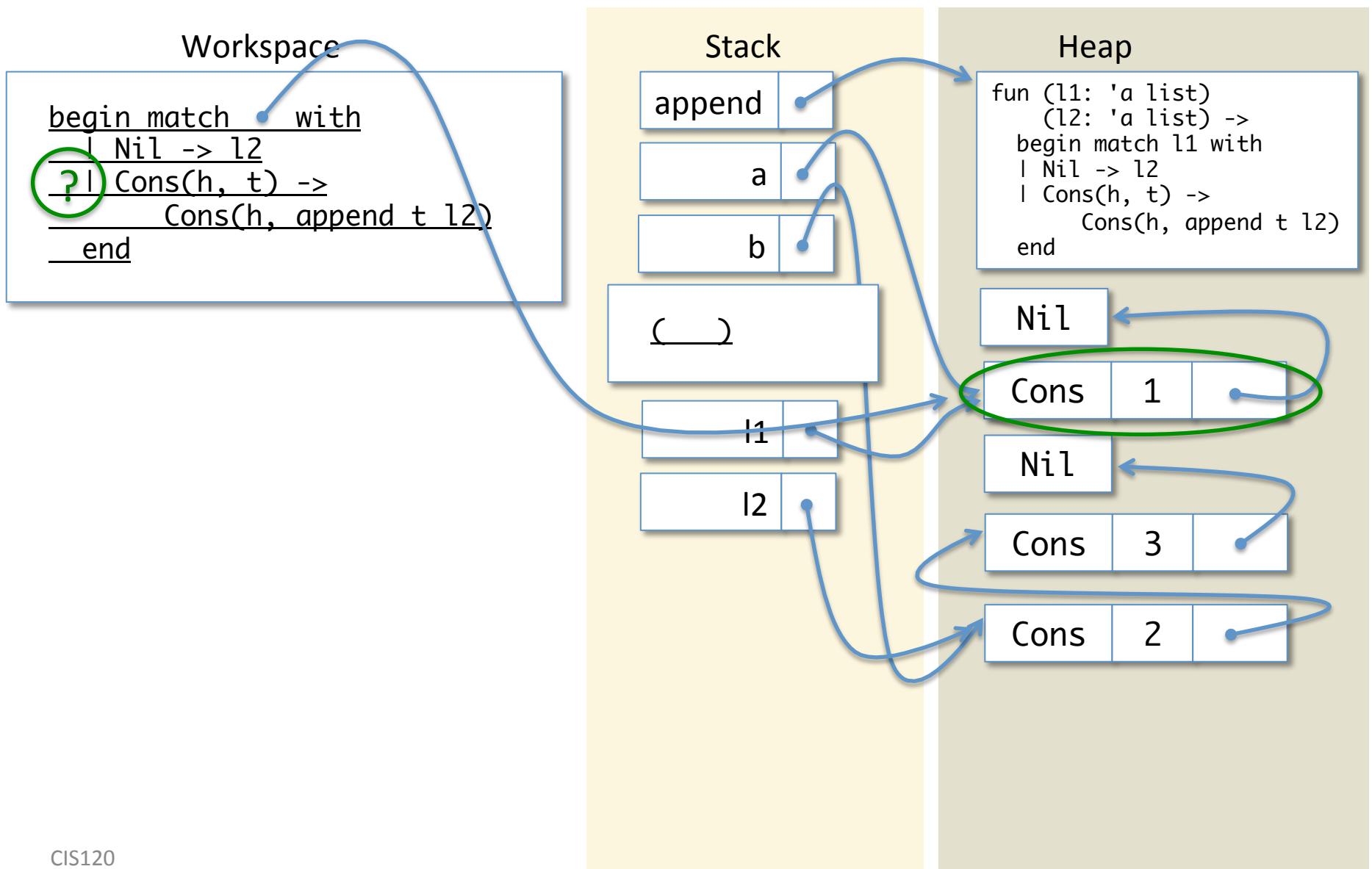
Match Expression



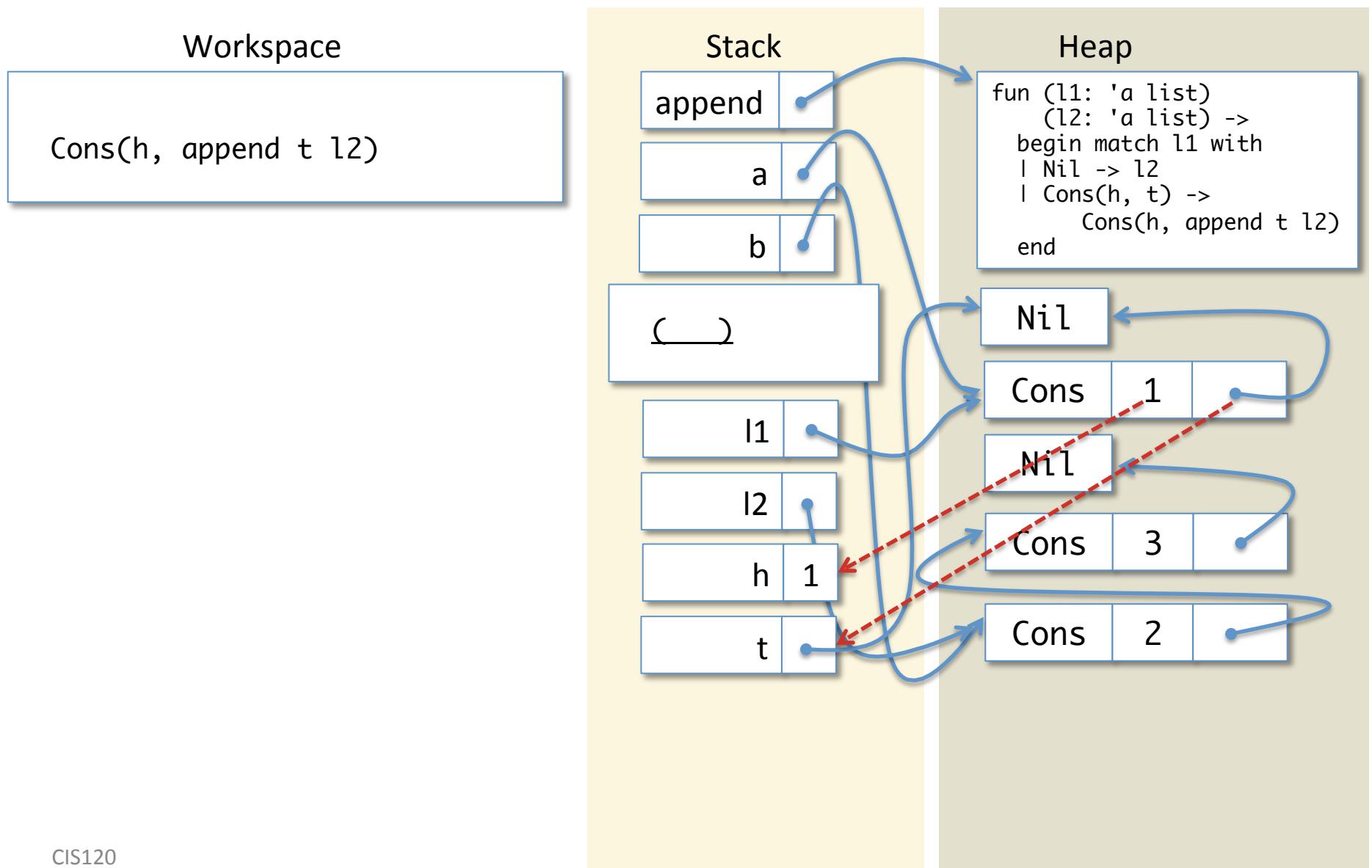
Nil case Doesn't Match



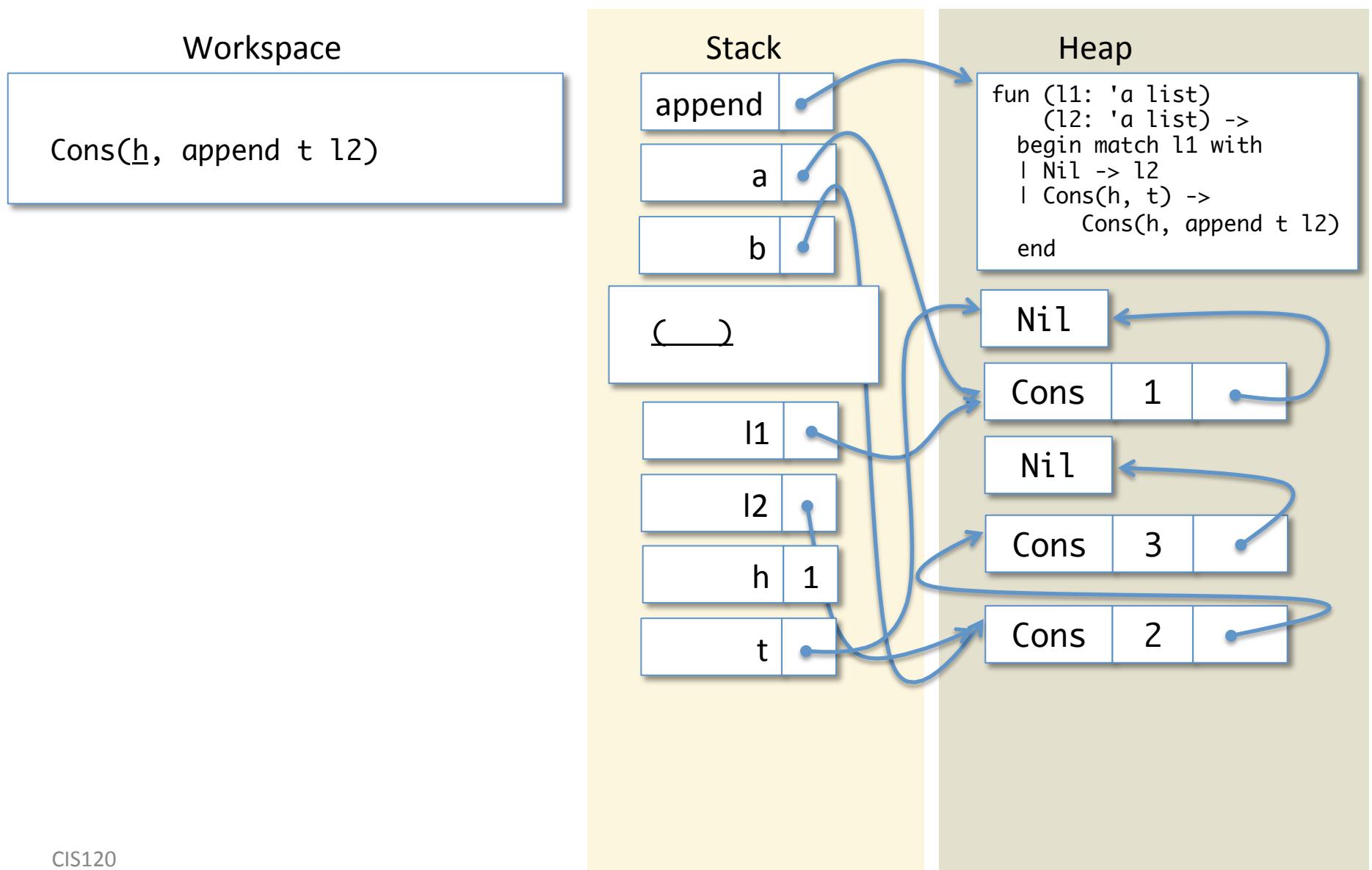
Cons case Does Match



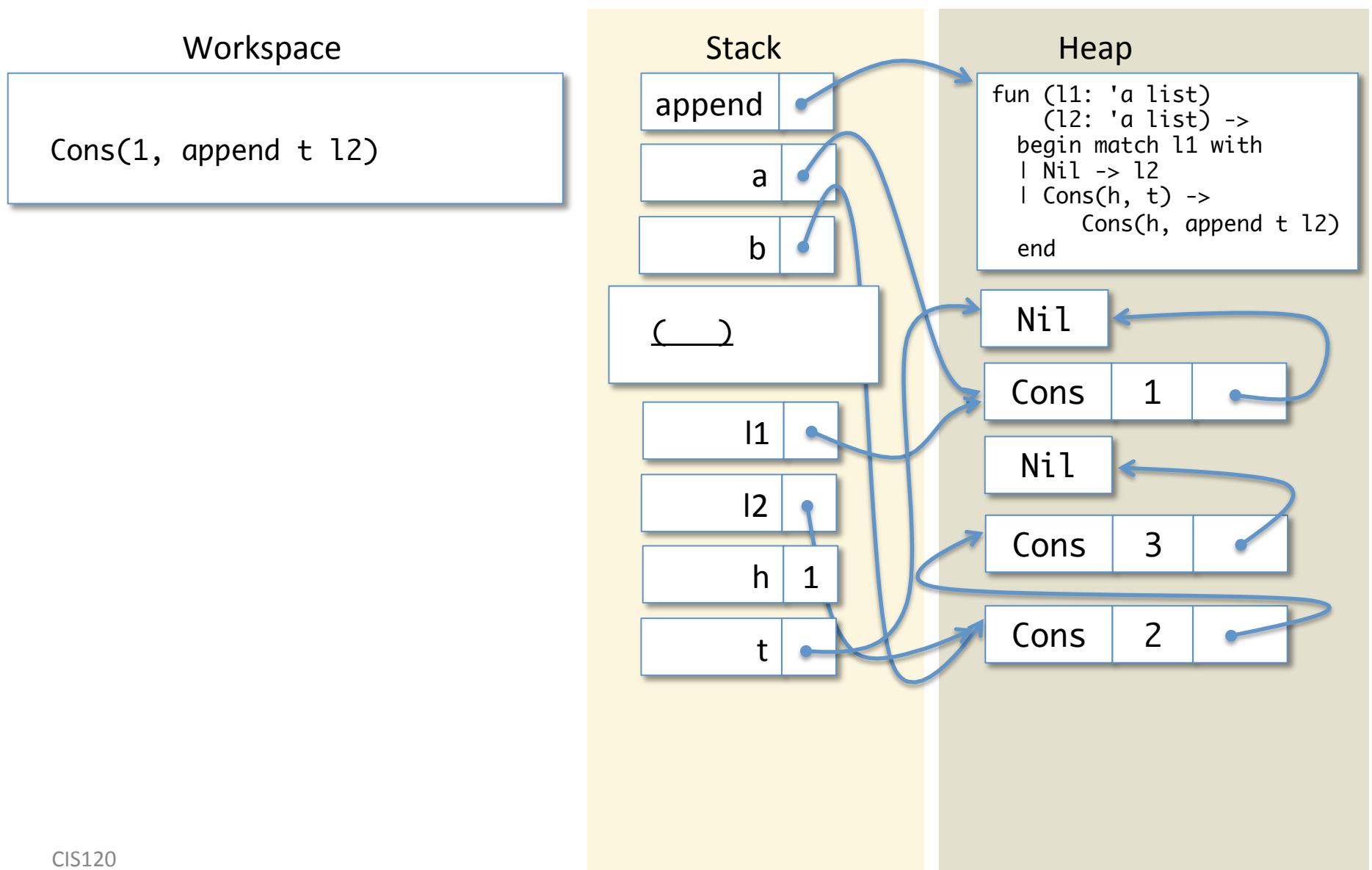
Simplify the Branch: push h, t



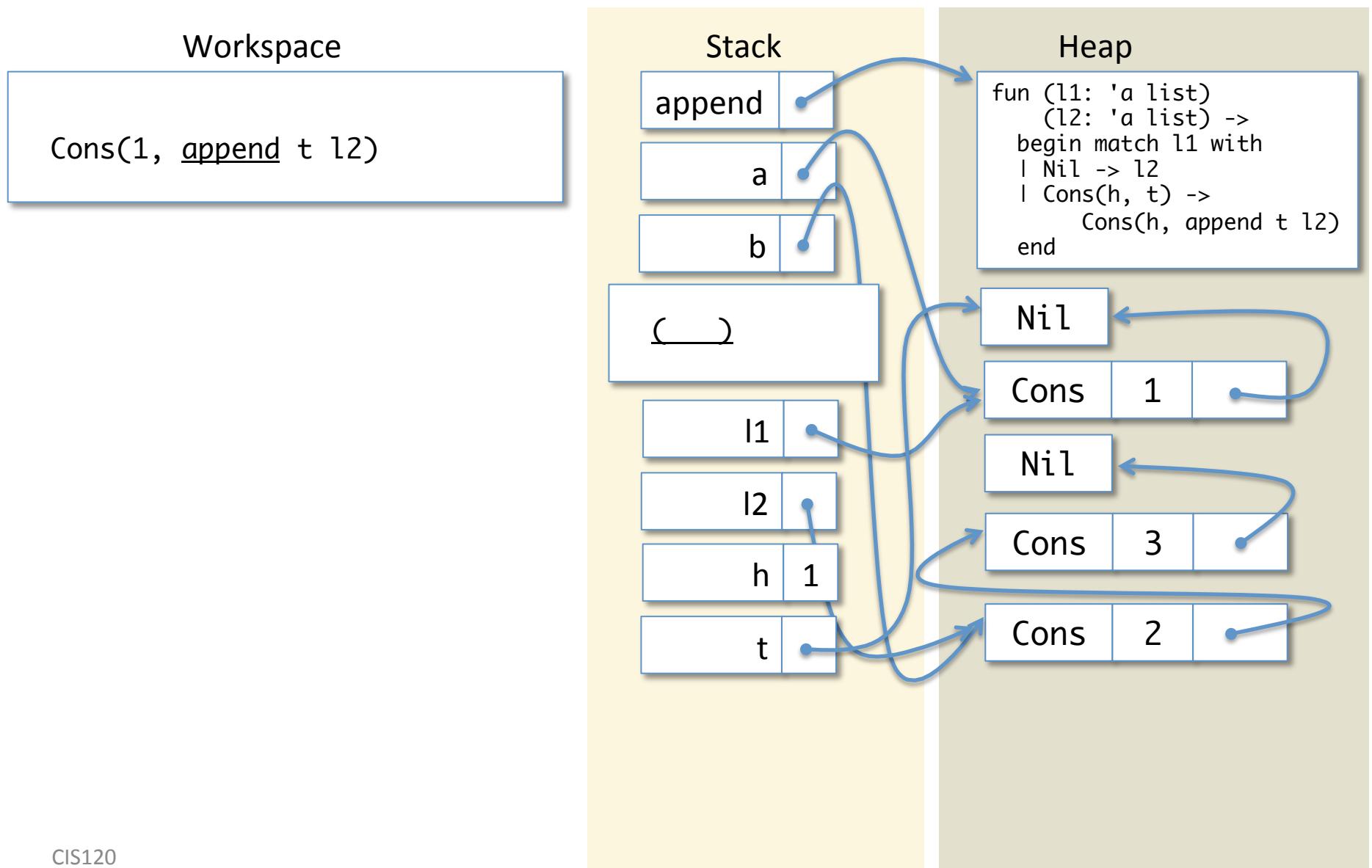
Lookup 'h'



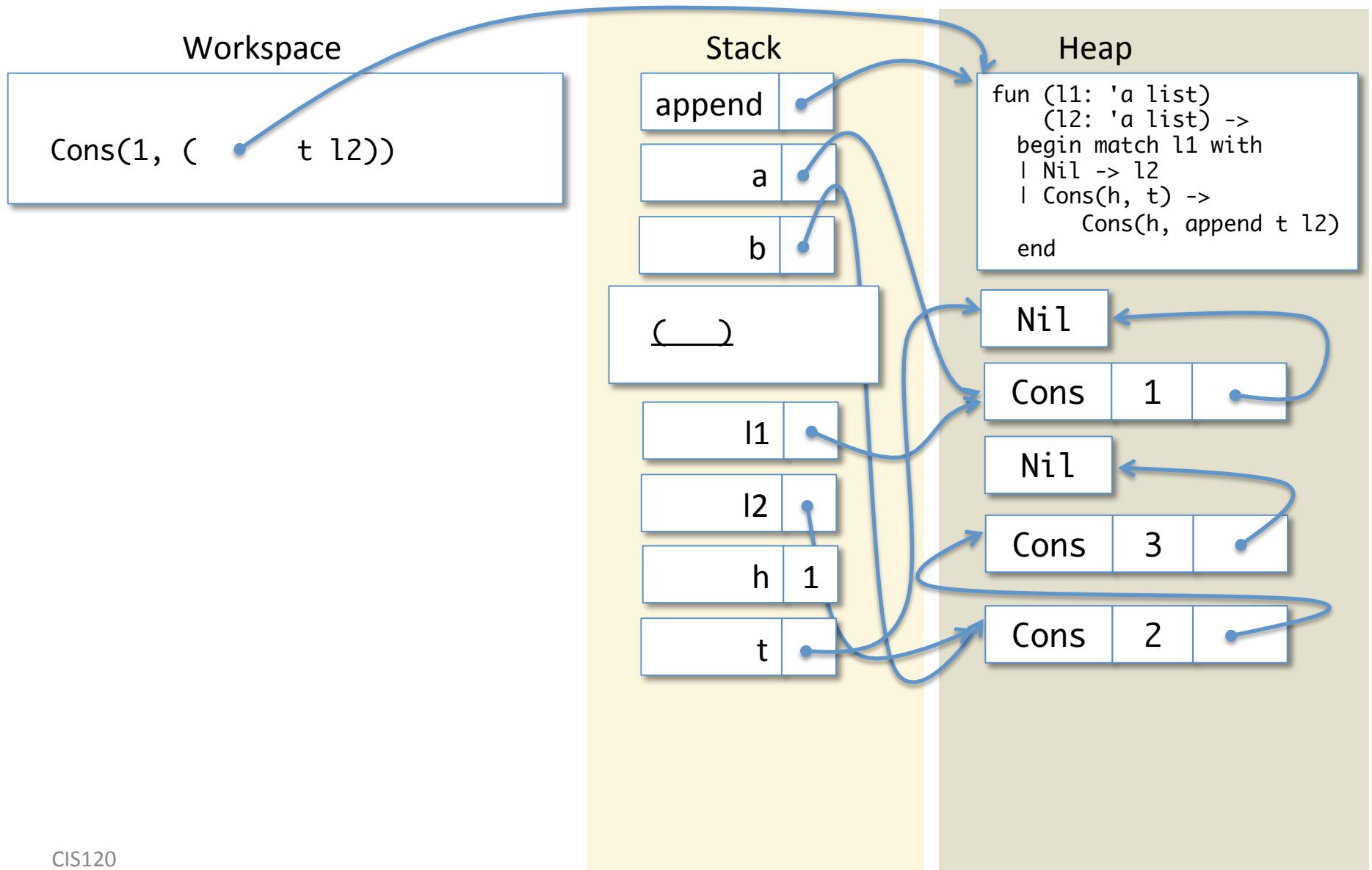
Lookup 'h'



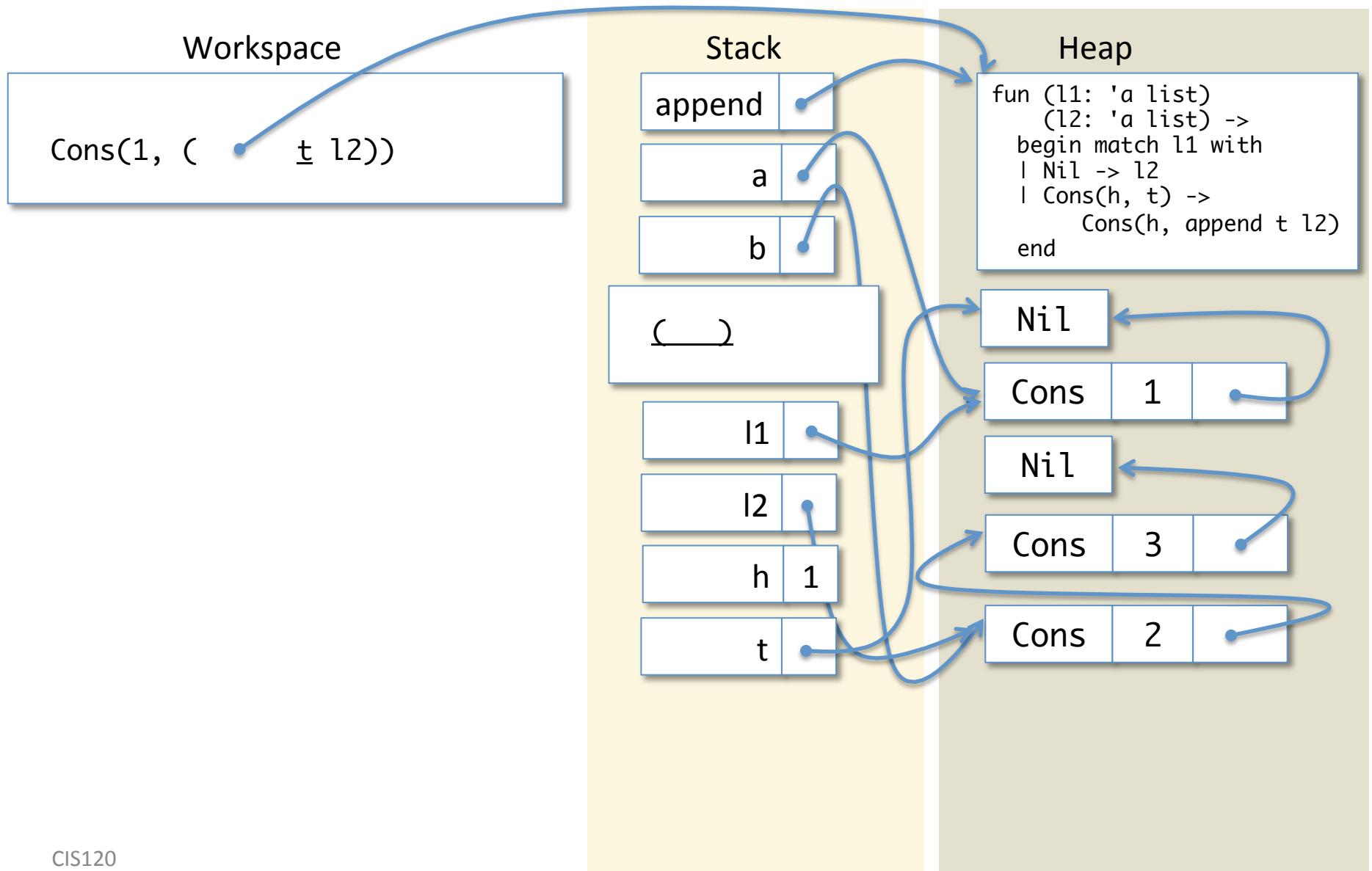
Lookup 'append'



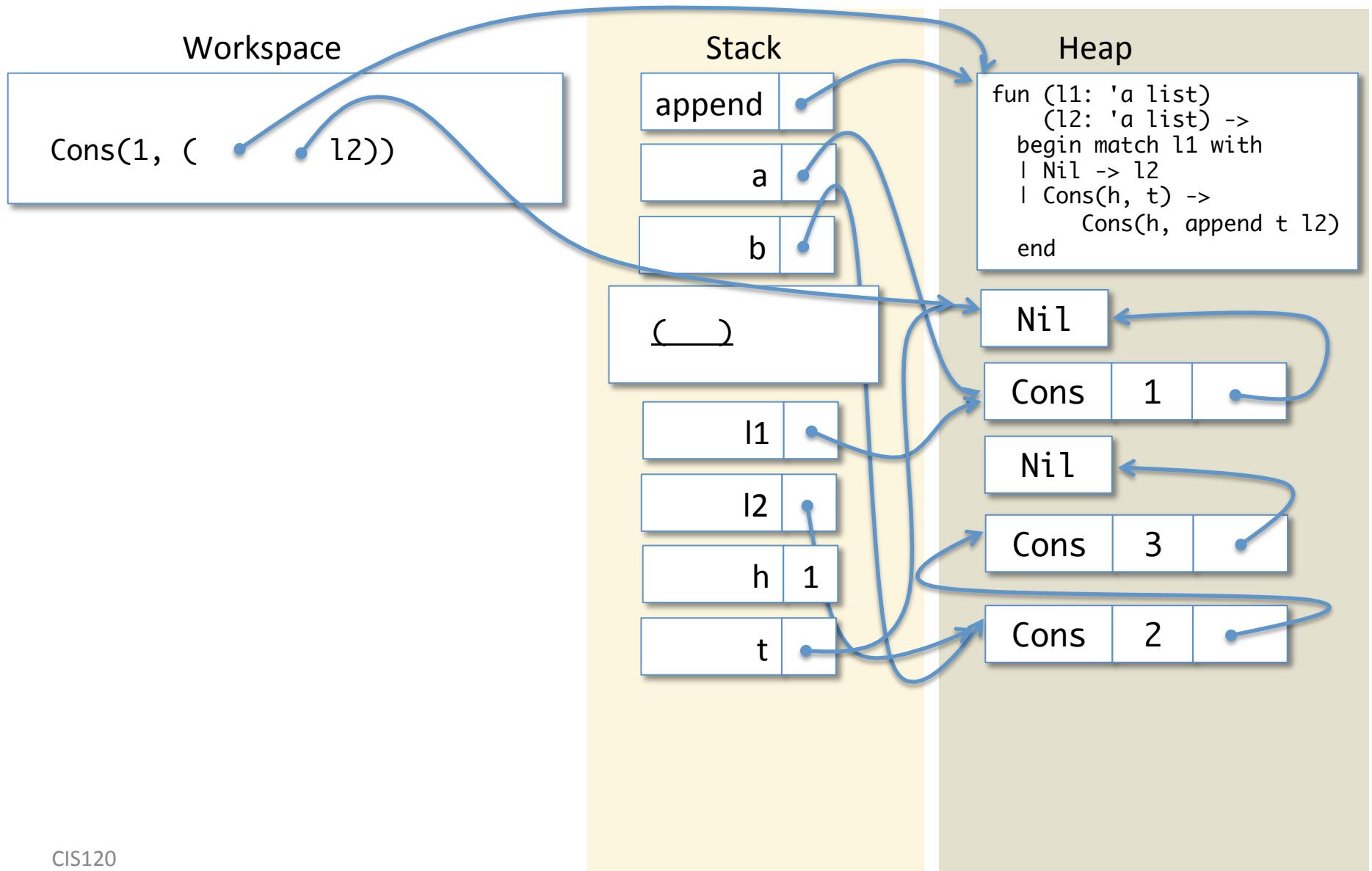
Lookup 'append'



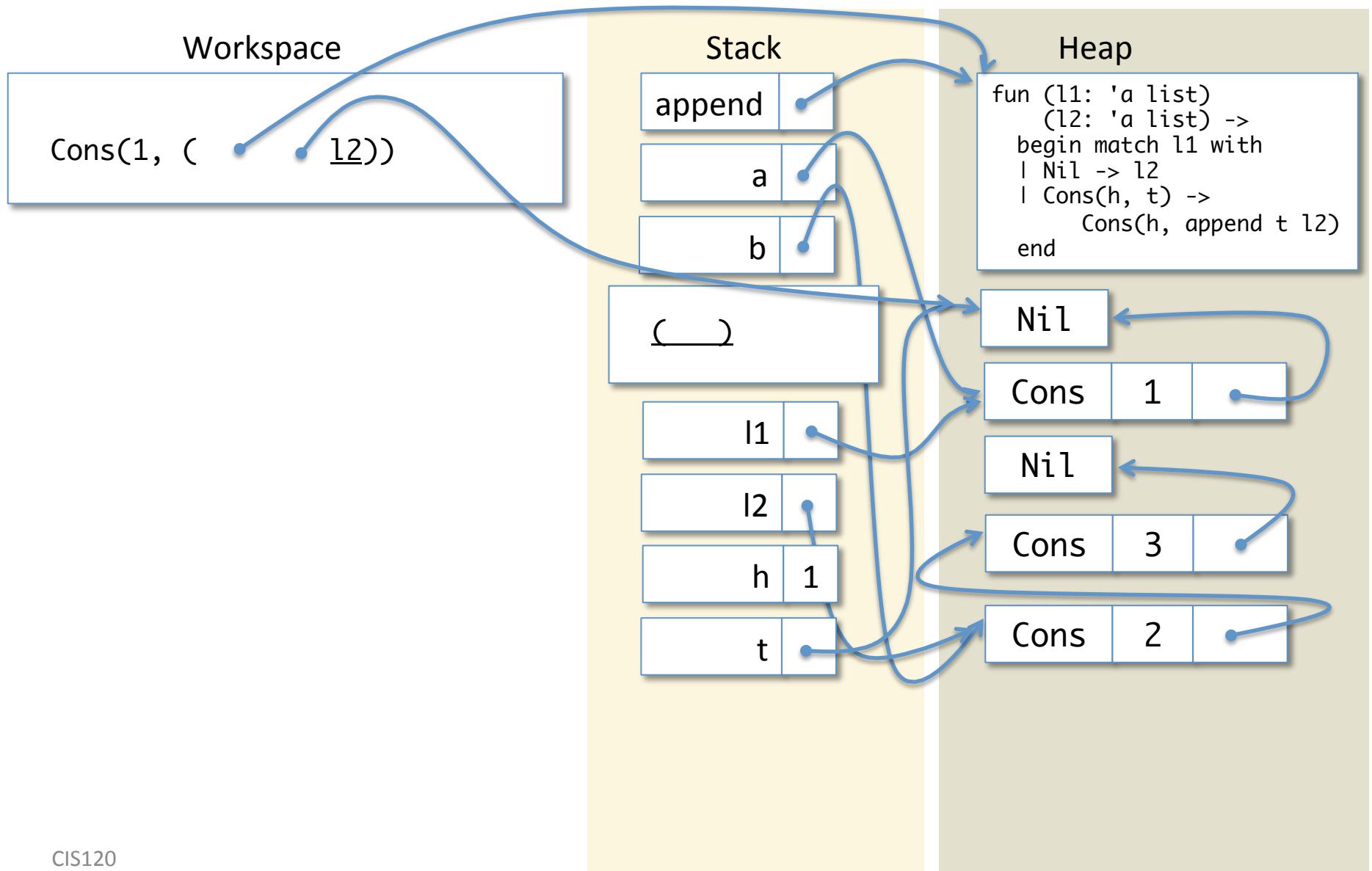
Lookup 't'



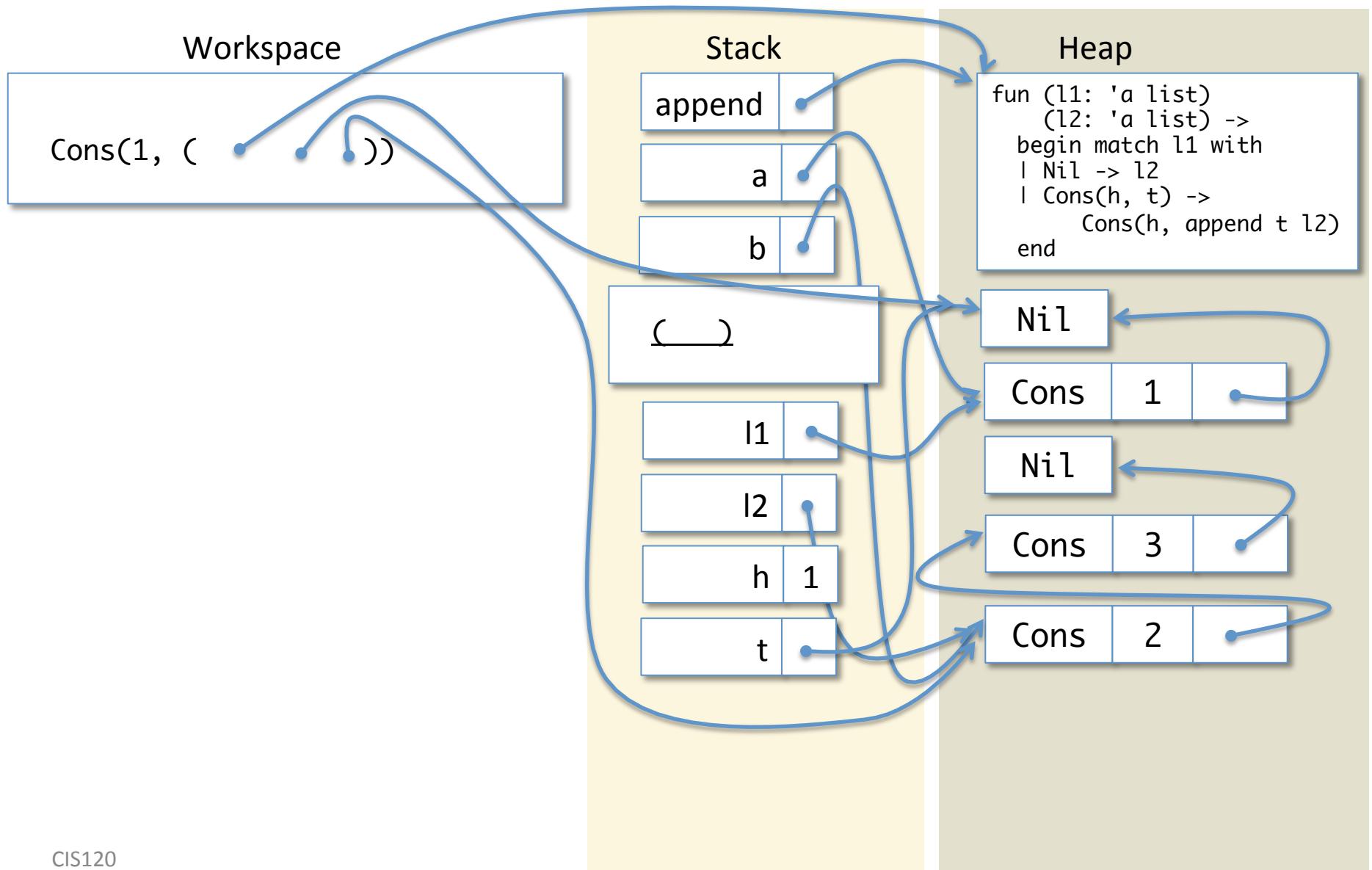
Lookup 't'



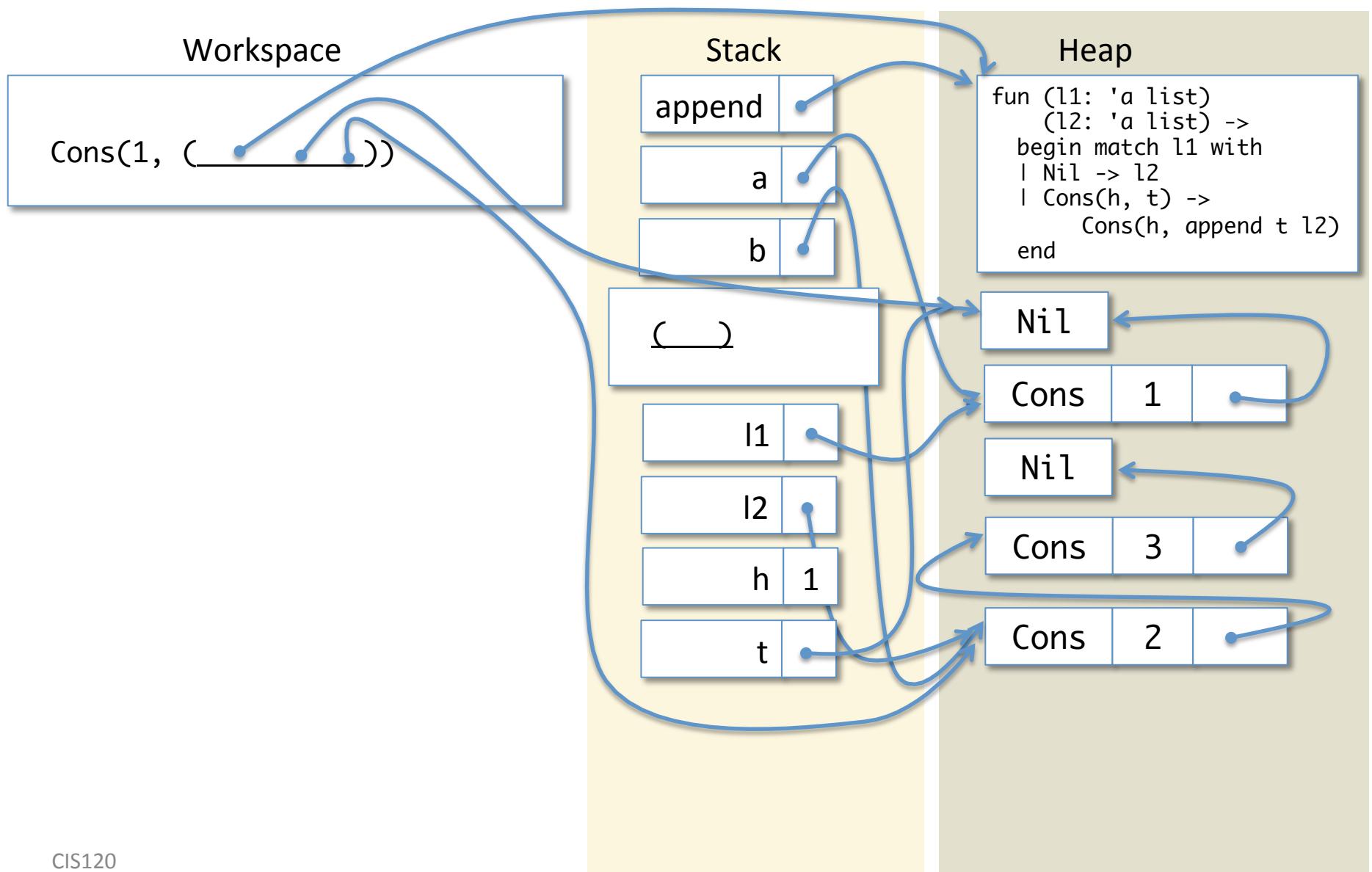
Lookup 'l2'



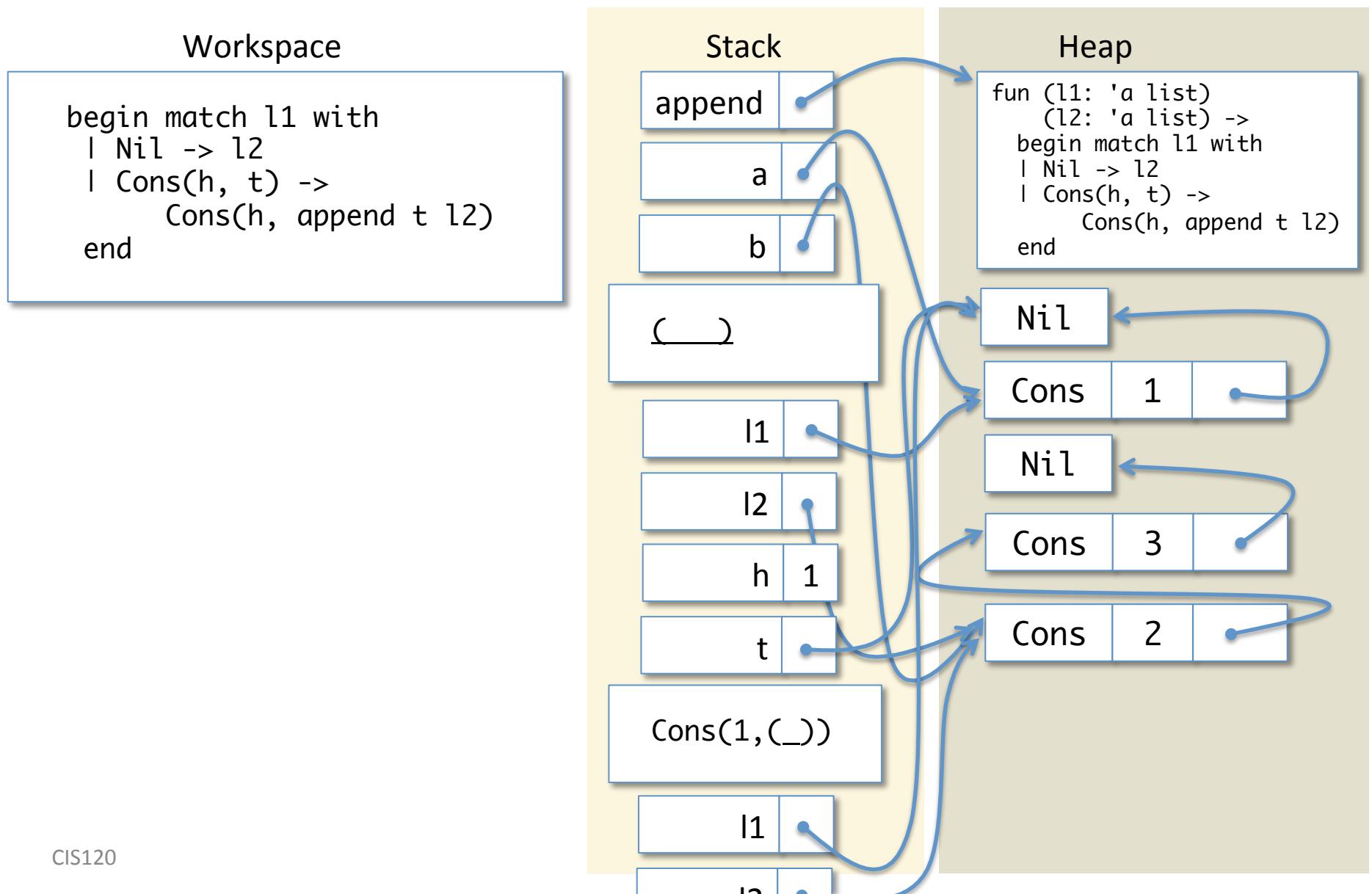
Lookup 'l2'



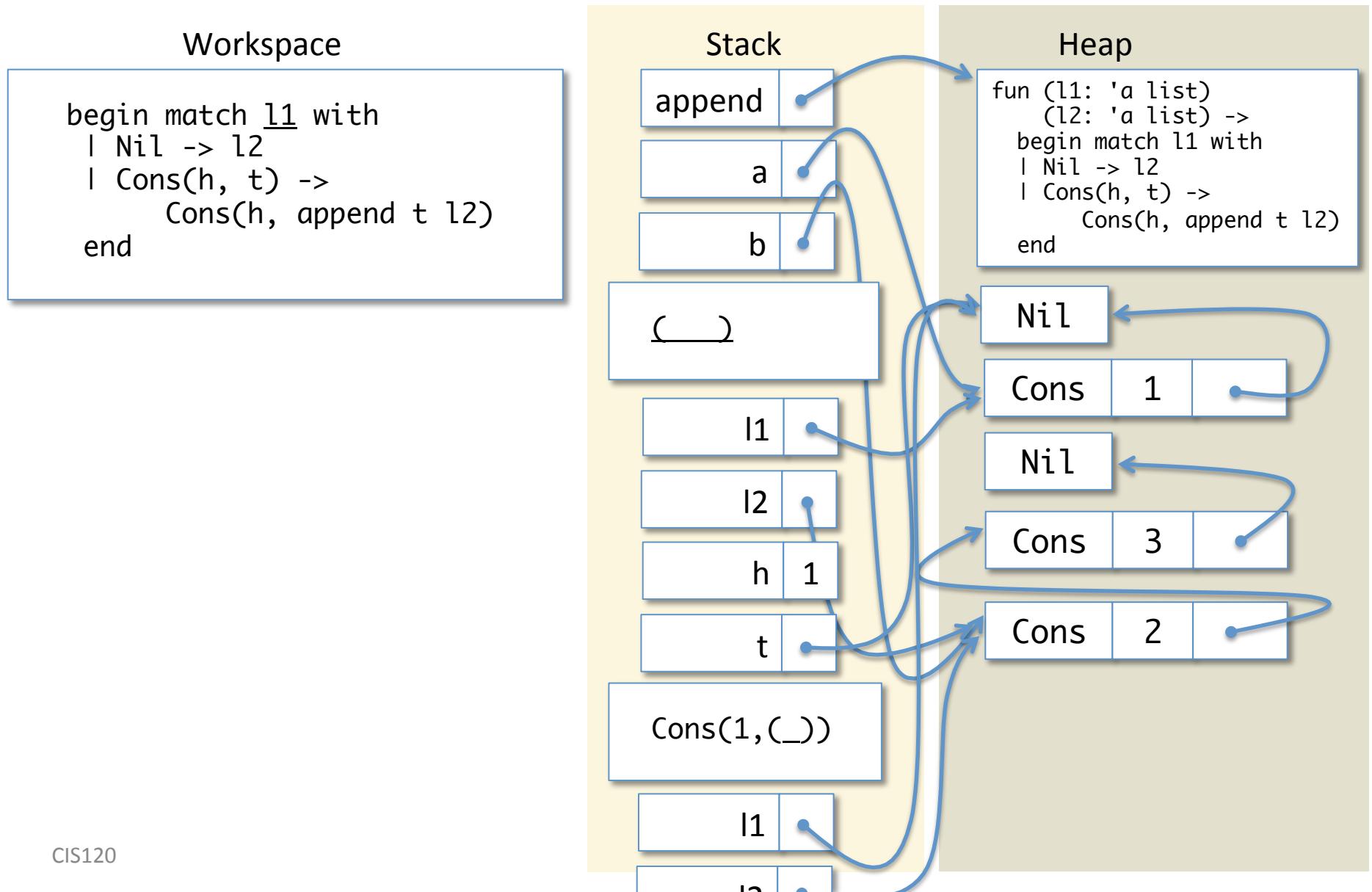
Do the Function Call



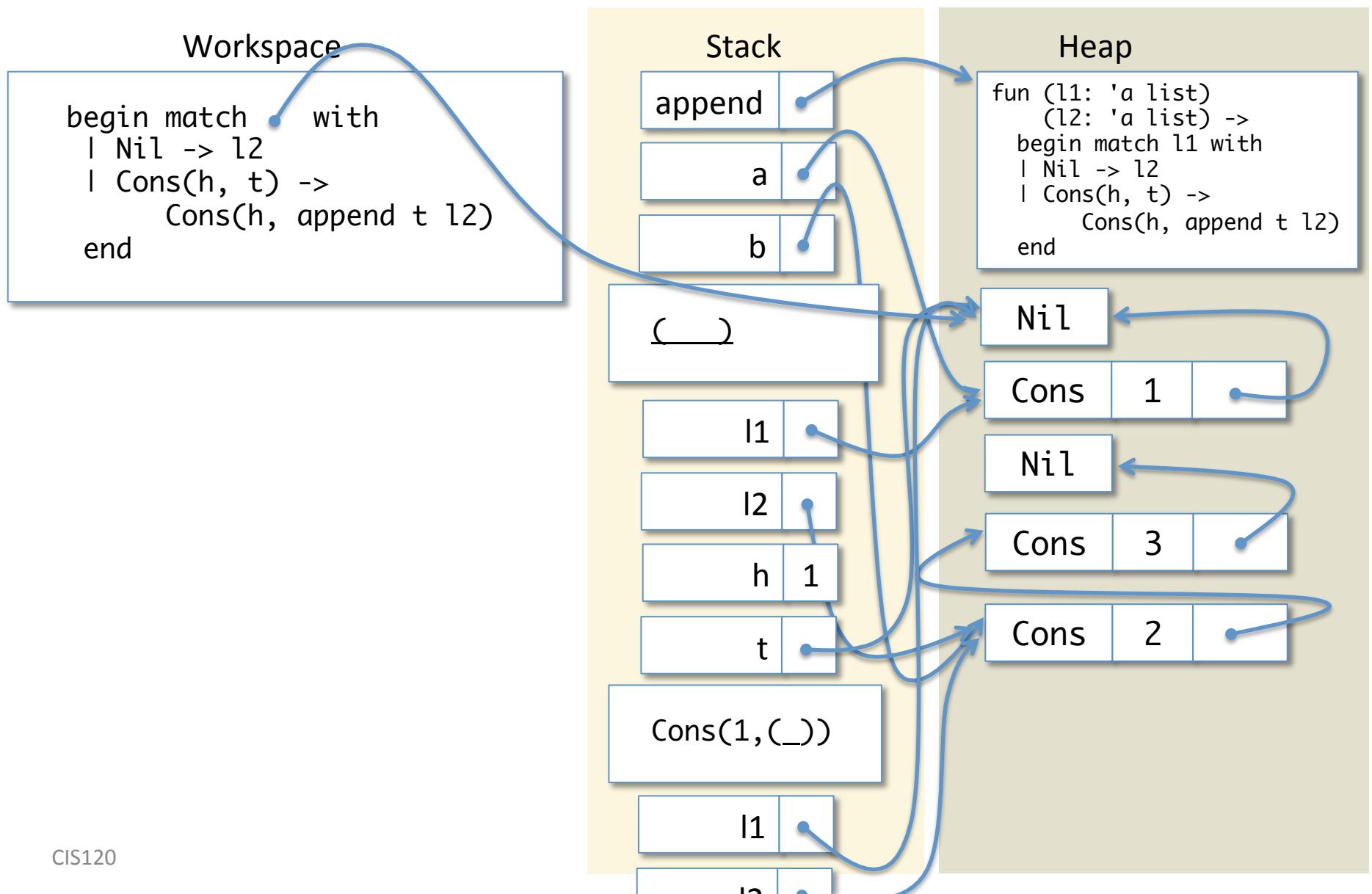
Save the Workspace; push l1, l2



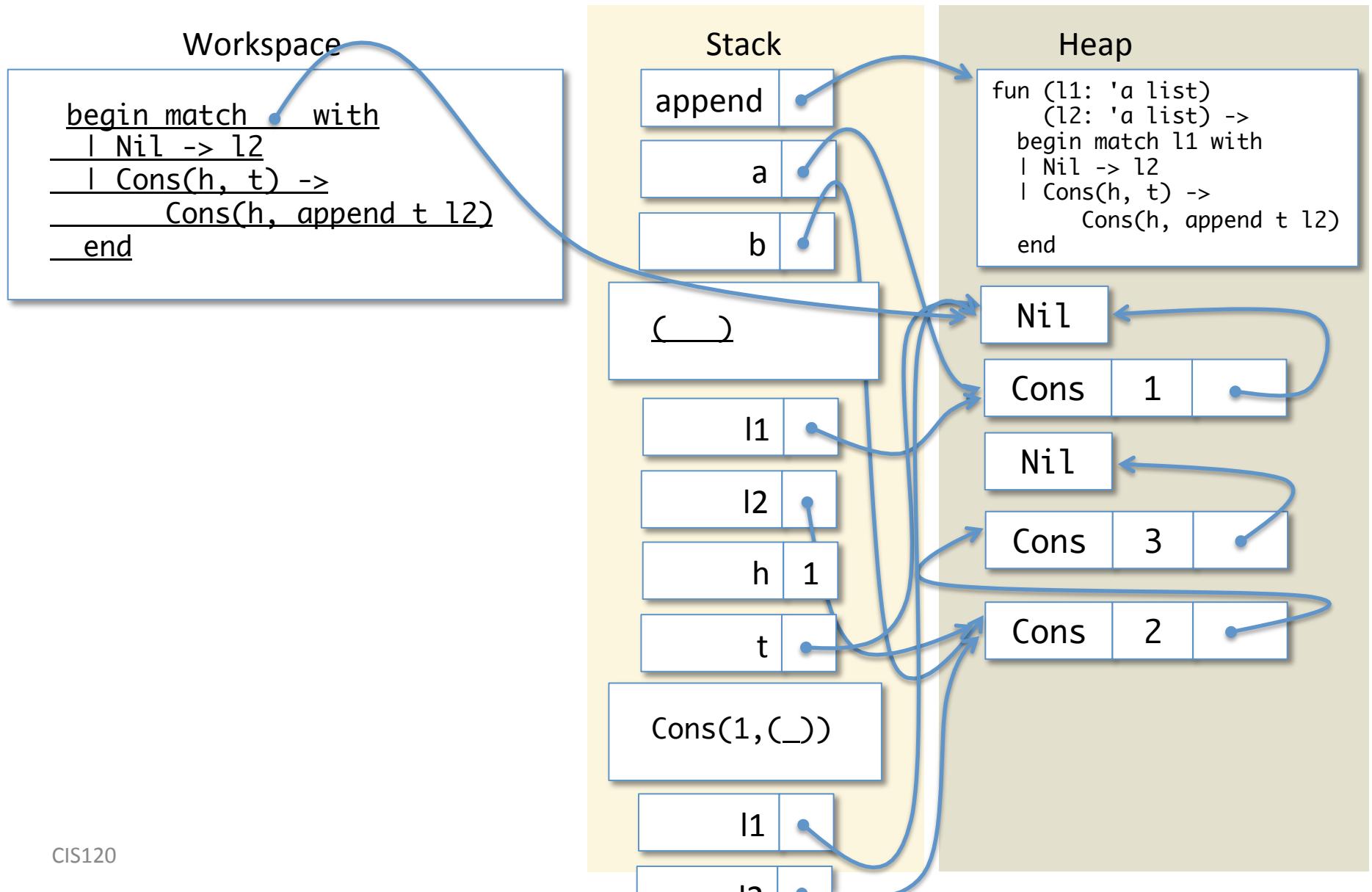
Lookup 'l1'



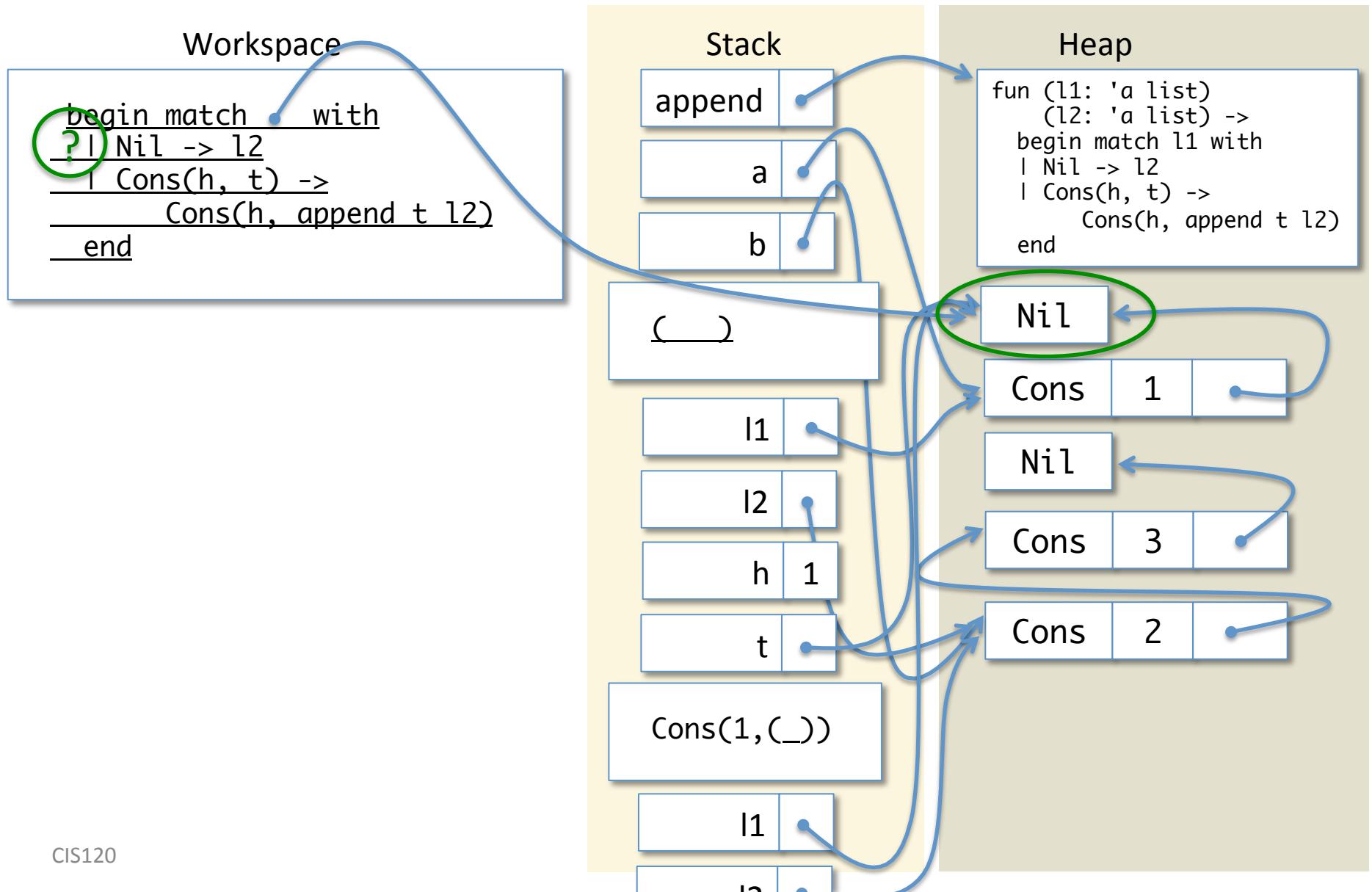
Lookup 'l1'



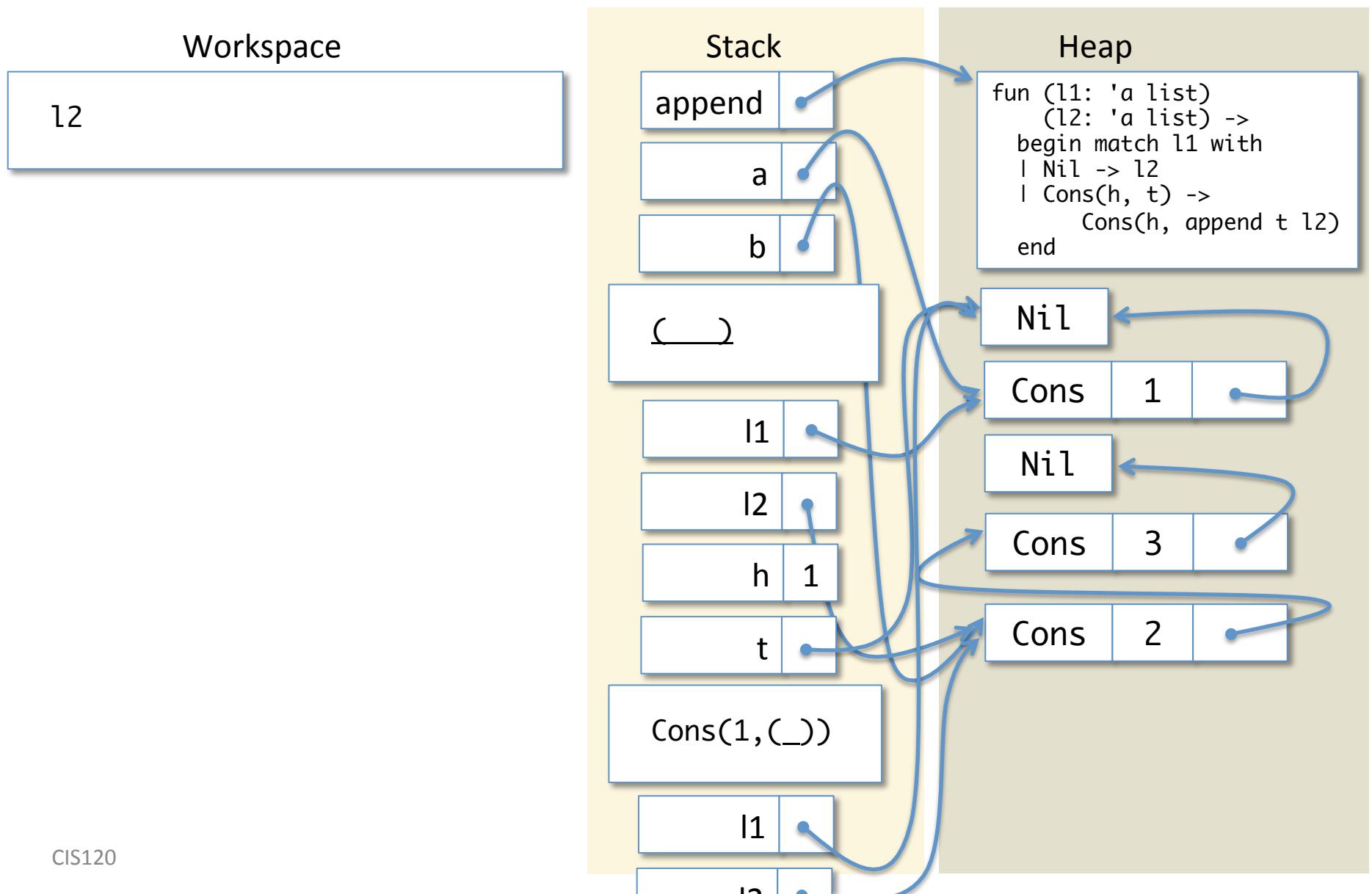
Match Expression



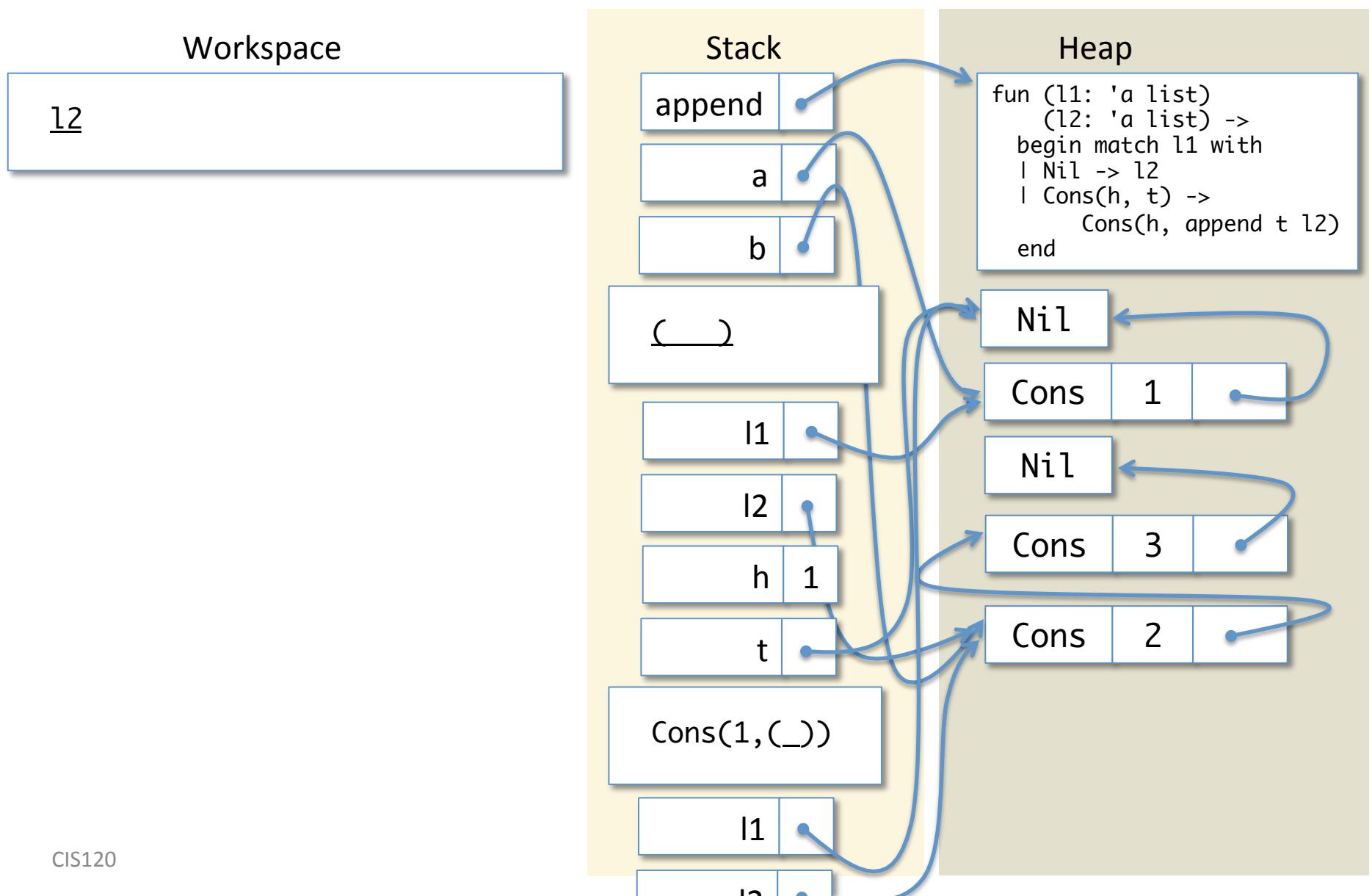
The Nil case Matches



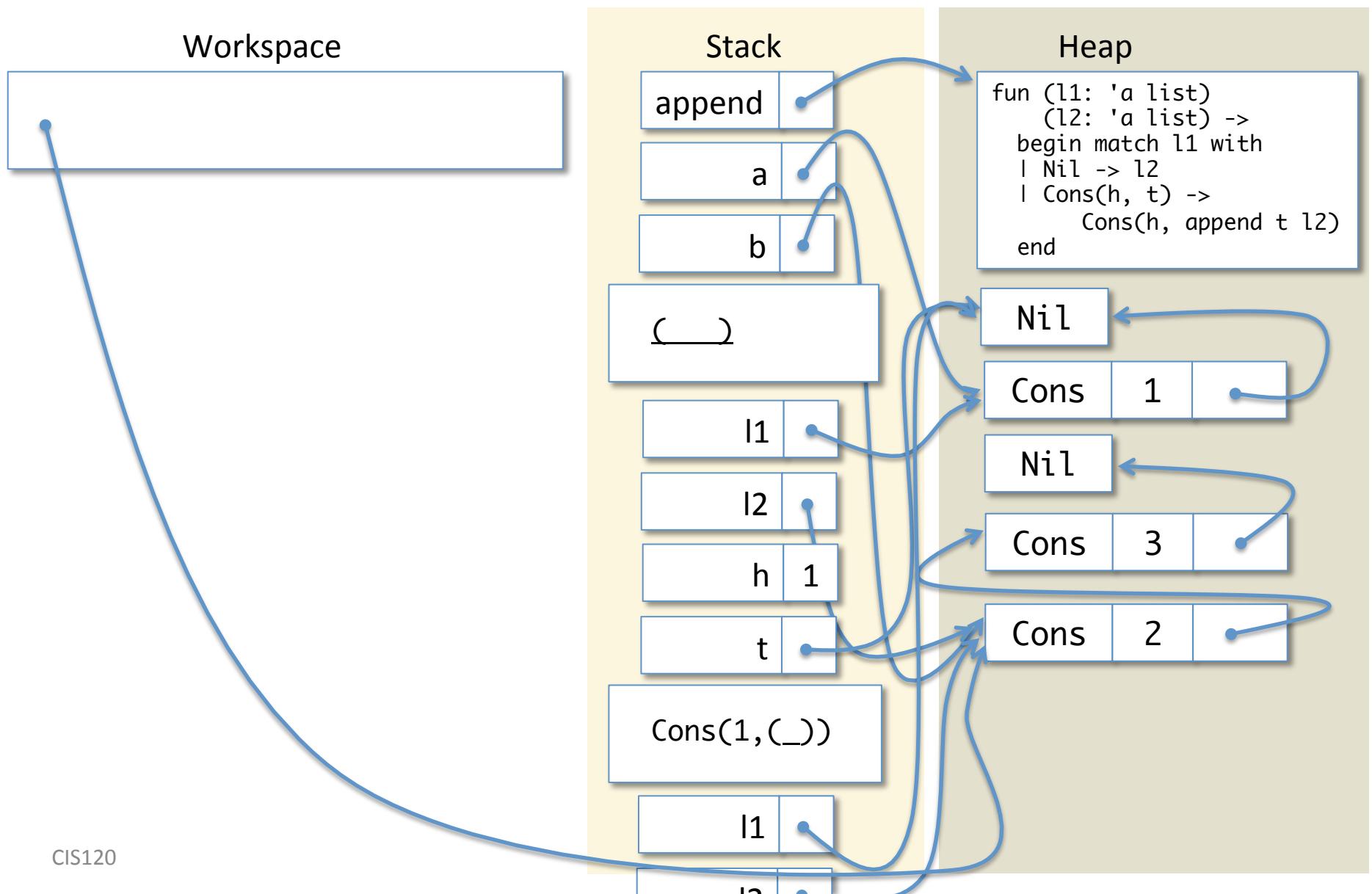
Simplify the Branch (nothing to push)



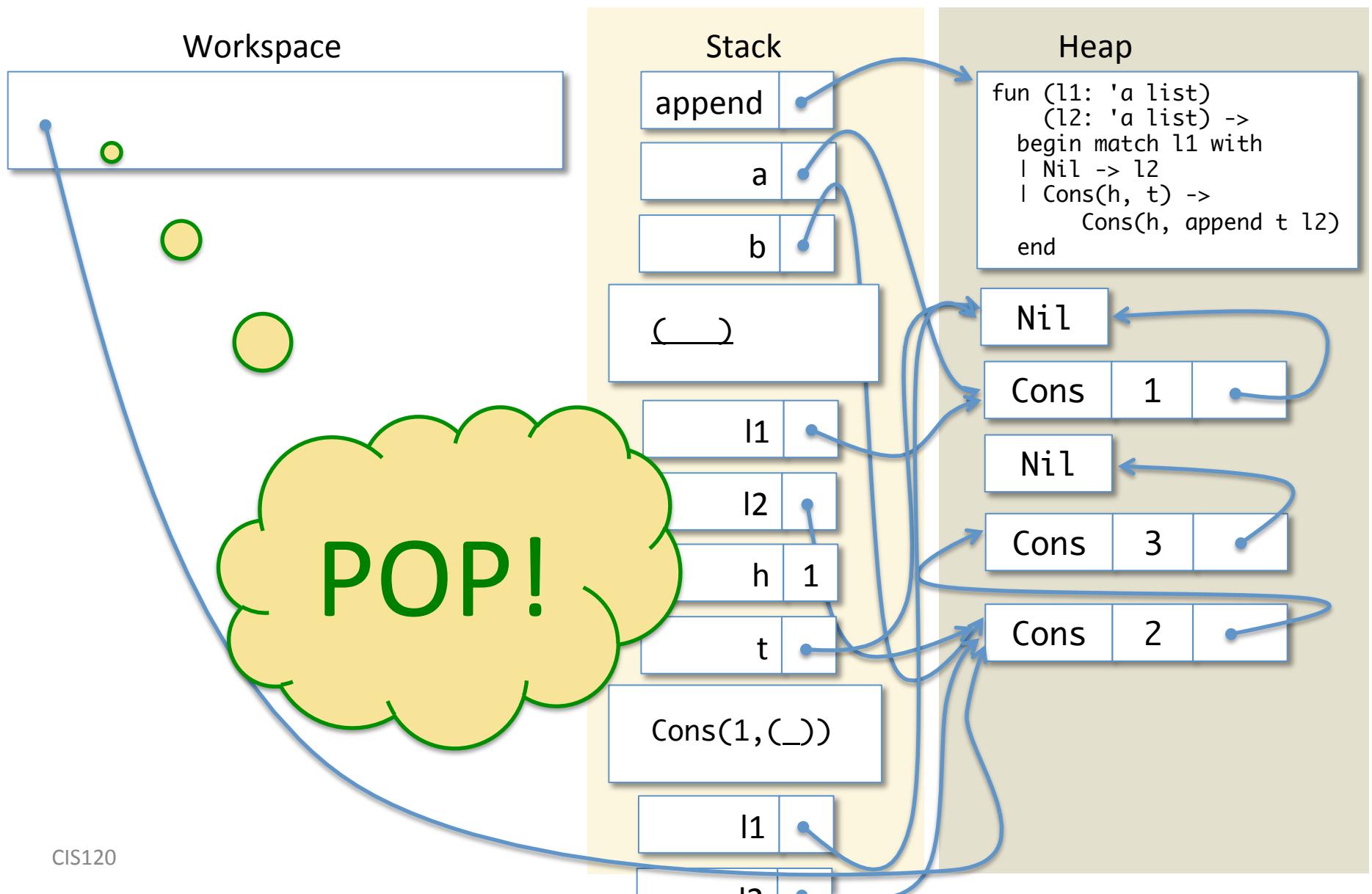
Lookup 'l2'



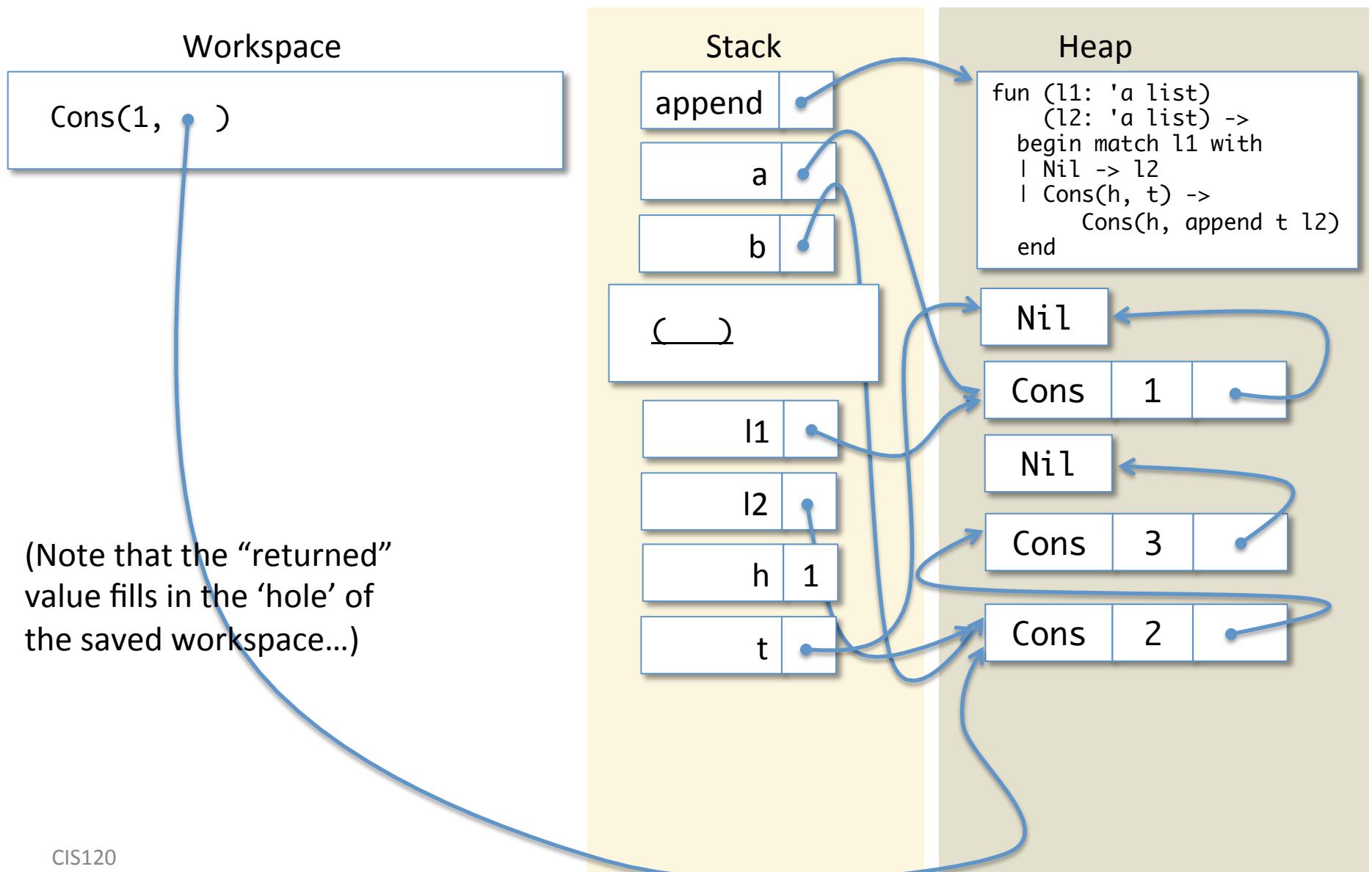
Lookup 'l2'



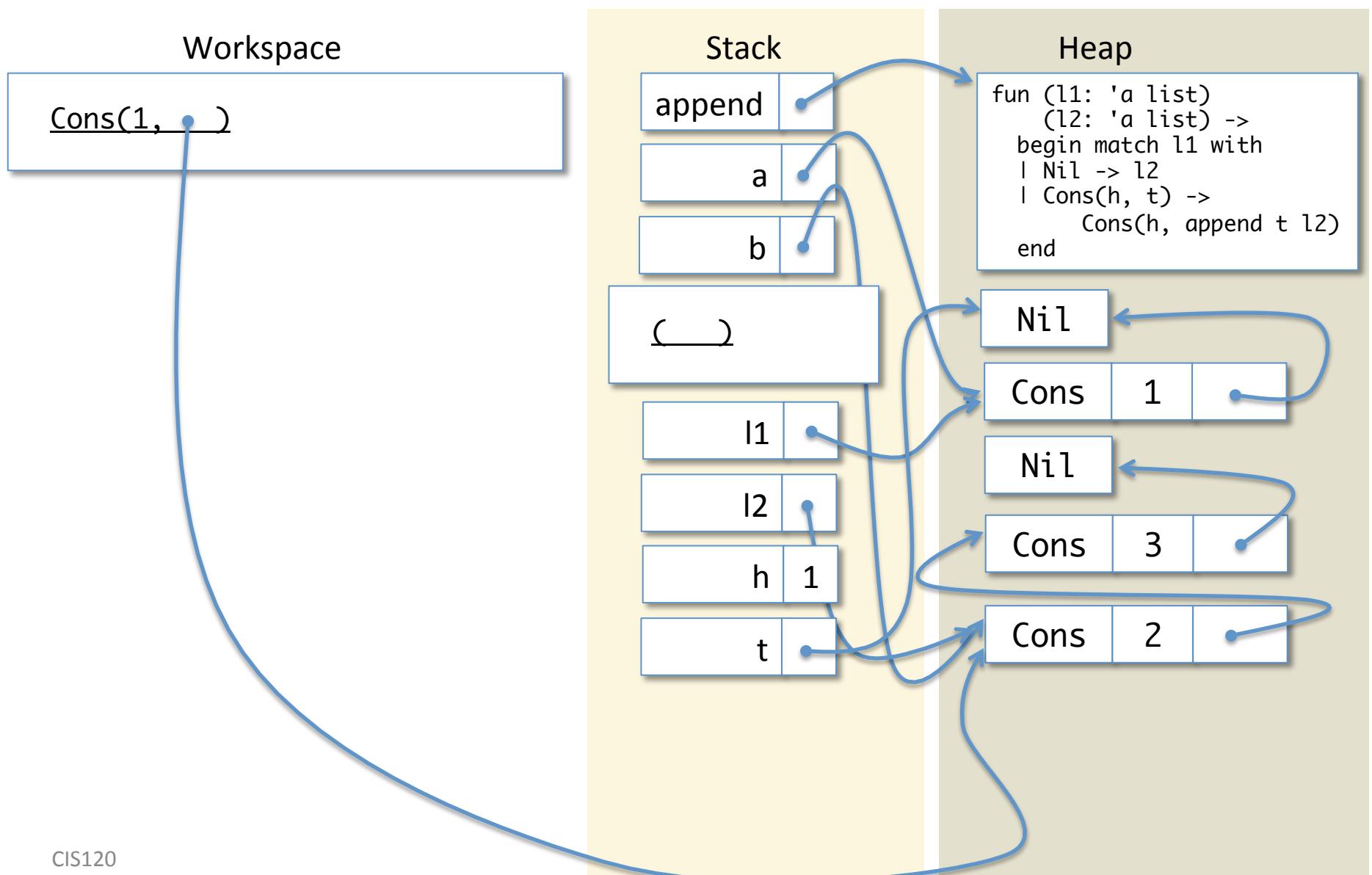
Done! Pop stack to last Workspace



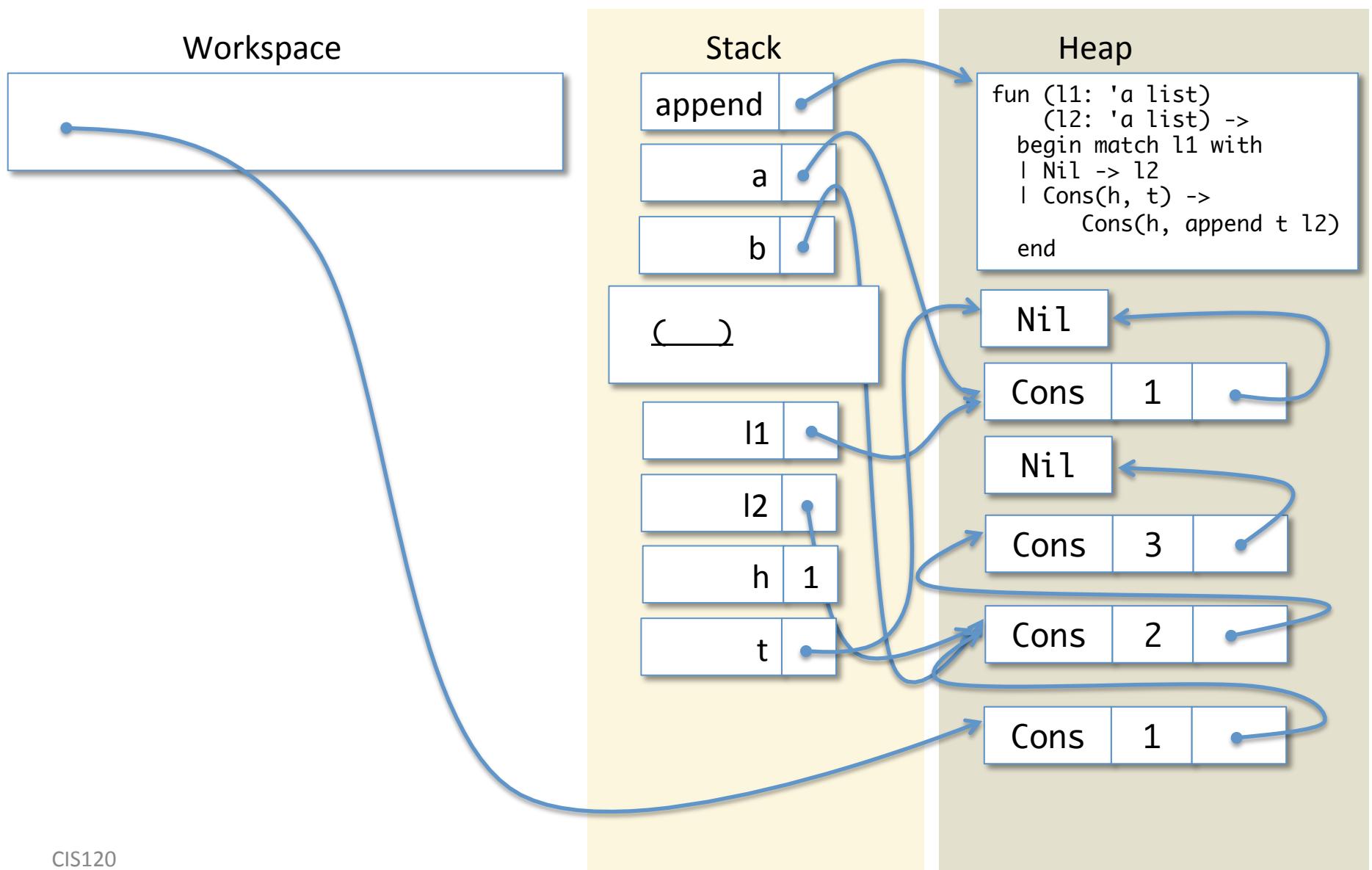
Done! Pop stack to last Workspace



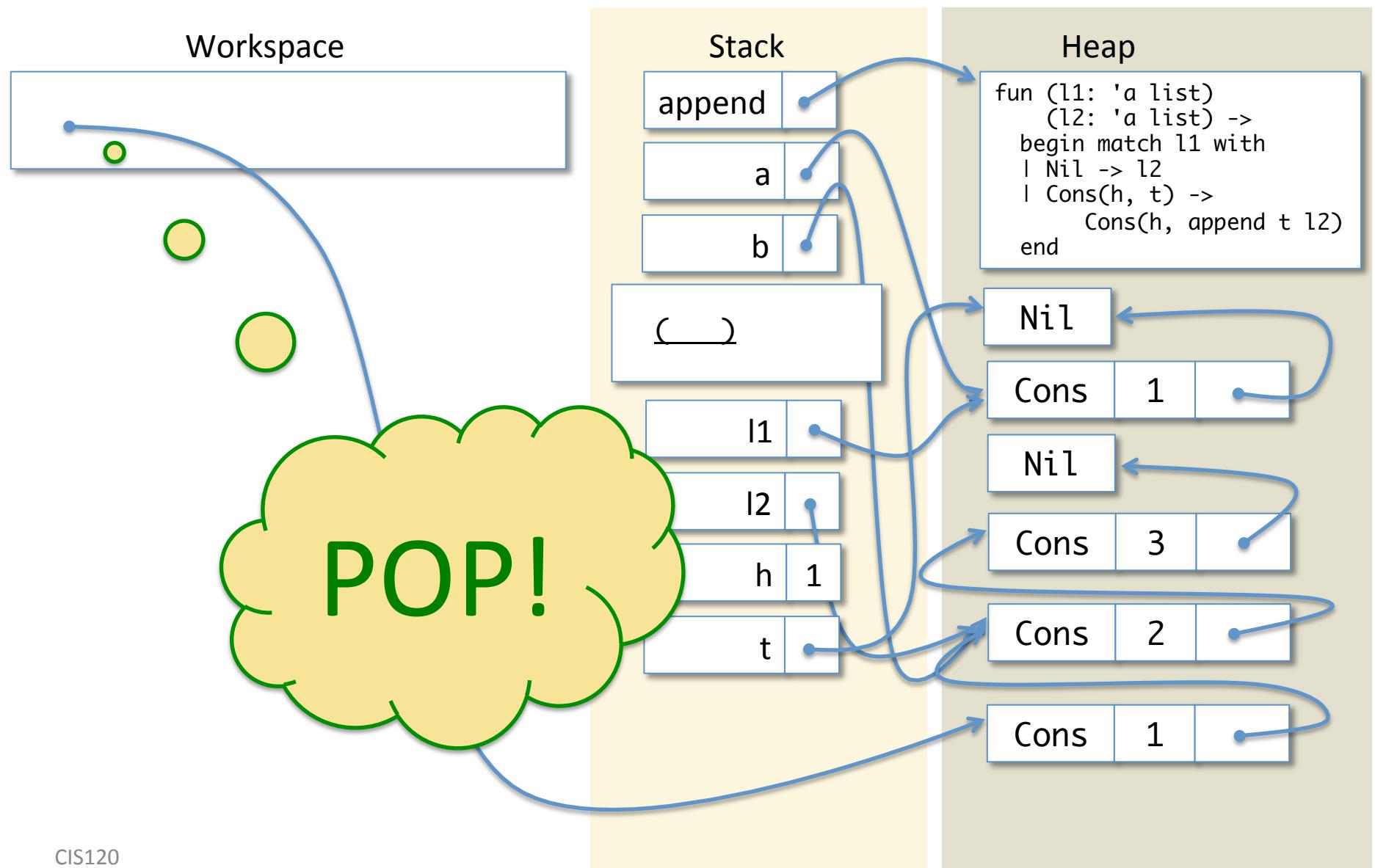
Allocate a Cons cell



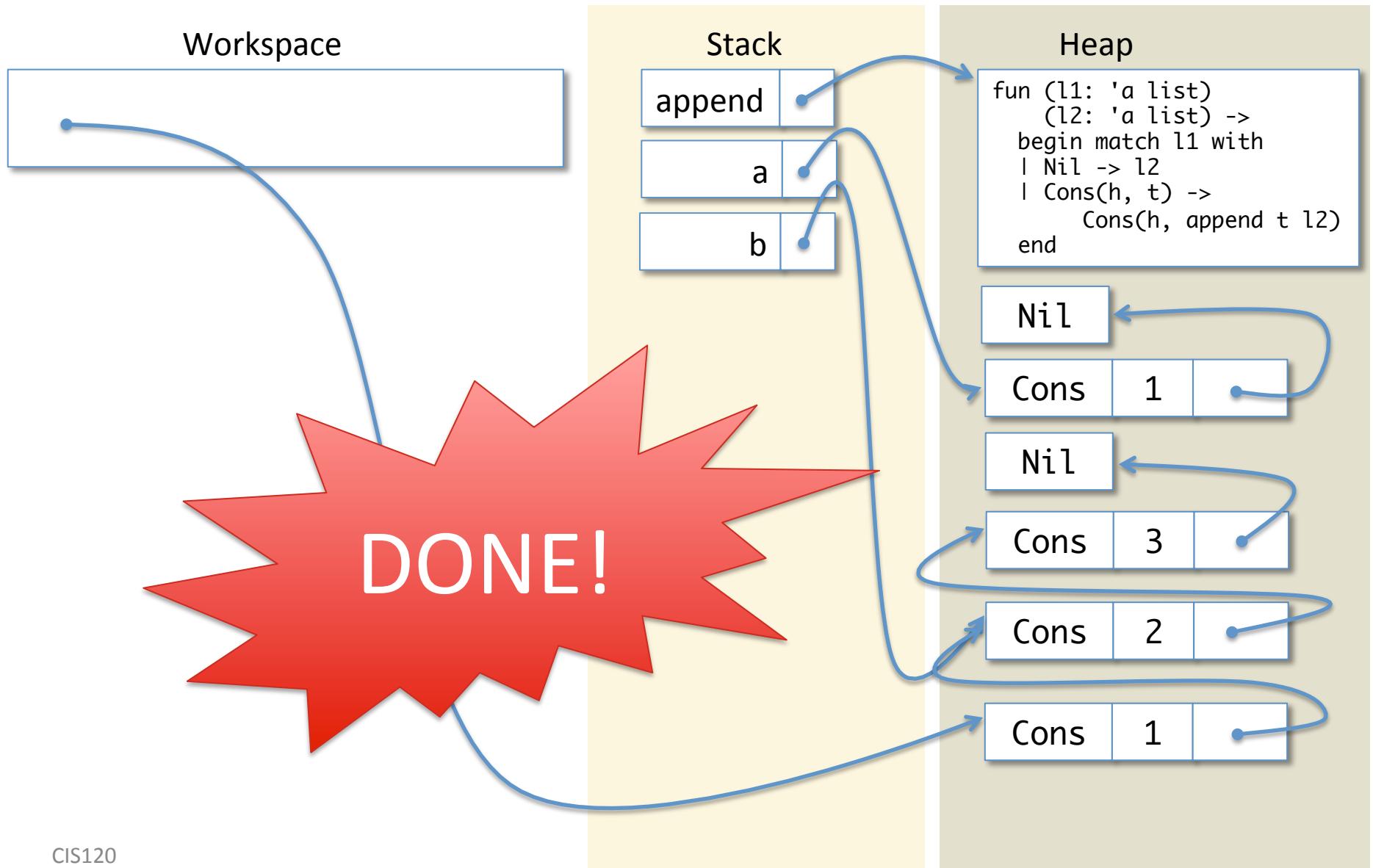
Allocate a Cons cell



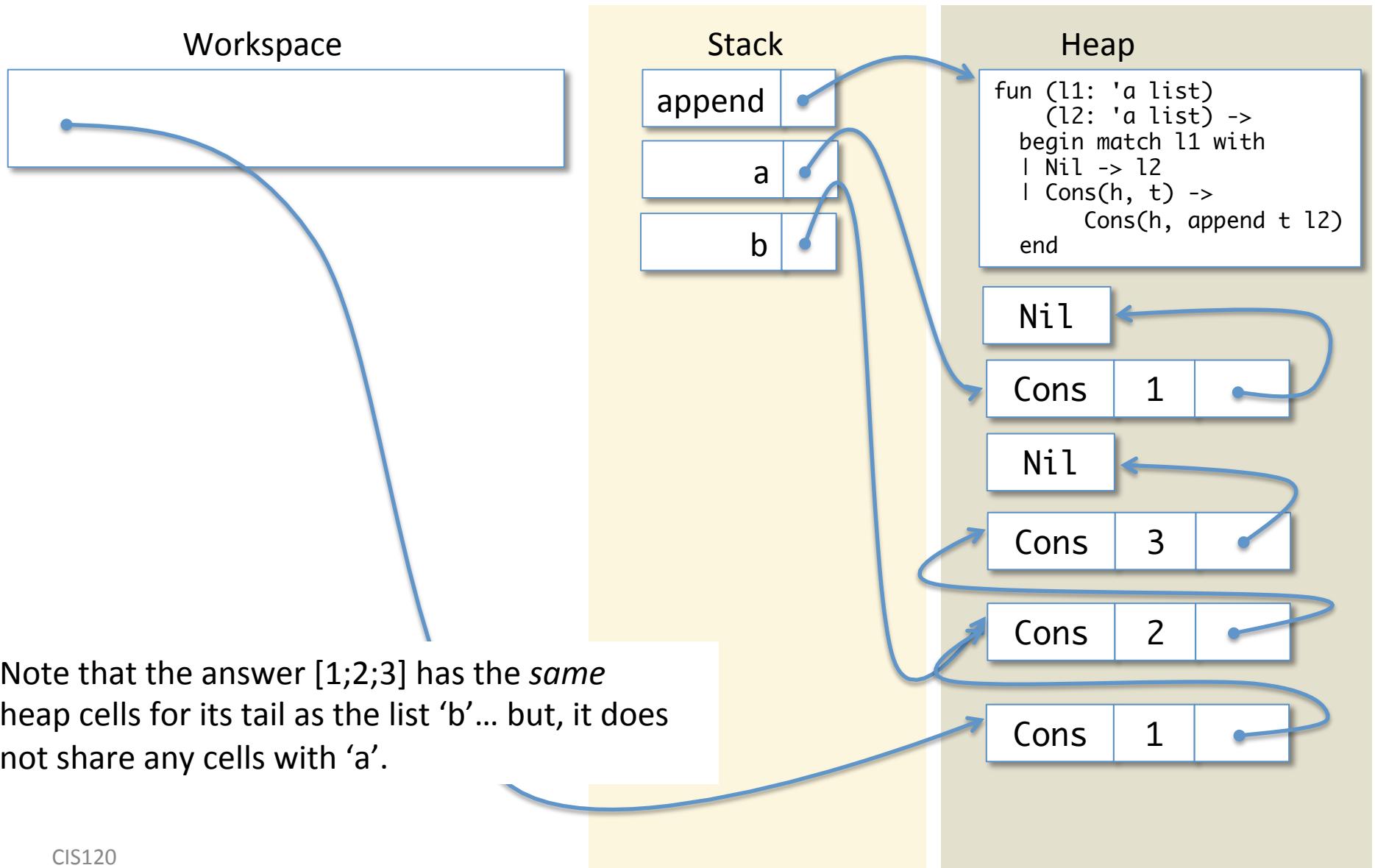
Done! Pop stack to last Workspace



Done! (PHEW!)



Done! (PHEW!)



Simplifying Match

- A match expression

```
begin match e with
  | pat1 -> branch1
  | ...
  | patn -> branchn
end
```

is ready if e is a value

- Note that e will always be a pointer to a constructor cell in the heap
- This expression is simplified by finding the first pattern pat_i that matches the cell and adding new bindings for the pattern variables (to the parts of e that line up) to the end of the stack
- replacing the whole match expression in the workspace with the corresponding branch_i

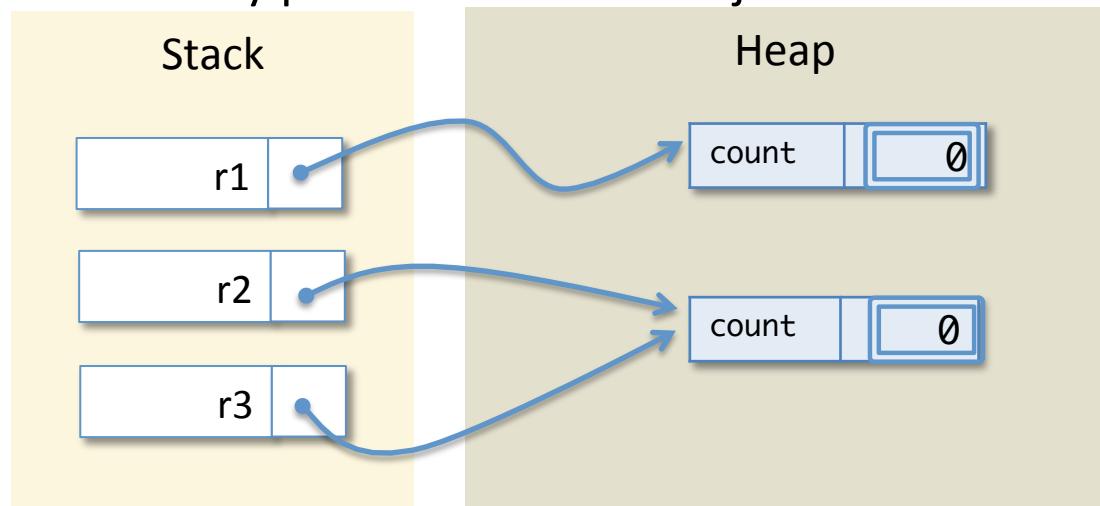
Reference and Equality

= vs. ==

Reference Equality

- Suppose we have two counters. How do we know whether they share the same internal state?
 - type counter = { mutable count : int }
 - We could increment one and see whether the other's value changes.
 - But we could also just test whether the references alias directly.
- Ocaml uses ‘`==`’ to mean *reference equality*:
 - two reference values are ‘`==`’ if they point to the same object in the heap; so:

```
r2 == r3  
not (r1 == r2)  
r1 = r2
```



Structural vs. Reference Equality

- *Structural (in)equality*: $v1 = v2$ $v1 \neq v2$
 - recursively traverses over the *structure* of the data, comparing the two values' components for structural equality
 - function values are never structurally equivalent to anything
 - structural equality can go into an infinite loop (on cyclic structures)
 - appropriate for comparing *immutable* datatypes
- *Reference (in)equality*: $v1 == v2$ $v1 != v2$
 - Only looks at where the two references point in the heap
 - function values are only equal to themselves
 - equates strictly fewer things than structural equality
 - appropriate for comparing *mutable* datatypes

What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0; } in  
let p2 : point = p1 in  
  
p1 = p2
```

1. true
2. false
3. runtime error
4. compile-time error

Answer: true

What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0; } in  
let p2 : point = p1 in  
  
p1 == p2
```

1. true
2. false
3. runtime error
4. compile-time error

Answer: true

What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0; } in  
let p2 : point = { x = 0; y = 0; } in  
  
p1 == p2
```

1. true
2. false
3. runtime error
4. compile-time error

Answer: false

What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0; } in
let p2 : point = { x = 0; y = 0; } in
let l1 : point list = [p1] in
let l2 : point list = [p2] in

l1 = l2
```

1. true
2. false
3. runtime error
4. compile-time error

Answer: true

What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0; } in
let p2 : point = p1 in
let l1 : point list = [p1] in
let l2 : point list = [p2] in

l1 == l2
```

1. true
2. false
3. runtime error
4. compile-time error

Answer: false

Putting State to Work

Mutable Queues

A design problem

Suppose you are implementing a website to sell tickets to a very popular music event. To be fair, you would like to allow people to select seats first come, first served. How would you do it?

- Understand the problem
 - Some people may visit the website to buy tickets while others are still selecting their seats
 - Need to remember the order in which people purchase tickets
- Define the interface
 - Need a data structure to store ticket purchasers
 - Need to add purchasers to the *end* of the line
 - Need to allow purchasers at the *beginning* of the line to select seats
 - Both kinds of access must be efficient to handle the volume

(Mutable) Queue Interface

```
module type QUEUE =
sig
  (* abstract type *)
  type 'a queue

  (* Make a new, empty queue *)
  val create : unit -> 'a queue

  (* Determine if the queue is empty *)
  val is_empty : 'a queue -> bool

  (* Add a value to the end of the queue *)
  val enq : 'a -> 'a queue -> unit

  (* Remove the first value (if any) and return it *)
  val deq : 'a queue -> 'a

end
```

We can tell, just looking at this interface, that it is for a MUTABLE data structure. How?

Because queues are mutable, we must allocate a new one every time we need one.

Adding an element to the queue returns unit because it *modifies* the given queue.

Specify the behavior via test cases

```
let test () : bool =
  let q : int queue = create () in
  enq 1 q;
  enq 2 q;
  1 = deq q
;; run_test "queue test 1" test

let test () : bool =
  let q : int queue = create () in
  enq 1 q;
  enq 2 q;
  let _ = deq q in
  2 = deq q
;; run_test "queue test 2" test
```

What value should replace ??? so that the following test passes?

```
let test () : bool =
  let q : int queue = create () in
  enq 1 q;
  let _ = deq q in
  enq 2 q;
  ??? = deq q

;; run_test "enq after deq" test
```

1. 1
2. 2
3. None
4. failwith “empty queue”

Answer: 2

Implementing Linked Queues

Representing links

Implement the behavior

```
module ListQueue : QUEUE = struct

  type 'a queue = { mutable contents : 'a list }

  let create () : 'a queue =
    { contents = [] }

  let is_empty (q:'a queue) : bool =
    q.contents = []

  let enq (x:'a) (q:'a queue) : unit =
    q.contents <- (q.contents @ [x])

  let deq (q:'a queue) : 'a =
    begin match q.contents with
      | [] -> failwith "deq called on empty queue"
      | x::tl -> q.contents <- tl; x
    end
end
```

Here we are using type abstraction to protect the state. Outside of the module, no one knows that queues are implemented with a mutable structure. So, only these functions can modify this structure.

A Better Implementation

- Implementation is slow because of append:
 - `q.contents @ [x]` copies the entire list each time
 - As the queue gets longer, it takes longer to add data
 - Only has a *single* reference to the beginning of the list
- Let's do it again with TWO references, one to the beginning (head) and one to the end (tail).
 - Dequeue by updating the head reference (as before)
 - Enqueue by updating the tail of the list
- Challenge: The list itself must be mutable
 - because we add to one end and remove from the other

Data Structure for Mutable Queues

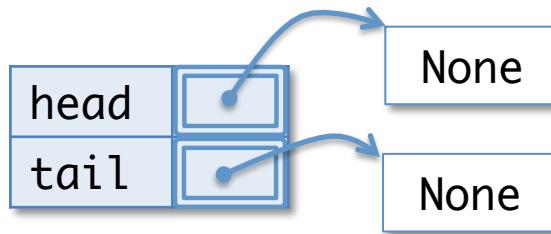
```
type 'a qnode = {  
    v: 'a;  
    mutable next : 'a qnode option  
}  
  
type 'a queue = { mutable head : 'a qnode option;  
                  mutable tail : 'a qnode option }
```

There are two parts to a mutable queue:

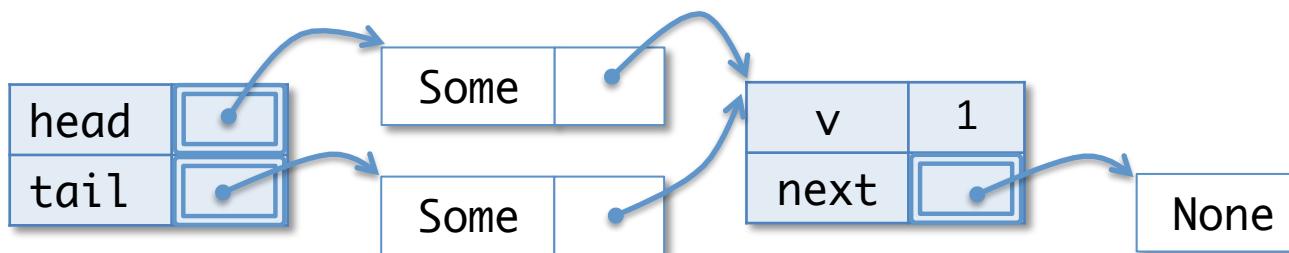
1. the “internal nodes” of the queue, with links from one to the next
2. a record with links to the head and tail nodes

All of the links are *optional* so that the queue can be empty.

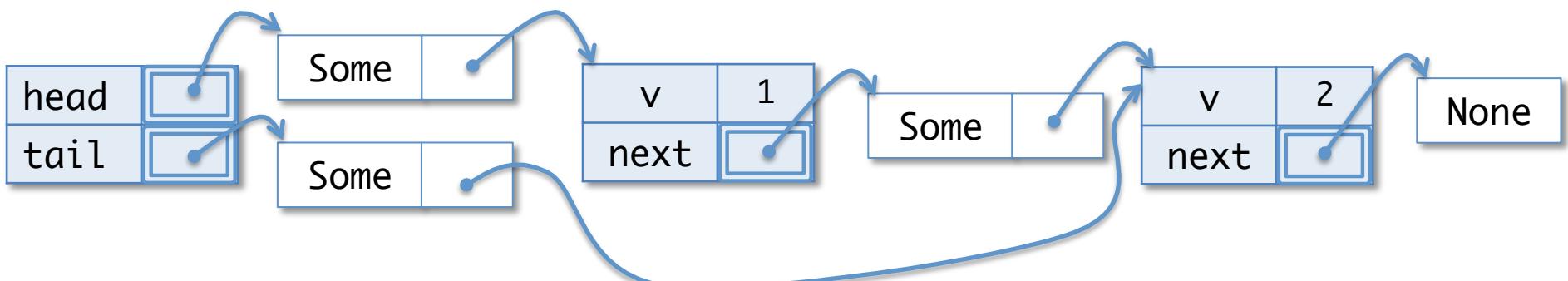
Queues in the Heap



An empty queue

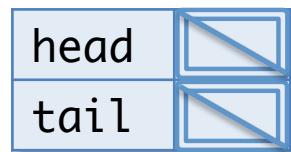


A queue with one element

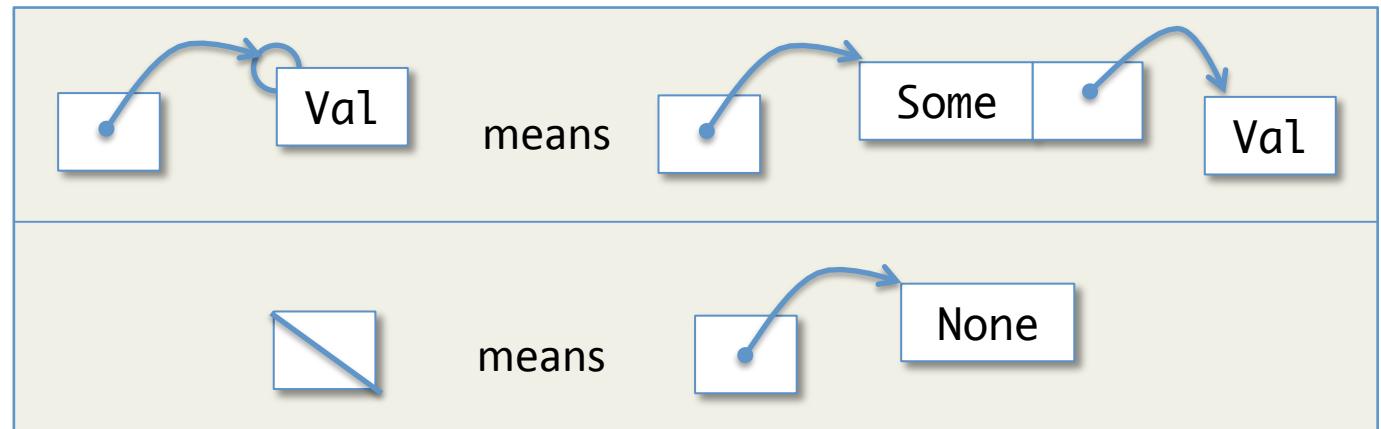


A queue with two elements

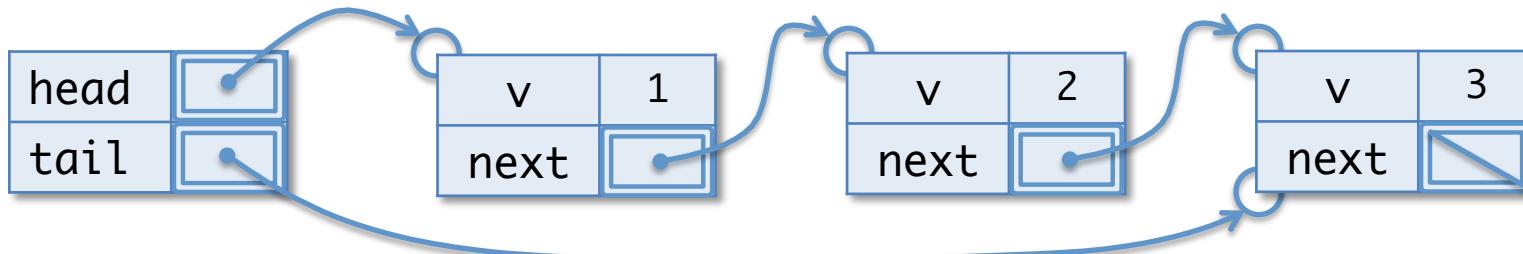
Visual Shorthand: Abbreviating Options



An empty queue



A queue with one element

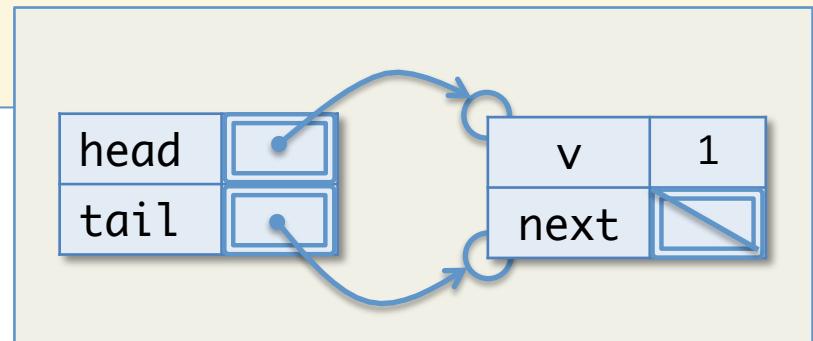


A queue with three elements

Given the queue datatype shown below, which expression creates a 1-element queue in the heap:

```
type 'a qnode = {  
    v: 'a;  
    mutable next : 'a qnode option  
}
```

```
type 'a queue = { mutable head : 'a qnode option;  
                  mutable tail : 'a qnode option }
```



1. `let q = { head = None; tail = None }`
2. `let q = { head = 1; tail = None }`
3. `let q = let qn = { v= 1; next = None } in
 { head = qn; tail = None }`
4. `let q = let qn = { v= 1; next = None } in
 { head = Some qn; tail = Some qn }`

Answer: 4

Programming Languages and Techniques (CIS120)

Lecture 15

October 4, 2017

Queues

Lecture notes: Chapter 16

Announcements

- No recitation sections this week (Fall Break!)
- Dr. Zdancewic will cover the noon lecture on Weds.
- Homework 4
 - due on October 10th
- Midterm 1
 - *October 13th in Class*
 - Where? Last Names:
 - A – M Leidy Labs 10 (Here)
 - N – Z Meyerson Hall B1
 - Covers lecture material through Chapter 13
 - Review materials (old exams) on course website
- **Review Session:**
 - Wednesday, Oct. 11th 6:00-8:00pm, Towne 100

Putting State to Work: Mutable Queues

A design problem

Suppose you are implementing a website for constituents to submit questions to their political representatives. To be fair, you would like to deal with questions in first-come, first-served order. How would you do it?

- Understand the problem
 - Need to keep track of pending questions, in the order in which they were submitted
- Define the interface
 - Need a data structure to store questions
 - Need to add questions to the *end* of the queue
 - Need to allow responders to retrieve questions from the *beginning* of the queue
 - Both kinds of access must be efficient to handle large volume

(Mutable) Queue Interface

```
module type QUEUE =
sig
  (* abstract type *)
  type 'a queue

  (* Make a new, empty queue *)
  val create : unit -> 'a queue

  (* Determine if a queue is empty *)
  val is_empty : 'a queue -> bool

  (* Add a value to the end of a queue *)
  val enq : 'a -> 'a queue -> unit

  (* Remove the first value (if any) and return it *)
  val deq : 'a queue -> 'a

end
```

Q: We can tell, just looking at this interface, that it is for a MUTABLE data structure. How?

Since queues are mutable, we must allocate a new one every time we need one.

A: Adding an element to a queue returns `unit` because it *modifies* the given queue.

Specify the behavior via test cases

```
let test () : bool =
  let q : int queue = create () in
  enq 1 q;
  enq 2 q;
  1 = deq q
;; run_test "queue test 1" test

let test () : bool =
  let q : int queue = create () in
  enq 1 q;
  enq 2 q;
  let _ = deq q in
  2 = deq q
;; run_test "queue test 2" test
```

Implementing Linked Queues

Representing links

Data Structure for Mutable Queues

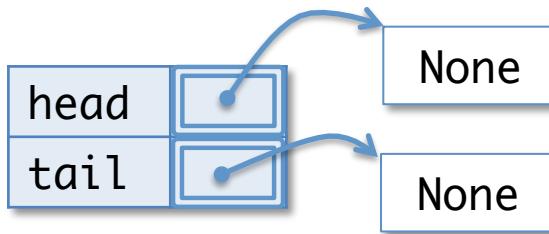
```
type 'a qnode = {  
    v: 'a;  
    mutable next : 'a qnode option  
}  
  
type 'a queue = { mutable head : 'a qnode option;  
                  mutable tail : 'a qnode option }
```

There are two parts to a mutable queue:

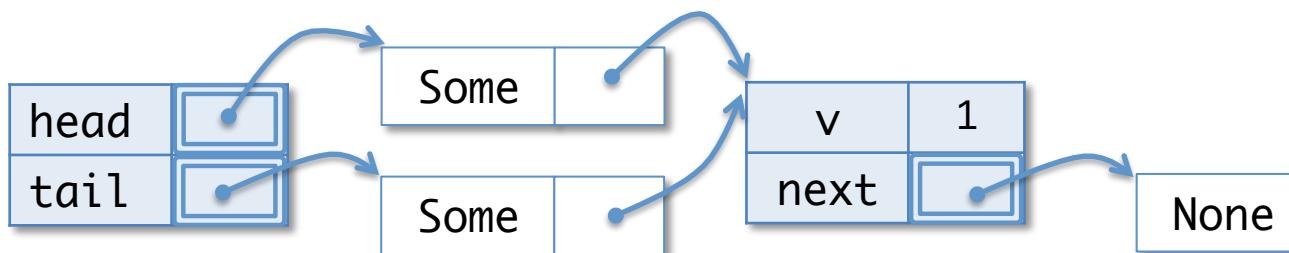
1. the “internal nodes” of the queue, with links from one to the next
2. a record with links to the head and tail nodes

All of the links are *optional* so that the queue can be empty.

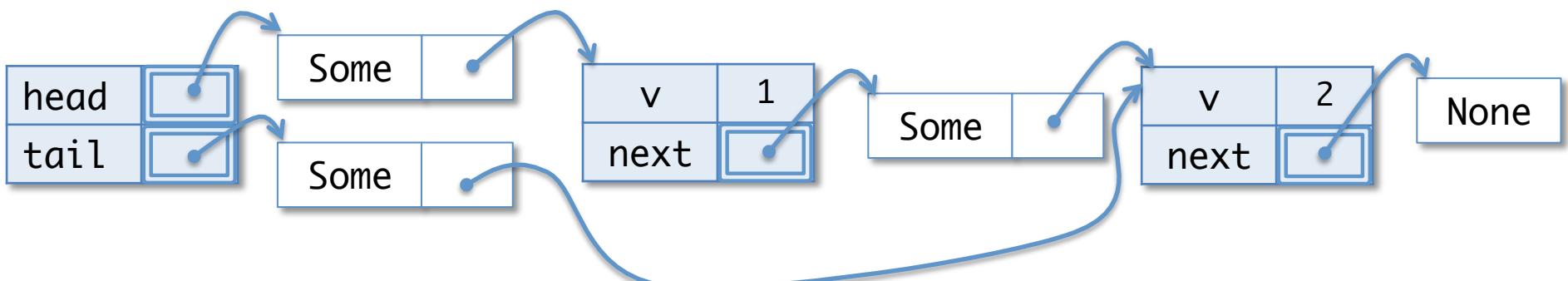
Queues in the Heap



An empty queue

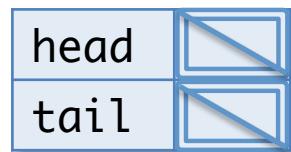


A queue with one element

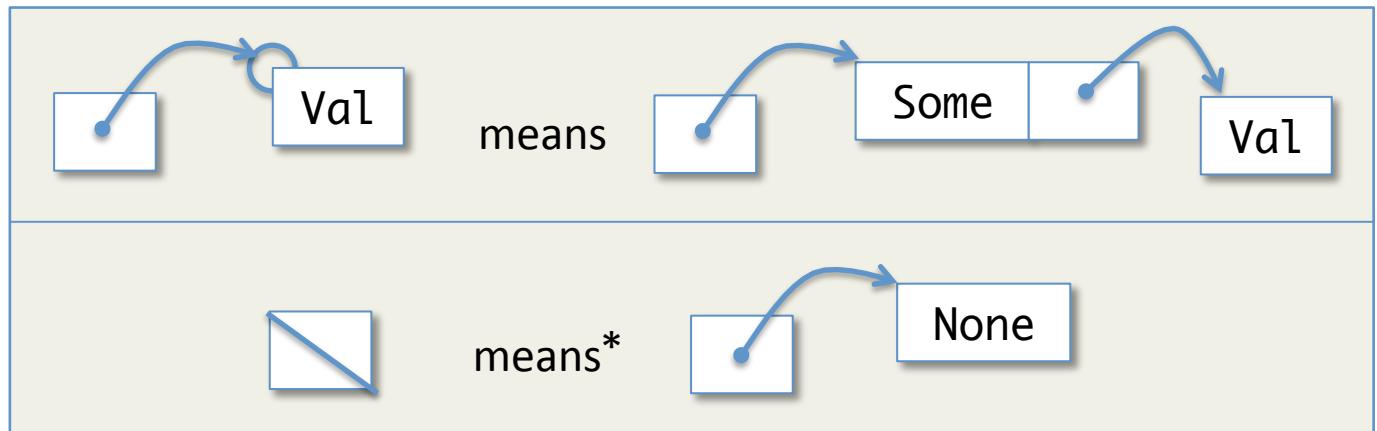


A queue with two elements

Visual Shorthand: Abbreviating Options

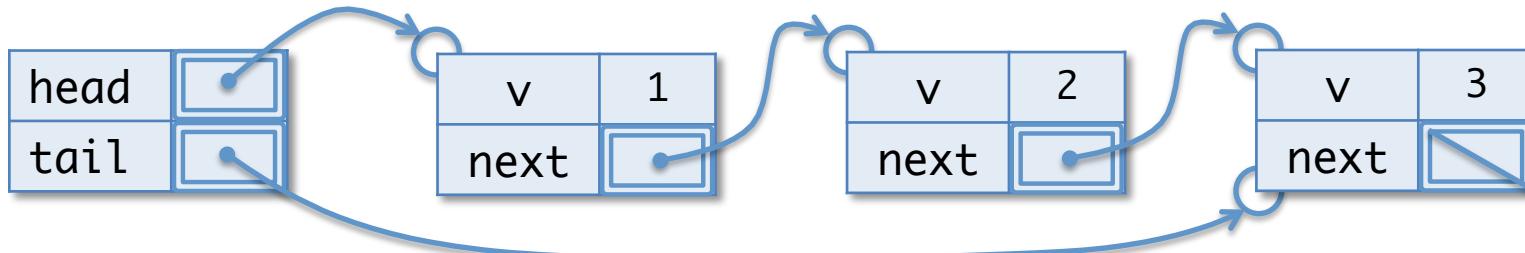


An empty queue



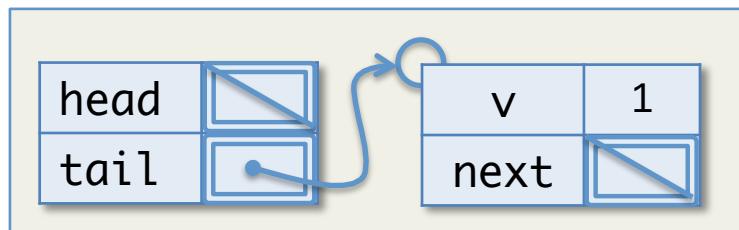
A queue with one element

*Note: Ocaml can optimize "nullary" constructors like Nil, None, Empty so that they aren't allocated in the heap. This is why `None == None` even though `not ((Some x) == (Some x))`. Be careful with equality and options.

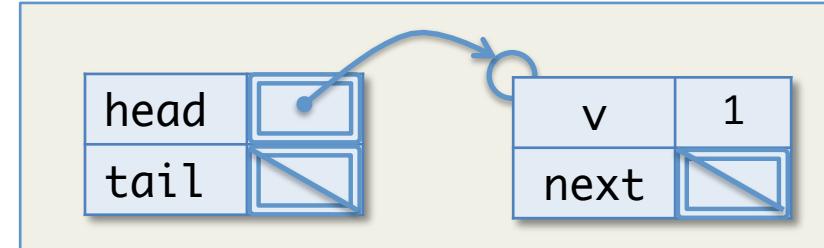


A queue with three elements

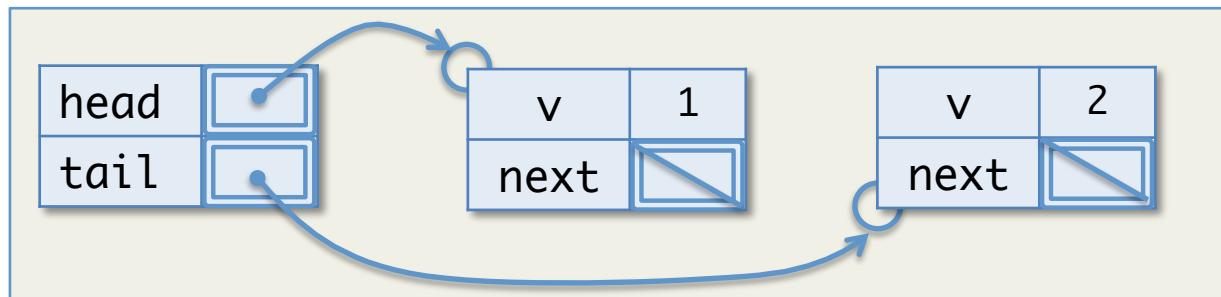
“Bogus” values of type `int queue`



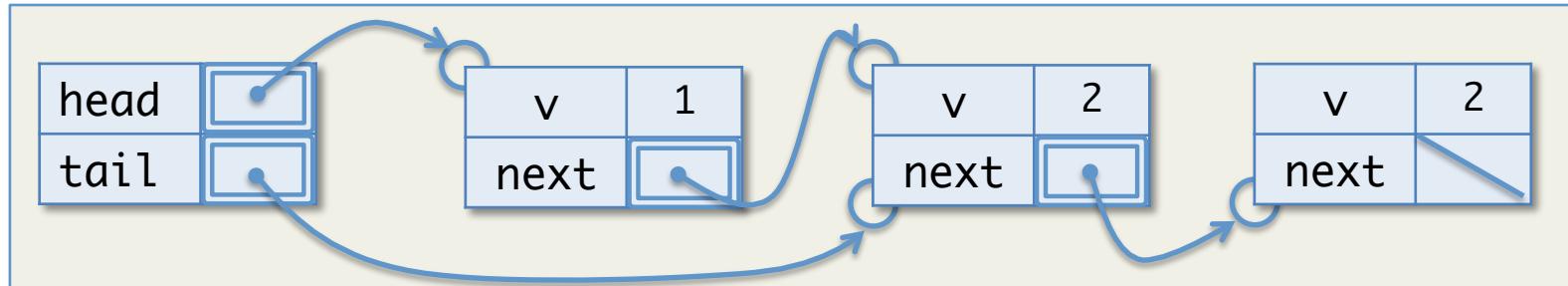
head is None, tail is Some



head is Some, tail is None



tail is not reachable from the head



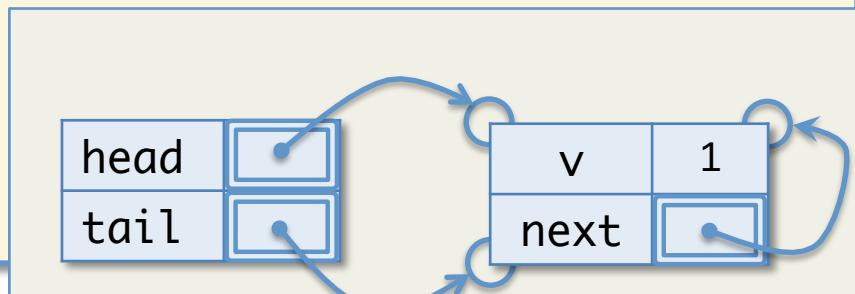
tail doesn't point to the last element of the queue

Given the queue datatype shown below, is it possible to create a *cycle* of references in the heap. (i.e. a way to get back to the same place by following references.)

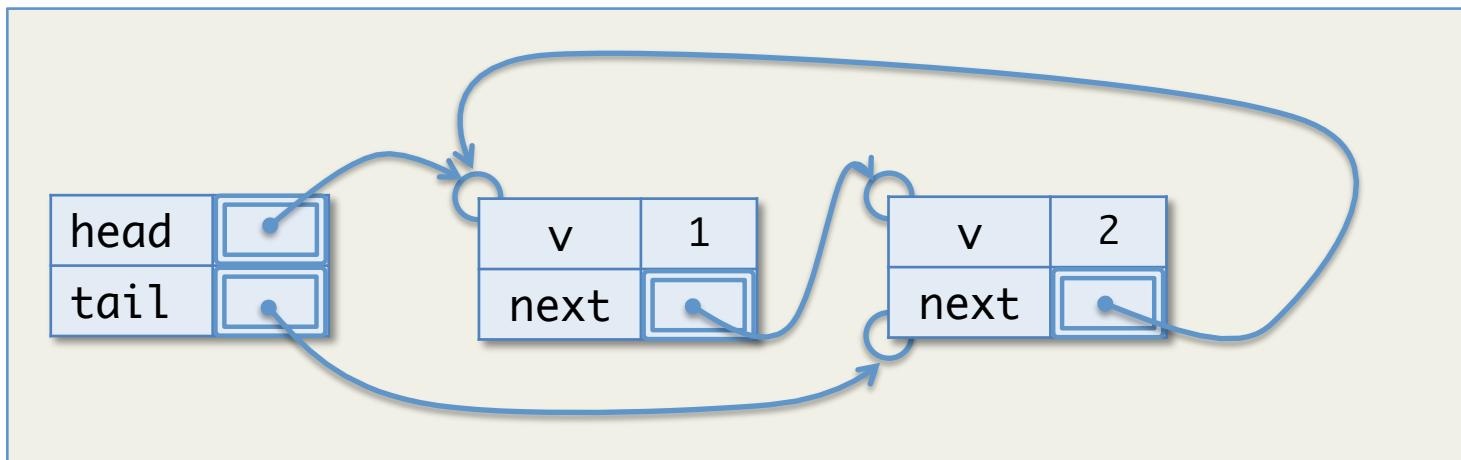
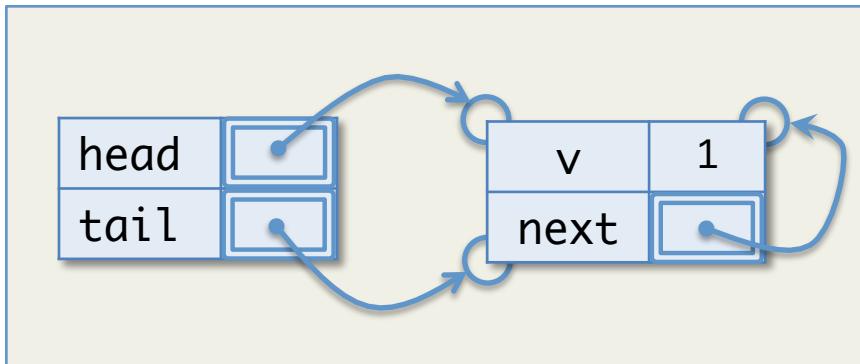
```
type 'a qnode = {  
    v: 'a;  
    mutable next : 'a qnode option  
}  
  
type 'a queue = { mutable head : 'a qnode option;  
                  mutable tail : 'a qnode option }
```

- 1. yes
- 2. no
- 3. not sure

Answer: 1



Cyclic int queue values



(And infinitely many more...)

Linked Queue Invariants

- Just as we imposed some restrictions on which trees count as legitimate Binary Search Trees, Linked Queues must also satisfy representation *invariants*:

Either:

(1) head and tail are both None (i.e. the queue is empty)

or

(2) head is Some n1, tail is Some n2 and

- n2 is reachable from n1 by following ‘next’ pointers
- n2.next is None

- We can prove that these properties suffice to rule out all of the “bogus” examples.
- Each queue operation may assume that these invariants hold of its inputs, and must ensure that the invariants hold when it’s done.

Either:

(1) `head` and `tail` are both `None` (i.e. the queue is empty)

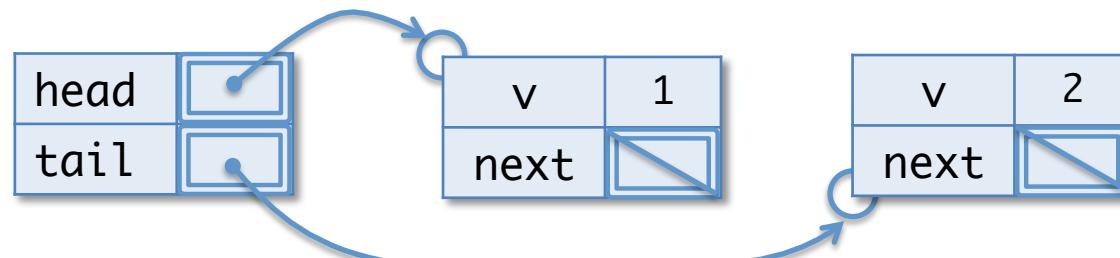
or

(2) `head` is `Some n1`, `tail` is `Some n2` and

- `n2` is reachable from `n1` by following 'next' pointers
- `n2.next` is `None`

Is this a valid queue?

1. Yes
2. No



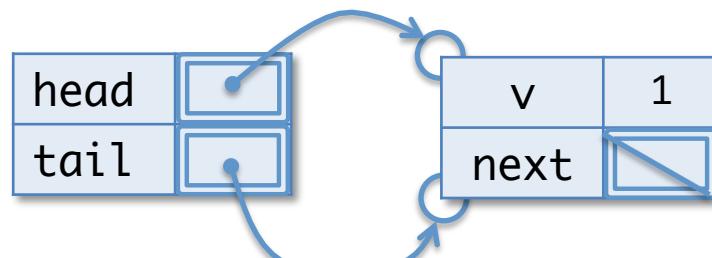
ANSWER: No

Either:

- (1) head and tail are both None (i.e. the queue is empty)
- or
- (2) head is Some n1, tail is Some n2 and
 - n2 is reachable from n1 by following 'next' pointers
 - n2.next is None

Is this a valid queue?

- 1. Yes
- 2. No



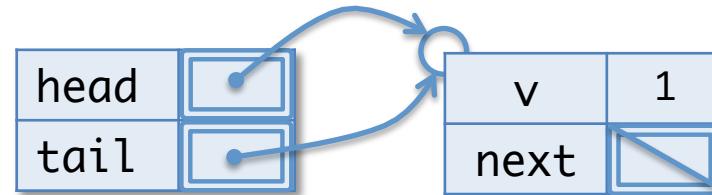
ANSWER: Yes

Either:

- (1) `head` and `tail` are both `None` (i.e. the queue is empty)
- or
- (2) `head` is `Some n1`, `tail` is `Some n2` and
 - `n2` is reachable from `n1` by following 'next' pointers
 - `n2.next` is `None`

Is this a valid queue?

1. Yes
2. No



ANSWER: Yes

Implementing Linked Queues

q.ml

create and is_empty

```
(* create an empty queue *)
let create () : 'a queue =
  { head = None;
    tail = None }
```

```
(* determine whether a queue is empty *)
let is_empty (q:'a queue) : bool =
  q.head = None
```

- *create establishes* the queue invariants
 - both head and tail are None
- *is_empty assumes* the queue invariants
 - it doesn't have to check that q.tail is None

enq

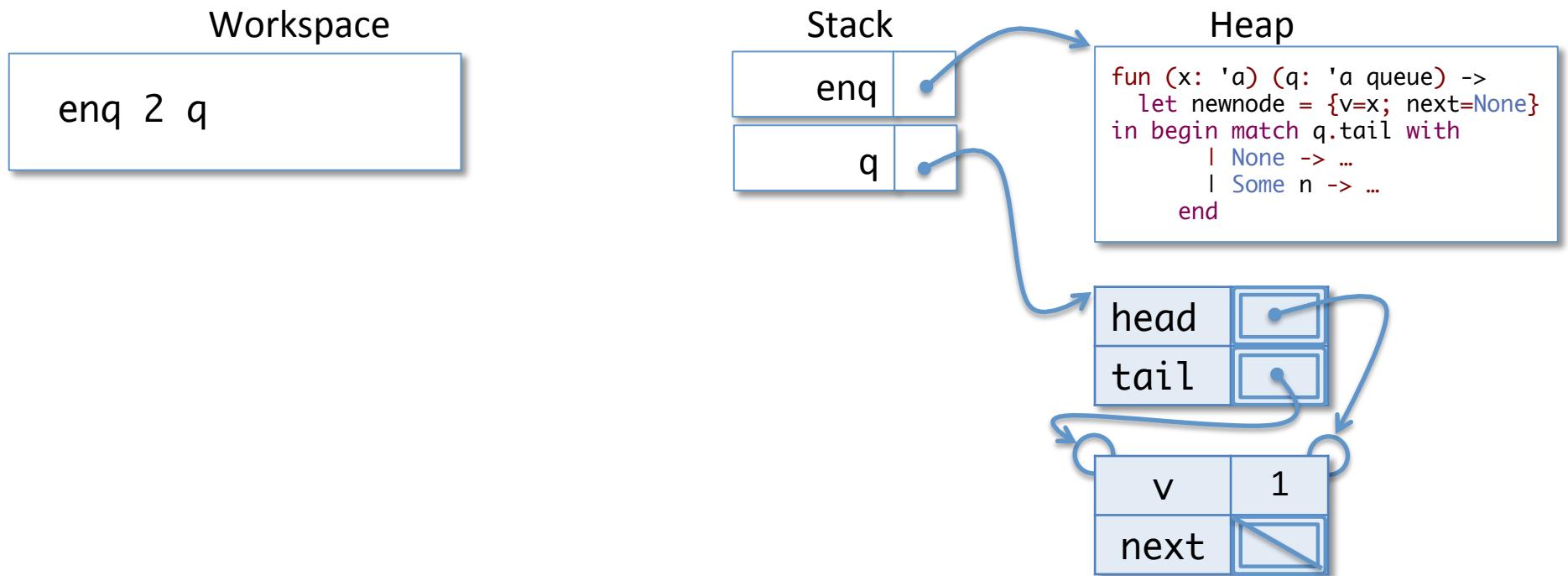
```
(* add an element to the tail of a queue *)
let enq (x: 'a) (q: 'a queue) : unit =
  let newnode = {v=x; next=None} in
  begin match q.tail with
    | None ->
        q.head <- Some newnode;
        q.tail <- Some newnode
    | Some n ->
        n.next <- Some newnode;
        q.tail <- Some newnode
  end
```

- The code for `enq` is informed by the queue invariant:
 - either the queue is empty, and we just update head and tail, or
 - the queue is non-empty, in which case we have to “patch up” the “next” link of the old tail node to maintain the queue invariant.

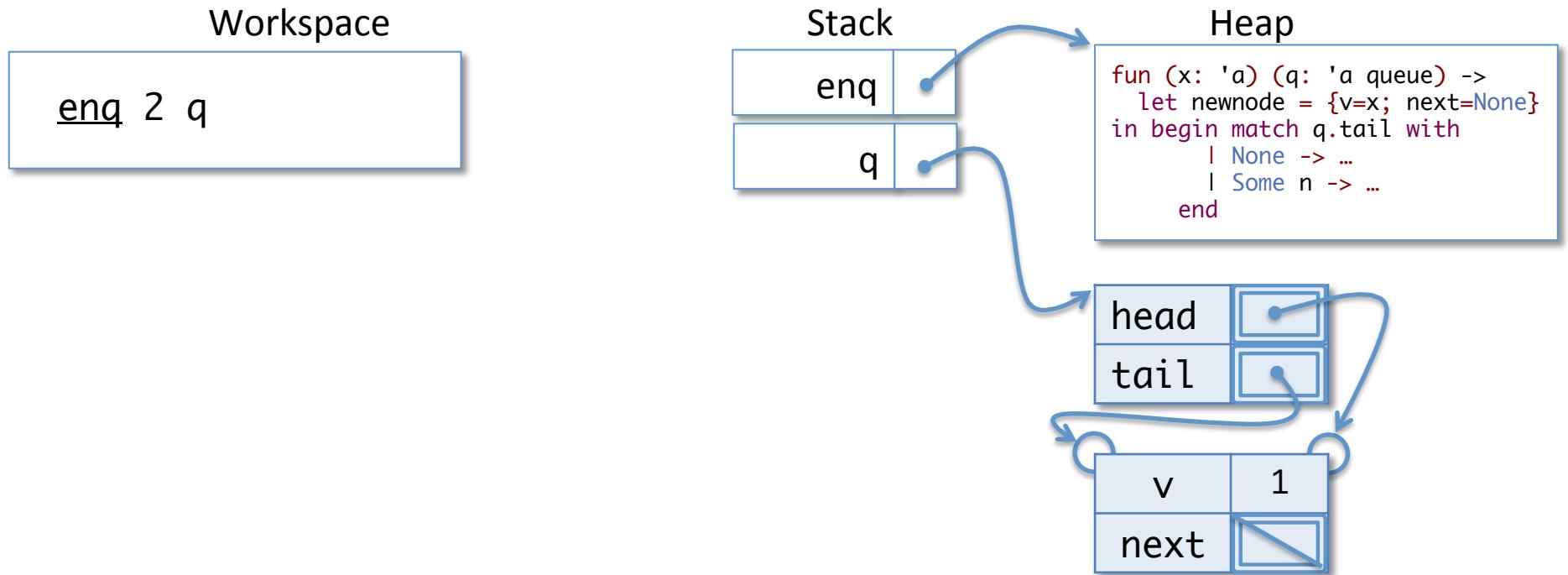
What is your current level of comfort with the Abstract Stack Machine?

1. got it well under control
2. OK but need to work with it a little more
3. a little puzzled
4. very puzzled
5. *very very puzzled* :-)

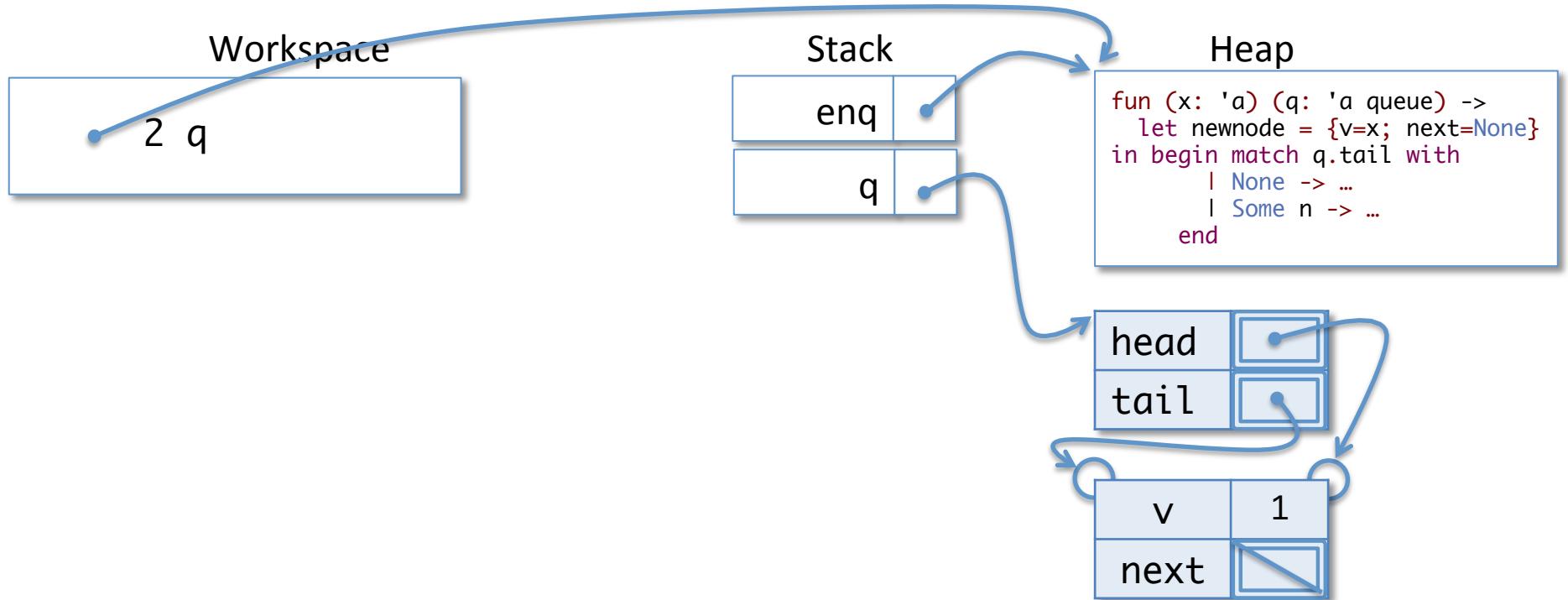
Calling Enq on a non-empty queue



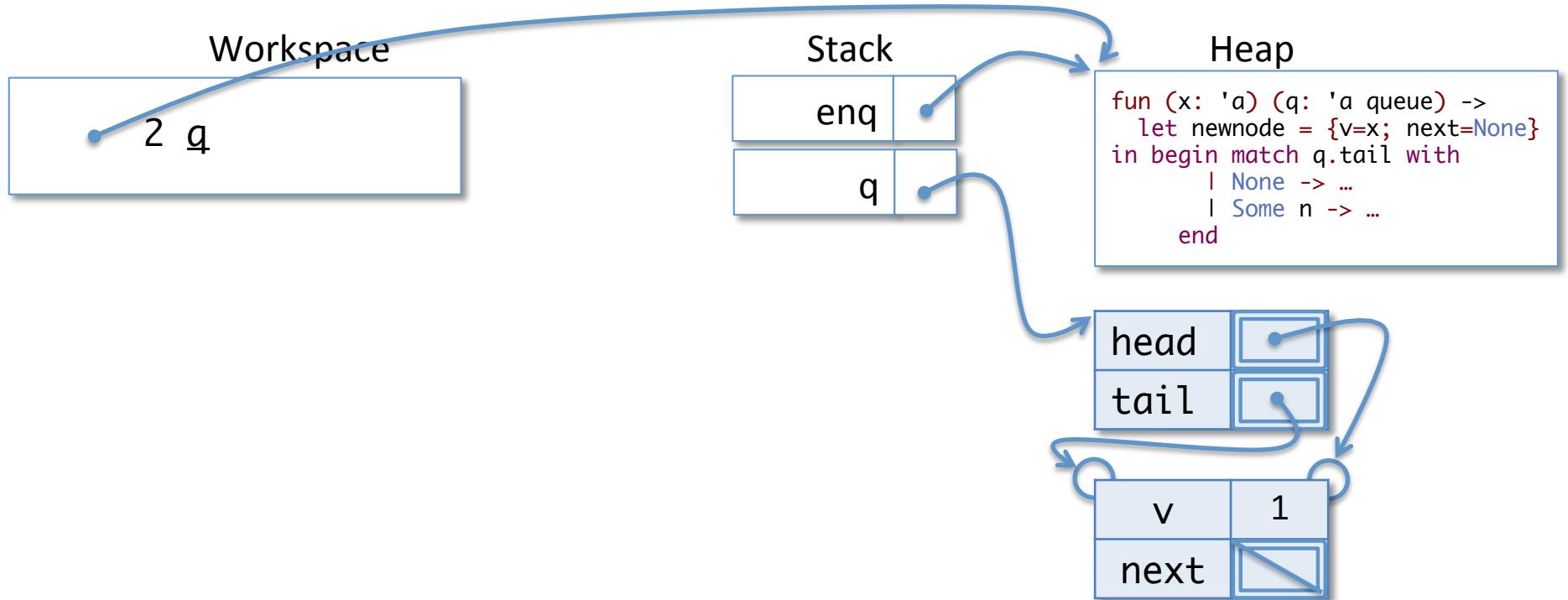
Calling Enq on a non-empty queue



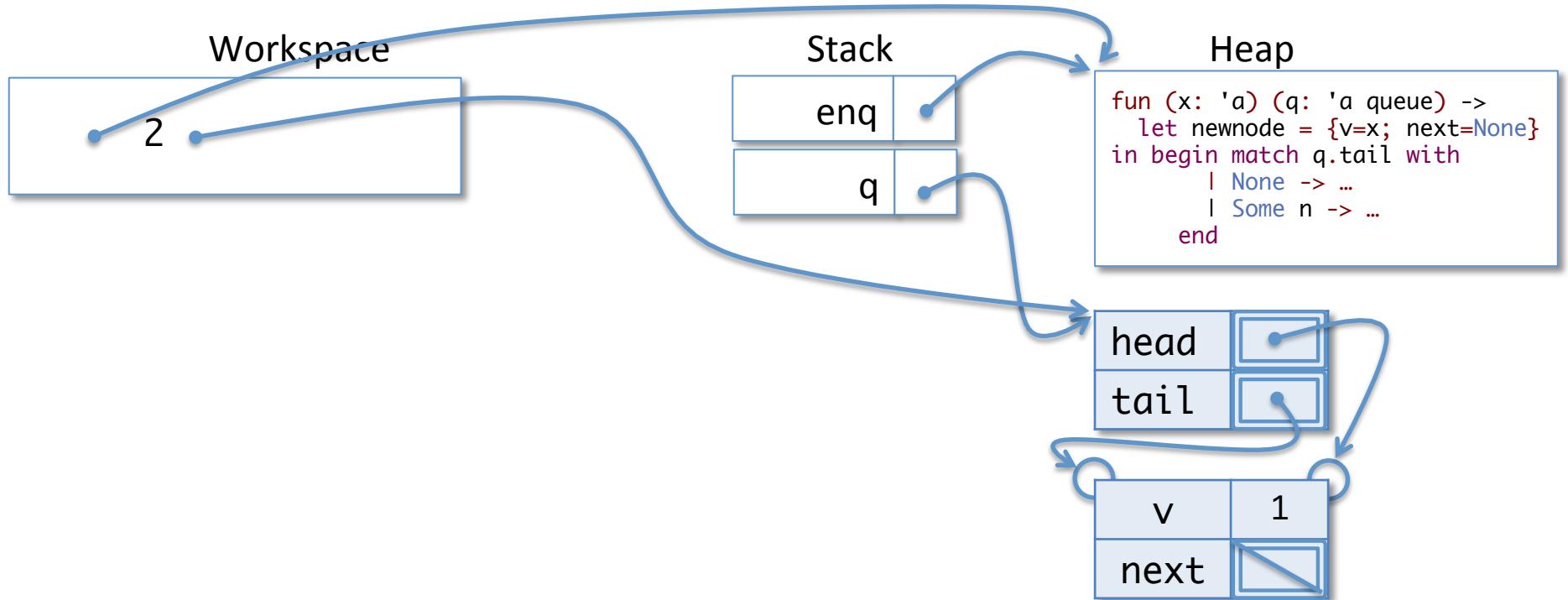
Calling Enq on a non-empty queue



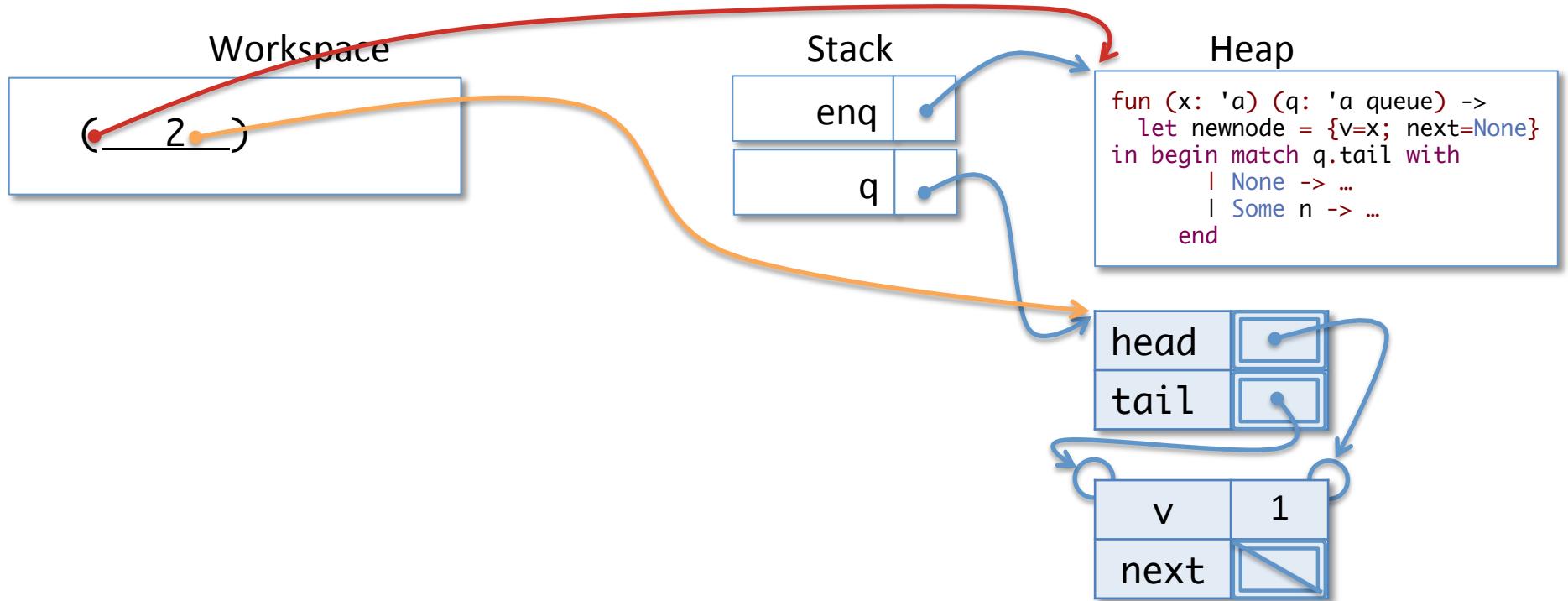
Calling Enq on a non-empty queue



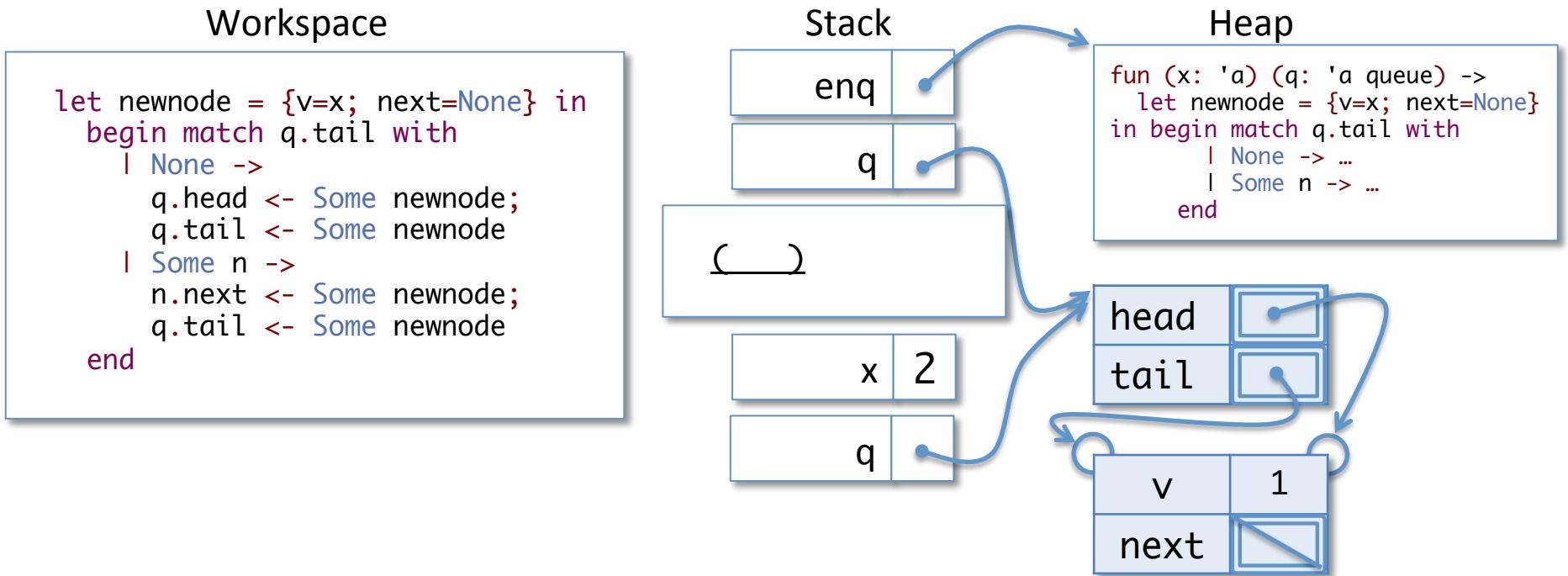
Calling Enq on a non-empty queue



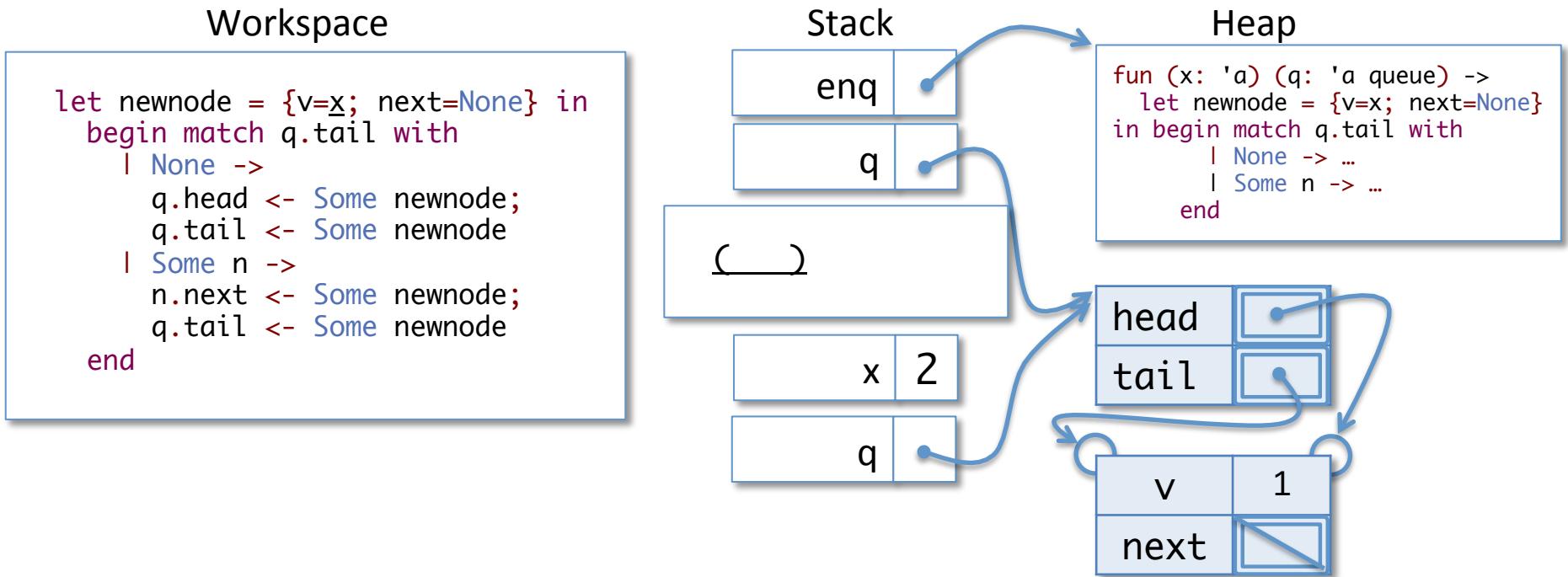
Calling Enq on a non-empty queue



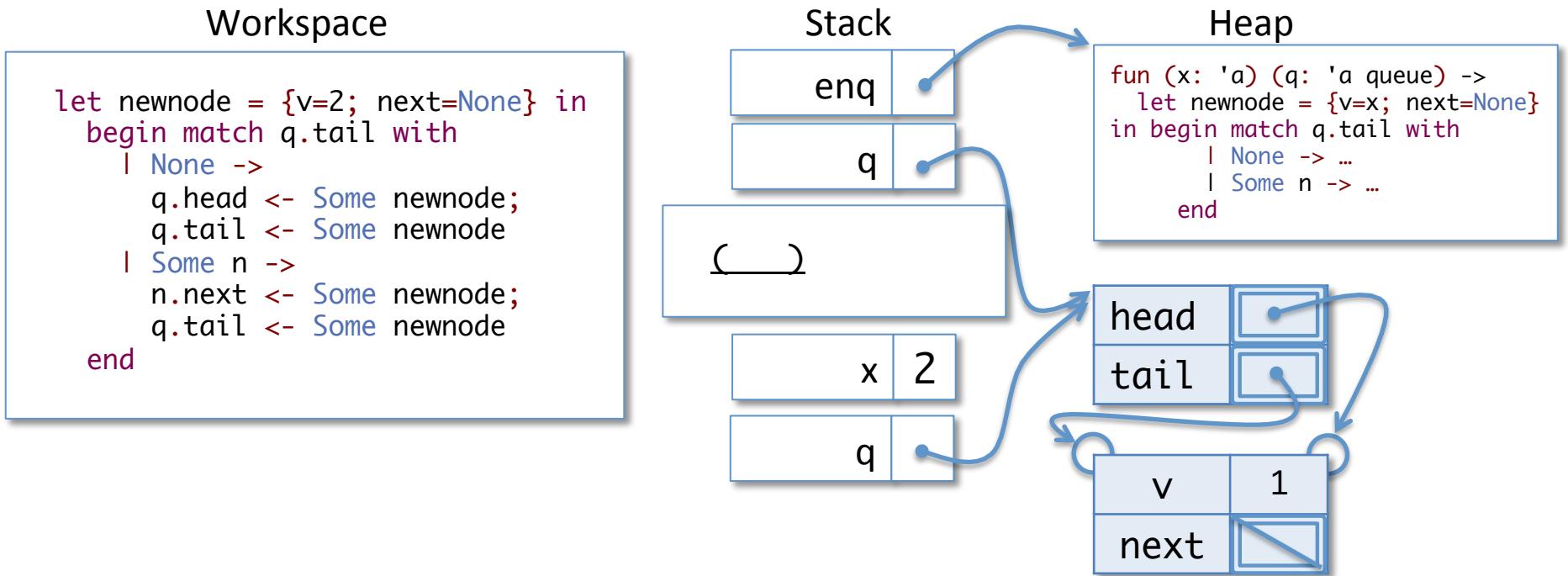
Calling Enq on a non-empty queue



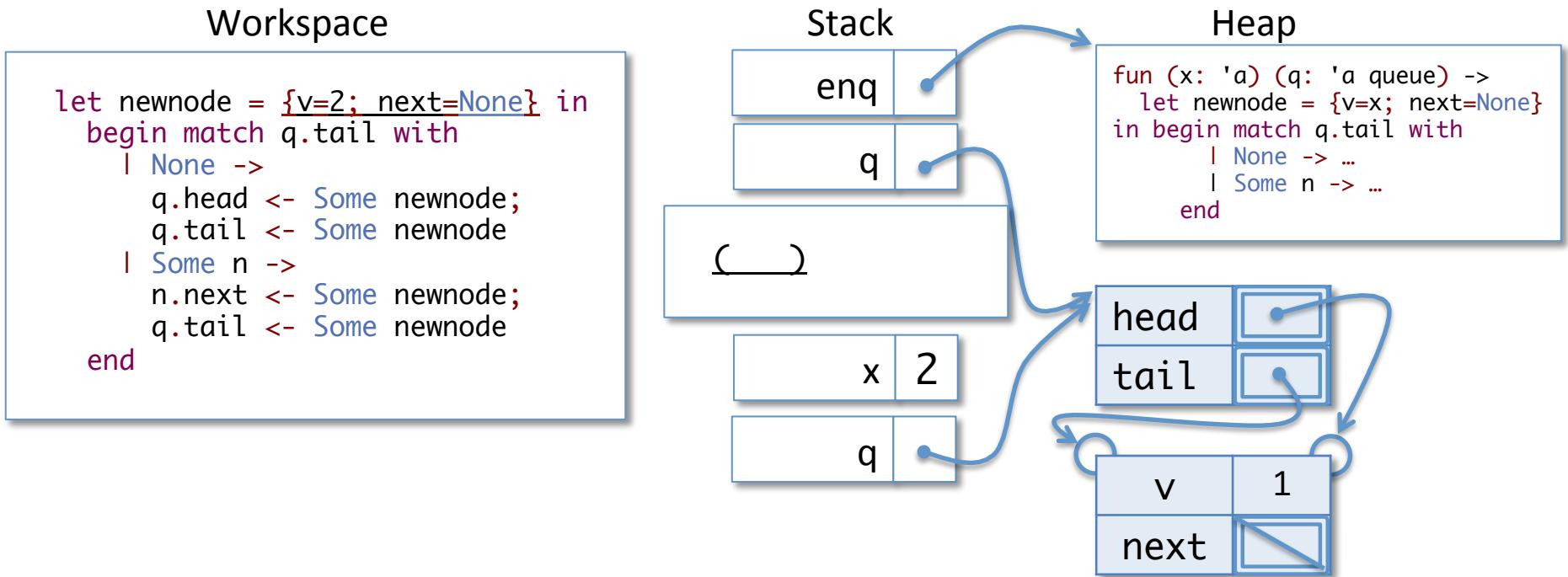
Calling Enq on a non-empty queue



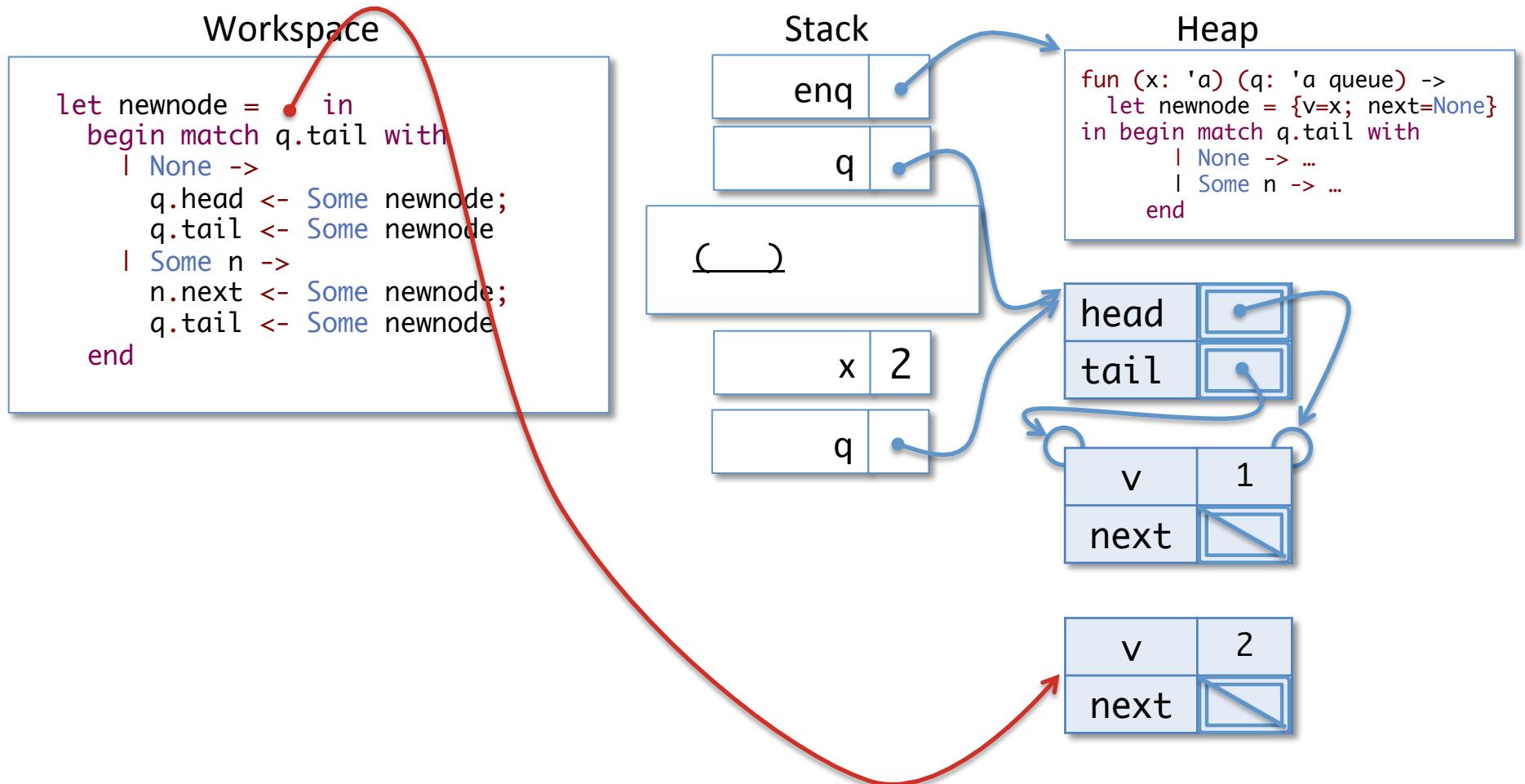
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue

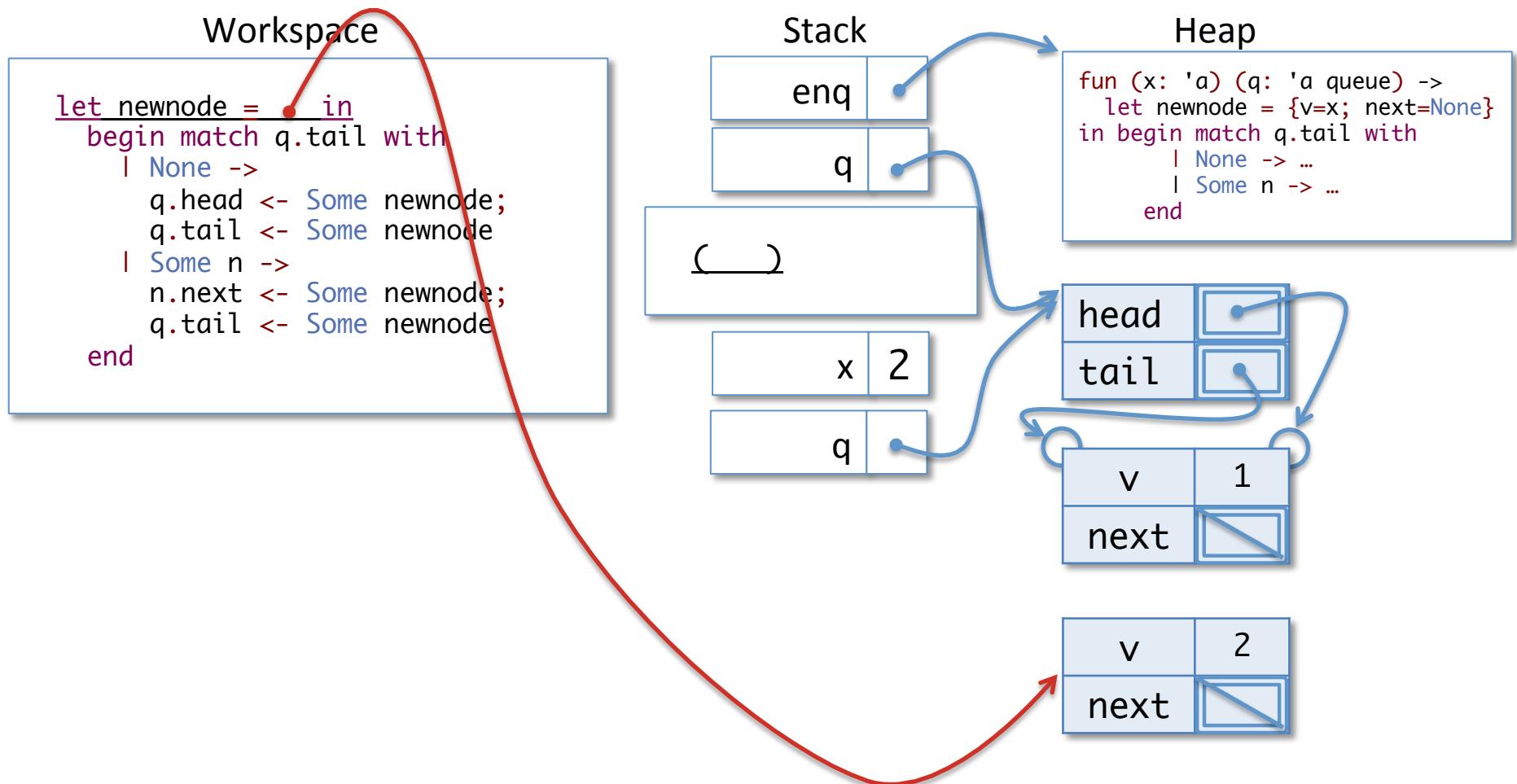


Calling Enq on a non-empty queue

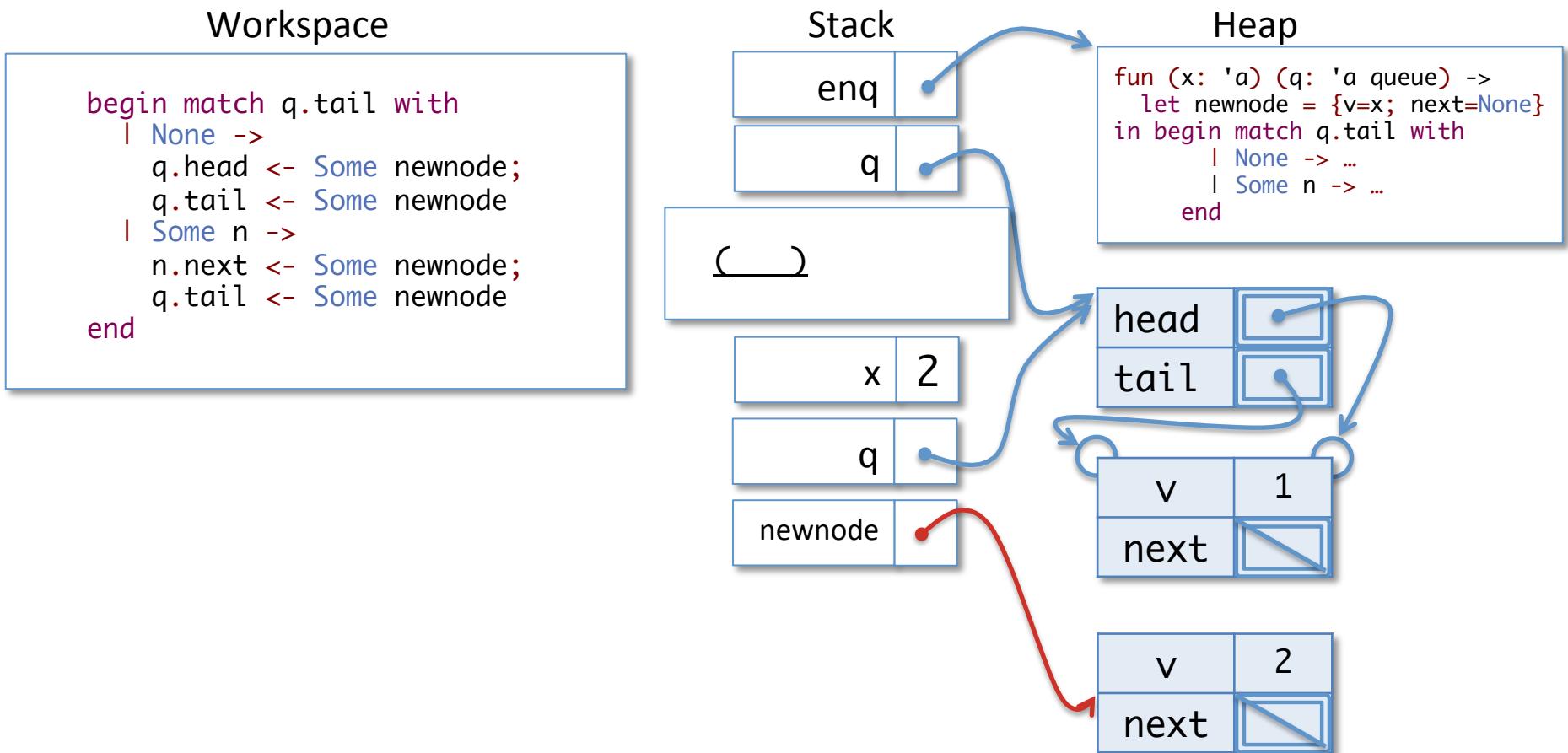


Note: there is no “Some bubble”: this is a qnode, not a qnode option.

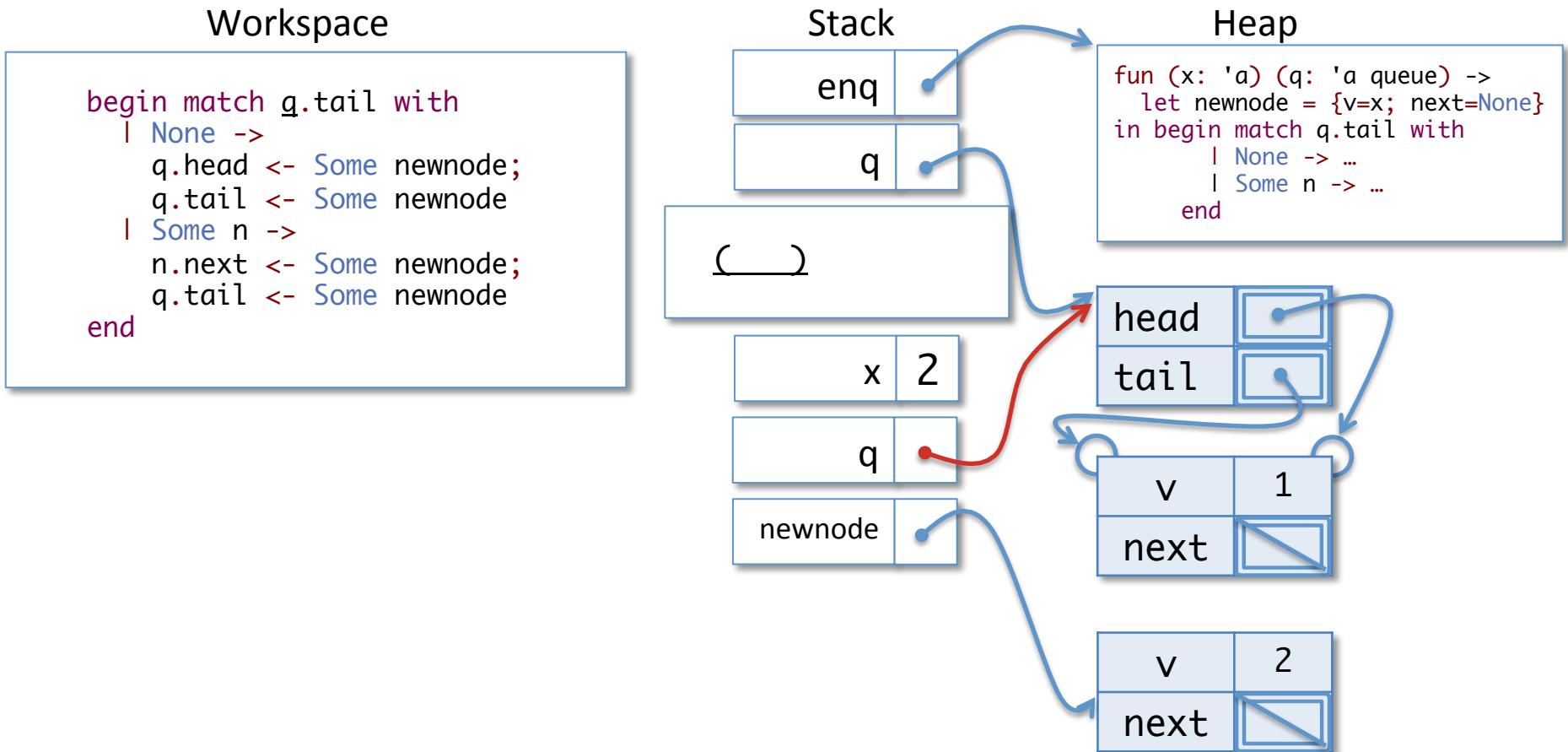
Calling Enq on a non-empty queue



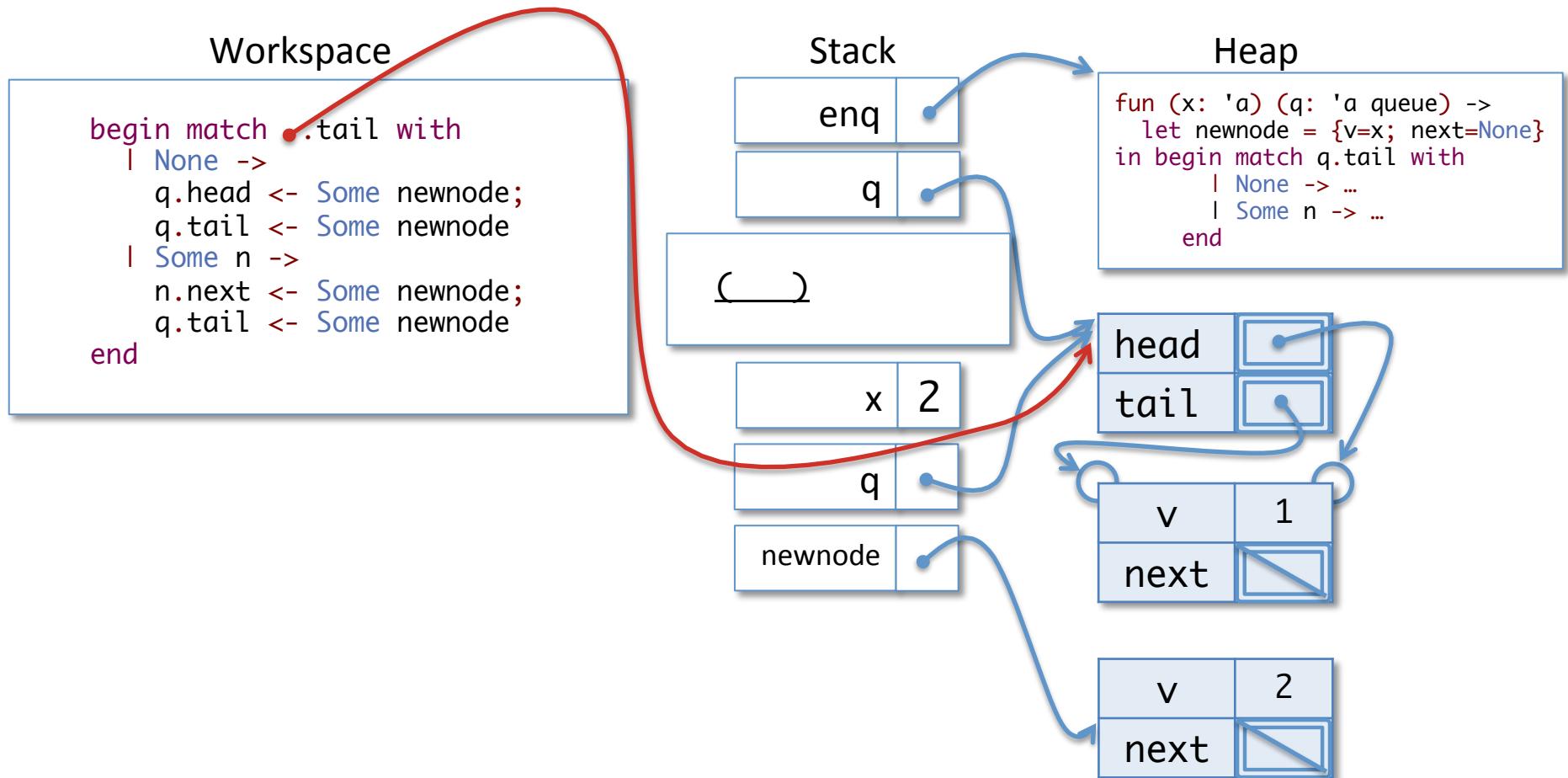
Calling Enq on a non-empty queue



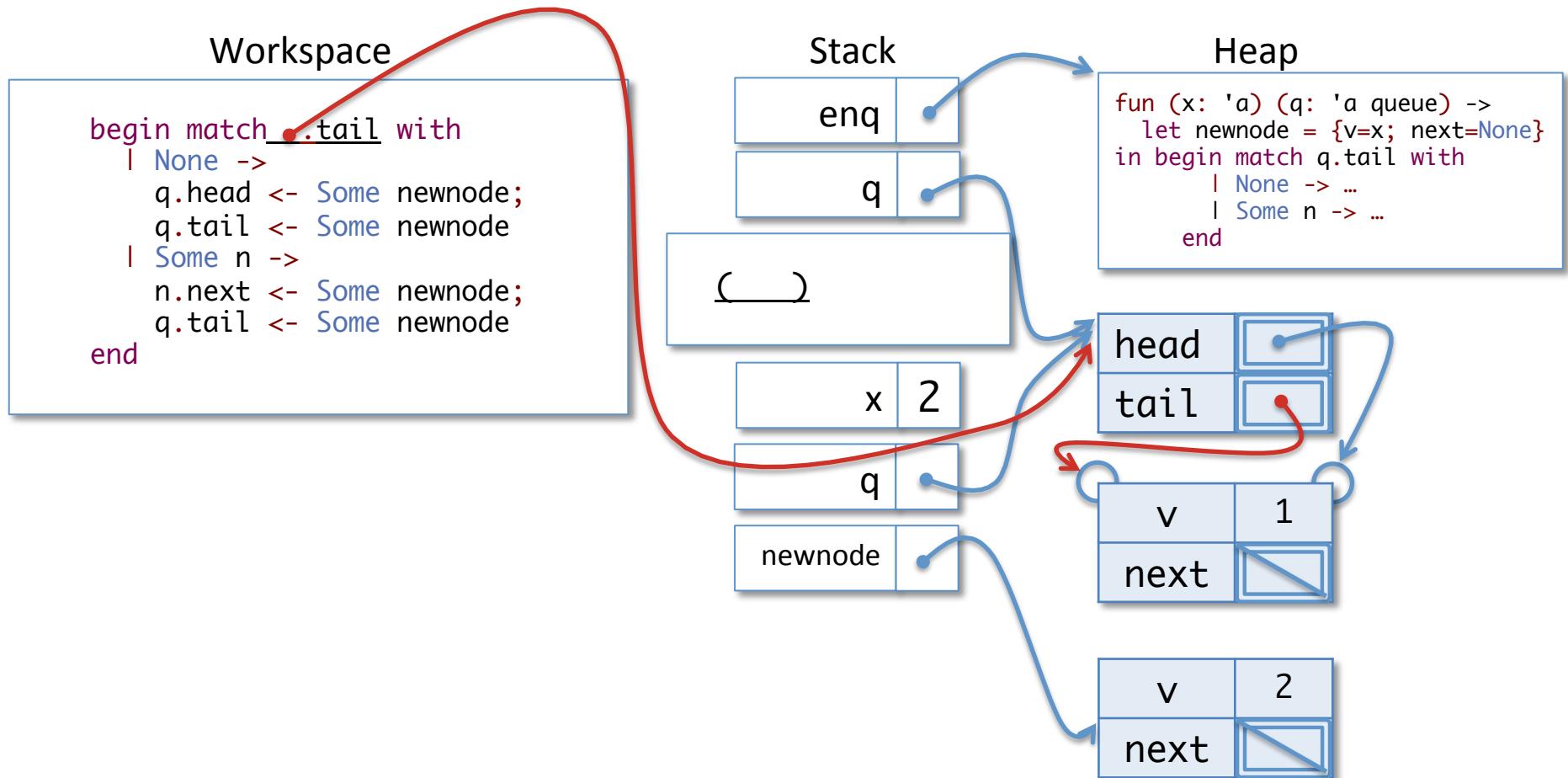
Calling Enq on a non-empty queue



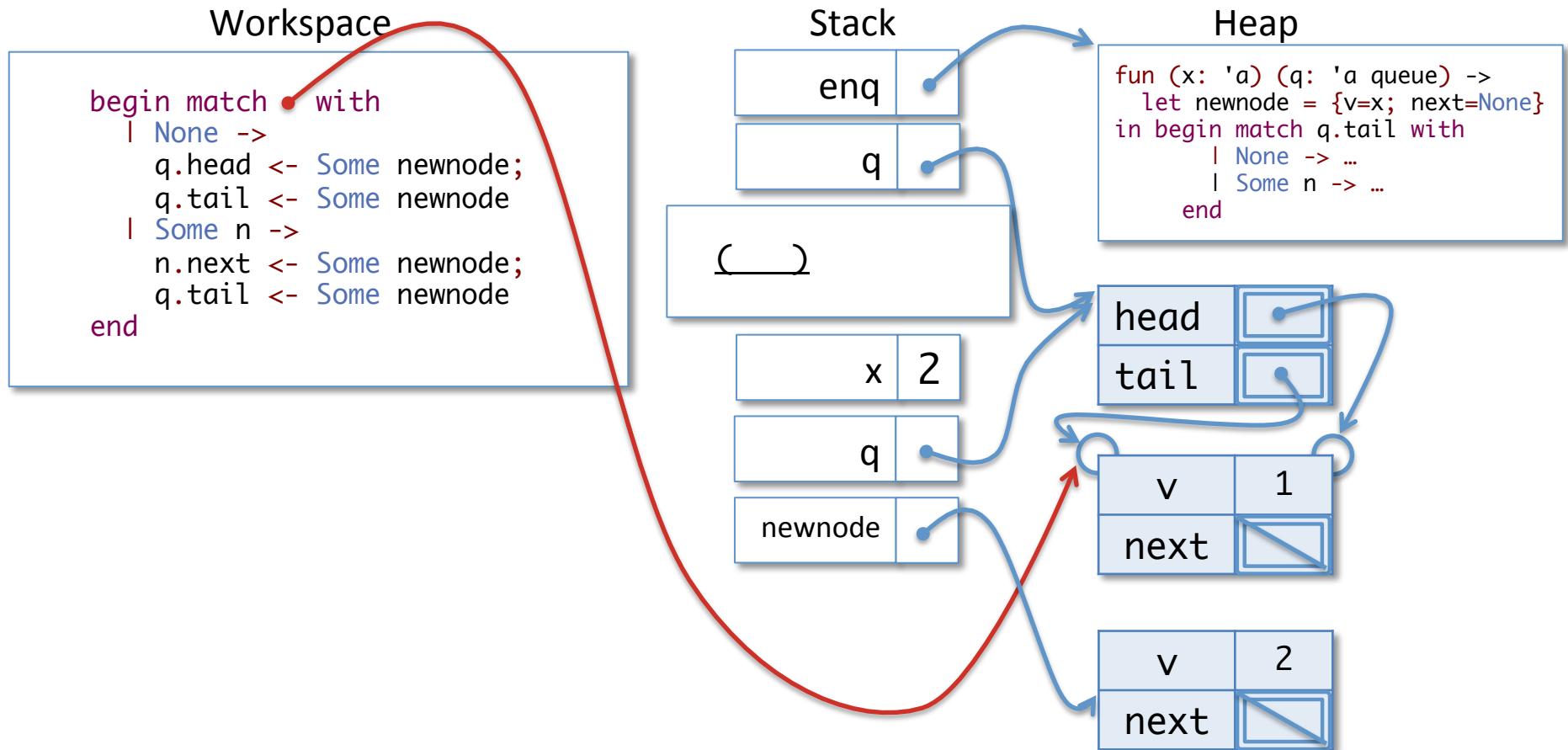
Calling Enq on a non-empty queue



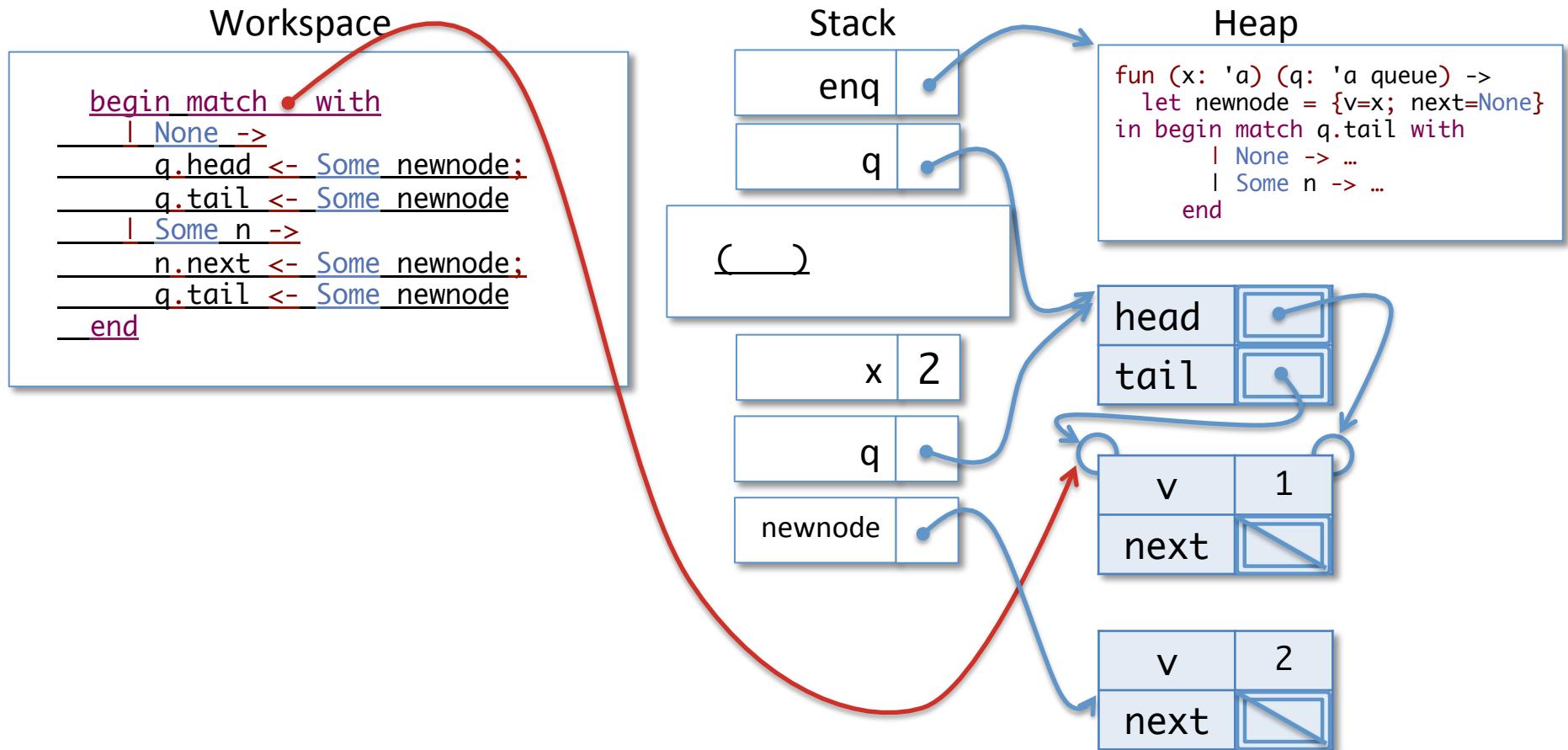
Calling Enq on a non-empty queue



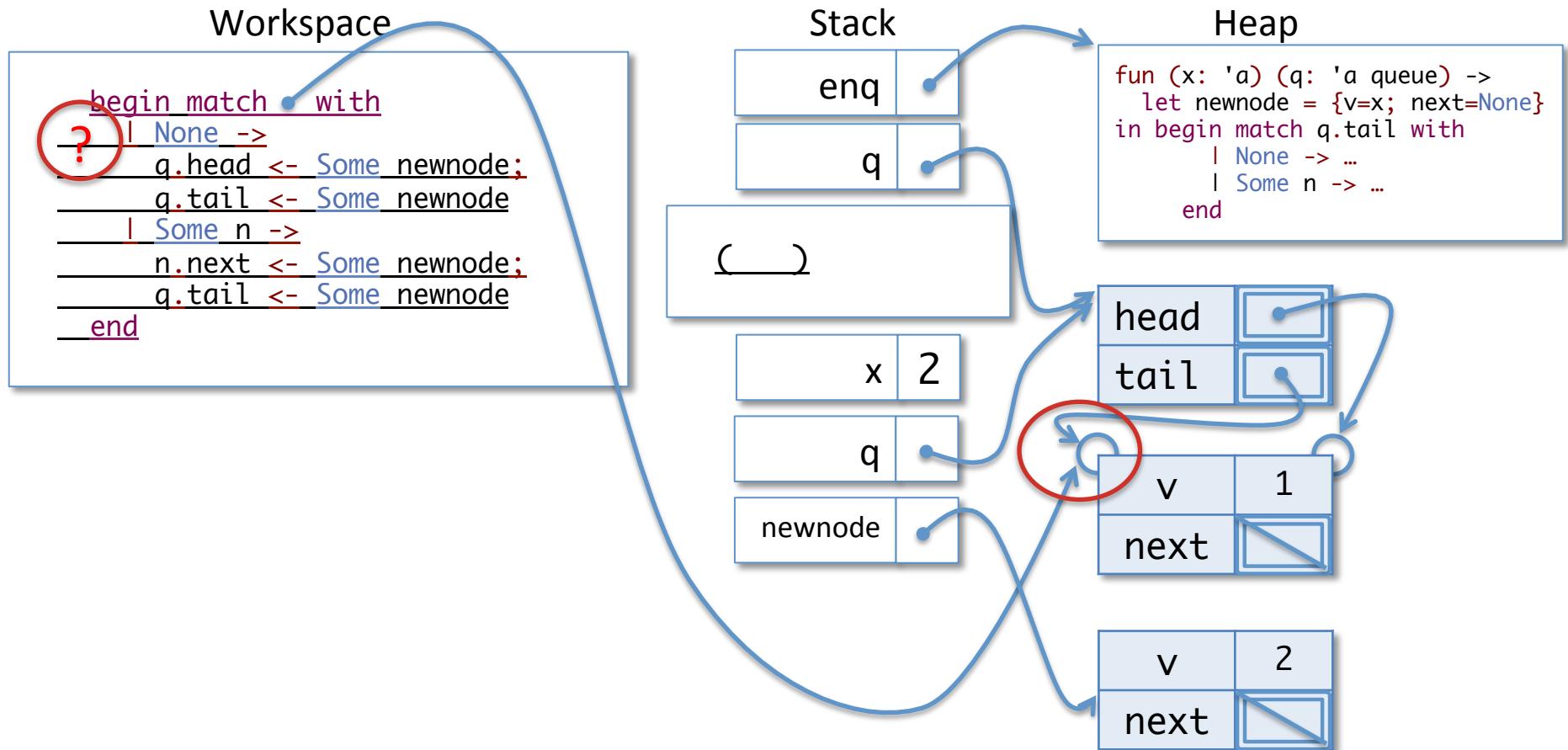
Calling Enq on a non-empty queue



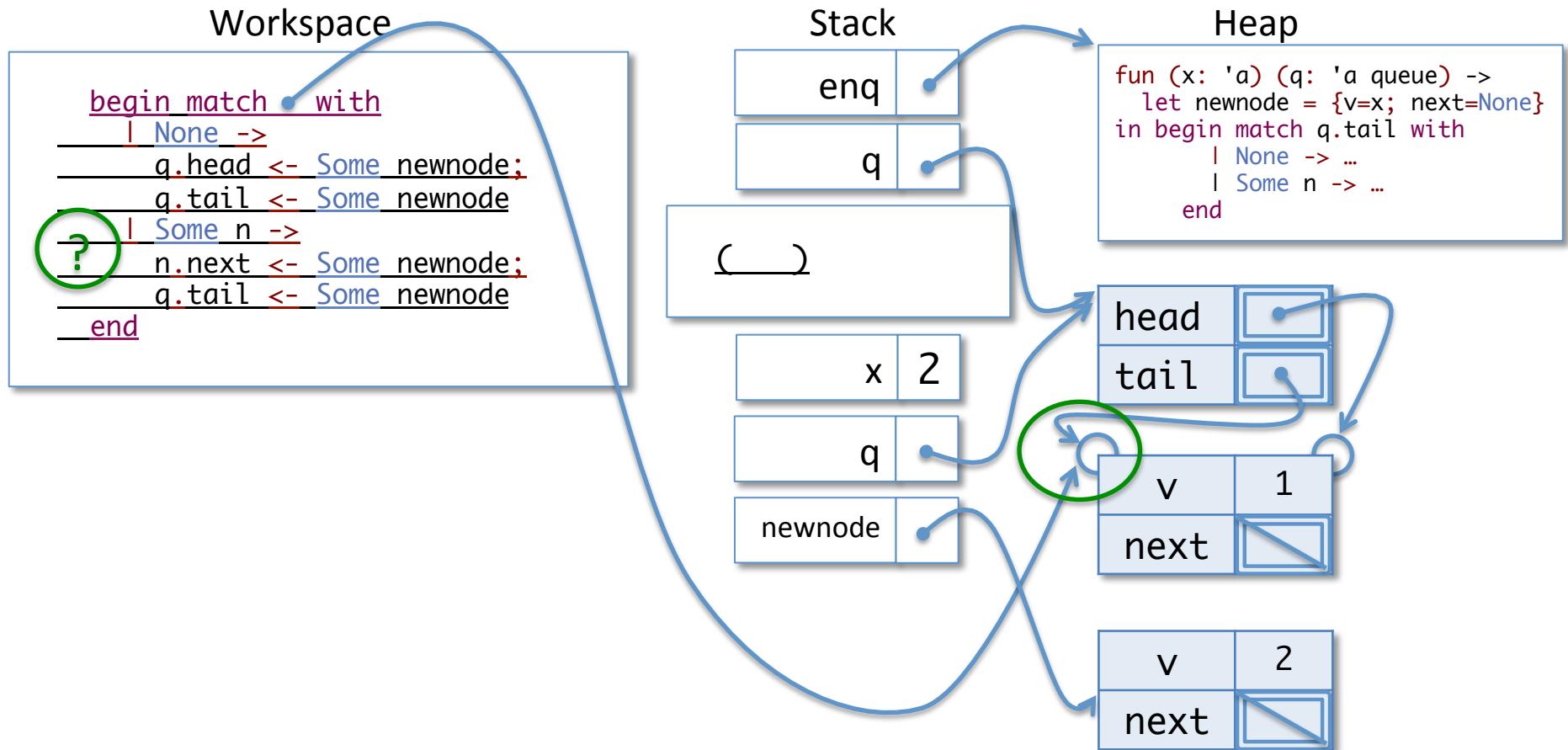
Calling Enq on a non-empty queue



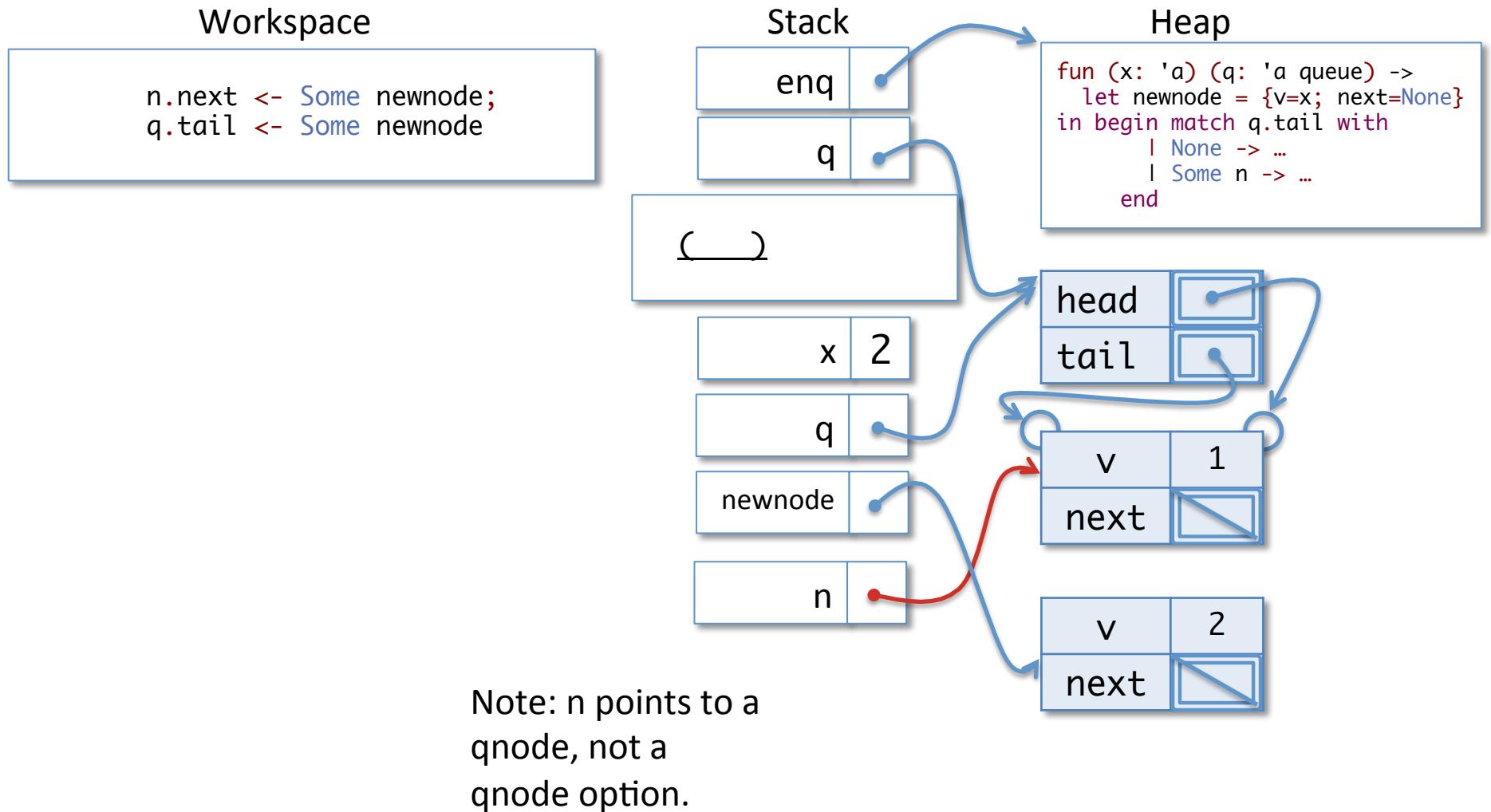
Calling Enq on a non-empty queue



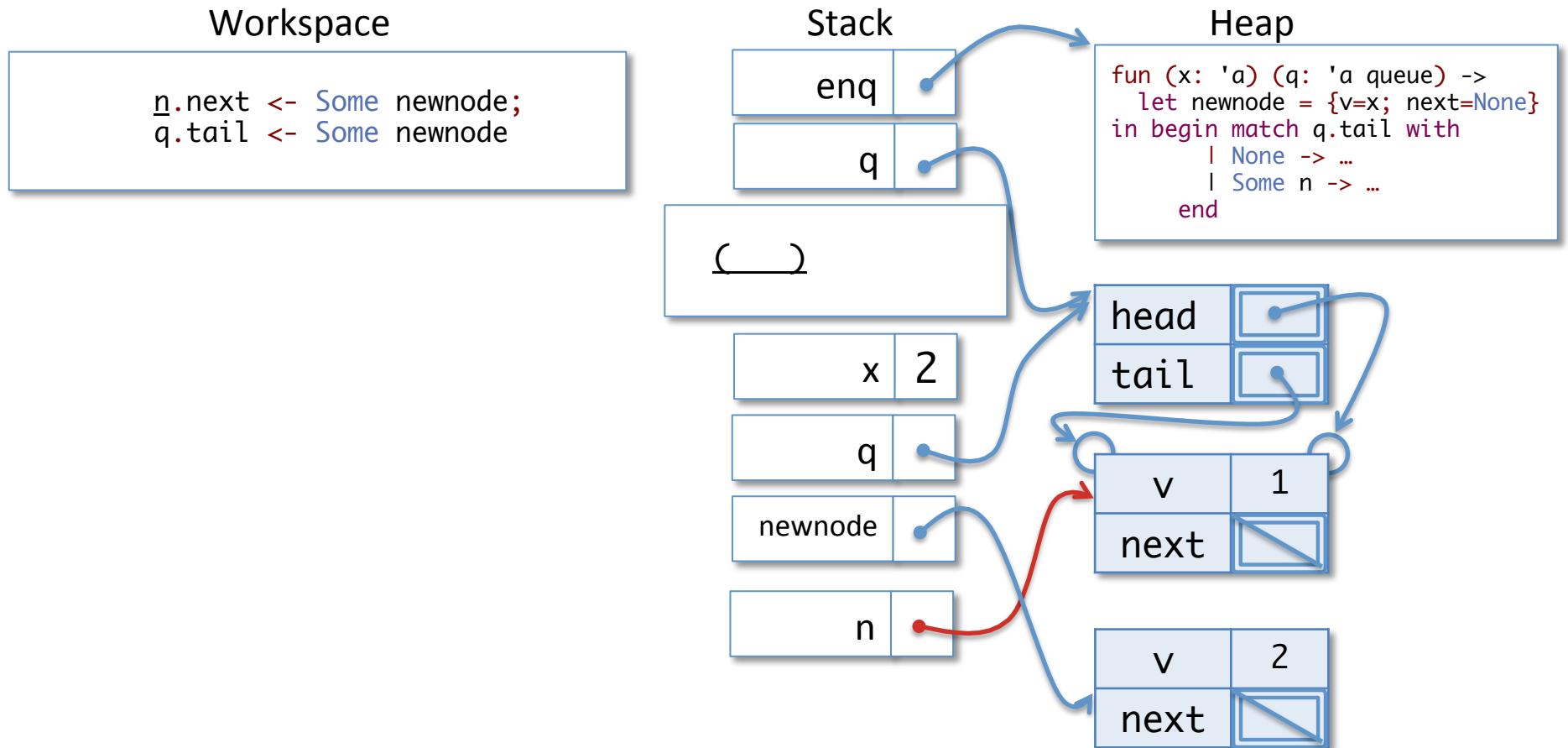
Calling Enq on a non-empty queue



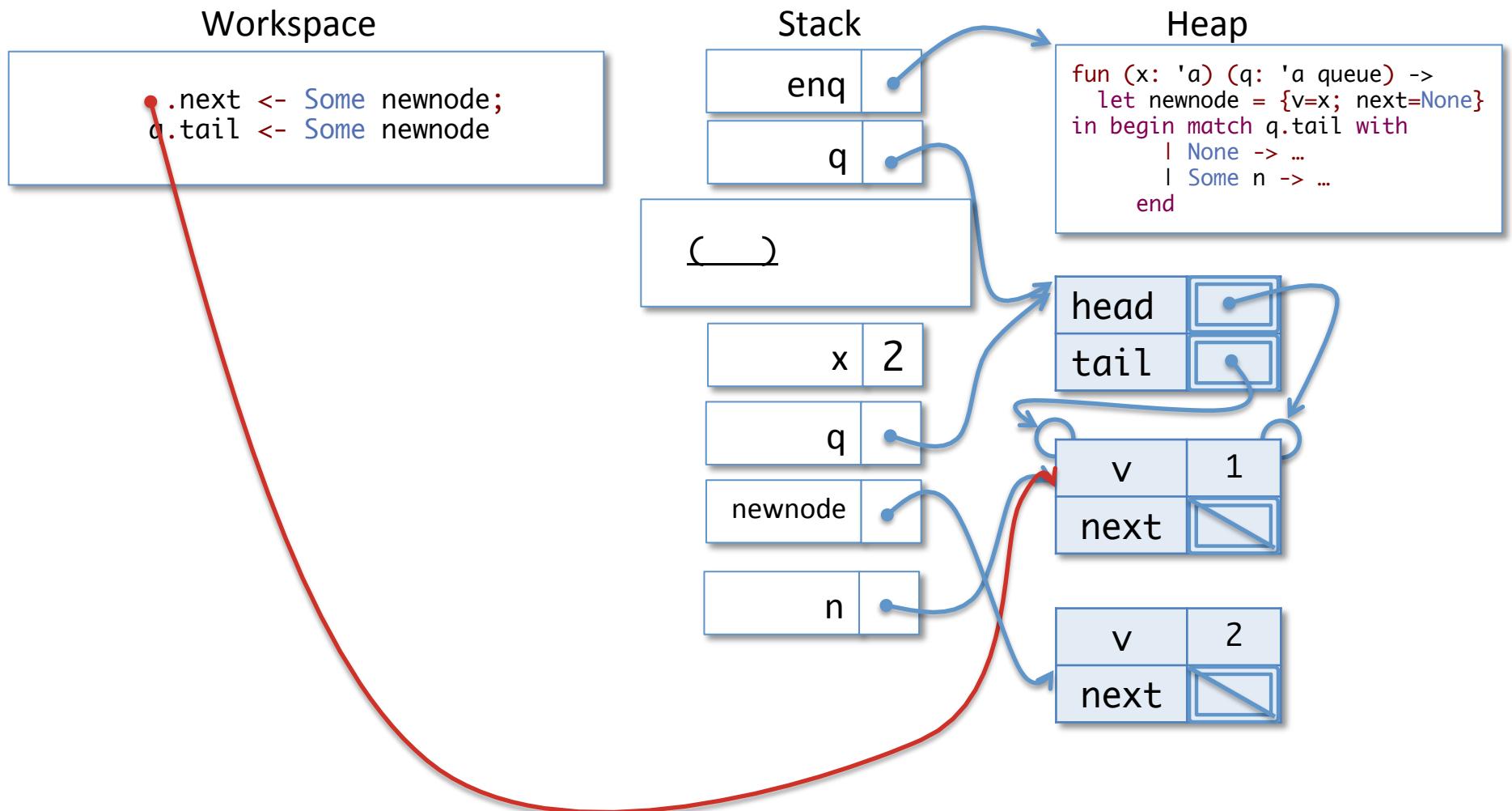
Calling Enq on a non-empty queue



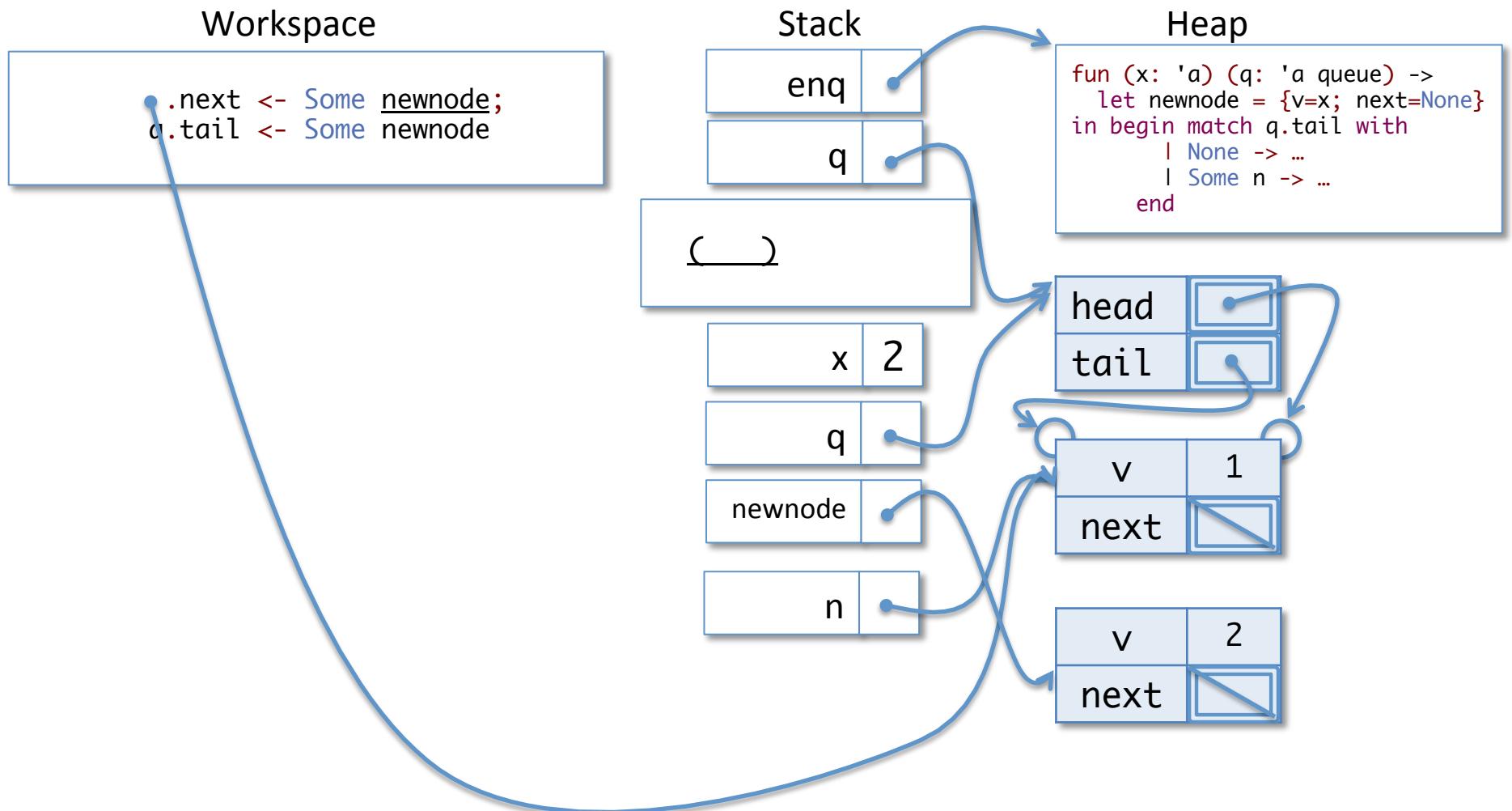
Calling Enq on a non-empty queue



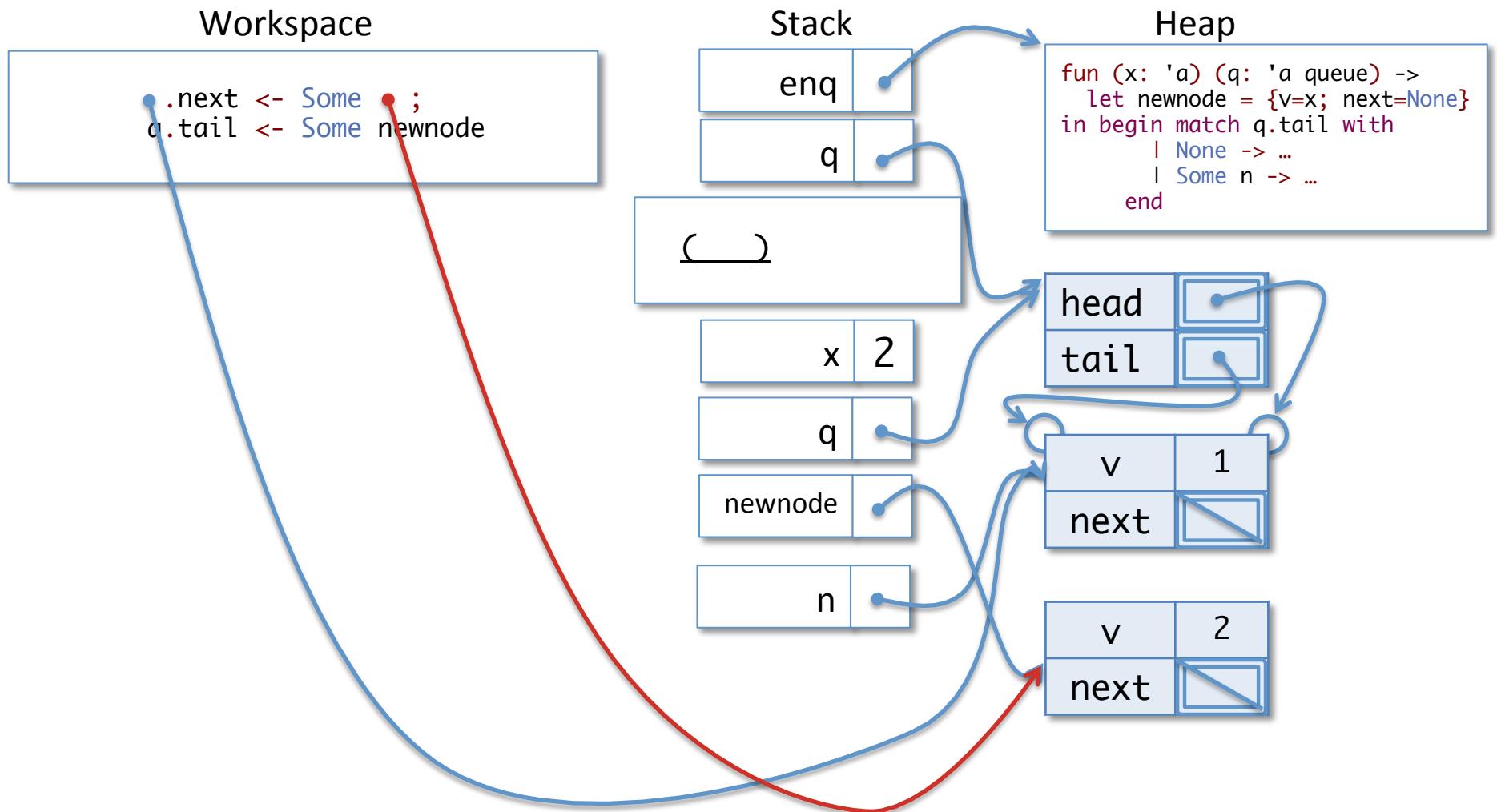
Calling Enq on a non-empty queue



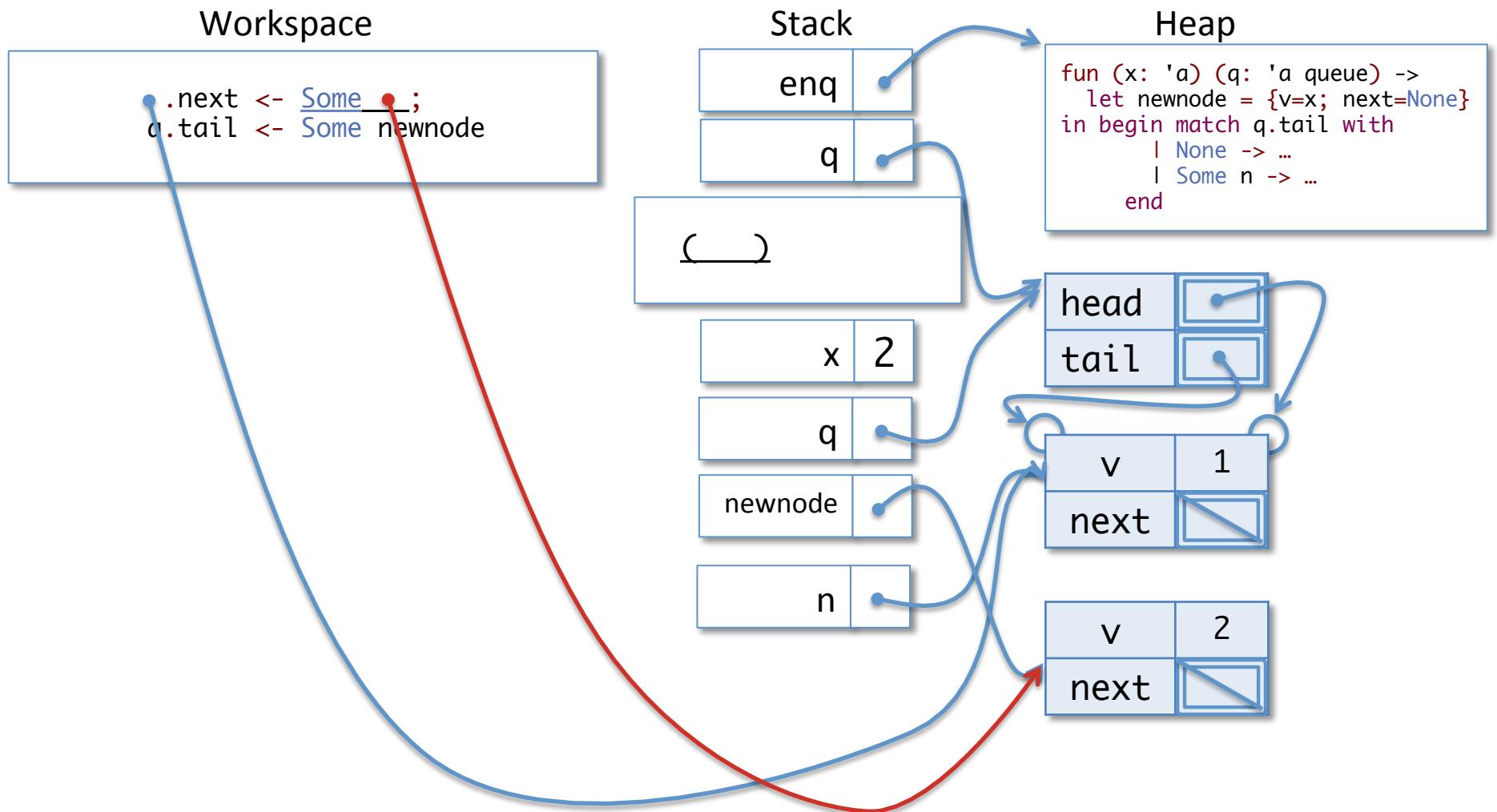
Calling Enq on a non-empty queue



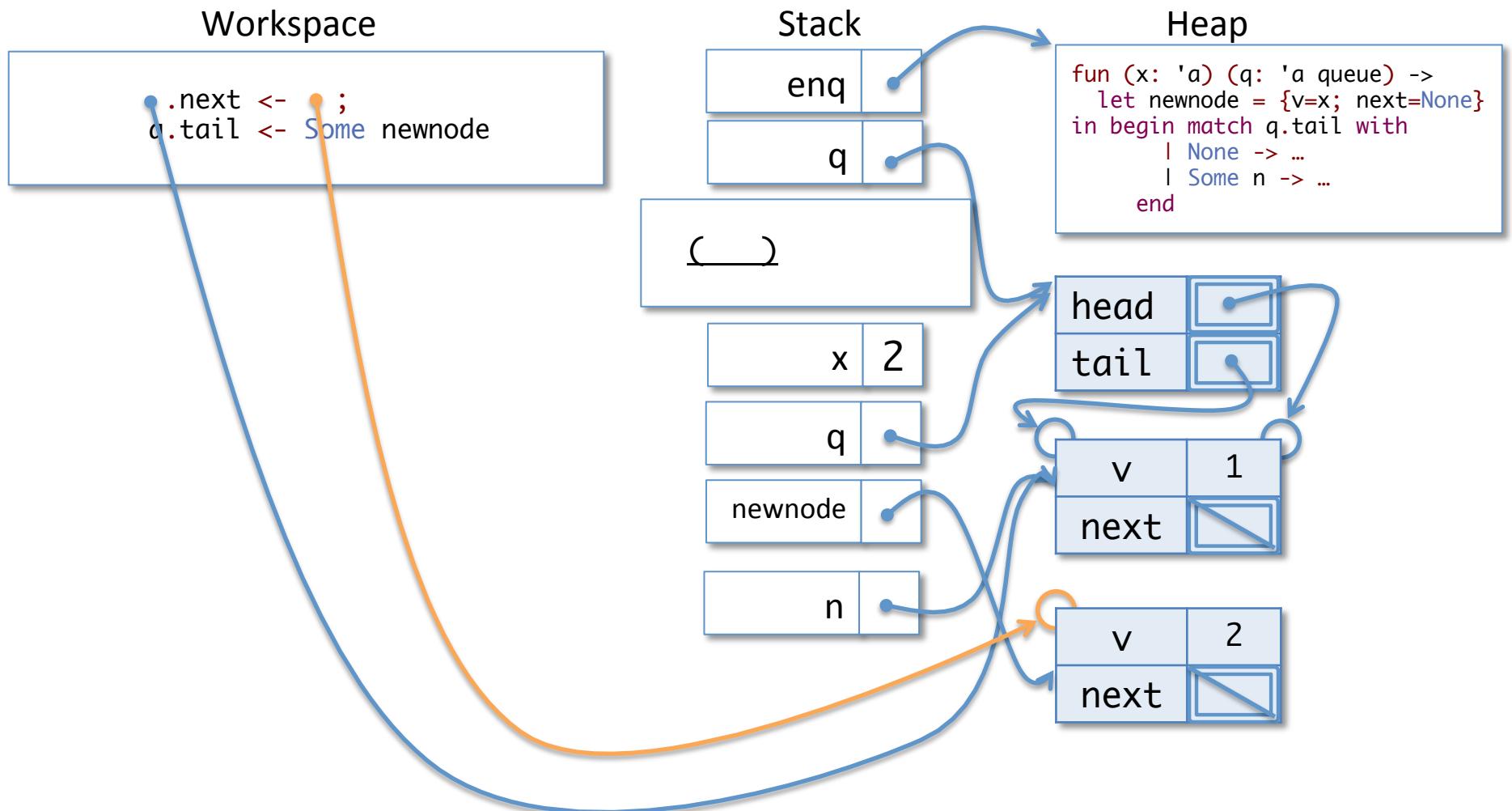
Calling Enq on a non-empty queue



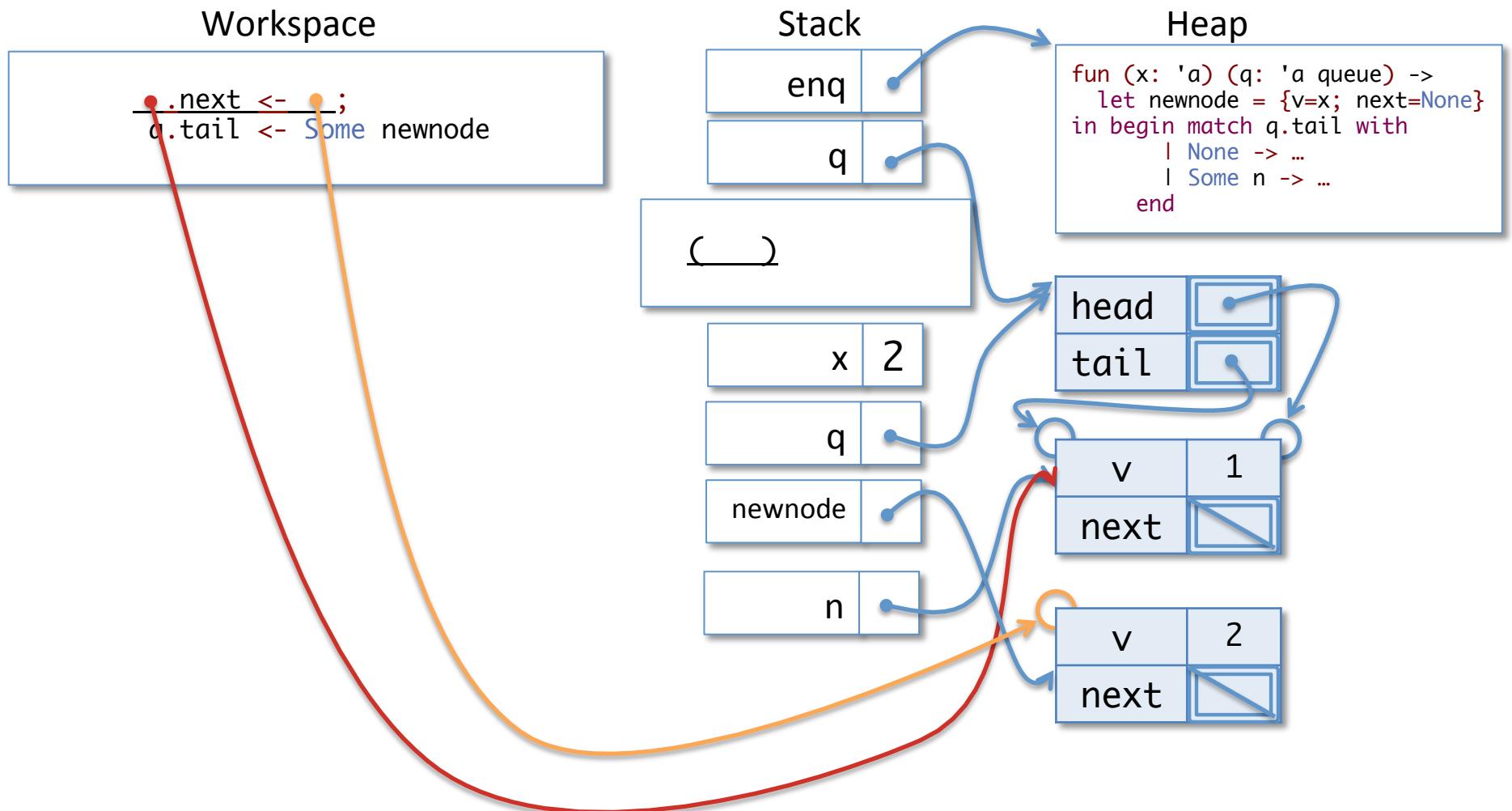
Calling Enq on a non-empty queue



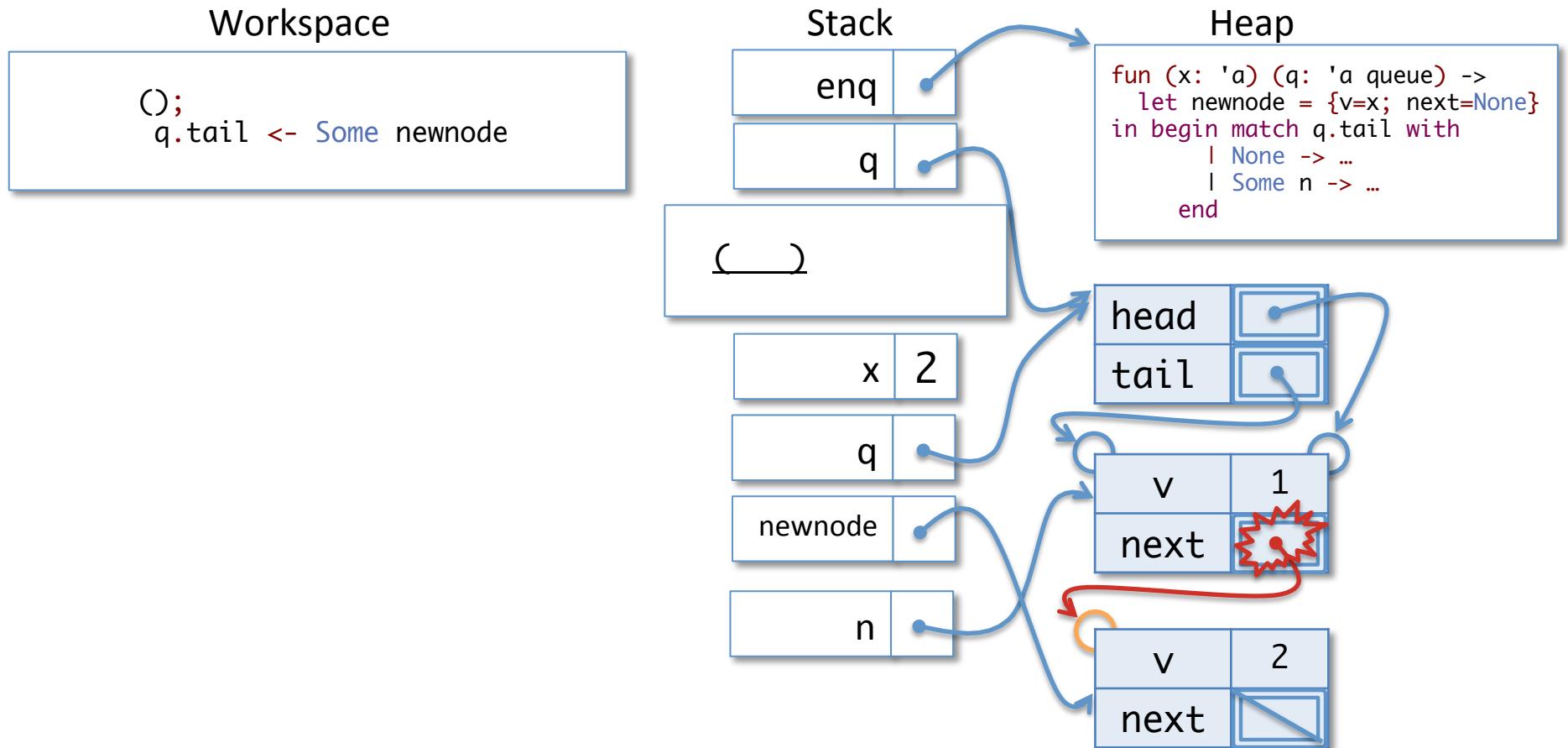
Calling Enq on a non-empty queue



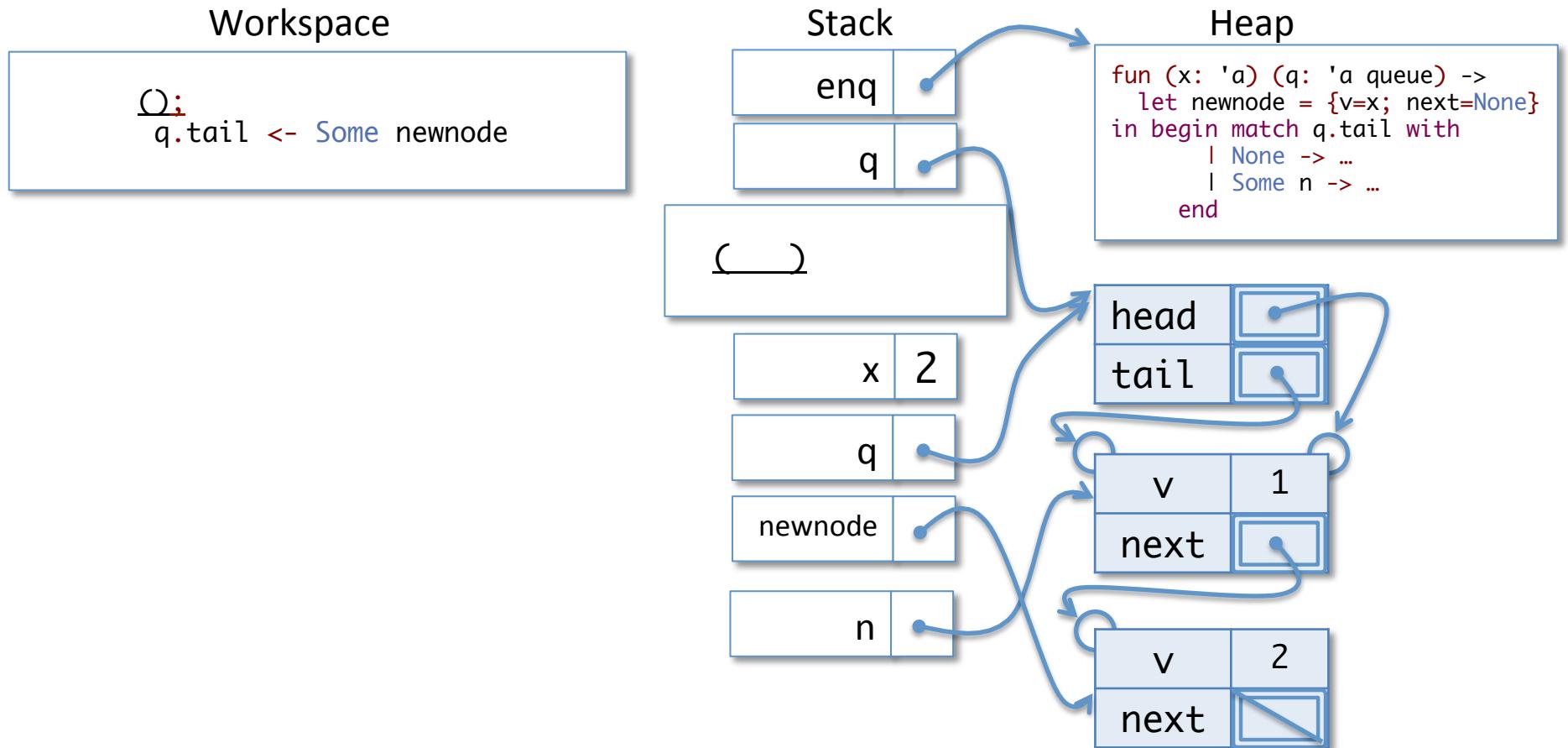
Calling Enq on a non-empty queue



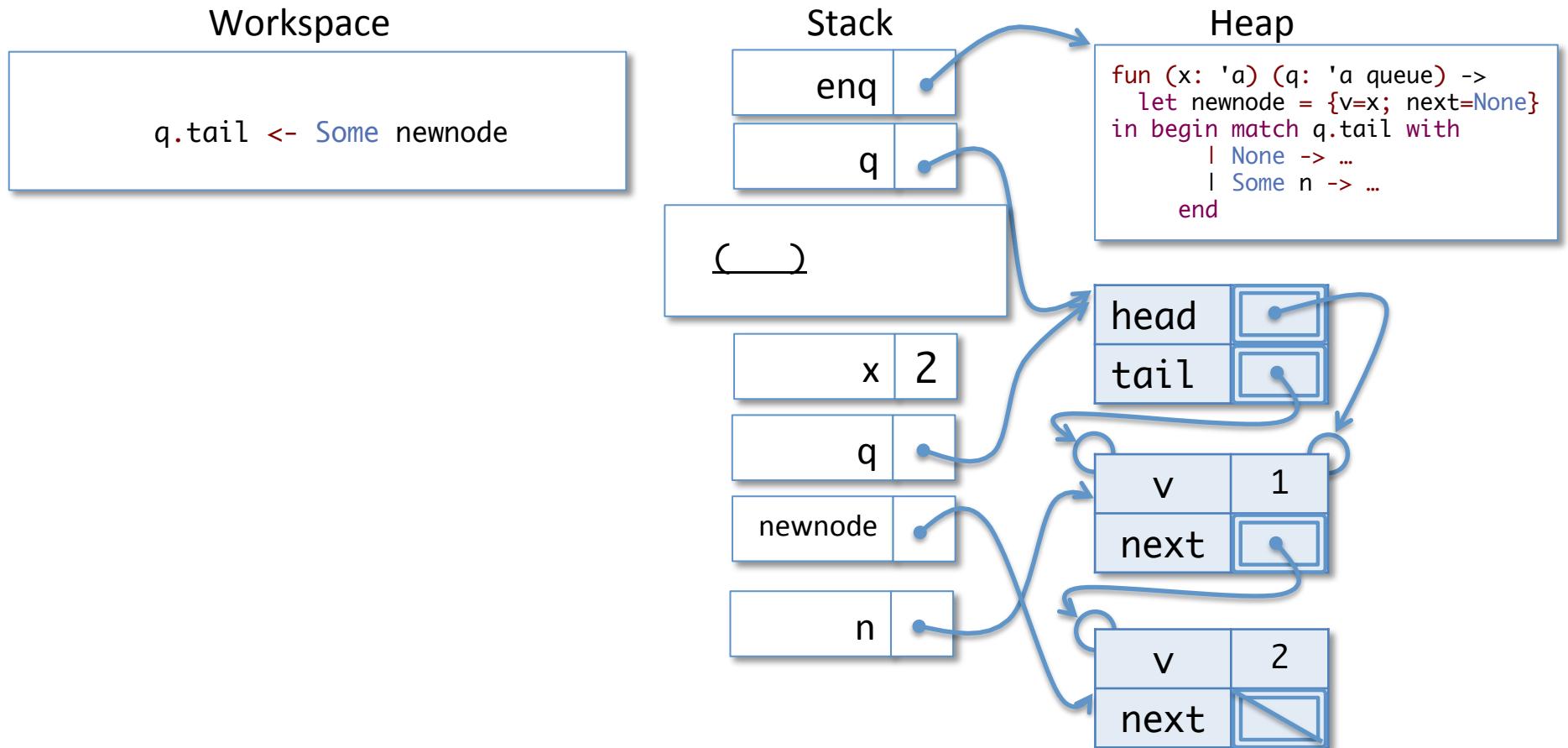
Calling Enq on a non-empty queue



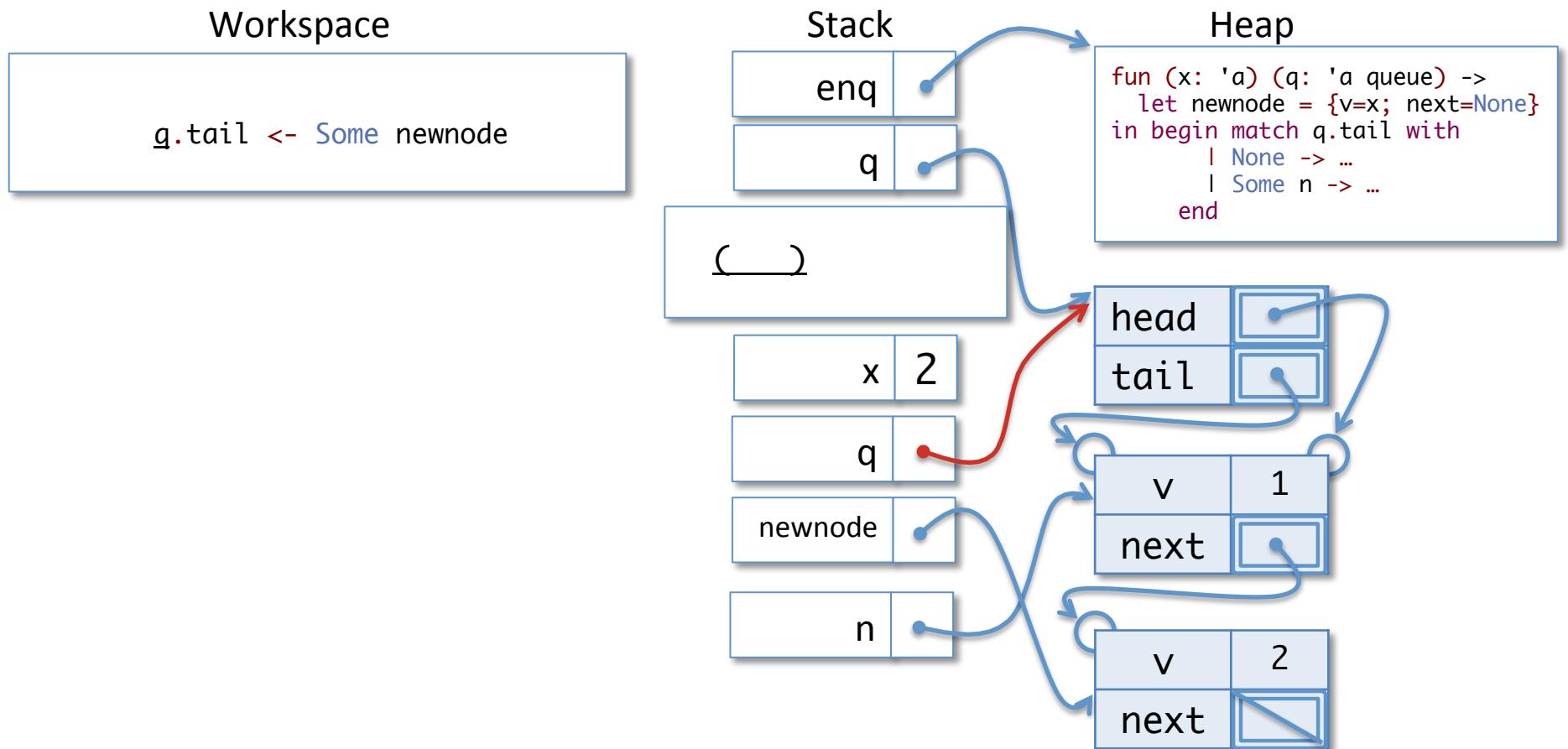
Calling Enq on a non-empty queue



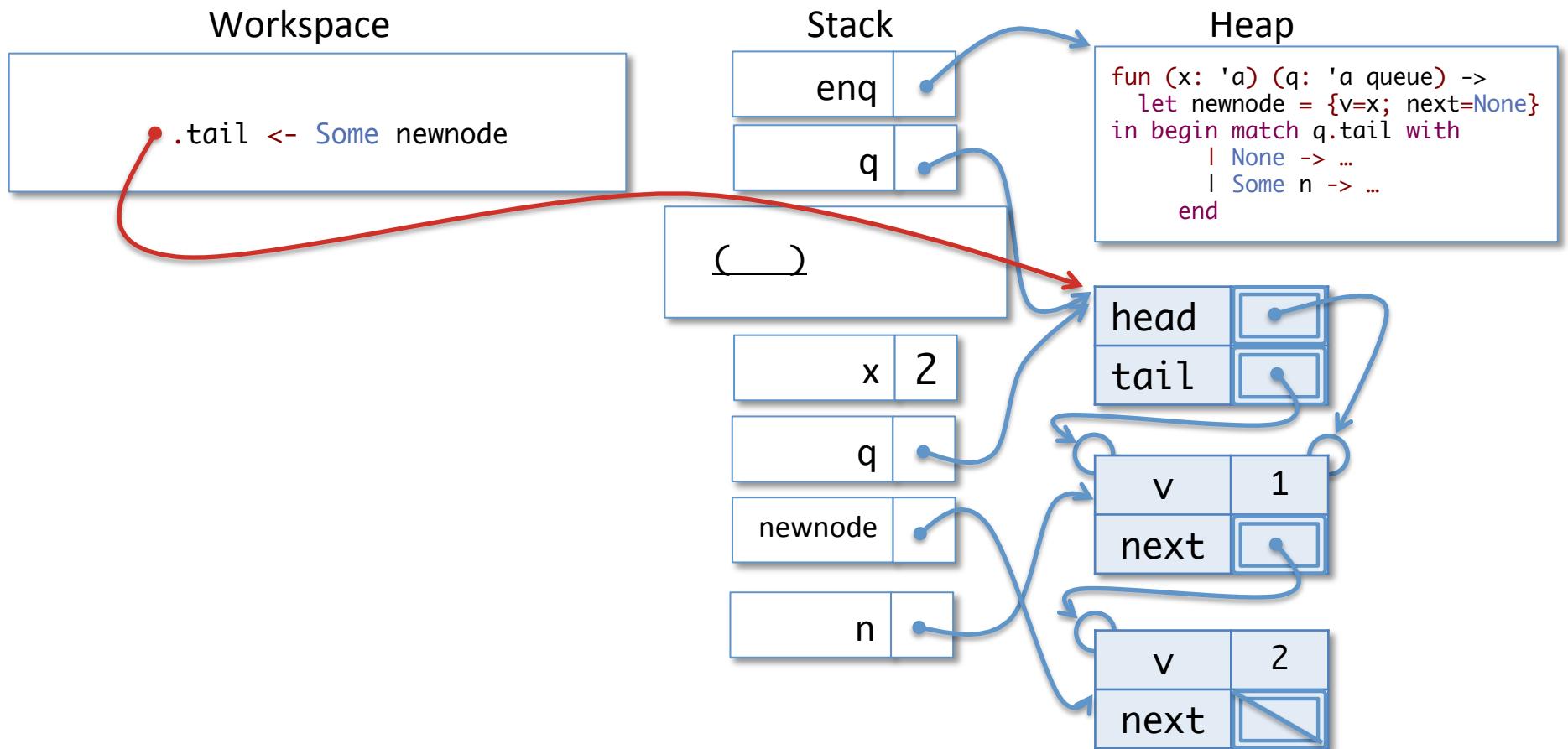
Calling Enq on a non-empty queue



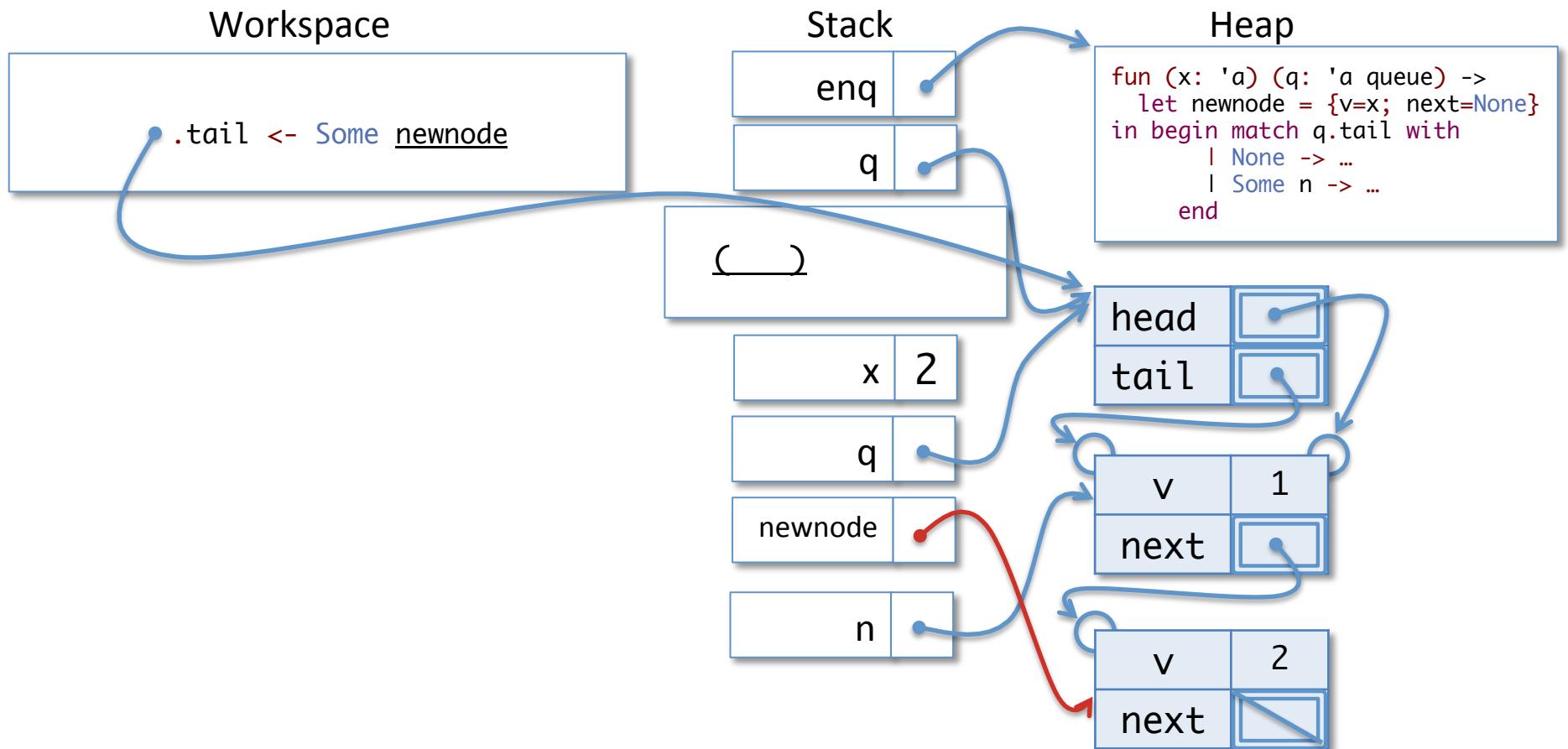
Calling Enq on a non-empty queue



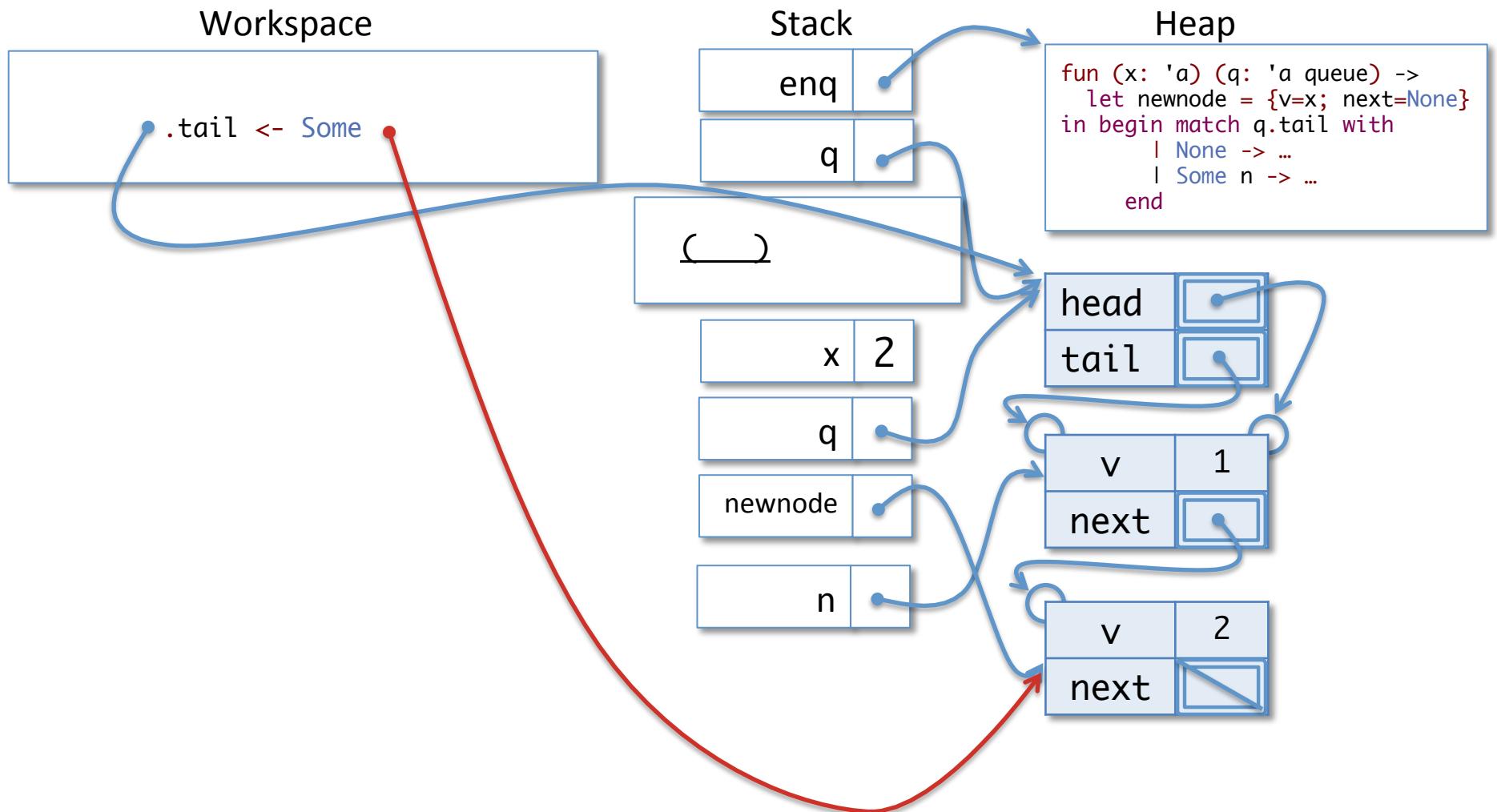
Calling Enq on a non-empty queue



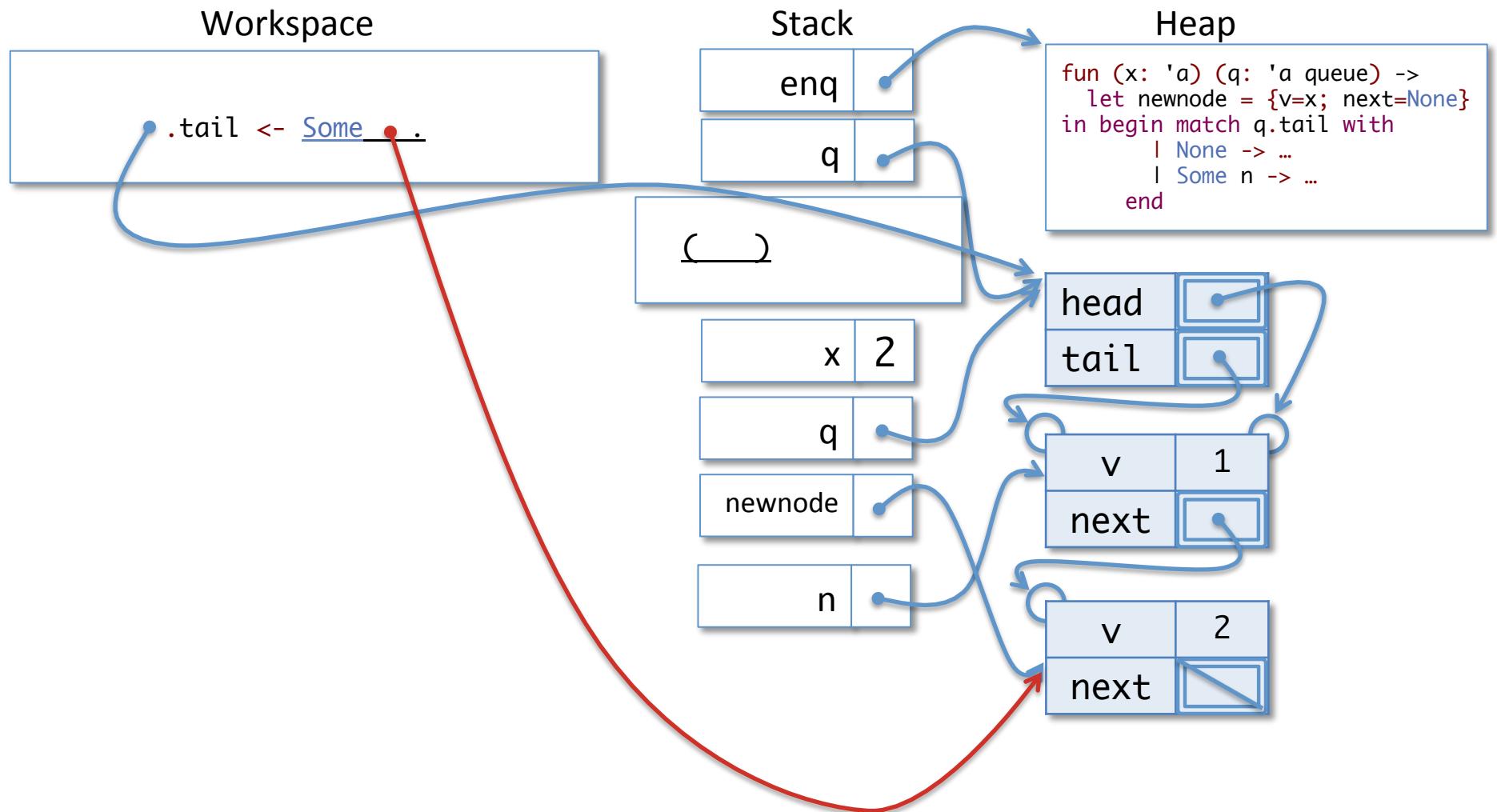
Calling Enq on a non-empty queue



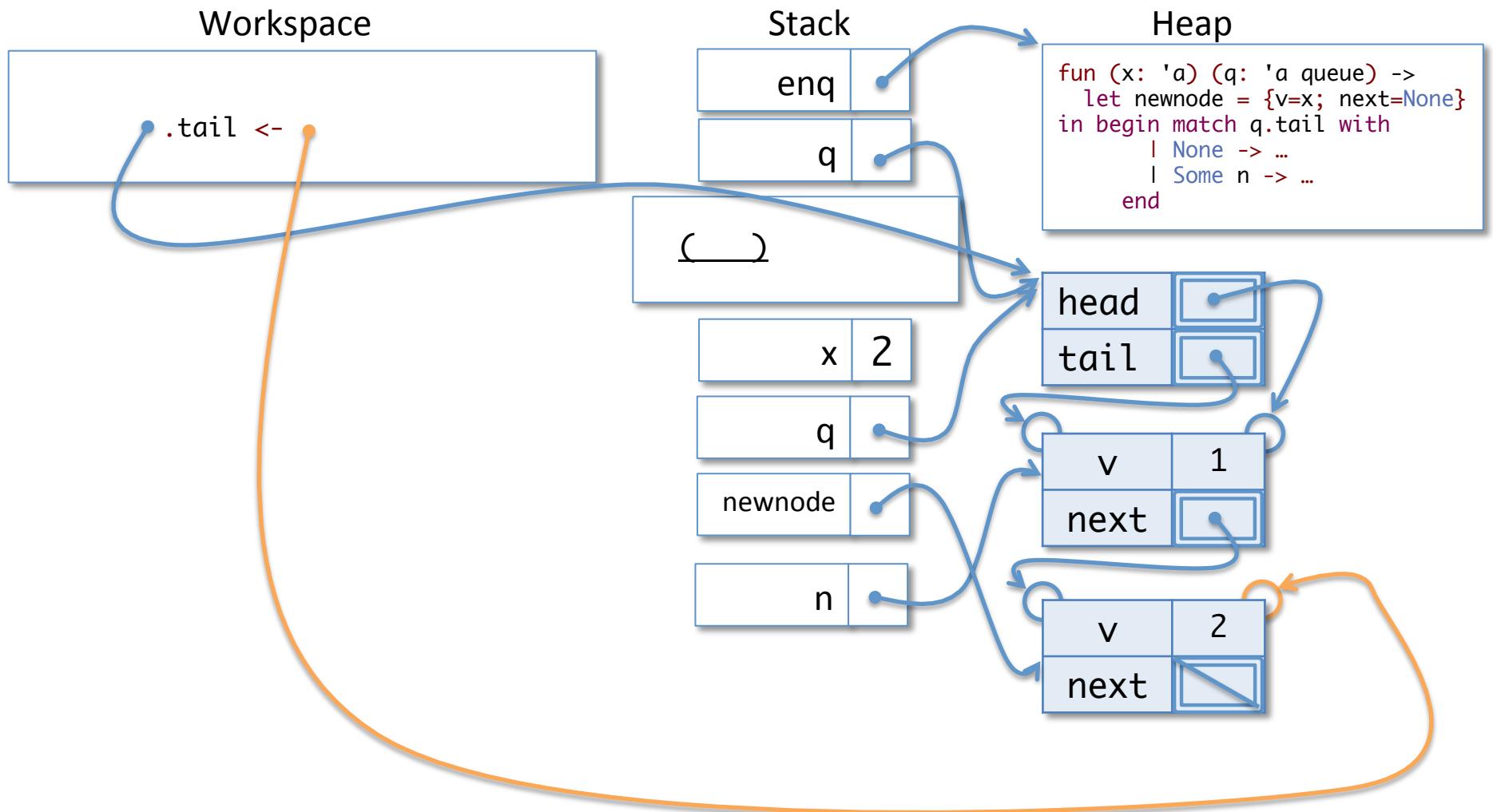
Calling Enq on a non-empty queue



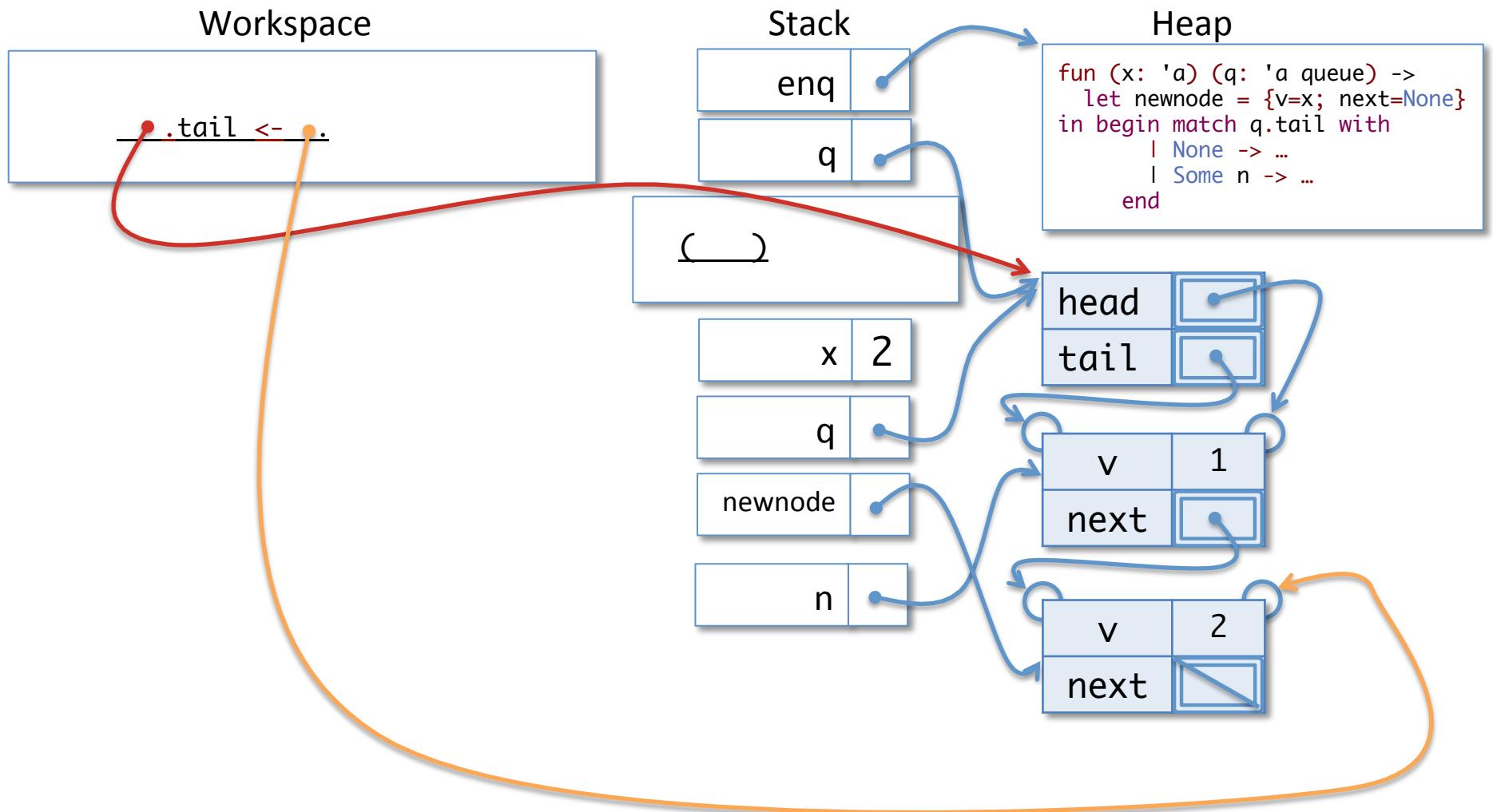
Calling Enq on a non-empty queue



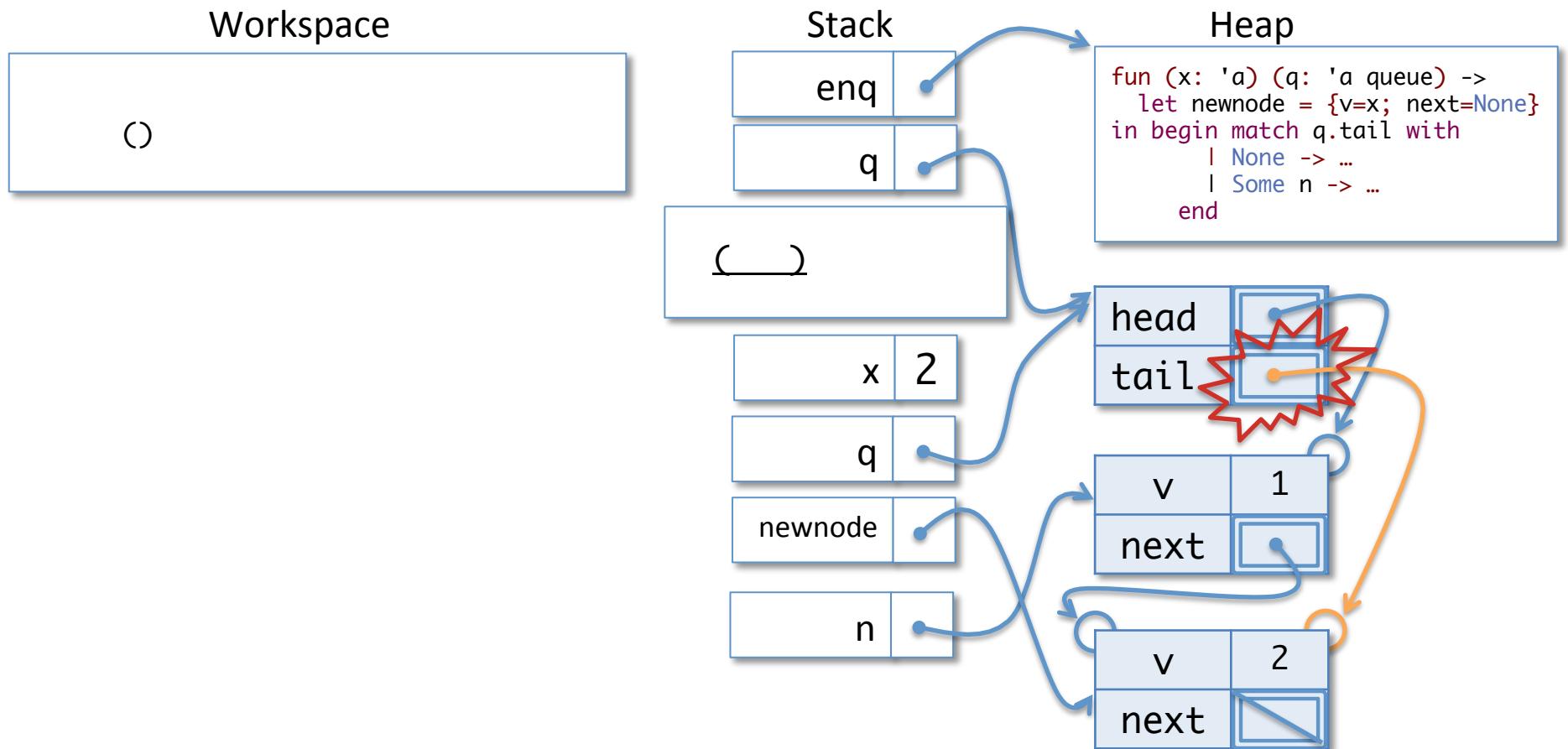
Calling Enq on a non-empty queue



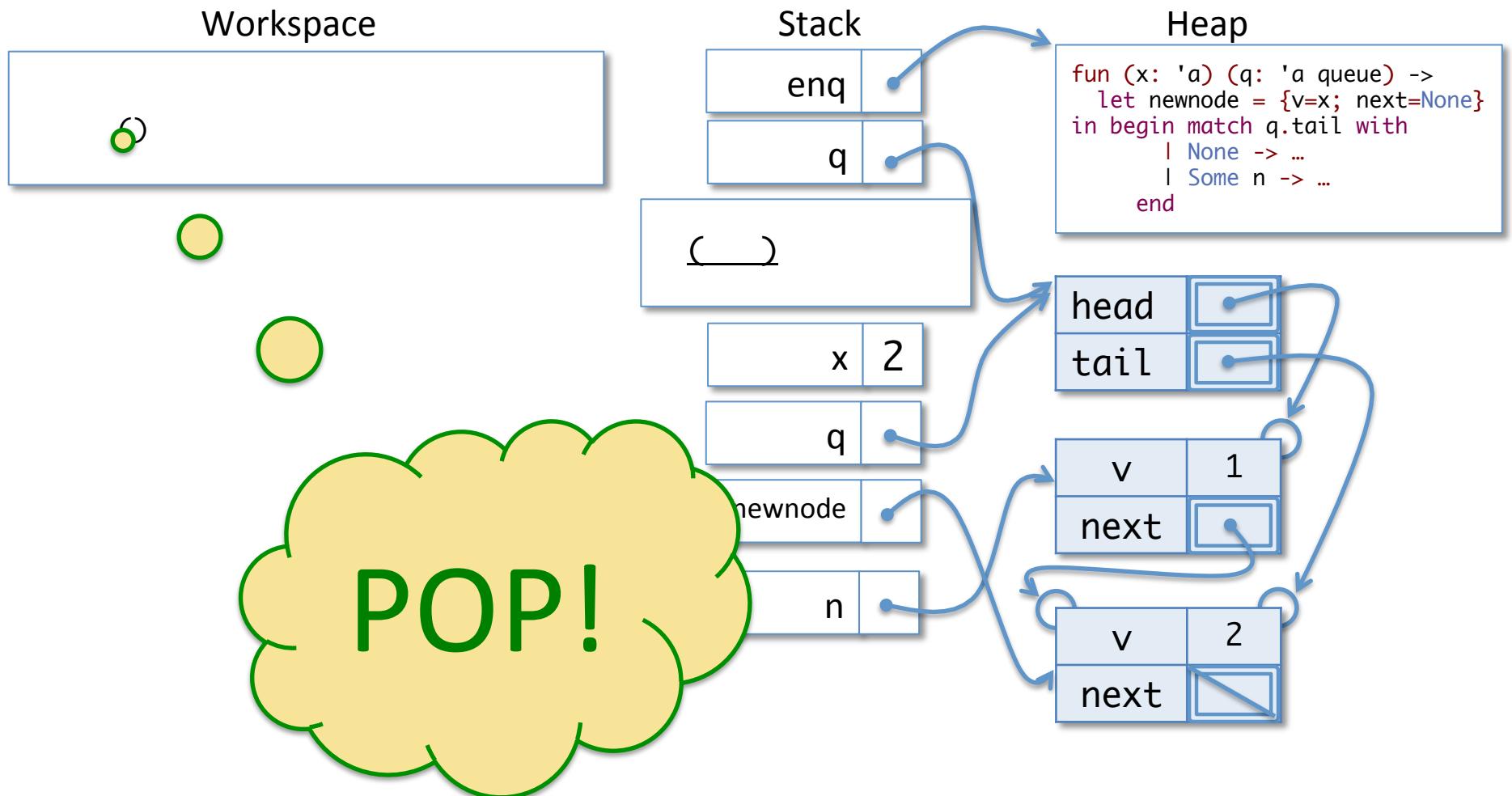
Calling Enq on a non-empty queue



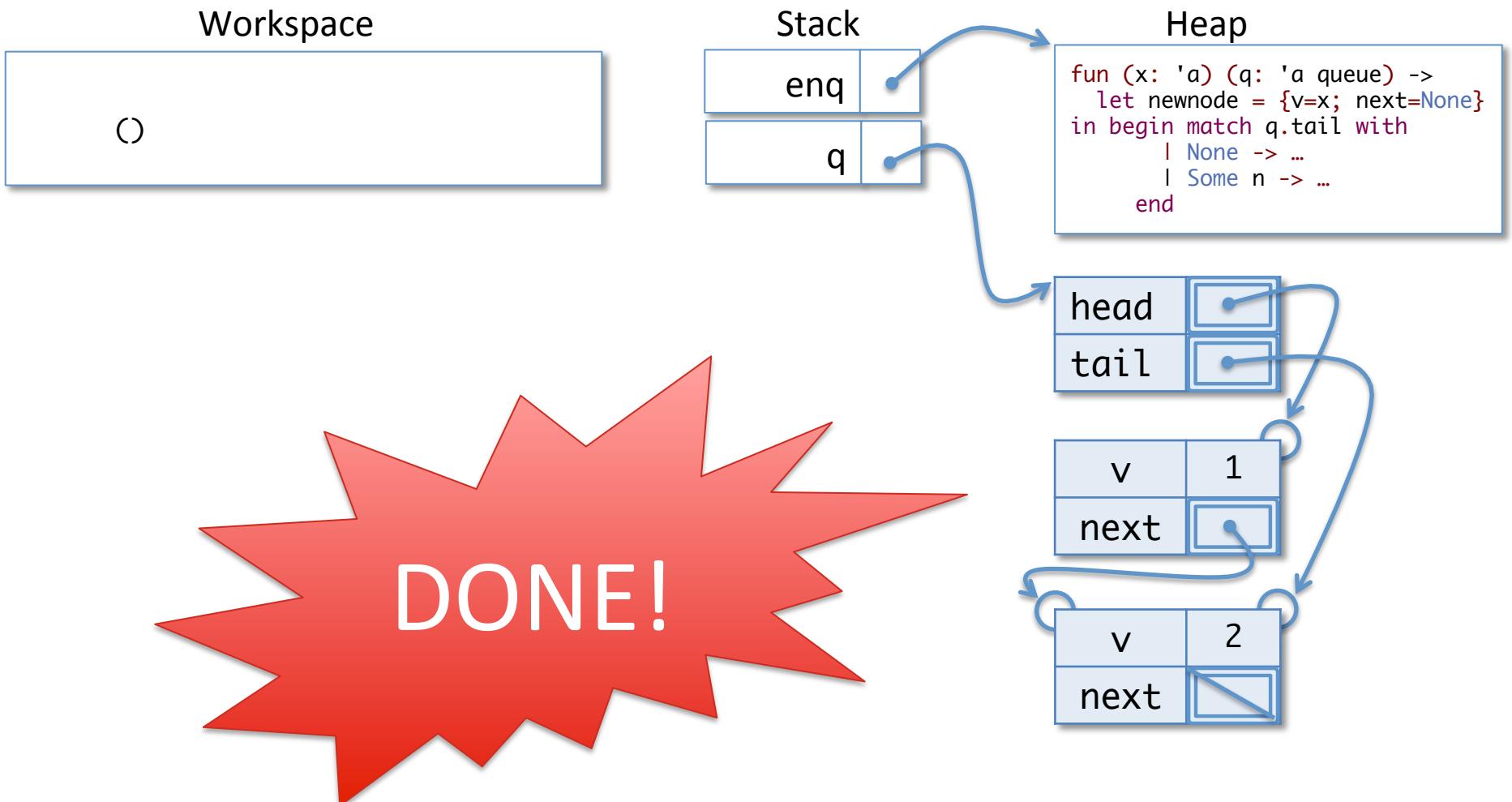
Calling Enq on a non-empty queue



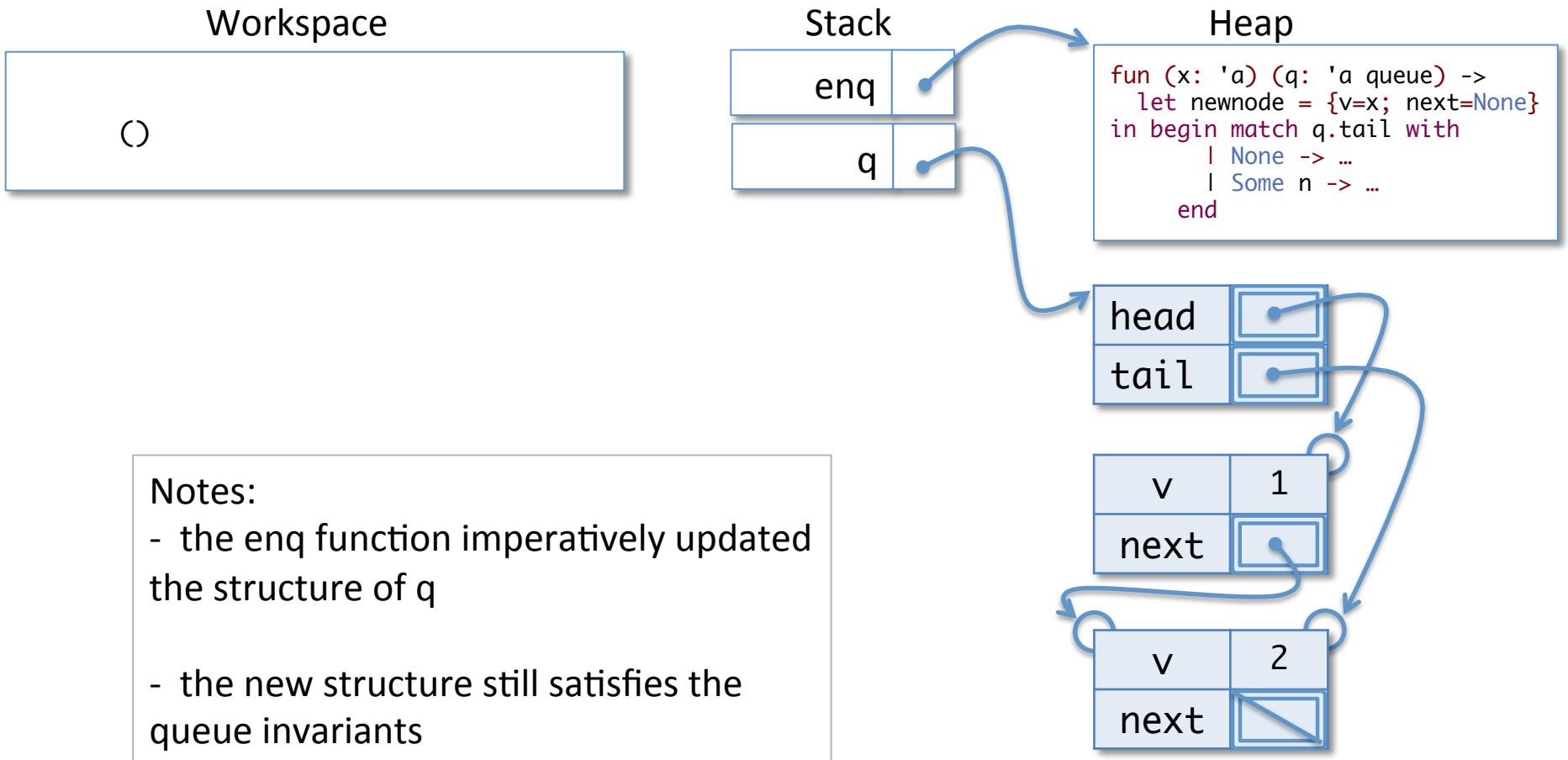
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue



Calling Enq on a non-empty queue



Challenge problem - buggy deq

```
type 'a qnode = { v: 'a; mutable next:'a qnode option }
```

```
type 'a queue = { mutable head : 'a qnode option;  
                  mutable tail : 'a qnode option }
```

(* remove element at the head of queue and return it *)

```
let deq (q: 'a queue) : 'a =  
begin match q.head with  
| None ->  
    failwith "empty queue"  
| Some n ->  
    q.head <- n.next;  
    n.v  
end
```

3.

```
let q = create () in  
enq 1 q;  
ignore (deq q);  
enq 2 q;  
2 = deq q
```

ANSWER: 3

Which test case shows the bug?

1.

```
let q = create () in  
enq 1 q;  
1 = deq q
```

2.

```
let q = create () in  
enq 1 q;  
enq 2 q;  
ignore (deq q);  
2 = deq q
```

4. All of them

deq

```
(* remove an element from the head of the queue *)
let deq (q: 'a queue) : 'a =
  begin match q.head with
    | None ->
        failwith "empty queue"
    | Some n ->
        q.head <- n.next;
        if n.next = None then q.tail <- None;
        n.v
  end
```

- The code for `deq` must also “patch pointers” to maintain the queue invariant:
 - The head pointer is always updated to the next element in the queue.
 - If the removed node was the last one in the queue, the tail pointer must be updated to `None`

Mutable Queues: Queue Length

working with singly linked data structures

Queue Length

- Suppose we want to extend the interface with a length function:

```
module type QUEUE =
sig
  (* type of the data structure *)
  type 'a queue
  ...
  (* Get the length of the queue *)
  val length : 'a queue -> int
end
```

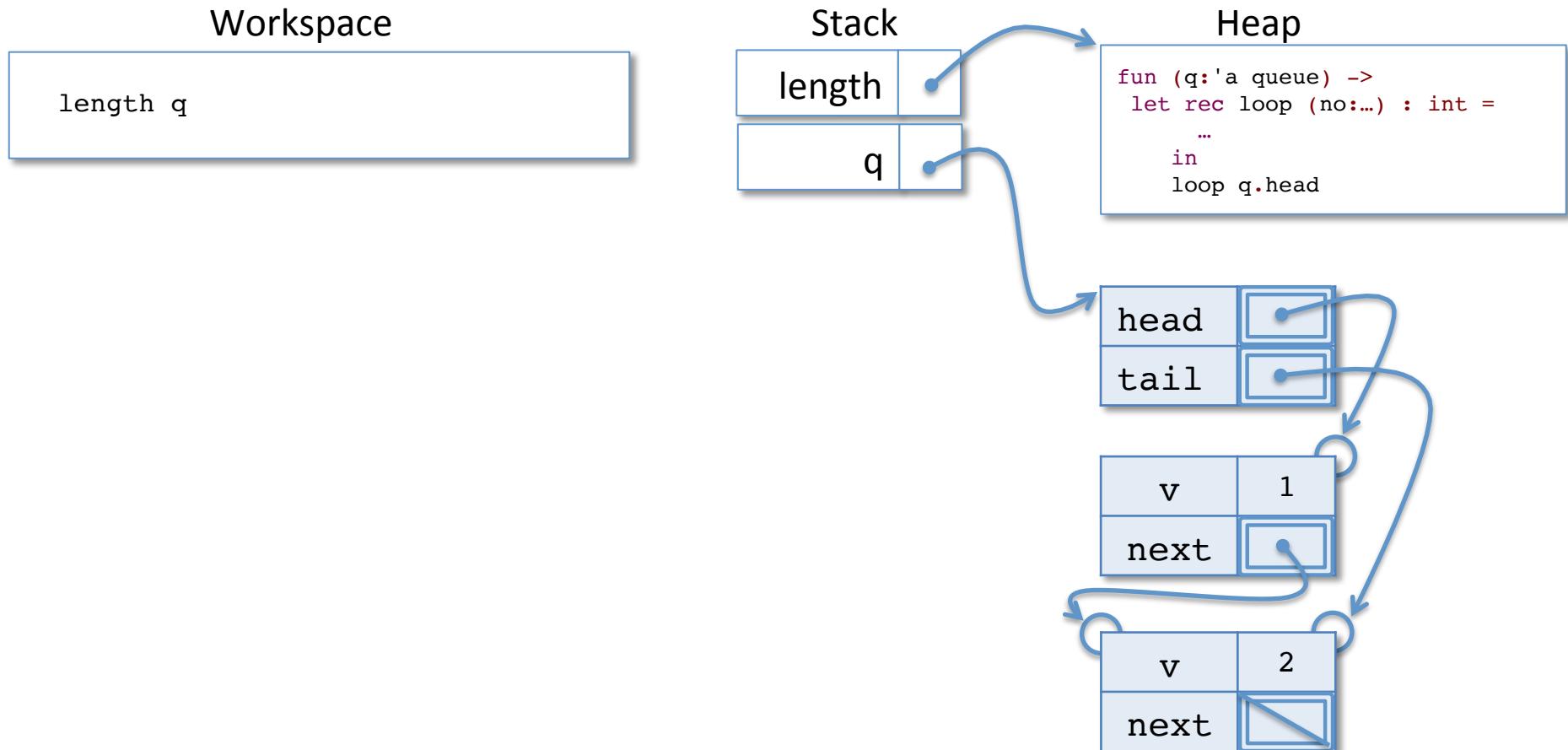
- How can we implement it?

length (recursively)

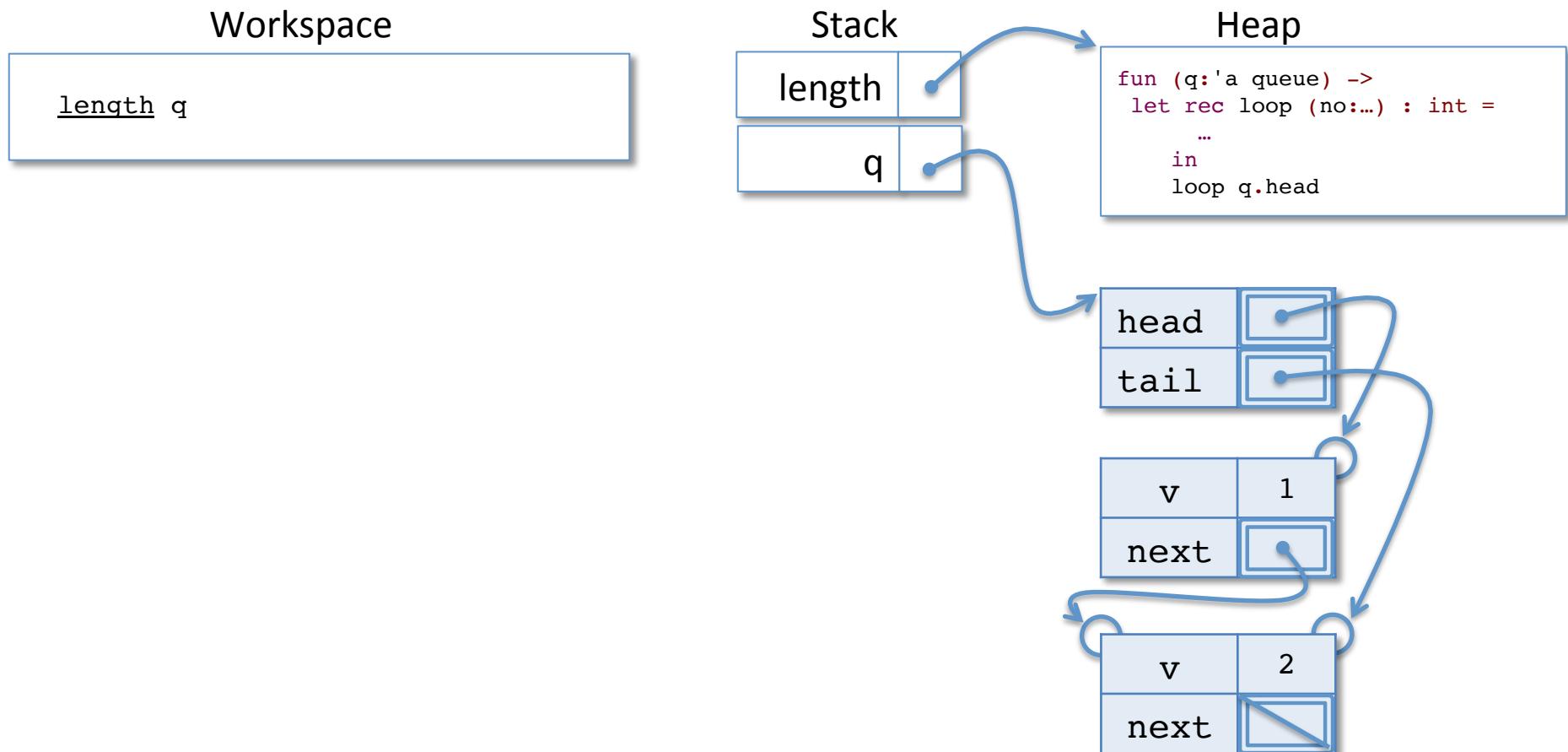
```
(* Calculate the length of the queue recursively *)
let length (q:'a queue) : int =
  let rec loop (no: 'a qnode option) : int =
    begin match no with
      | None -> 0
      | Some n -> 1 + (loop n.next)
    end
  in
  loop q.head
```

- This code for `length` uses a helper function, `loop`:
 - the correctness depends crucially on the queue invariant
 - what happens if we pass in a bogus `q` that is cyclic?
- The height of the ASM stack is proportional to the length of the queue
 - That seems inefficient... why should it take so much space?

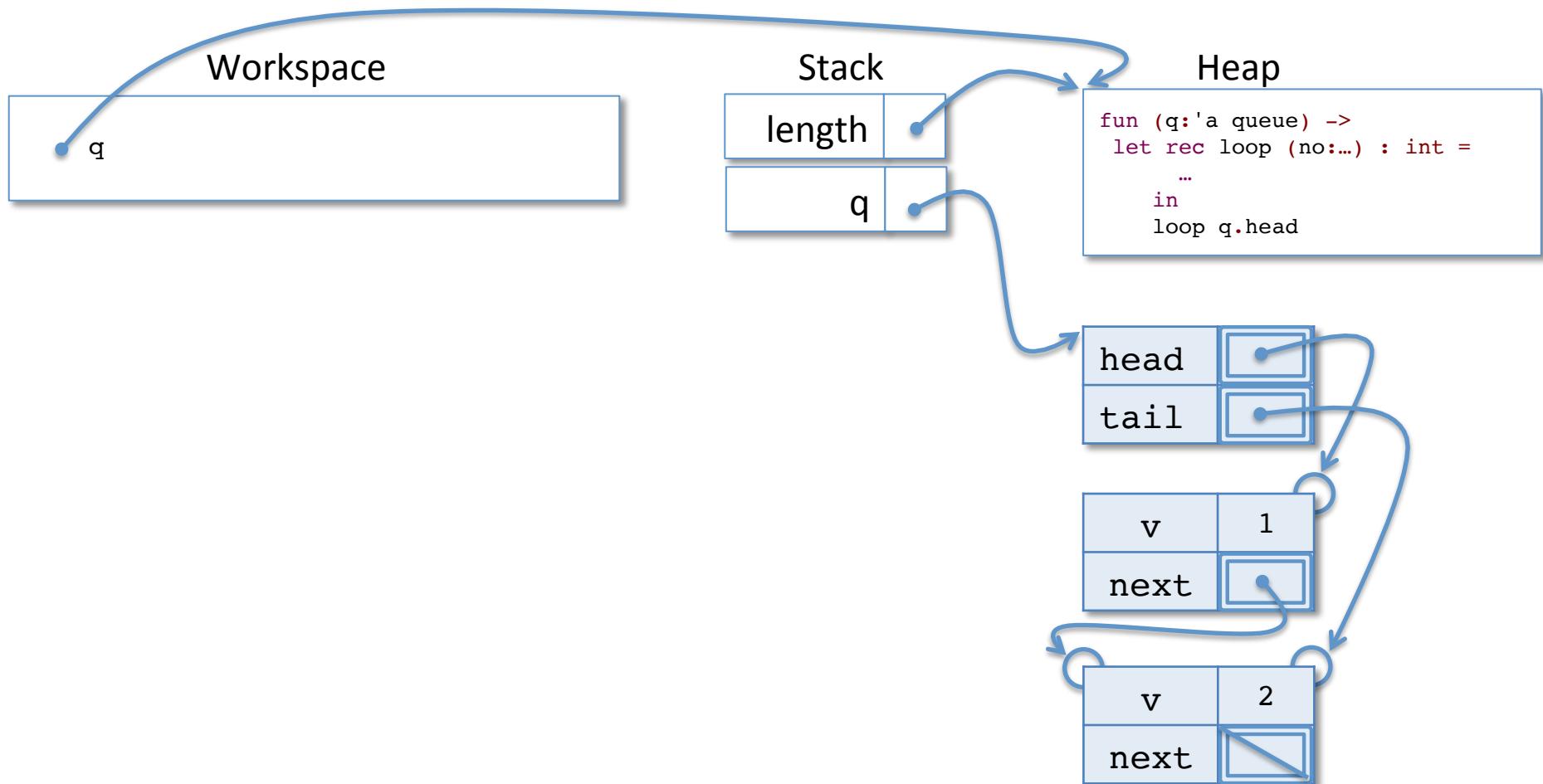
Evaluating length



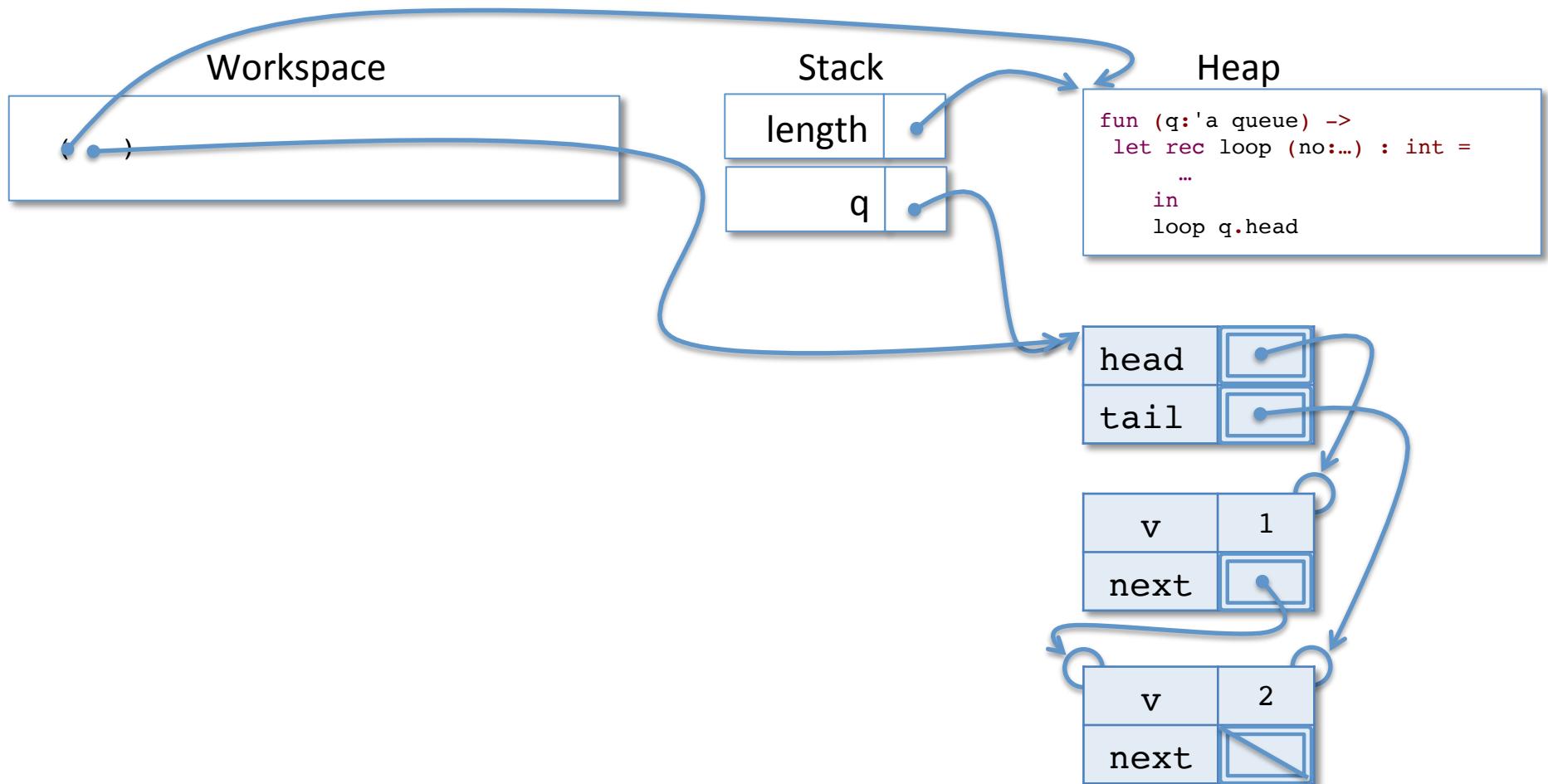
Evaluating length



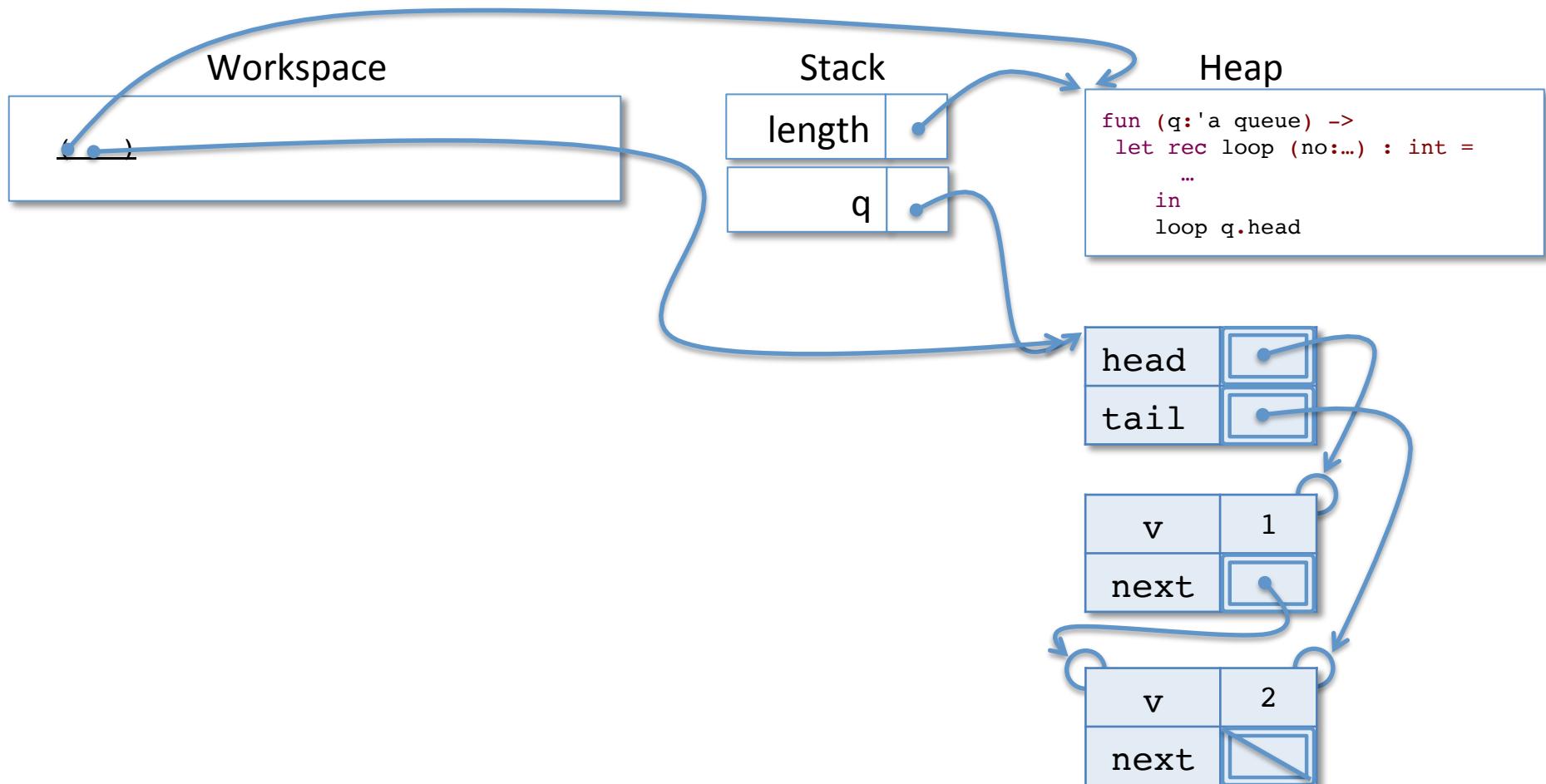
Evaluating length



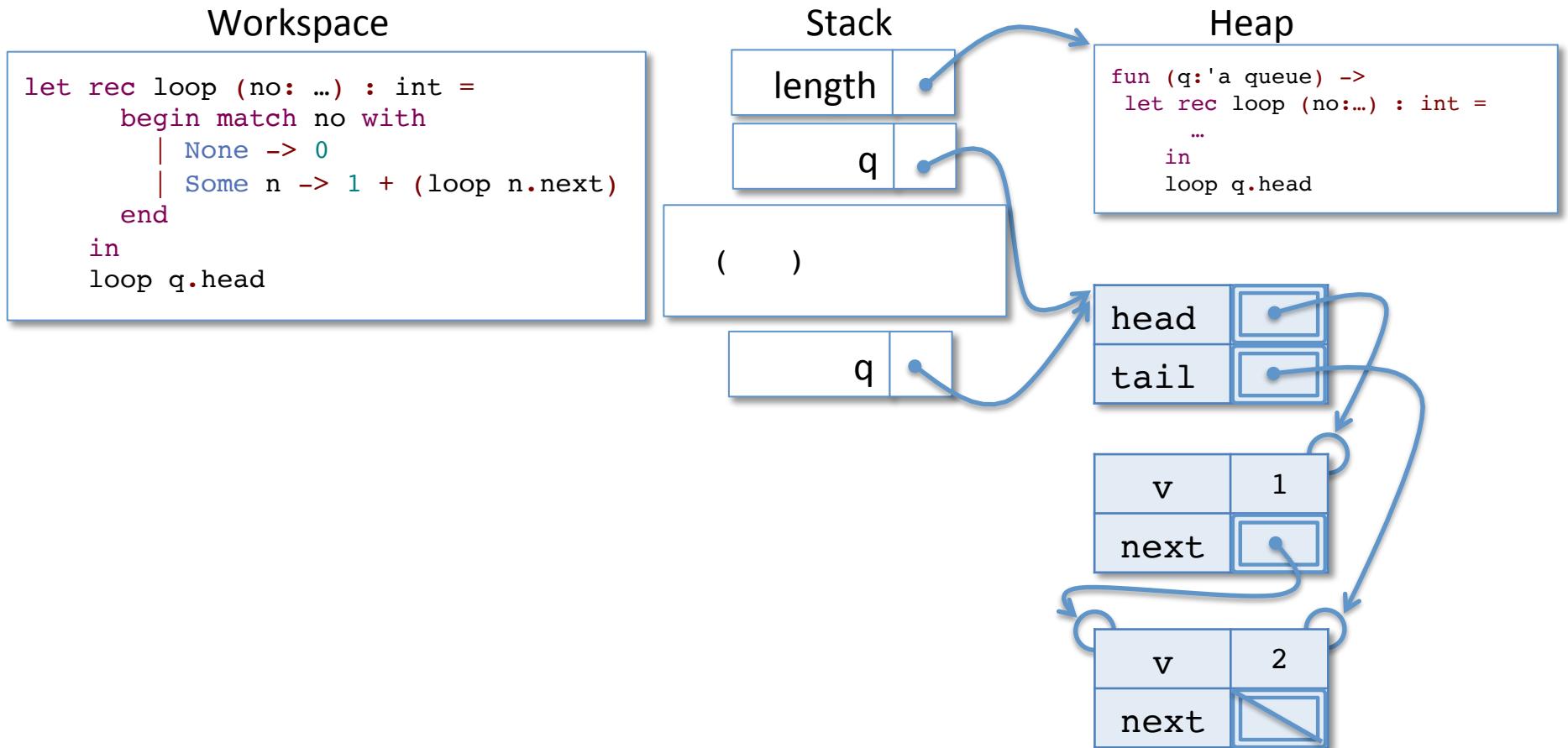
Evaluating length



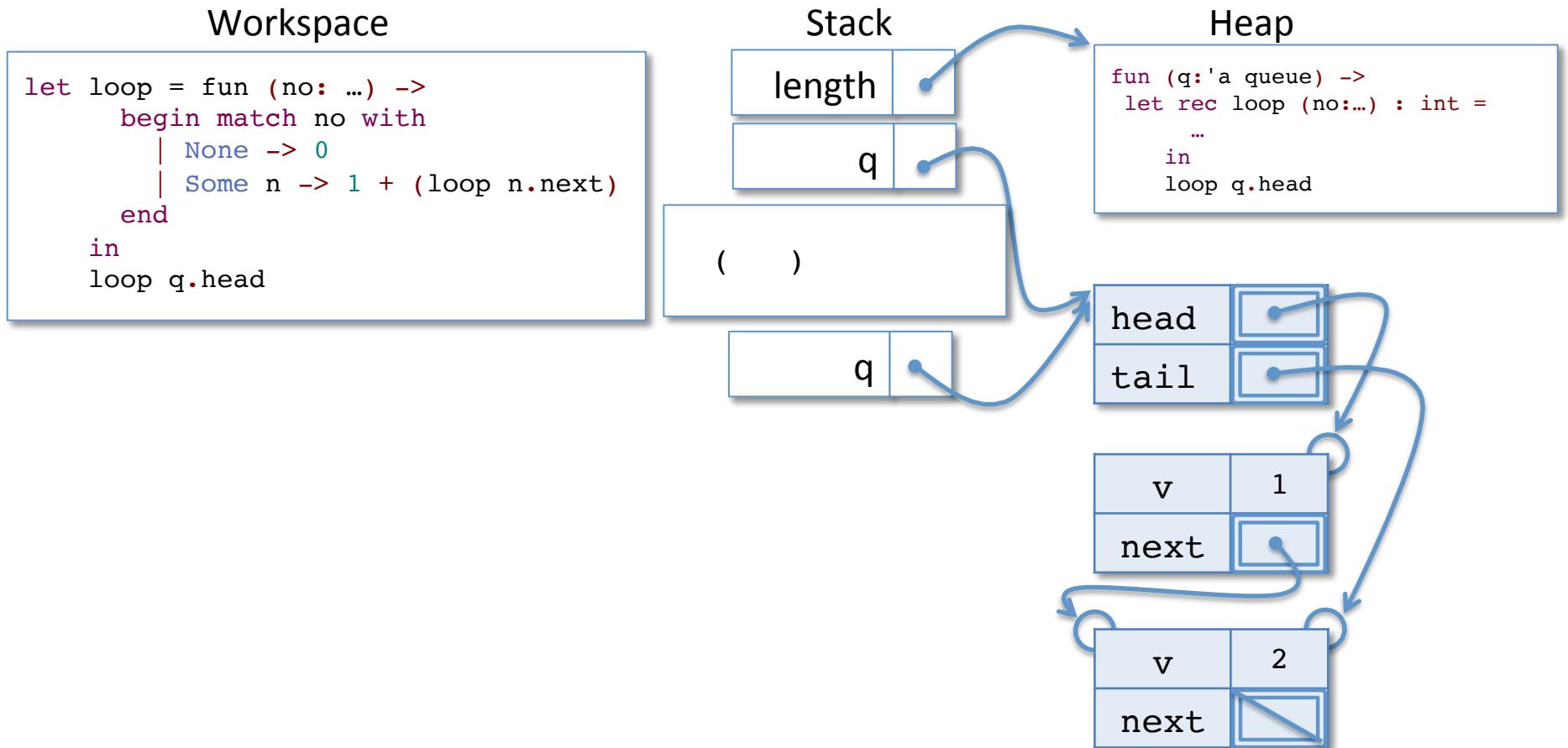
Evaluating length



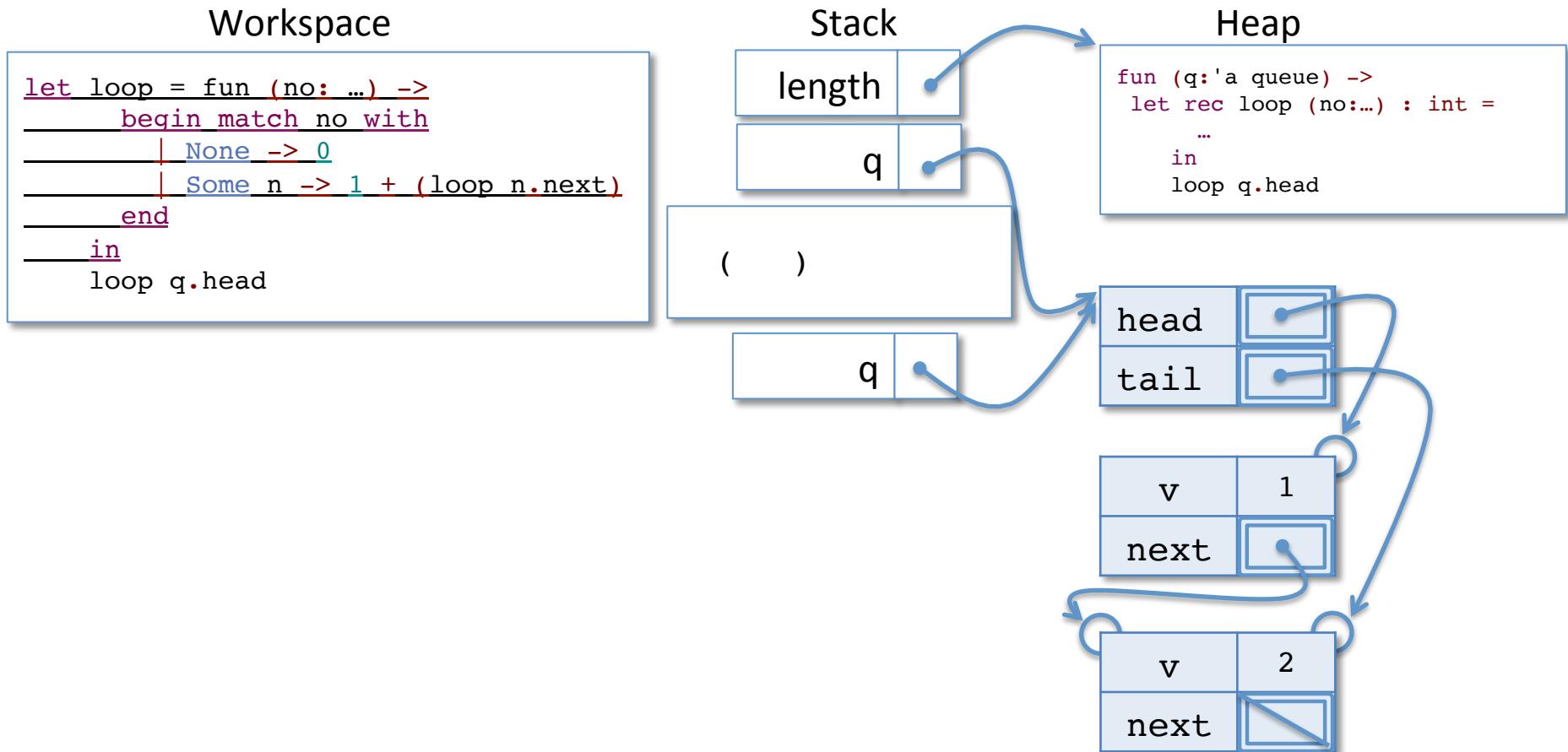
Evaluating length



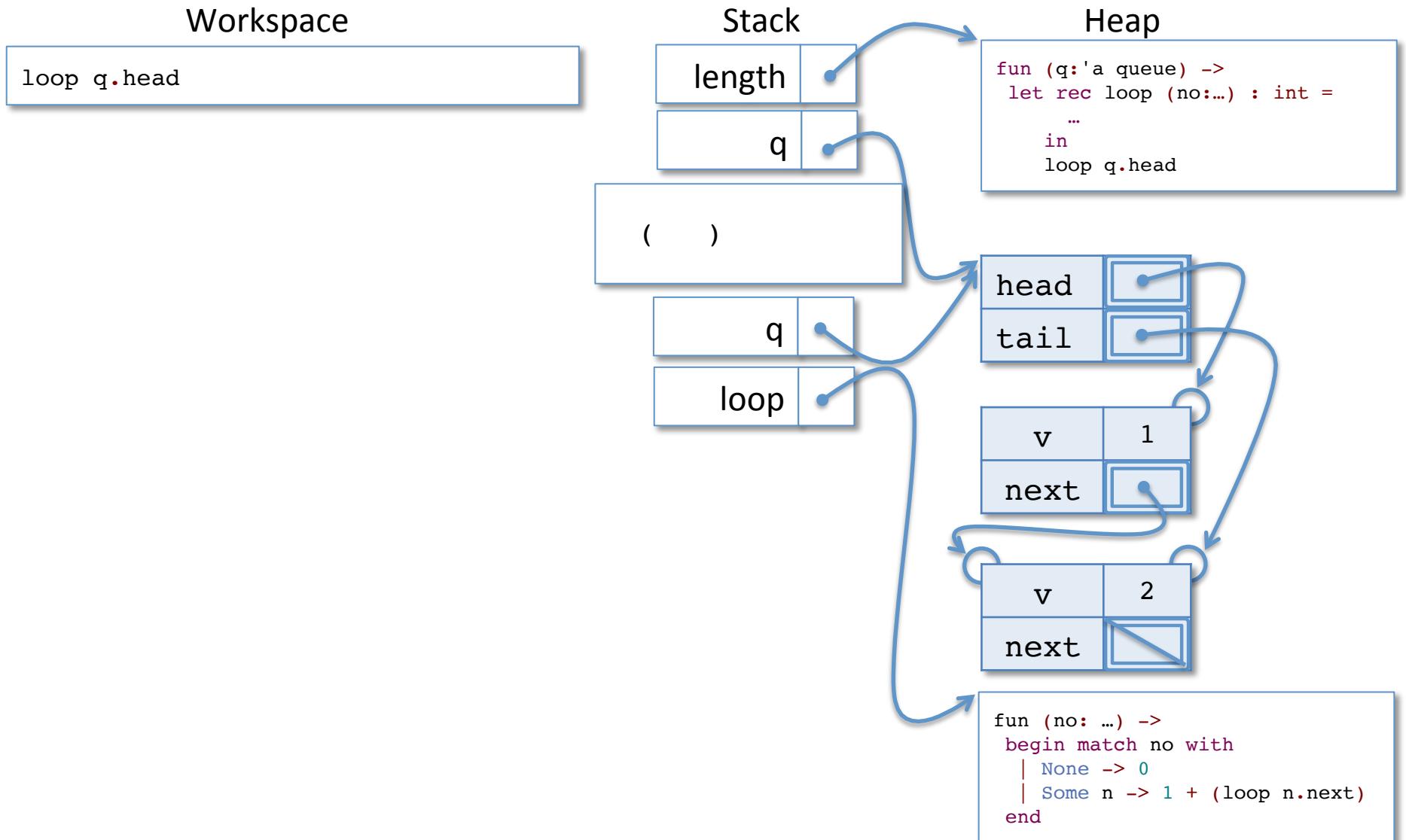
Evaluating length



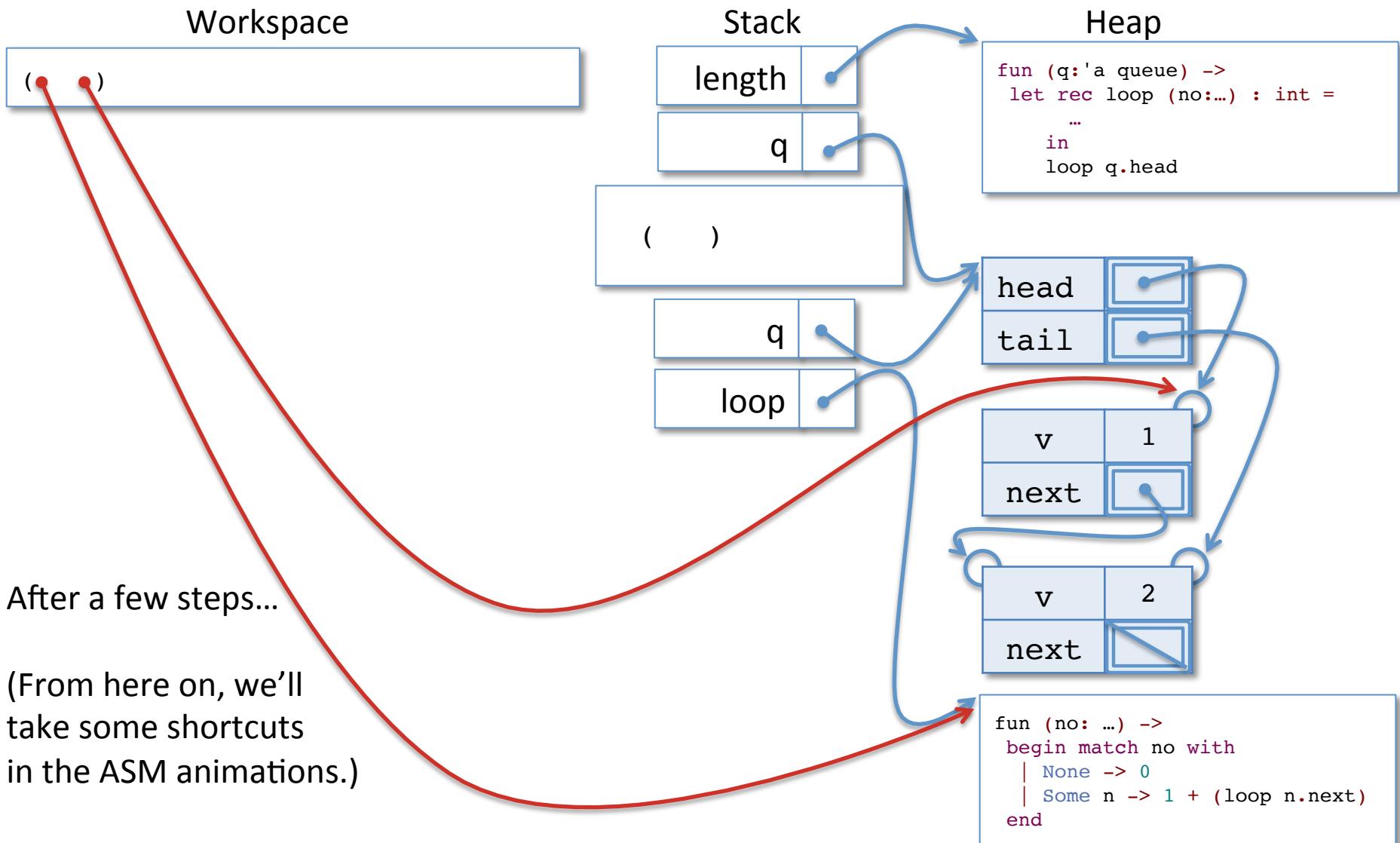
Evaluating length



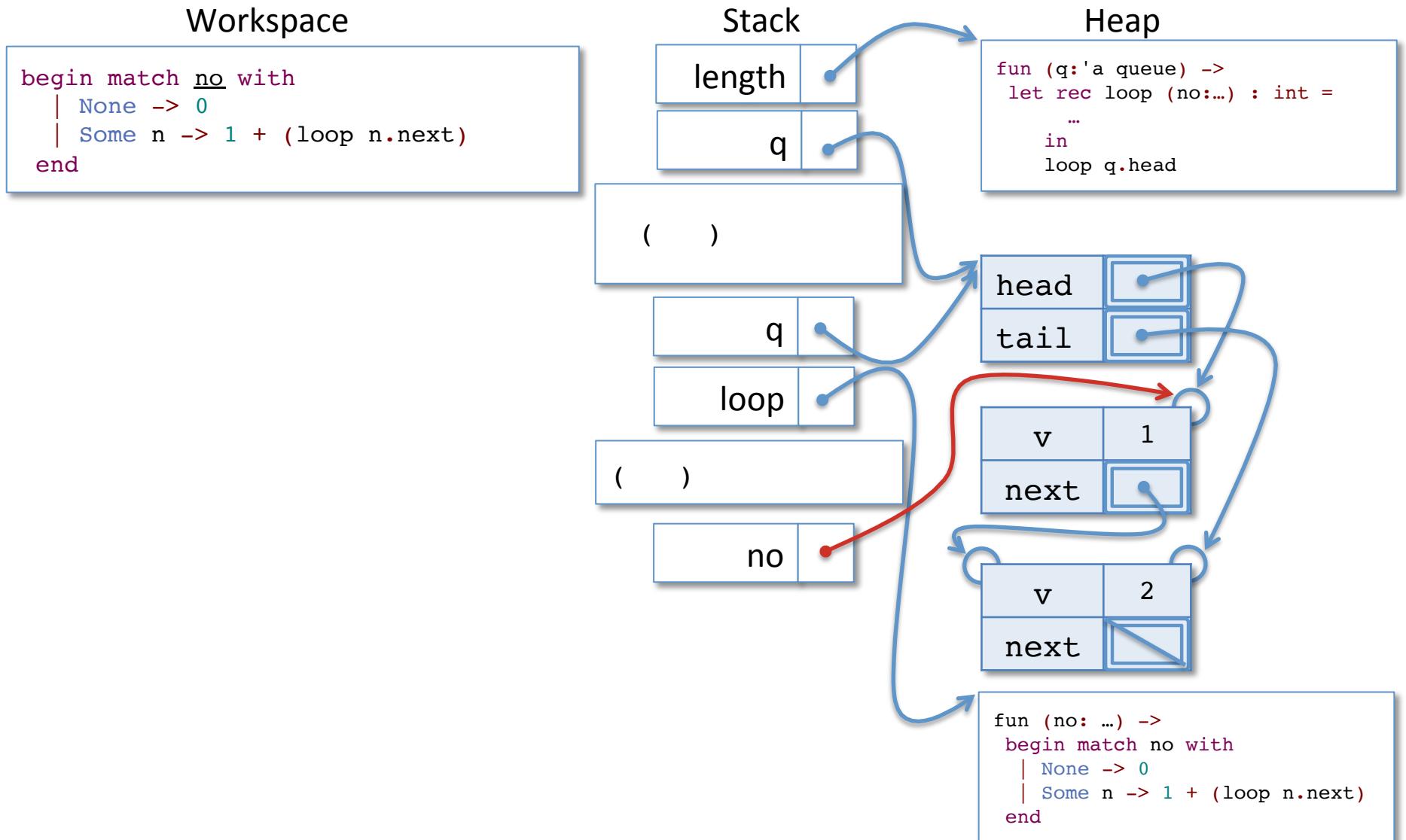
Evaluating length



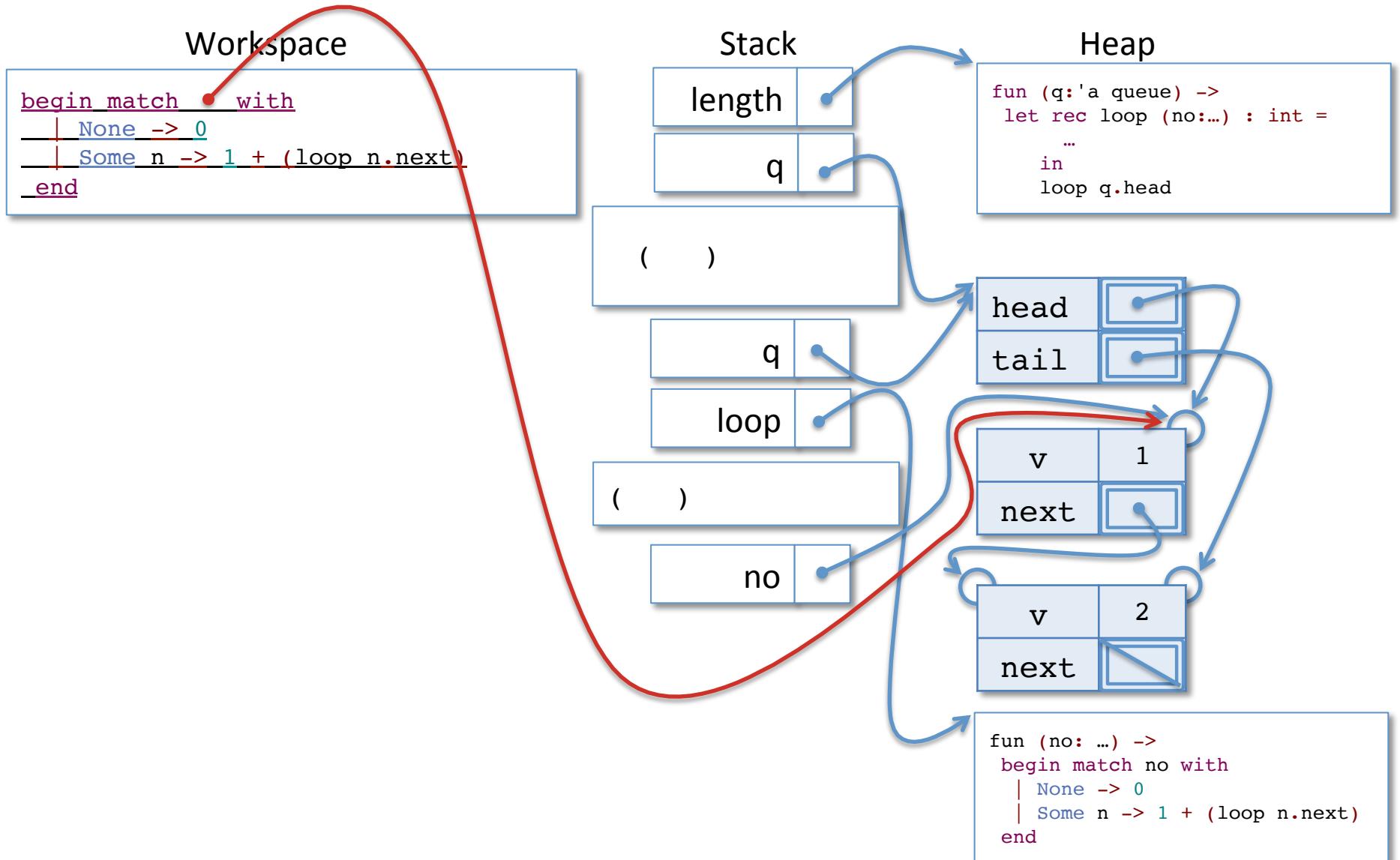
Evaluating length



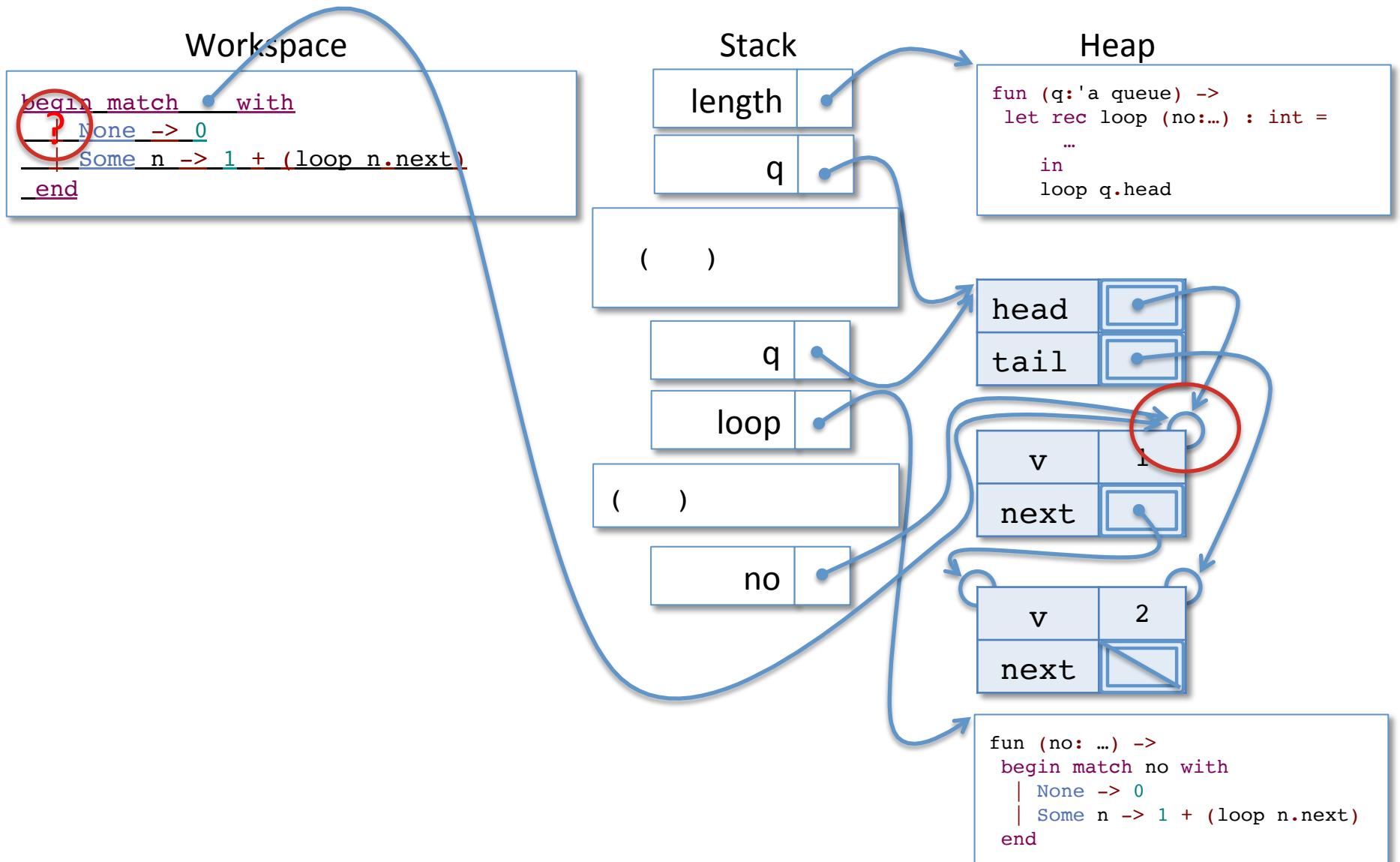
Evaluating length



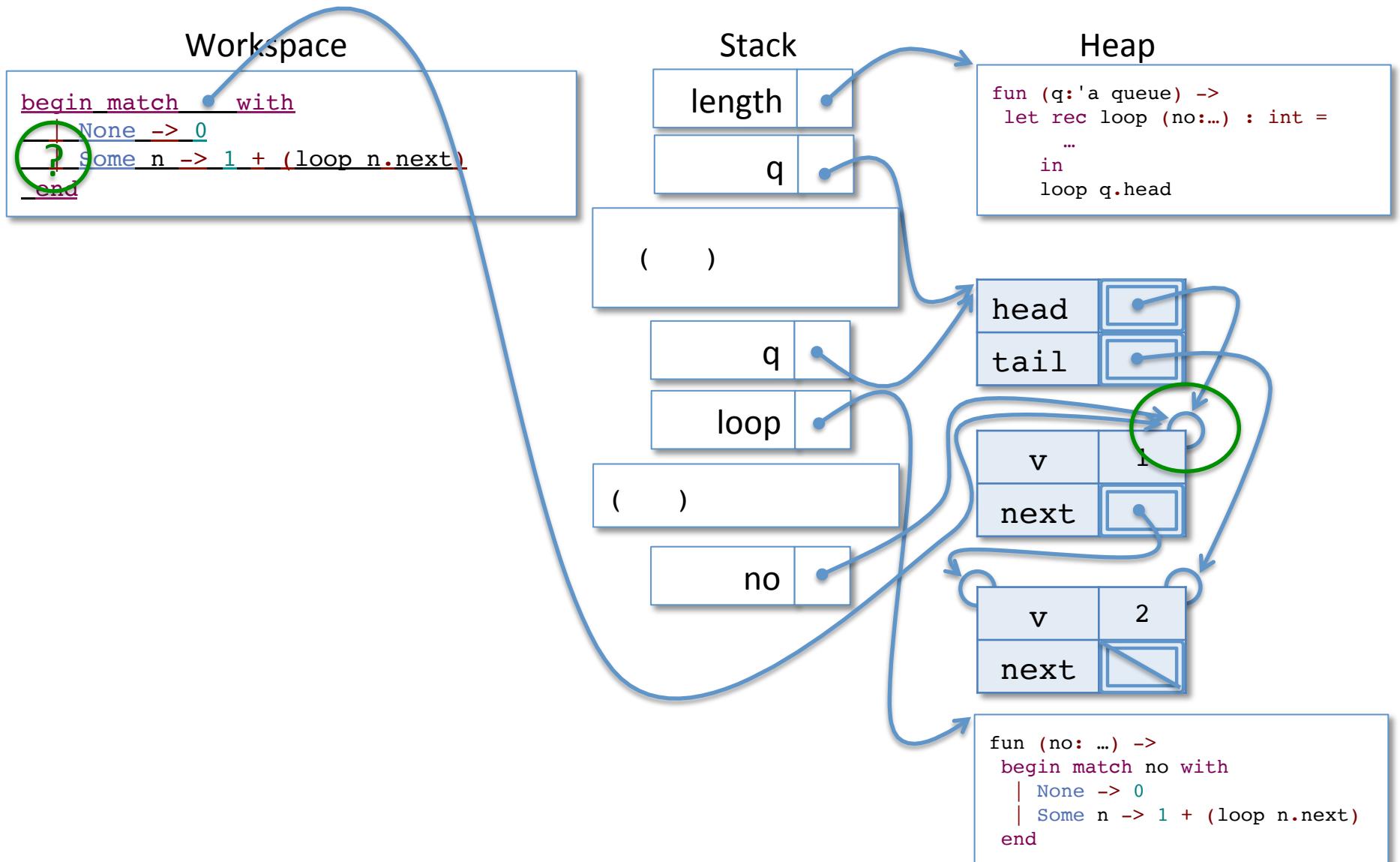
Evaluating length



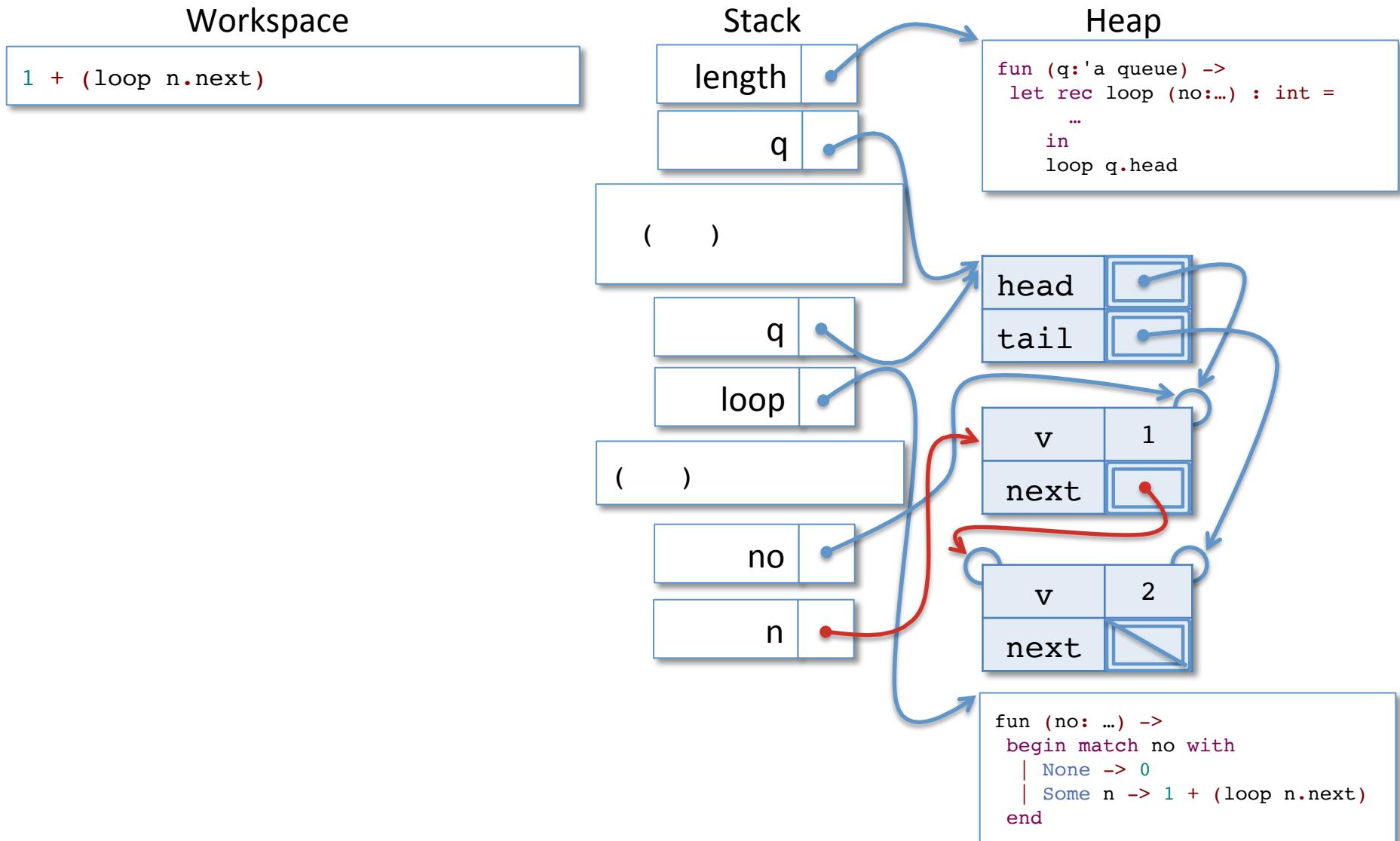
Evaluating length



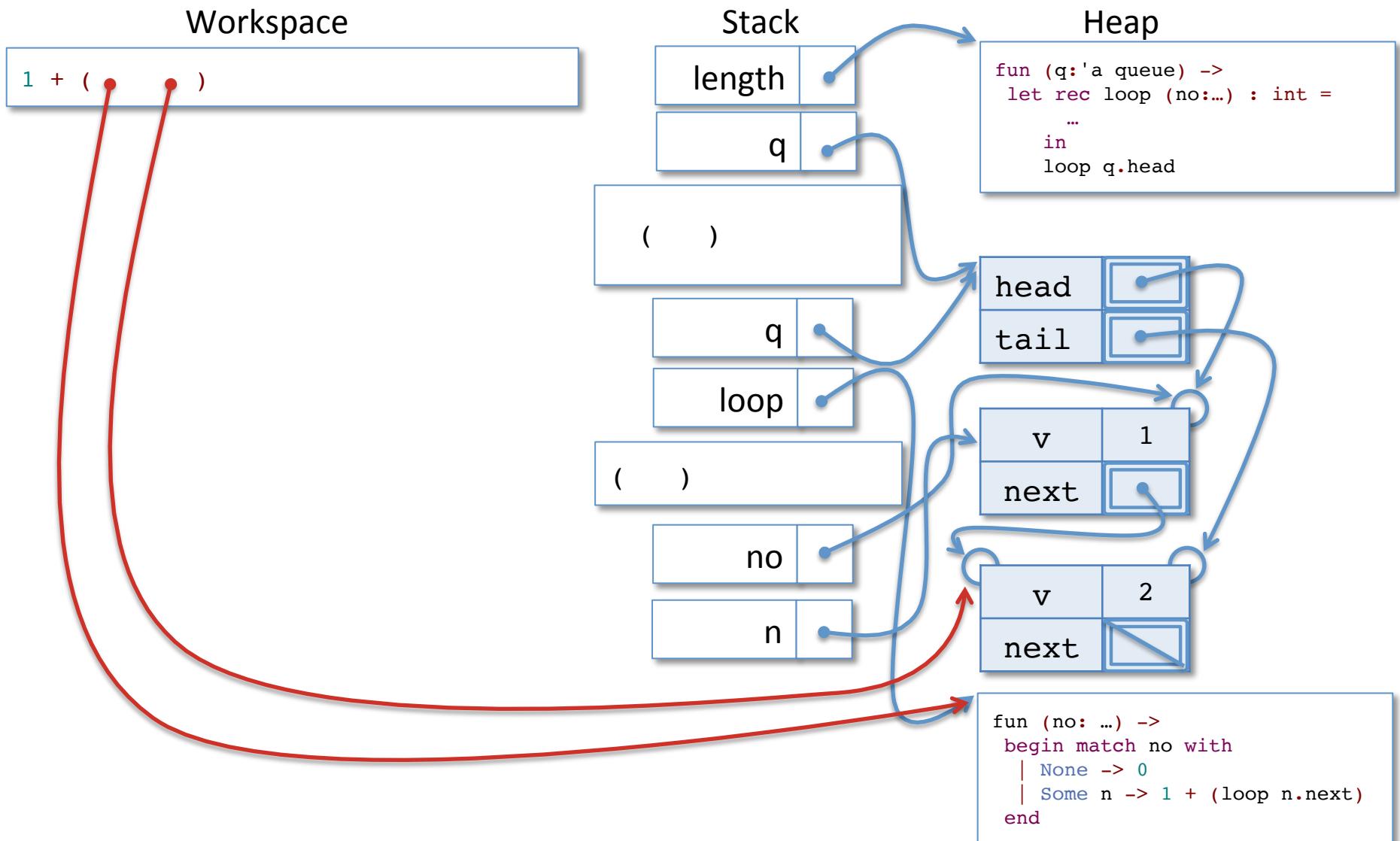
Evaluating length



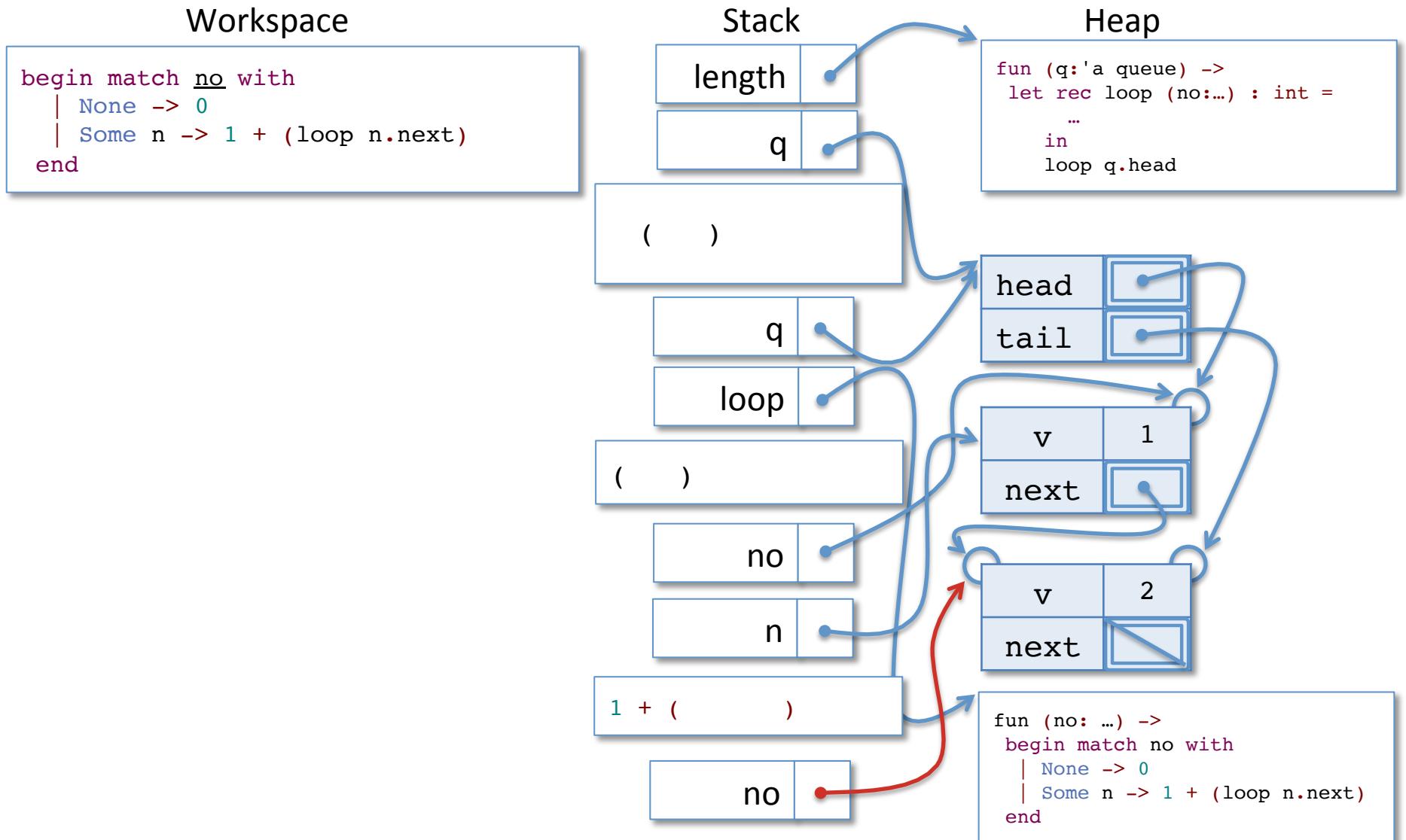
Evaluating length



Evaluating length

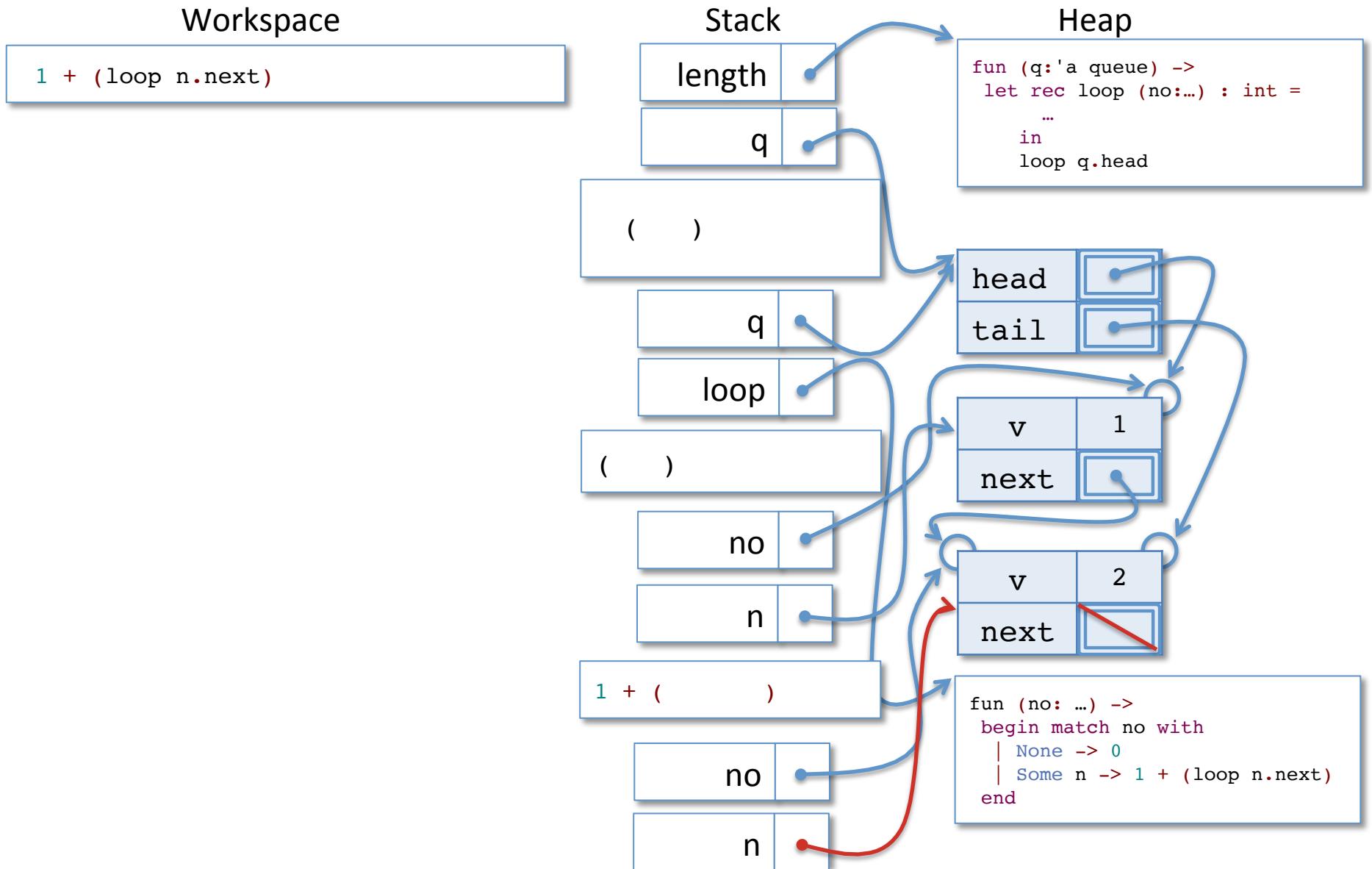


Evaluating length



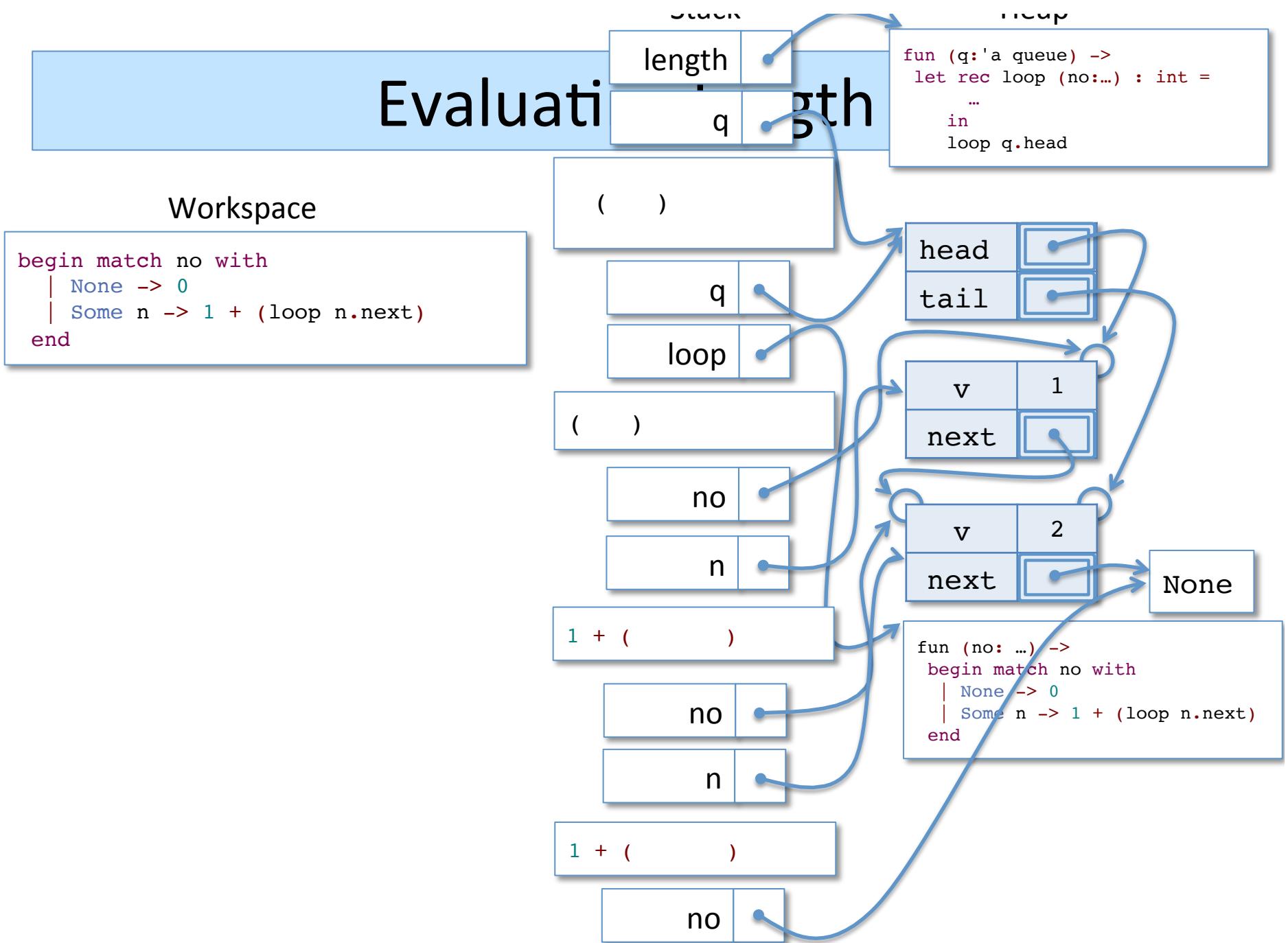
...after a few steps...

Evaluating length

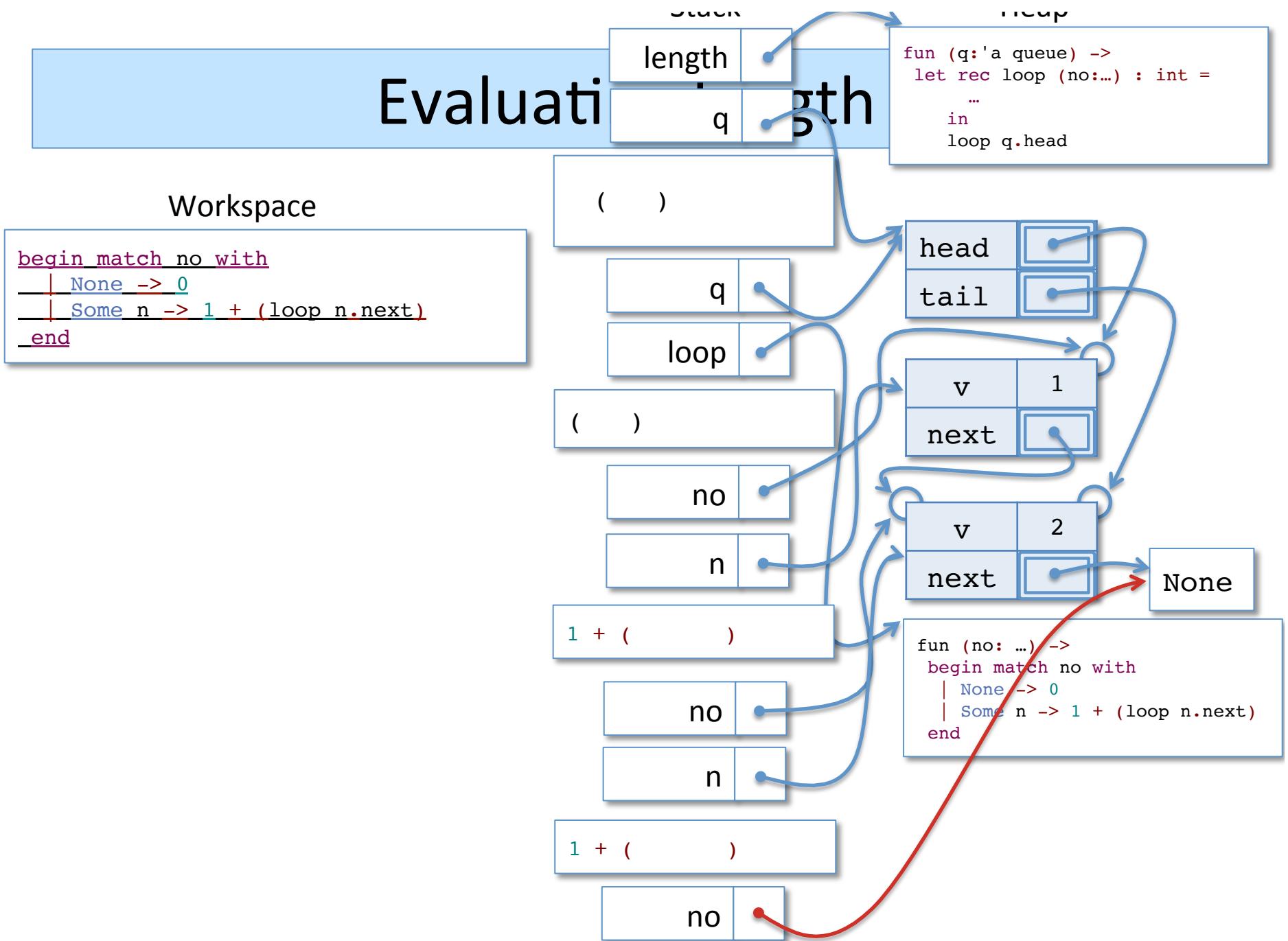


...after a few more steps...

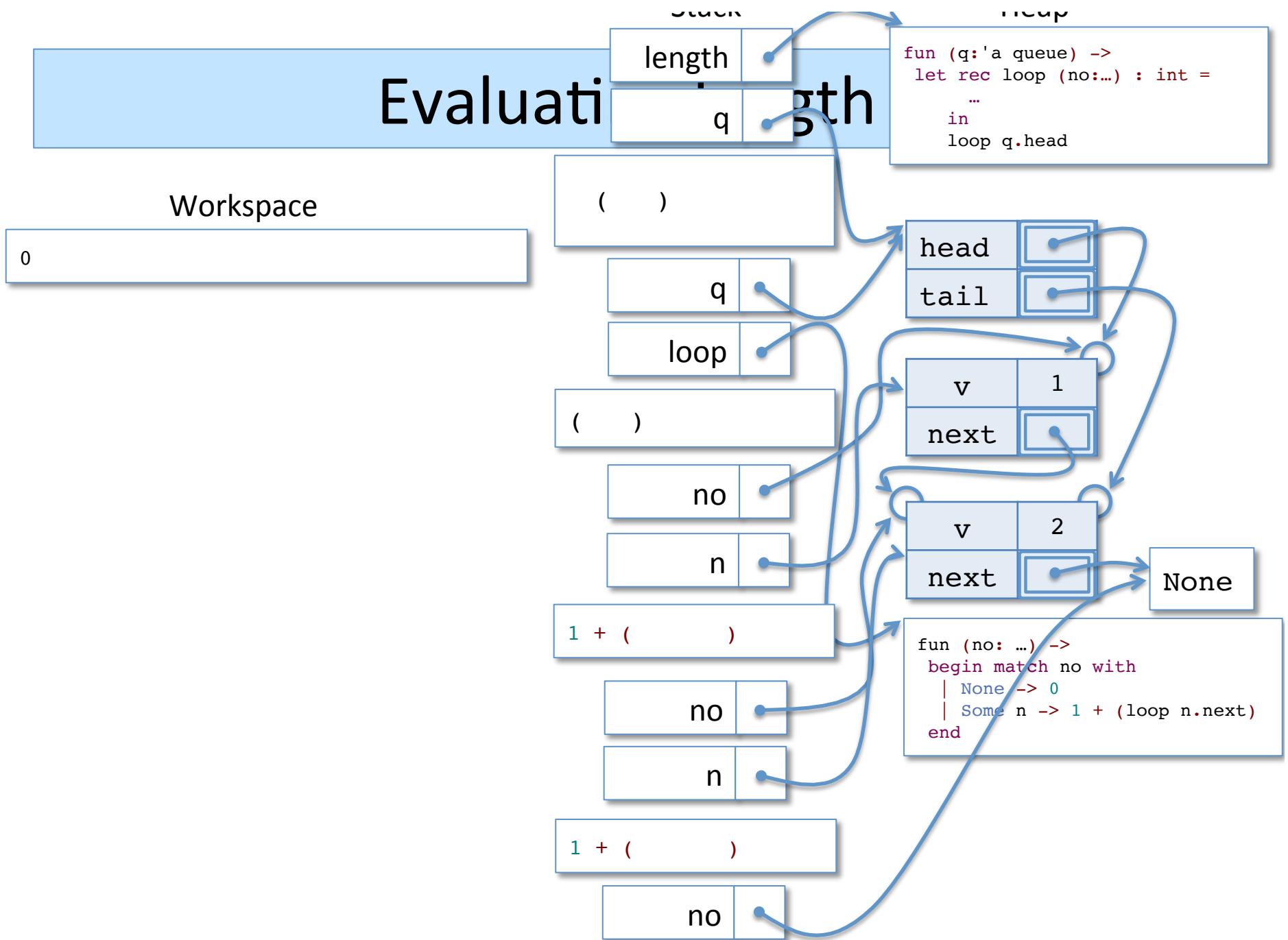
Evaluation



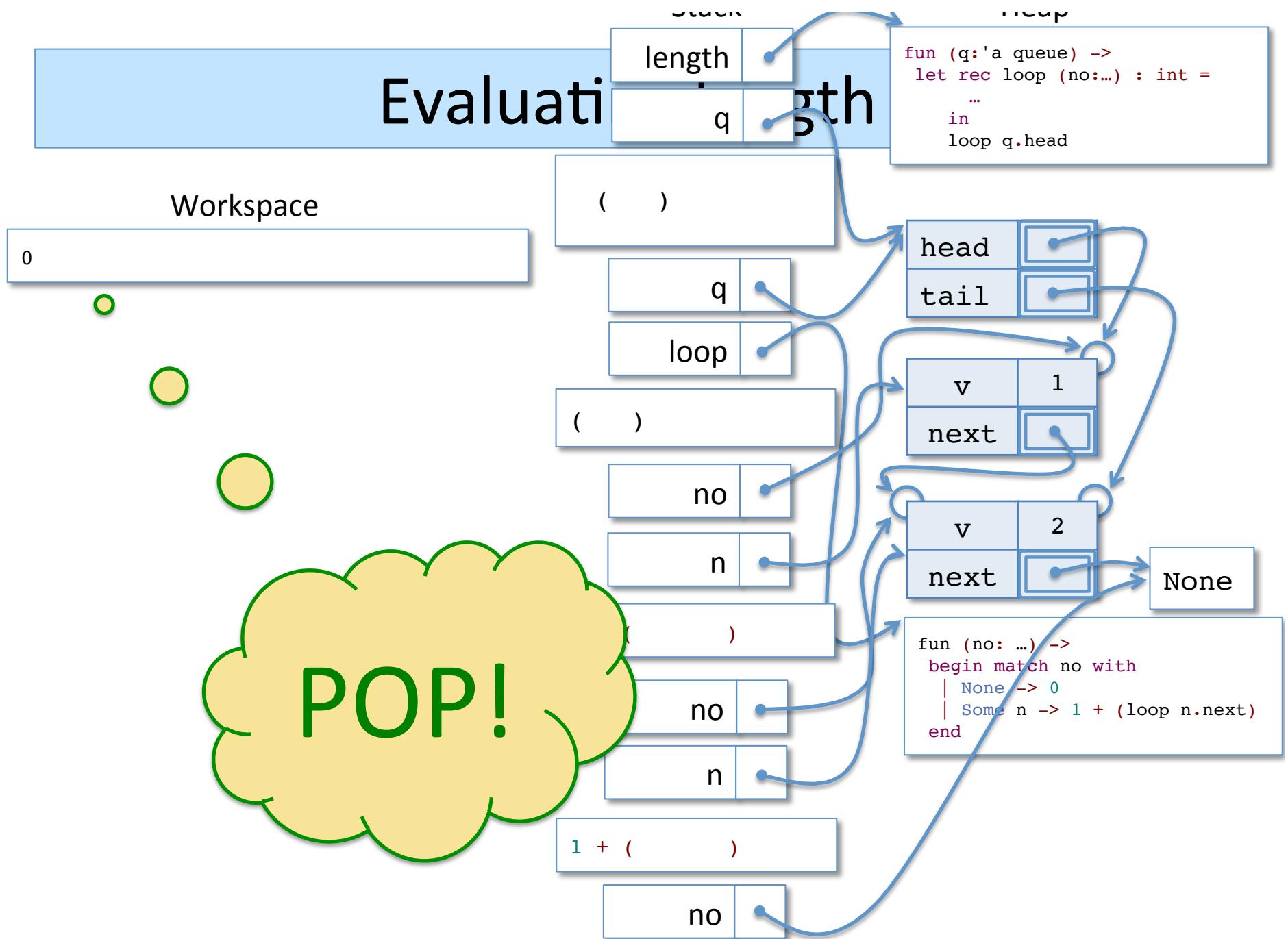
Evaluation



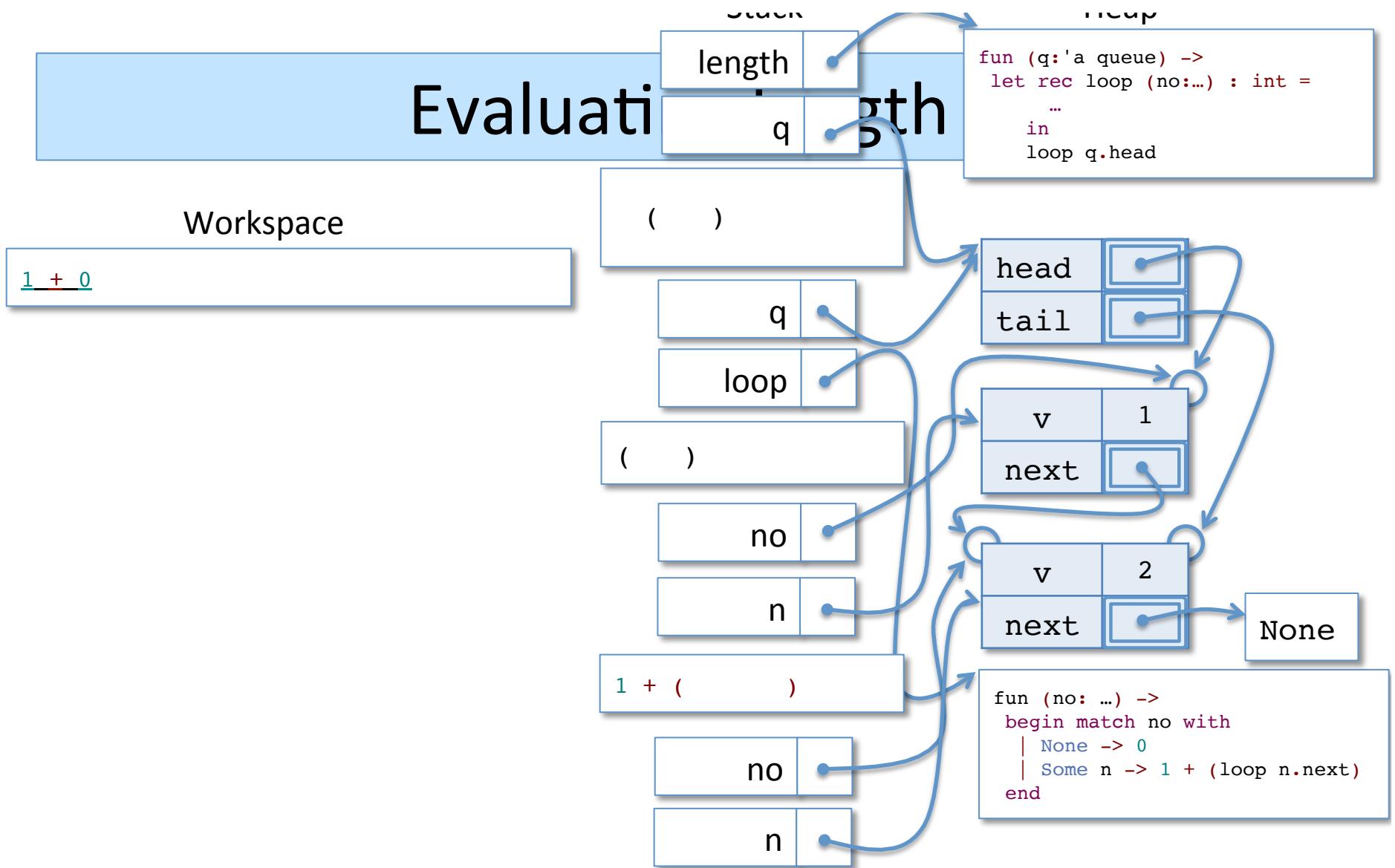
Evaluation



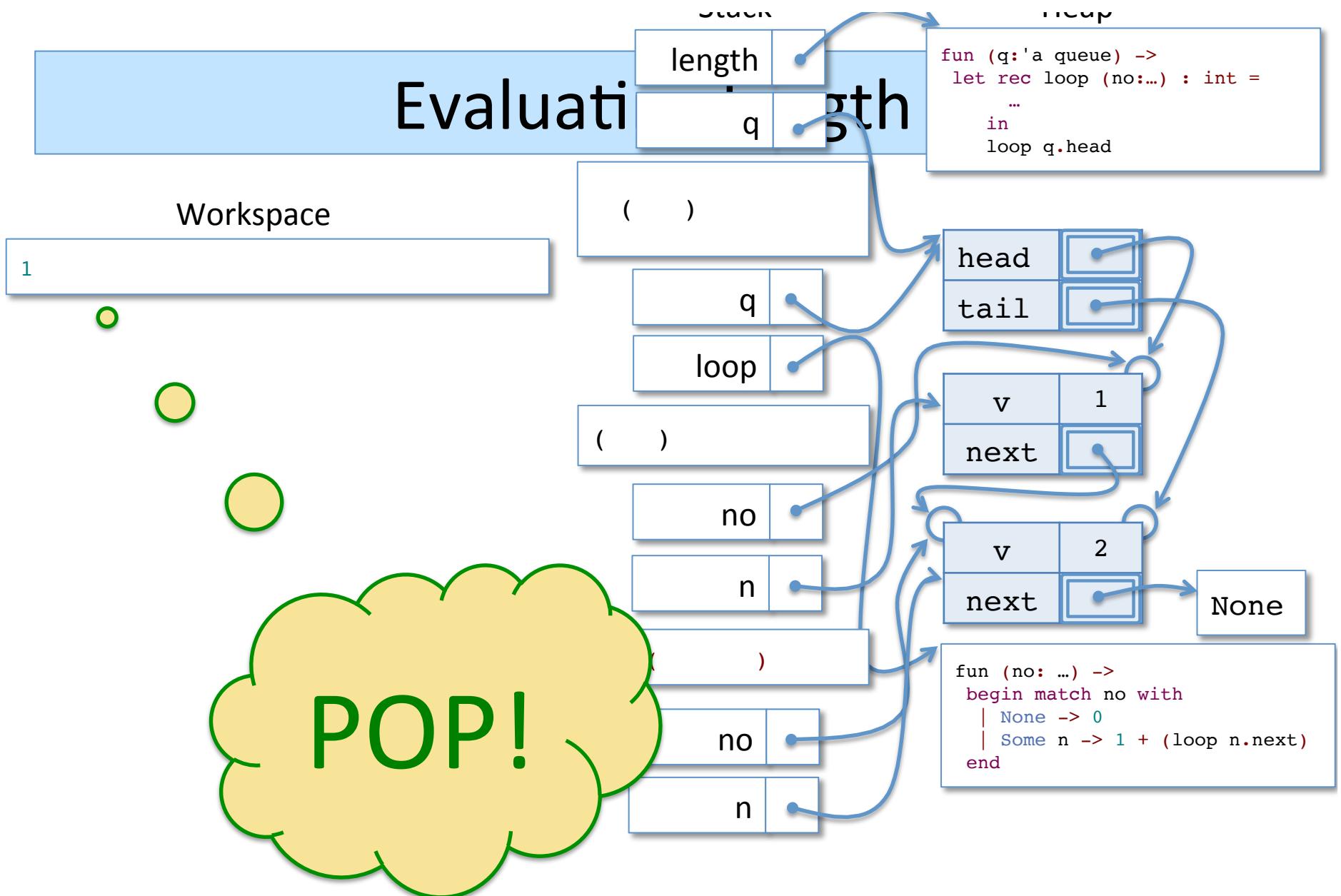
Evaluation



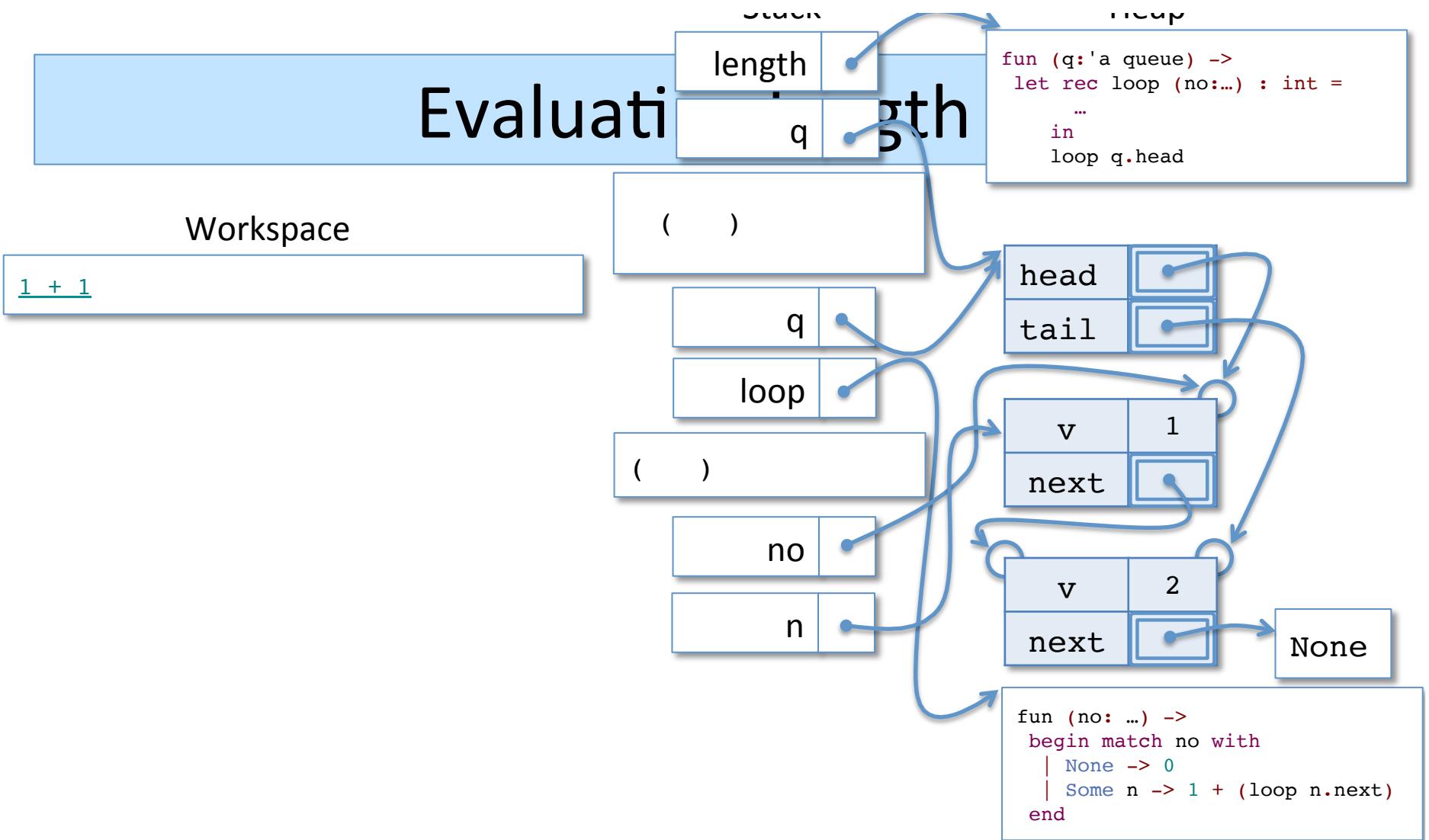
Evaluation



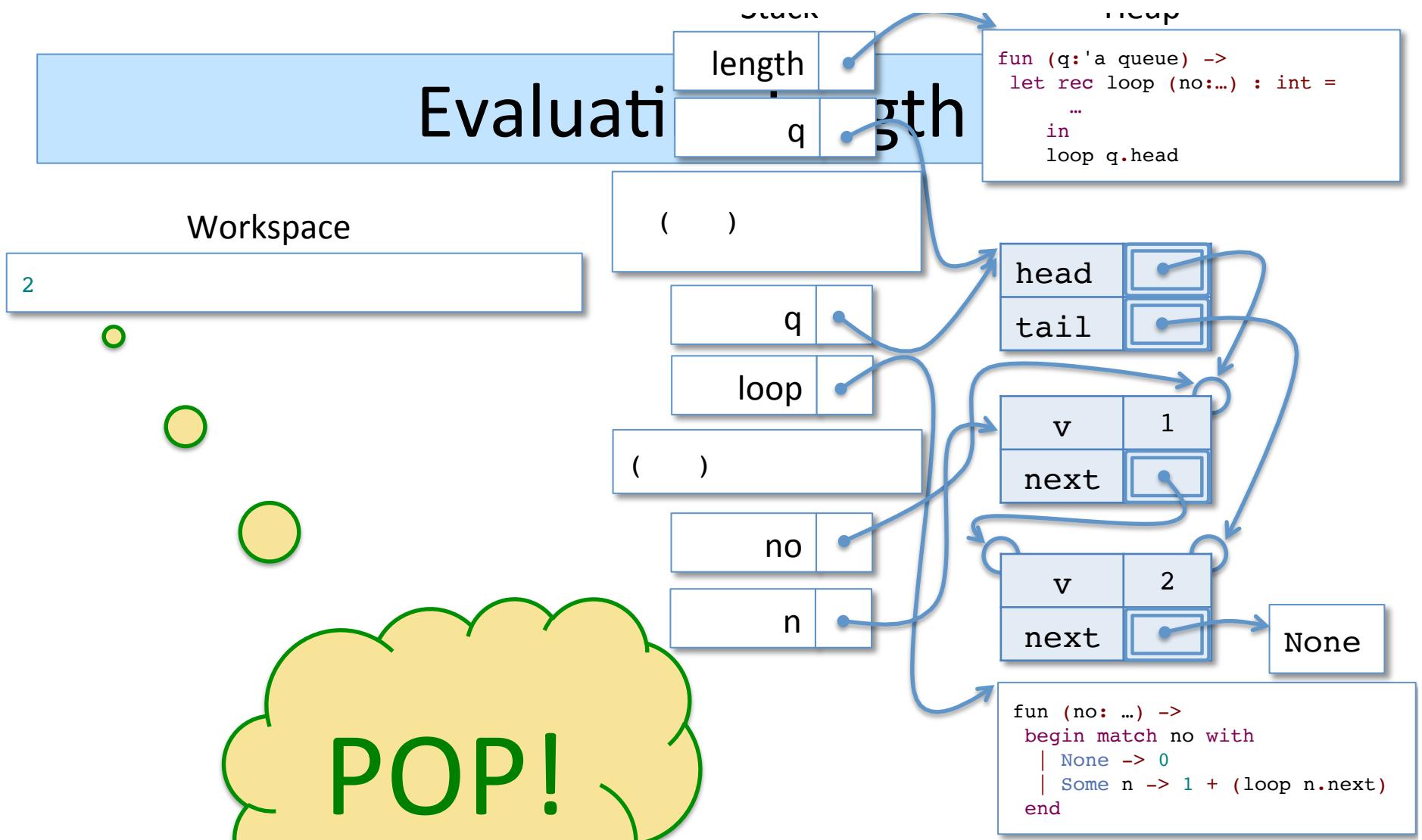
Evaluati



Evaluation

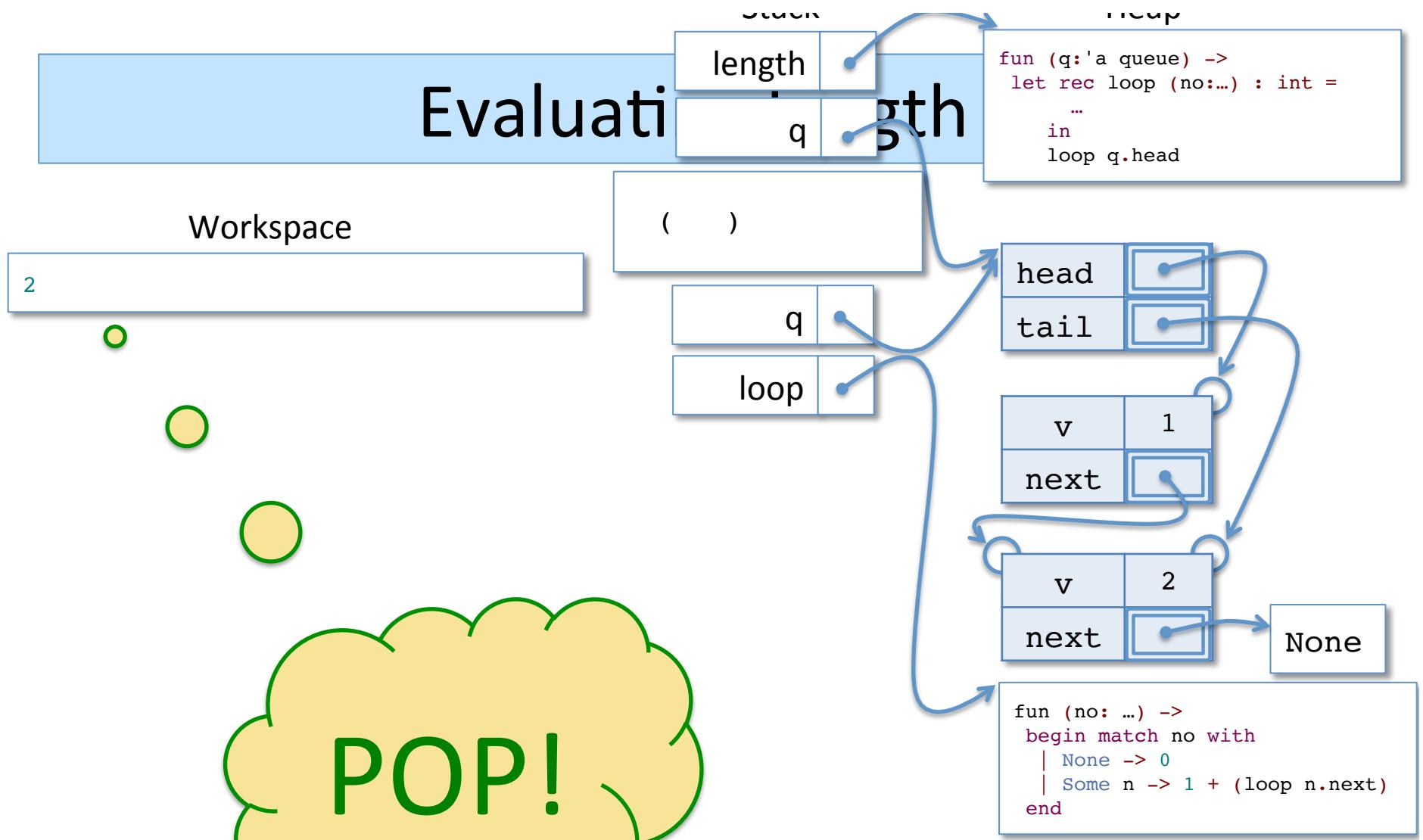


Evaluation



POP!

Evaluation



Evaluating length

Workspace

2

Stack

| | |
|--------|---|
| length | • |
| q | • |

Heap

```
fun (q:'a queue) ->
  let rec loop (no:...) : int =
    ...
    in
      loop q.head
```

head

tail

v

next

v

next

1

2

None

DONE!

```
fun (no: ...) ->
  begin match no with
    | None -> 0
    | Some n -> 1 + (loop n.next)
  end
```

Iteration

Using tail calls for loops

length (using iteration)

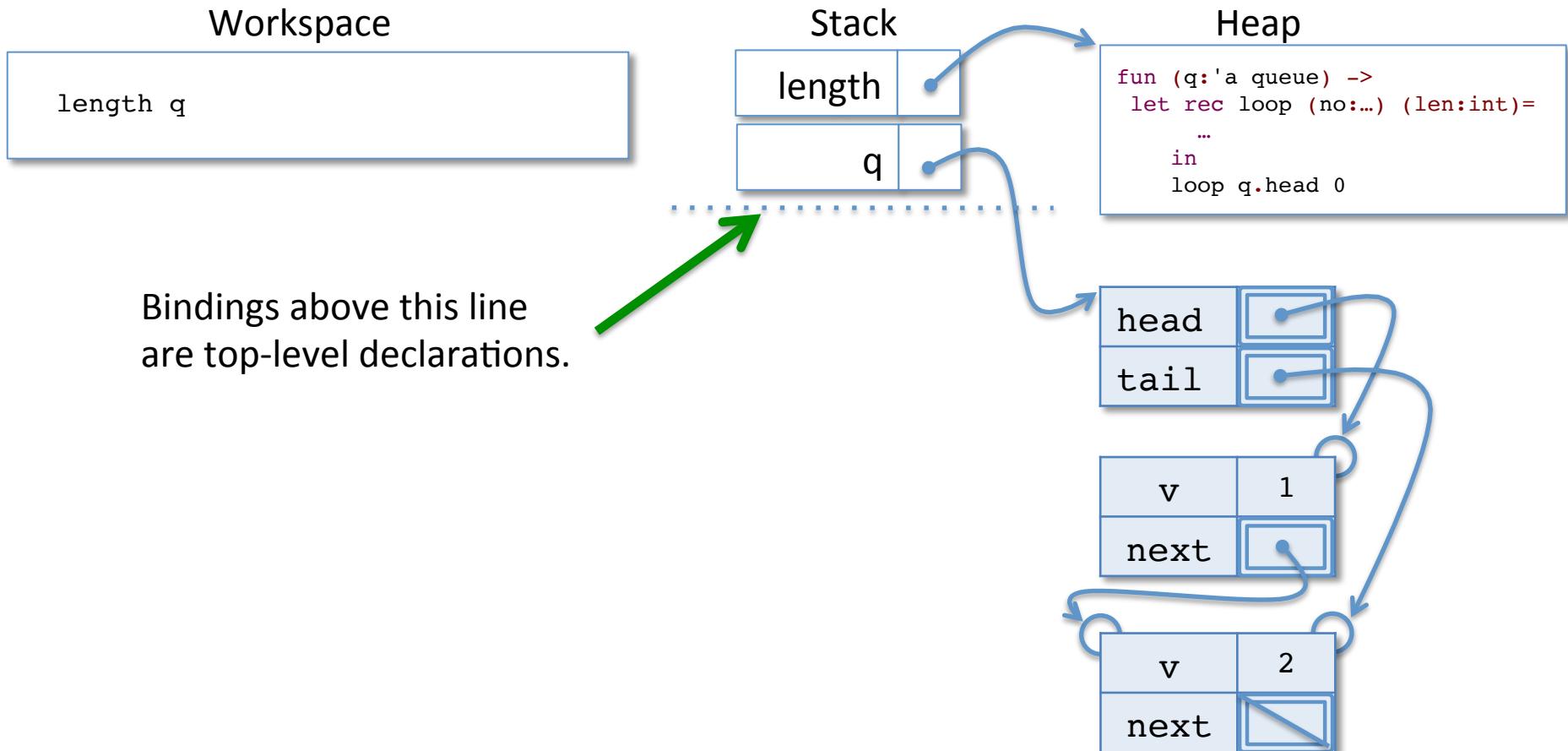
```
(* Calculate the length of the list using iteration *)
let length (q:'a queue) : int =
    let rec loop (no:'a qnode option) (len:int) : int =
        begin match no with
            | None -> len
            | Some n -> loop n.next (1+ len)
        end
    in
    loop q.head 0
```

- This code for `length` also uses a helper function, `loop`:
 - This loop takes an extra argument, `len`, called the *accumulator*
 - Unlike the previous solution, the computation happens “on the way down” as opposed to “on the way back up”
 - Note that `loop` will always be called in an empty workspace—the results of the call to `loop` never need to be used to compute another expression. In contrast, we had $(1 + (\text{loop} \dots))$ in the recursive version.

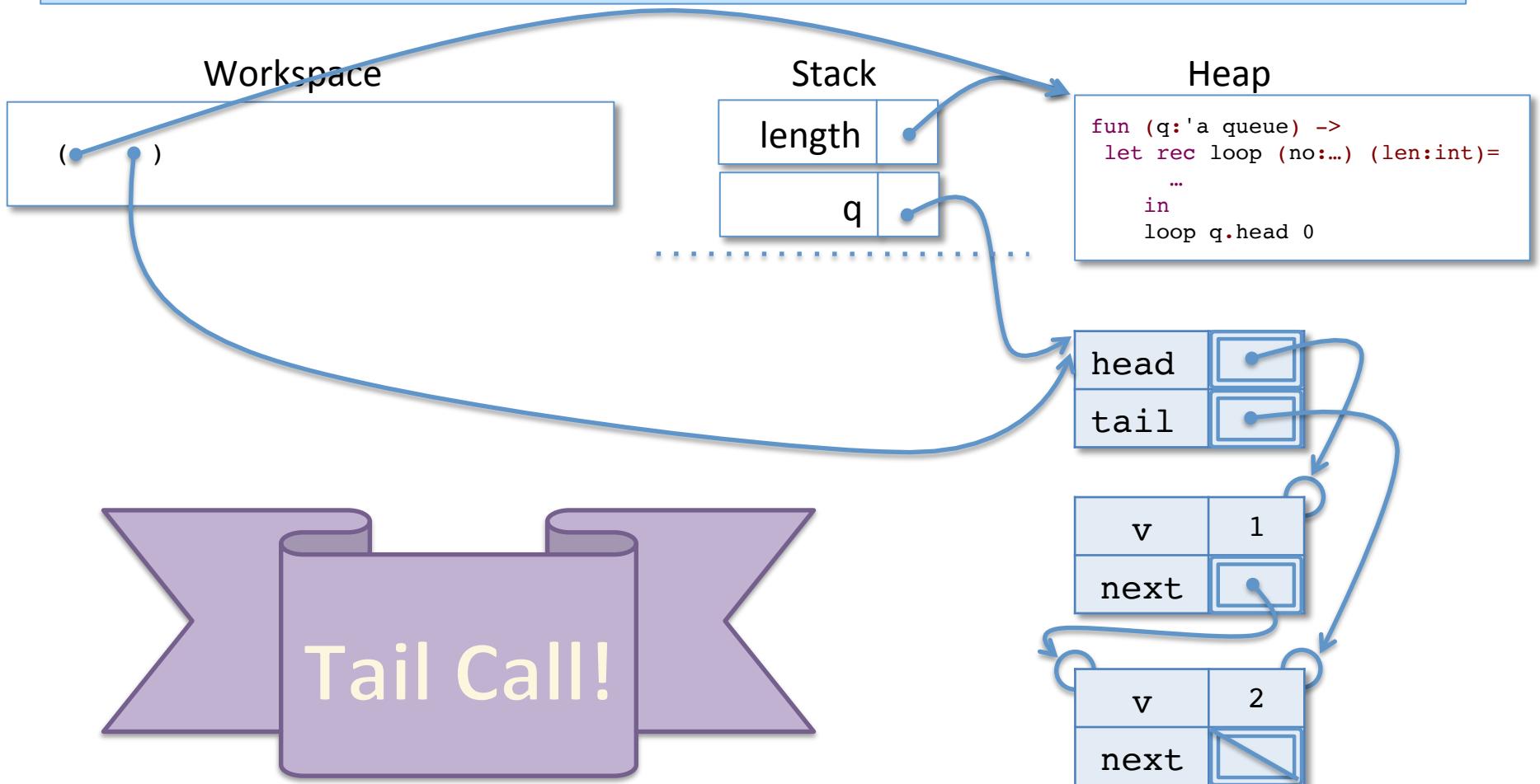
Tail Call Optimization

- Why does it matter that ‘loop’ is only called in an empty workspace?
- We can *optimize* the abstract stack machine:
 - The workspace pushed onto the stack tells us “what to do” when the function call returns.
 - If the pushed workspace is empty, we will always ‘pop’ immediately after the function call returns.
 - So there is no need to save the empty workspace on the stack!
 - Moreover, any local variables that were pushed so that the current workspace could evaluate will no longer be needed, so we can eagerly pop them too.
- The upshot is that we can execute a tail recursion just like a ‘for’ loop in Java or C, using a constant amount of stack space.

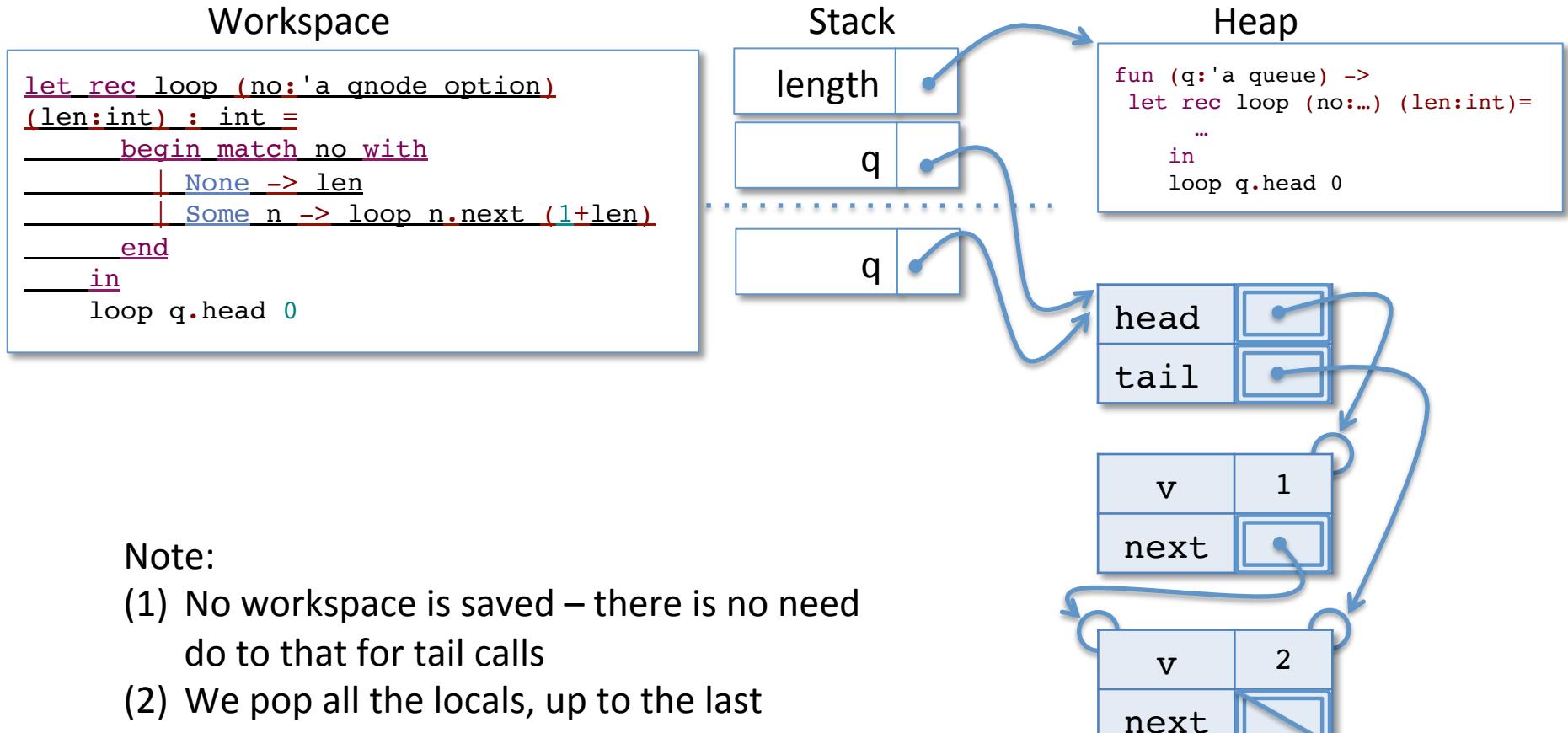
Tail Calls and Iterative length



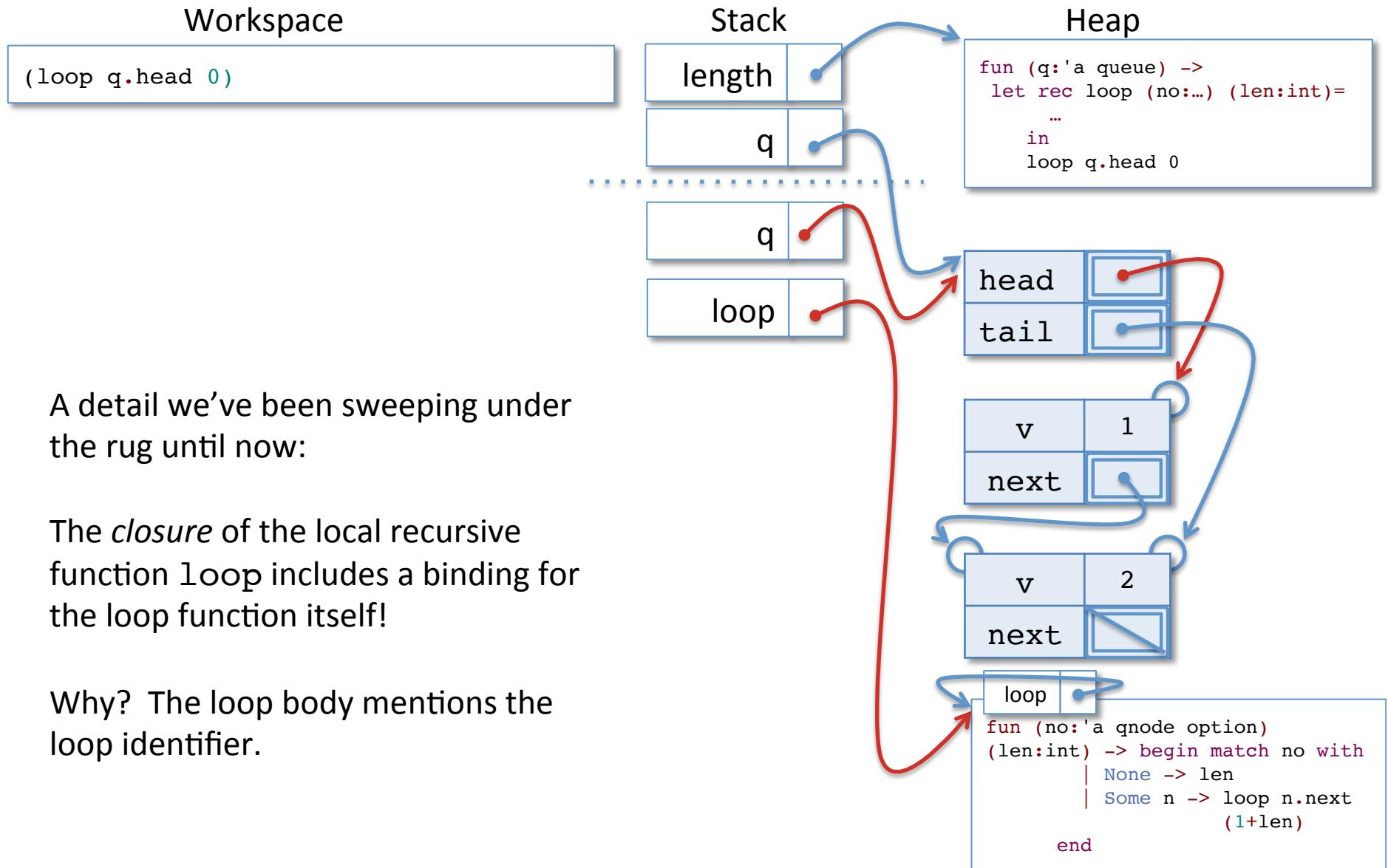
Tail Calls and Iterative length



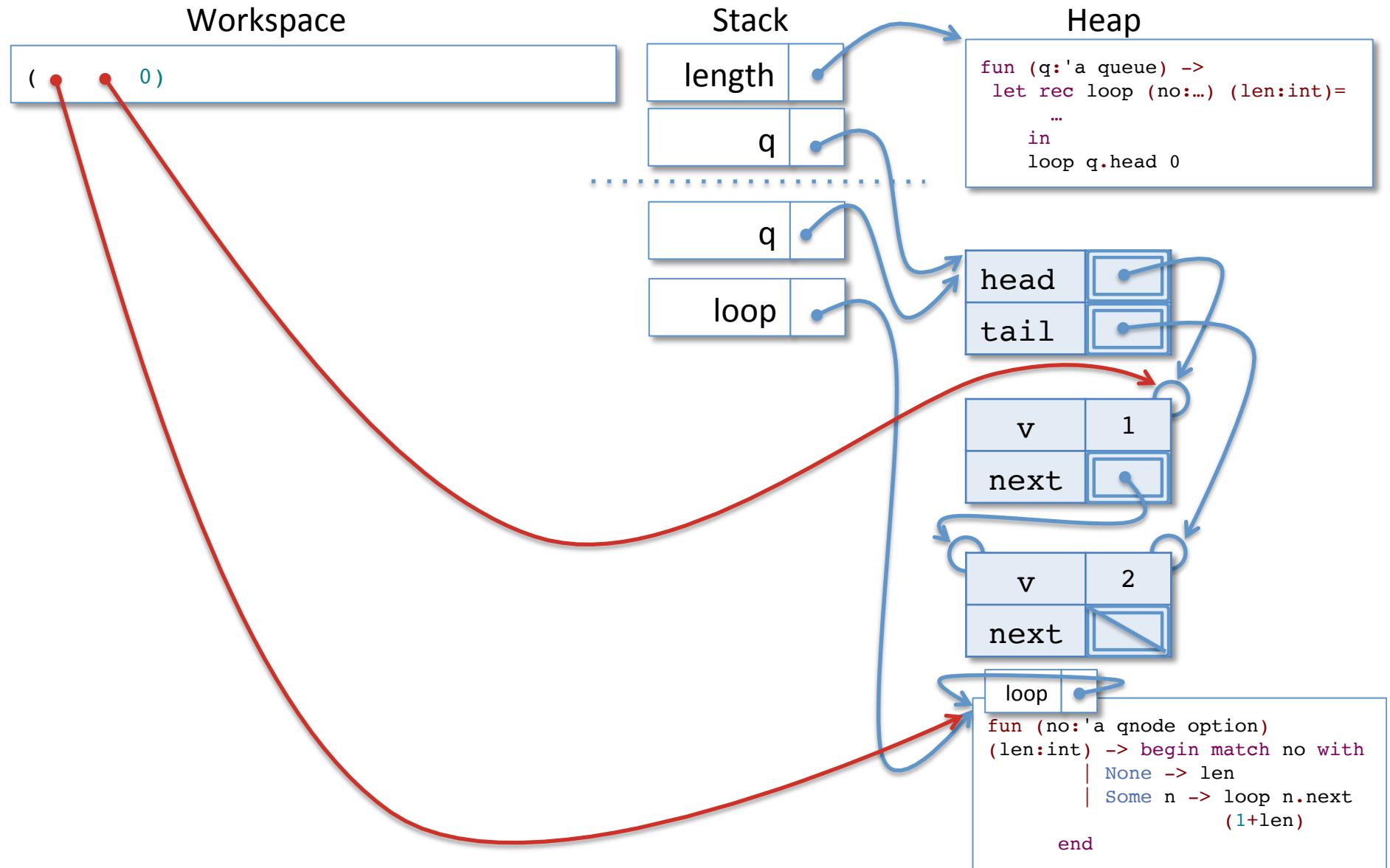
Tail Calls and Iterative length



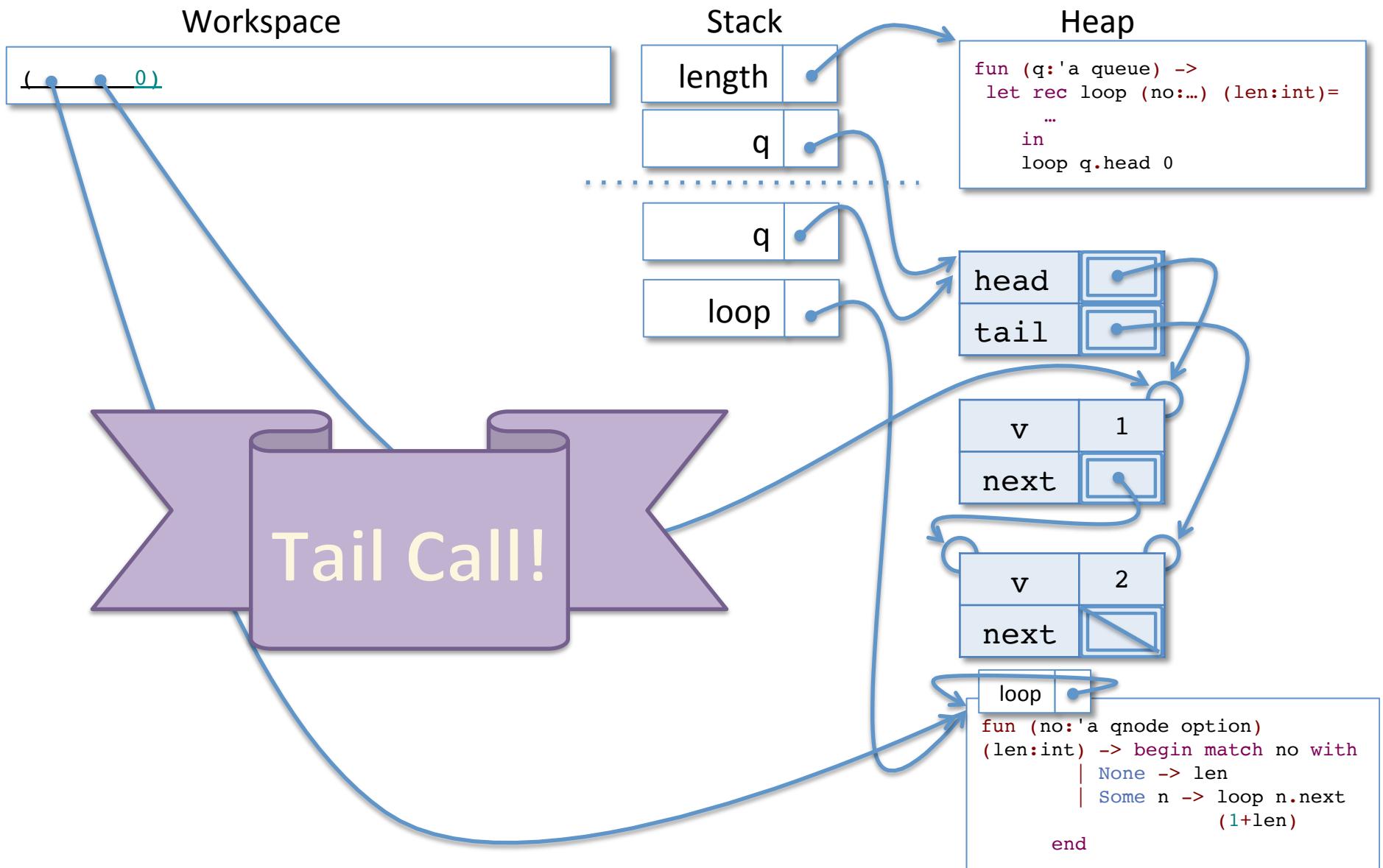
Tail Calls and Iterative length



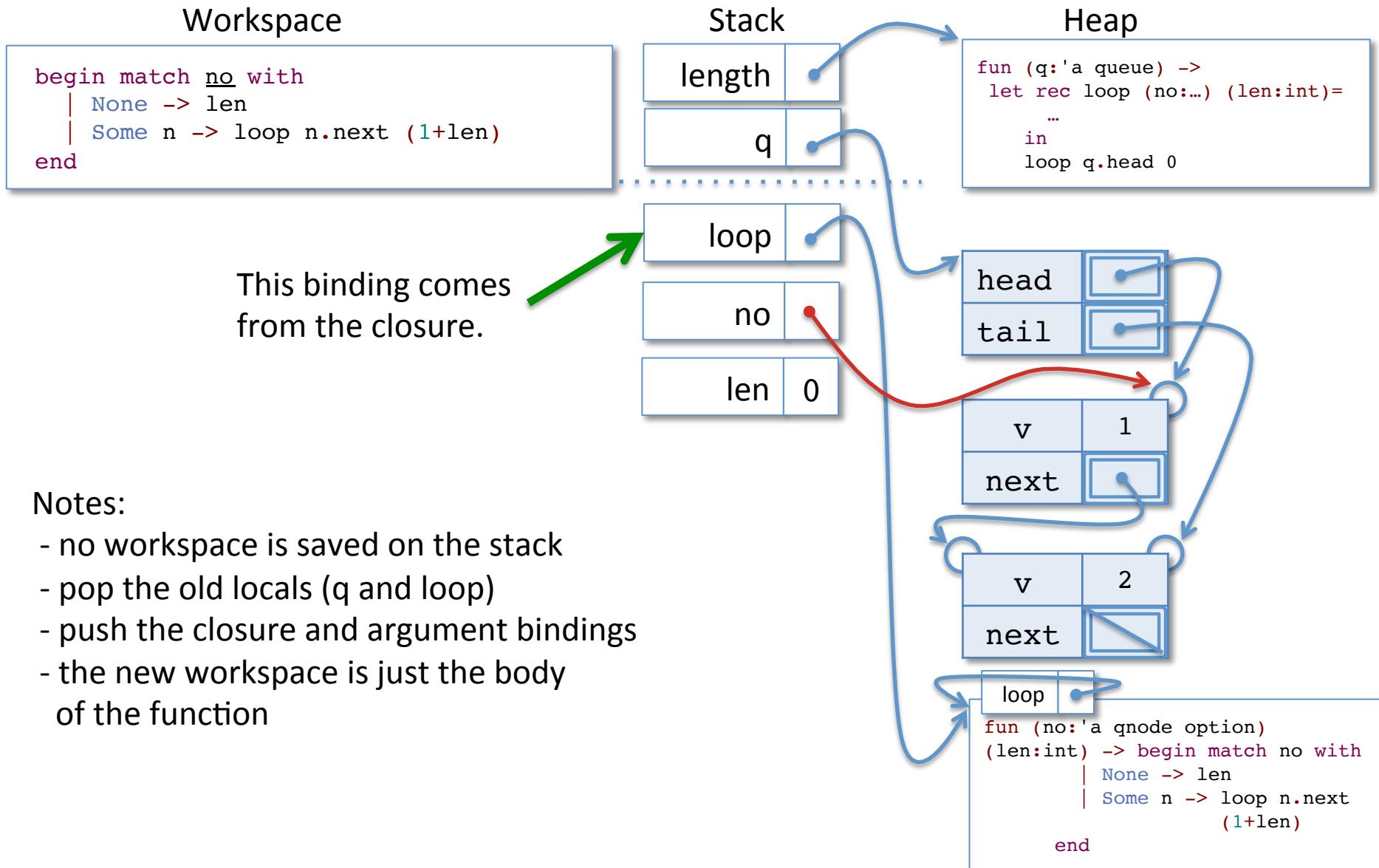
Tail Calls and Iterative length



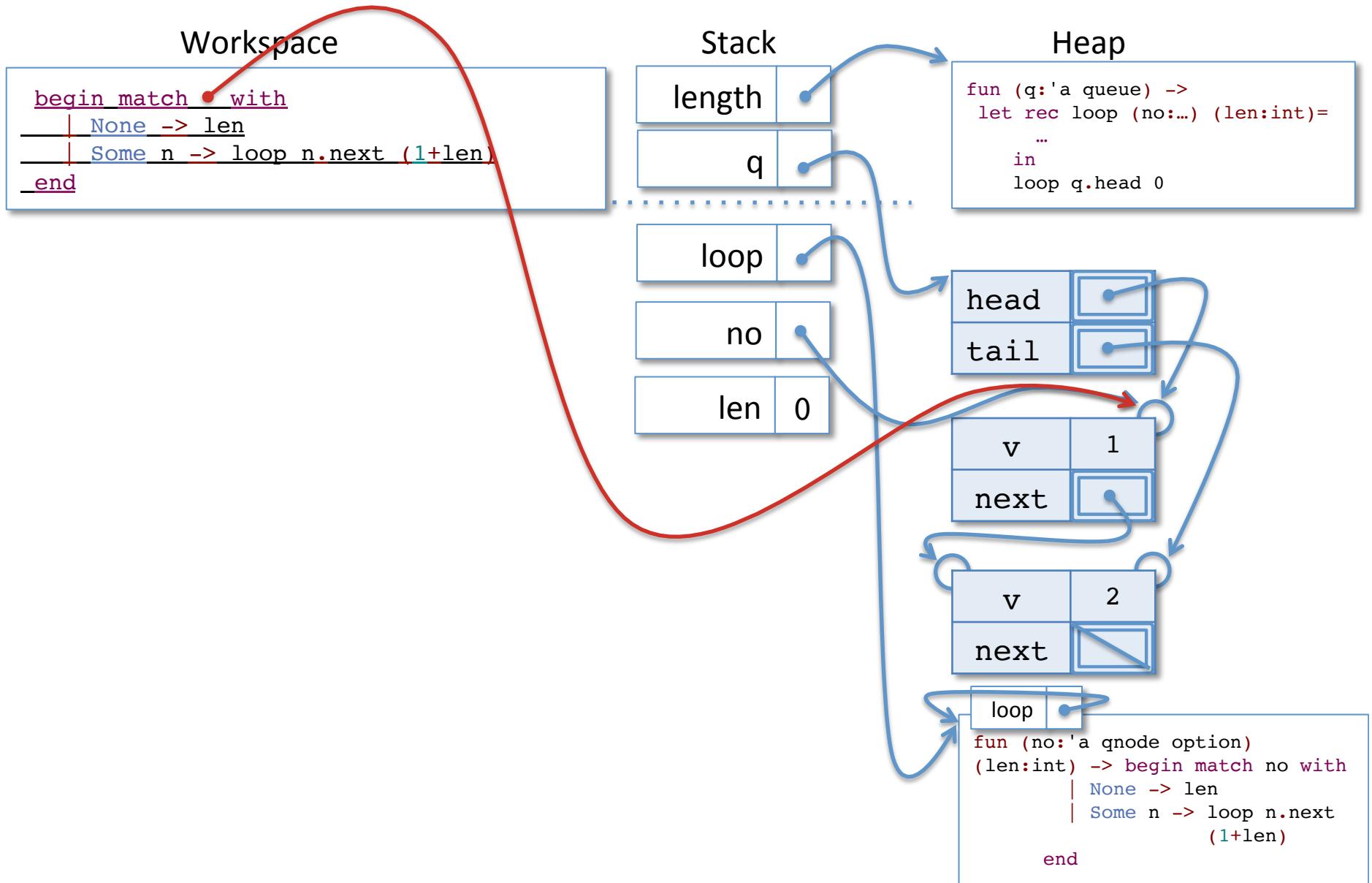
Tail Calls and Iterative length



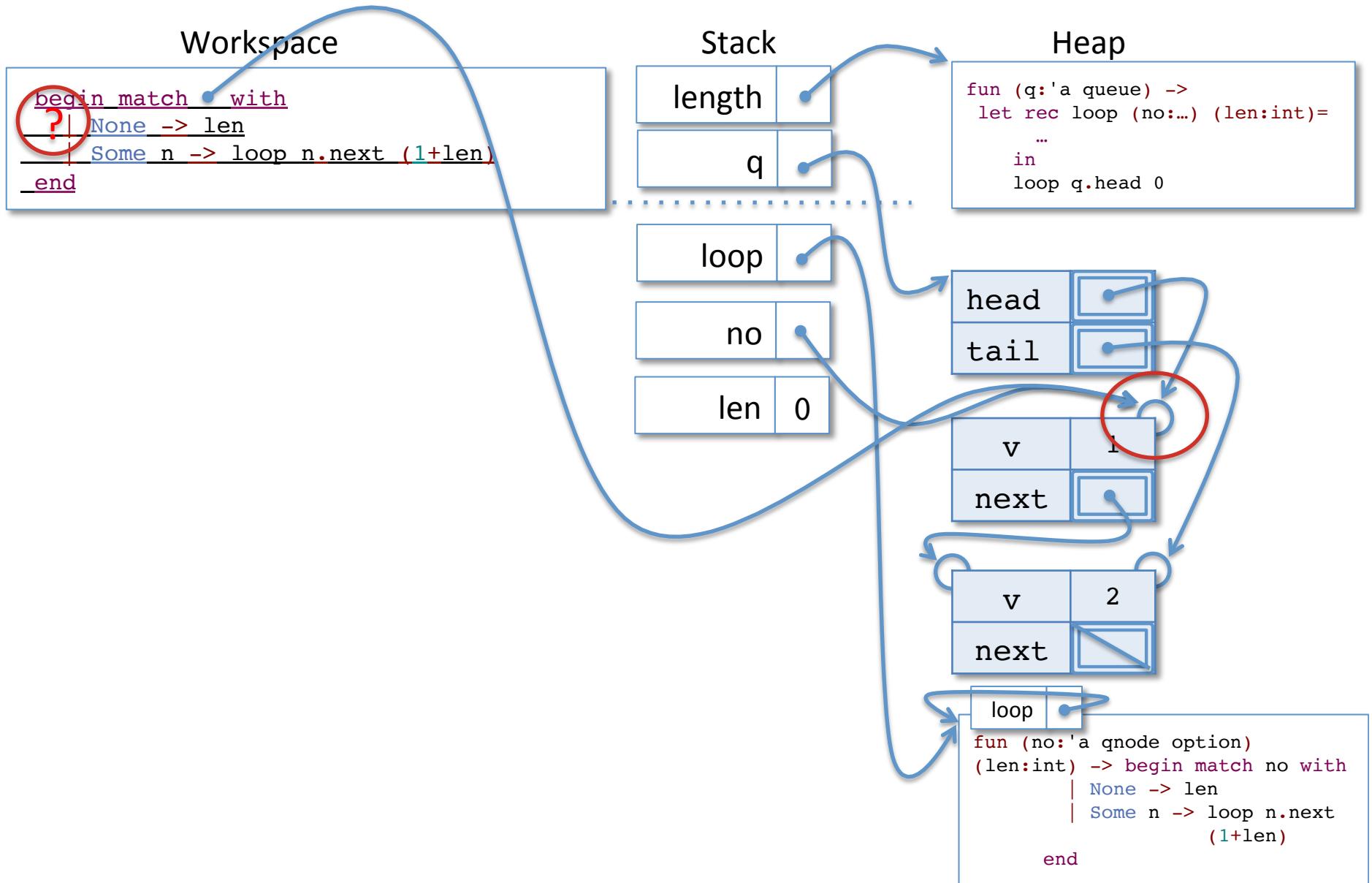
Tail Calls and Iterative length



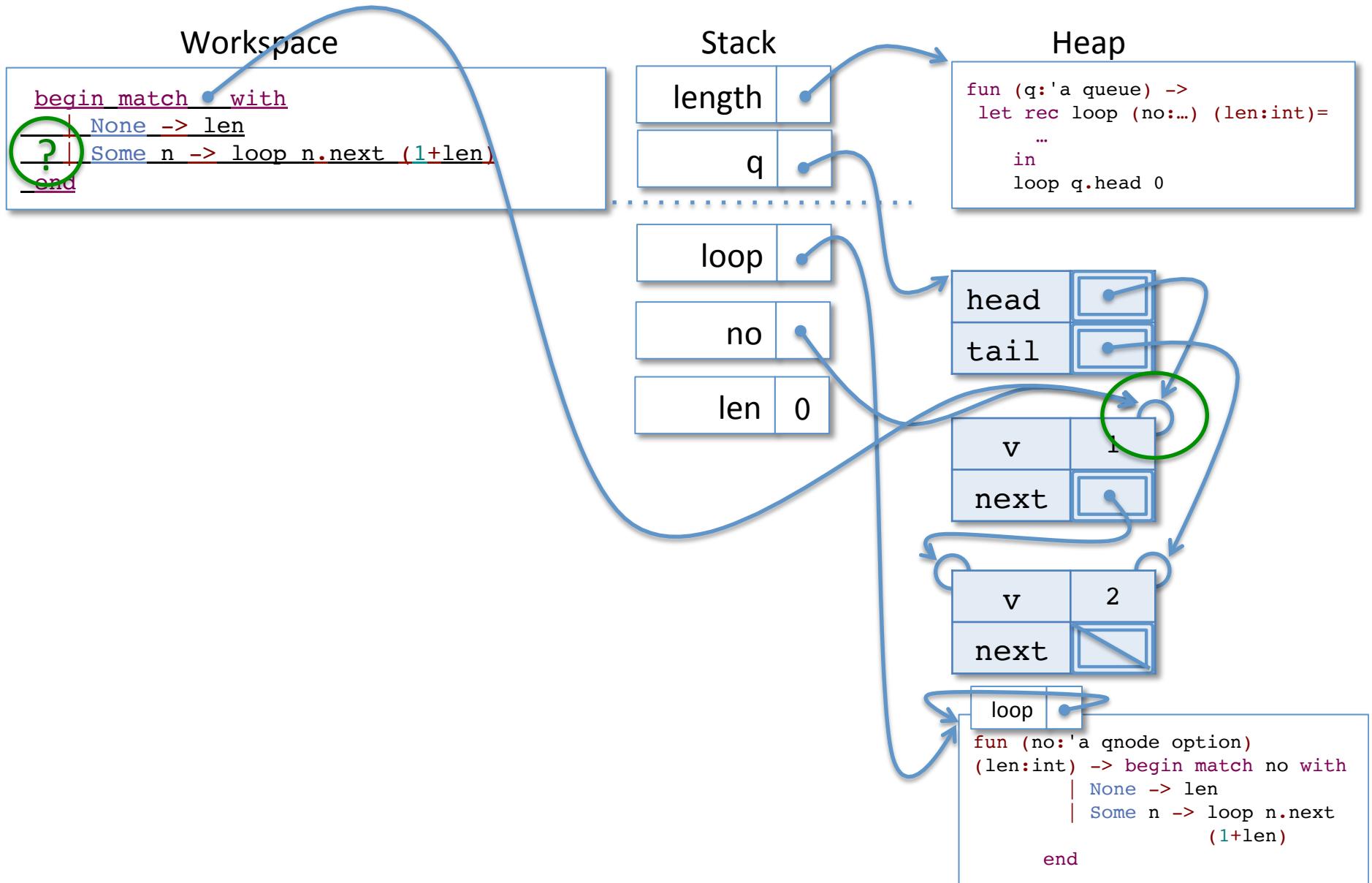
Tail Calls and Iterative length



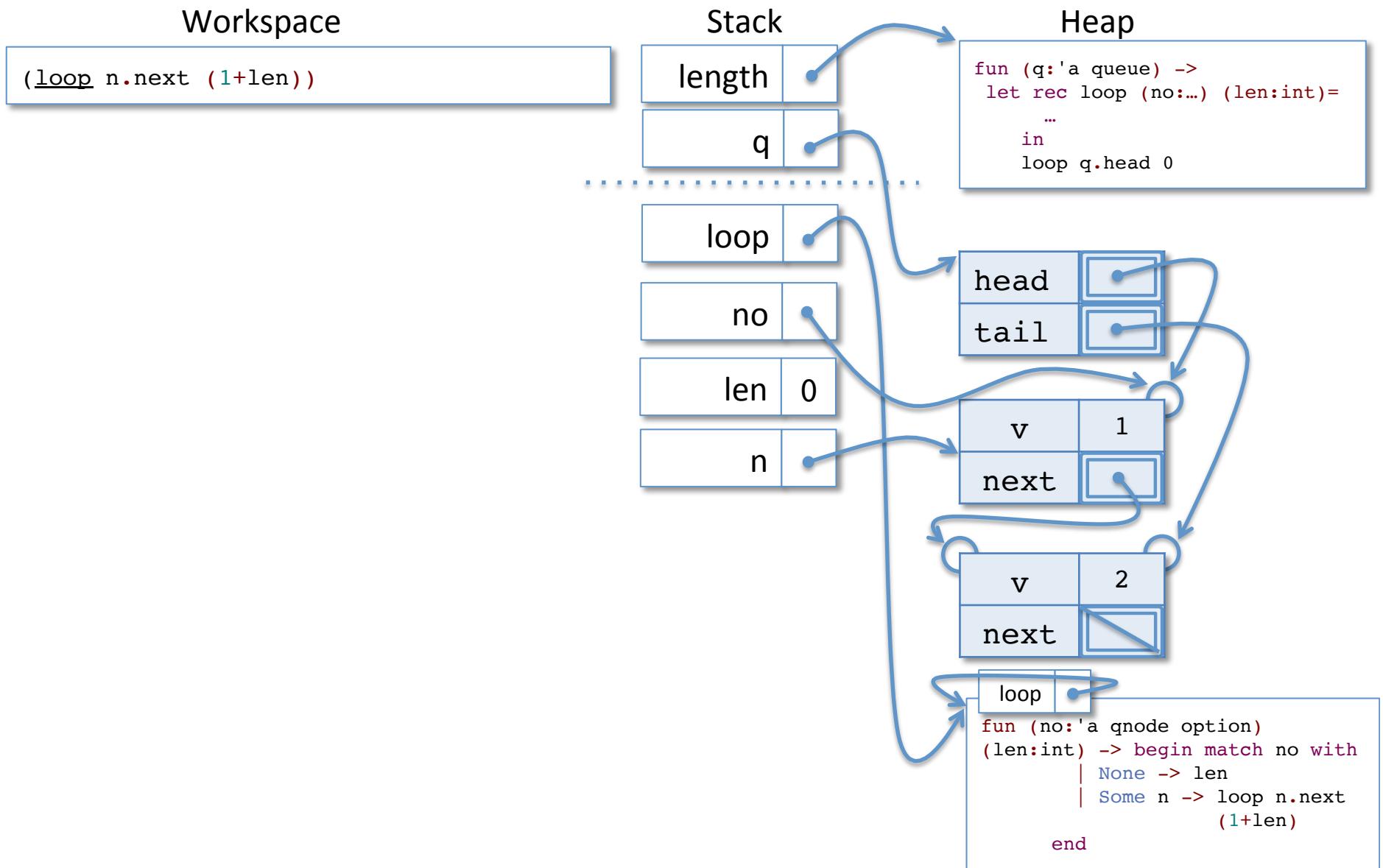
Tail Calls and Iterative length



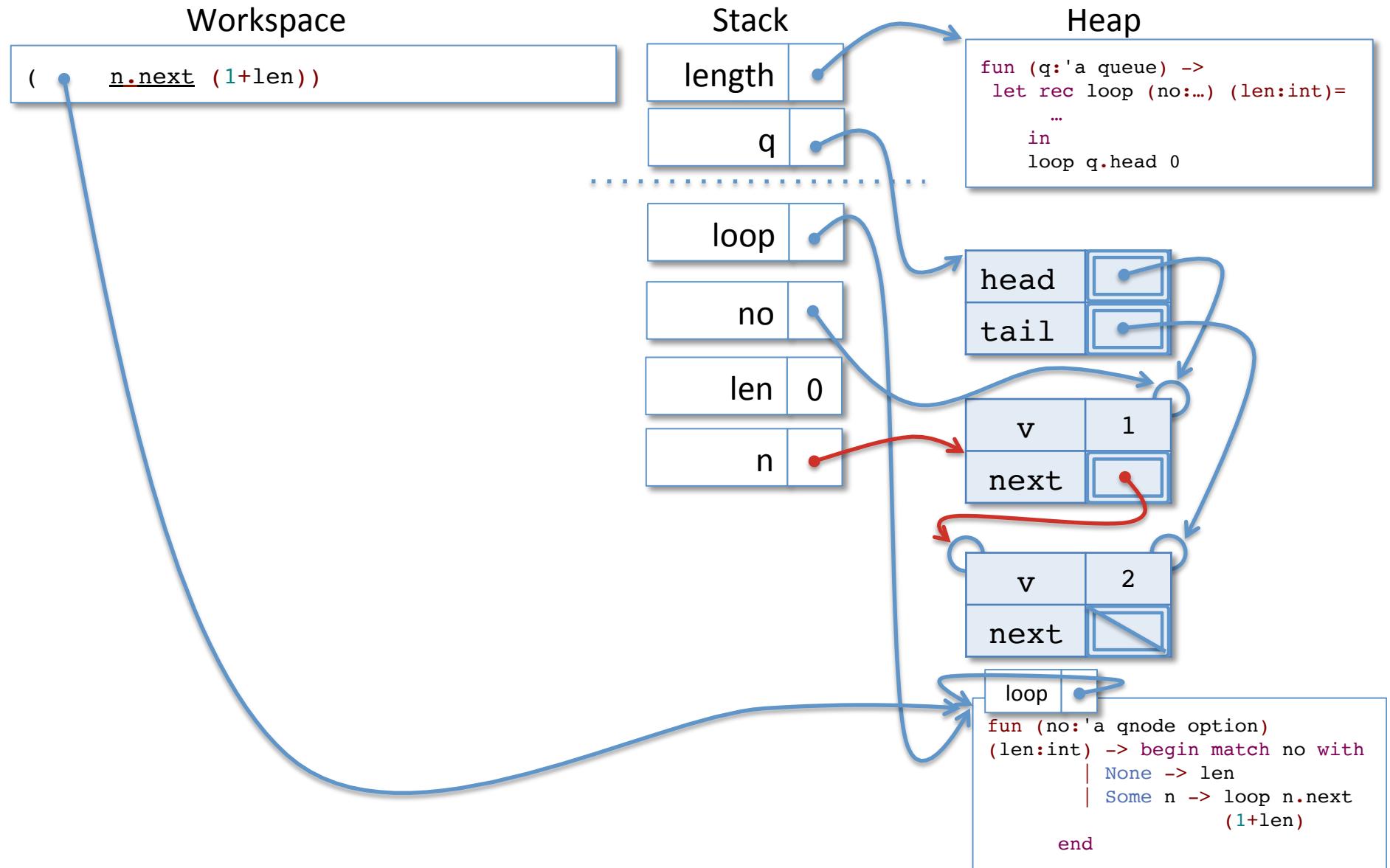
Tail Calls and Iterative length



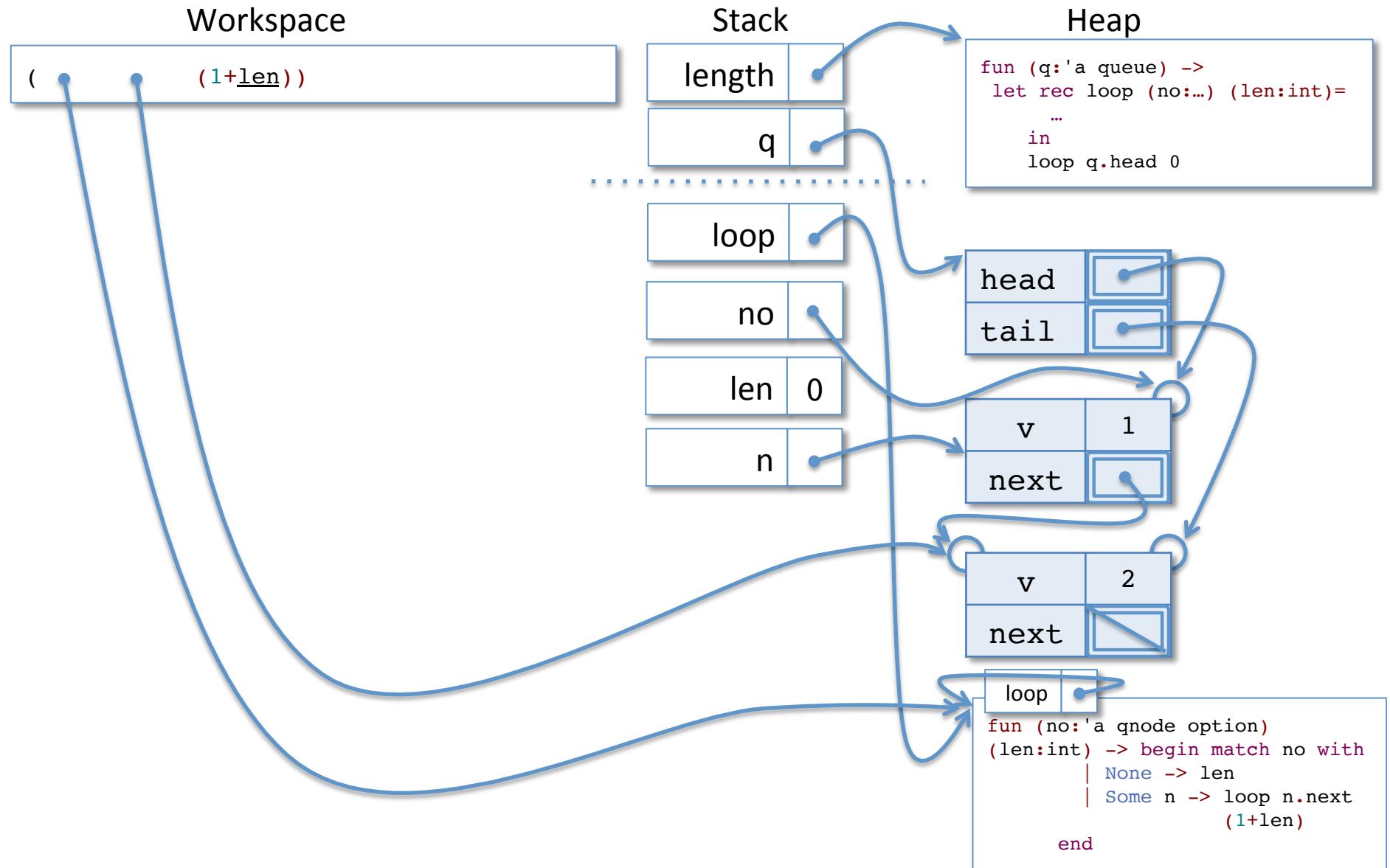
Tail Calls and Iterative length



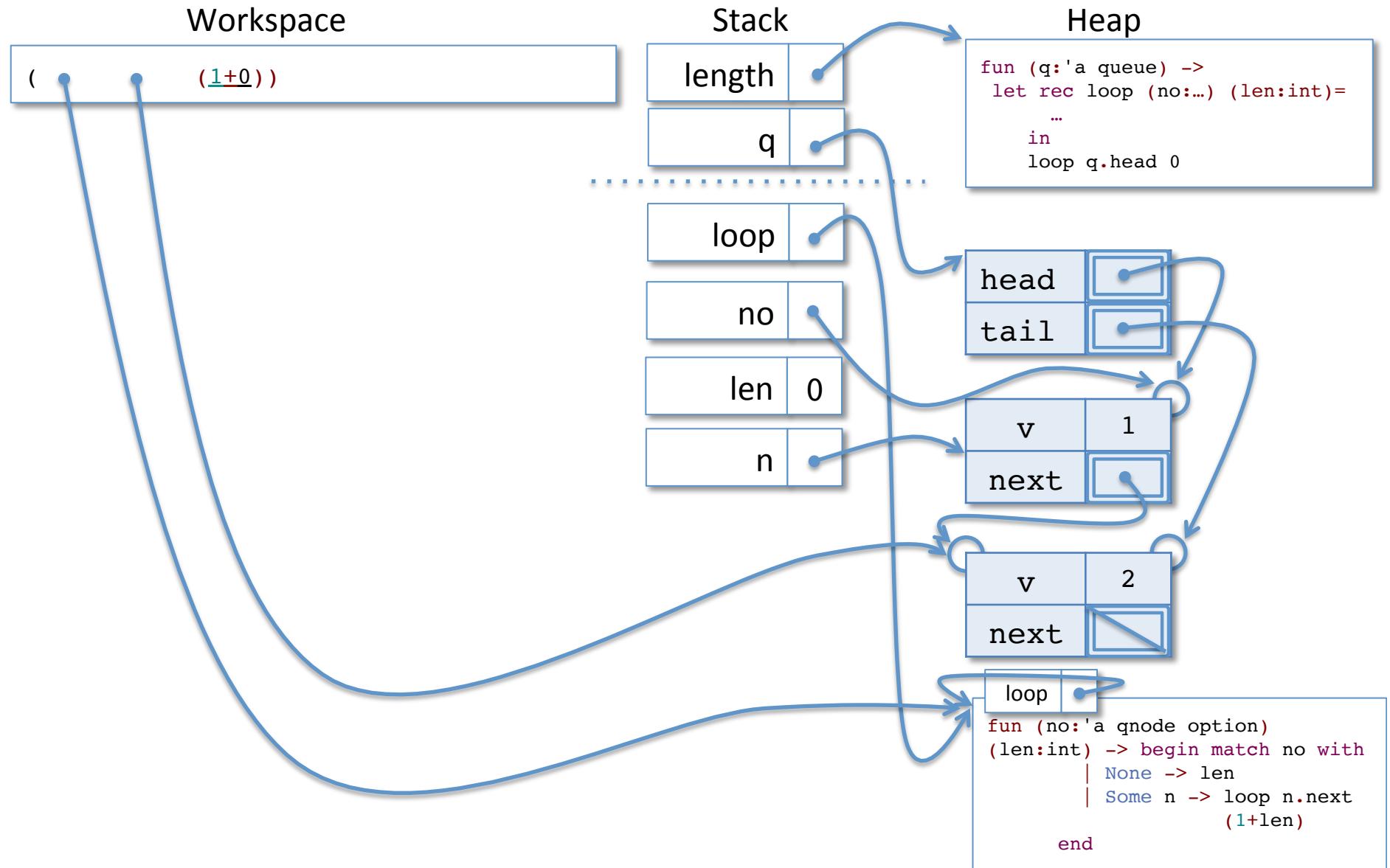
Tail Calls and Iterative length



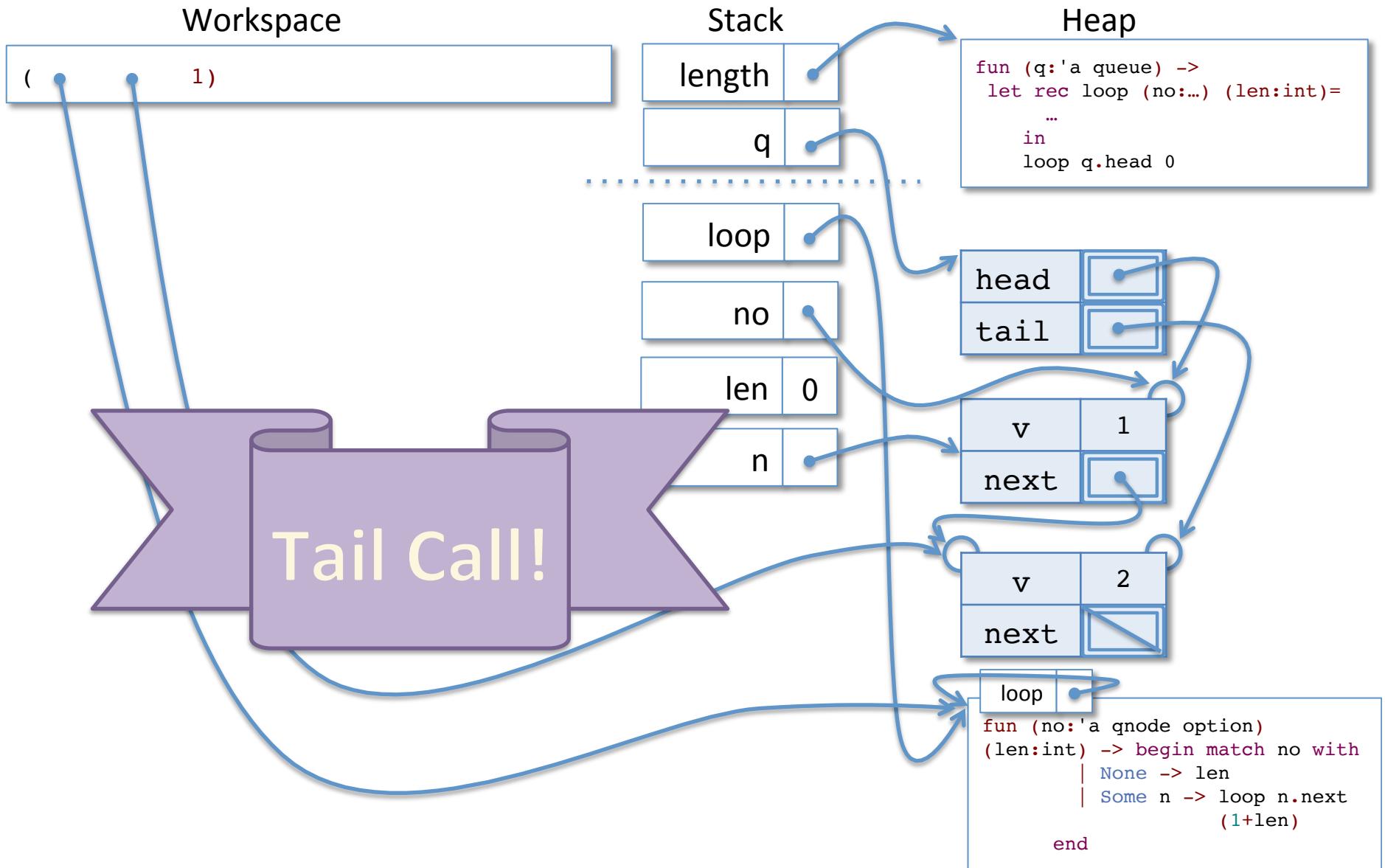
Tail Calls and Iterative length



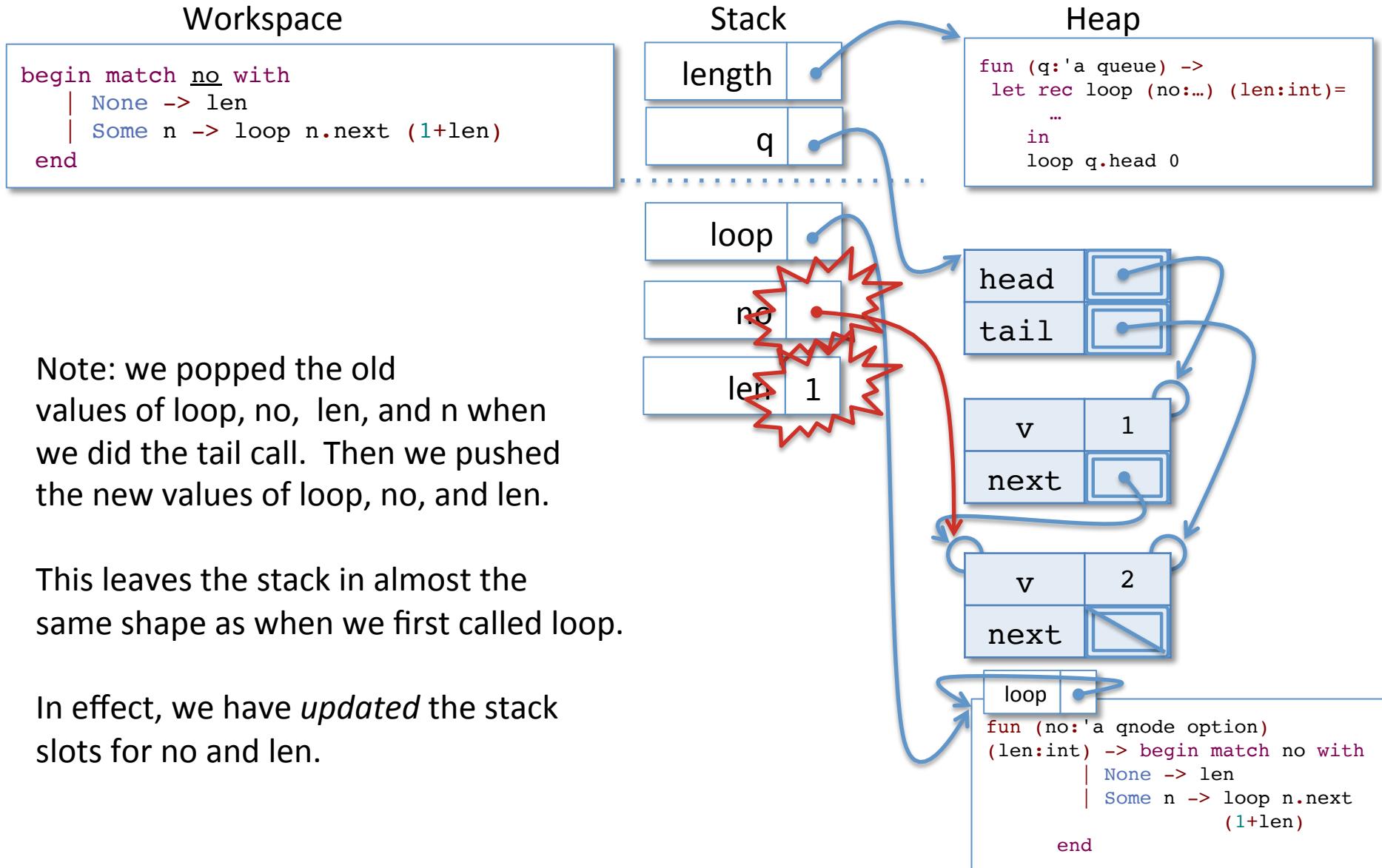
Tail Calls and Iterative length



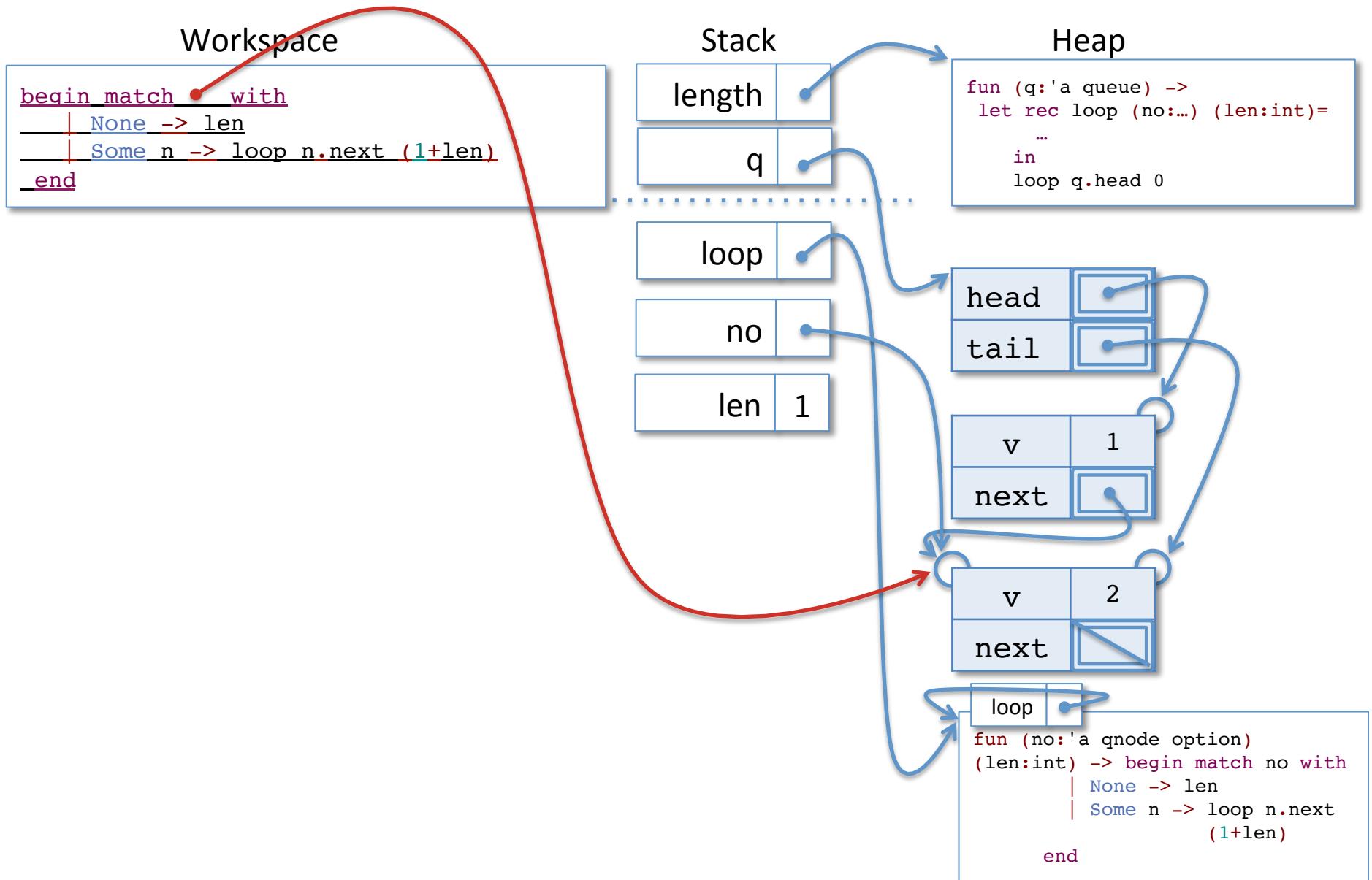
Tail Calls and Iterative length



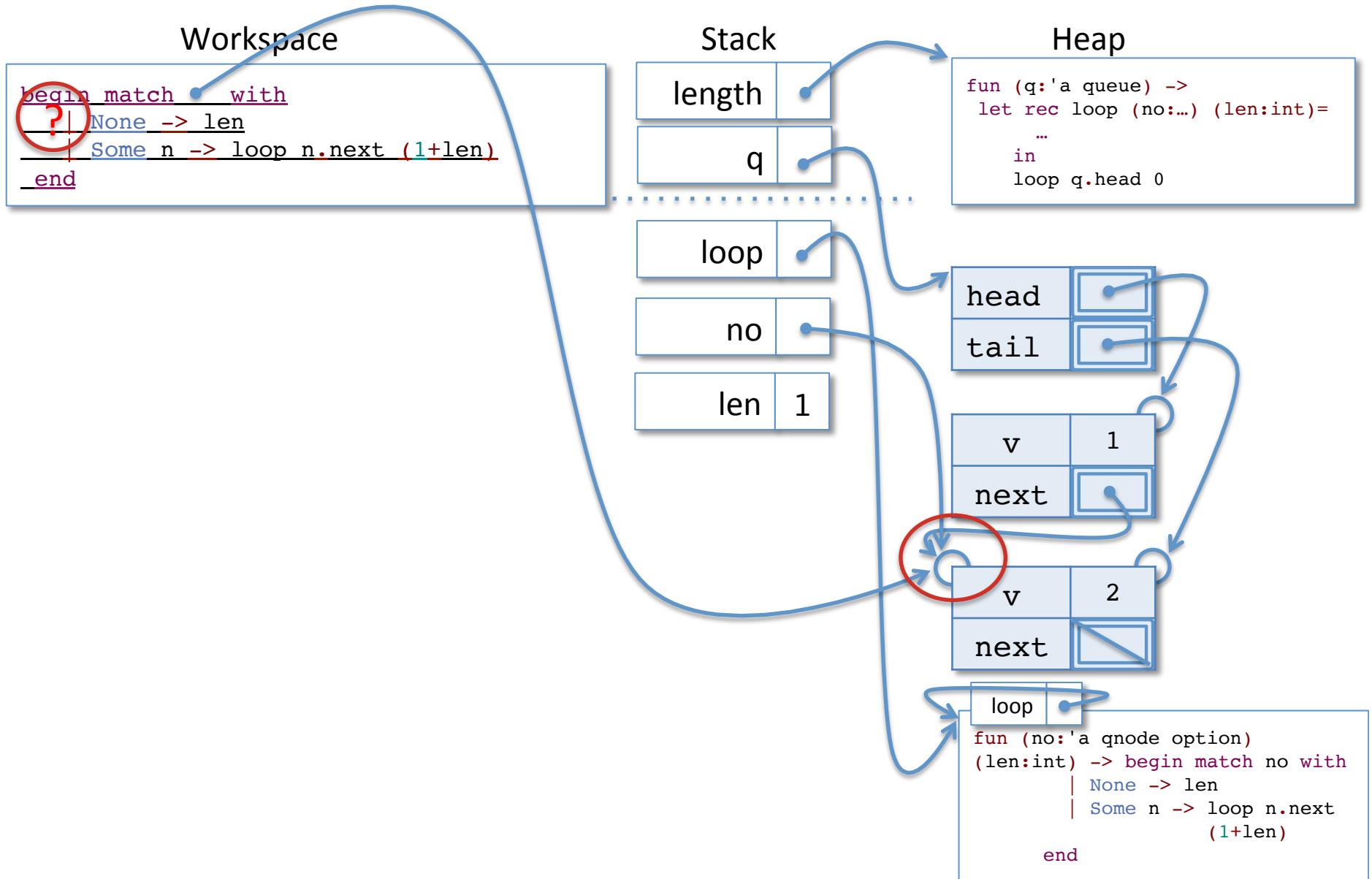
Tail Calls and Iterative length



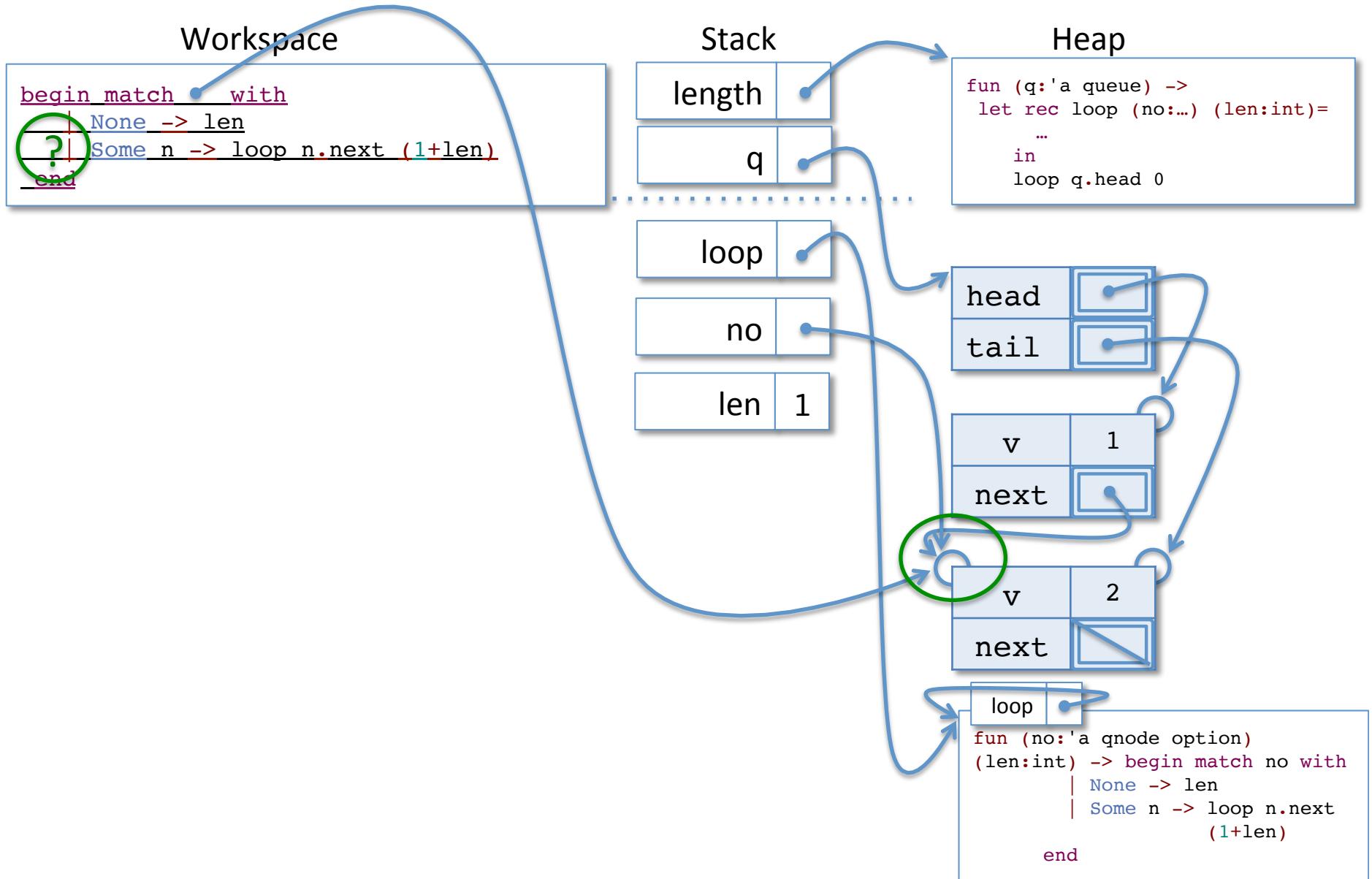
Tail Calls and Iterative length



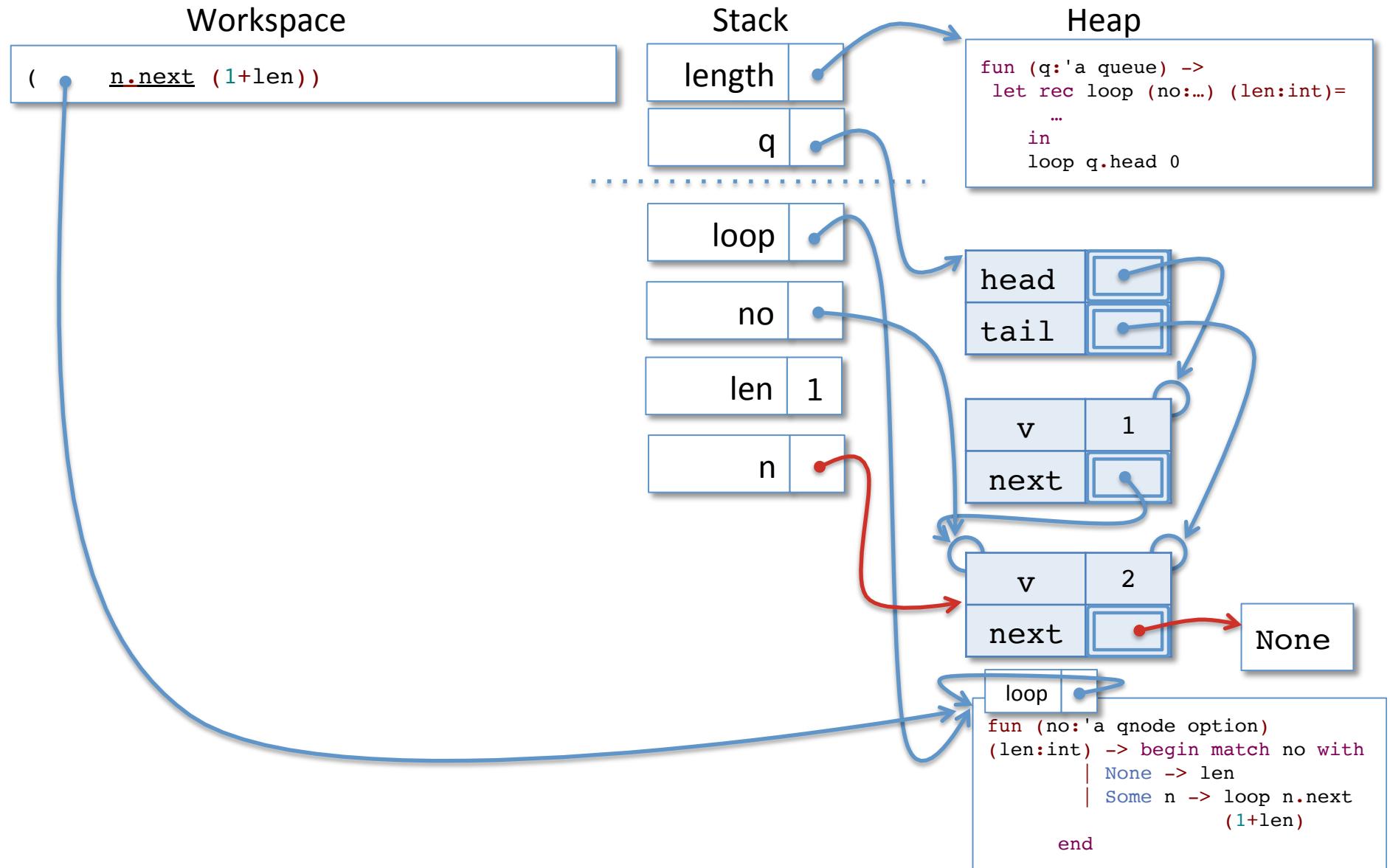
Tail Calls and Iterative length



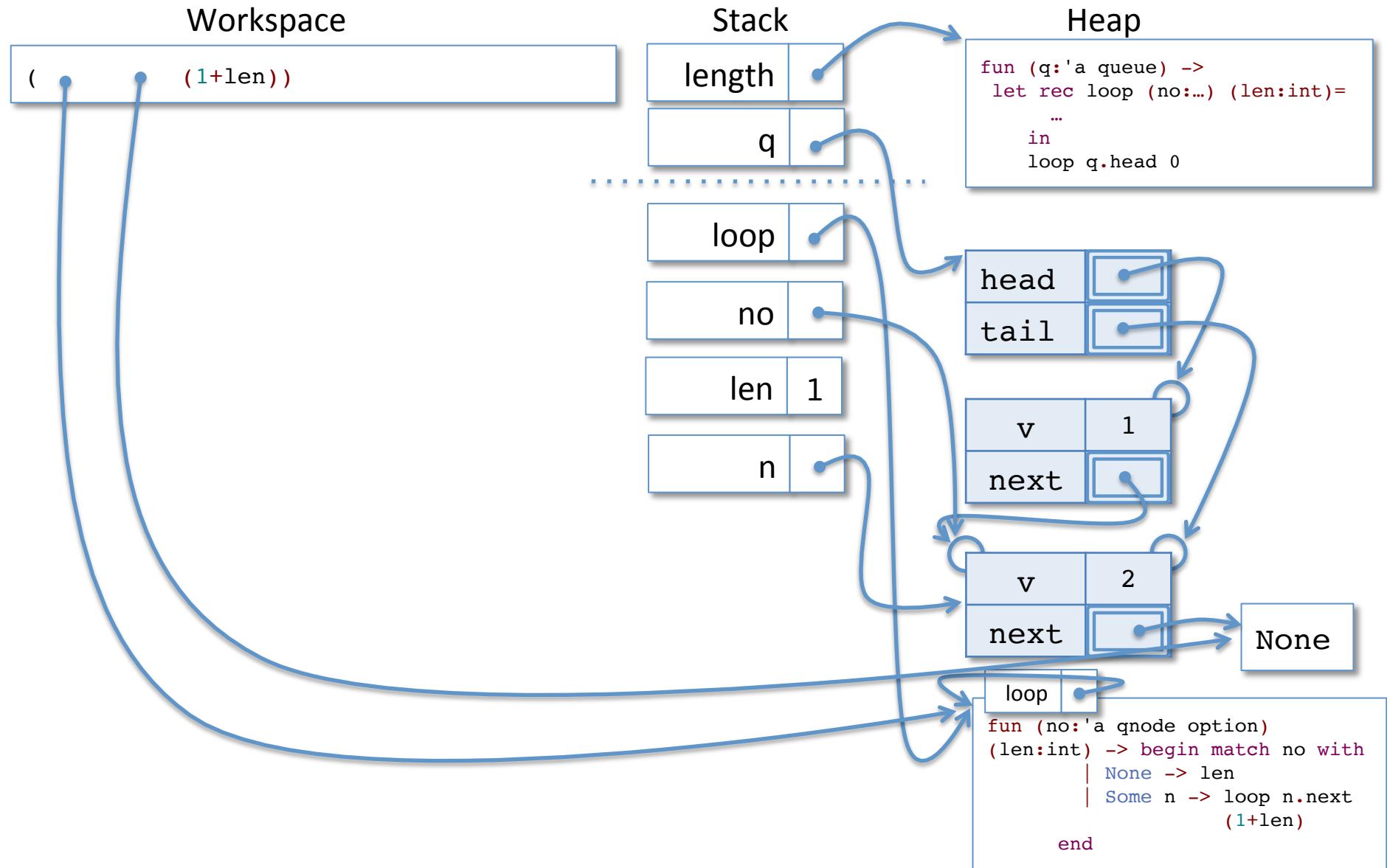
Tail Calls and Iterative length



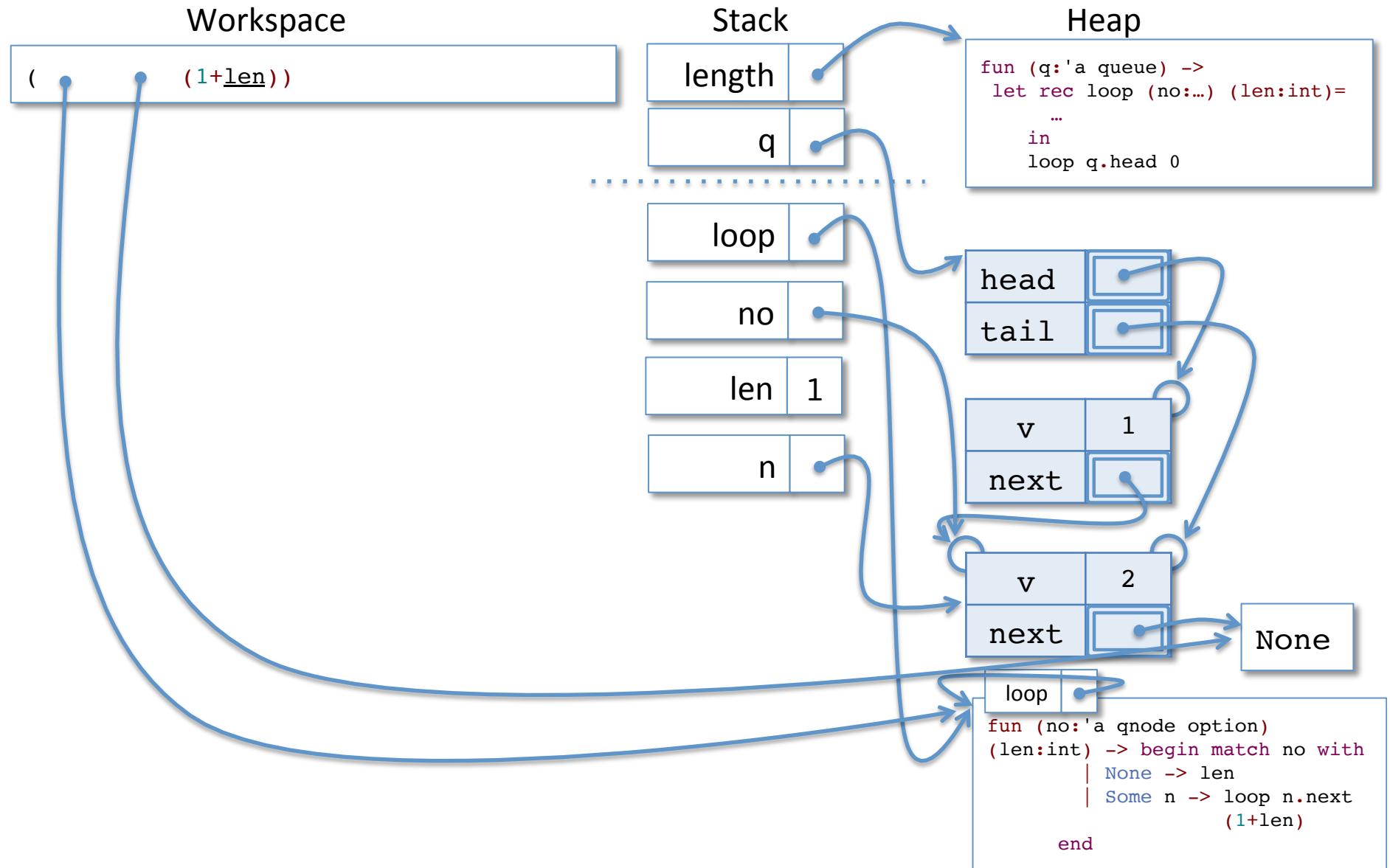
Tail Calls and Iterative length



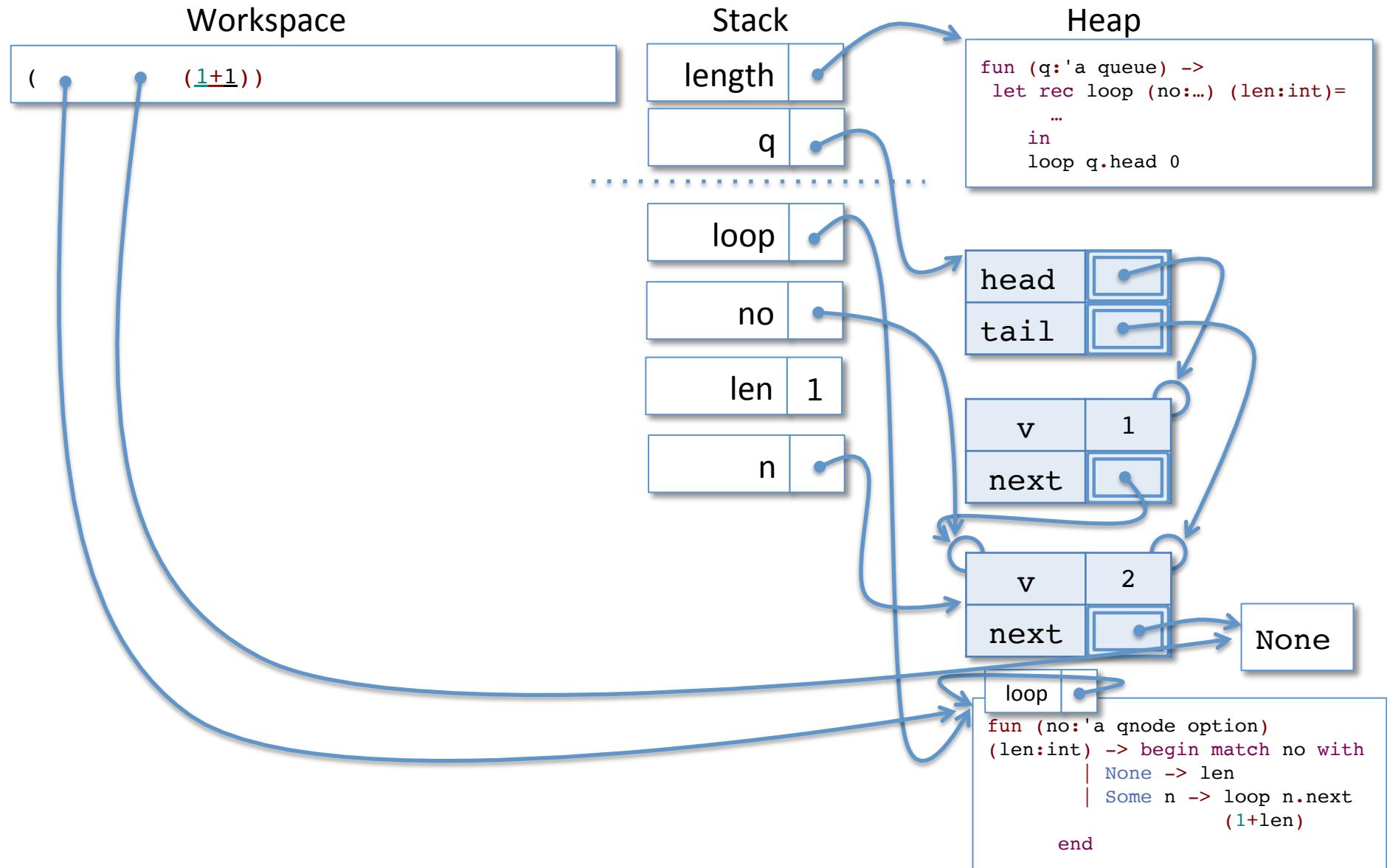
Tail Calls and Iterative length



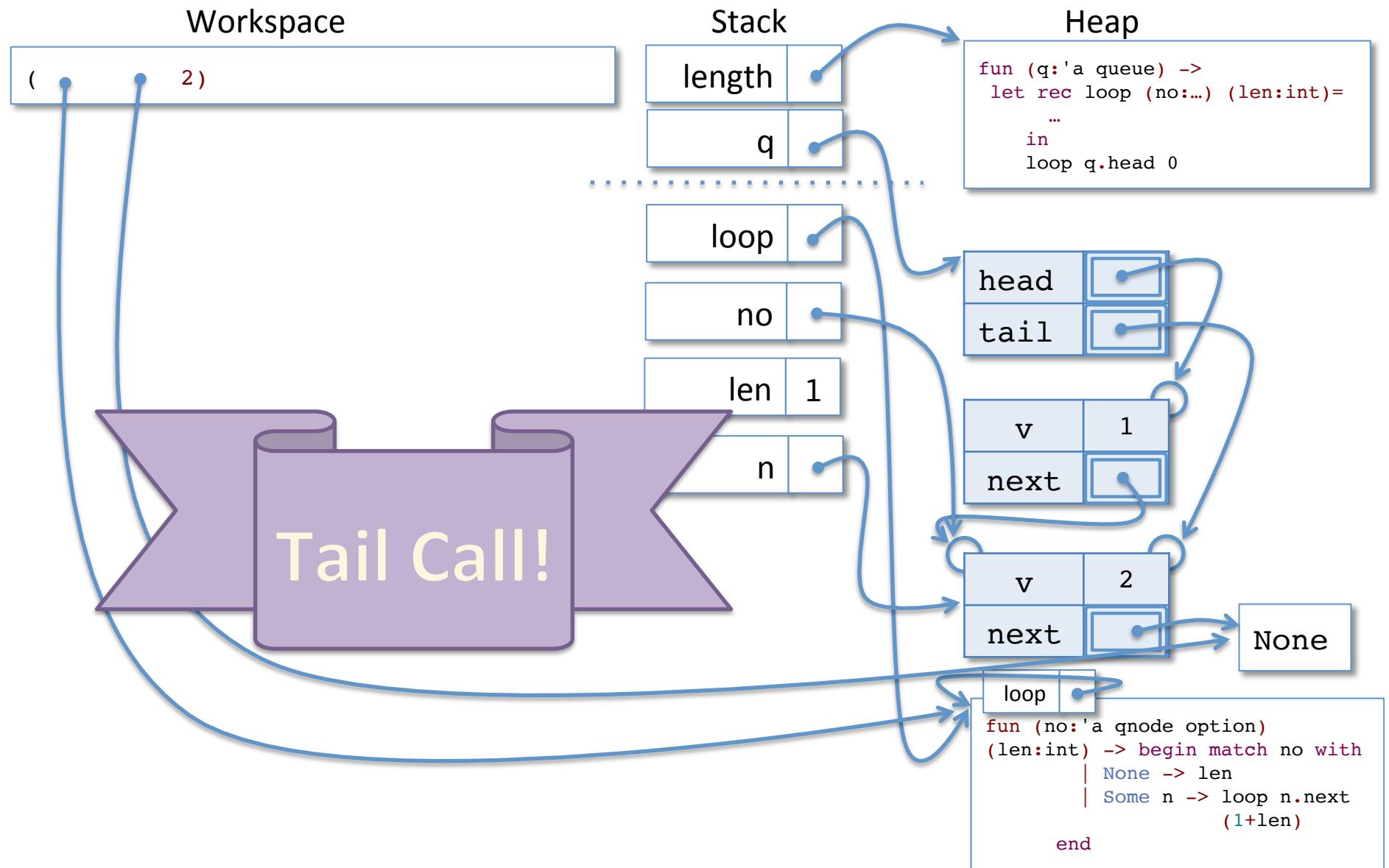
Tail Calls and Iterative length



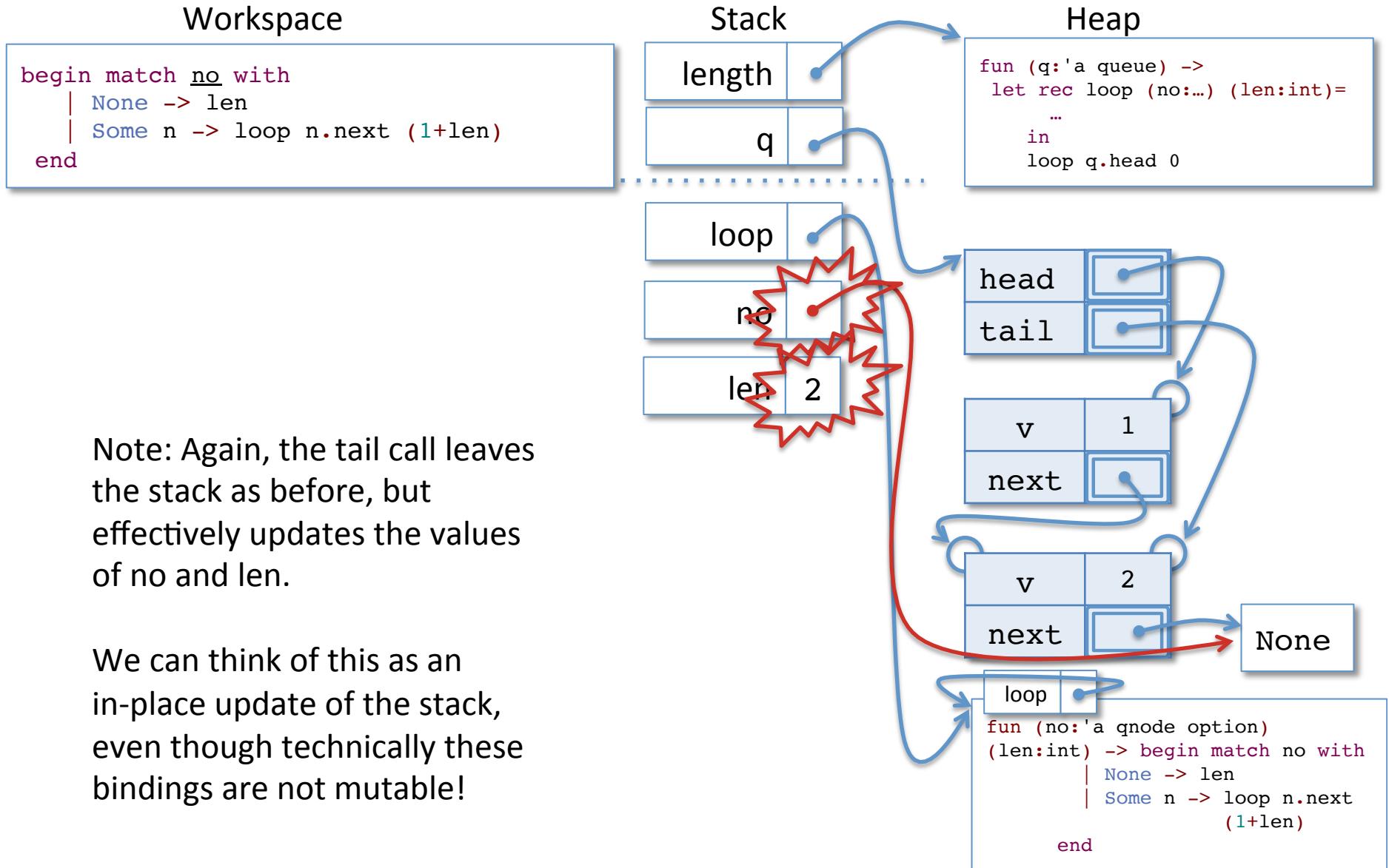
Tail Calls and Iterative length



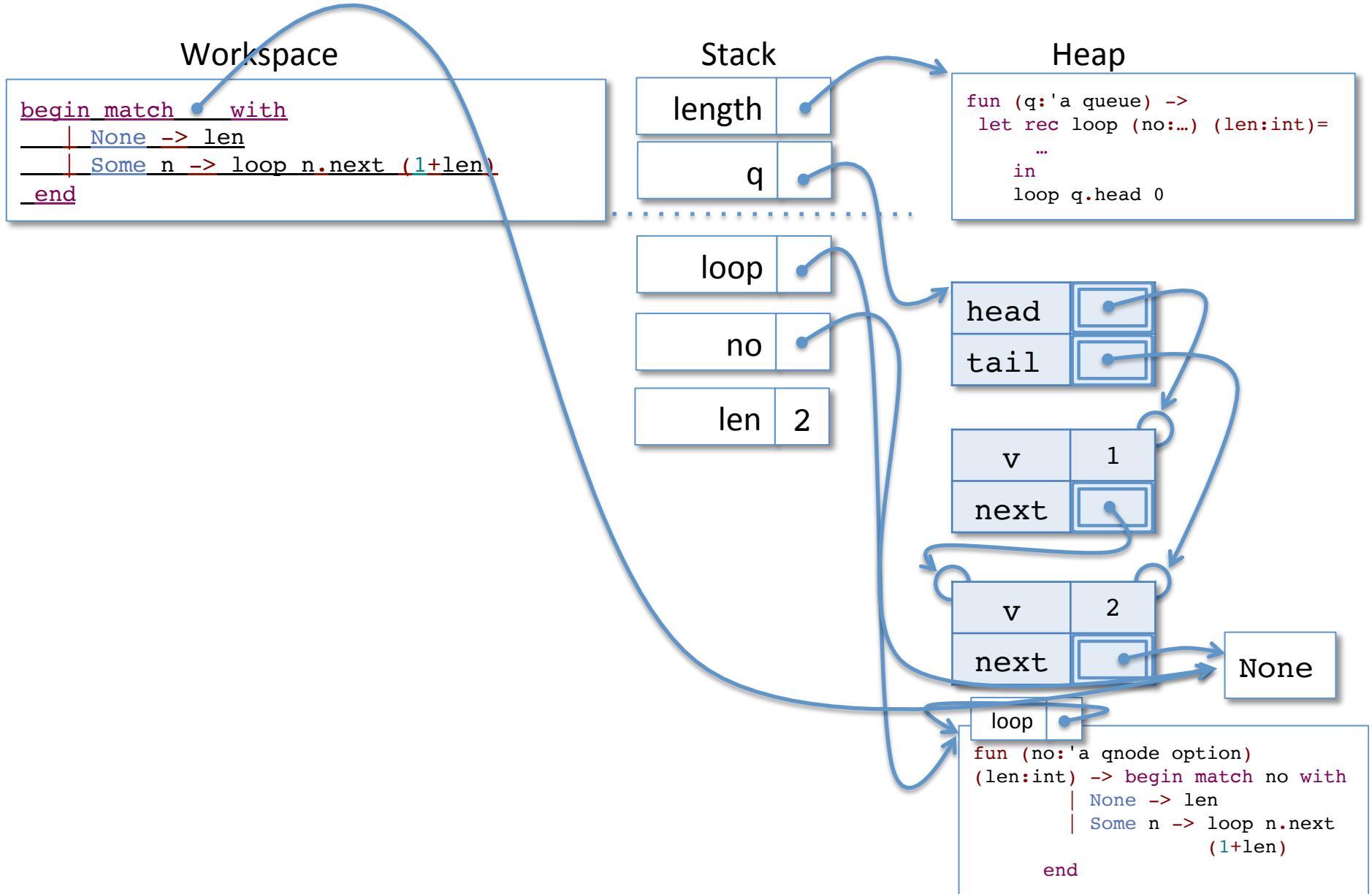
Tail Calls and Iterative length



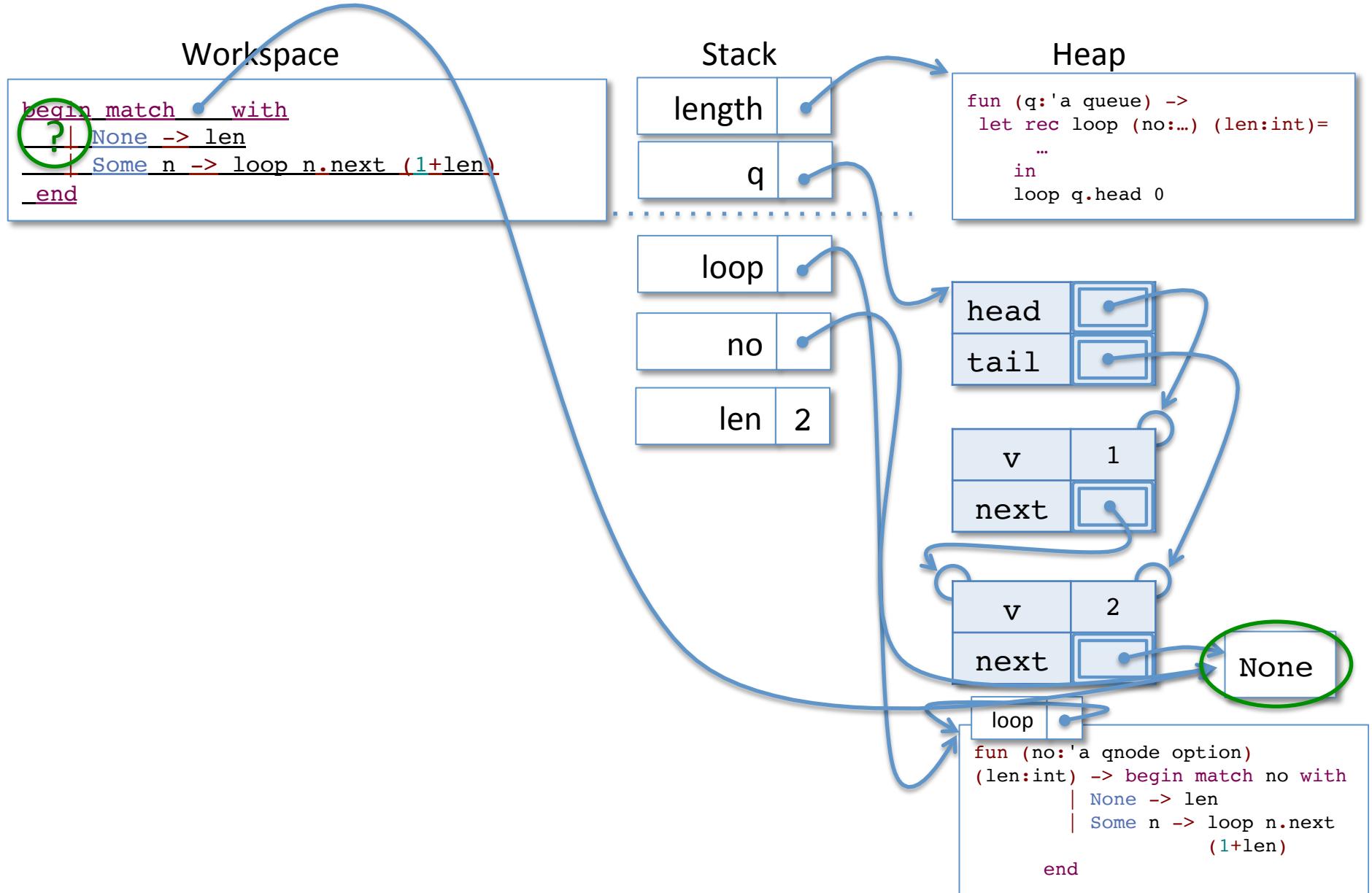
Tail Calls and Iterative length



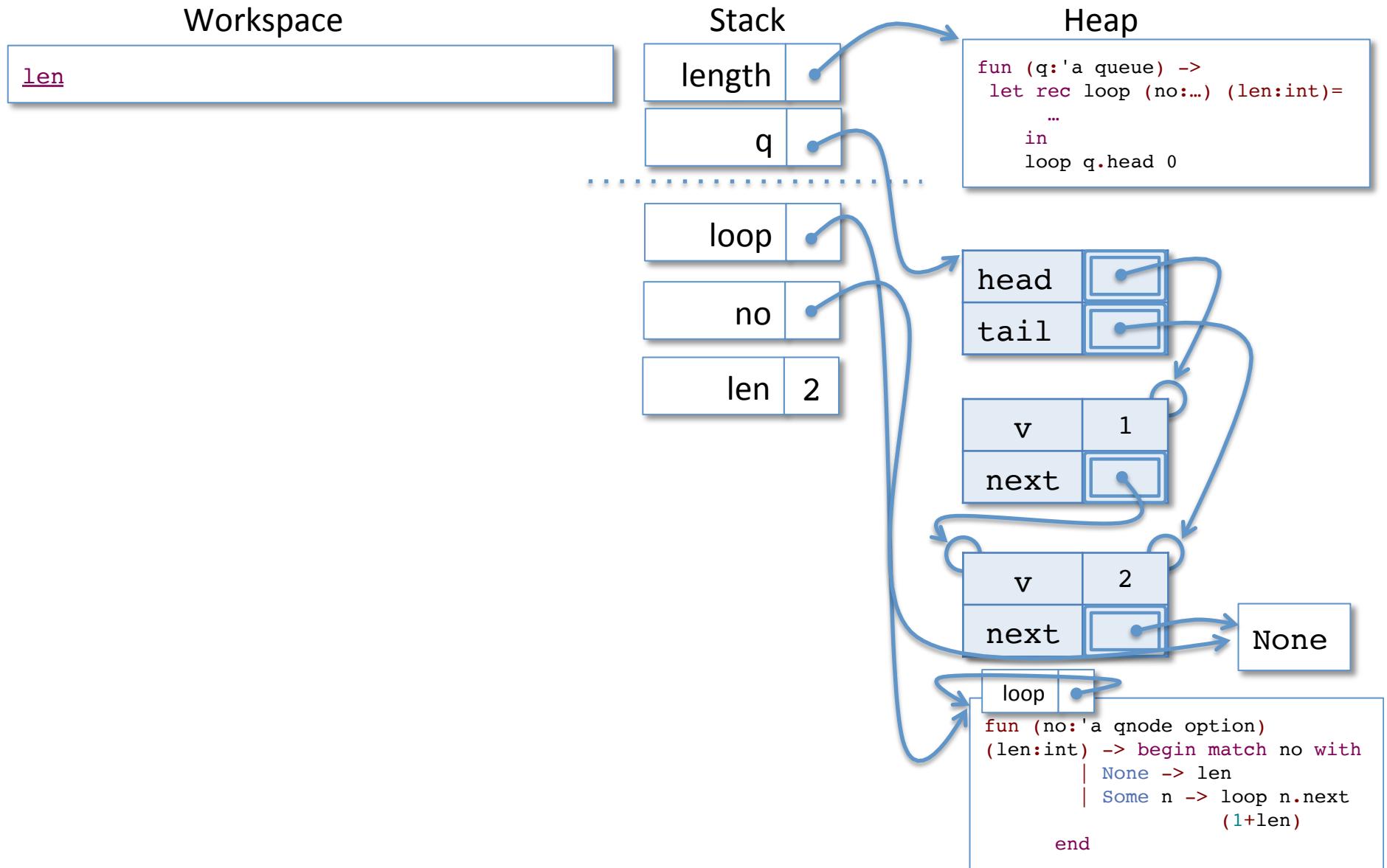
Tail Calls and Iterative length



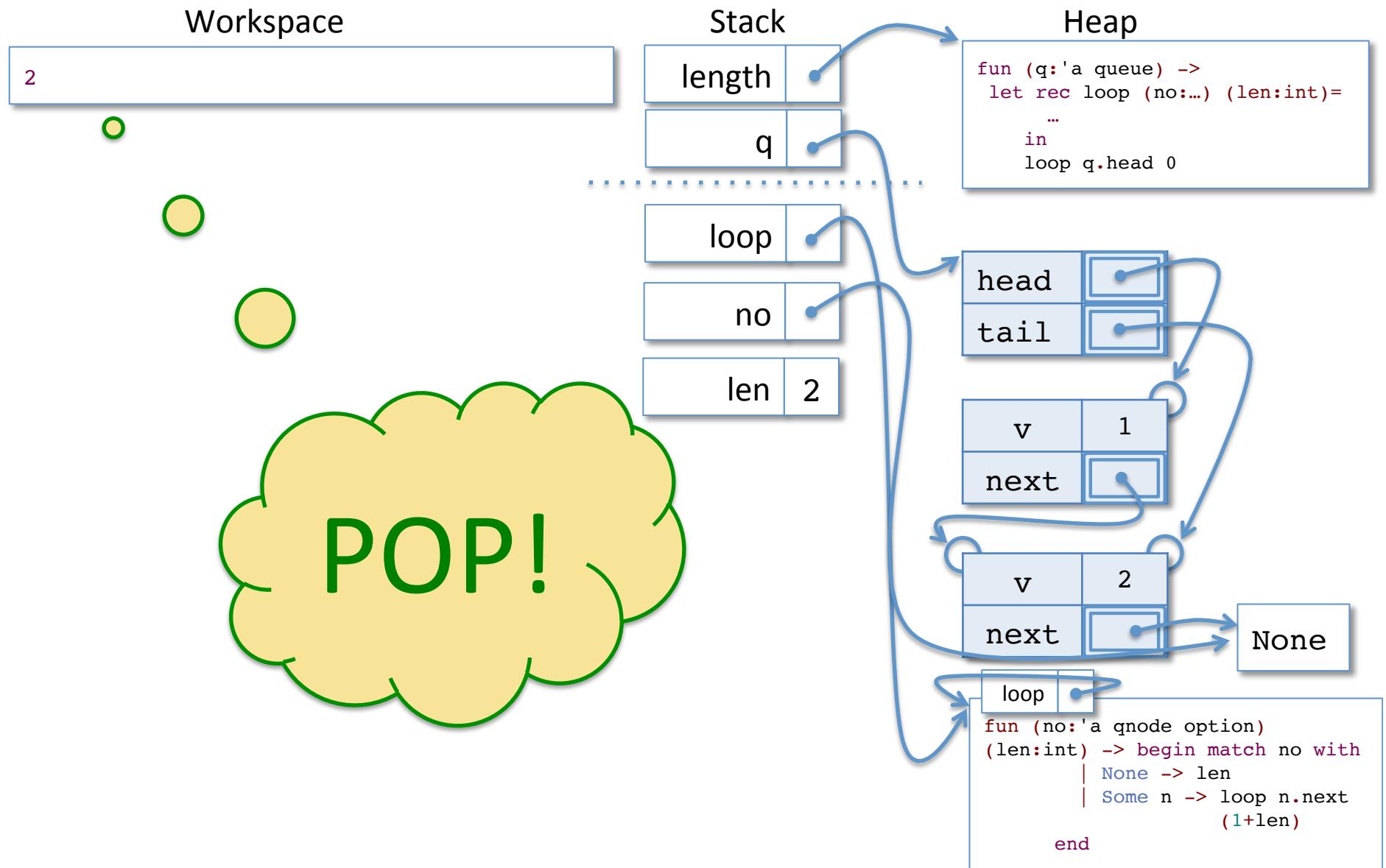
Tail Calls and Iterative length



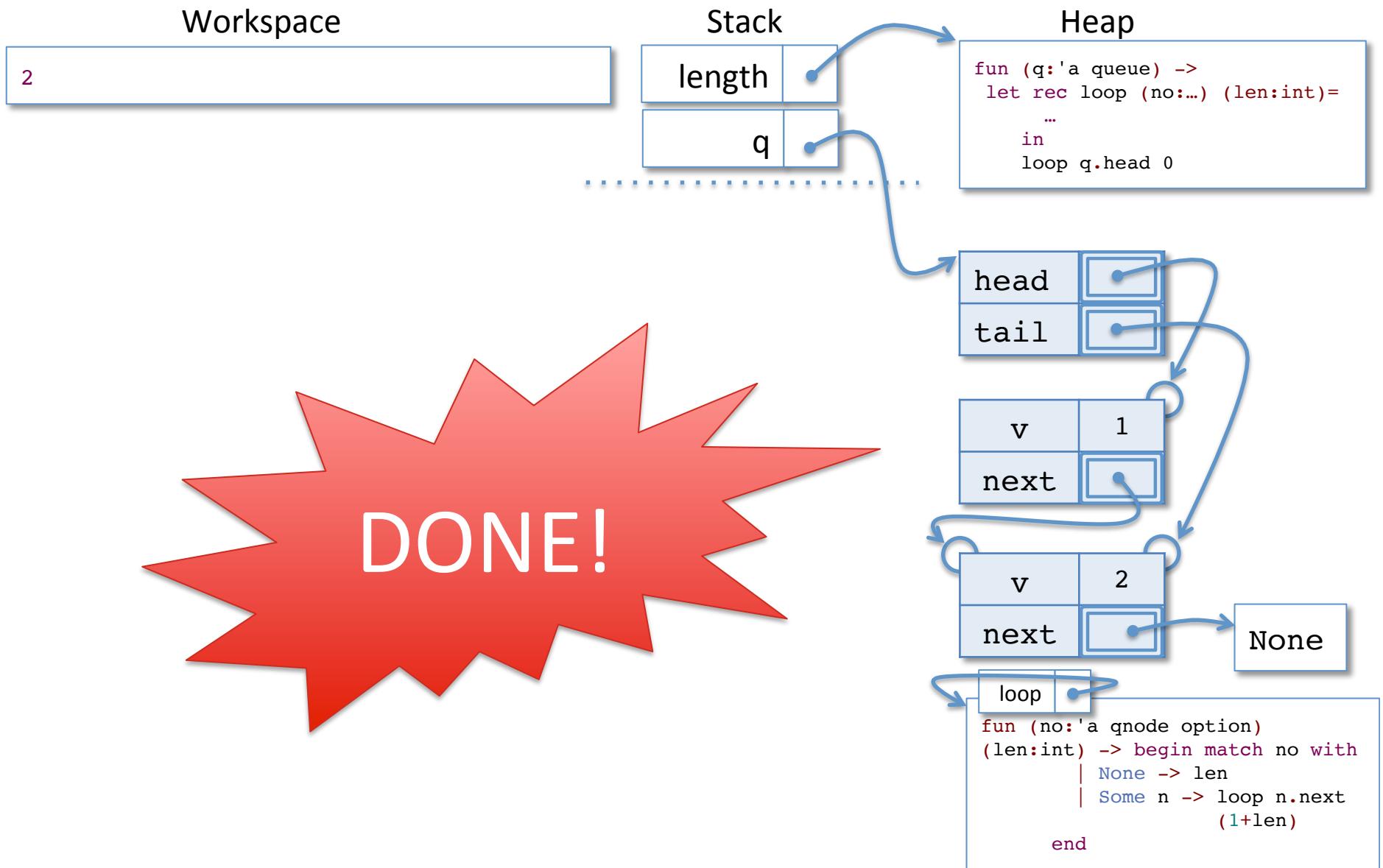
Tail Calls and Iterative length



Tail Calls and Iterative length



Tail Calls and Iterative length



Some Observations

- Tail call optimization lets the stack take only a fixed amount of space.
- The “recursive” call to loop effectively updates some of the stack bindings in place.
 - We can think of these bindings as the *state* being modified by each iteration of the loop.
- These two properties are the essence of iteration.
 - They are the difference between general recursion and iteration

What happens when you run this function on a (valid) queue containing 2 elements?

```
let f (q:'a queue) : int =
    let rec loop (qn:'a qnode option) : int =
        begin match qn with
            | None -> 0
            | Some n -> 1 + loop qn
        end
    in loop q.head
```

1. The value 2 is returned
2. The value 0 is returned
3. StackOverflow
4. Your program hangs

ANSWER: 3

What happens when you run this function on a (valid) queue containing 2 elements?

```
let f (q:'a queue) : int =
  let rec loop (qn:'a qnode option) (len:int) : int =
    begin match qn with
      | None -> len
      | Some n -> loop qn (len + 1)
    end
  in loop q.head 0
```

1. The value 2 is returned
2. The value 0 is returned
3. StackOverflow
4. Your program hangs

ANSWER: 4

Infinite Loops

```
(* Accidentally go into an infinite loop... *)
let accidental_infinite_loop (q:'a queue) : int =
  let rec loop (qn:'a qnode option) (len:int) : int =
    begin match qn with
      | None -> len
      | Some n -> loop qn (len + 1)
    end
  in loop q.head 0
```

- This program will go into an infinite loop.
- Unlike a non-tail-recursive program, which uses some space on each recursive call, there is no resource being exhausted, so the program will “silently diverge” and simply never produce an answer...

Programming Languages and Techniques (CIS120)

Lecture 16

October 9th 2017

Queues: Iteration & Tail Recursion

Chapter 16

Announcements

- Homework 4
 - due TOMORROW October 10th
- Homework 5: GUI Programming
 - Available soon: Due October 24th
- Midterm 1
 - *October 13th in Class*
 - Where? Last Names:
 - A – M Leidy Labs 10 (Here)
 - N – Z Meyerson Hall B1
 - Covers lecture material through Chapter 13
 - Review materials (old exams) on course website
- Review Session:
 - *Wednesday, Oct. 11th 6:00-8:00pm, Towne 100*

Mutable Queues: Queue Length

working with singly linked data structures

Queue Length

- Suppose we want to extend the interface with a length function:

```
module type QUEUE =
sig
  (* type of the data structure *)
  type 'a queue
  ...
  (* Get the length of the queue *)
  val length : 'a queue -> int
end
```

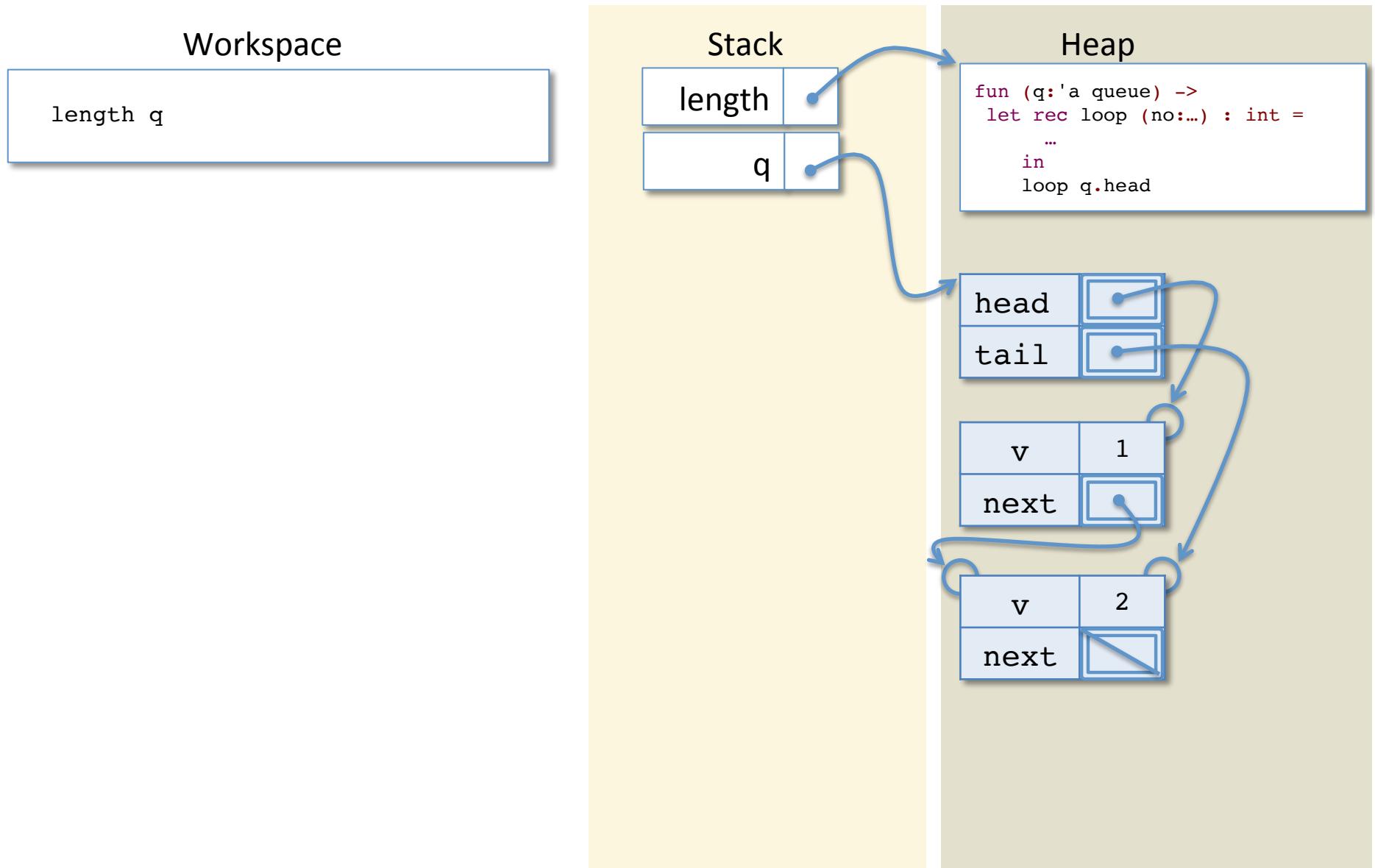
- How can we implement it?

length (recursively)

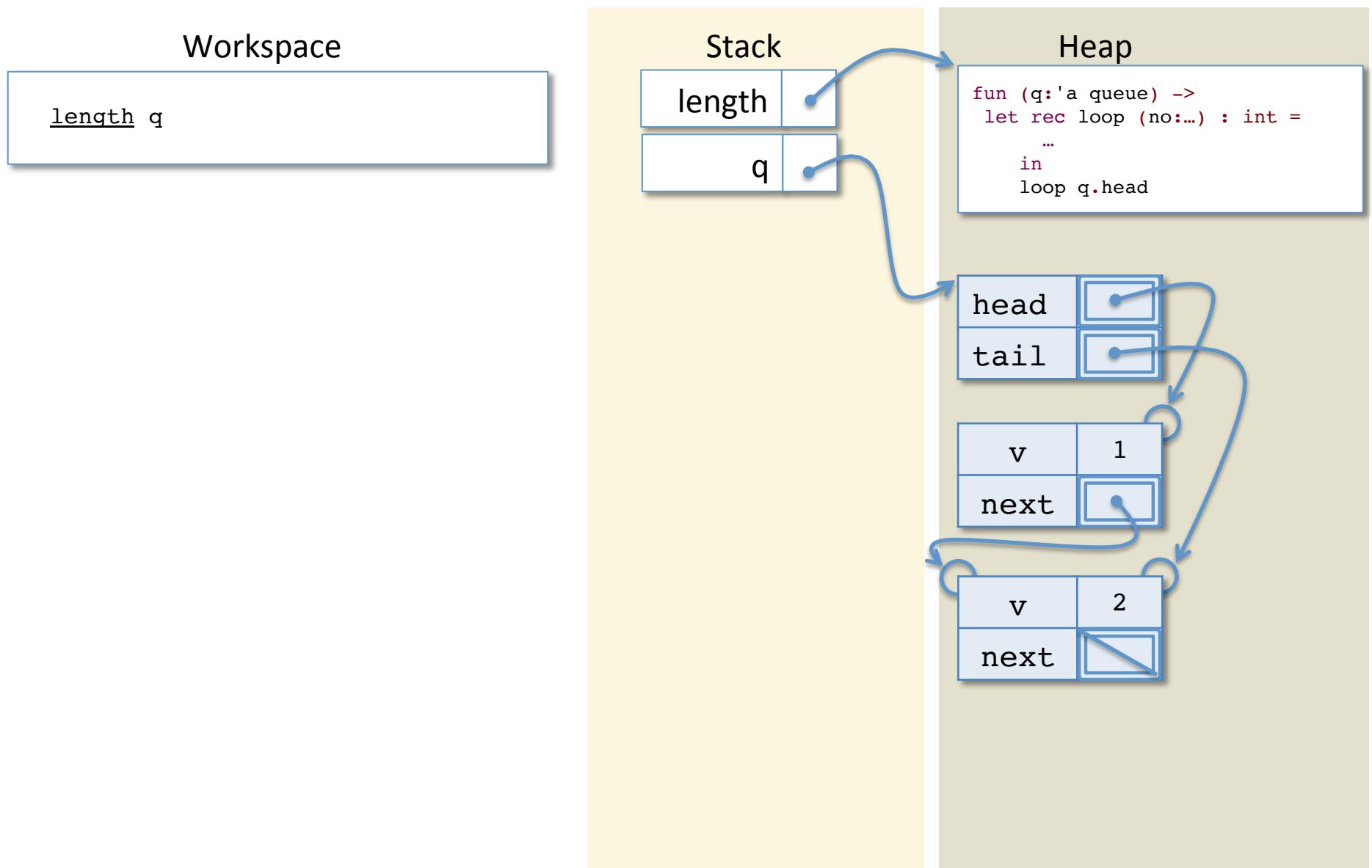
```
(* Calculate the length of the queue recursively *)
let length (q:'a queue) : int =
  let rec loop (no: 'a qnode option) : int =
    begin match no with
      | None -> 0
      | Some n -> 1 + (loop n.next)
    end
  in
  loop q.head
```

- This code for `length` uses a helper function, `loop`:
 - the correctness depends crucially on the queue invariant
 - what happens if we pass in a bogus `q` that is cyclic?
- The height of the ASM stack is proportional to the length of the queue
 - That seems inefficient... why should it take so much space?

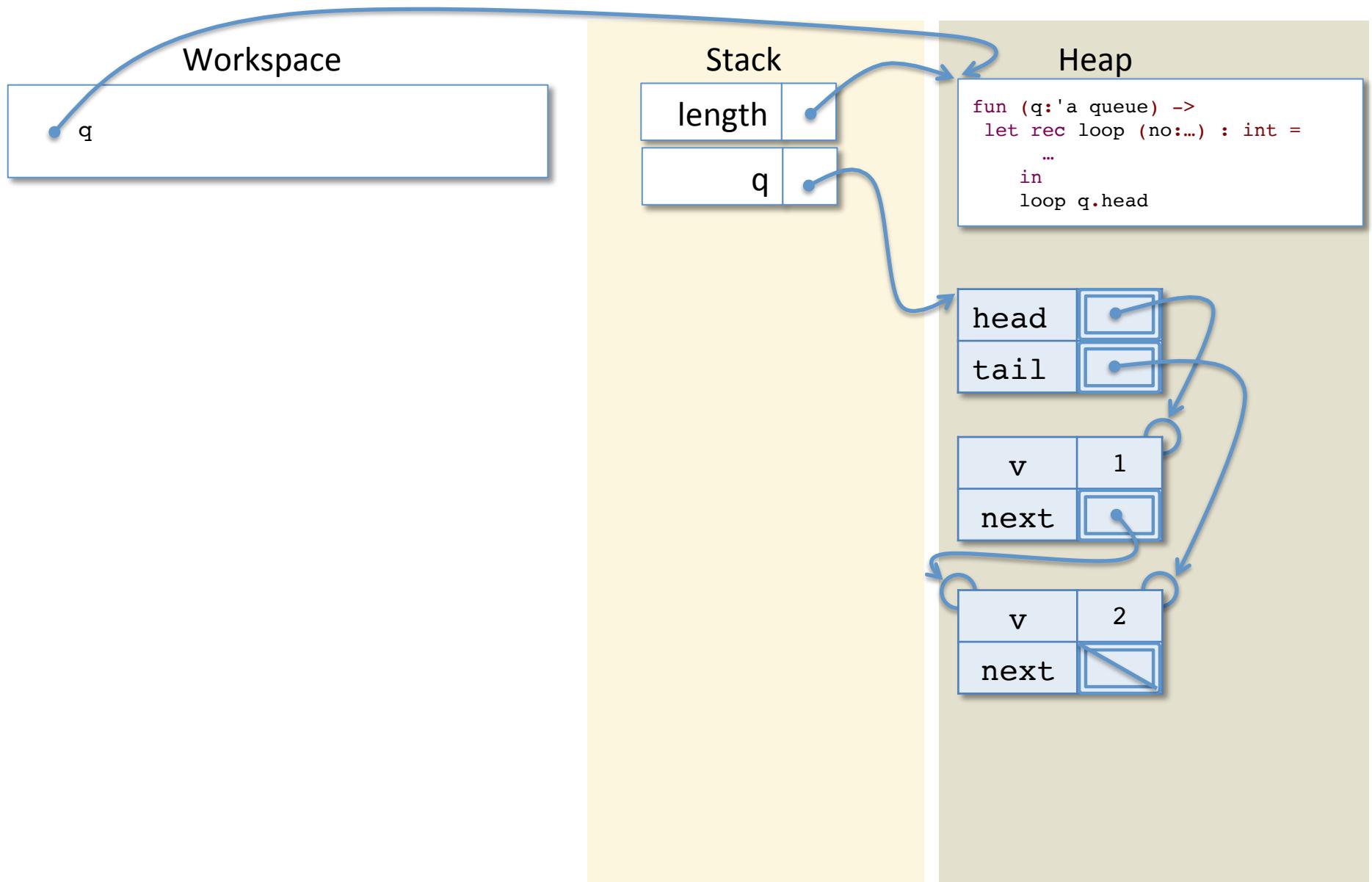
Evaluating length



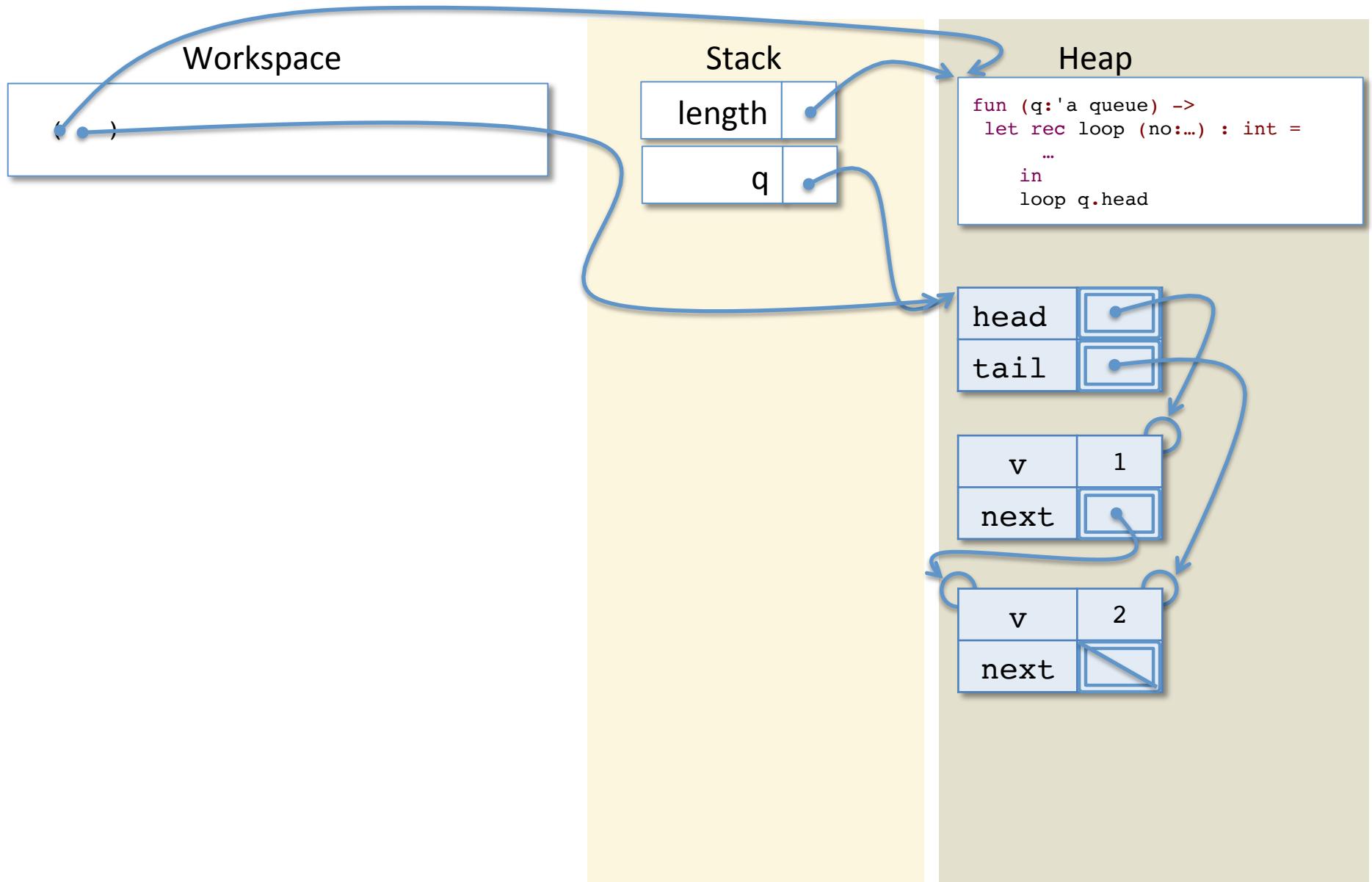
Evaluating length



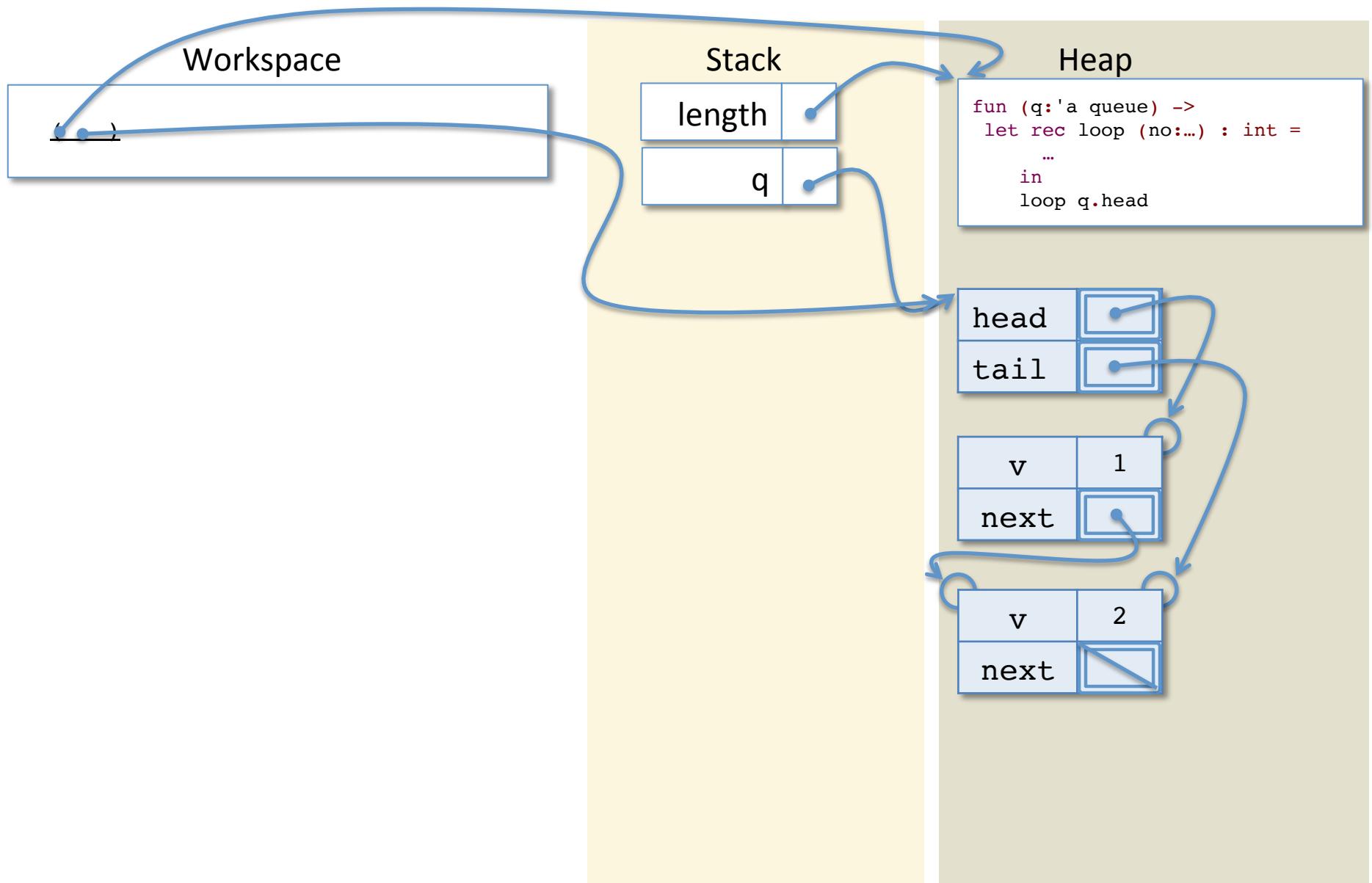
Evaluating length



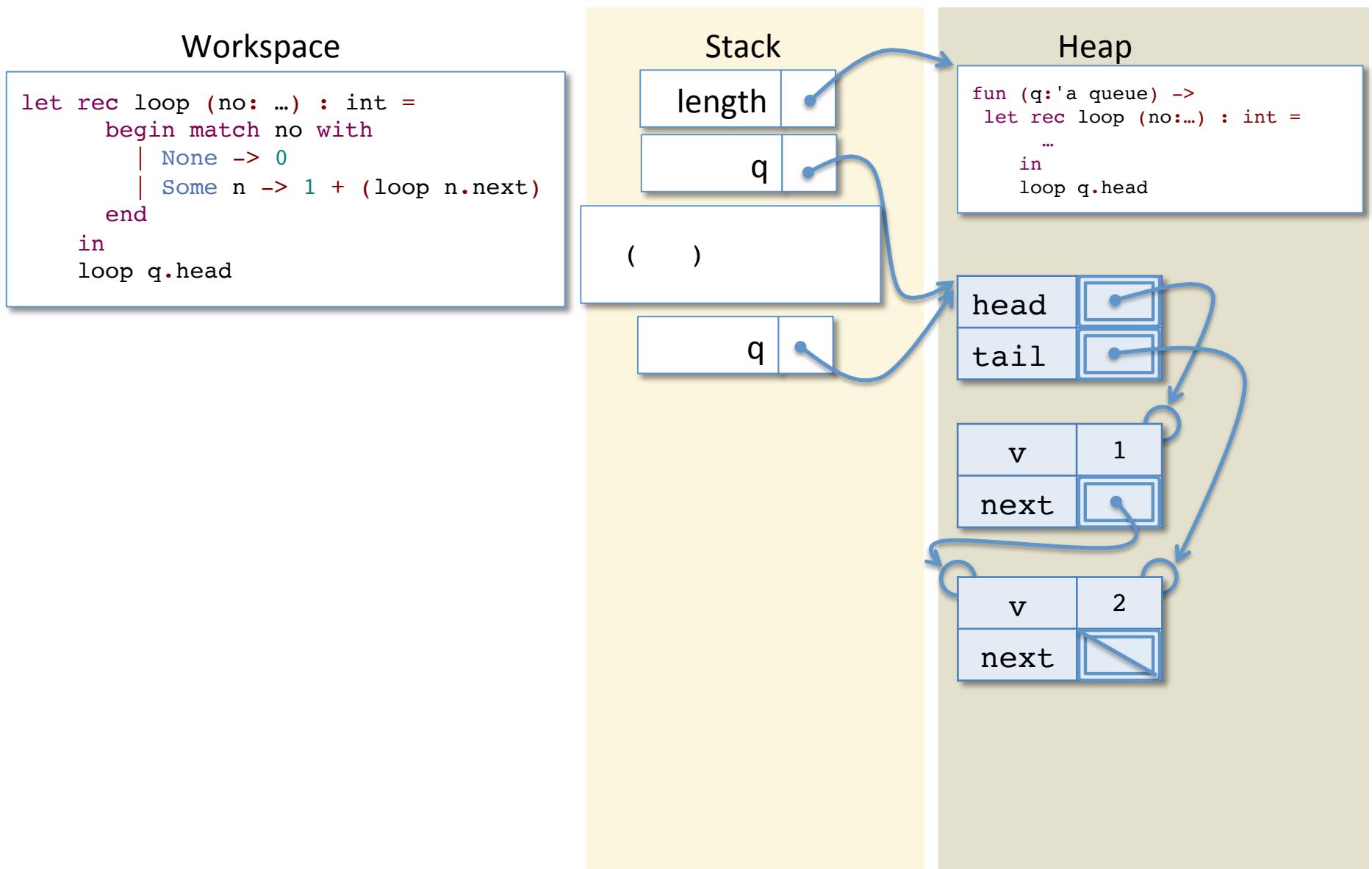
Evaluating length



Evaluating length



Evaluating length

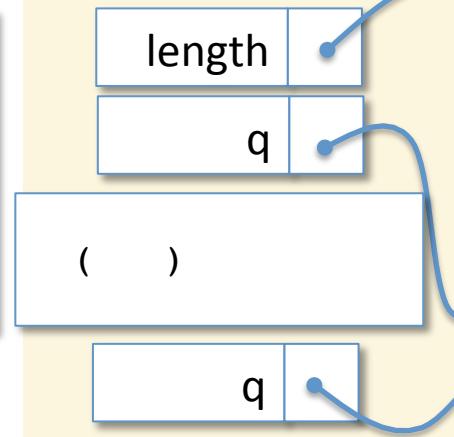


Evaluating length

Workspace

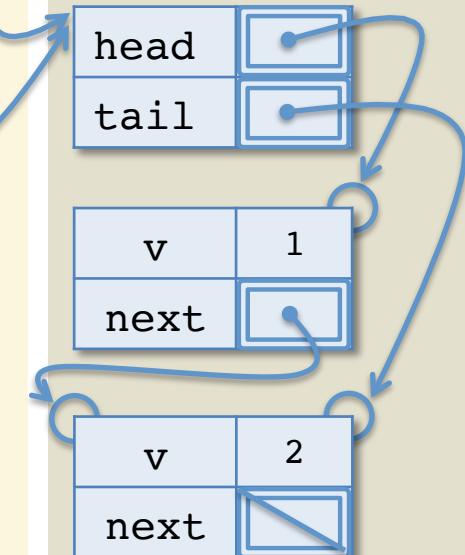
```
let loop = fun (no: ...) ->
  begin match no with
    | None -> 0
    | Some n -> 1 + (loop n.next)
  end
in
loop q.head
```

Stack

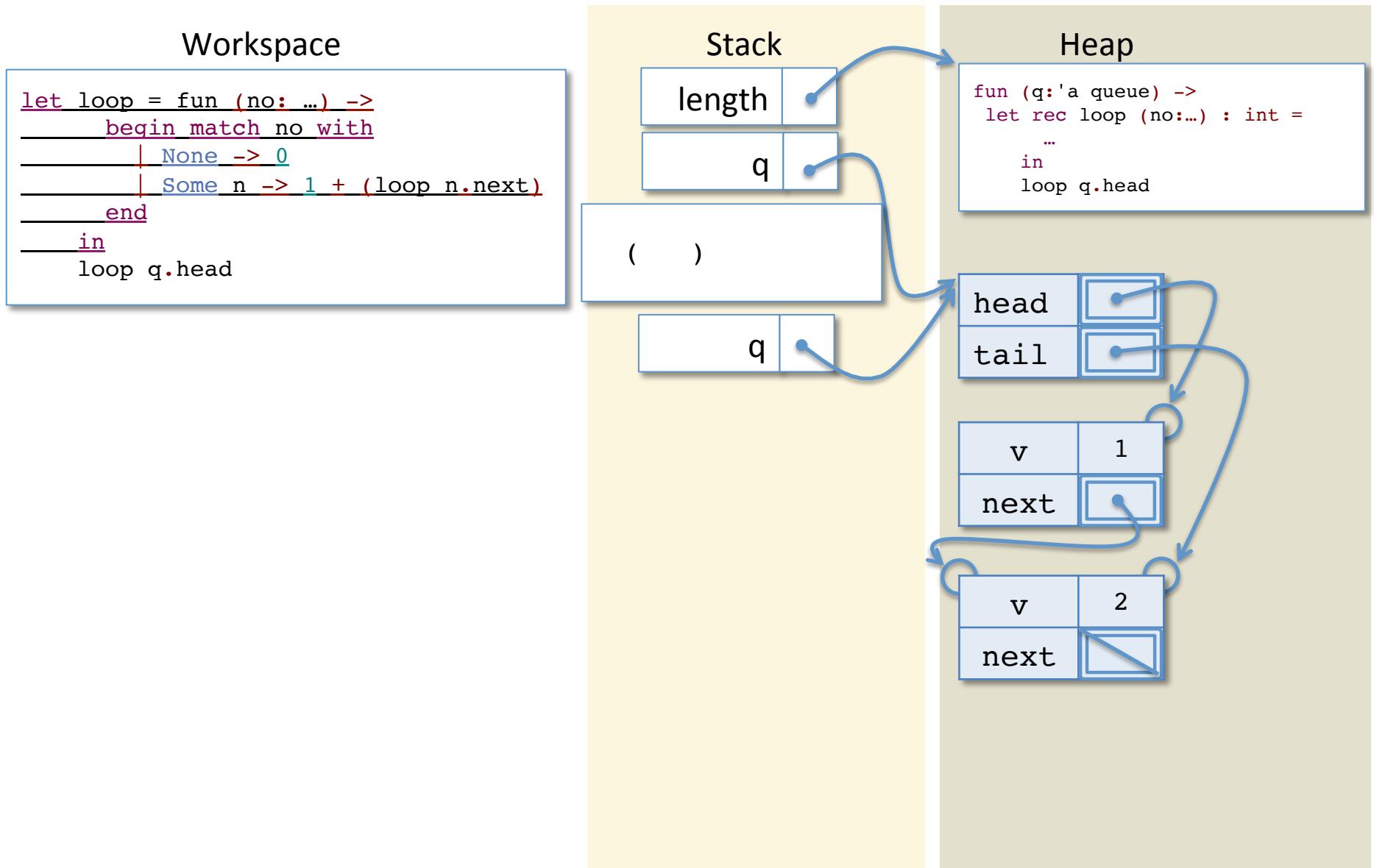


Heap

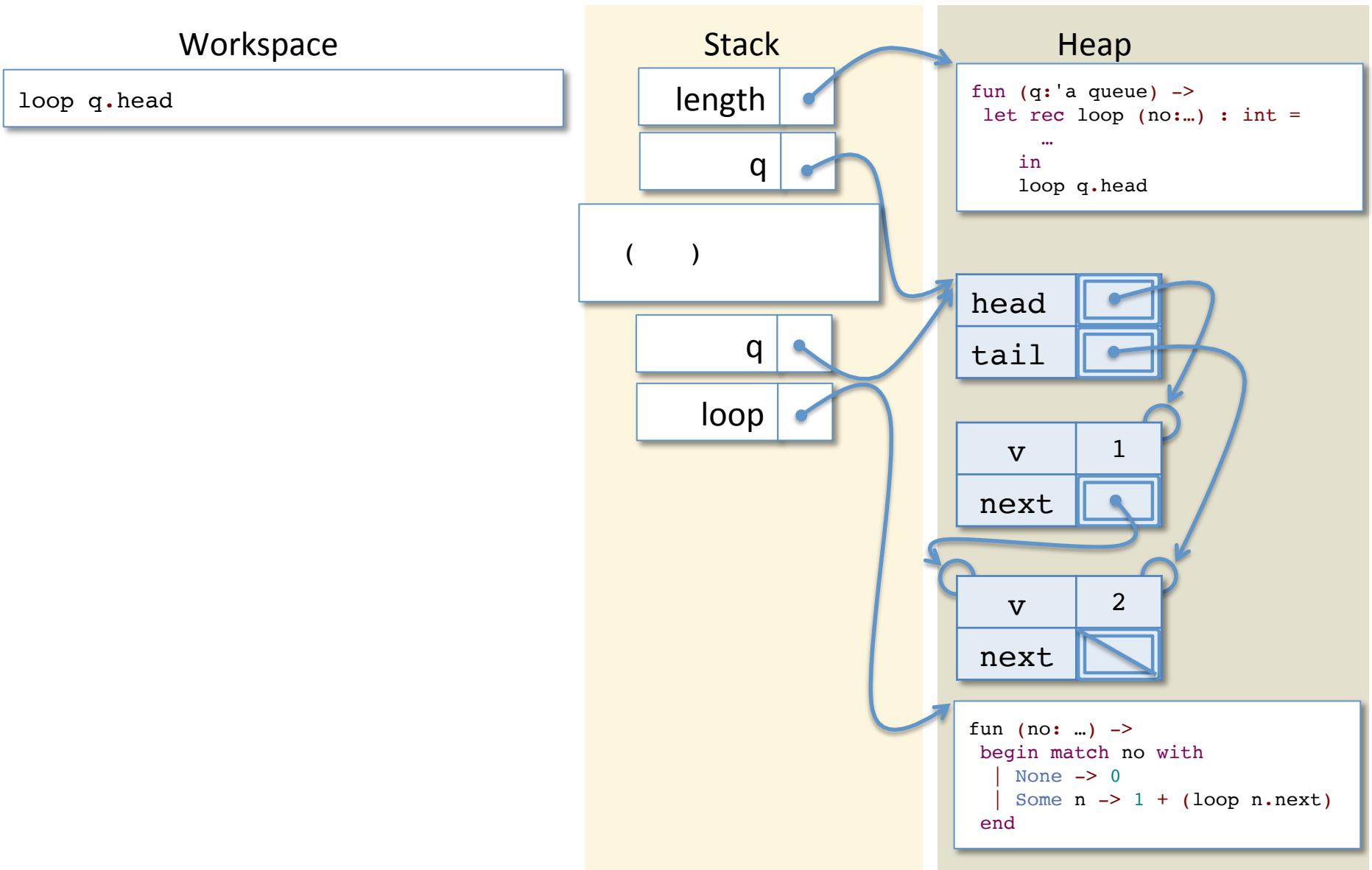
```
fun (q:'a queue) ->
let rec loop (no:...) : int =
  ...
in
loop q.head
```



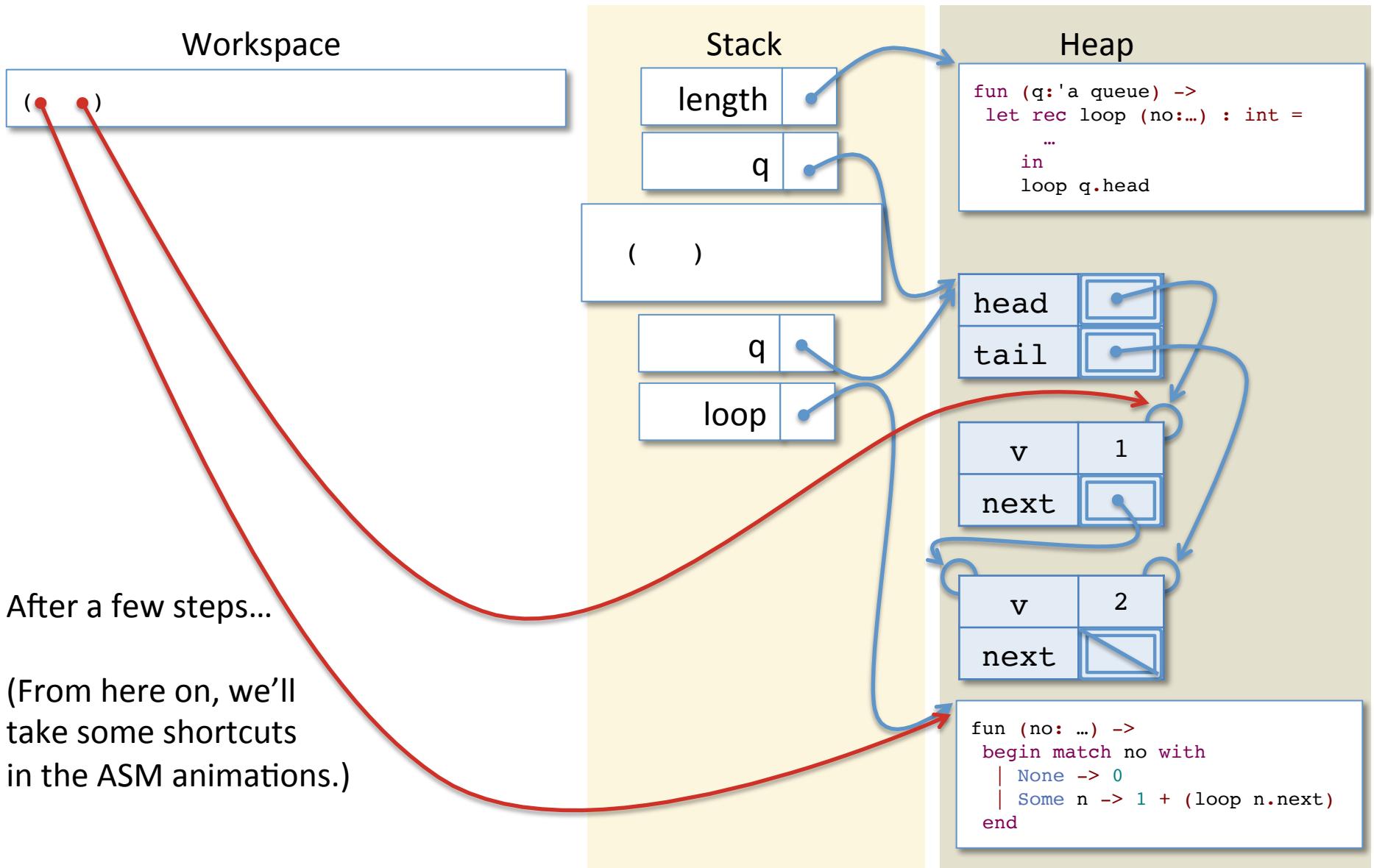
Evaluating length



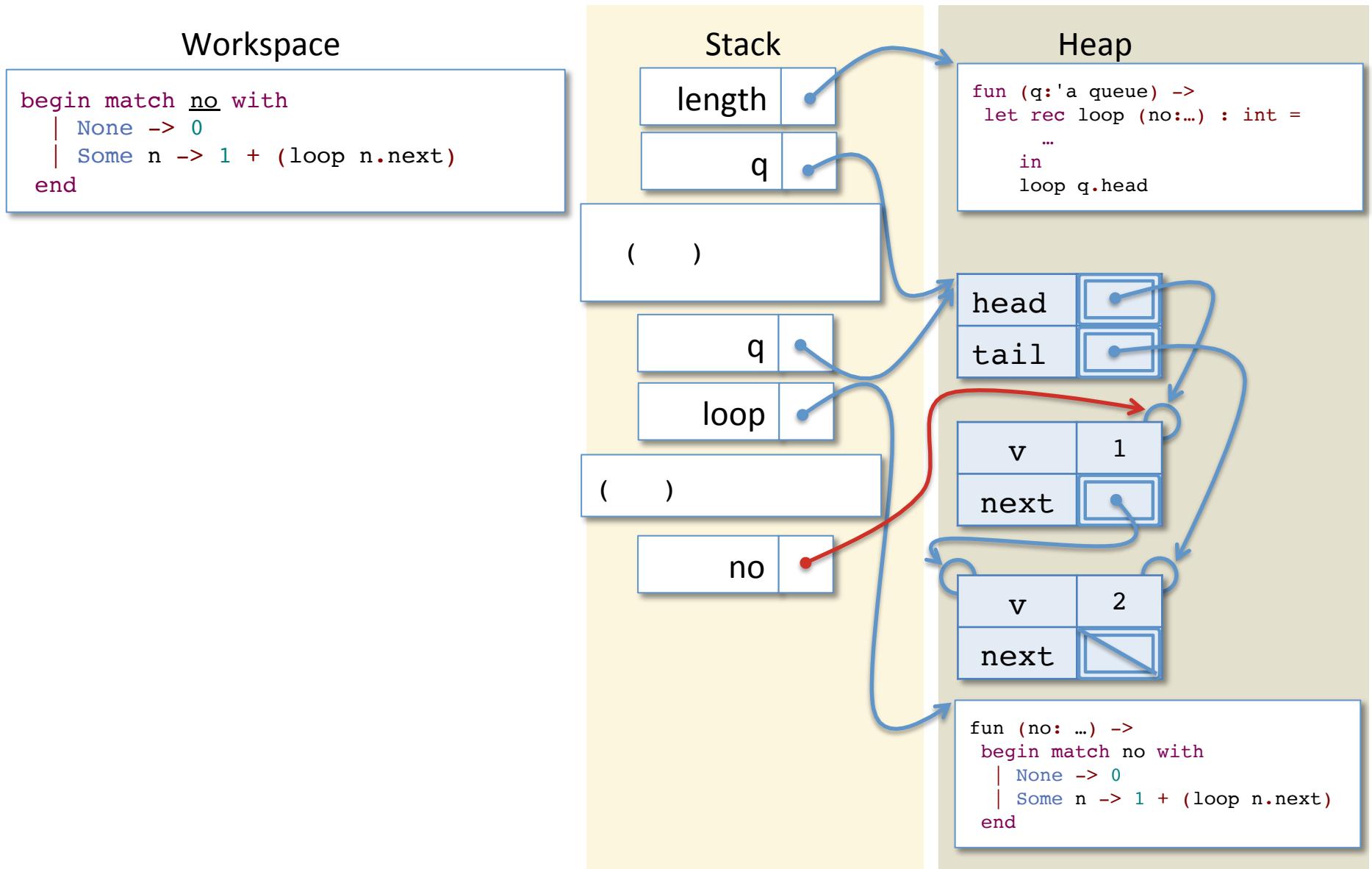
Evaluating length



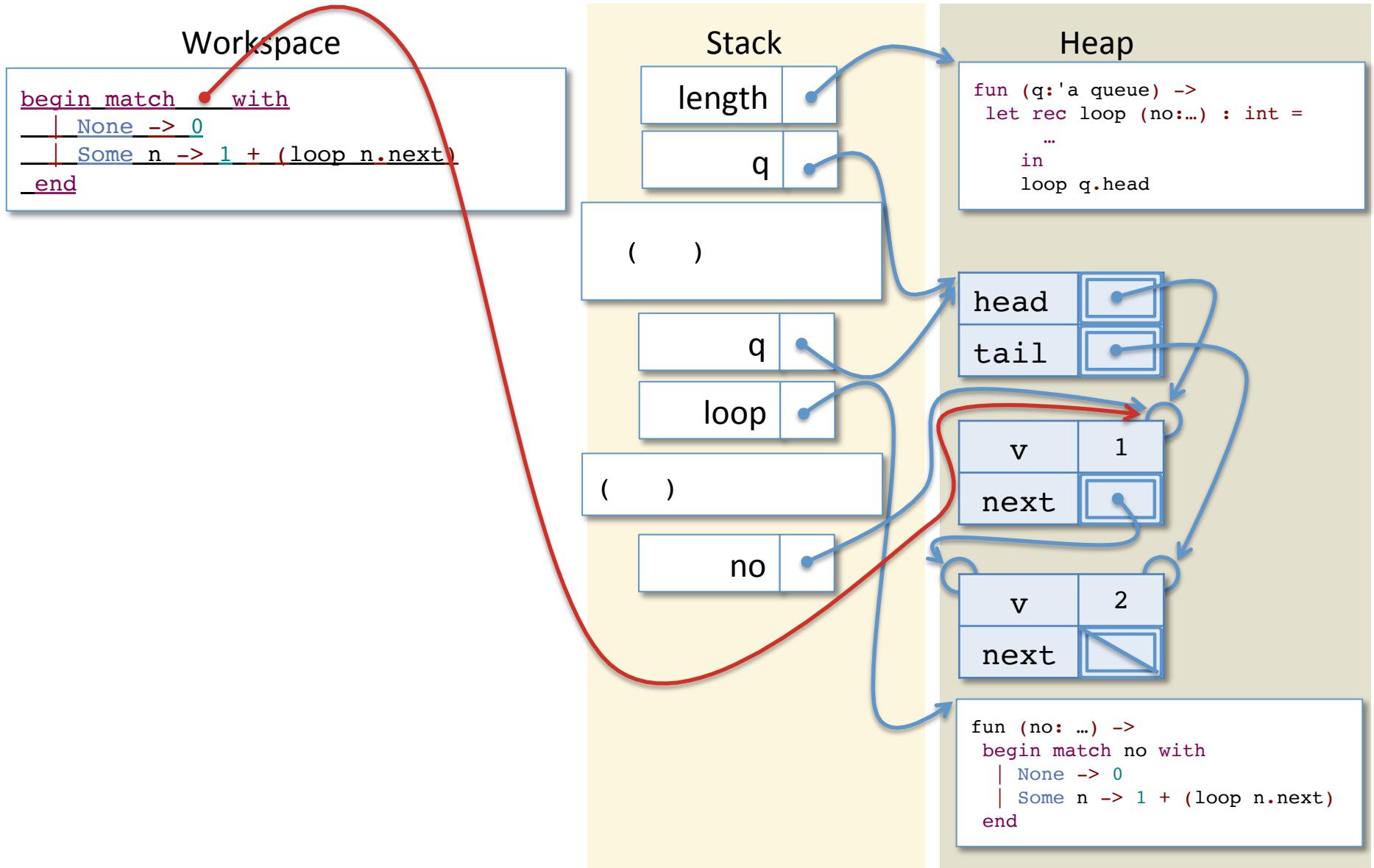
Evaluating length



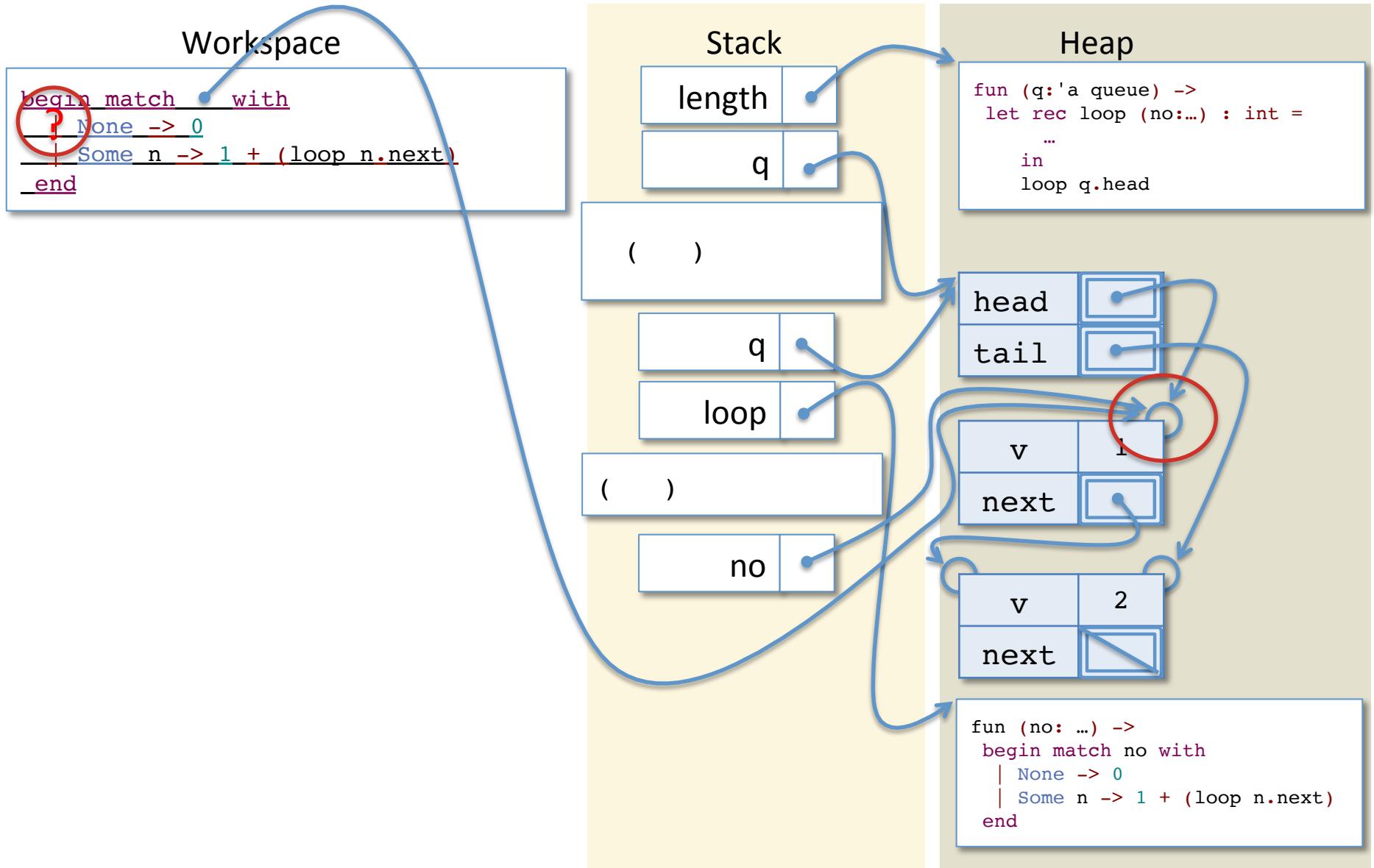
Evaluating length



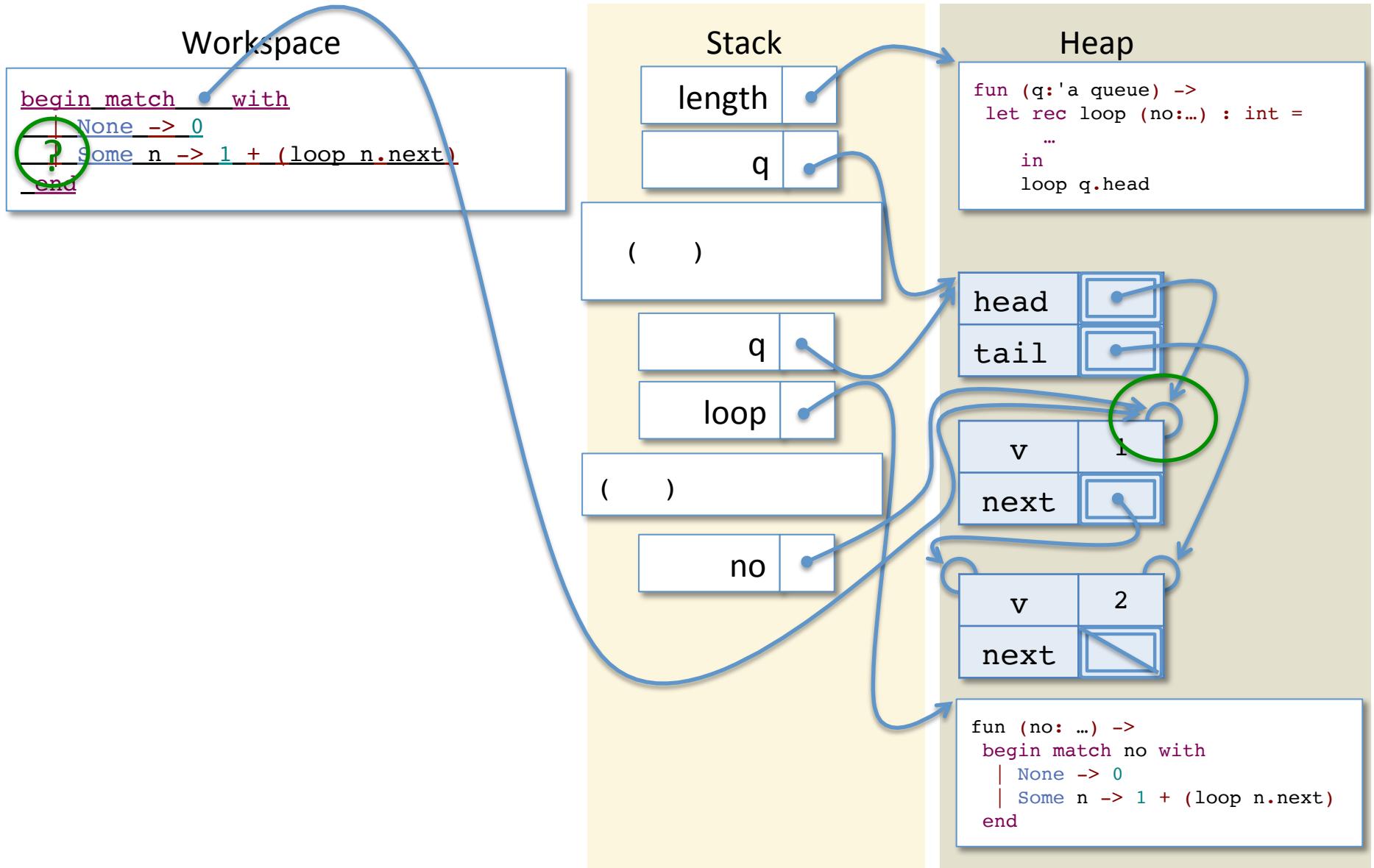
Evaluating length



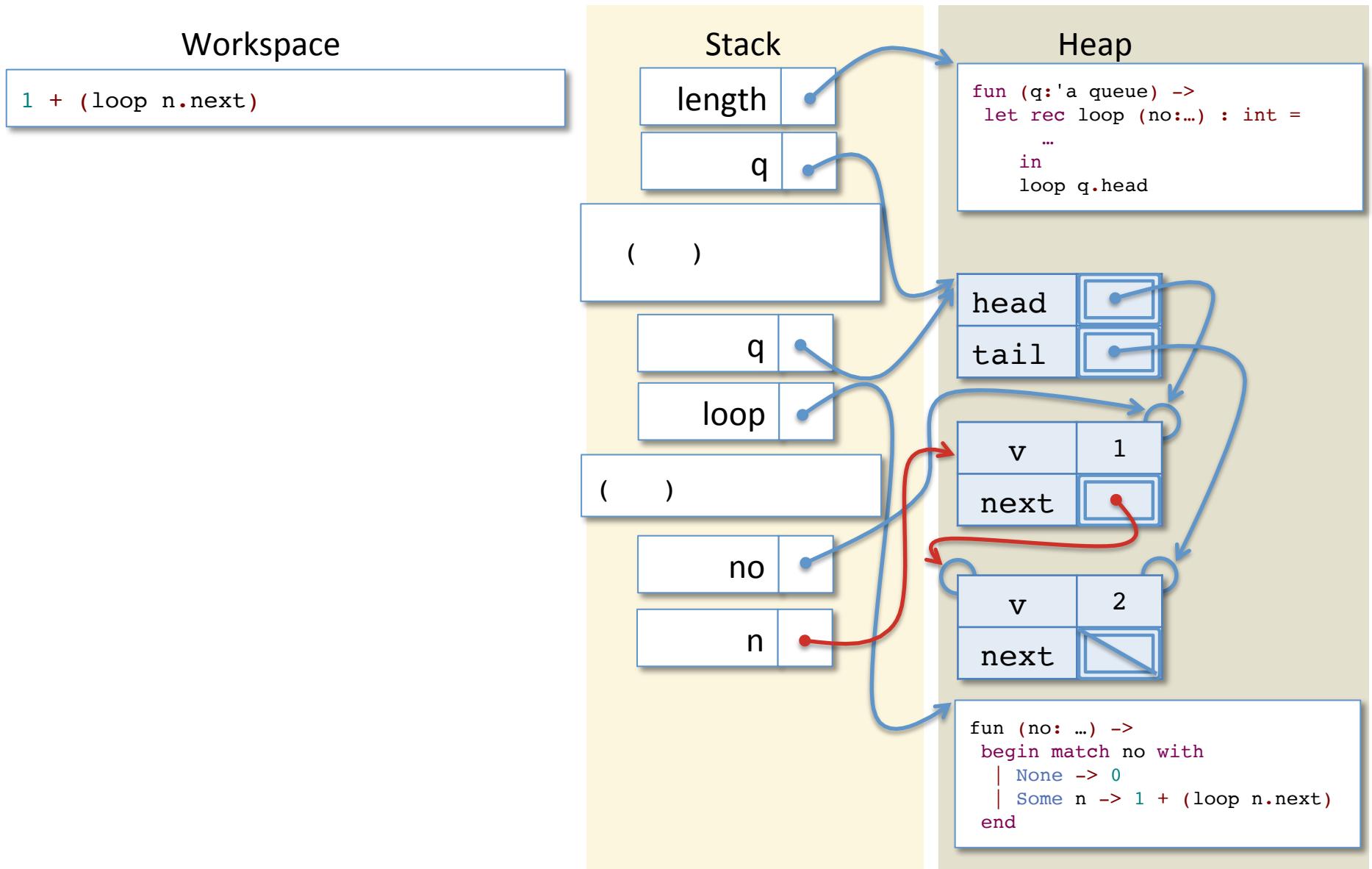
Evaluating length



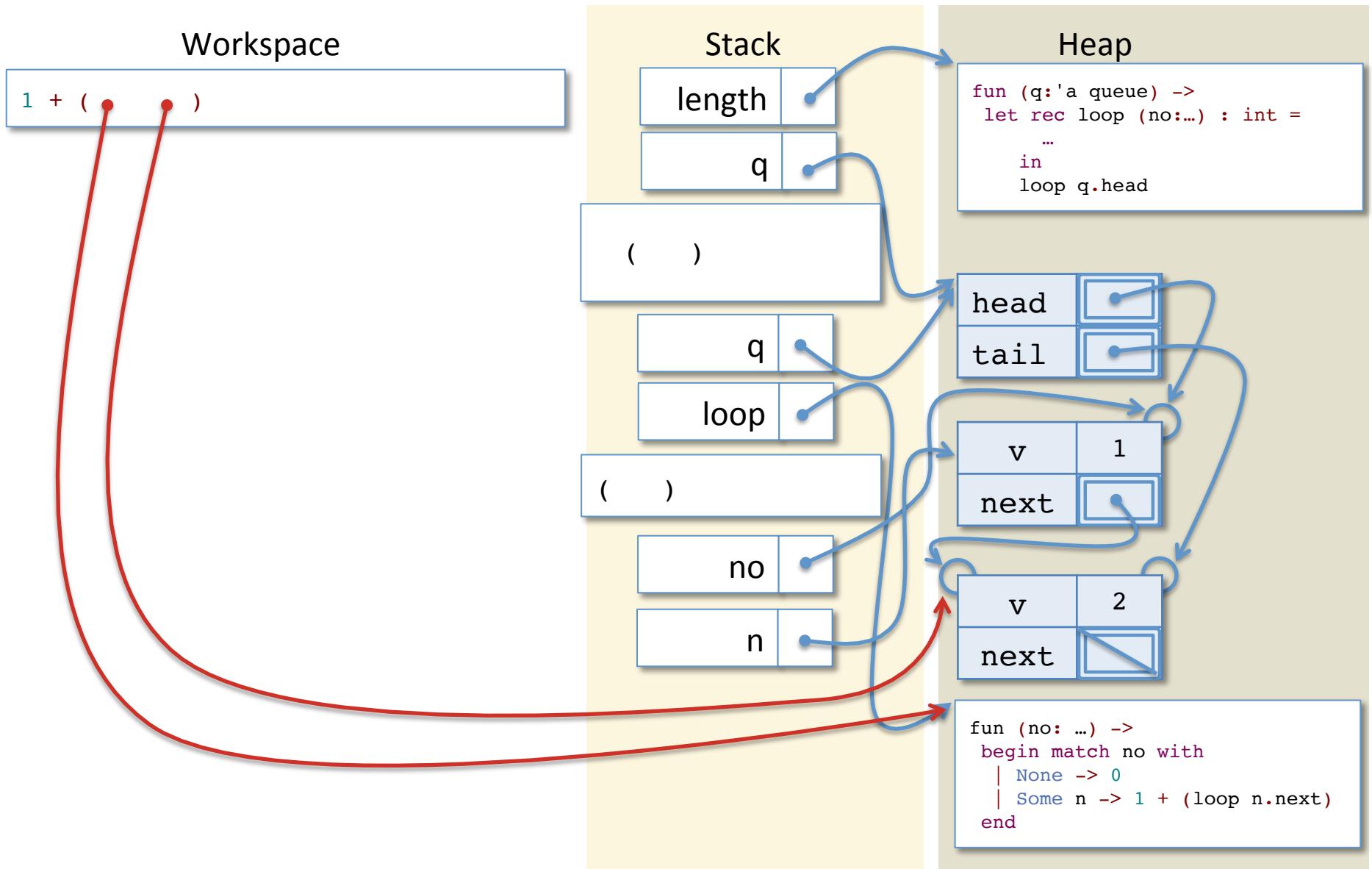
Evaluating length



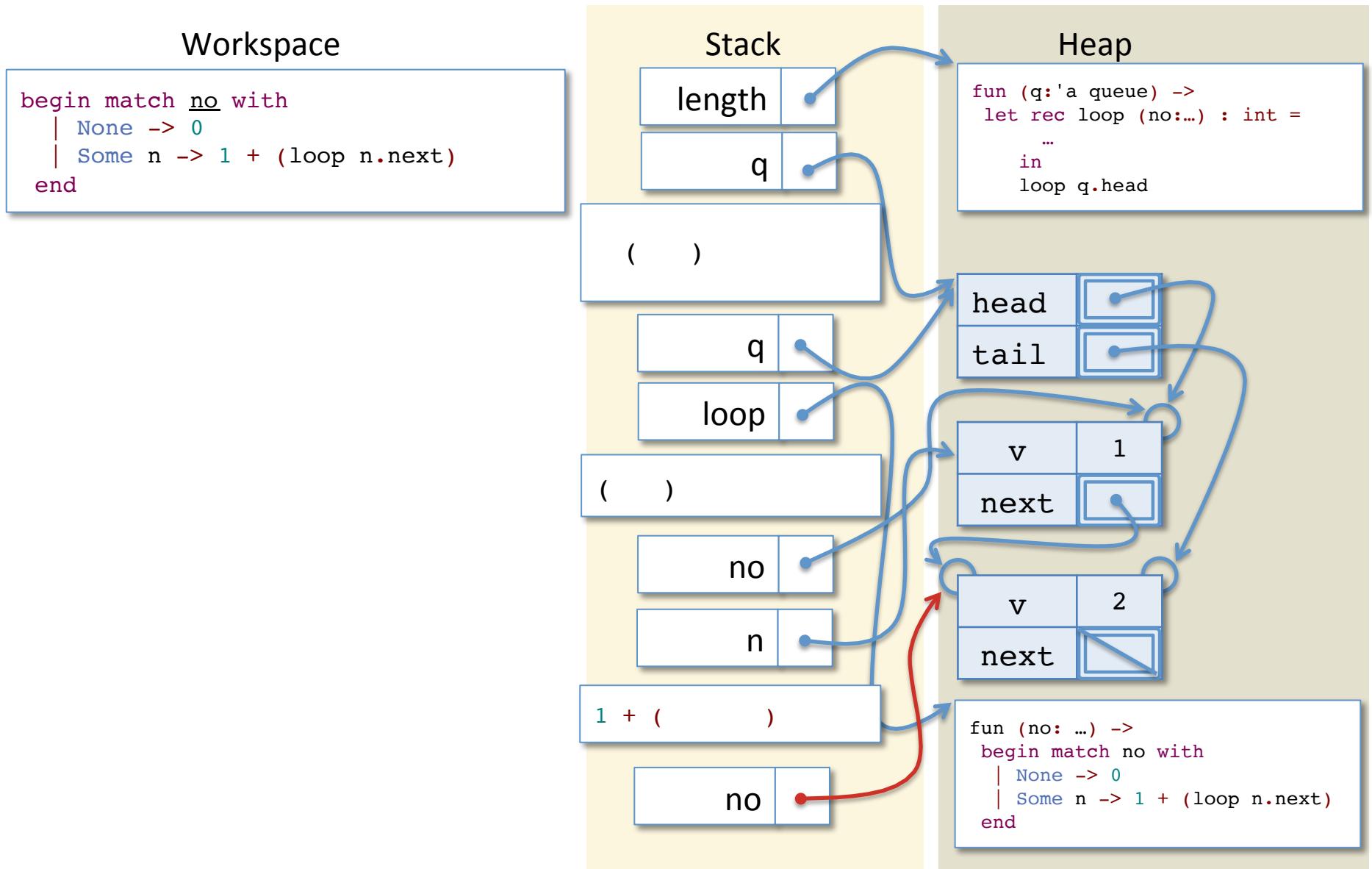
Evaluating length



Evaluating length

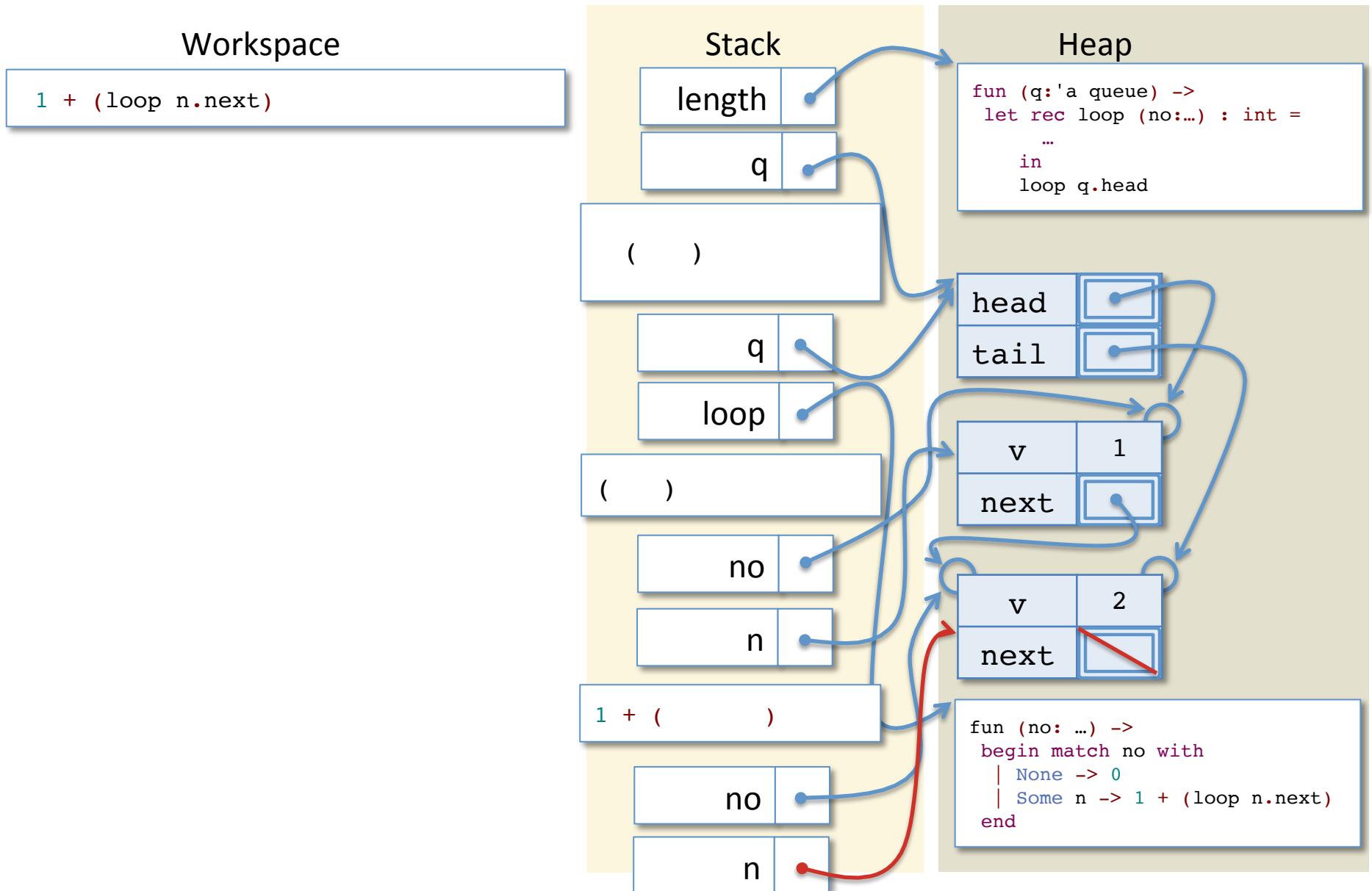


Evaluating length



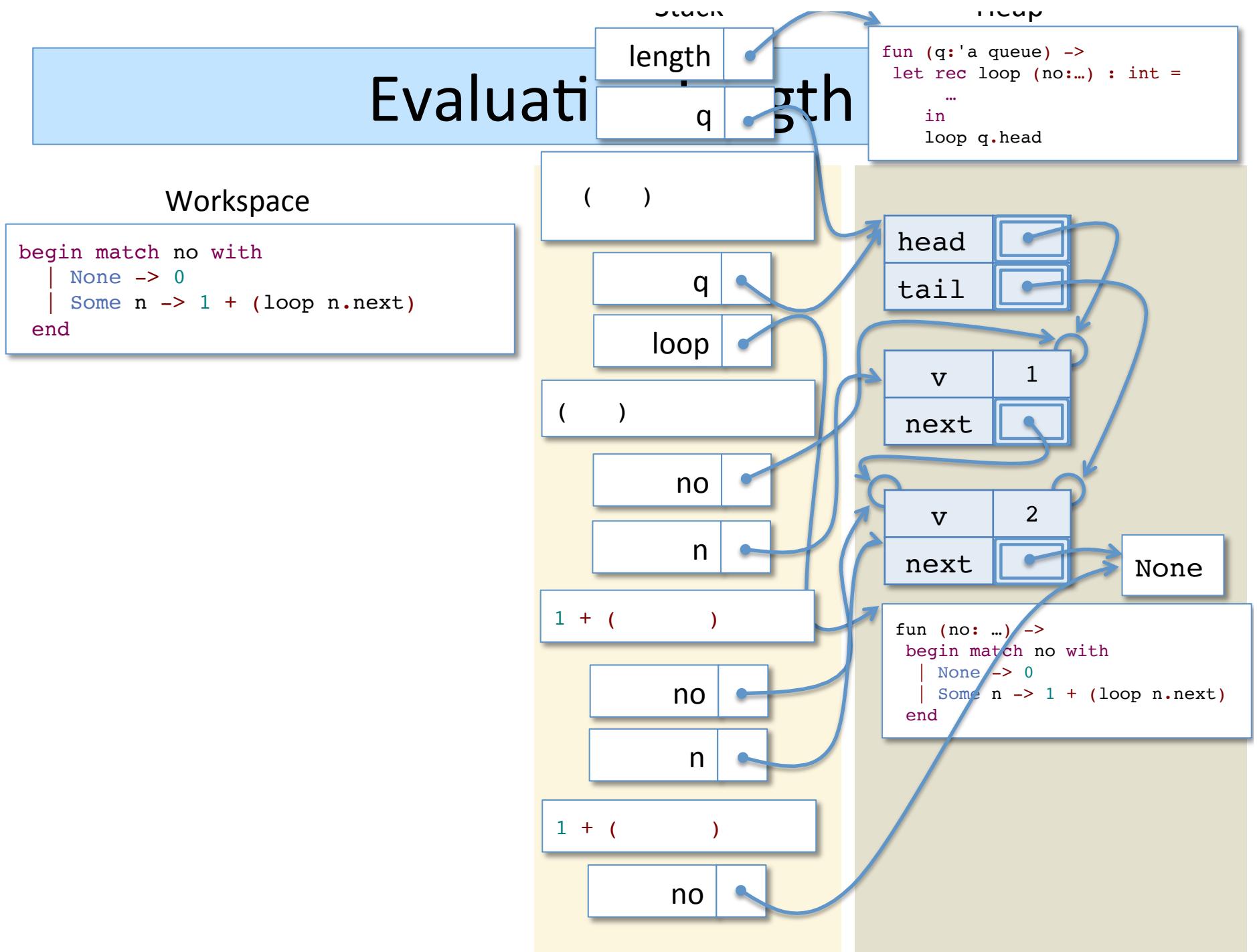
...after a few steps...

Evaluating length

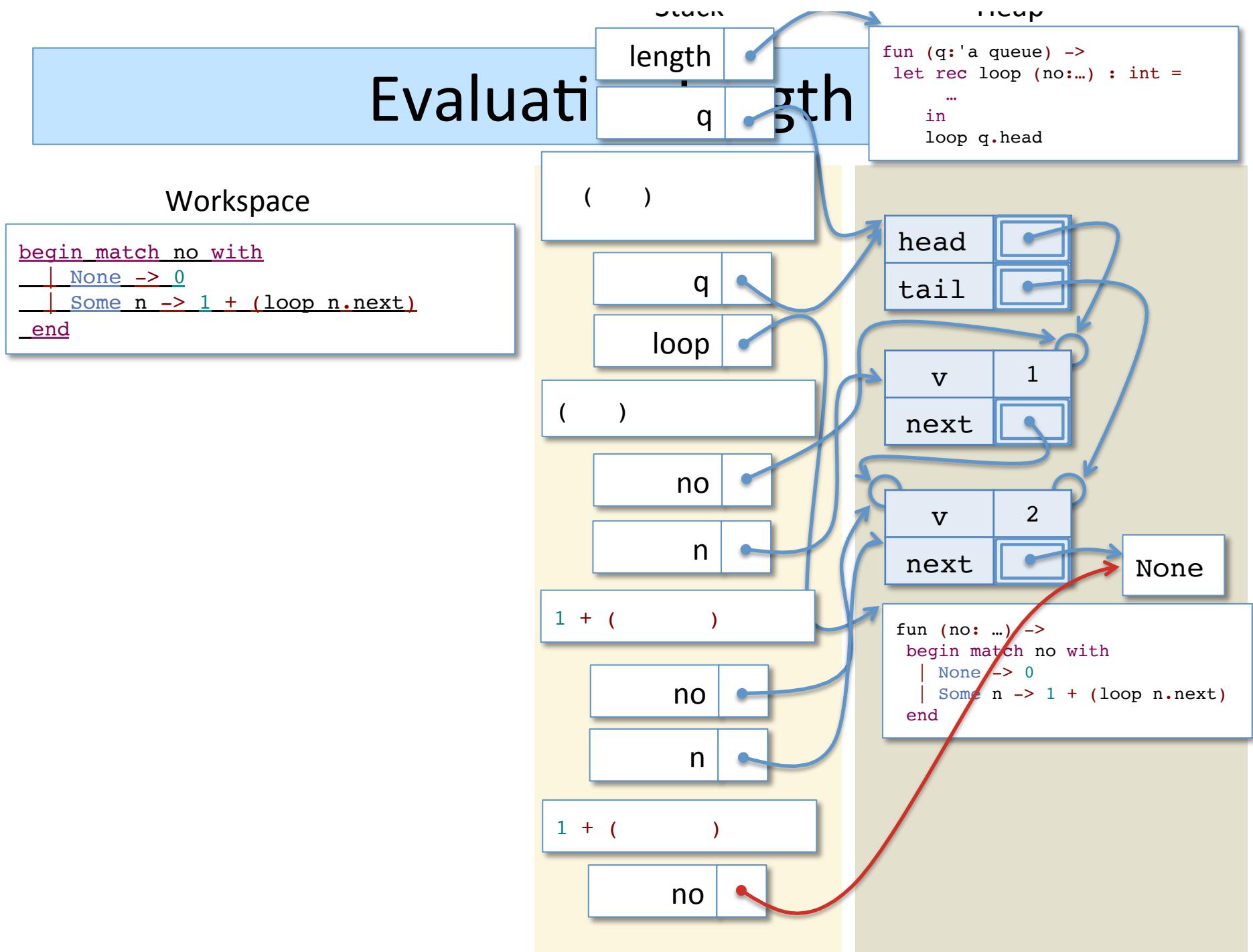


...after a few more steps...

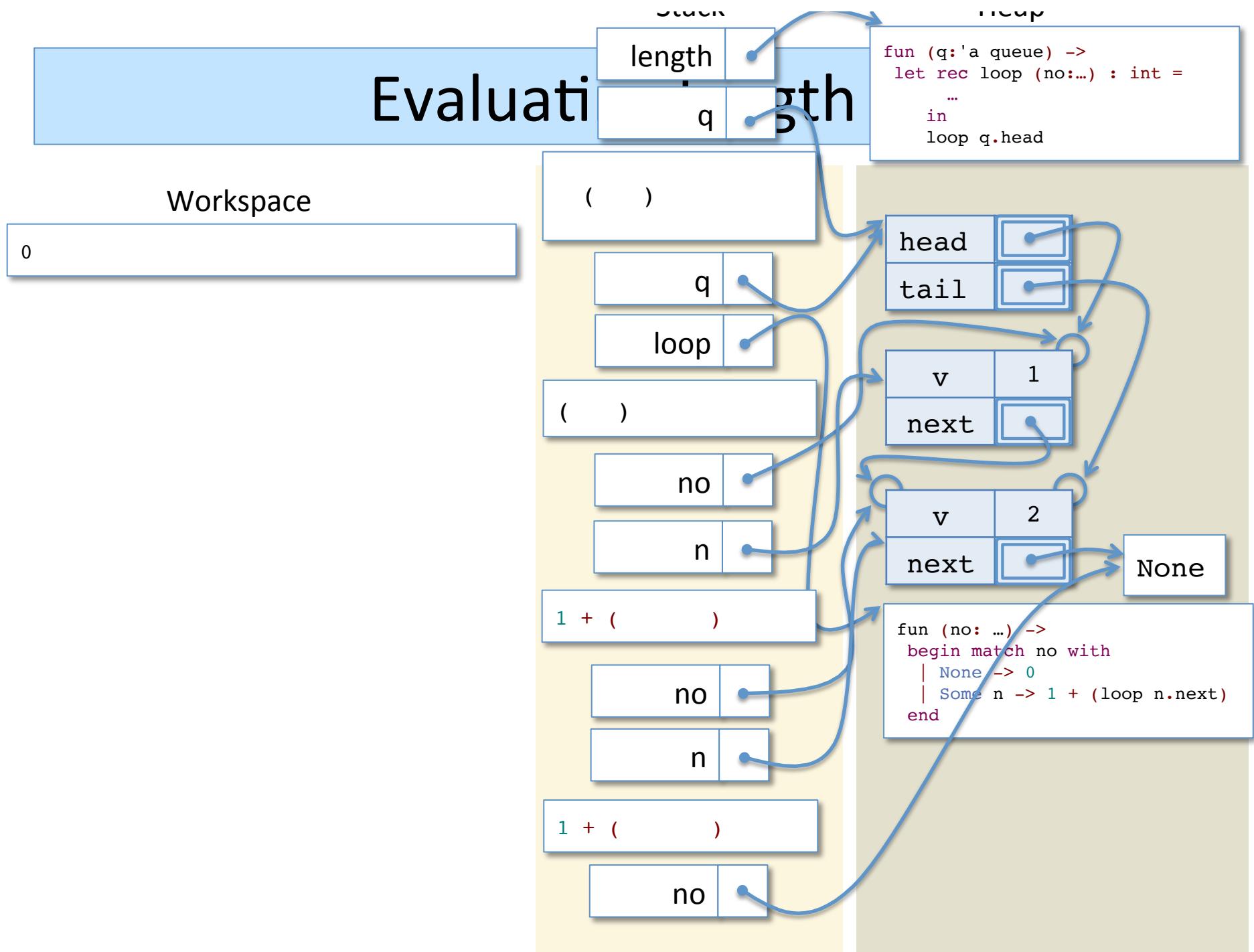
Evaluation



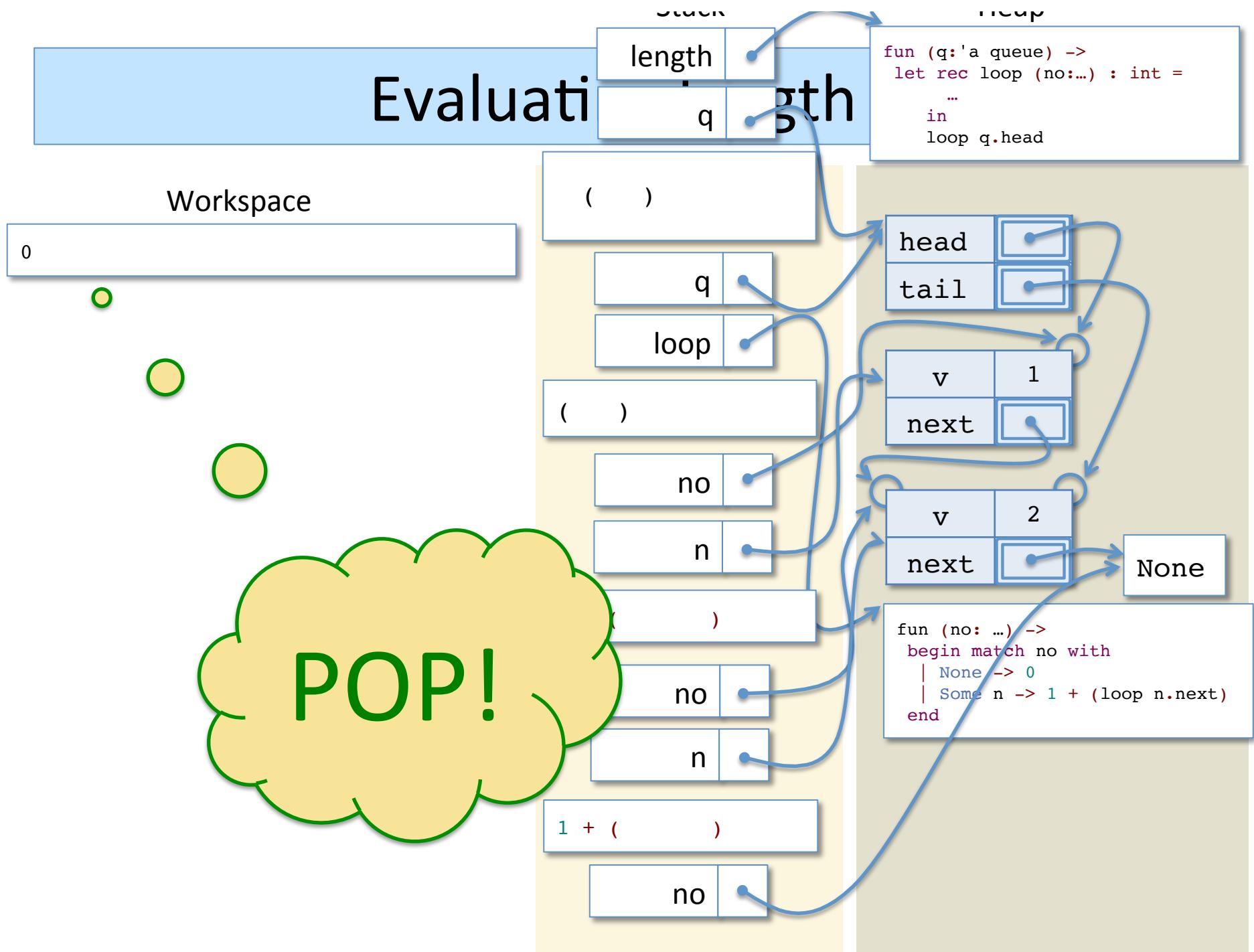
Evaluation



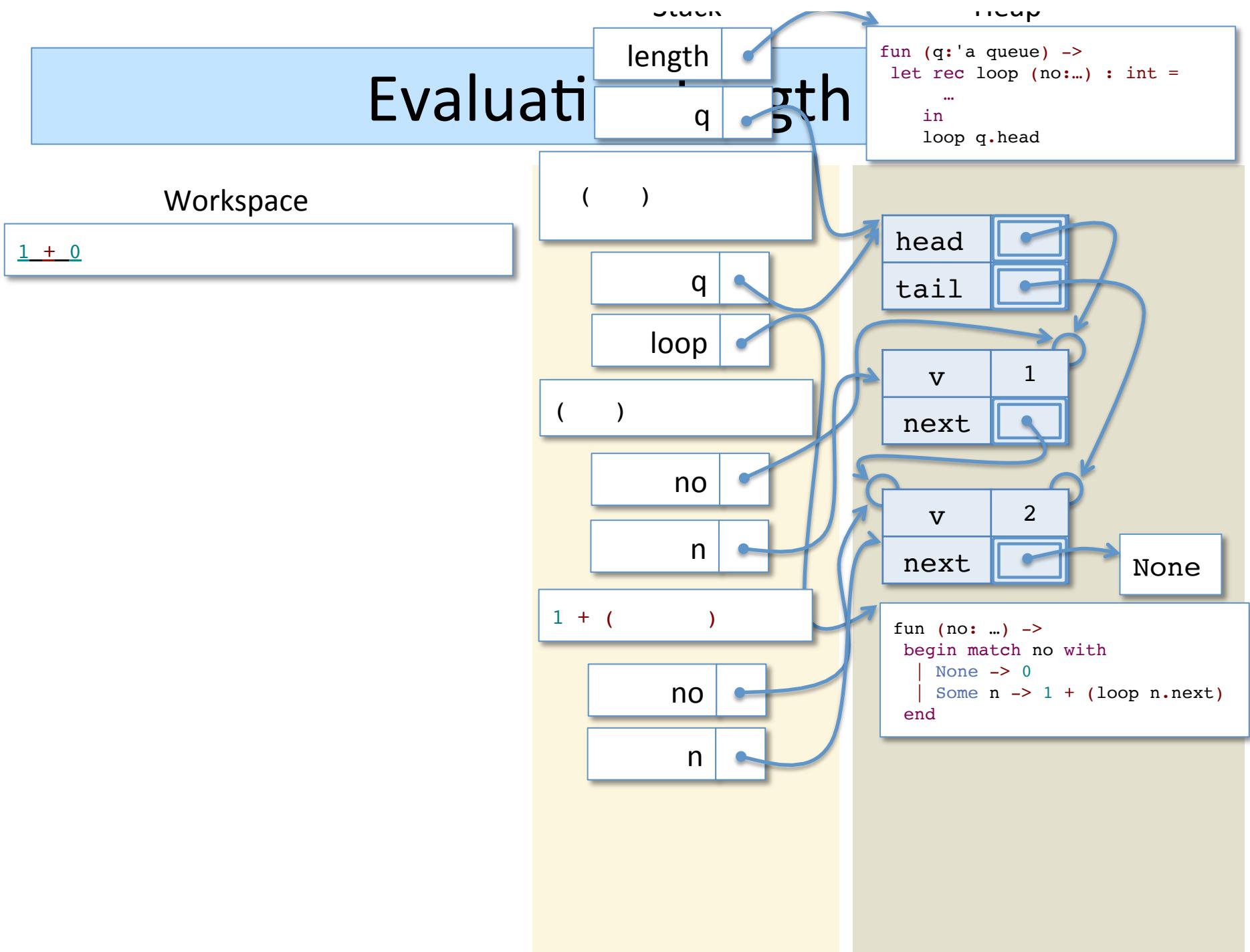
Evaluation



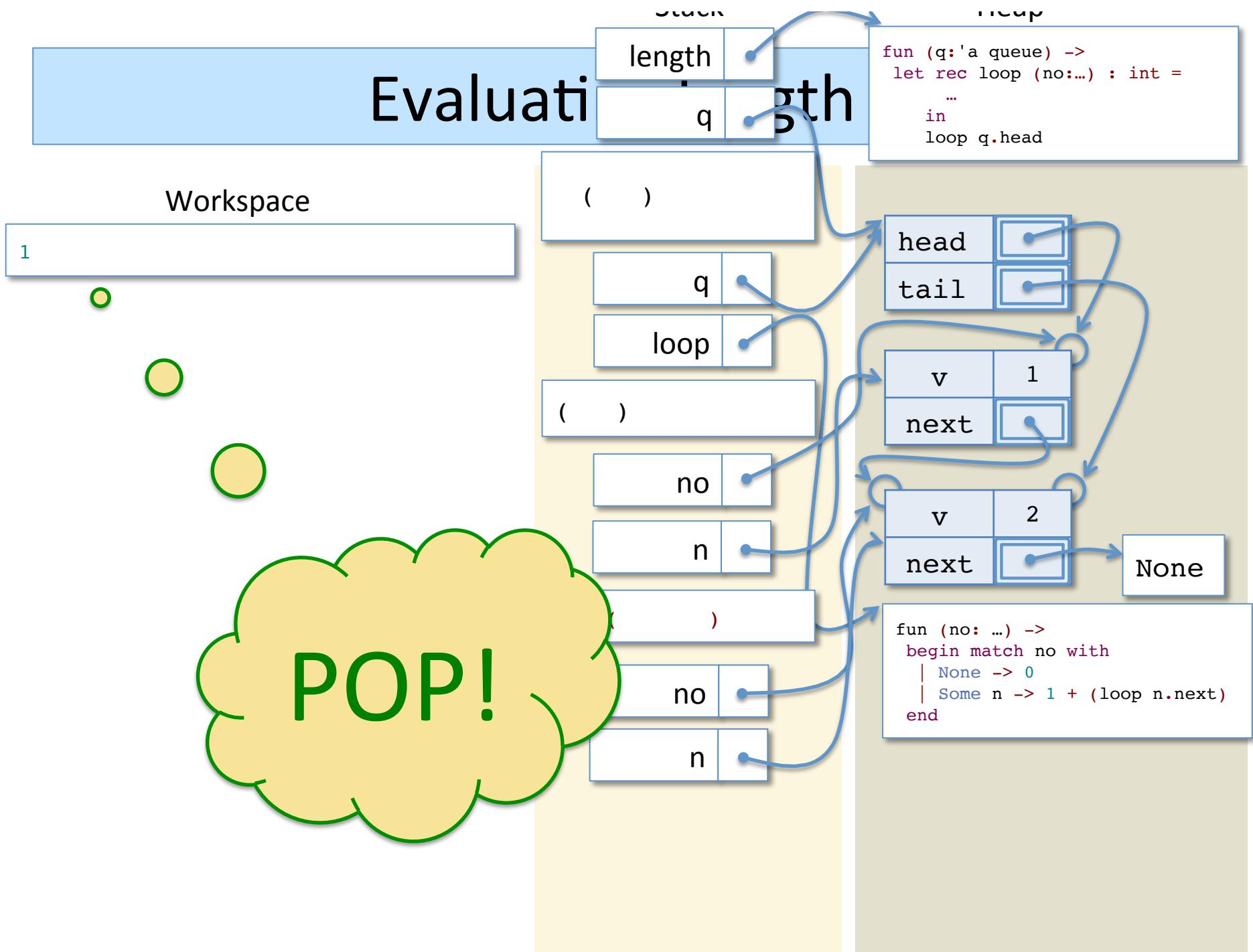
Evaluation



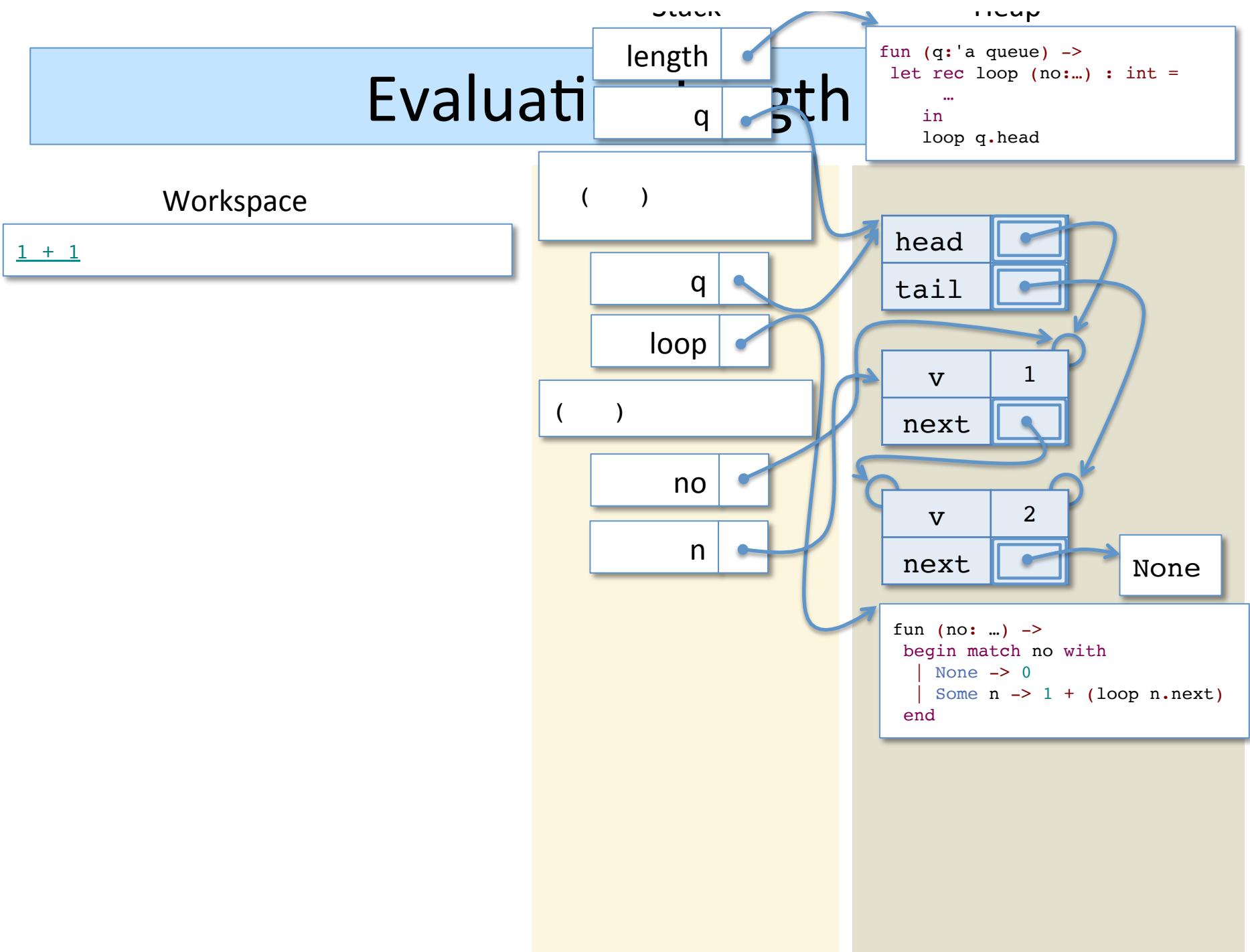
Evaluation



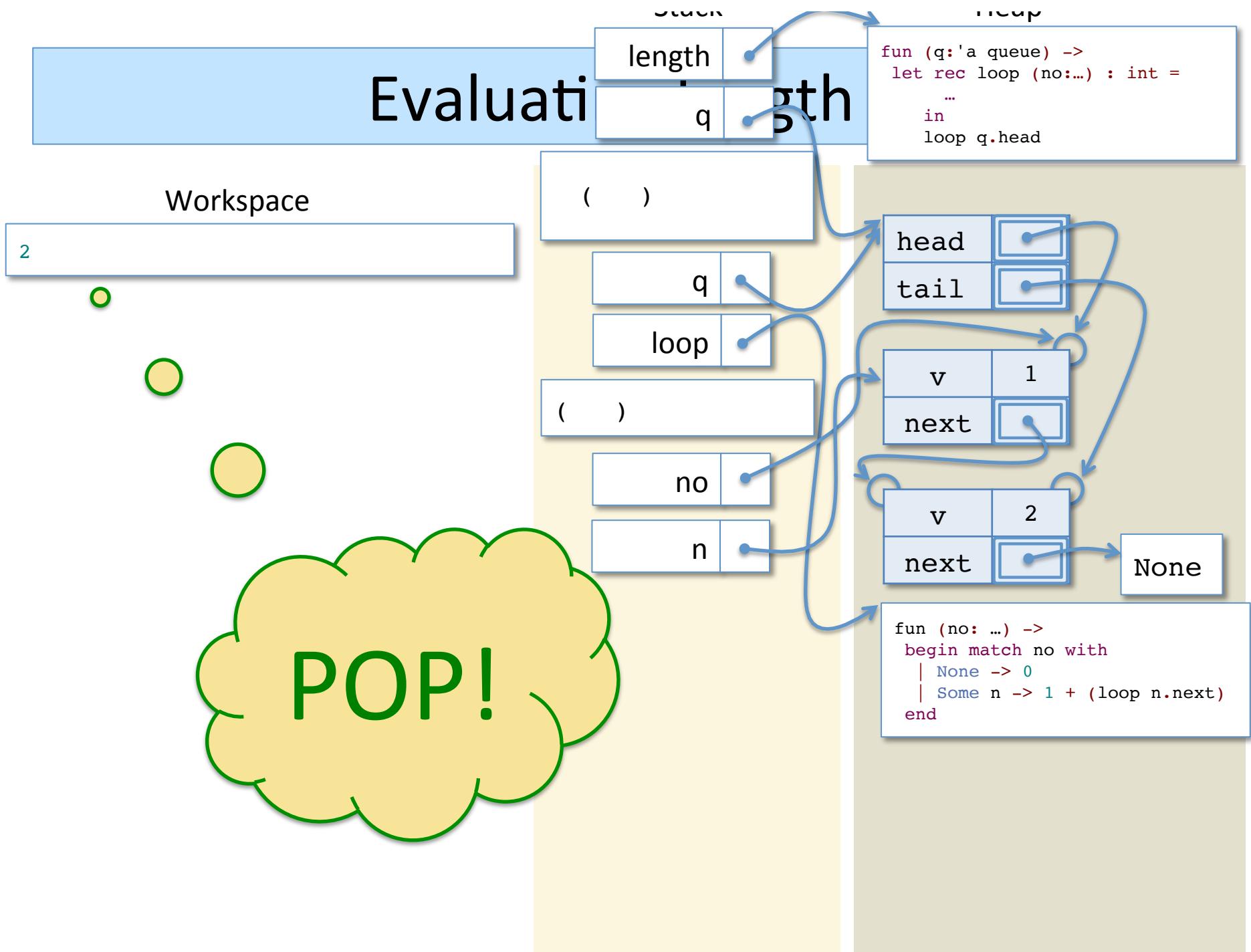
Evaluation



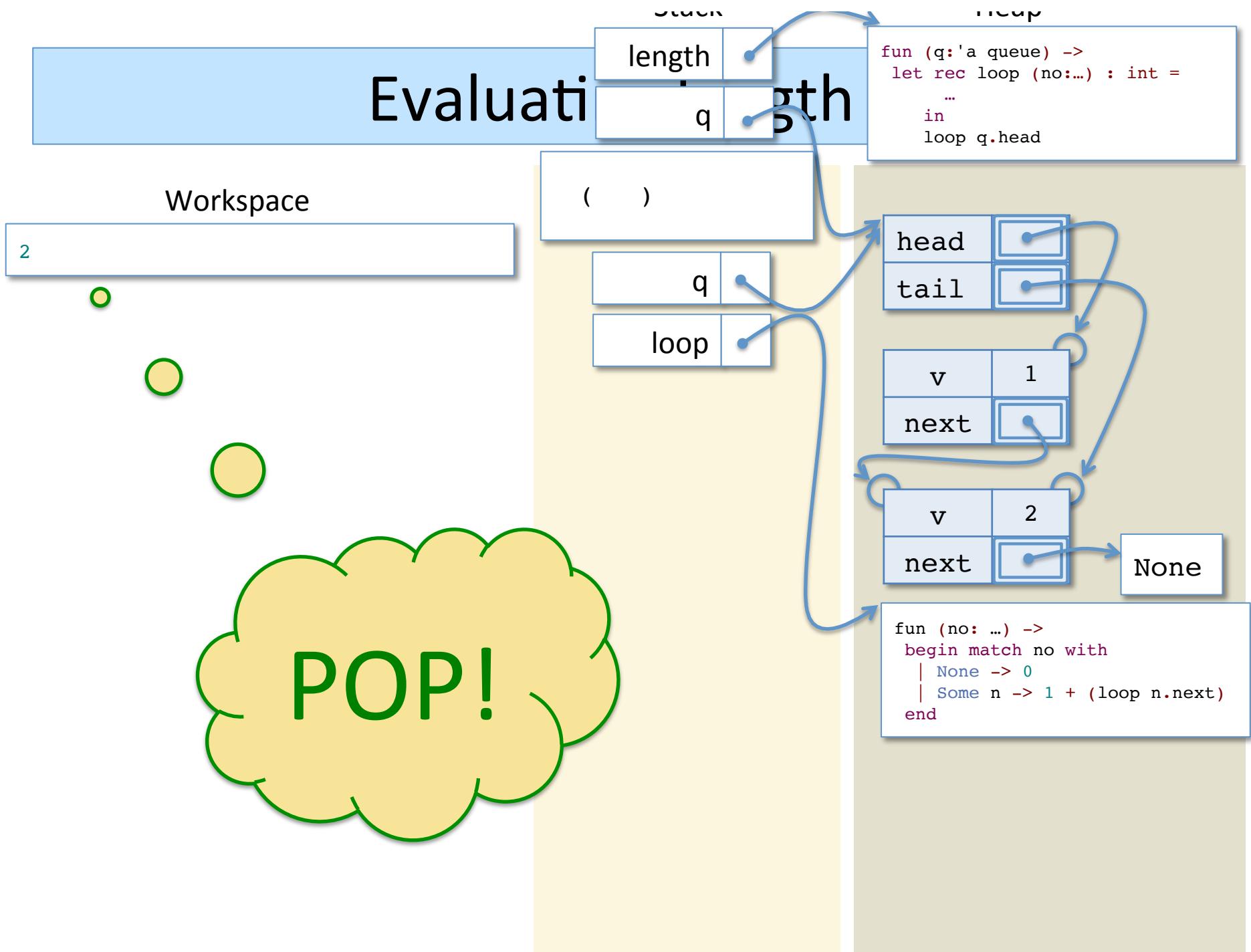
Evaluation



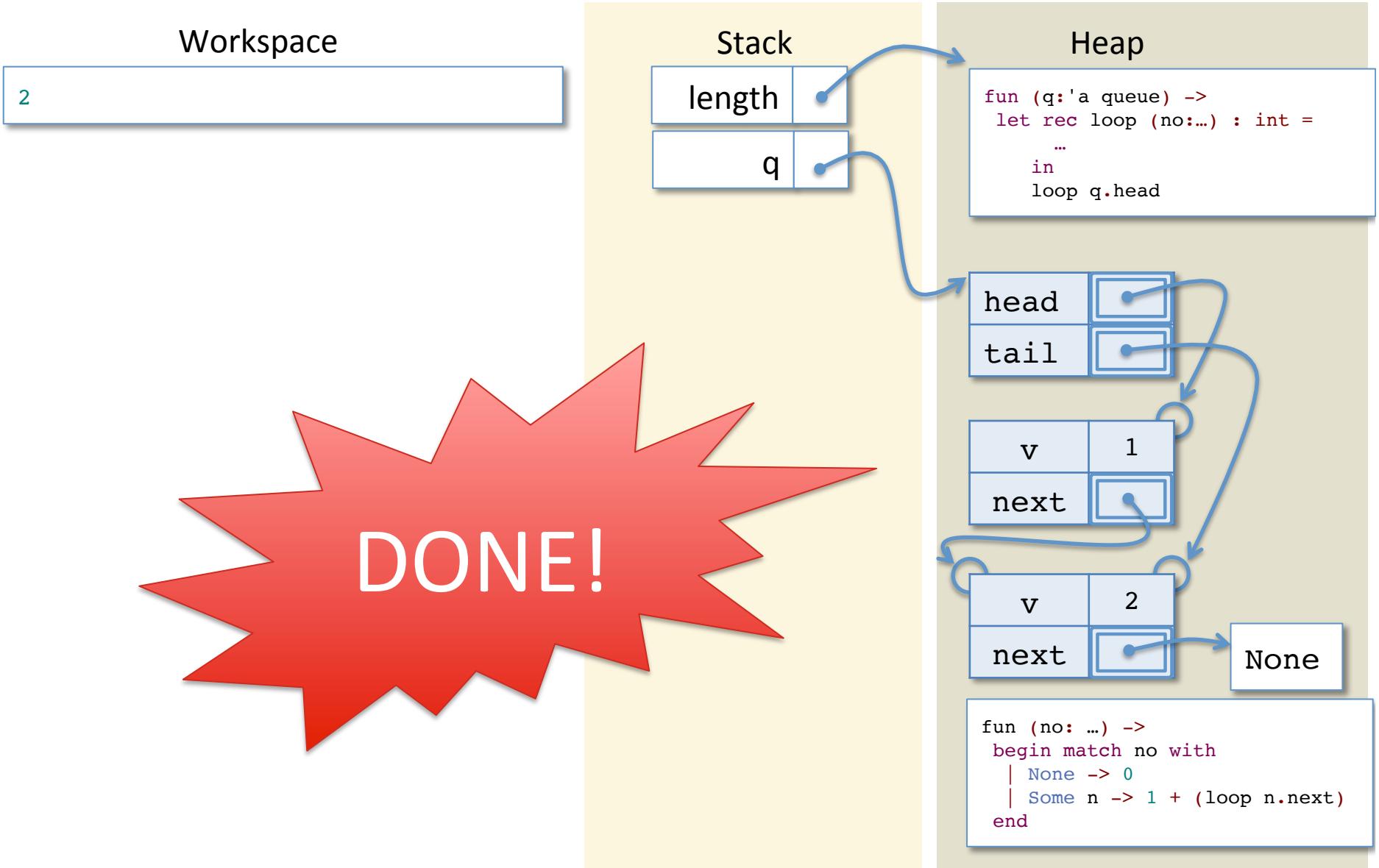
Evaluation



Evaluation



Evaluating length



Iteration

Using tail calls for loops

length (using iteration)

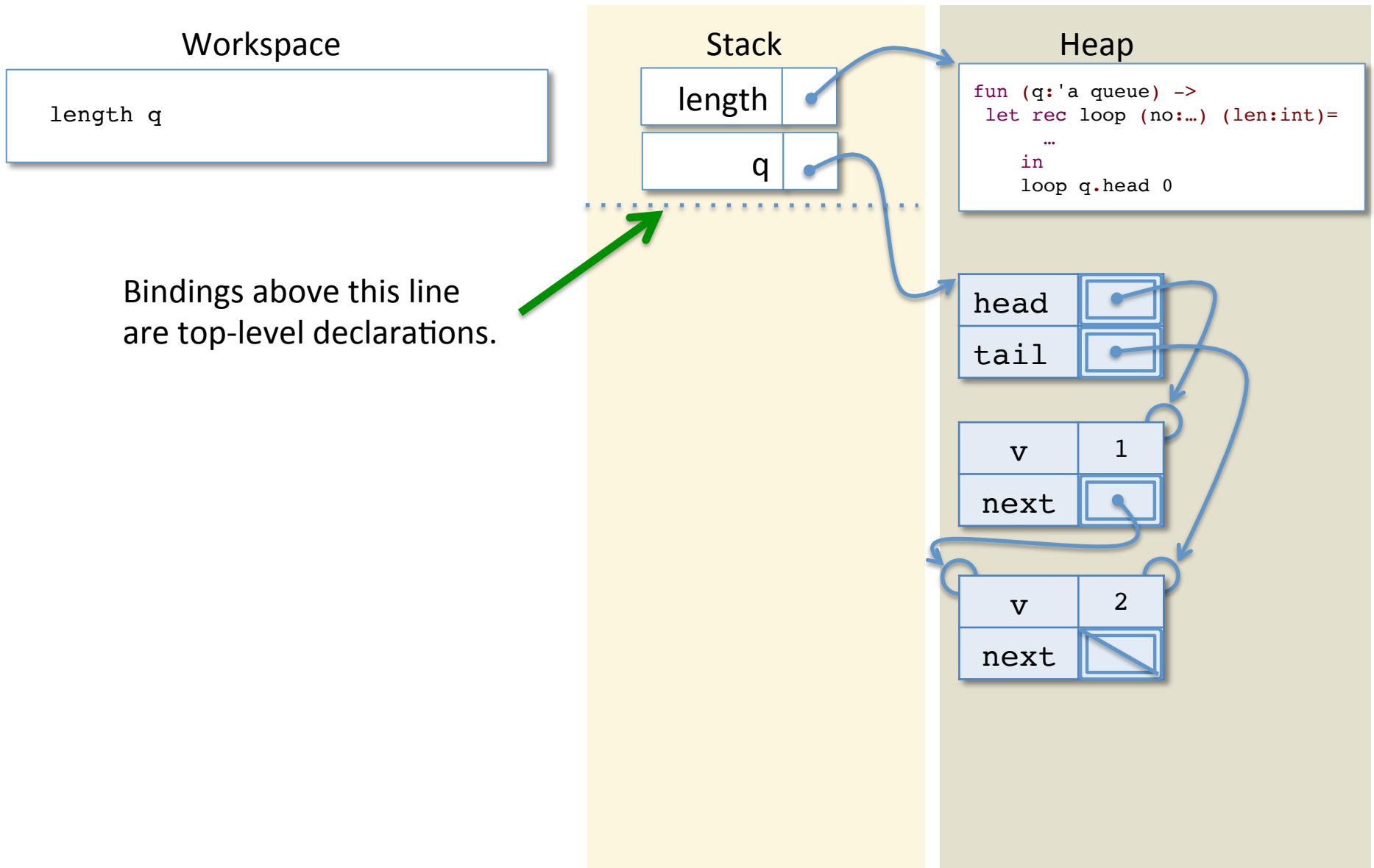
```
(* Calculate the length of the list using iteration *)
let length (q:'a queue) : int =
    let rec loop (no:'a qnode option) (len:int) : int =
        begin match no with
            | None -> len
            | Some n -> loop n.next (1+ len)
        end
    in
    loop q.head 0
```

- This code for `length` also uses a helper function, `loop`:
 - This loop takes an extra argument, `len`, called the *accumulator*
 - Unlike the previous solution, the computation happens “on the way down” as opposed to “on the way back up”
 - Note that `loop` will always be called in an empty workspace—the results of the call to `loop` never need to be used to compute another expression. In contrast, we had $(1 + (\text{loop} \dots))$ in the recursive version.

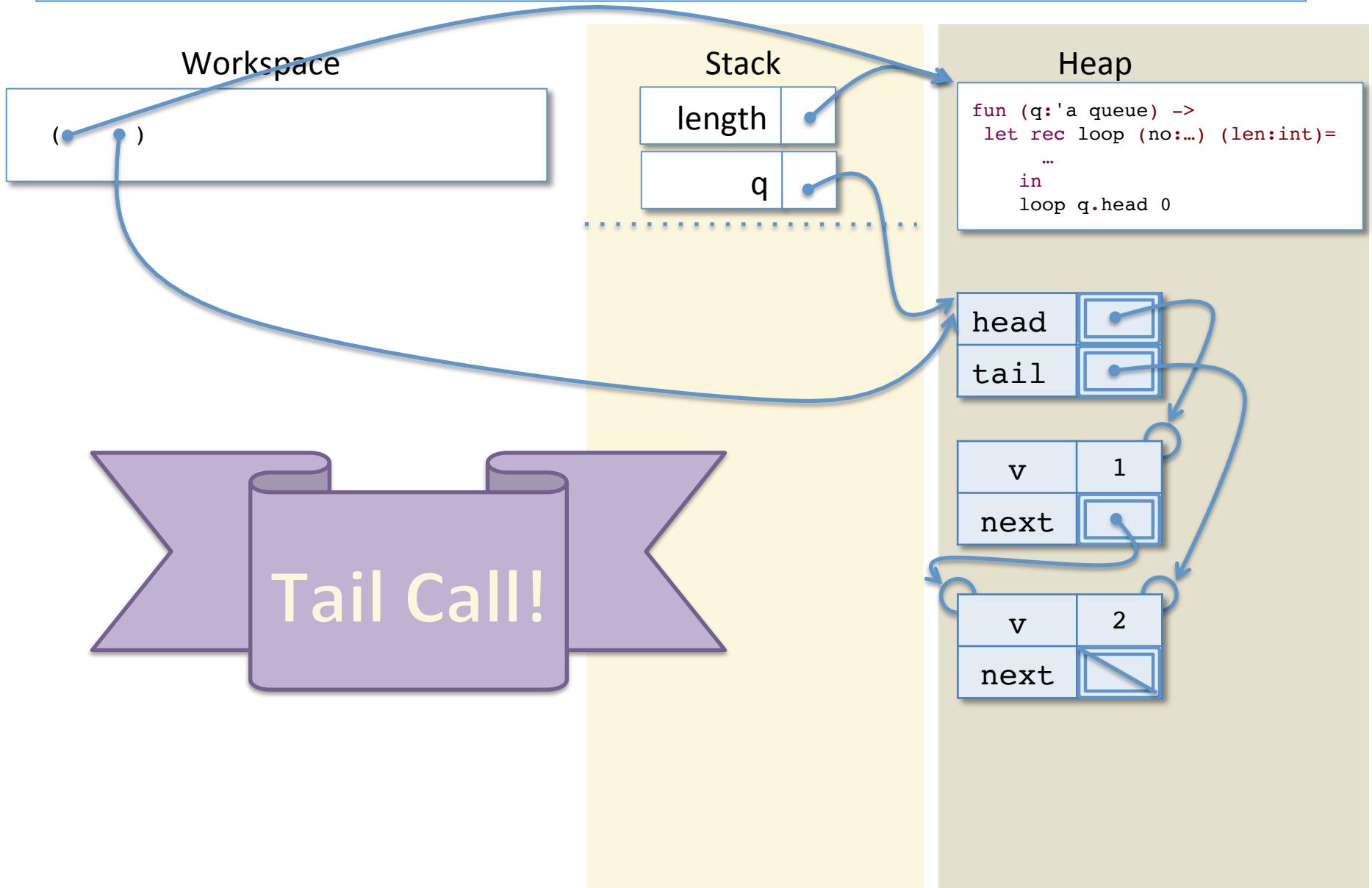
Tail Call Optimization

- Why does it matter that ‘loop’ is only called in an empty workspace?
- We can *optimize* the abstract stack machine:
 - The workspace pushed onto the stack tells us “what to do” when the function call returns.
 - If the pushed workspace is empty, we will always ‘pop’ immediately after the function call returns.
 - So there is no need to save the empty workspace on the stack!
 - Moreover, any local variables that were pushed so that the current workspace could evaluate will no longer be needed, so we can eagerly pop them too.
- The upshot is that we can execute a tail recursion just like a ‘for’ loop in Java or C, using a constant amount of stack space.

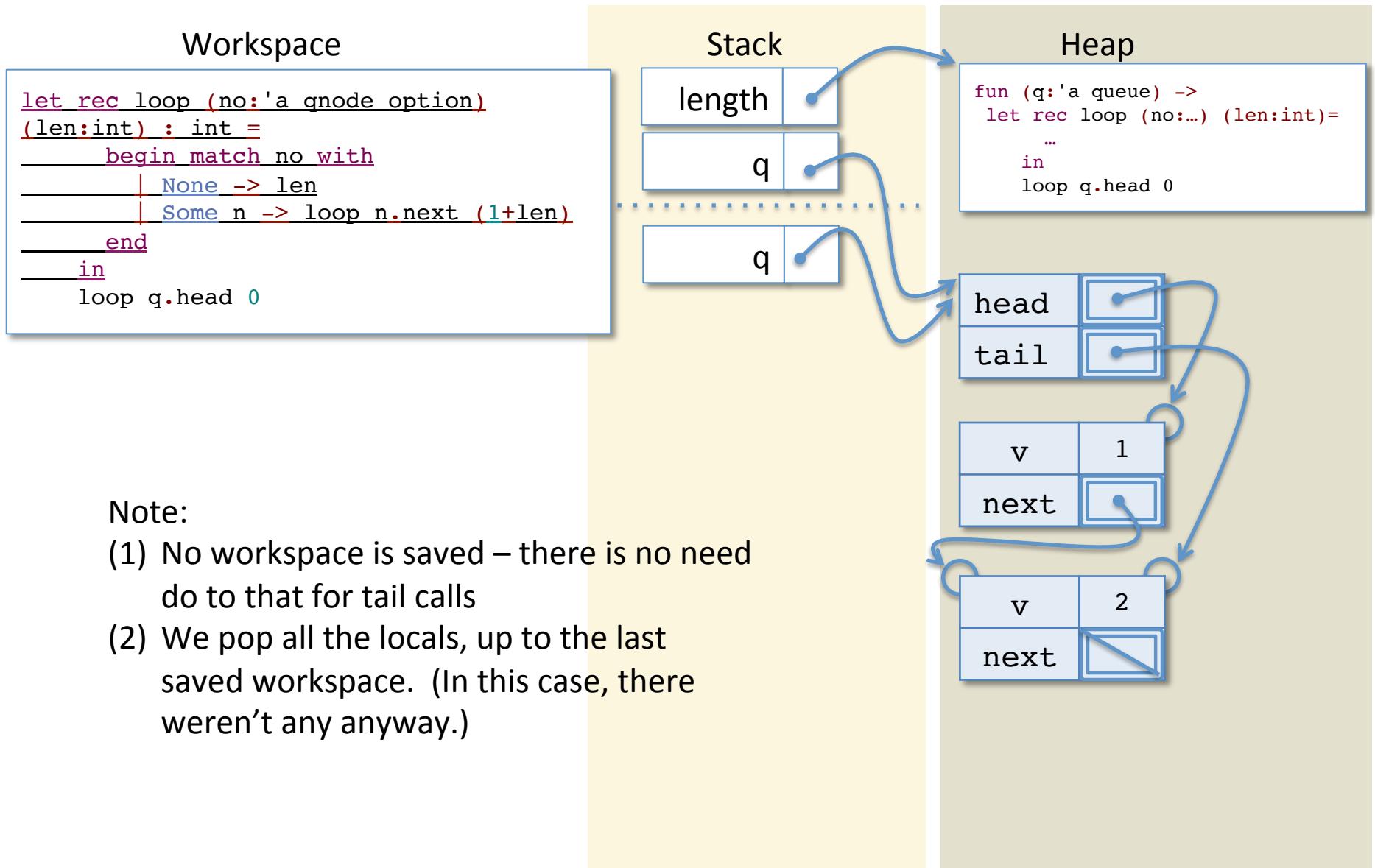
Tail Calls and Iterative length



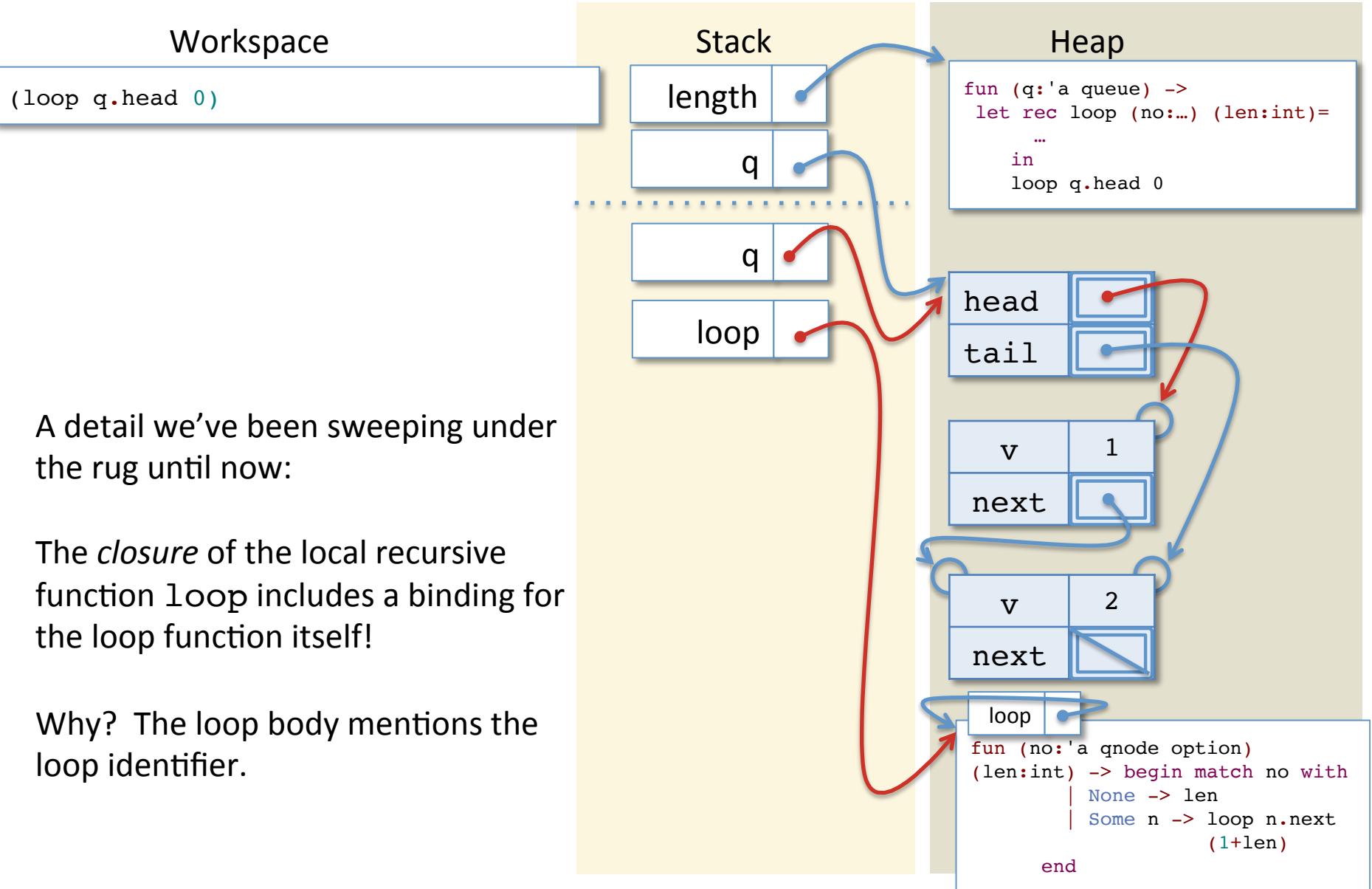
Tail Calls and Iterative length



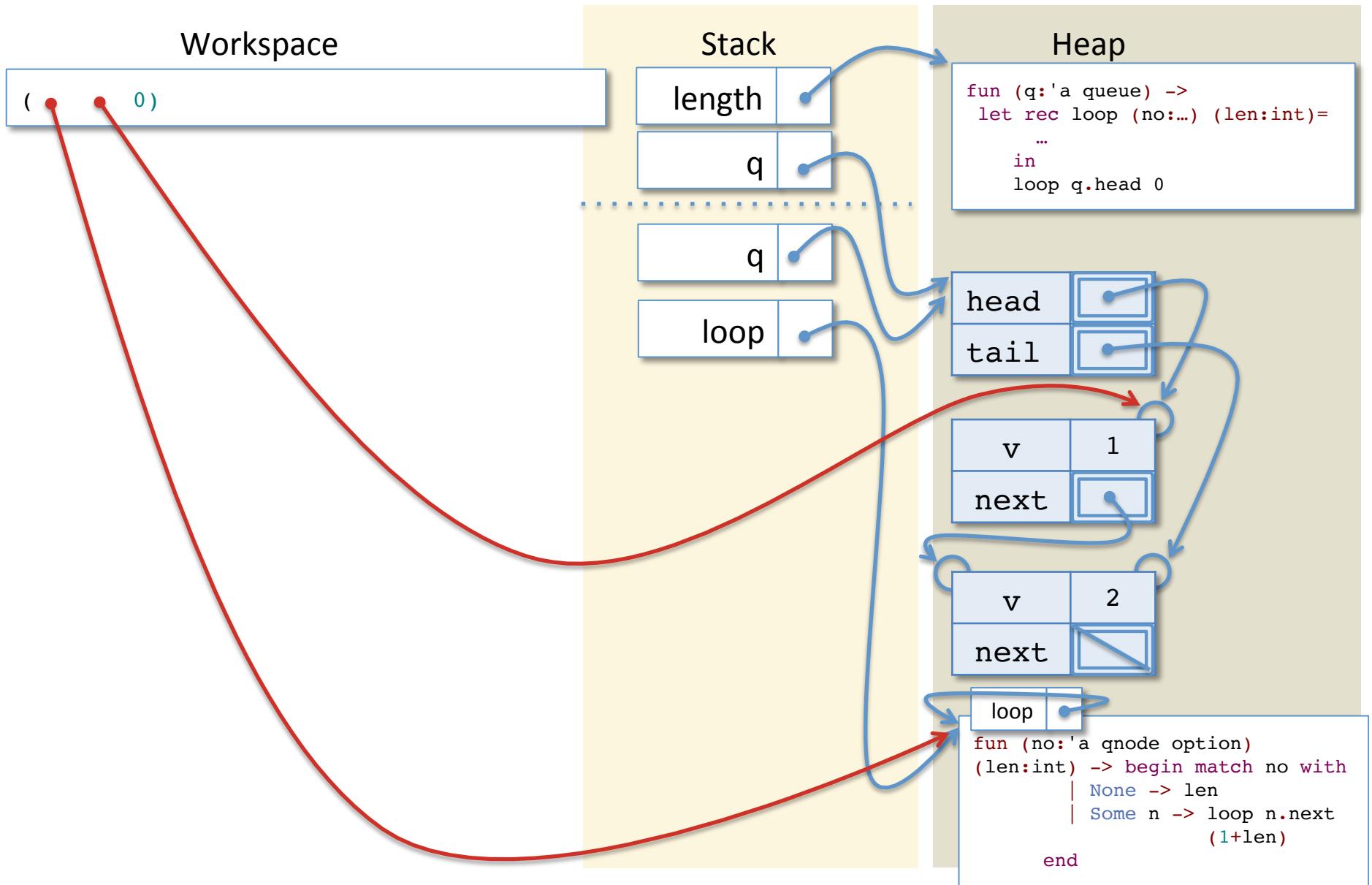
Tail Calls and Iterative length



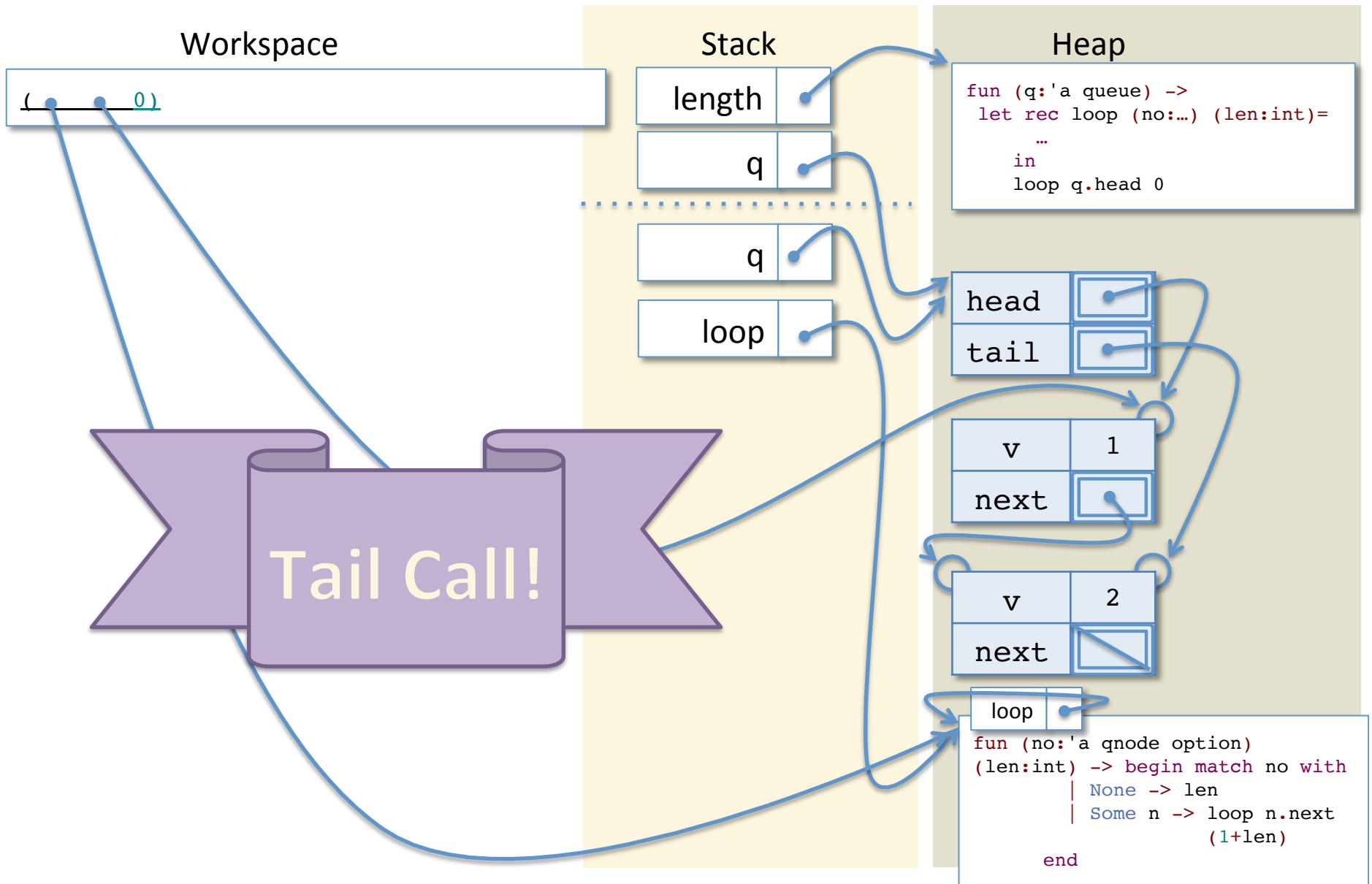
Tail Calls and Iterative length



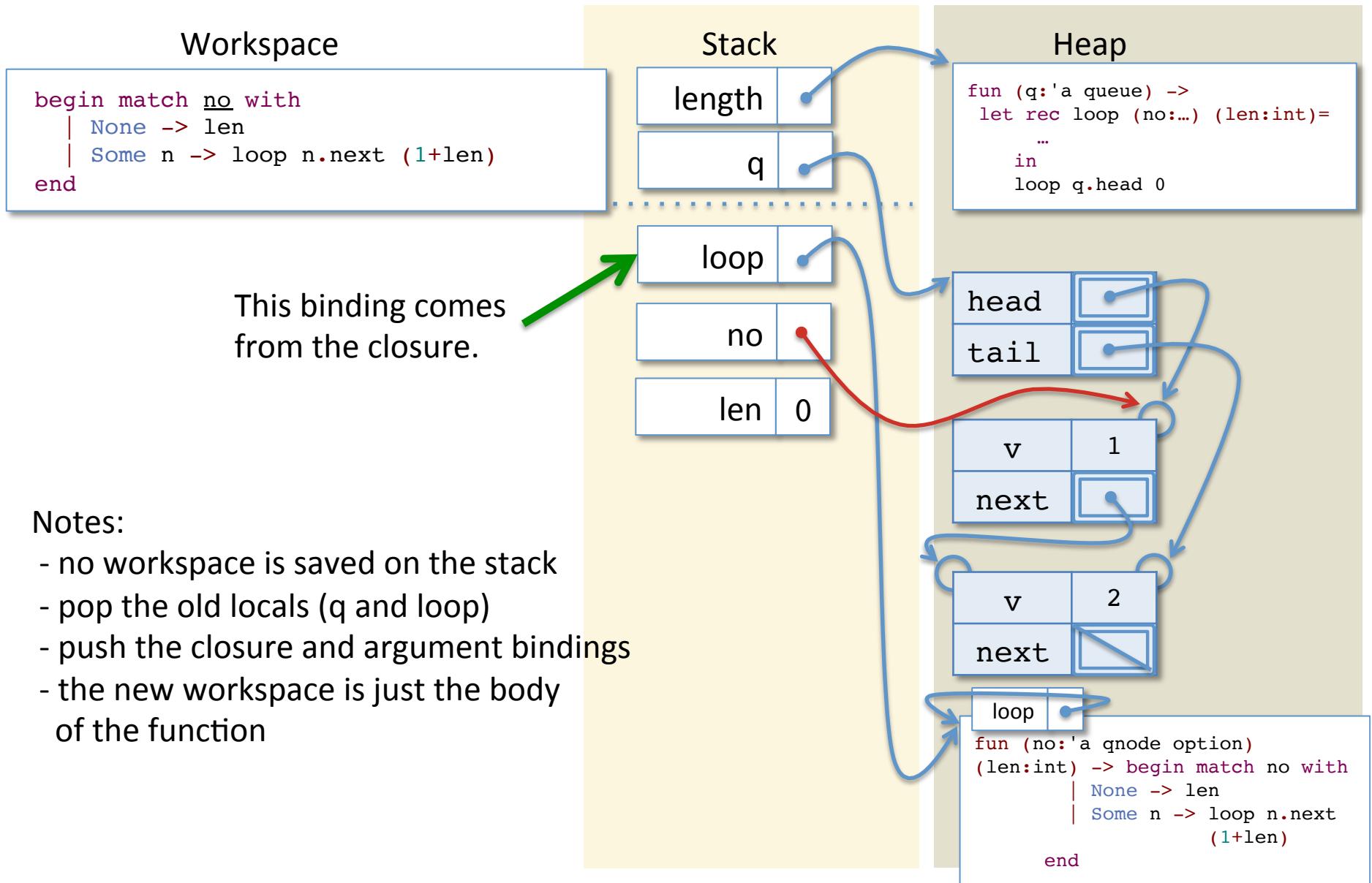
Tail Calls and Iterative length



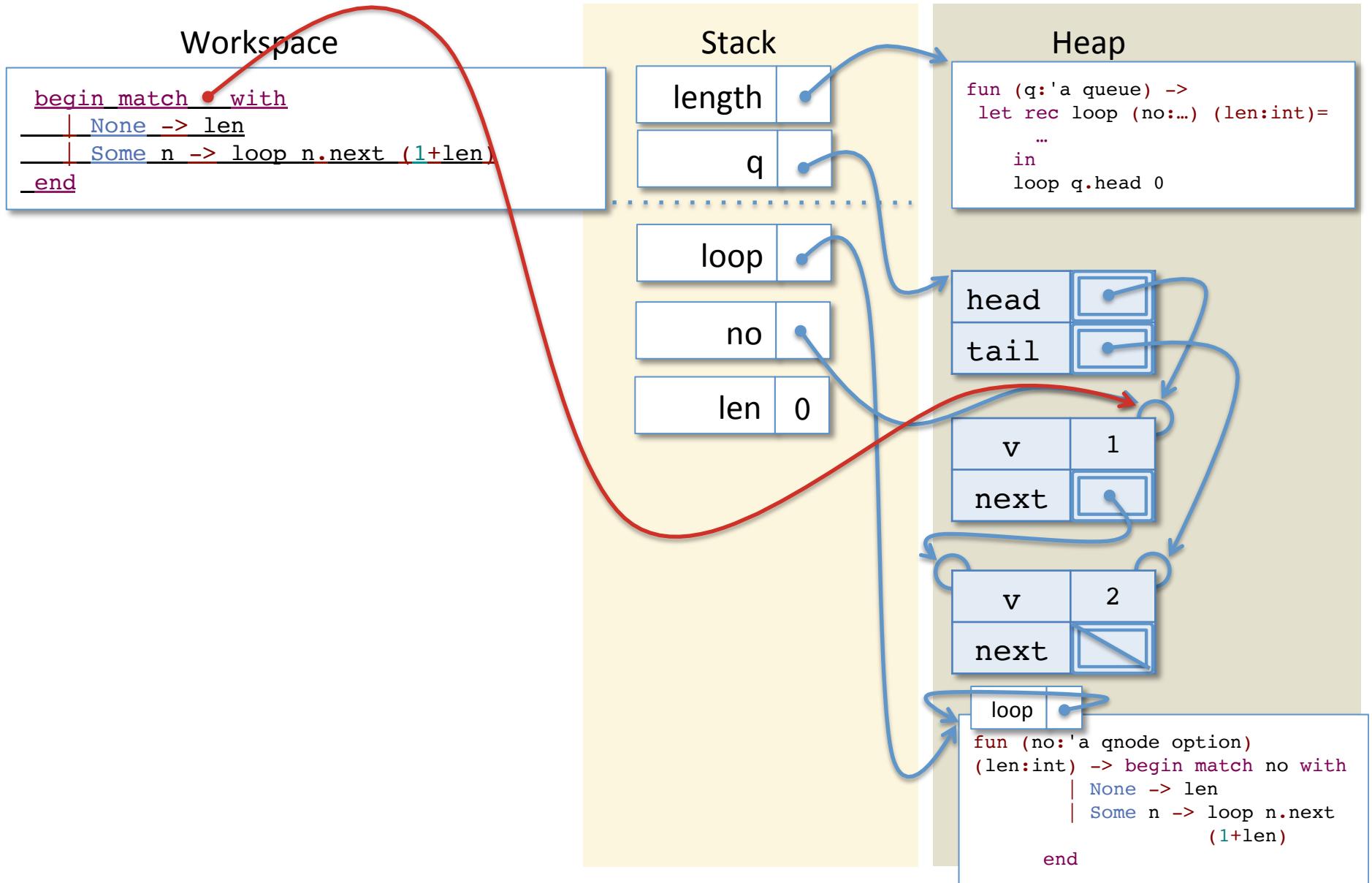
Tail Calls and Iterative length



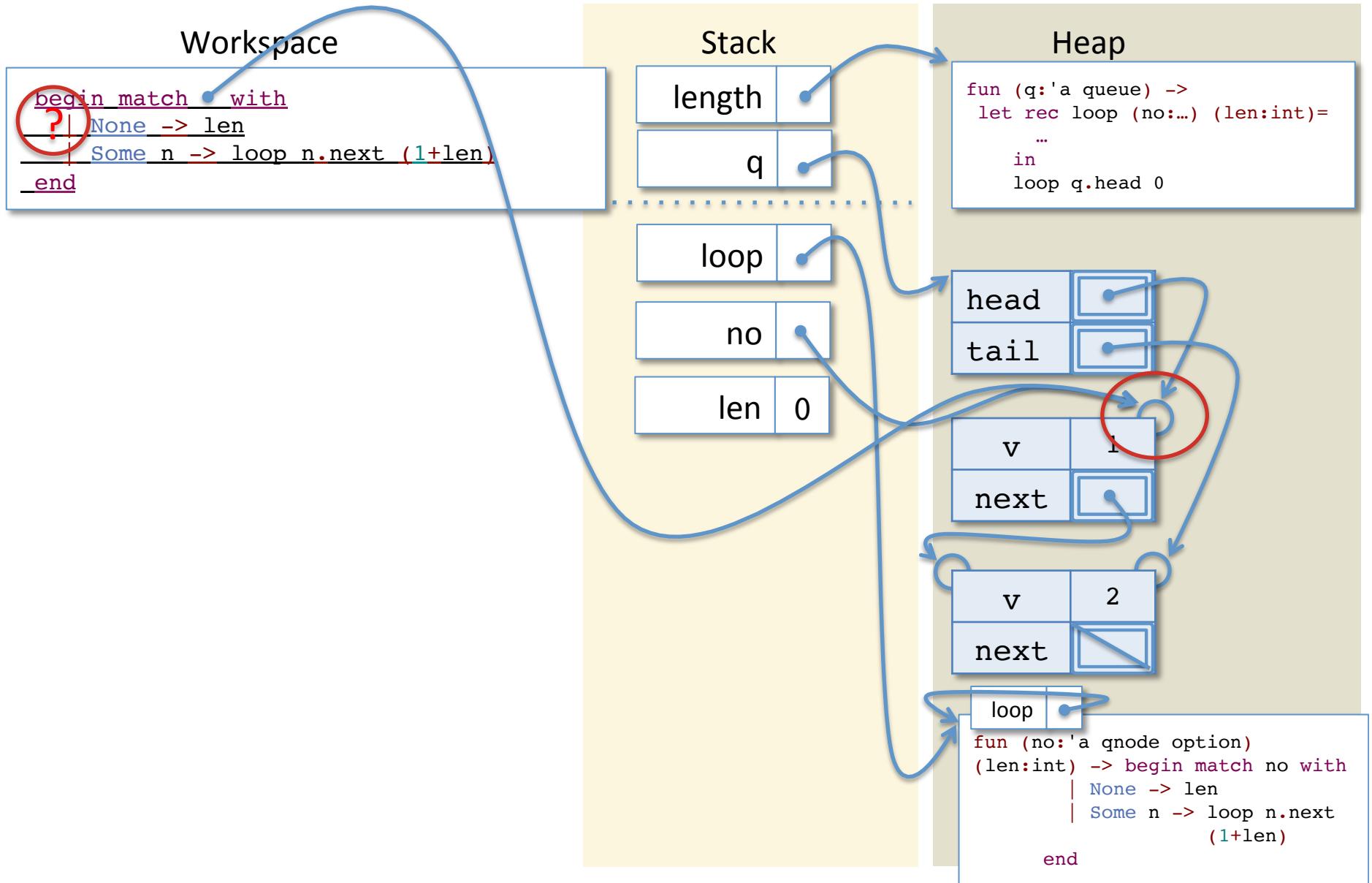
Tail Calls and Iterative length



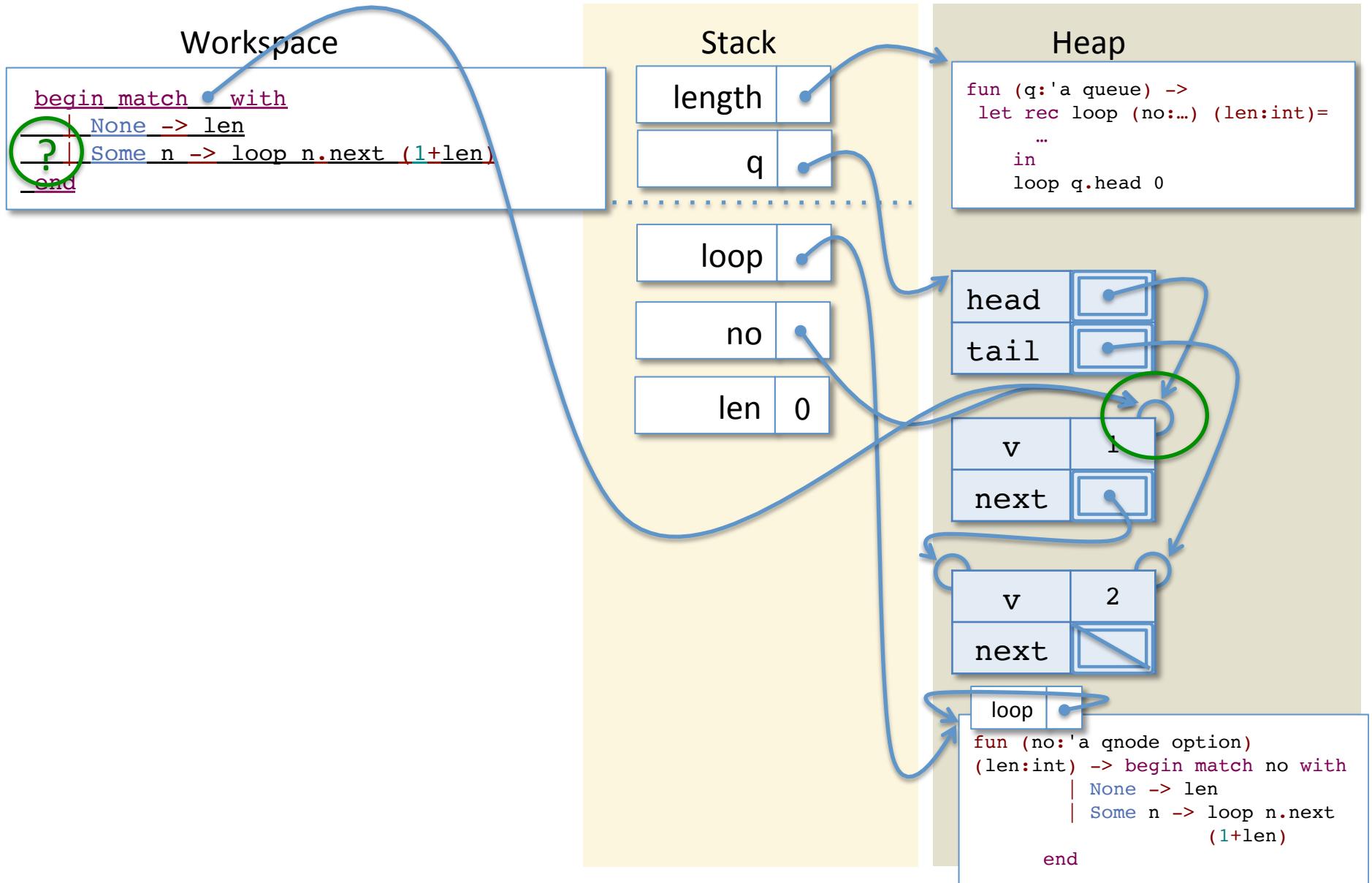
Tail Calls and Iterative length



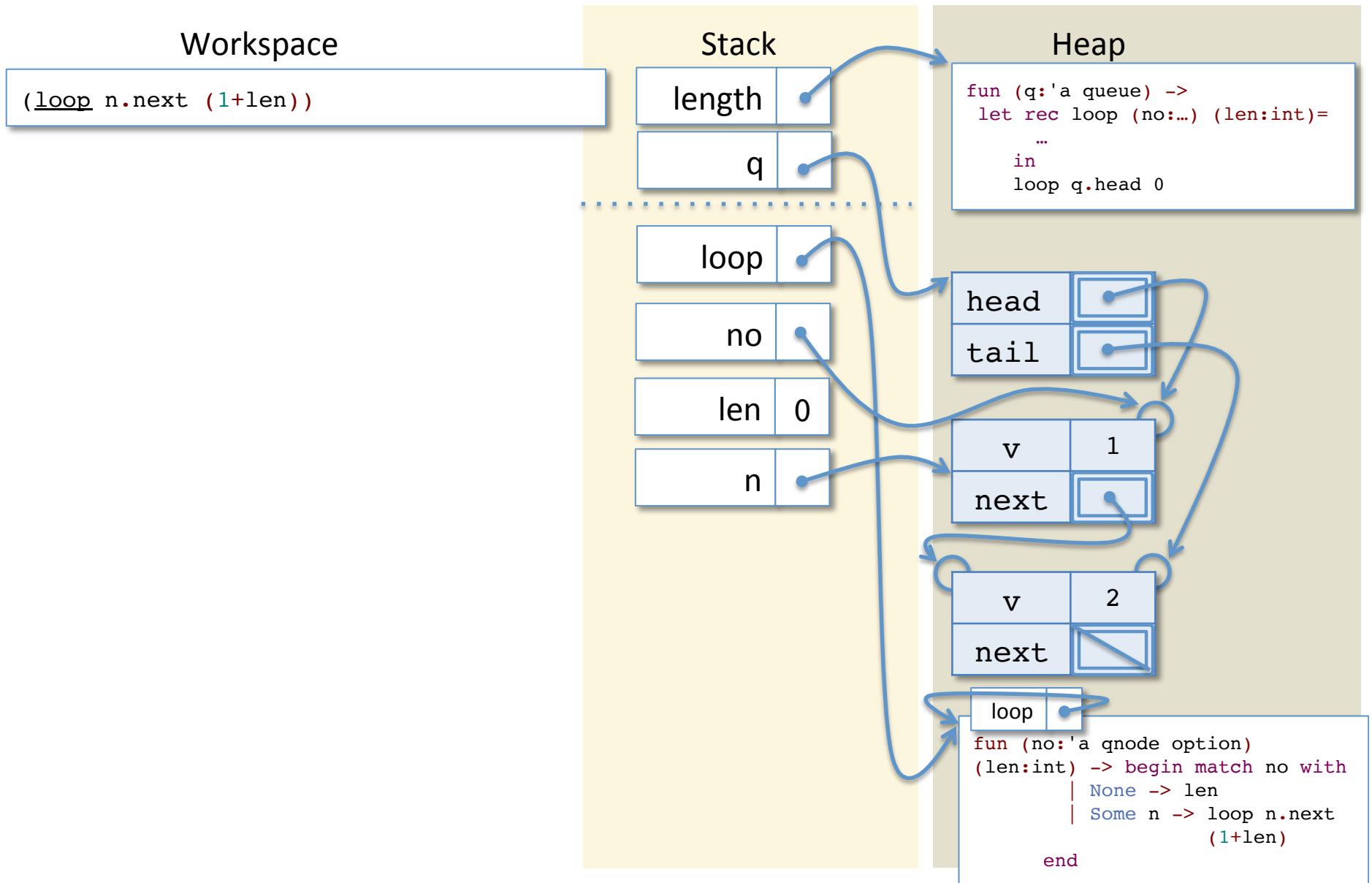
Tail Calls and Iterative length



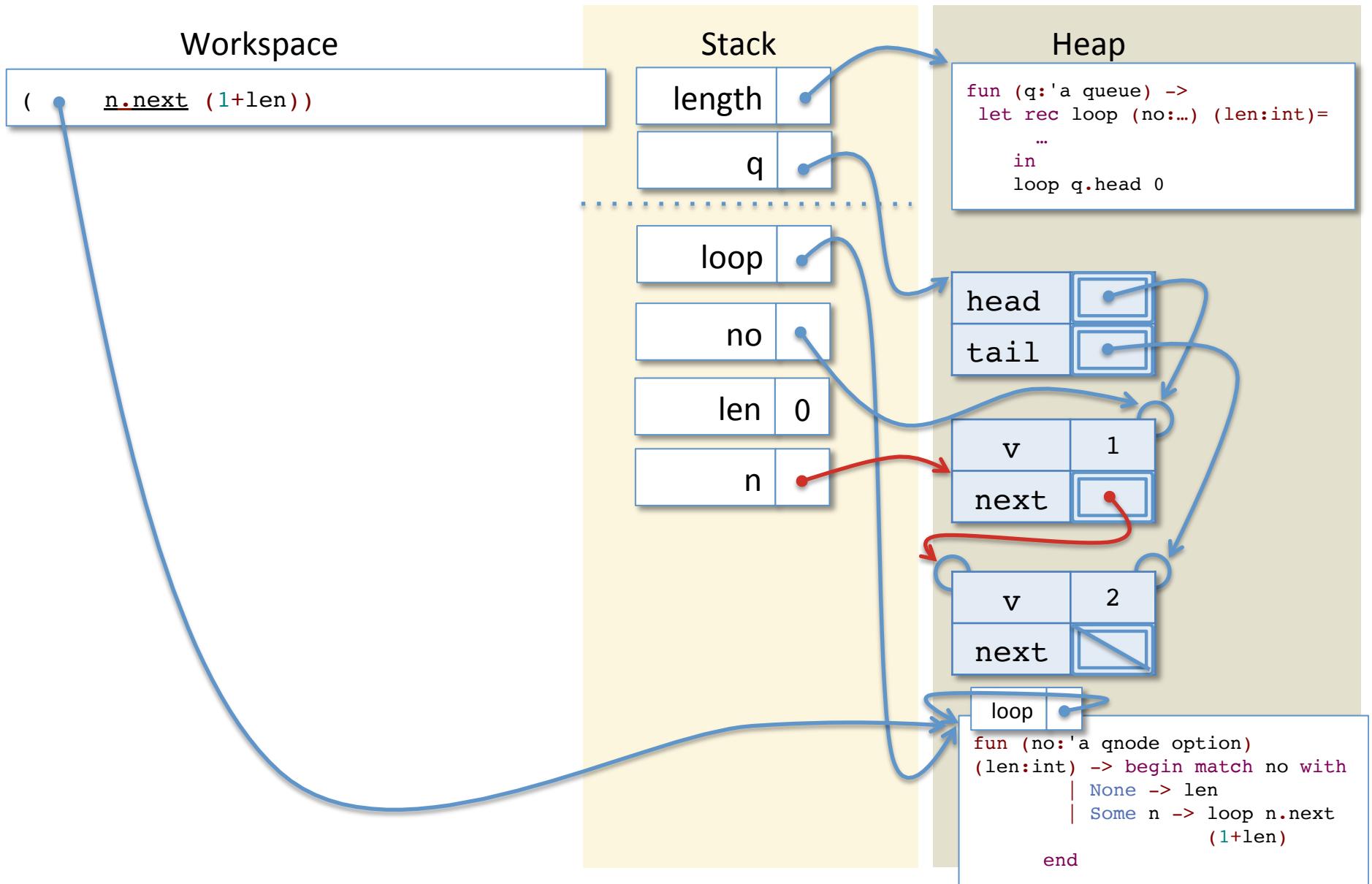
Tail Calls and Iterative length



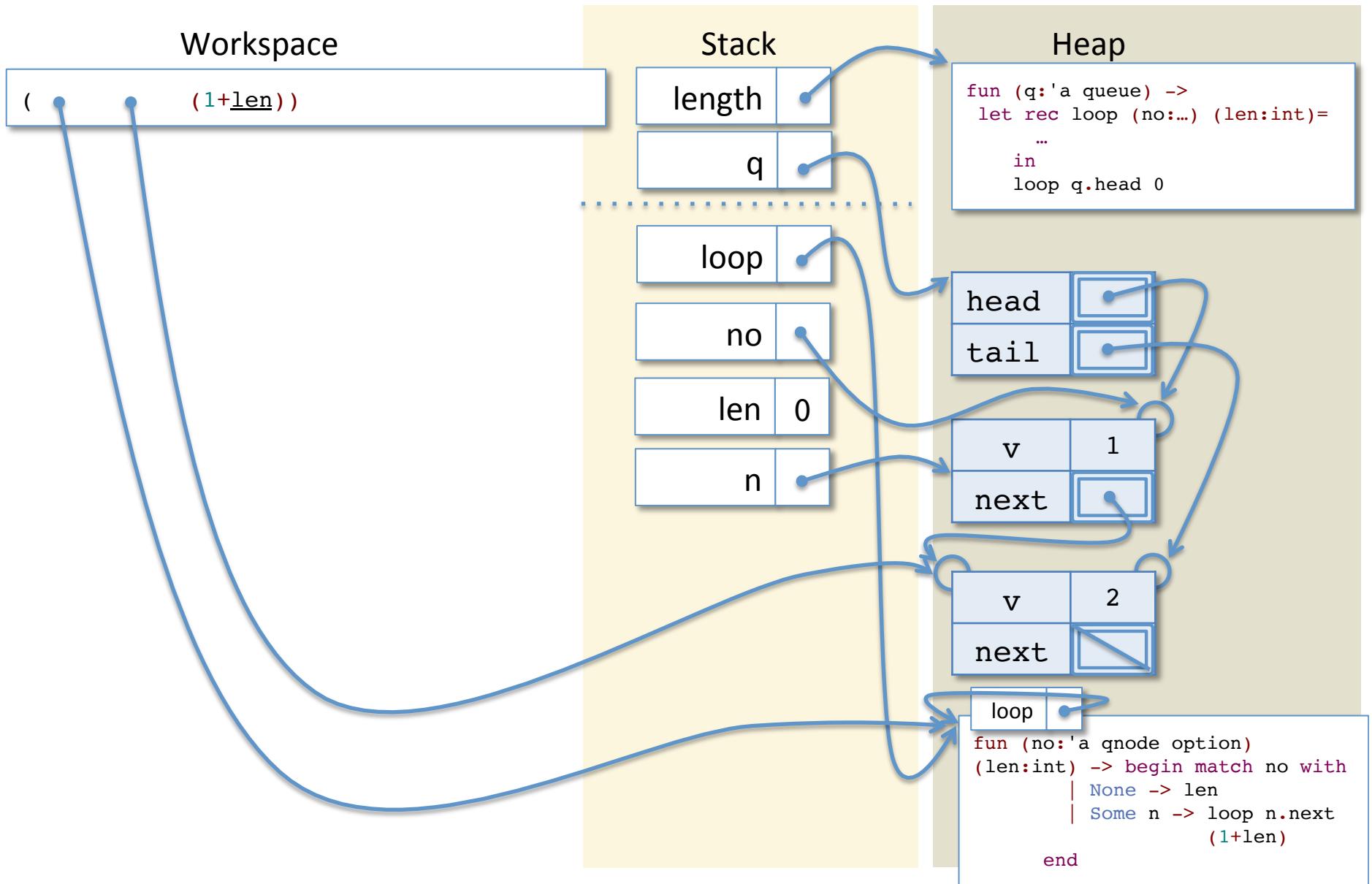
Tail Calls and Iterative length



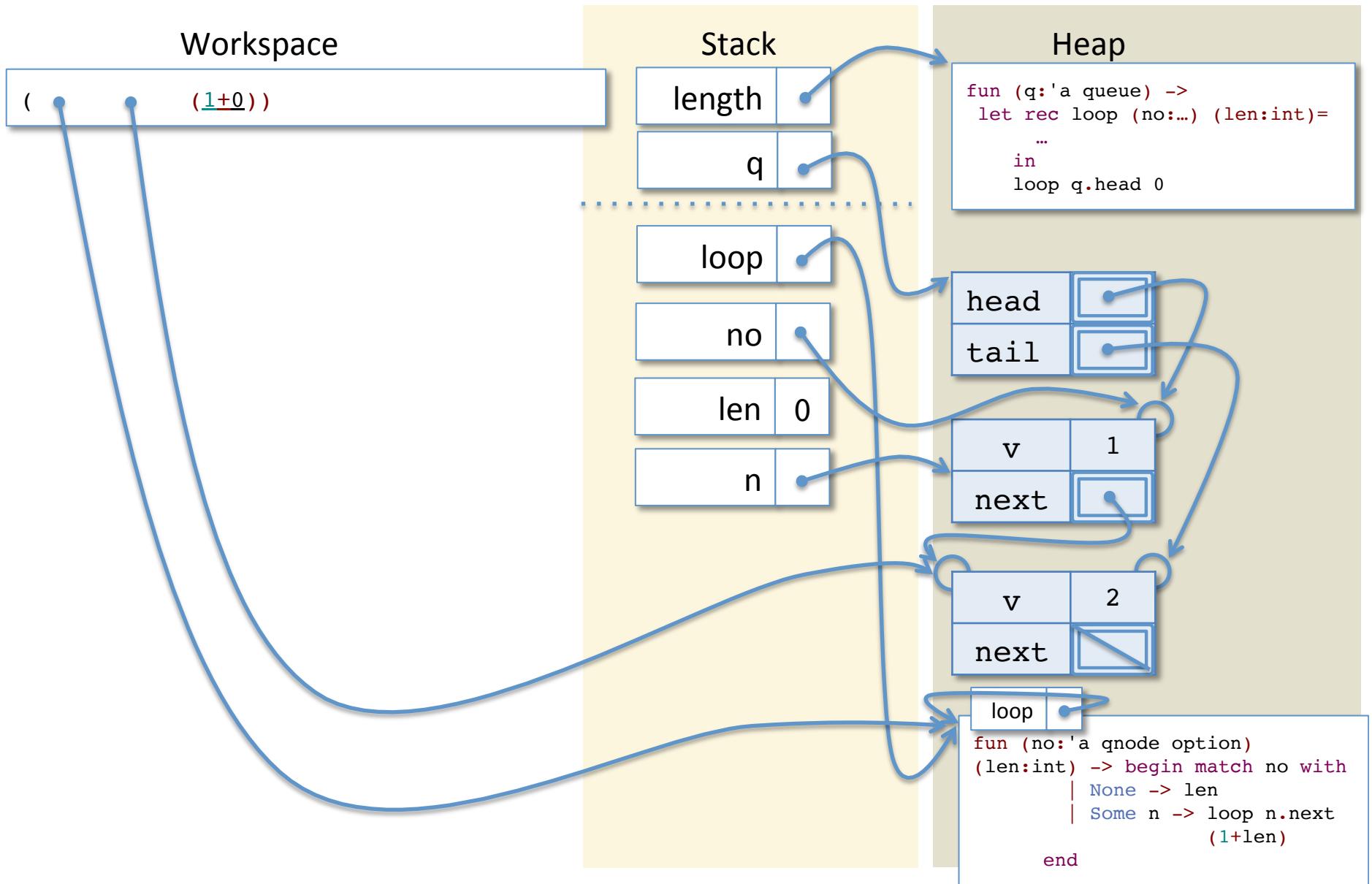
Tail Calls and Iterative length



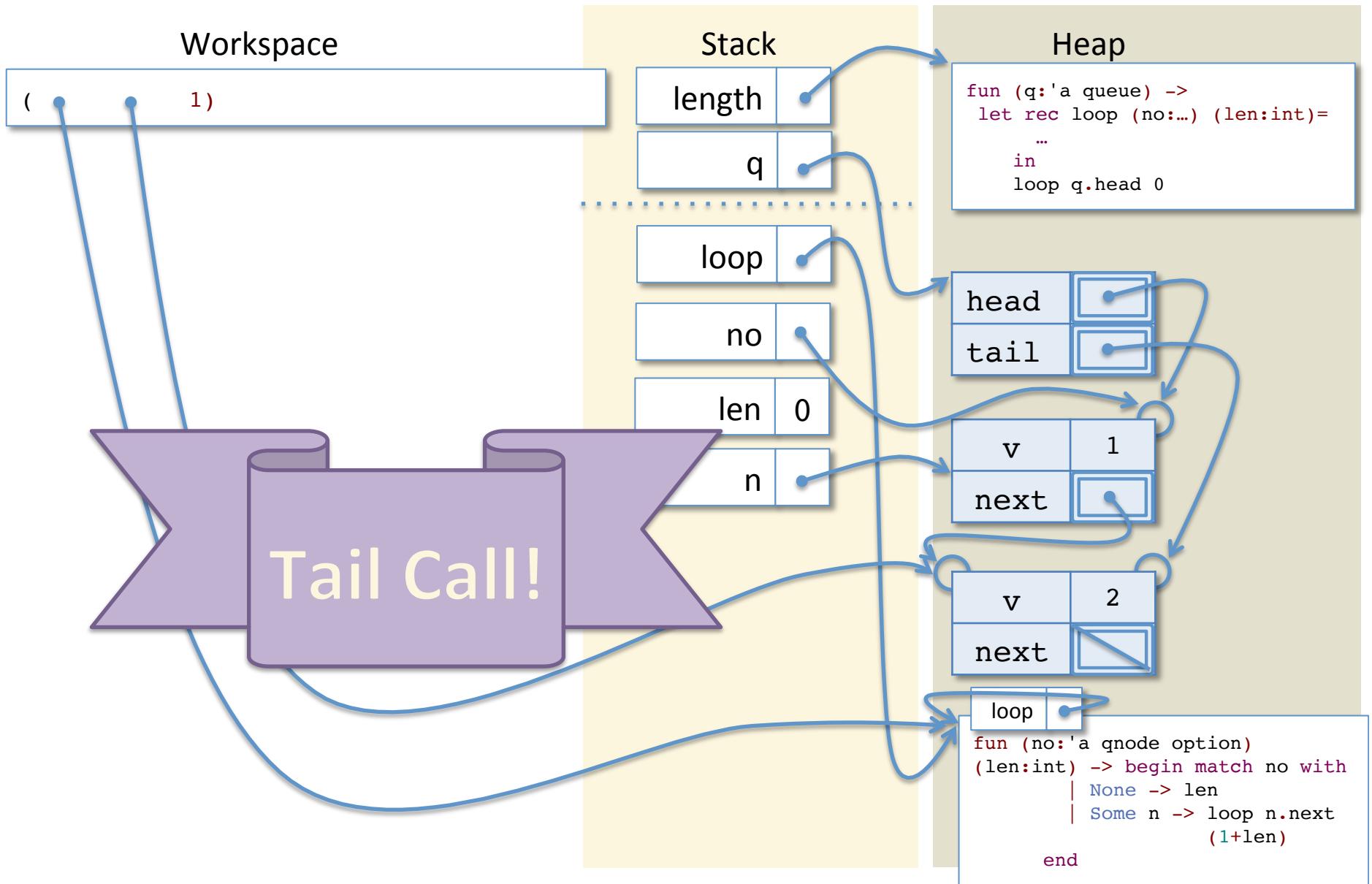
Tail Calls and Iterative length



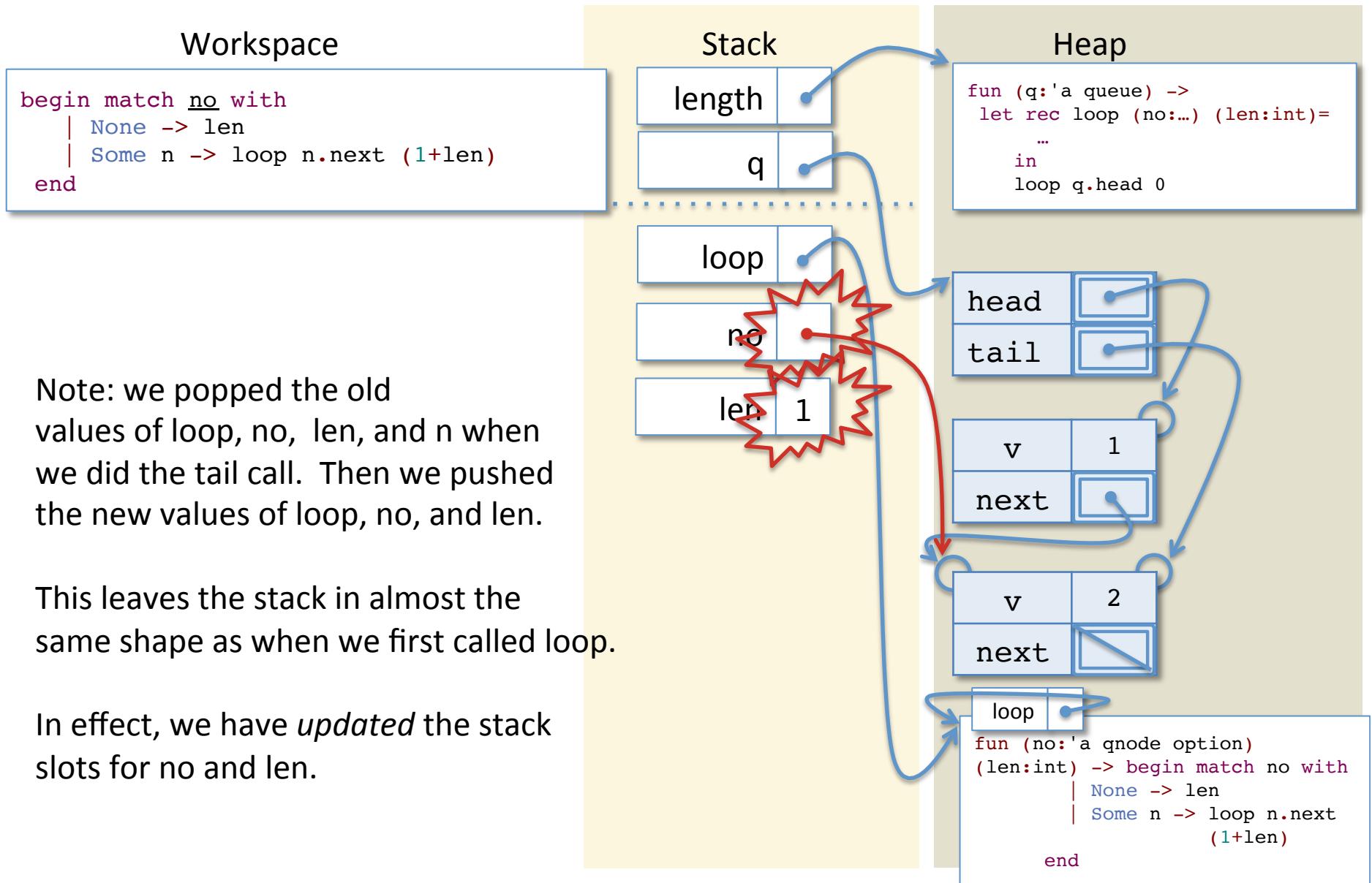
Tail Calls and Iterative length



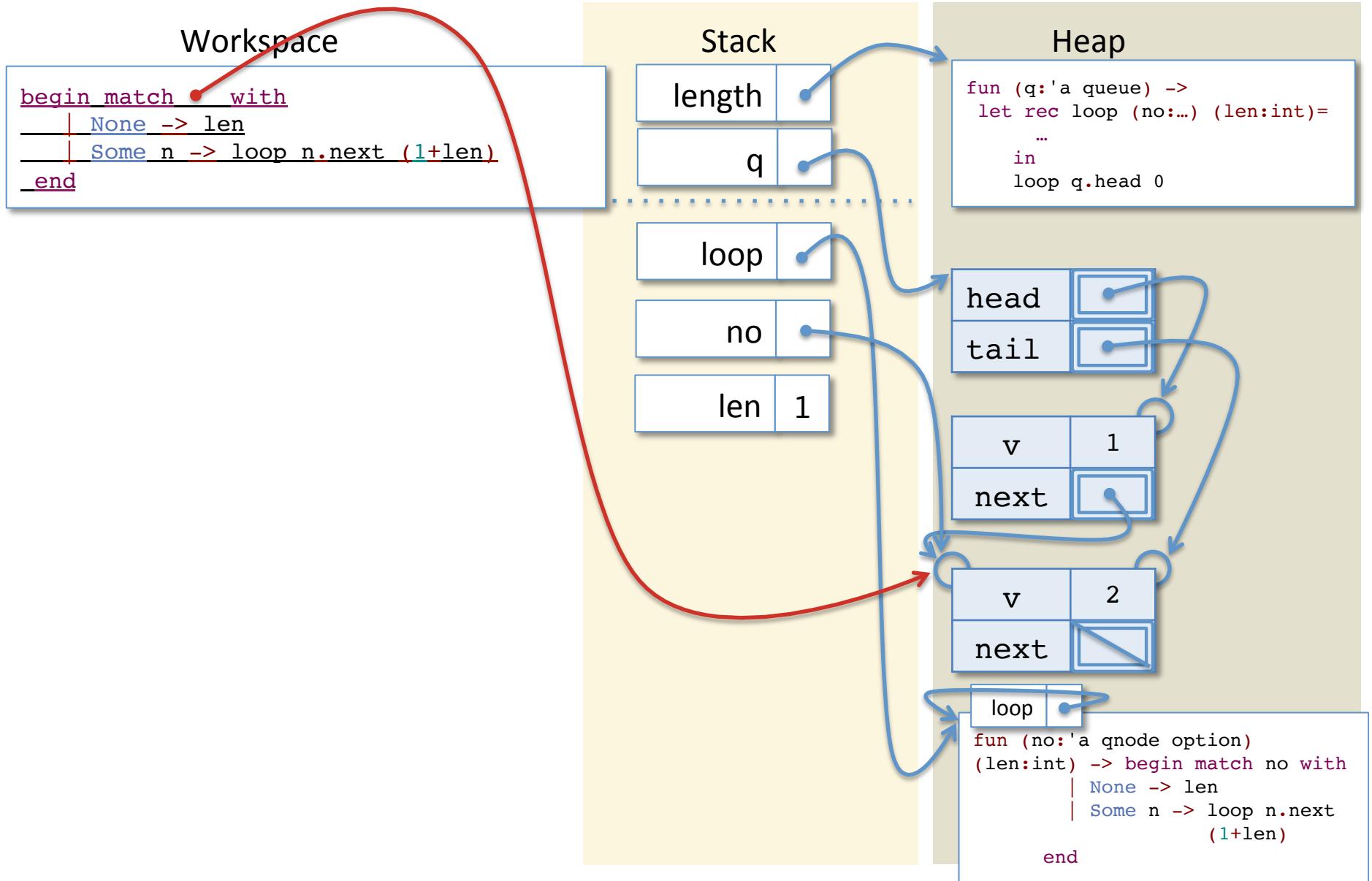
Tail Calls and Iterative length



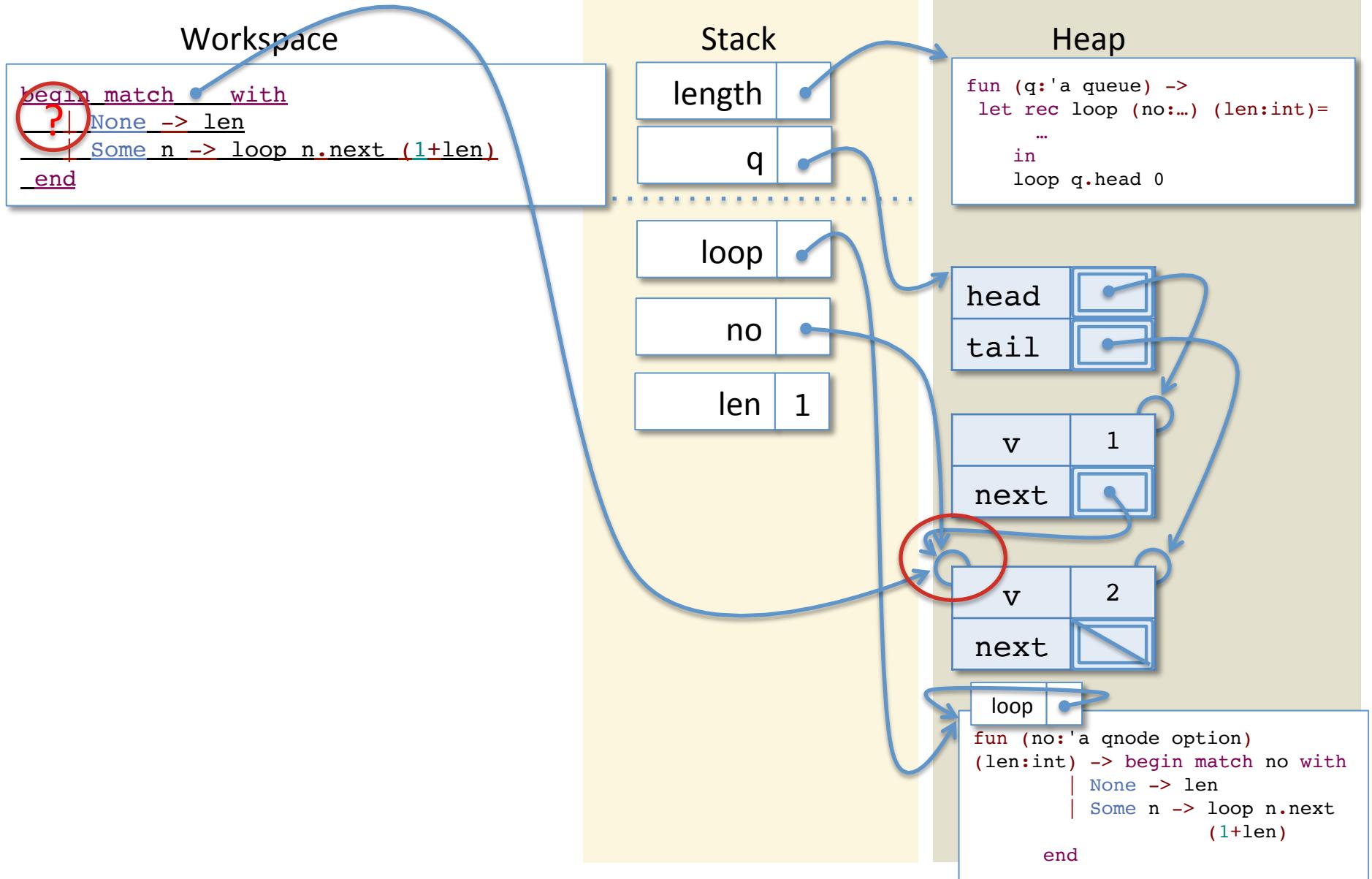
Tail Calls and Iterative length



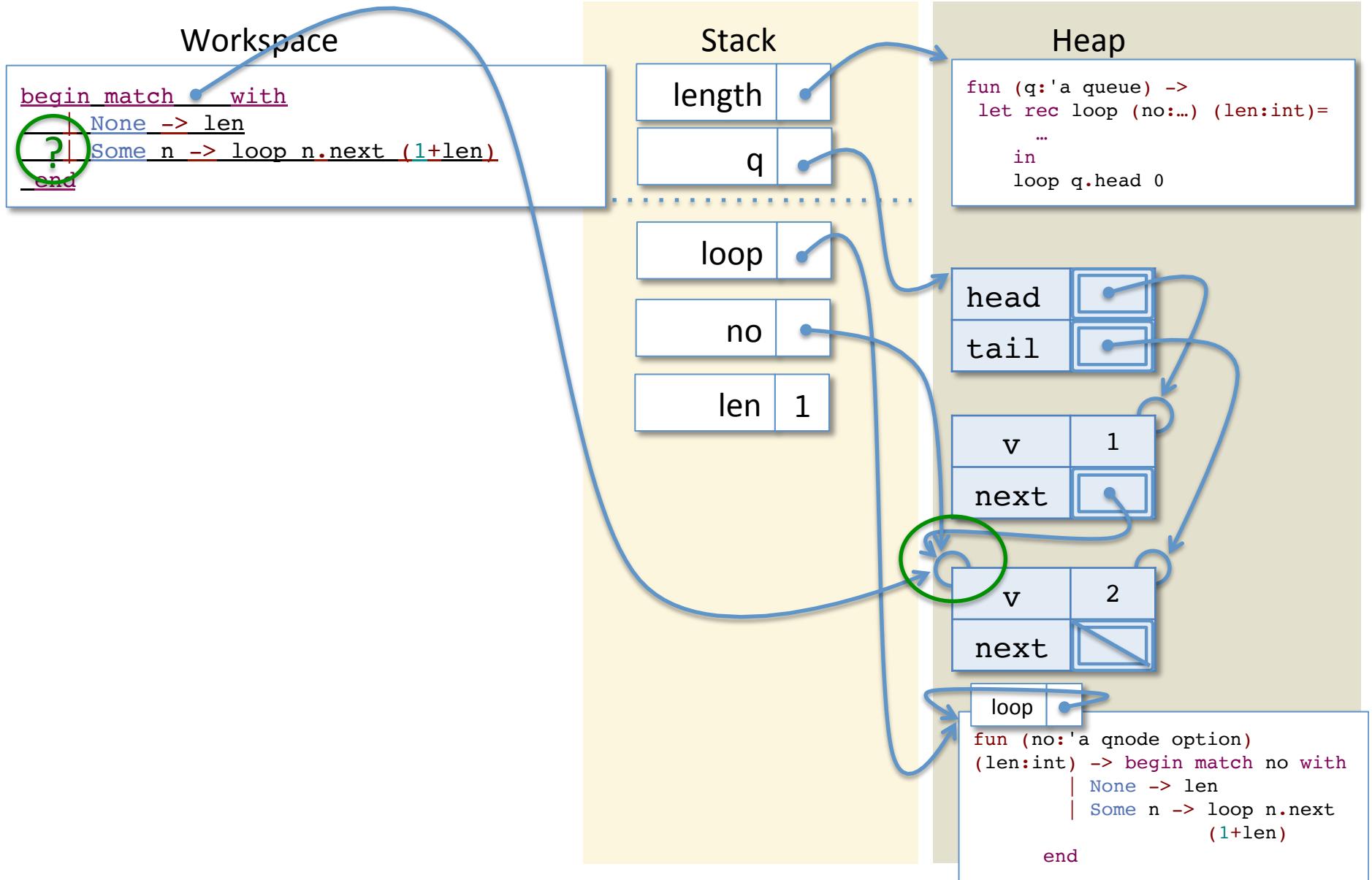
Tail Calls and Iterative length



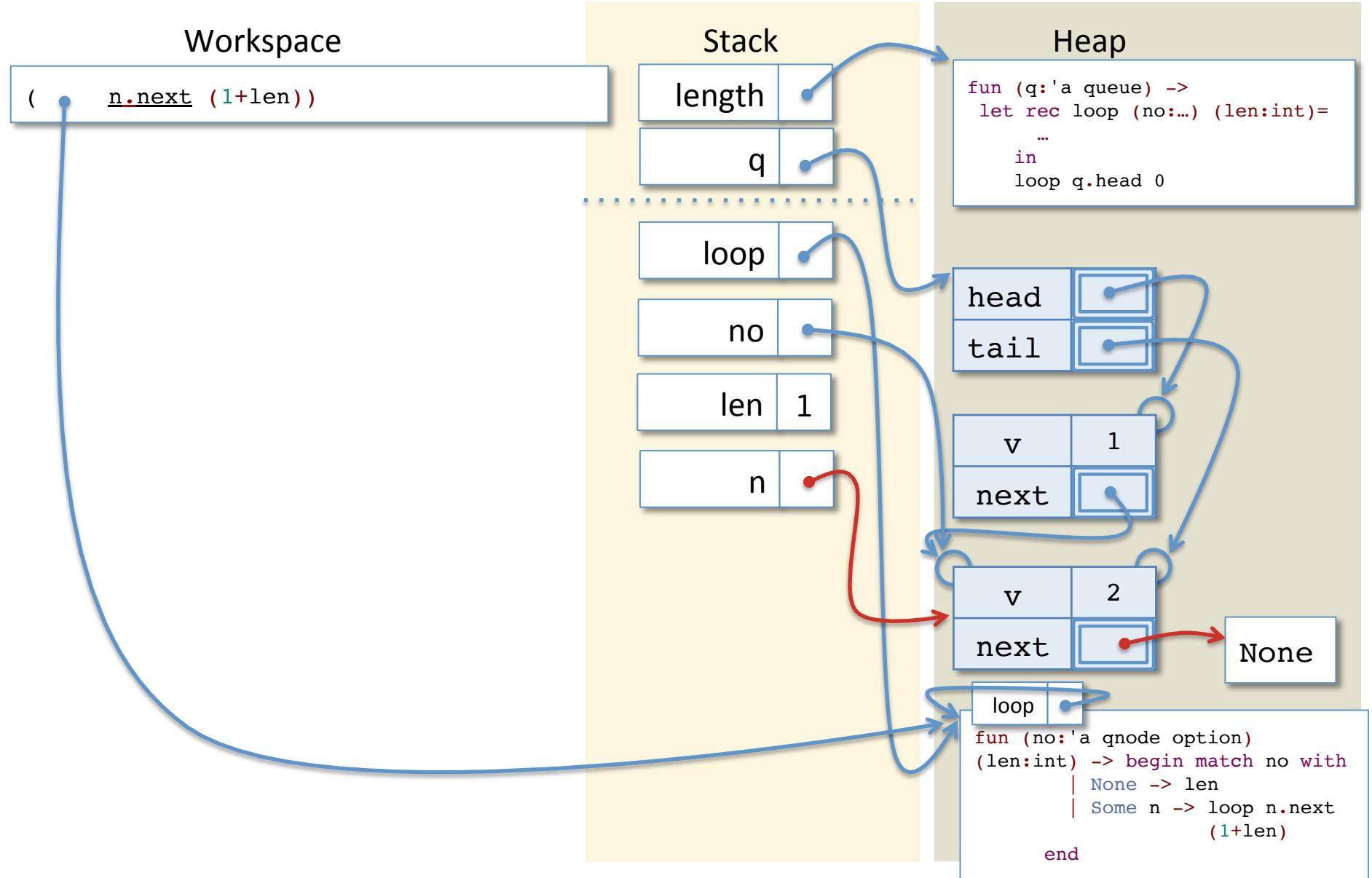
Tail Calls and Iterative length



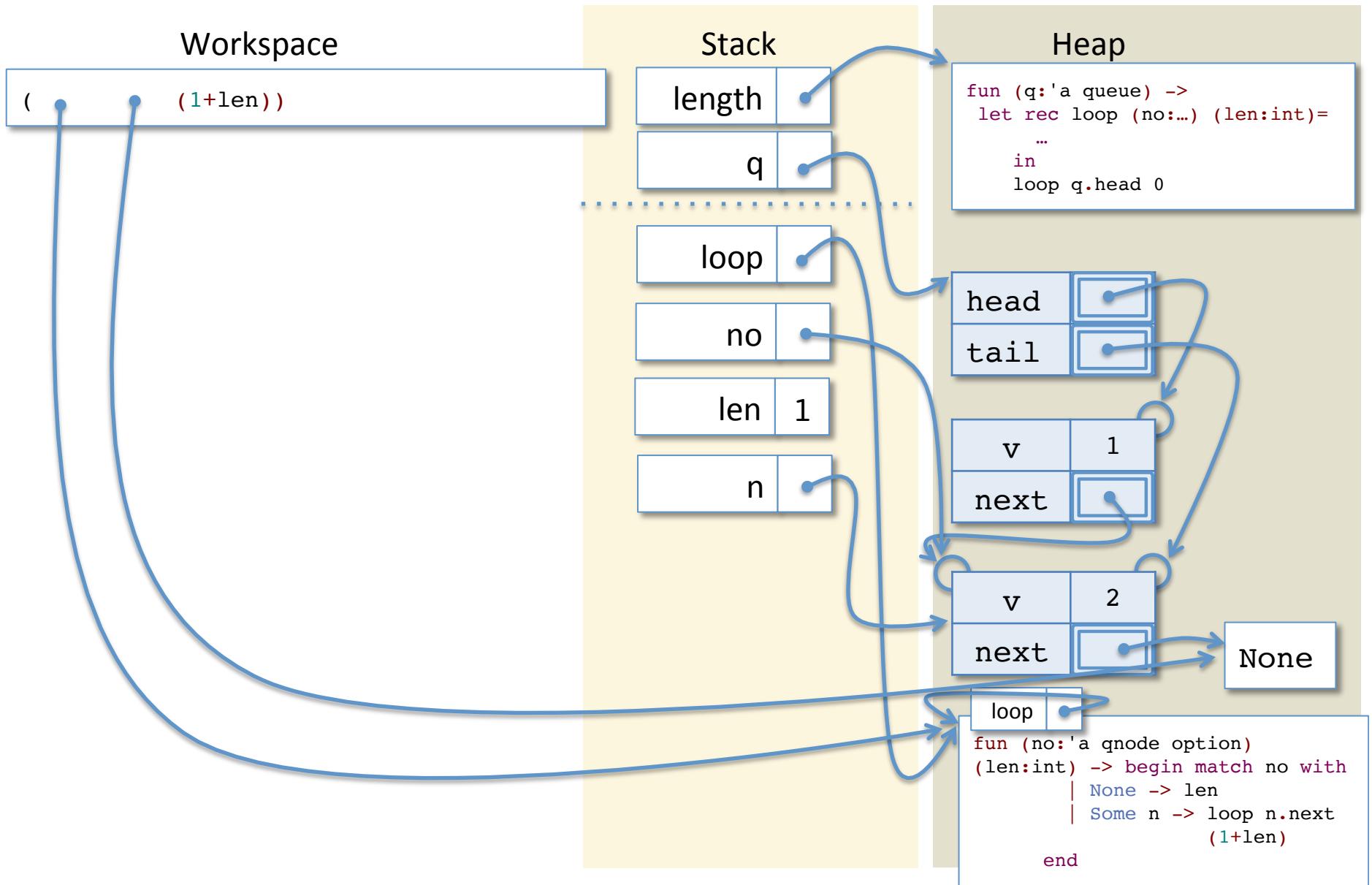
Tail Calls and Iterative length



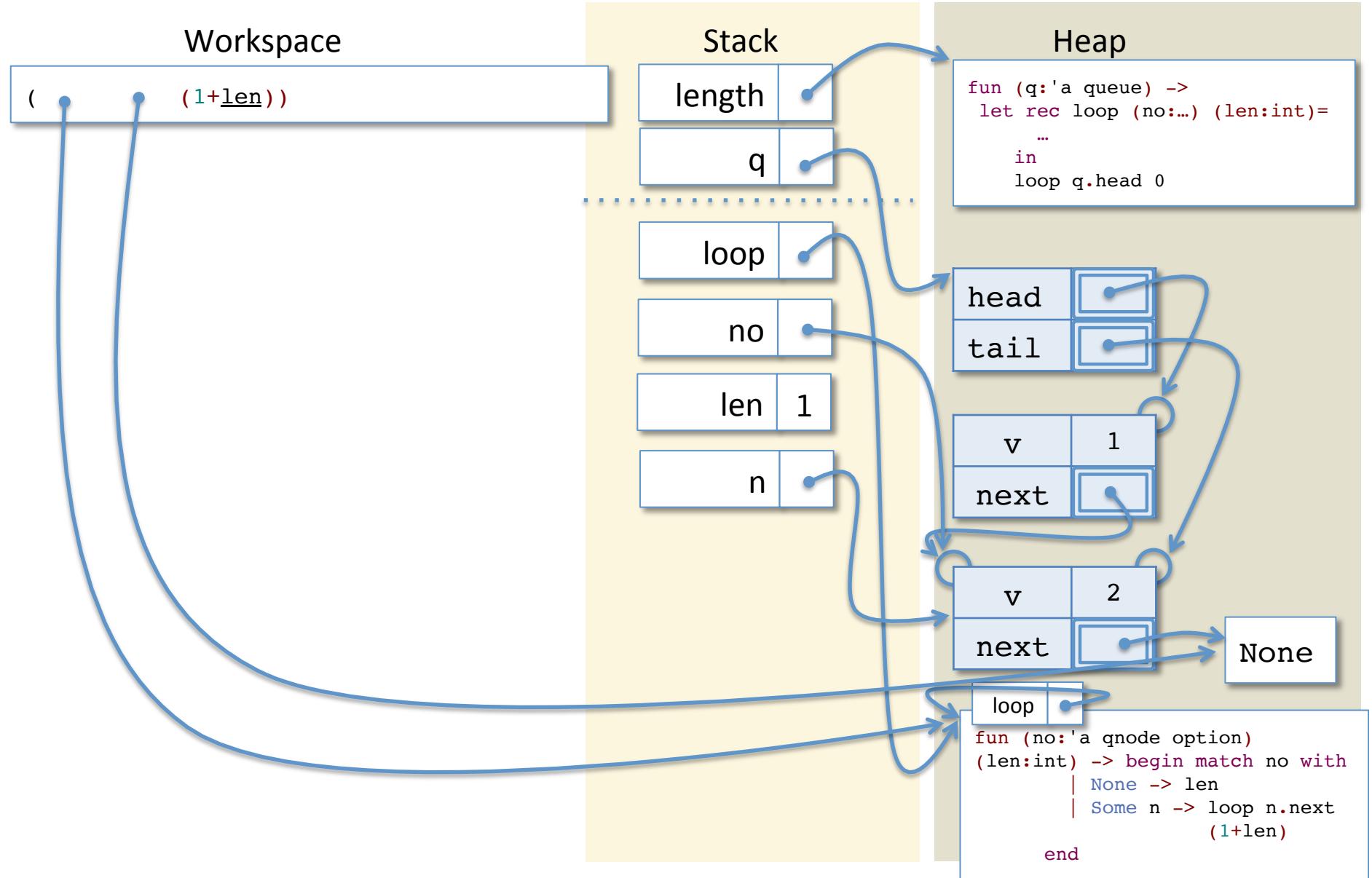
Tail Calls and Iterative length



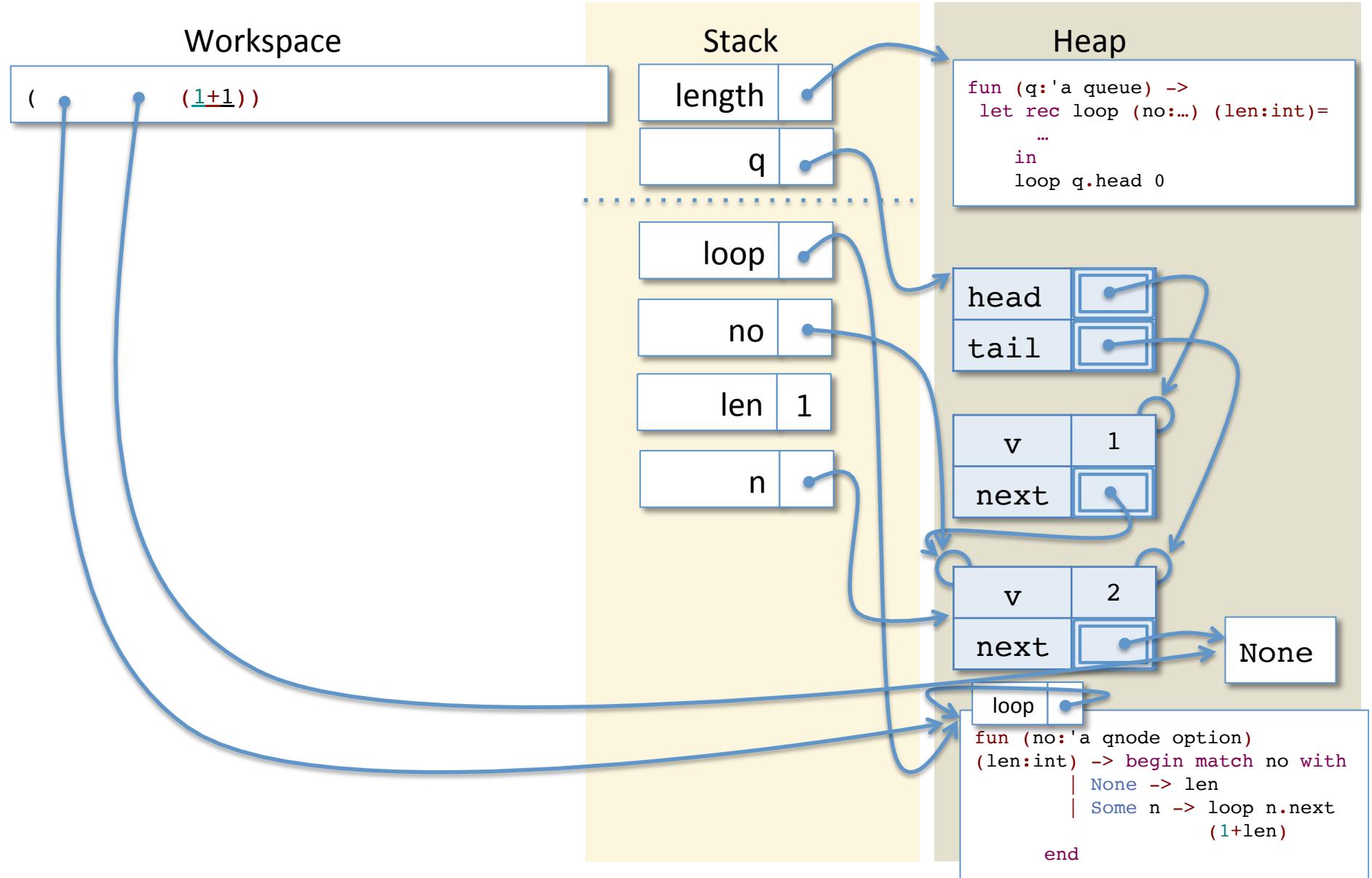
Tail Calls and Iterative length



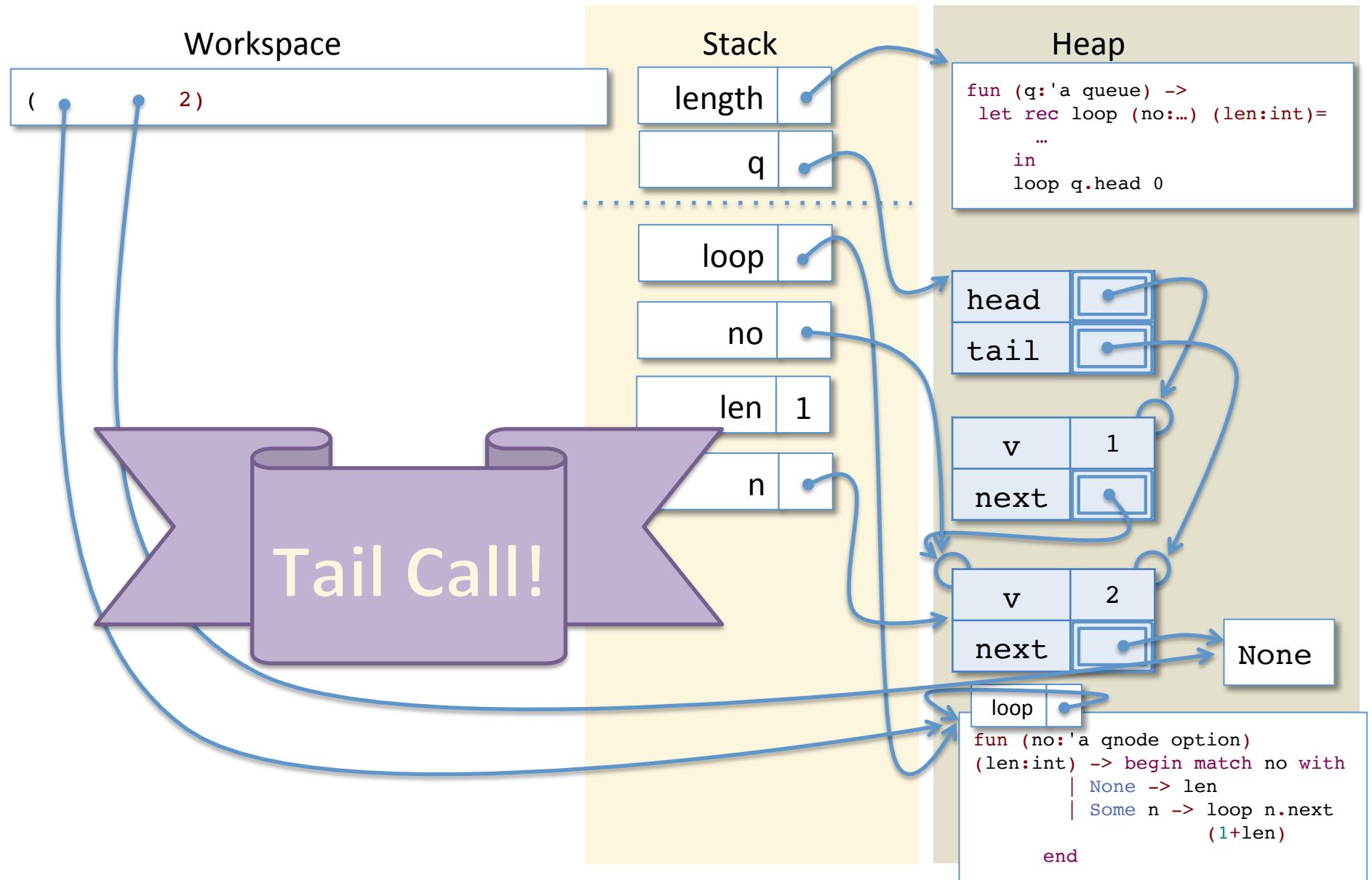
Tail Calls and Iterative length



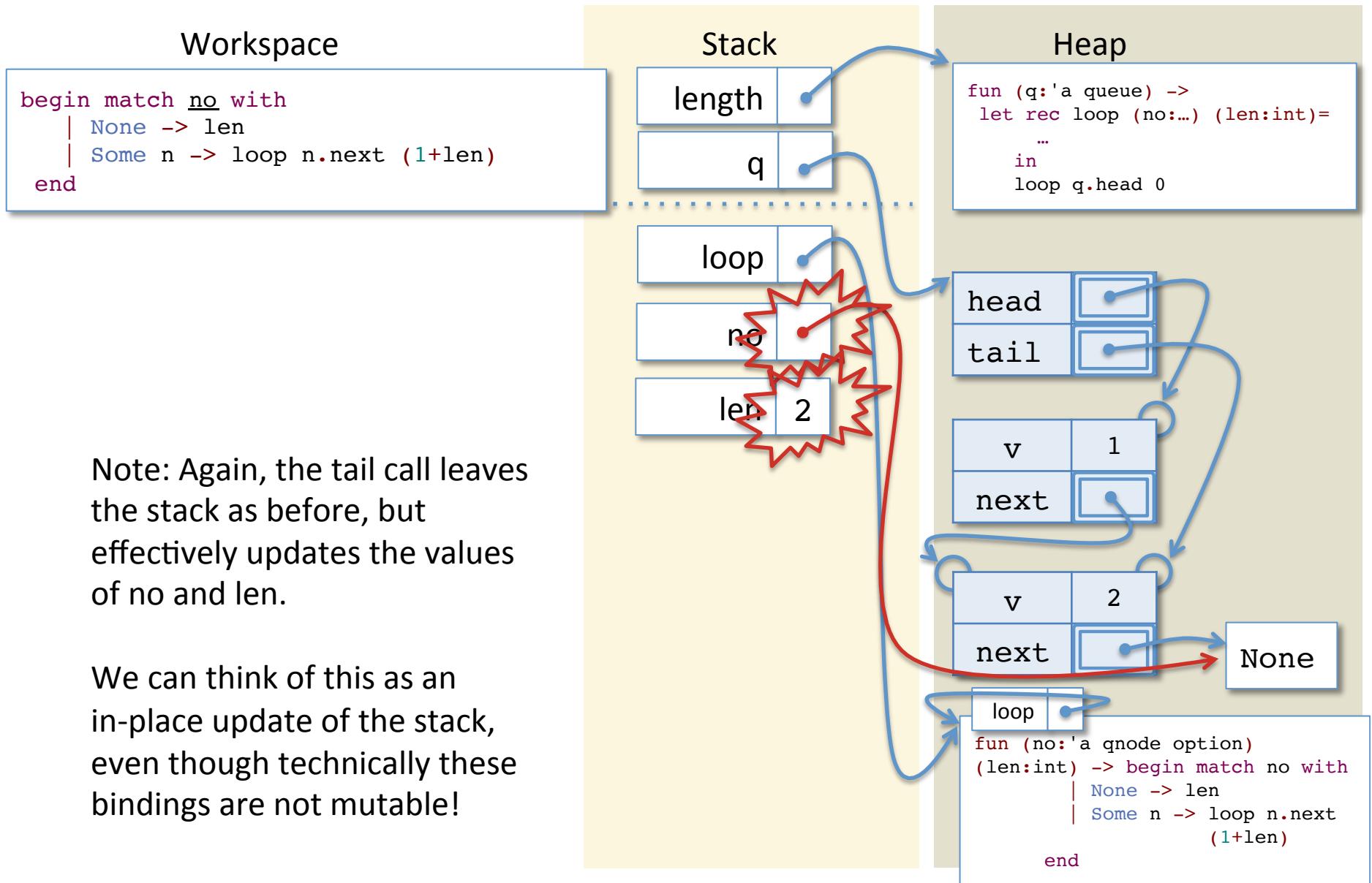
Tail Calls and Iterative length



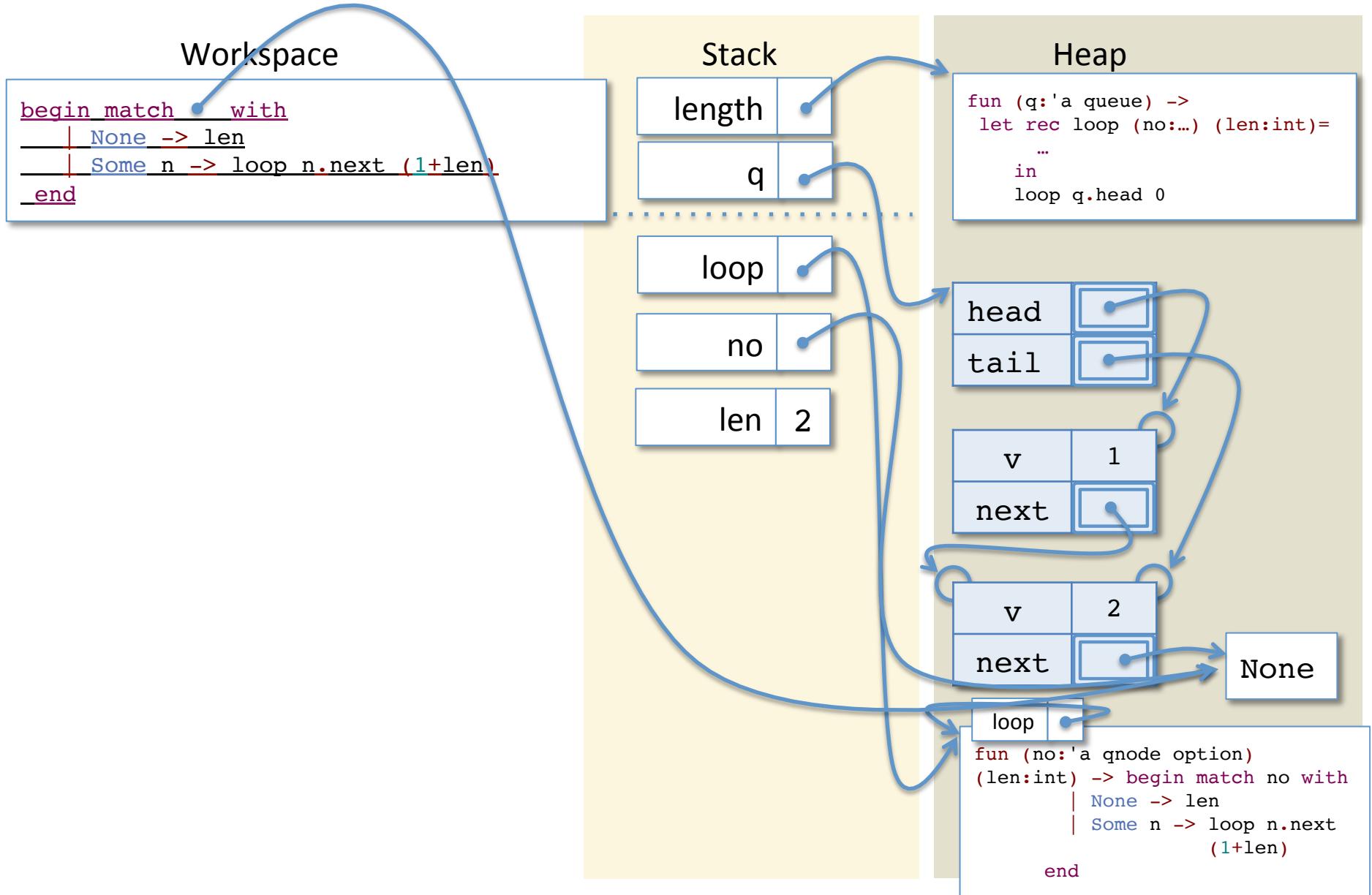
Tail Calls and Iterative length



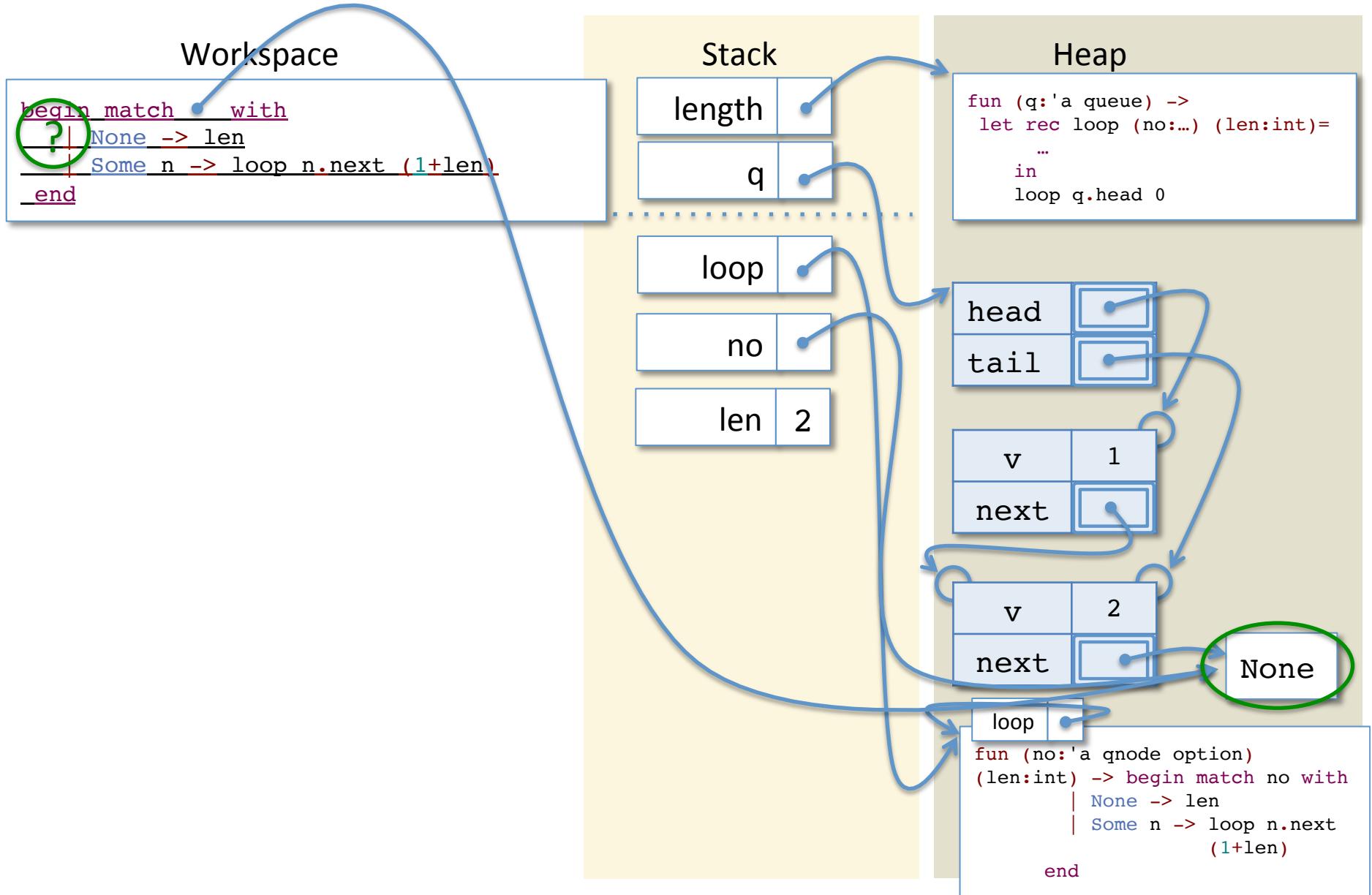
Tail Calls and Iterative length



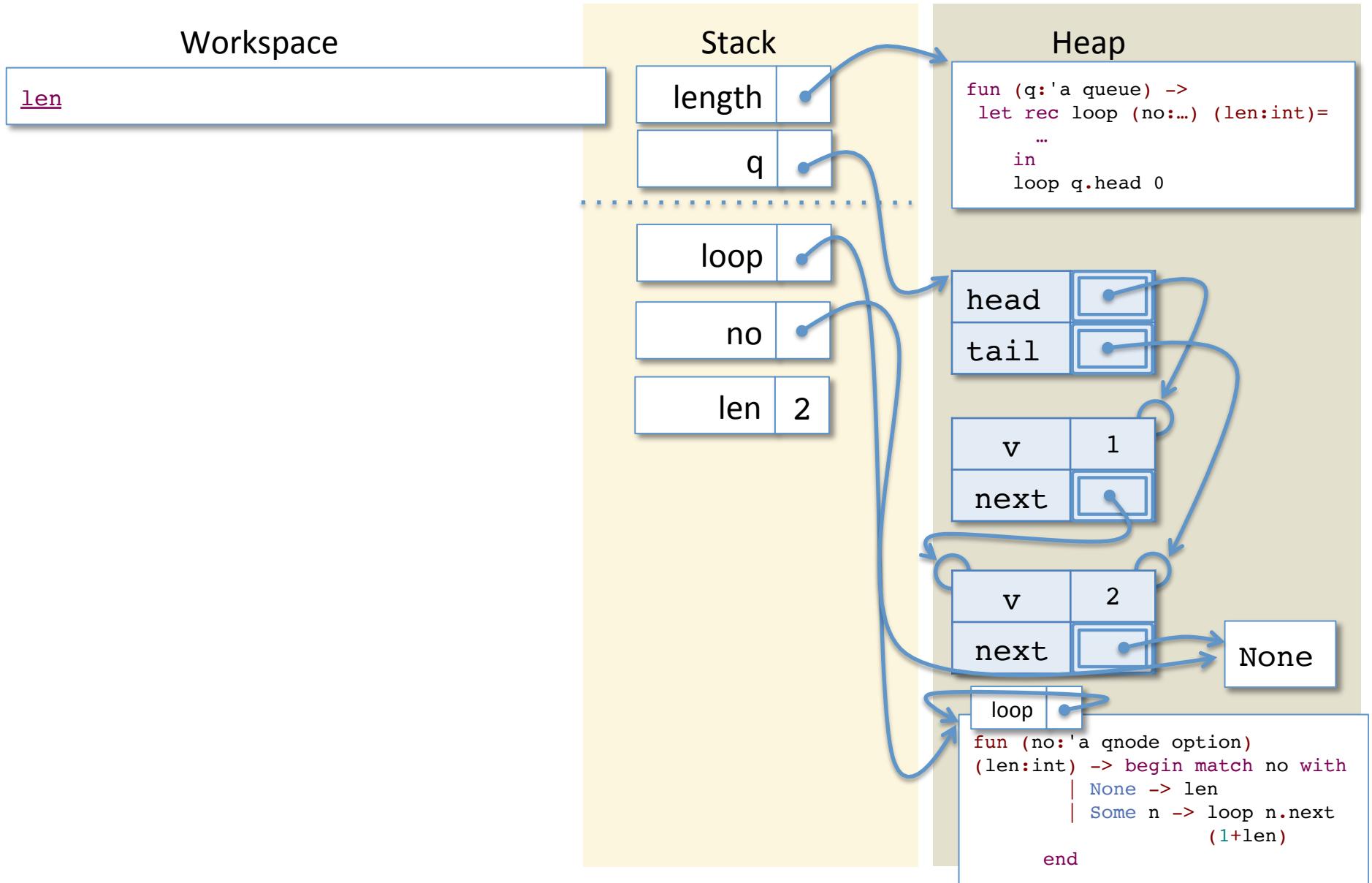
Tail Calls and Iterative length



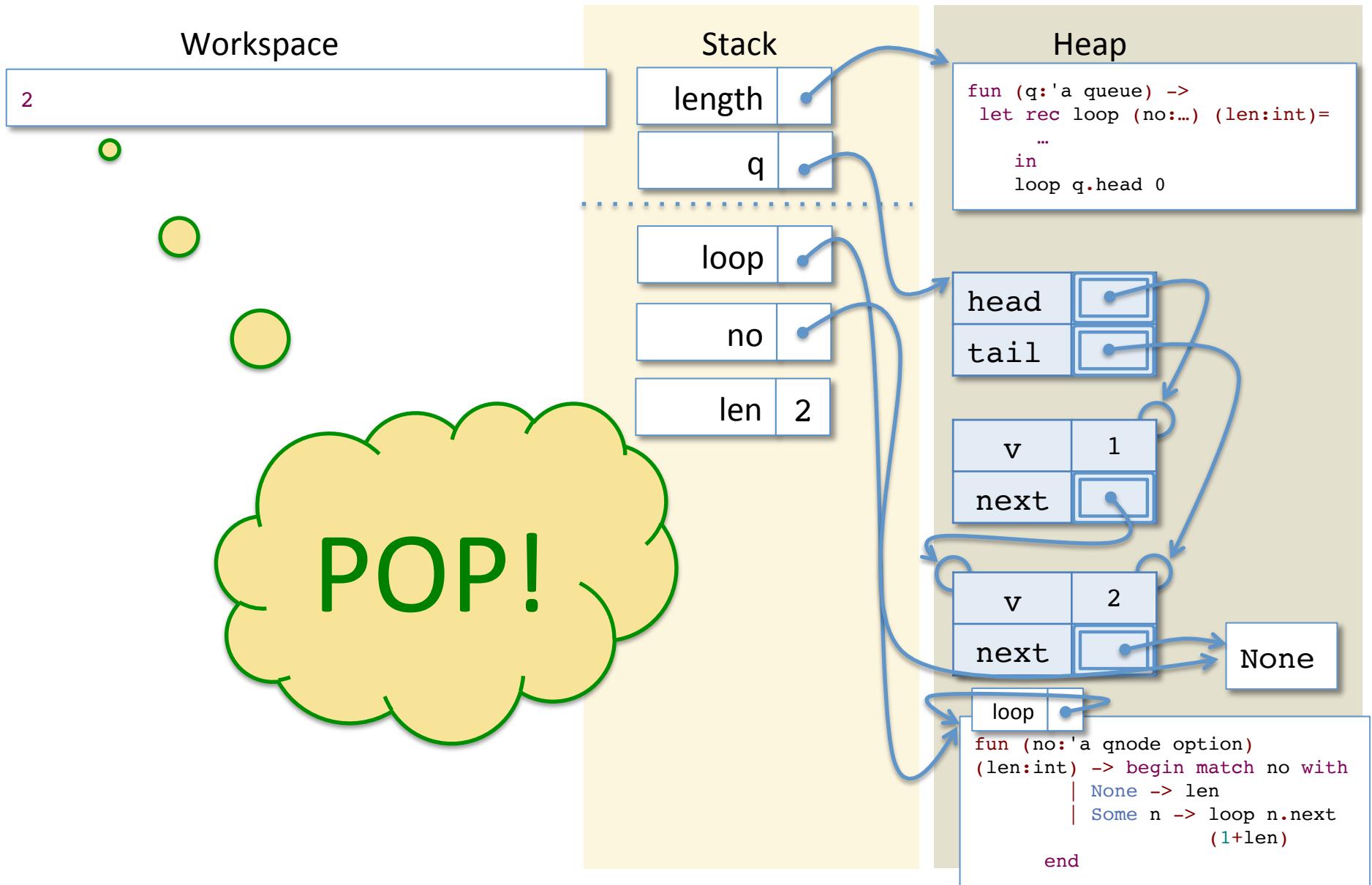
Tail Calls and Iterative length



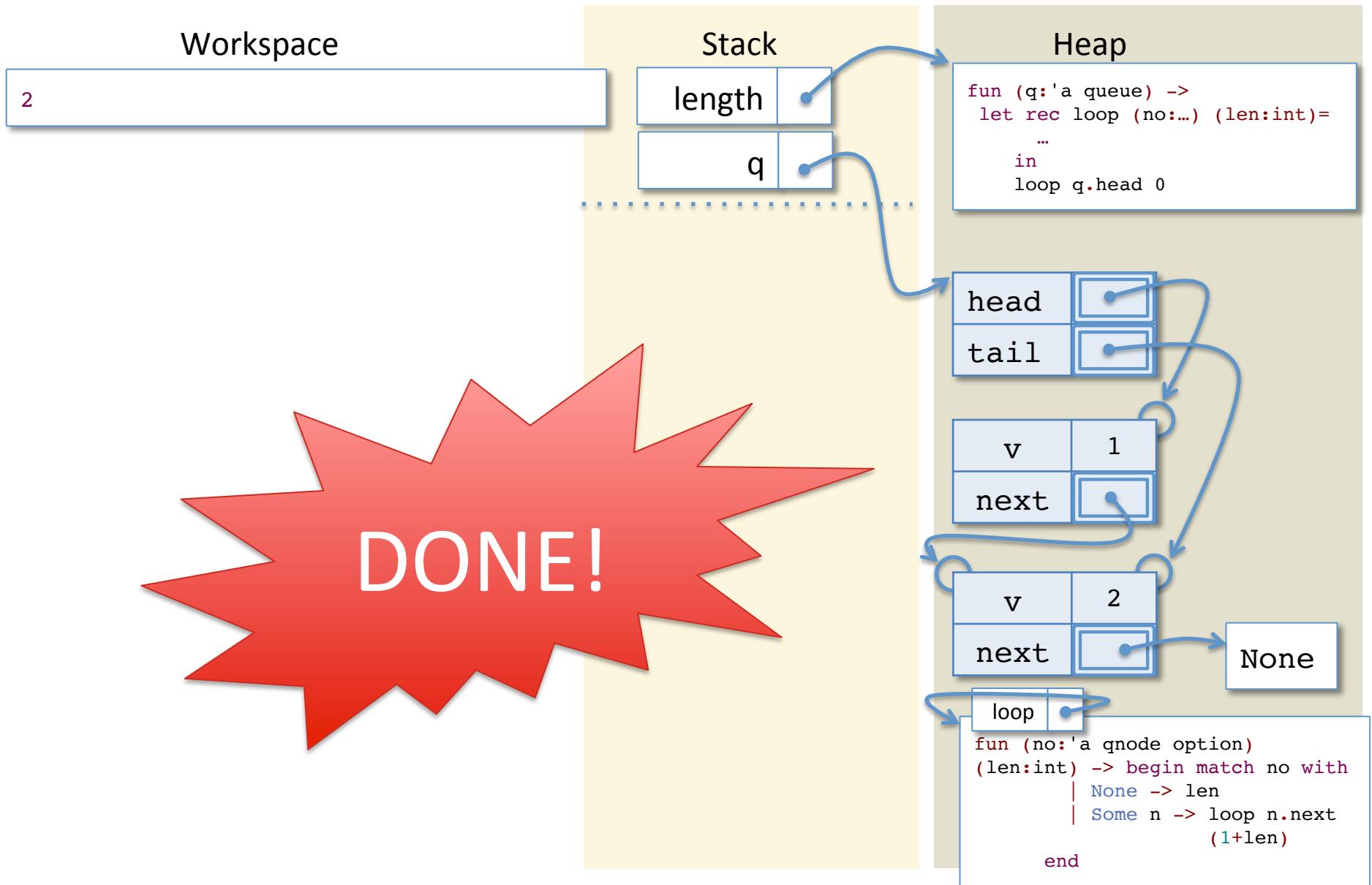
Tail Calls and Iterative length



Tail Calls and Iterative length



Tail Calls and Iterative length



Some Observations

- Tail call optimization lets the stack take only a fixed amount of space.
- The “recursive” call to loop effectively updates some of the stack bindings in place.
 - We can think of these bindings as the *state* being modified by each iteration of the loop.
- These two properties are the essence of iteration.
 - They are the difference between general recursion and iteration

What happens when you run this function on a (valid) queue containing 2 elements?

```
let f (q:'a queue) : int =
    let rec loop (qn:'a qnode option) : int =
        begin match qn with
            | None -> 0
            | Some n -> 1 + loop qn
        end
    in loop q.head
```

1. The value 2 is returned
2. The value 0 is returned
3. StackOverflow
4. Your program hangs

ANSWER: 3

What happens when you run this function on a (valid) queue containing 2 elements?

```
let f (q:'a queue) : int =
  let rec loop (qn:'a qnode option) (len:int) : int =
    begin match qn with
      | None -> len
      | Some n -> loop qn (len + 1)
    end
  in loop q.head 0
```

1. The value 2 is returned
2. The value 0 is returned
3. StackOverflow
4. Your program hangs

ANSWER: 4

Infinite Loops

```
(* Accidentally go into an infinite loop... *)
let accidental_infinite_loop (q:'a queue) : int =
  let rec loop (qn:'a qnode option) (len:int) : int =
    begin match qn with
      | None -> len
      | Some n -> loop qn (len + 1)
    end
  in loop q.head 0
```

- This program will go into an infinite loop.
- Unlike a non-tail-recursive program, which uses some space on each recursive call, there is no resource being exhausted, so the program will “silently diverge” and simply never produce an answer...

More iteration examples

to_list

print

get_tail

to_list (using iteration)

```
(* Retrieve the list of values stored in the queue,  
   ordered from head to tail. *)  
let to_list (q: 'a queue) : 'a list =  
  let rec loop (no: 'a qnode option) (l:'a list) : 'a list =  
    begin match no with  
      | None -> List.rev l  
      | Some n -> loop n.next (n.v::l)  
    end  
  in loop q.head []
```

- Here, the state maintained across each iteration of the loop is the queue “index pointer” no and the (reversed) list of elements traversed.
- The “exit case” post processes the list by reversing it.

print (using iteration)

```
let print (q:'a queue) (string_of_element:'a -> string) : unit =
  let rec loop (no: 'a qnode option) : unit =
    begin match no with
      | None -> ()
      | Some n -> print_endline (string_of_element n.v);
                     loop n.next
    end
  in
    print_endline "--- queue contents ---";
    loop q.head;
    print_endline "---- end of queue -----"
```

- Here, the only state needed is the queue “index pointer”.

Singly-linked Queue Processing

- General structure (schematically) :

```
(* Process a singly-linked queue. *)
let queue_operation (q: 'a queue) : 'b =
  let rec loop (current: 'a qnode option) (s:'a state) : 'b =
    begin match no with
      | None -> ... (* iteration complete, produce result *)
      | Some n -> ... (* do something with n,
                          create new loop state *)
        loop current.next new_s
    end
  in loop q.head init
```

- What is useful to put in the state?
 - Accumulated information about the queue (e.g. length so far)
 - Link to previous node (so that it could be updated, for example)

General Guidelines

- Processing *must* maintain the queue invariants
- Update the head and tail references (if necessary)
- If changing the link structure:
 - Sometimes useful to keep reference to the previous node
(allows removal of the current node)
- Drawing pictures of the queue heap structure is helpful
- If iterating over the whole queue (e.g. to find an element)
 - It is usually *not useful* to use helpers like “`is_empty`” or “`contains`” because you will have to account for those cases during the traversal anyway!

Programming Languages and Techniques (CIS120)

Lecture 17

October 11, 2017

Typechecking (revisited)

Announcements

- *Homework 4*
 - *Extended until TONIGHT at midnight*
- Homework 5: GUI Programming
 - Available soon: Due October 24th
- Midterm 1
 - *October 13th in Class*
 - Where? Last Names:
 - A – M Leidy Labs 10 (Here)
 - N – Z Meyerson Hall B1
 - Covers lecture material through Chapter 13
 - Review materials (old exams) on course website
- Review Session:
 - TONIGHT 6:00-8:00pm, Towne 100

Typechecking Revisited

How does OCaml* typecheck your code?

*Historical aside: the algorithm we are about to see is known as the Damas-Hindley-Milner type inference algorithm. Turing Award winner Robin Milner was, among other things, the inventor of "ML" (for "meta language"), from which OCaml gets its "ml".

OCaml Typechecking Errors

```
115  let length (q: 'a queue) : int =
116    let rec loop (qn : 'a qnode option) (acc: int) : int
117    •   | None ->   
118      •     This expression has type 'a list
119      •     but an expression was expected of type int
120      | Some n -> loop n.next (1 + acc)
121
122  type ('k, 'v) map = ('k * 'v) list
123
124  (* A finite map that contains no entries. *)
125  let empty () = []
126
127  let rec mem
128    begin ma
129    | [] ->
130    | (k,v):
131      if key
132        (key = k) || (mem key rest)
133    end
134
135  ;;; run_test "mem test" (fun () ->
136    mem "b" [("a",3); ("b",4)])
137
```

• This expression has type 'a list
but an expression was expected of type int

• This expression has type 'a list
but an expression was expected of type int

• Signature mismatch:
Values do not match:
 val empty : unit -> 'a list
is not included in
 val empty : ('k, 'v) map
File "finiteMap.ml", line 13, characters 2-27: Expected
declaration
File "finiteMap.ml", line 60, characters 6-11: Actual declaration

Typechecking

How do you determine the type of an expression?

1. Recursively determine the types of *all* of the sub-expressions
 - Some expressions have “obvious” types:
3 : int “foo” : string true : bool
 - Identifiers have the types assigned where they are bound
 - let and function arguments have type annotations
 - Or, take the types from the module signature
2. Expressions that *construct* structured values have compound types built from the types of sub-expressions:

| | |
|--------------------------------|-----------------------|
| (3, “foo”) | : int * string |
| (fun (x:int) -> x + 1) | : int -> int |
| Node(Empty, (3, “foo”), Empty) | : (int * string) tree |

Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:

- Given an expression of function type $f : T_1 \rightarrow T_2$
- and an argument expression $e : T_1$ (of the input type)
- $(f e) : T_2$

```
((fun (x:int) (y:bool) -> y) 3)  : ??
```

Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:

- Given a function $f : T_1 \rightarrow T_2$
- and an argument $e : T_1$ of the input type
- $(f e) : T_2$

$((\text{fun } (x:\text{int}) (\text{y:bool}) \rightarrow \text{y}) \ 3) \ : ??$

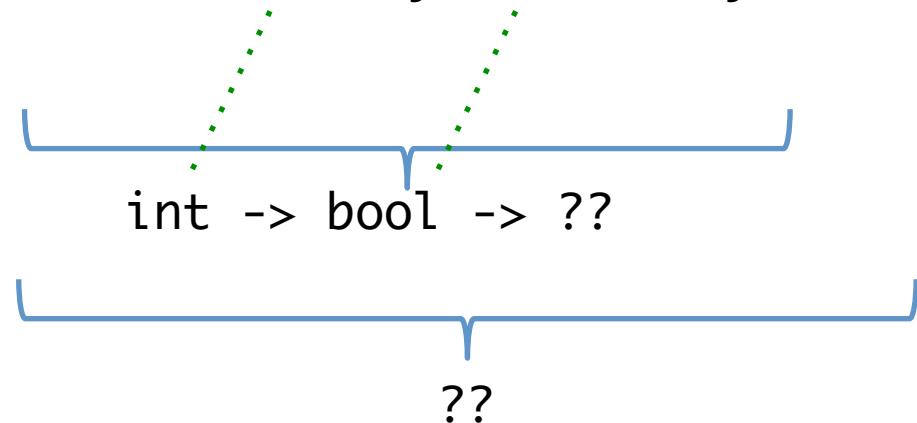


Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:

- Given a function $f : T_1 \rightarrow T_2$
- and an argument $e : T_1$ of the input type
- $(f e) : T_2$

$((\text{fun } (x:\text{int}) (\text{y:bool}) \rightarrow \text{y}) \ 3) \ : ??$



Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:

- Given a function $f : T_1 \rightarrow T_2$
- and an argument $e : T_1$ of the input type
- $(f e) : T_2$

$((\text{fun } (x:\text{int}) (y:\text{bool}) \rightarrow y) \ 3) \ : ??$

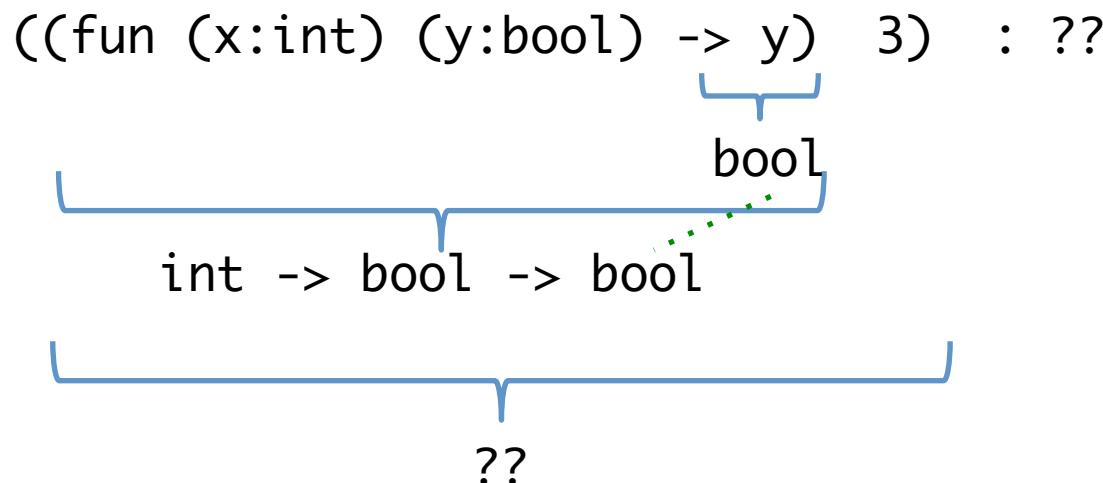
int -> bool -> ??

??

Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:

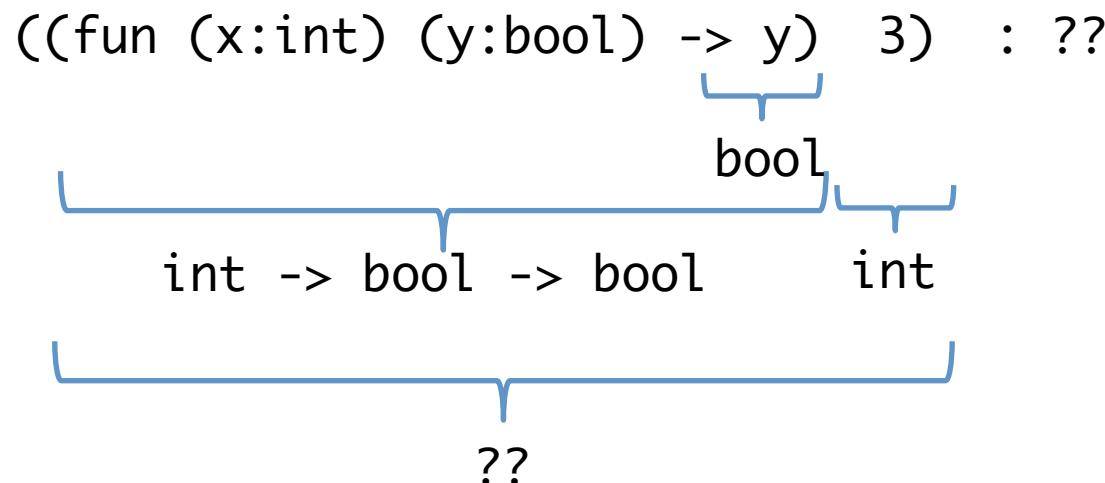
- Given a function $f : T_1 \rightarrow T_2$
- and an argument $e : T_1$ of the input type
- $(f e) : T_2$



Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:

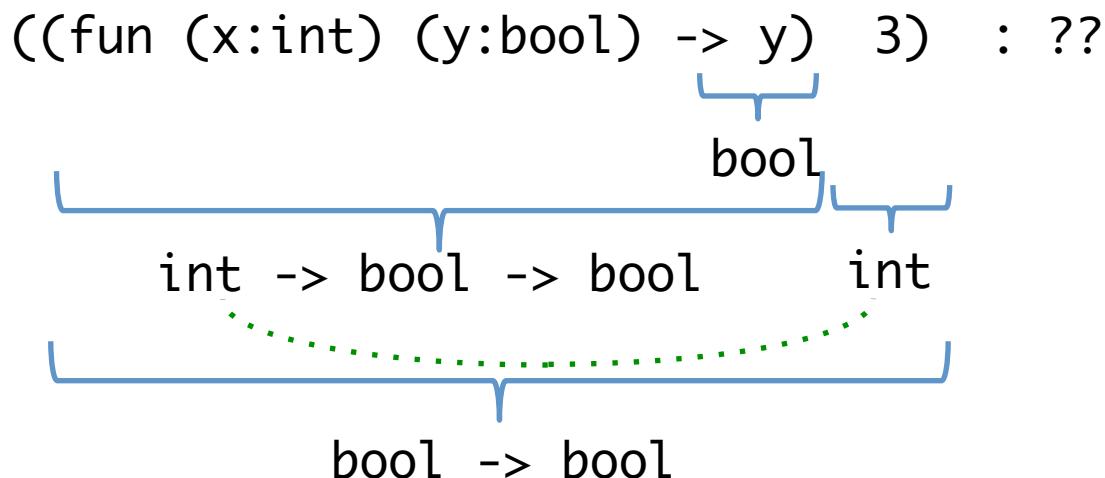
- Given a function $f : T_1 \rightarrow T_2$
- and an argument $e : T_1$ of the input type
- $(f e) : T_2$



Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:

- Given a function $f : T_1 \rightarrow T_2$
- and an argument $e : T_1$ of the input type
- $(f e) : T_2$



Here:
 $T_1 = \text{int}$
 $T_2 = \text{bool} \rightarrow \text{bool}$

Typechecking III

- For generic expressions:
 - *Unify* the types based on use:
 - Given a function $f : T_1 \rightarrow T_2$
 - and an argument $e : U_1$ of the input type
 - “*match up*” T_1 and U_1 to obtain information about type parameters in T_1 and U_1 based on their usage
 - Obtain an *instantiation*: e.g. ‘ $a = int\ list$ ’
 - *Propagate* that information to all occurrences of ‘ a ’

Example Typechecking Problem

```
empty    : ('k, 'v) map
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries  : ('k, 'v) map -> ('k * 'v) list
```

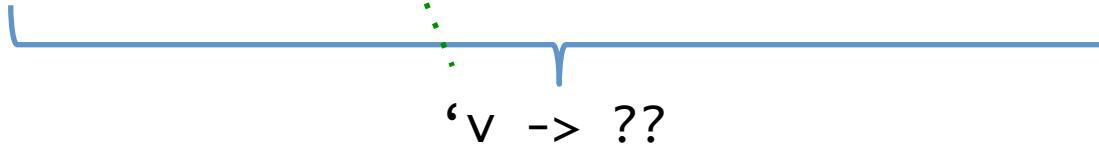
```
fun (x:'v) -> entries (add 3 x empty)
```

??

Example Typechecking Problem

```
empty    : ('k, 'v) map
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries  : ('k, 'v) map -> ('k * 'v) list
```

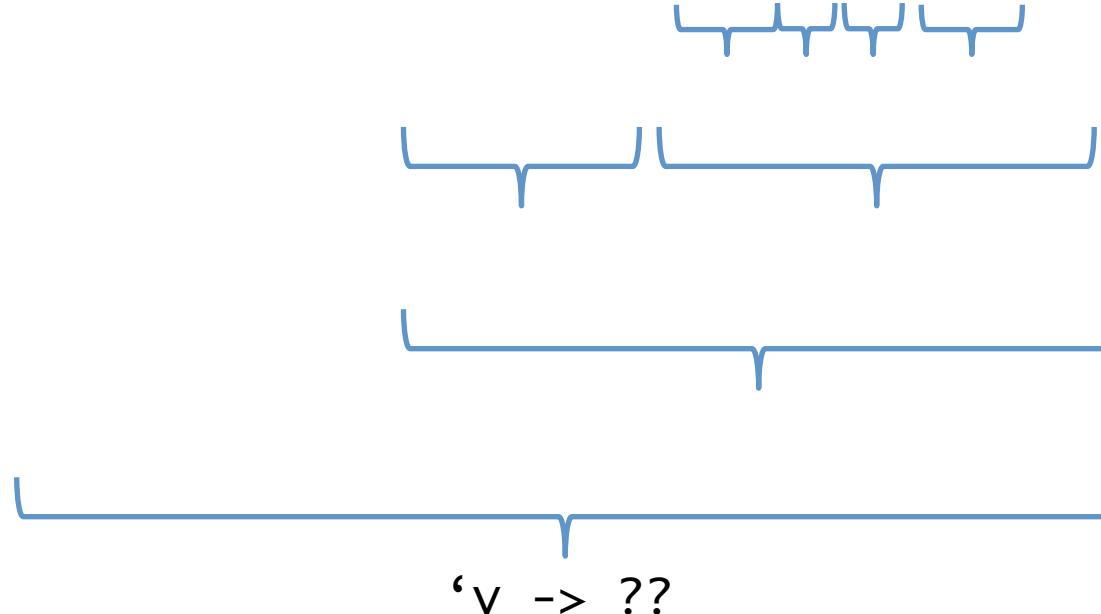
```
fun (x:'v) -> entries (add 3 x empty)
```



Example Typechecking Problem

```
empty    : ('k, 'v) map
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries  : ('k, 'v) map -> ('k * 'v) list
```

```
fun (x:'v) -> entries (add 3 x empty)
```



Example Typechecking Problem

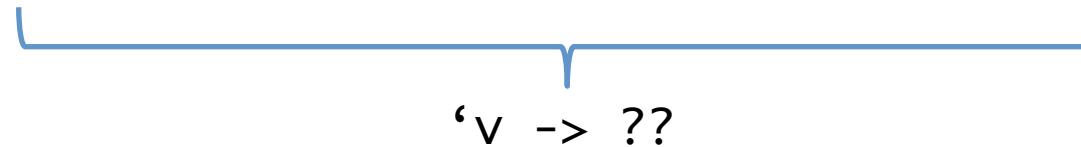
```
empty    : ('k, 'v) map
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries  : ('k, 'v) map -> ('k * 'v) list
```

```
fun (x:'v) -> entries (add 3 x empty)
```

int



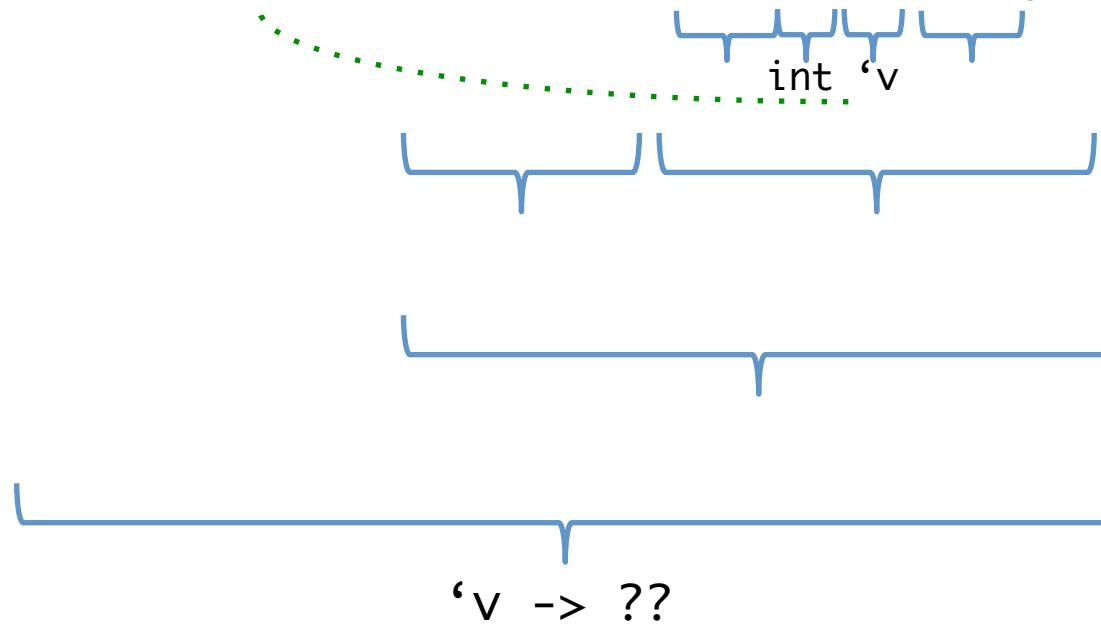
'v -> ??



Example Typechecking Problem

```
empty    : ('k, 'v) map
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries  : ('k, 'v) map -> ('k * 'v) list
```

```
fun (x:'v) -> entries (add 3 x empty)
```



Example Typechecking Problem

```
empty    : ('k, 'v) map
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries  : ('k, 'v) map -> ('k * 'v) list
```

```
fun (x:'v) -> entries (add 3 x empty)
```

The diagram illustrates the type annotations for the expression. A dotted green line connects the type annotations in the declarations to their corresponding positions in the expression. Brackets below the expression indicate the type of the argument 'x' and the return type of the function.

Annotations:

- int : under the number 3.
- $'v$: under the variable x .
- $('k, 'v) \text{ map}$: under the term empty .
- $'v -> ??$: under the term entries .

Example Typechecking Problem

```
empty    : ('k, 'v) map
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries  : ('k, 'v) map -> ('k * 'v) list
```

```
fun (x:'v) -> entries (add 3 x empty)
```

int 'v ('k, 'v) map

'v -> ??

Example Typechecking Problem

```
empty    : ('k, 'v) map
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries  : ('k, 'v) map -> ('k * 'v) list
```

```
fun (x:'v) -> entries (add 3 x empty)
```

int 'v ('k, 'v) map

??

'v -> ??

Example Typechecking Problem

```
empty    : ('k, 'v) map
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries  : ('k, 'v) map -> ('k * 'v) list
```

fun (x:'v) -> entries (add 3 x empty)

Application:

$T_1 = 'k$

$T_2 = 'v \rightarrow ('k, 'v) map \rightarrow ('k, 'v) map$

Instantiate: $'k = \text{int}$

$'v \rightarrow ??$

Example Typechecking Problem

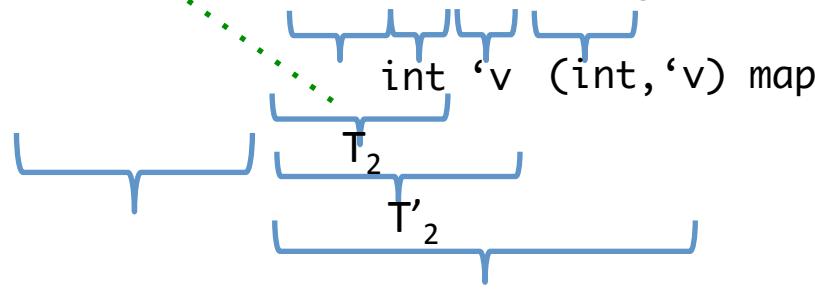
```
empty    : ('k, 'v) map
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries  : ('k, 'v) map -> ('k * 'v) list
```

fun (x:'v) -> entries (add 3 x empty)

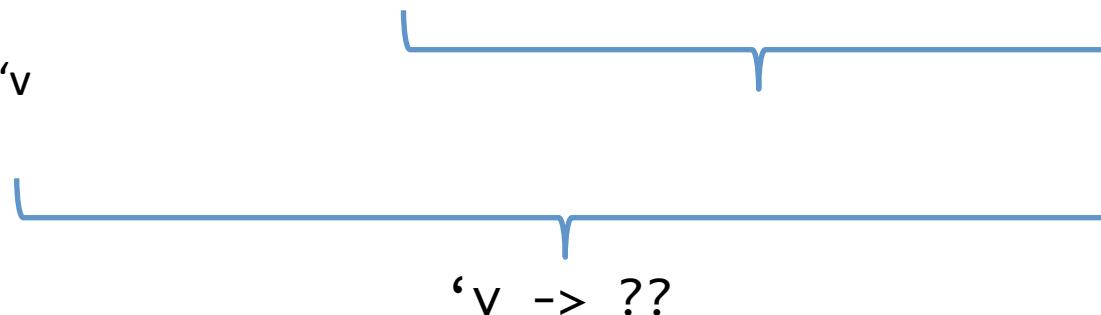
Another Application:

$T'_1 = 'v$

$T'_2 = (\text{int}, 'v) \text{ map} \rightarrow (\text{int}, 'v) \text{ map}$



Instantiate: $'v = 'v$



Example Typechecking Problem

```
empty    : ('k, 'v) map
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries  : ('k, 'v) map -> ('k * 'v) list
```

fun (x:'v) -> entries (add 3 x empty)

A third Application:

$T''_1 = (\text{int}, 'v) \text{ map}$

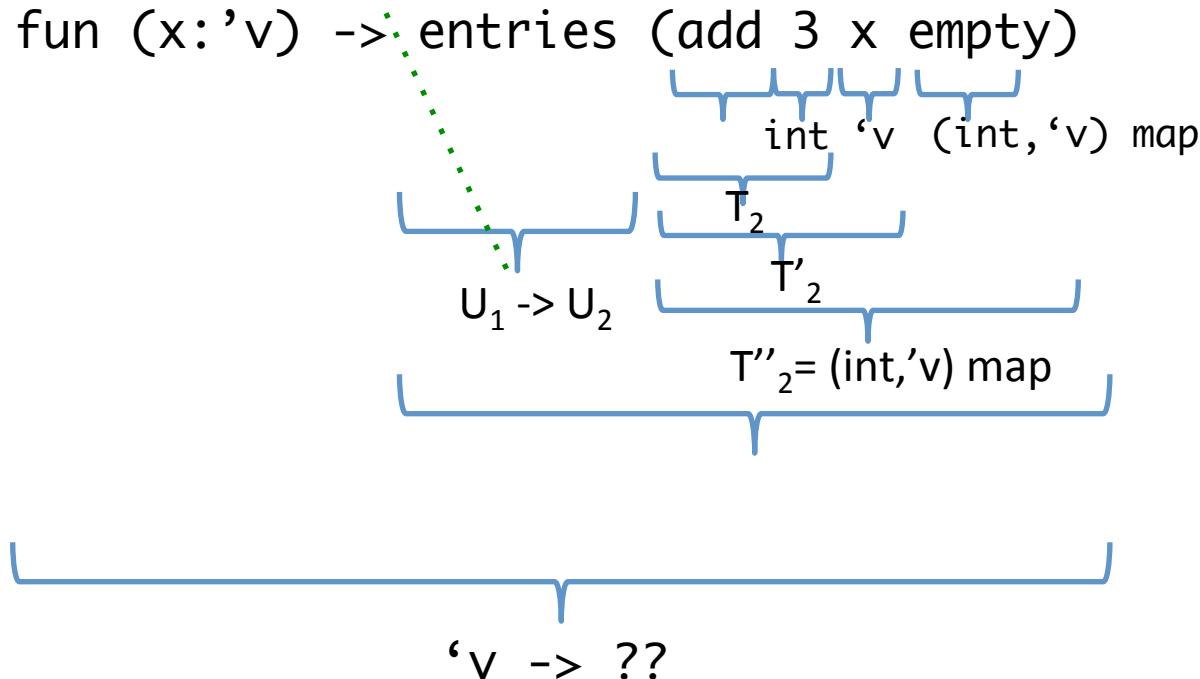
$T''_2 = (\text{int}, 'v) \text{ map}$

Argument and argument
type already agree

'v -> ??

Example Typechecking Problem

```
empty    : ('k, 'v) map
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries  : ('k, 'v) map -> ('k * 'v) list
```



Example Typechecking Problem

```
empty    : ('k, 'v) map
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries  : ('k, 'v) map -> ('k * 'v) list
```

Another Application:

$U_1 = ('k, 'v) \text{ map}$

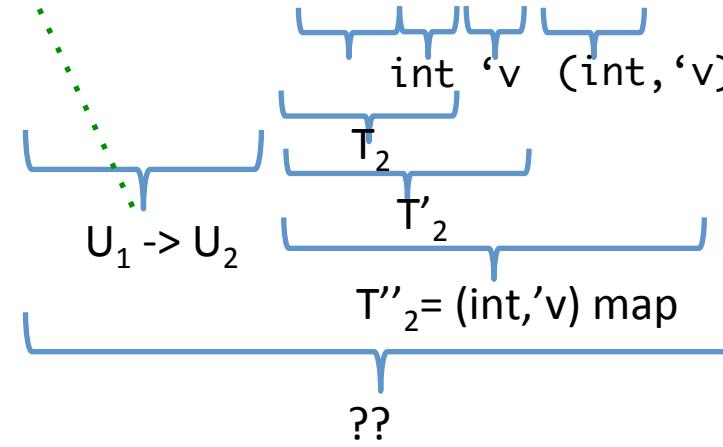
$U_2 = ('k * 'v) \text{ list}$

Unify U_1 with T''_2

$('k, 'v) \text{ map} \sim (\text{int}, 'v) \text{ map}$

Instantiate ' $k = \text{int}$ '

fun (x: 'v) -> entries (add 3 x empty)



$'v \rightarrow ??$

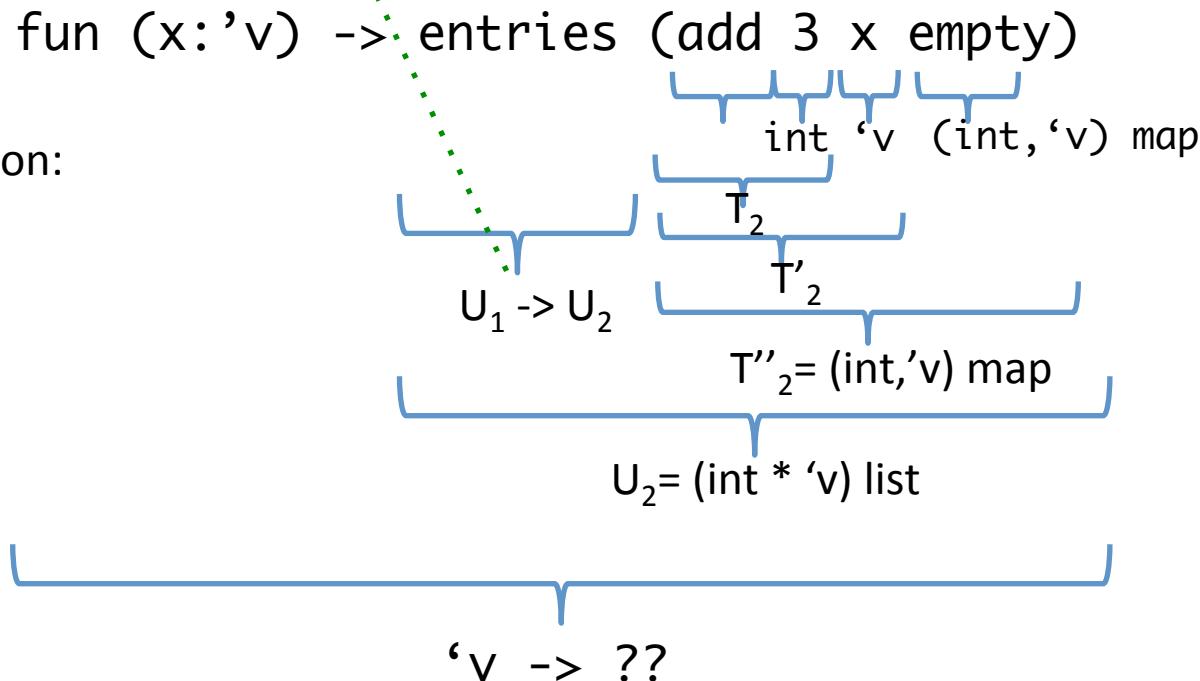
Example Typechecking Problem

```
empty    : ('k, 'v) map
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries  : ('k, 'v) map -> ('k * 'v) list
```

Another Application:

$U_1 = (\text{int}, 'v) \text{ map}$

$U_2 = (\text{int} * 'v) \text{ list}$



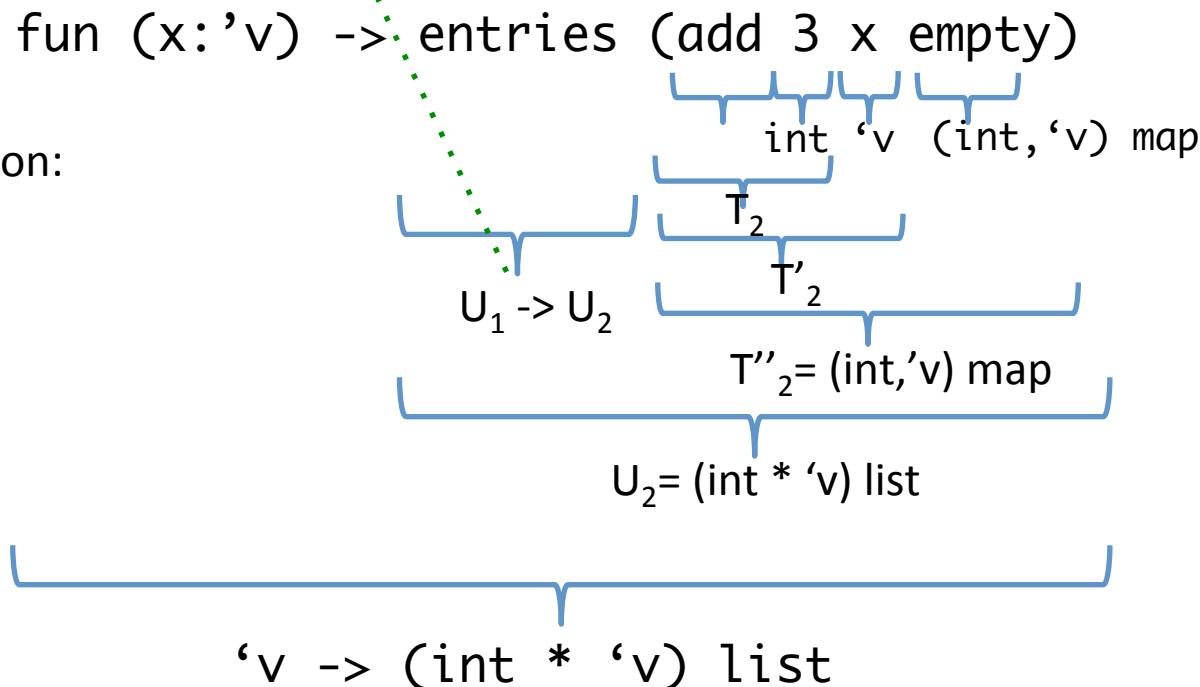
Example Typechecking Problem

```
empty    : ('k, 'v) map
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries  : ('k, 'v) map -> ('k * 'v) list
```

Another Application:

$U_1 = (\text{int}, 'v) \text{ map}$

$U_2 = (\text{int} * 'v) \text{ list}$



Ill-typed Expressions?

- An expression is ill-typed if, during this type checking process, inconsistent constraints are encountered:

```
empty    : ('k, 'v) map
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries  : ('k, 'v) map -> ('k * 'v) list
```

add 3 true (add “foo” false empty)

Error: found int but expected string

Hidden State

Encapsulating State

An “incr” function

- Functions with internal state

```
type counter_state = { mutable count:int }

let ctr = { count = 0 }

(* each call to incr will produce the next integer *)
let incr () : int =
  ctr.count <- ctr.count + 1;
  ctr.count
```

- Drawbacks:
 - *No abstraction*: There is only one counter in the world. If we want another, we need another `counter_state` value and another `incr` function.
 - *No encapsulation*: Any other code can modify `count`, too.

Using Hidden State

- Make a function that creates a counter state and an incr function each time a counter is needed.

```
(* More useful: a counter generator: *)
let mk_incr () : unit -> int =
  (* this ctr is private to the returned function *)
  let ctr = { count = 0 } in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count

(* make one counter *)
let incr1 : unit -> int = mk_incr ()

(* make another counter *)
let incr2 : unit -> int = mk_incr ()
```

What number is printed by this program?

```
let mk_incr () : unit -> int =
  let ctr = { count = 0 } in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count

let incr1 = mk_incr () (* make one counter *)
let incr2 = mk_incr () (* and another *)

let _ = incr1 () in print_int (incr2 ())
```

1. 1
2. 2
3. 3
4. other

Running mk_incr

Workspace

```
let mk_incr () : unit -> int =
  let ctr = {count = 0} in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count

let incr1 : unit -> int =
  mk_incr ()
```

Stack

Heap

Running mk_incr

Workspace

```
let mk_incr : unit -> unit ->
int = fun () ->
  let ctr = {count = 0} in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count

let incr1 : unit -> int =
mk_incr ()
```

Stack

Heap

Running mk_incr

Workspace

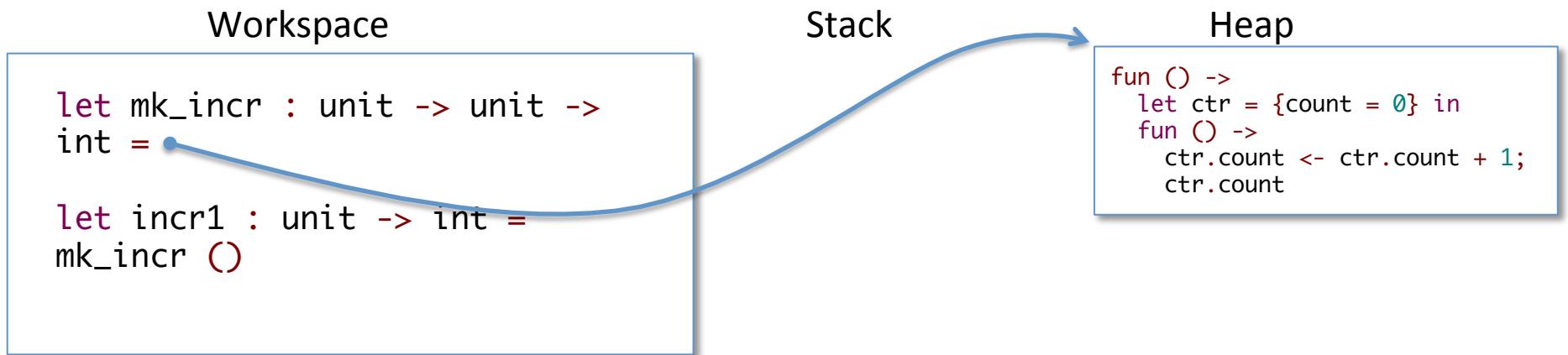
```
let mk_incr : unit -> unit ->
int = fun () ->
  let ctr = {count = 0} in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count

let incr1 : unit -> int =
mk_incr ()
```

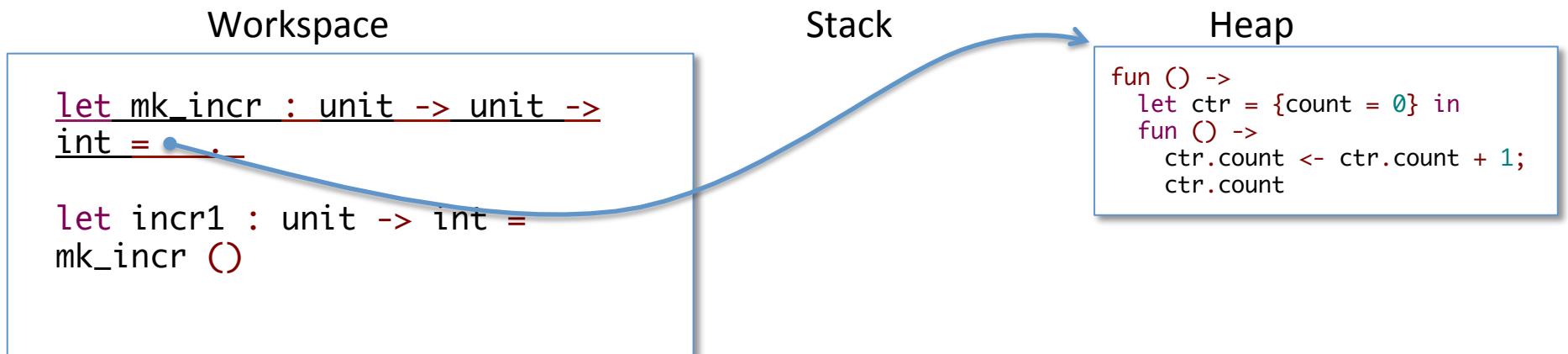
Stack

Heap

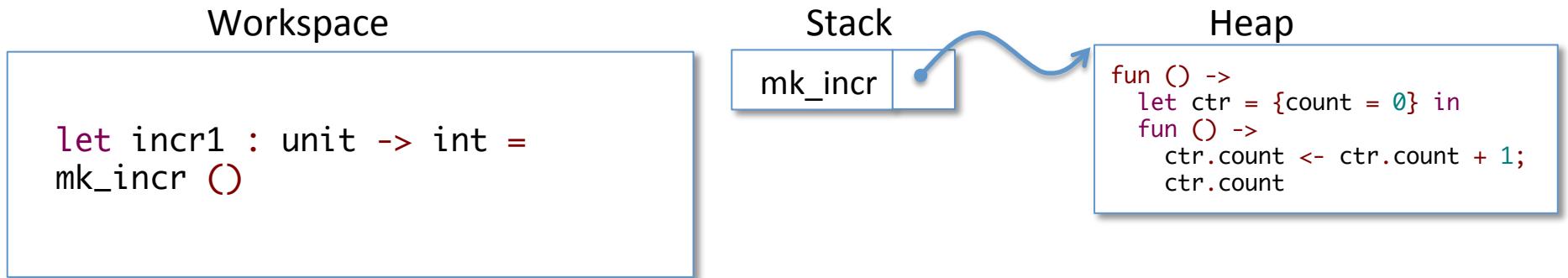
Running mk_incr



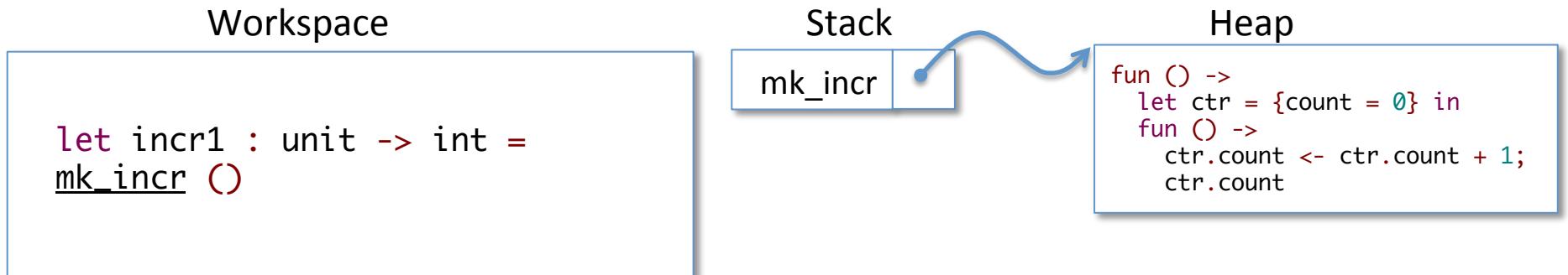
Running mk_incr



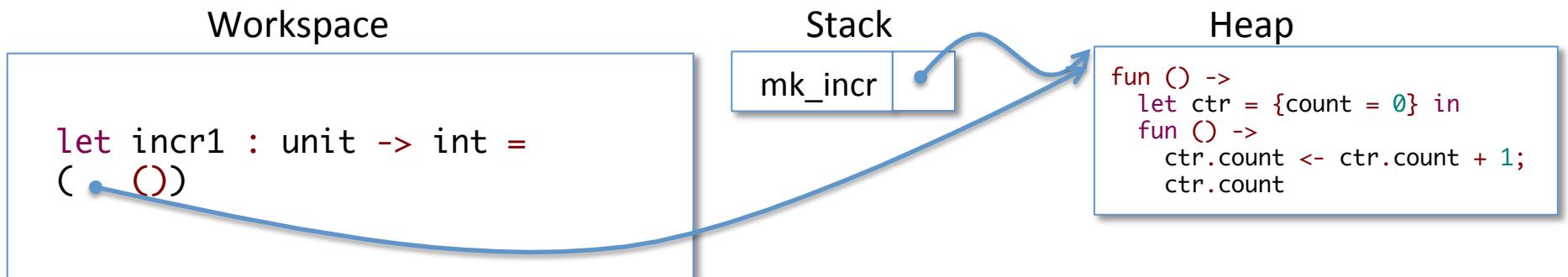
Running mk_incr



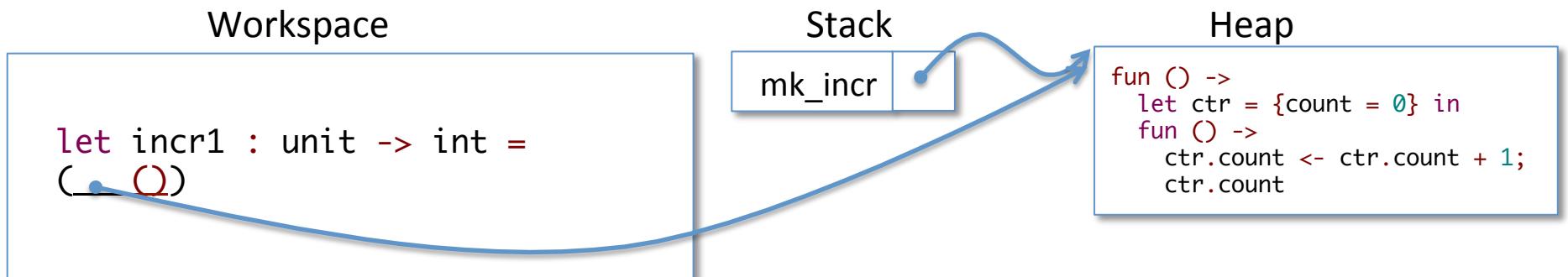
Running mk_incr



Running mk_incr



Running mk_incr



Running mk_incr

Workspace

```
let ctr = {count = 0} in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count
```

Stack

mk_incr

```
let incr1 : unit -> int =
  ___
```

Heap

```
fun () ->
  let ctr = {count = 0} in
    fun () ->
      ctr.count <- ctr.count + 1;
      ctr.count
```

Running mk_incr

Workspace

```
let ctr = {count = 0} in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count
```

Stack

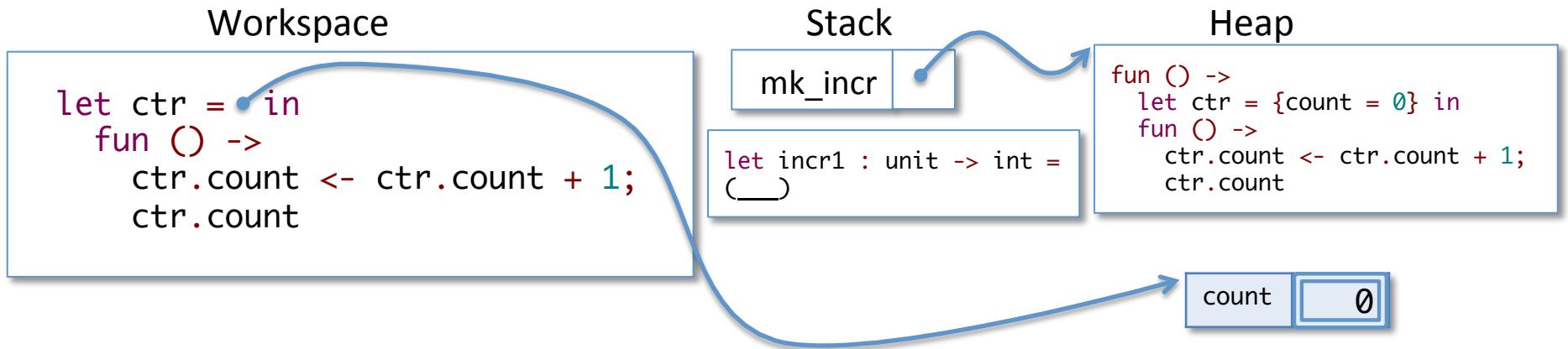
mk_incr

```
let incr1 : unit -> int =
  ___
```

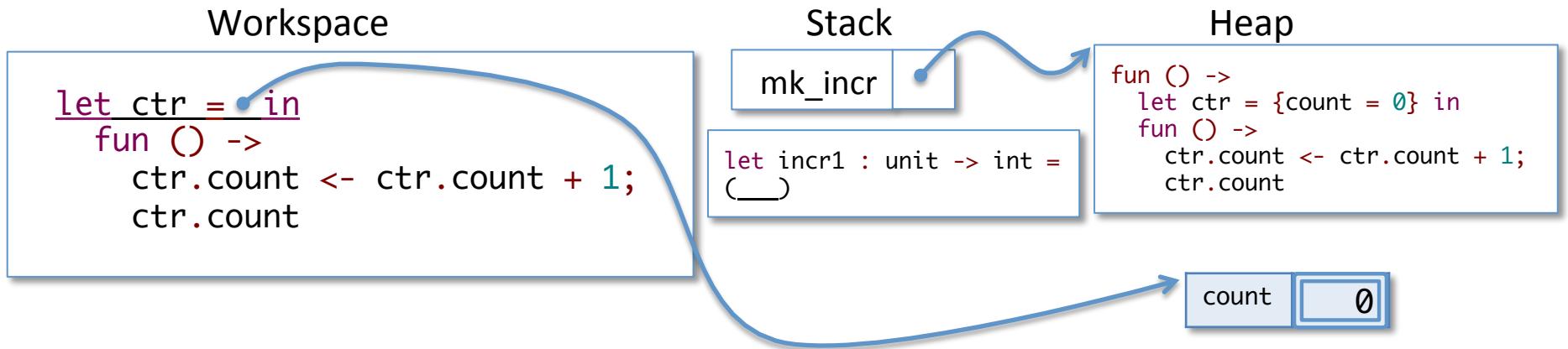
Heap

```
fun () ->
  let ctr = {count = 0} in
    fun () ->
      ctr.count <- ctr.count + 1;
      ctr.count
```

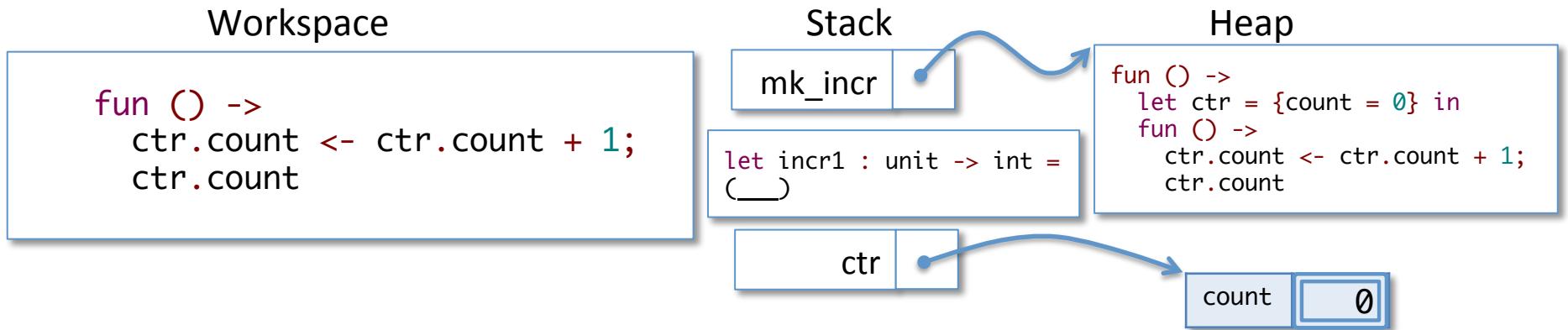
Running mk_incr



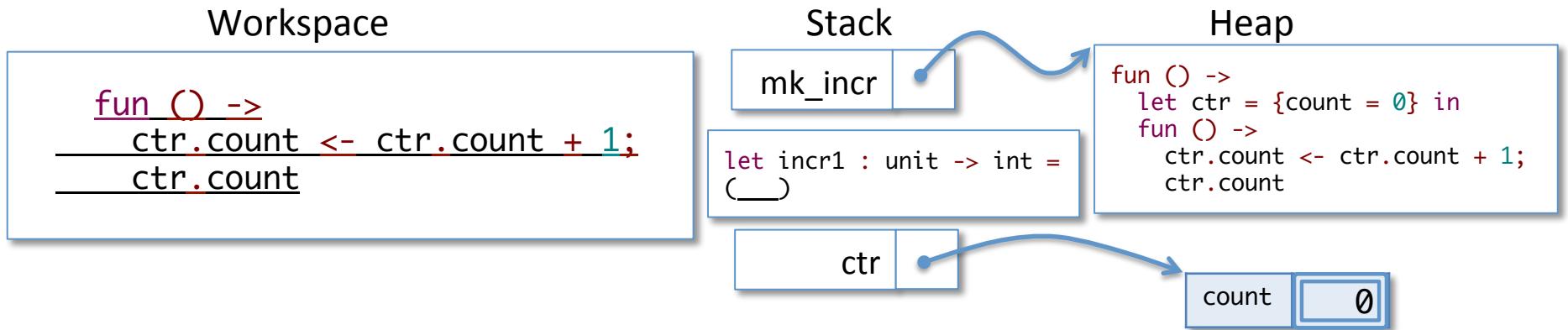
Running mk_incr



Running mk_incr

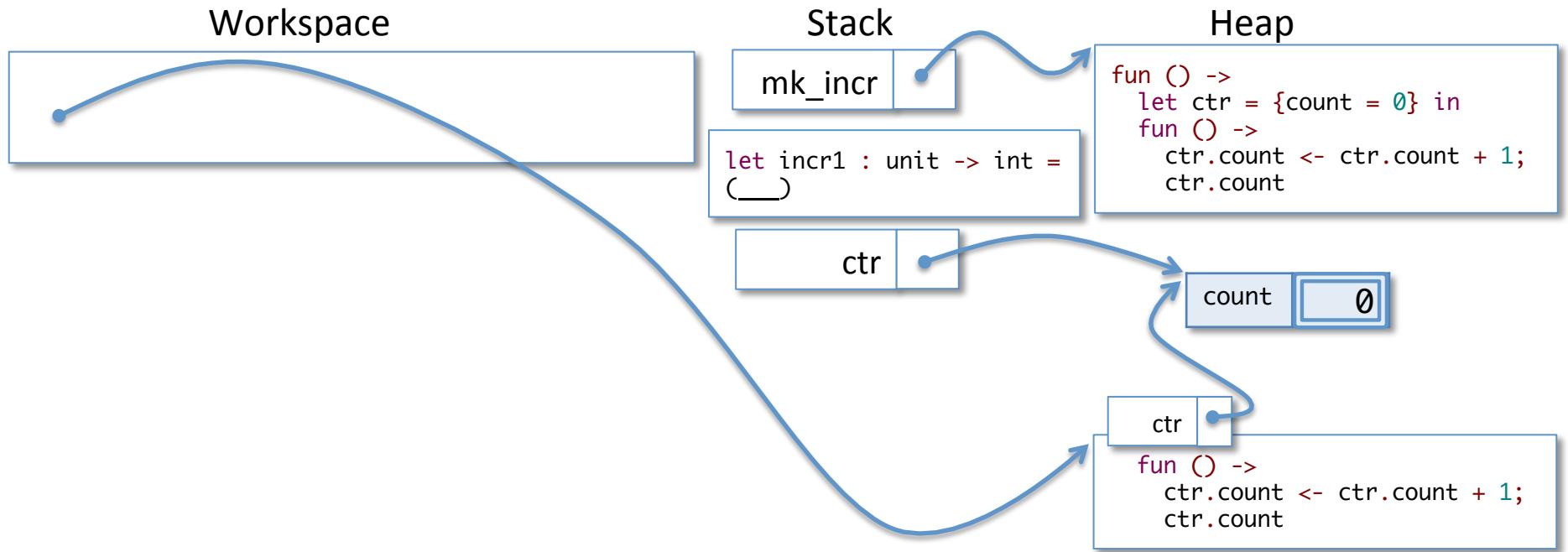


Running mk_incr



Key step!

Local Functions



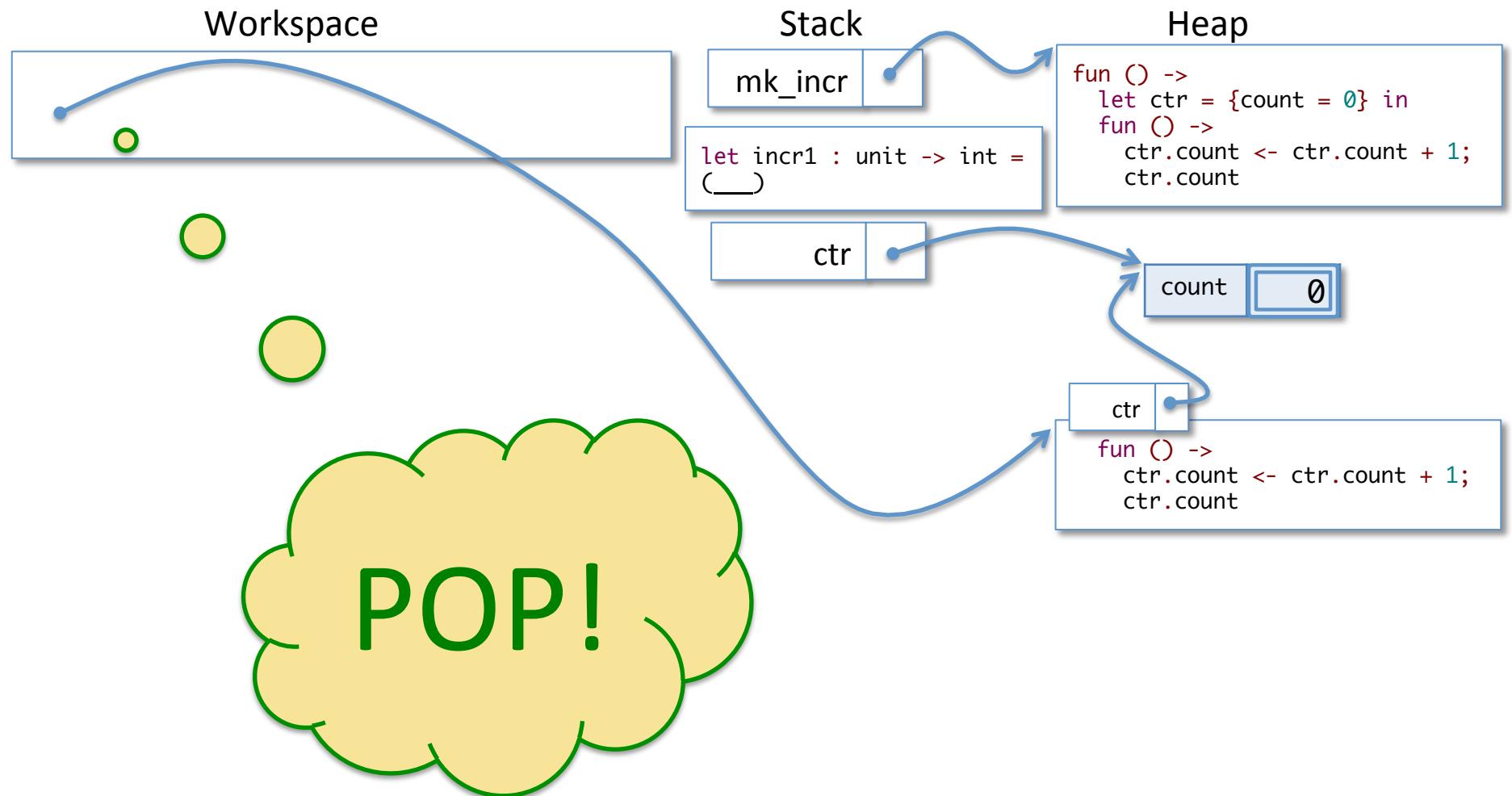
Note: We need one refinement of the ASM model that we've explained so far. Why?

The function body mentions “ctr”, which is on the stack (but about to be popped off)...

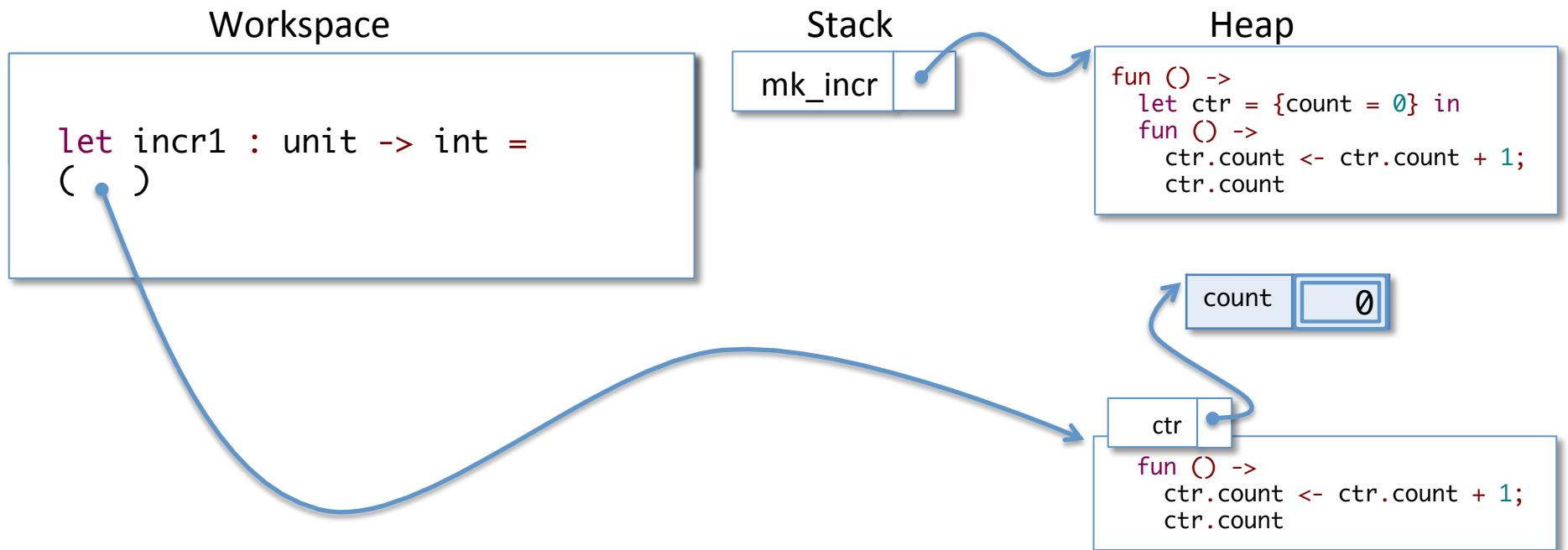
...so we save a copy of the needed stack bindings with the function itself.

This package of “function body plus needed bindings” is called a *closure*...

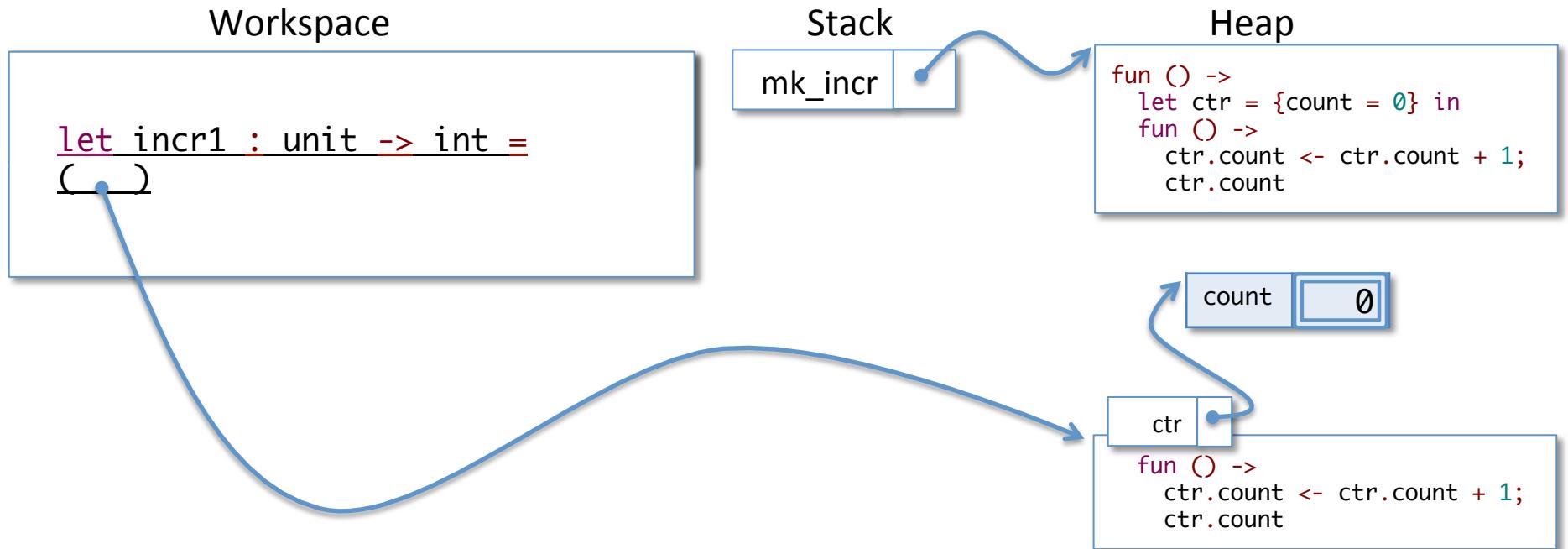
Local Functions



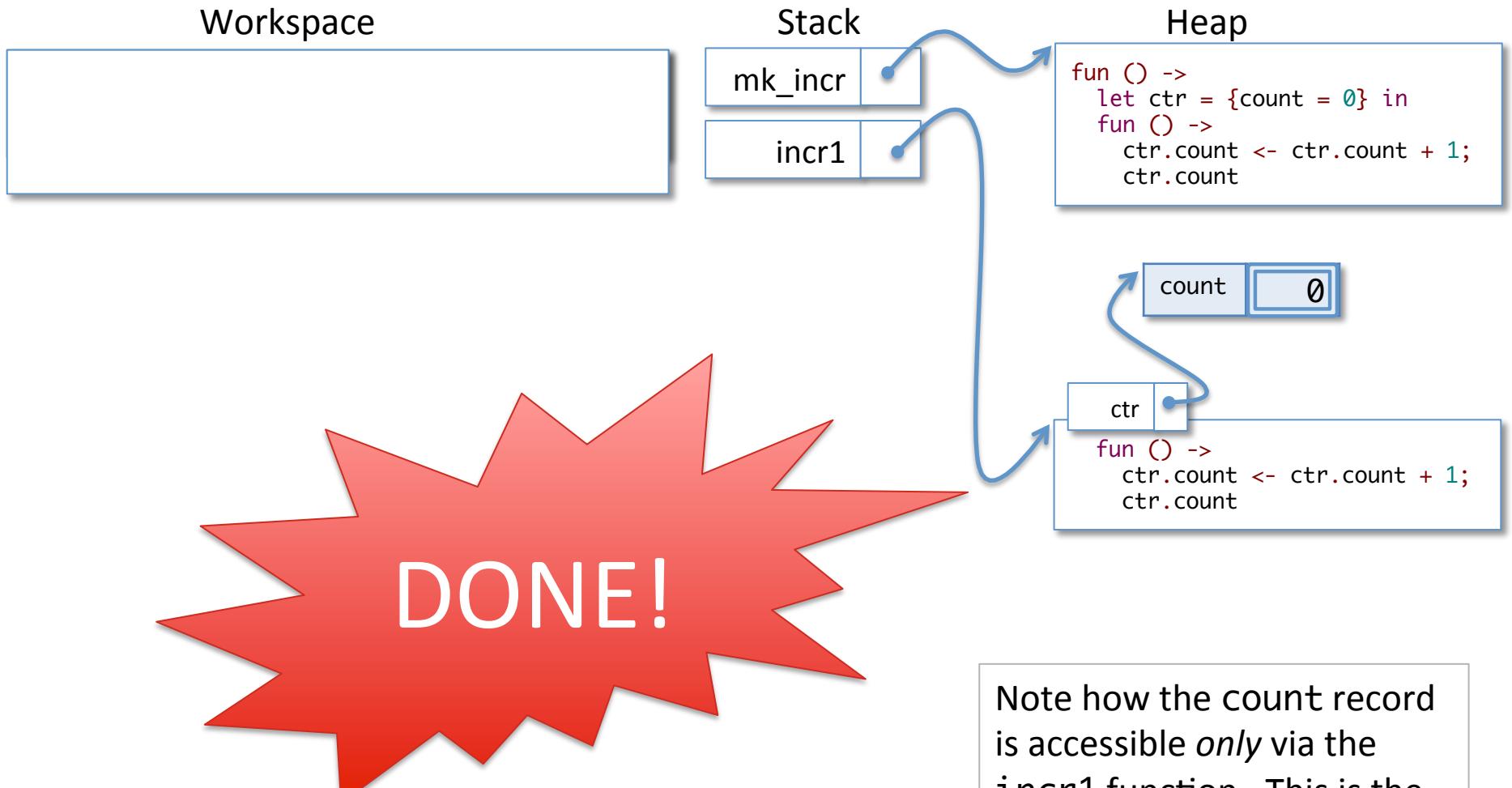
Local Functions



Local Functions

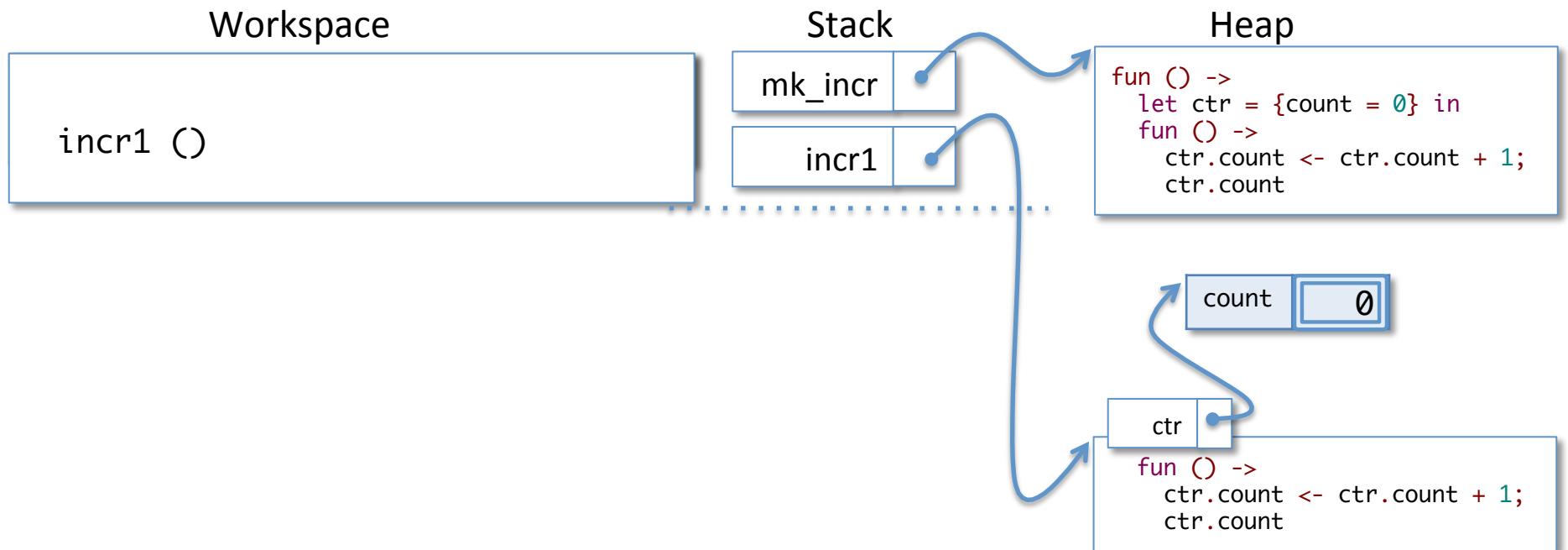


Local Functions

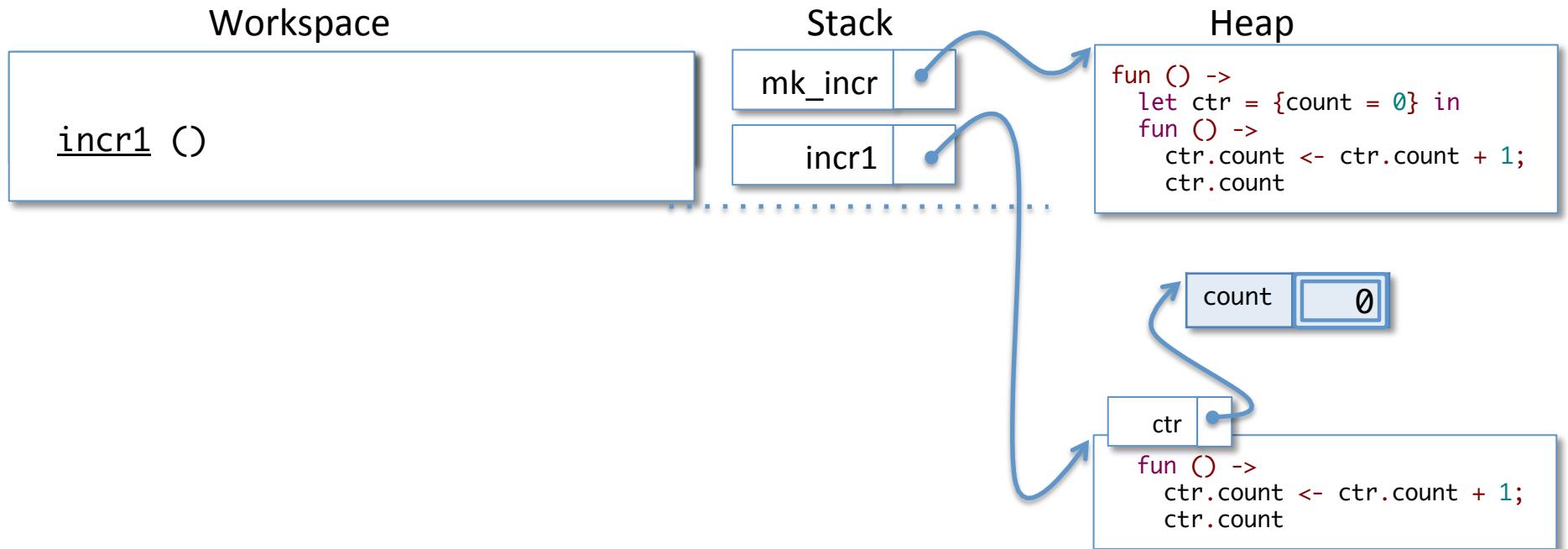


Note how the count record is accessible *only* via the `incr1` function. This is the sense in which the state is “local” to `incr1`.

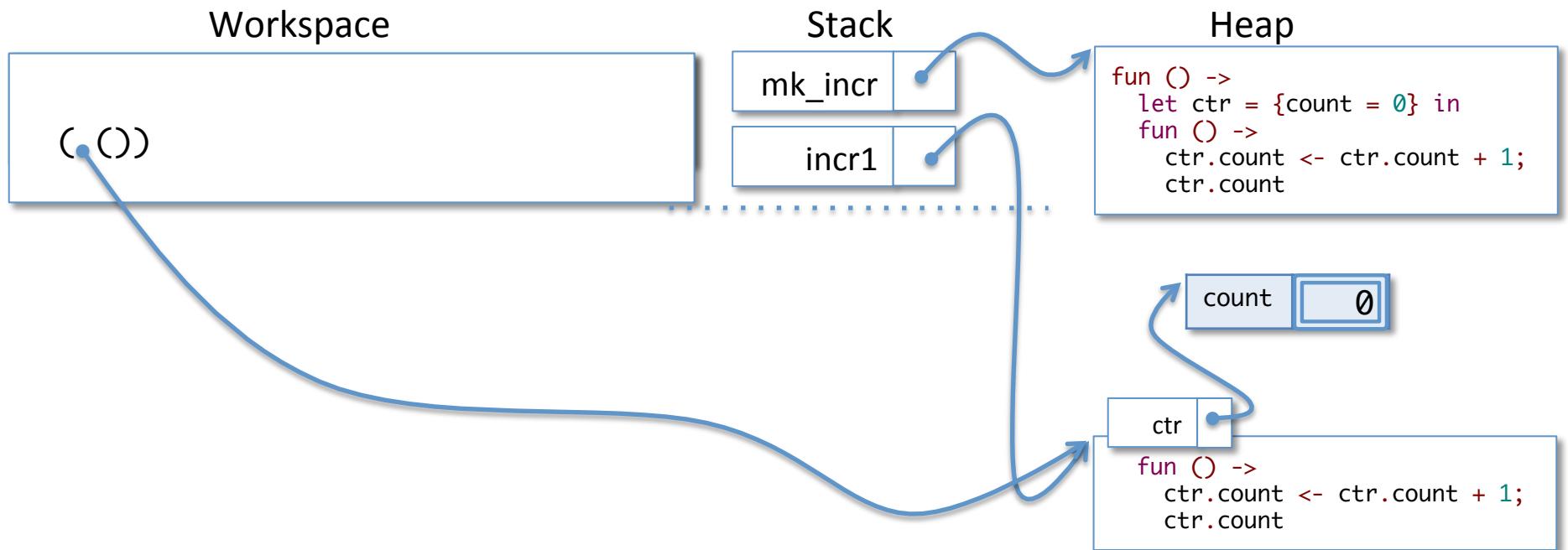
Now let's run “incr1 ()”



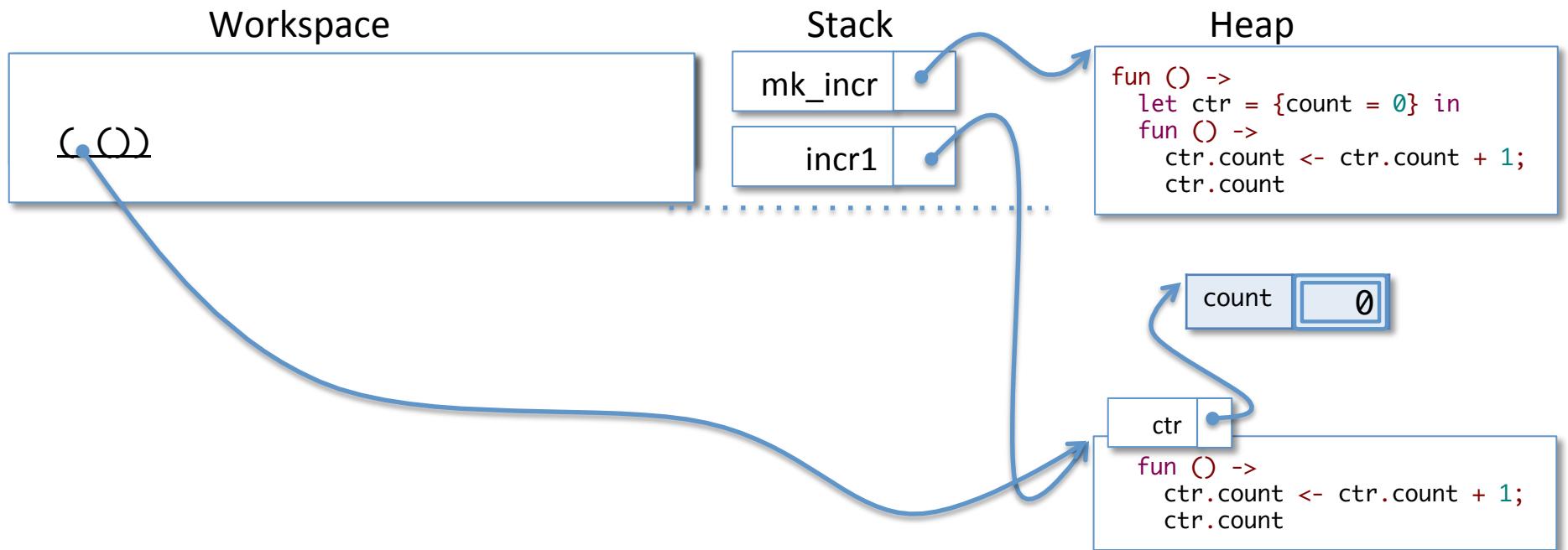
Now let's run “incr1 ()”



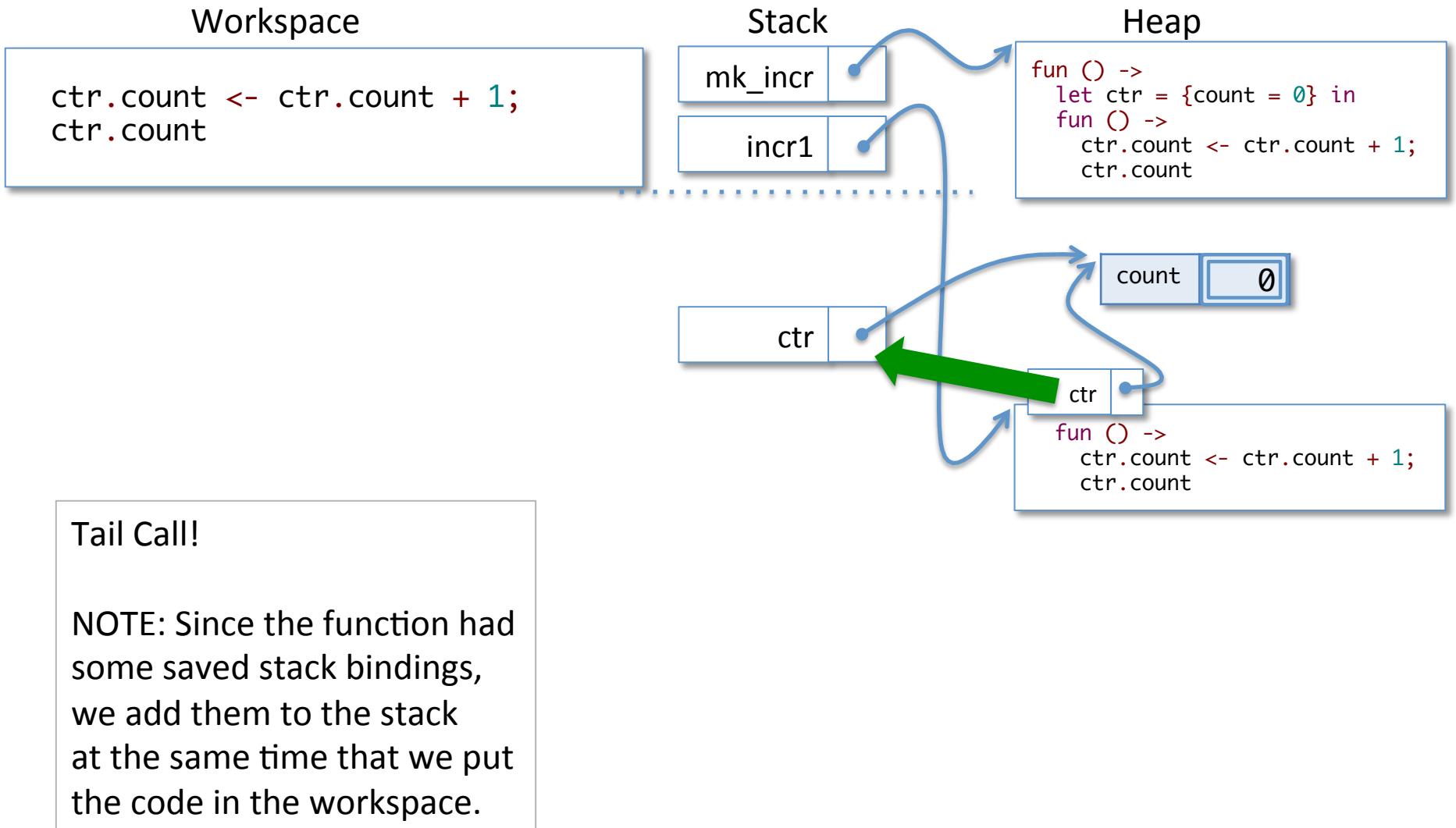
Now let's run “incr1 ()”



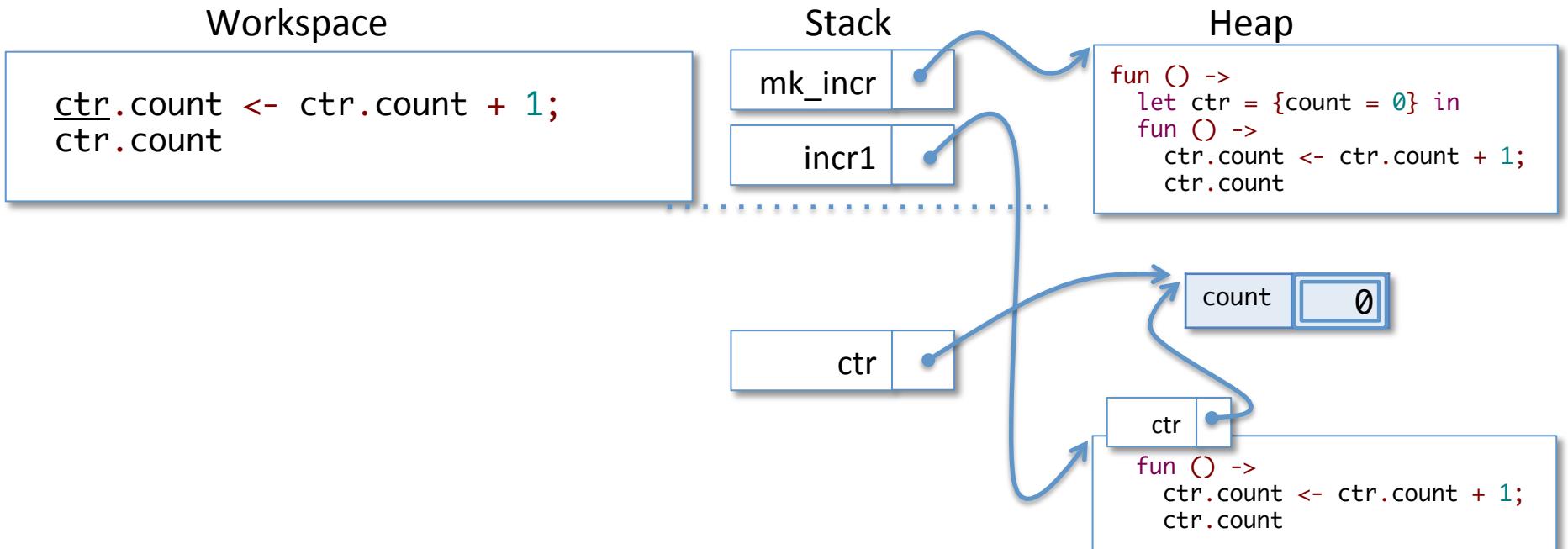
Now let's run “incr1 ()”



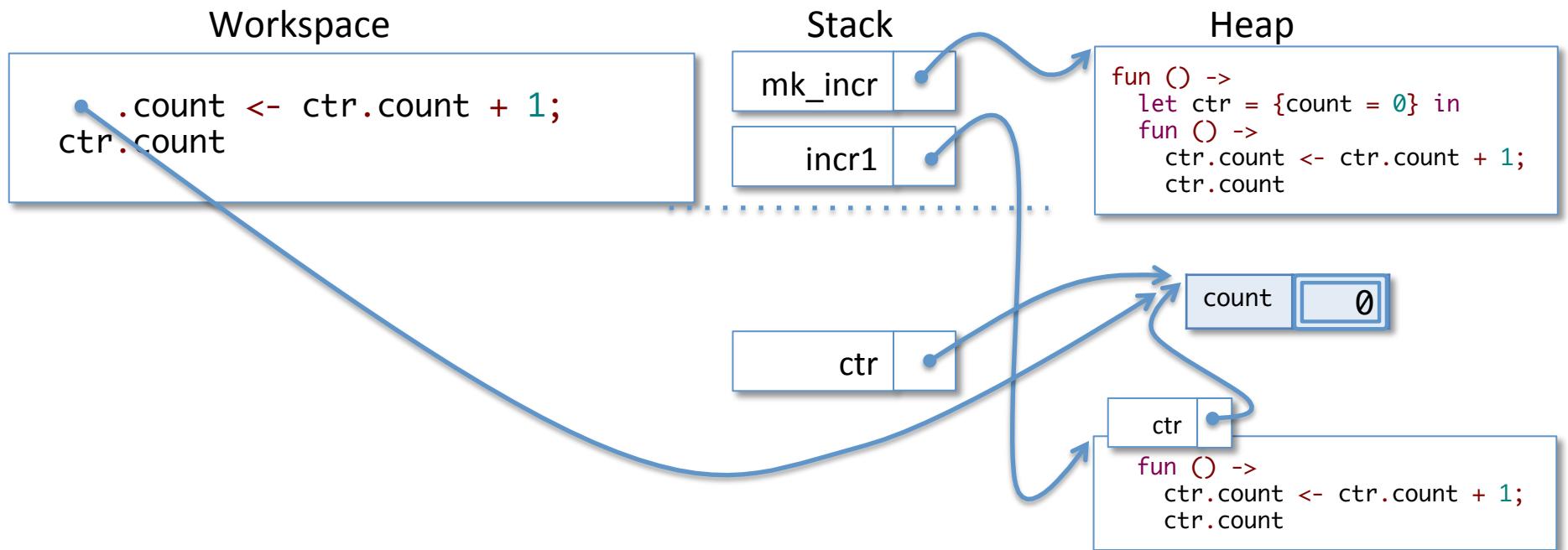
Now let's run “incr1 ()”



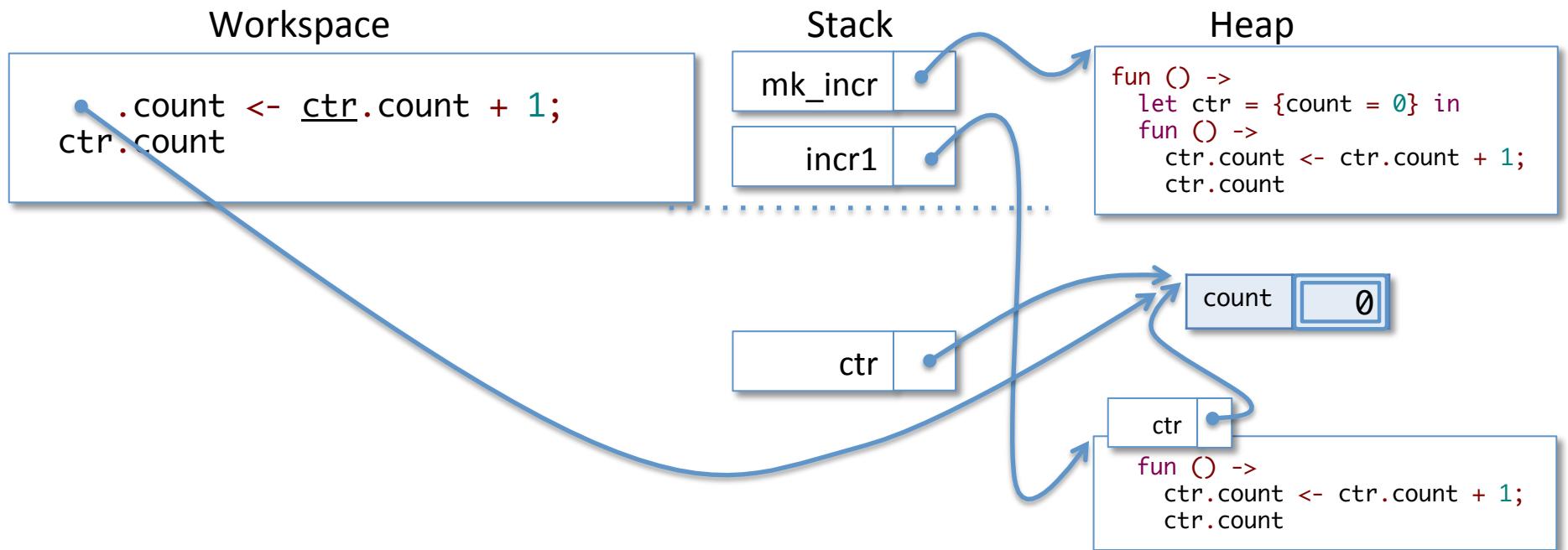
Now let's run “incr1 ()”



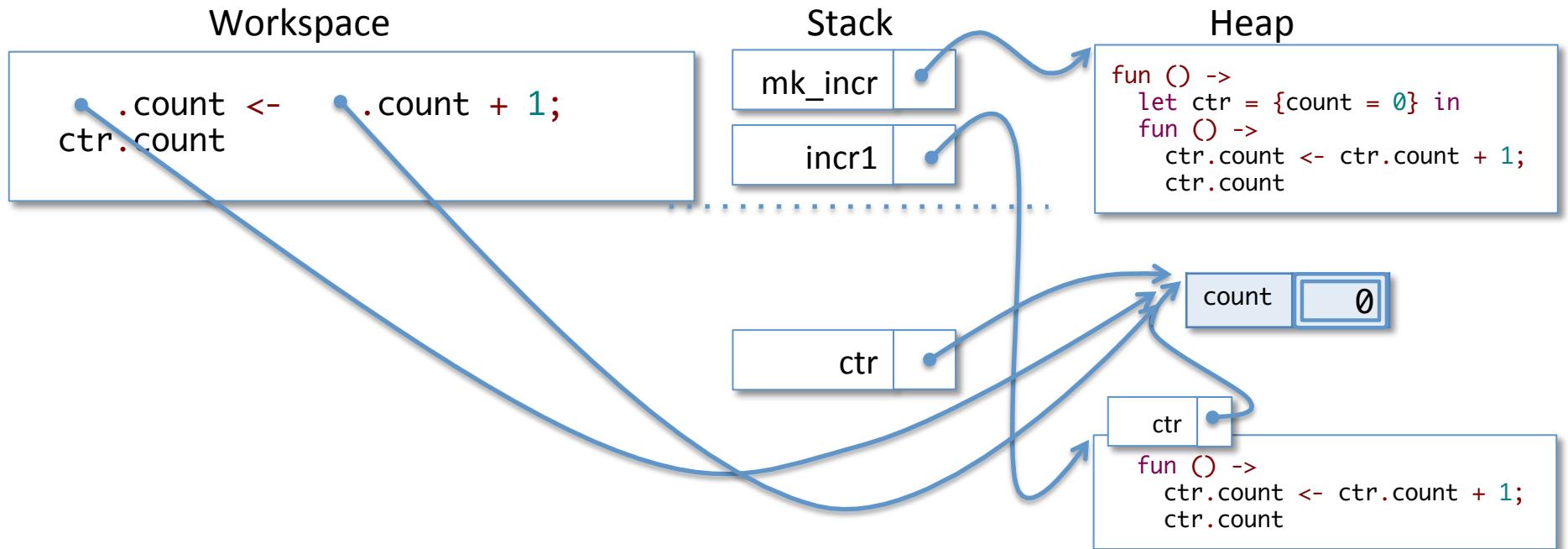
Now let's run “incr1 ()”



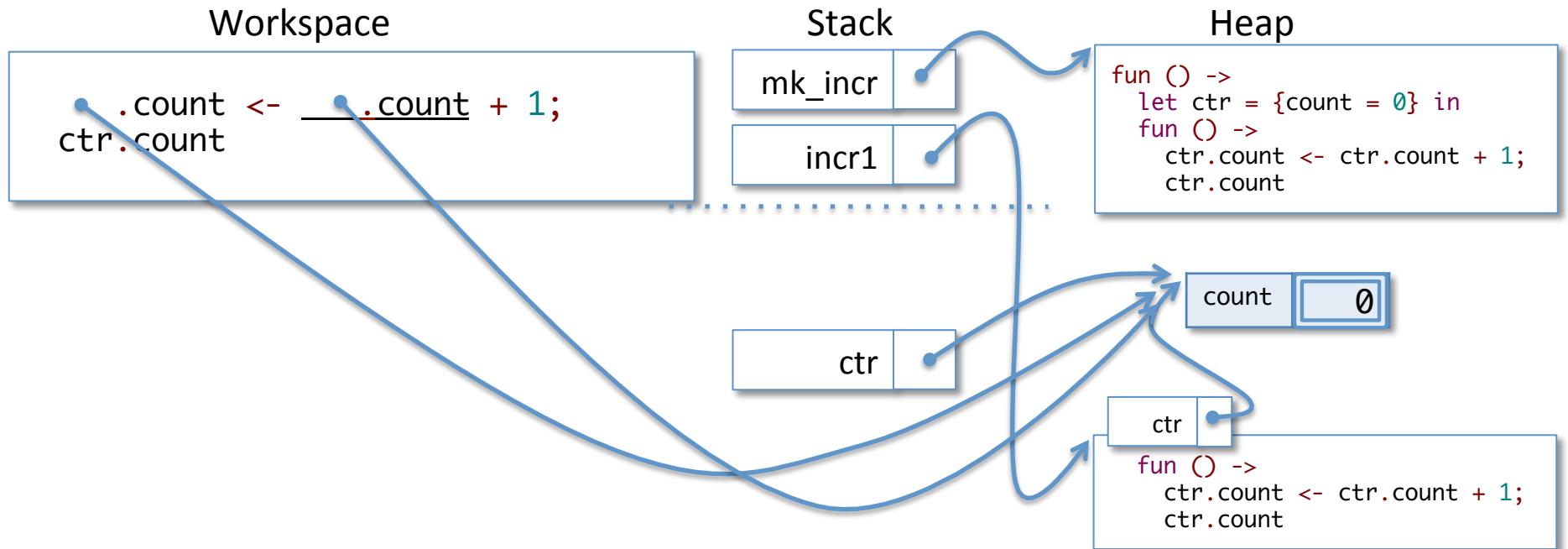
Now let's run “incr1 ()”



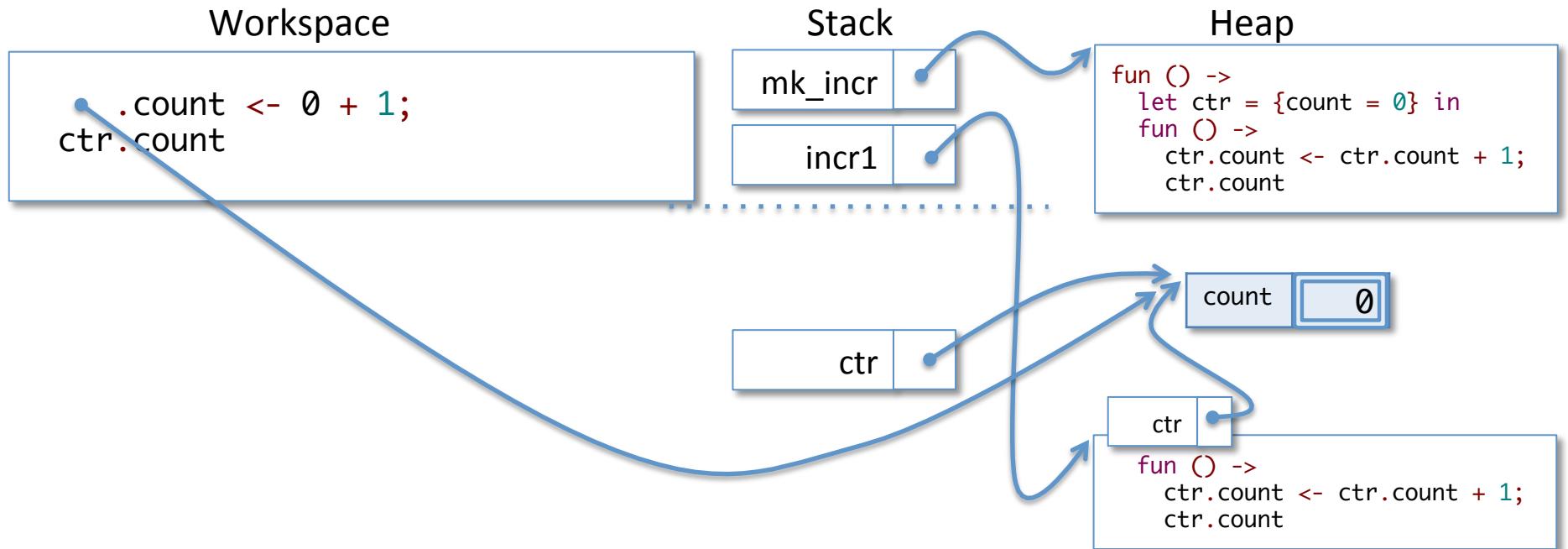
Now let's run “incr1 ()”



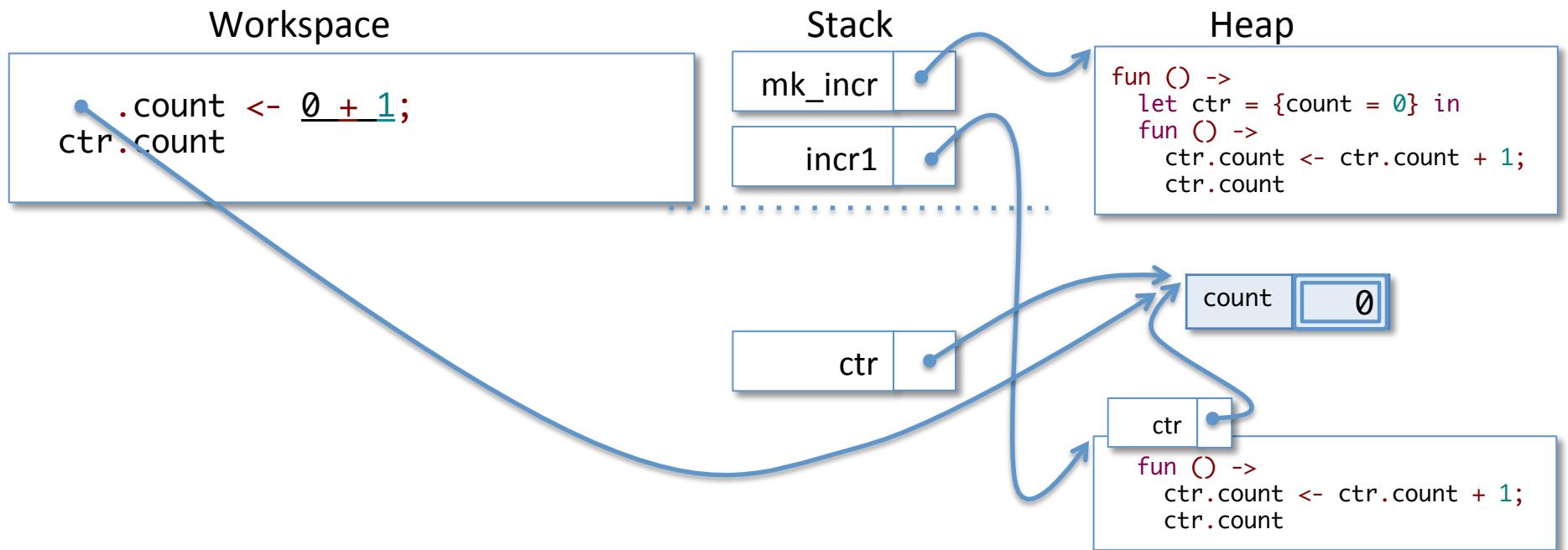
Now let's run “incr1 ()”



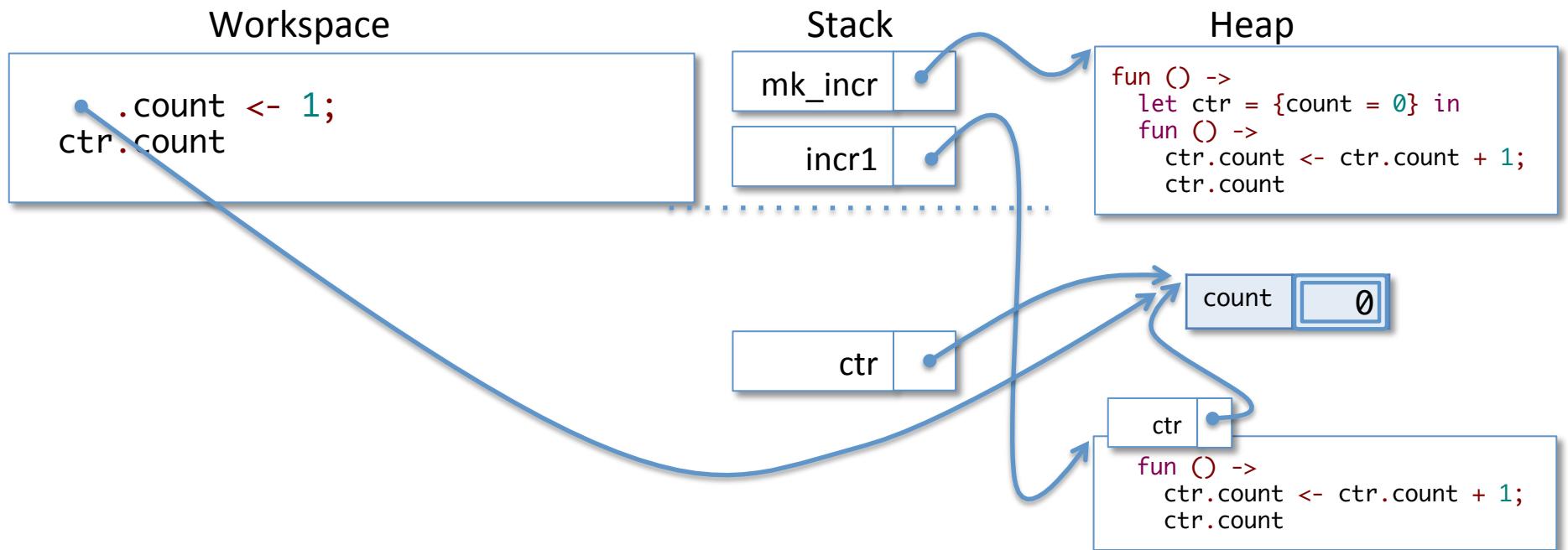
Now let's run “incr1 ()”



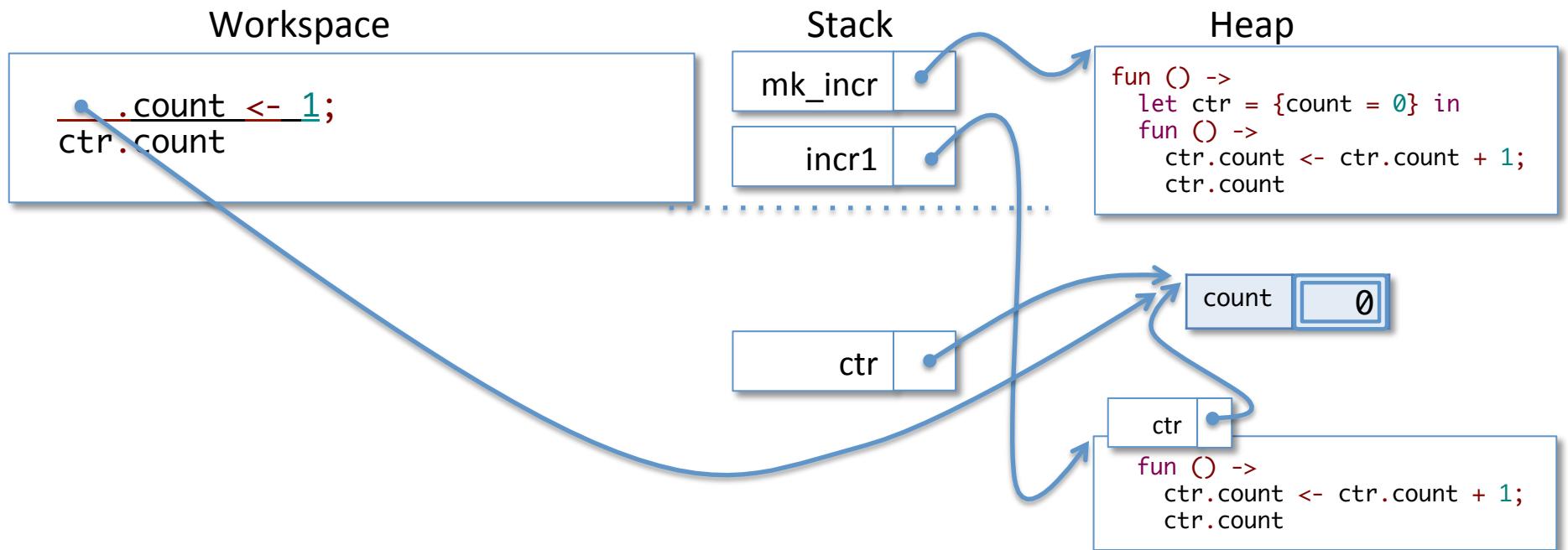
Now let's run “incr1 ()”



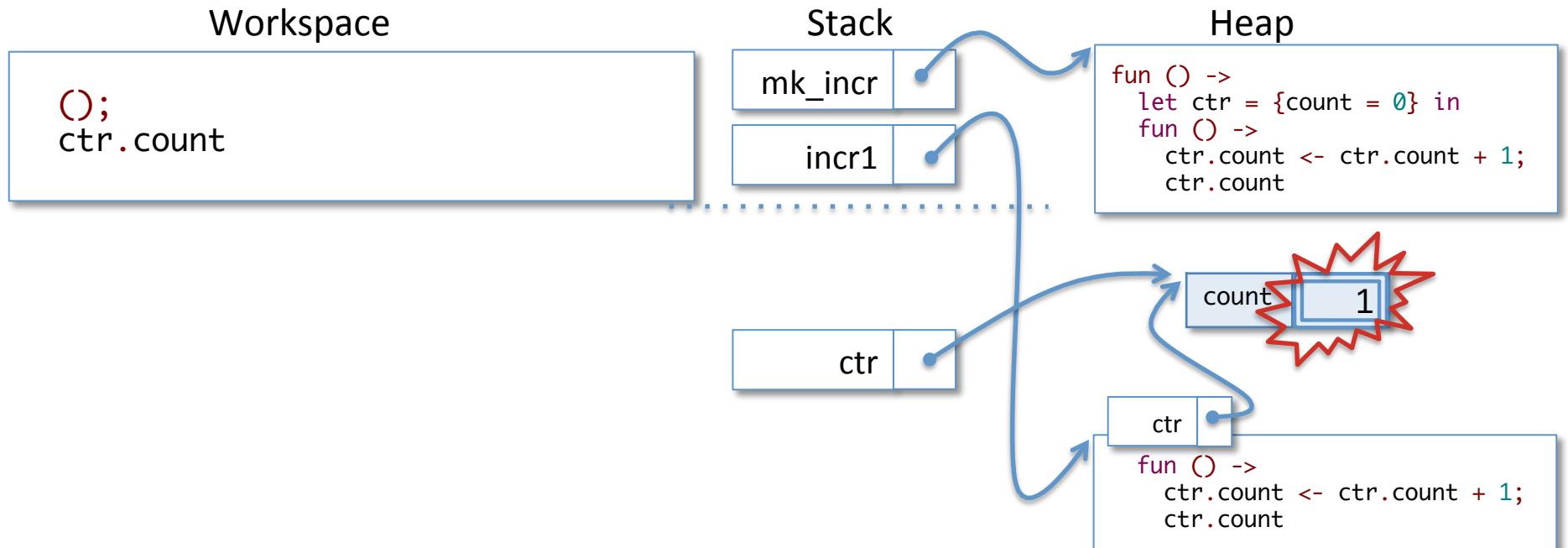
Now let's run “incr1 ()”



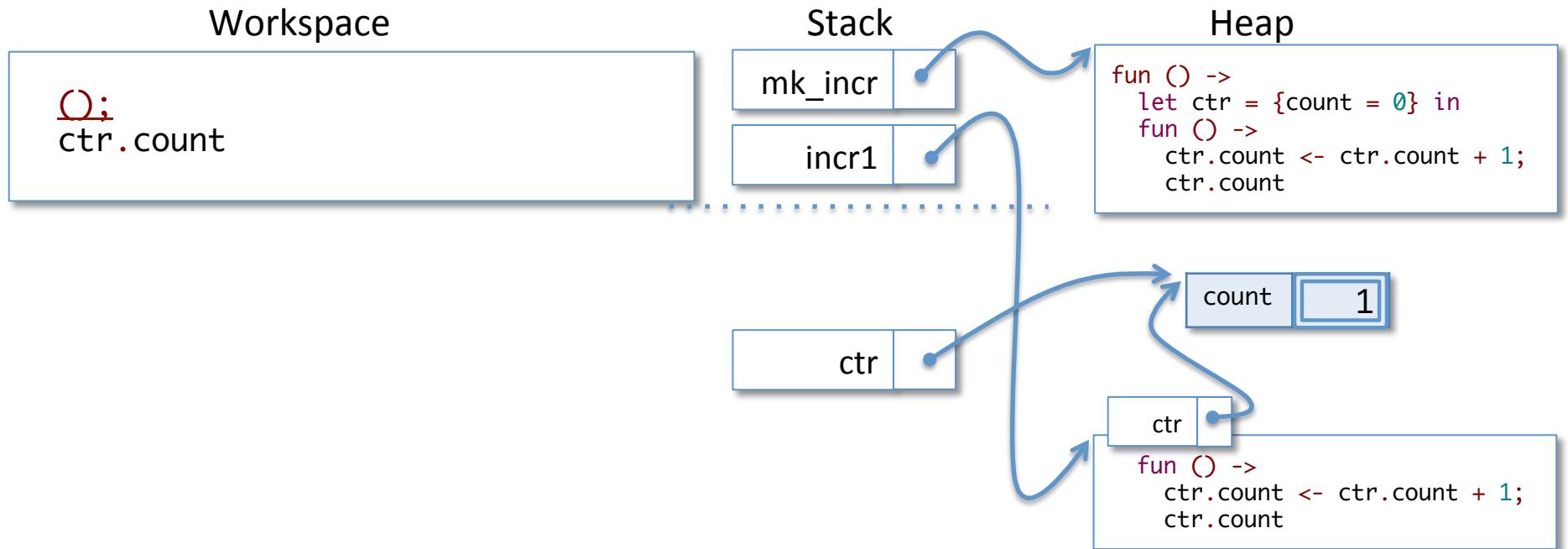
Now let's run “incr1 ()”



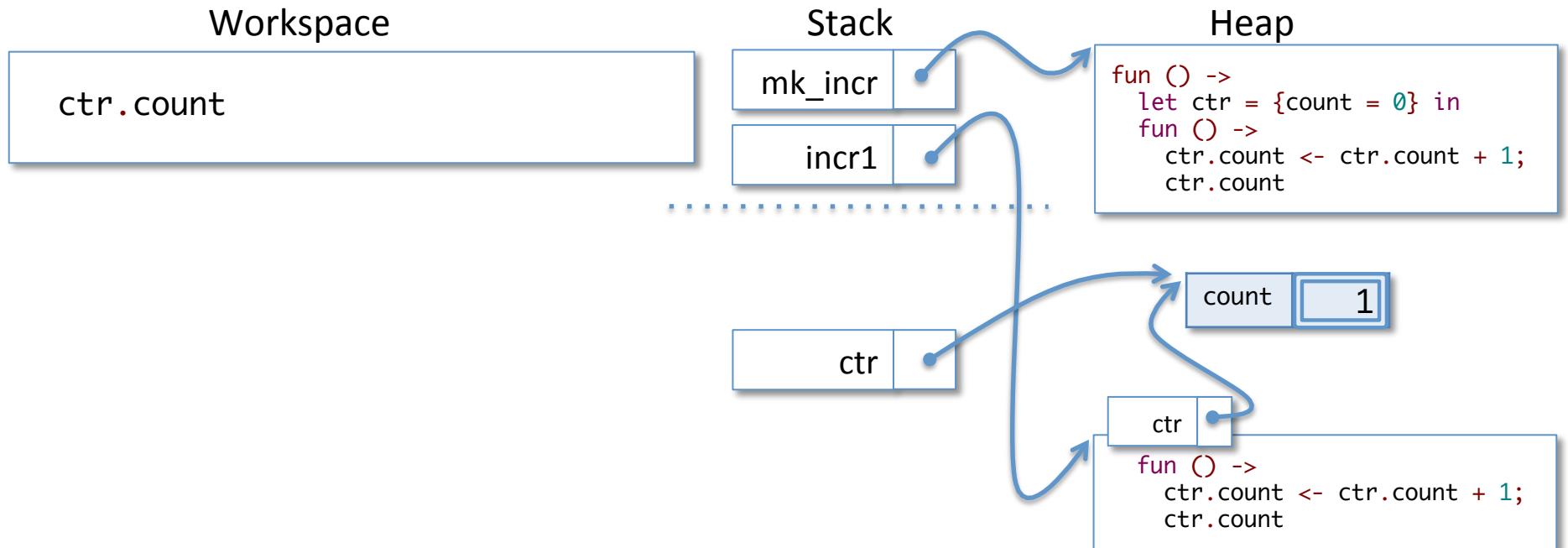
Now let's run “incr1 ()”



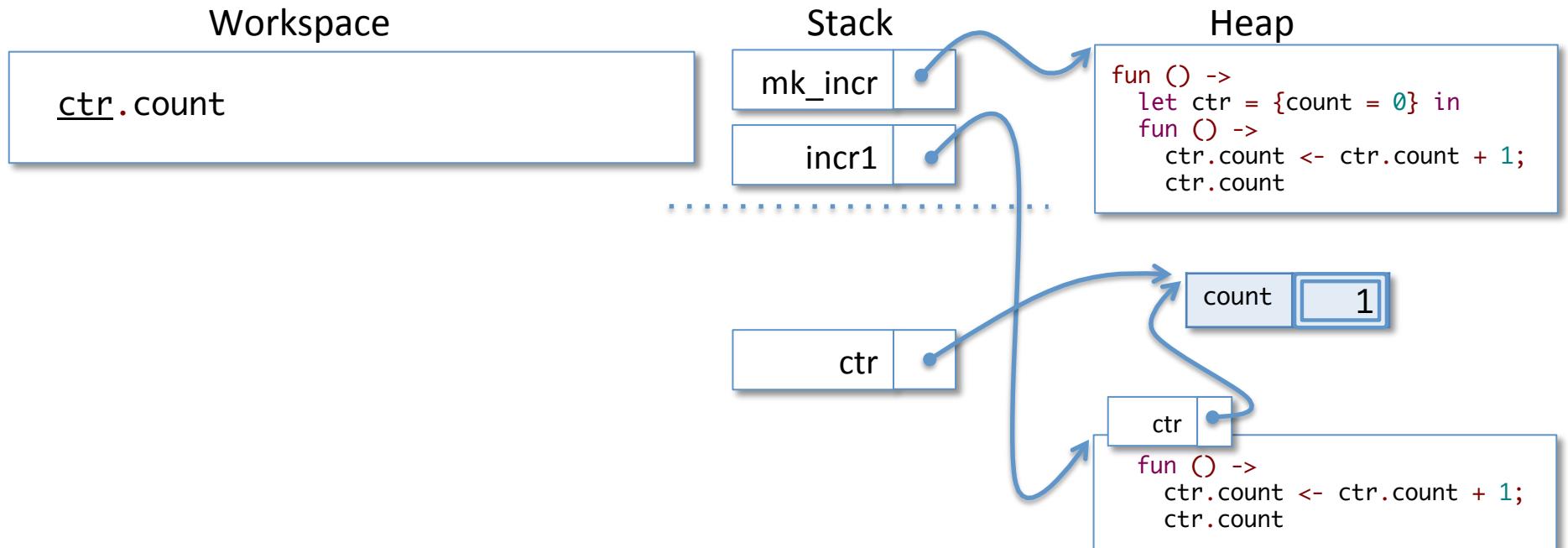
Now let's run “incr1 ()”



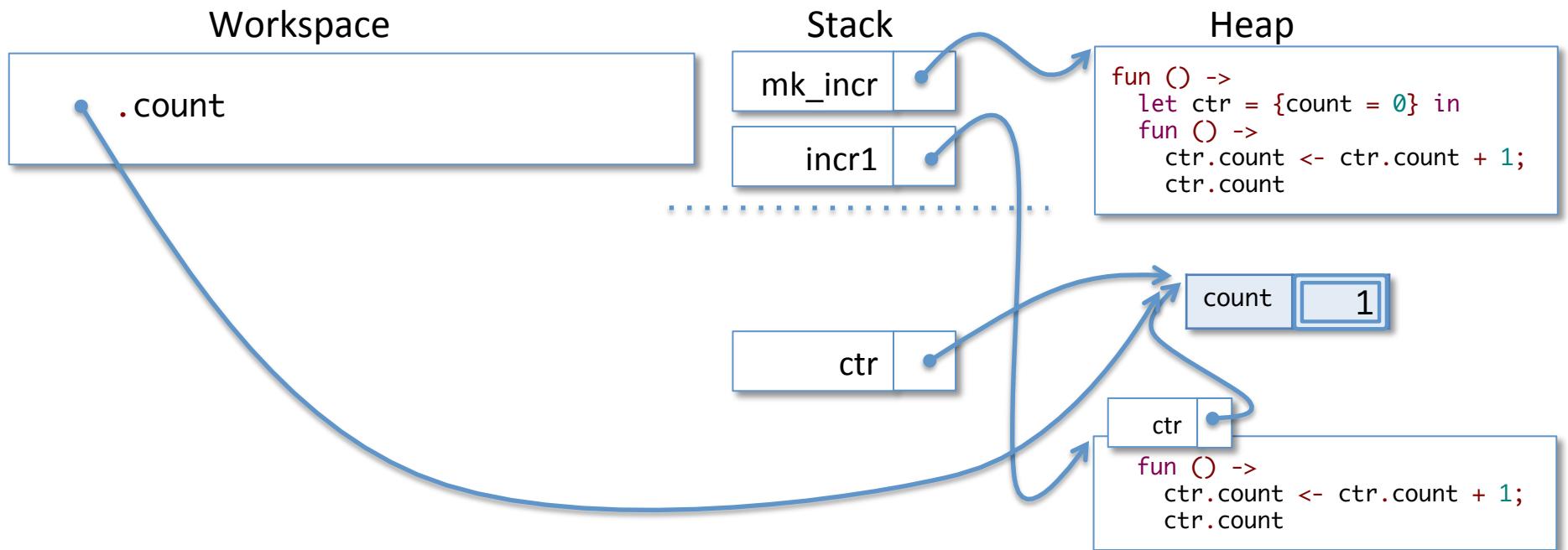
Now let's run “incr1 ()”



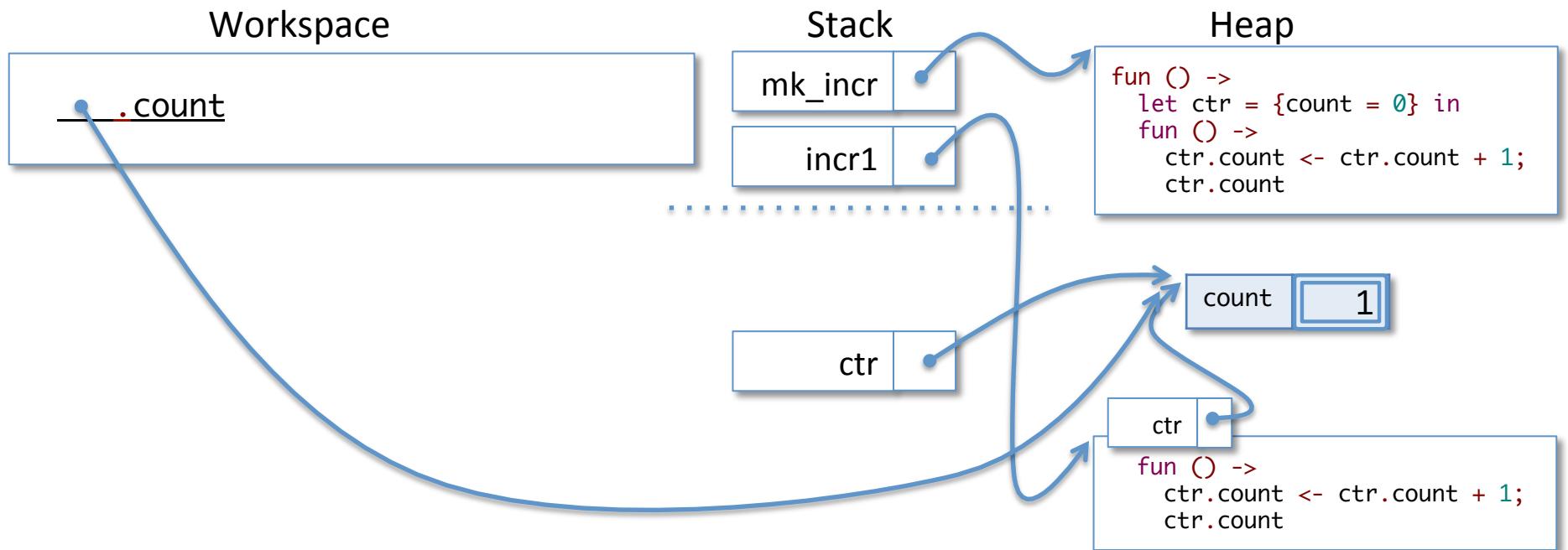
Now let's run “incr1 ()”



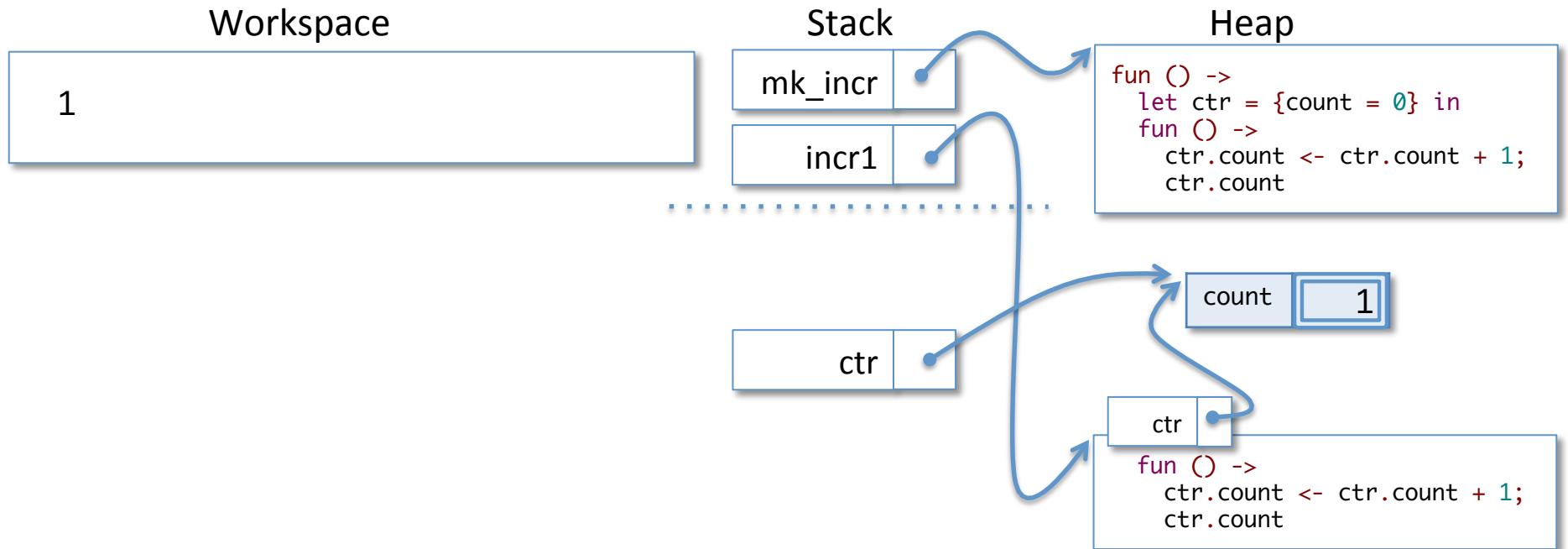
Now let's run “incr1 ()”



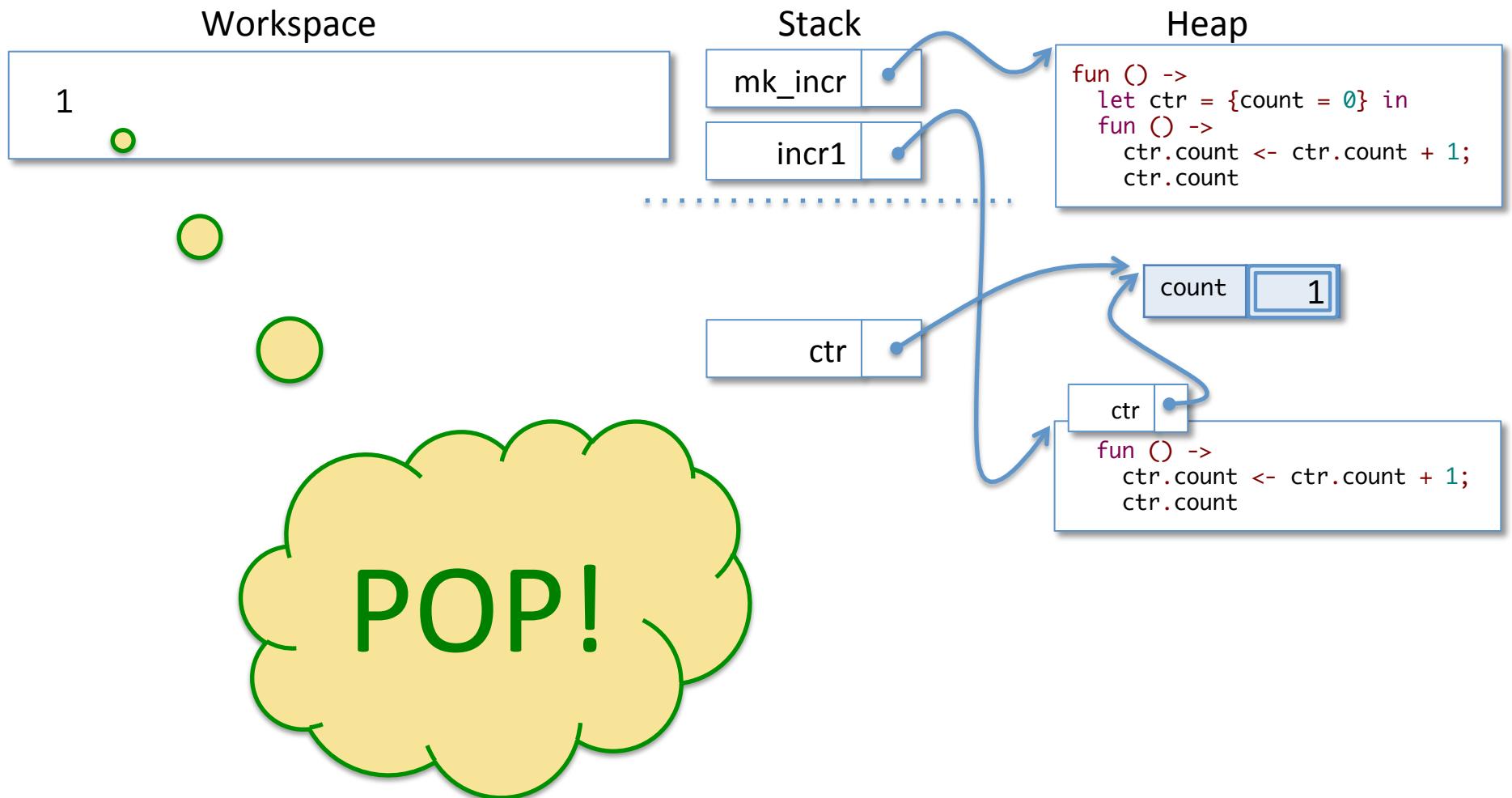
Now let's run “incr1 ()”



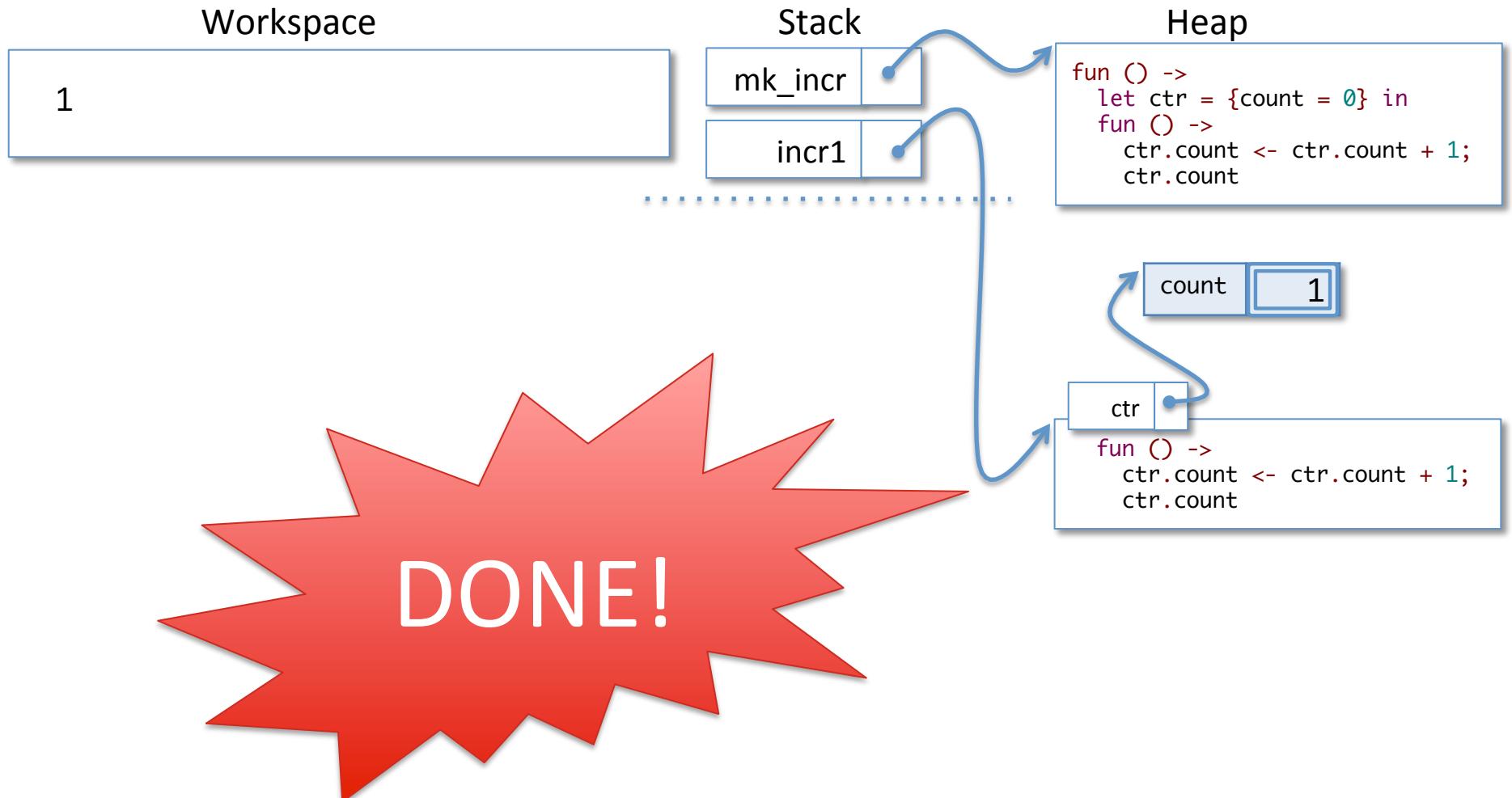
Now let's run “incr1 ()”



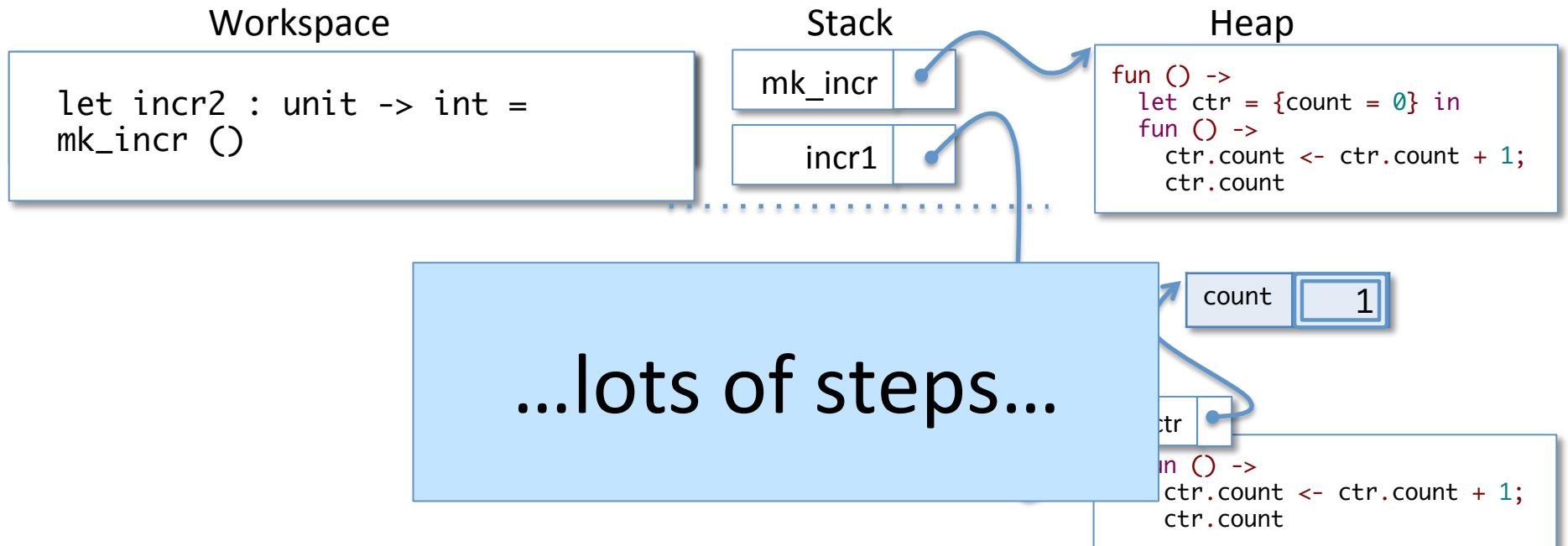
Now let's run “incr1 ()”



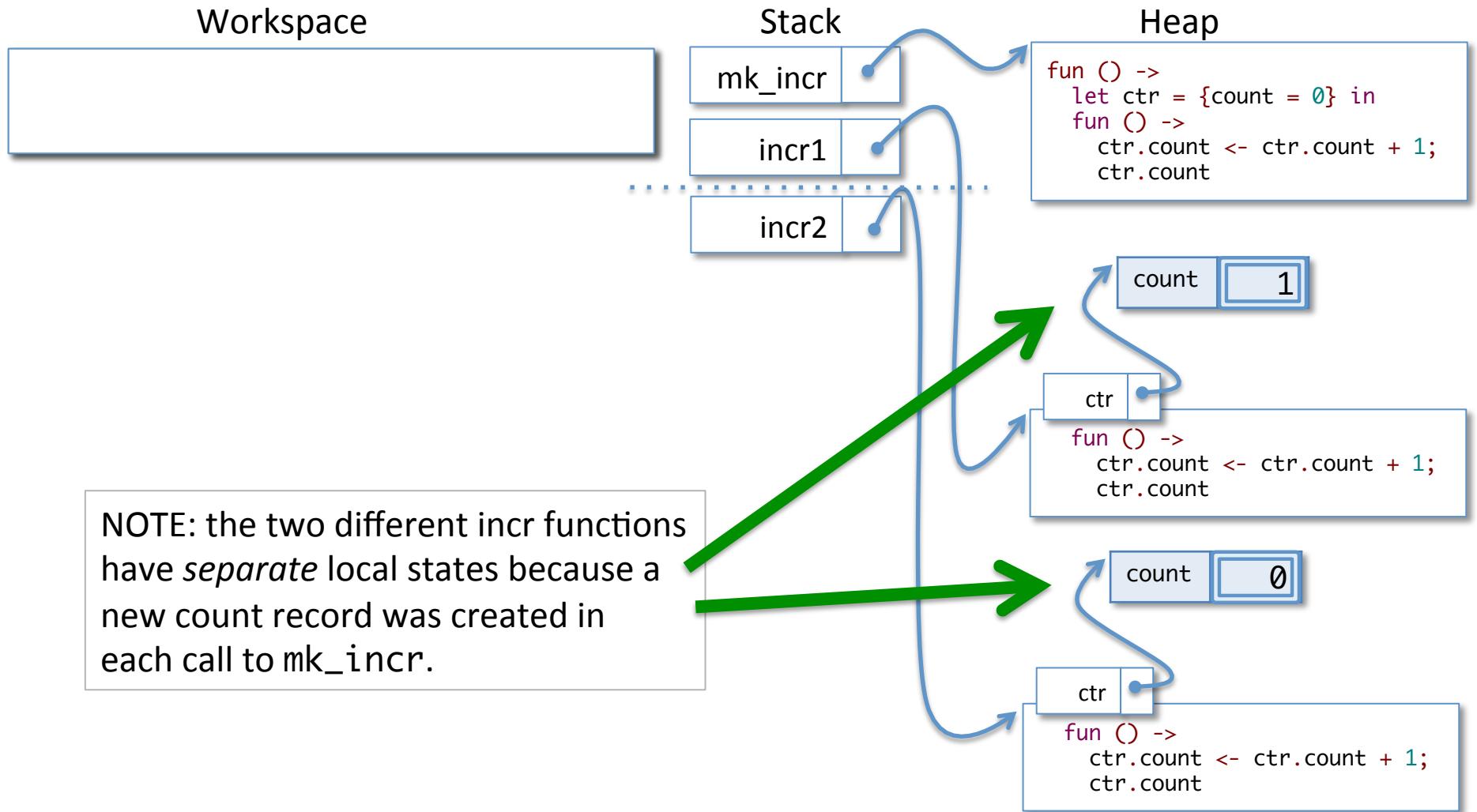
Now let's run “incr1 ()”



Now Let's run mk_incr again



After creating incr2...



Programming Languages and Techniques (CIS120)

Lecture 18

October 16th , 2017

Hidden state, Objects, GUI Library Design
Chapter 18

Announcements

- Midterm is Graded
 - Solutions & statistics will be released after makeup exams are processed
- HW05: GUI & Paint program
 - Available online
 - Due: Tuesday, October 24th at 11:59 pm
 - ***START EARLY!***
 - (No Codio exensions...)

Hidden State

Encapsulating State

An “incr” function

- Functions with internal state

```
type counter_state = { mutable count:int }

let ctr = { count = 0 }

(* each call to incr will produce the next integer *)
let incr () : int =
  ctr.count <- ctr.count + 1;
  ctr.count
```

- Drawbacks:
 - *No abstraction*: There is only one counter in the world. If we want another, we need another `counter_state` value and another `incr` function.
 - *No encapsulation*: Any other code can modify `count`, too.

Using Hidden State

- Make a function that creates a counter state and an incr function each time a counter is needed.

```
(* More useful: a counter generator: *)
let mk_incr () : unit -> int =
  (* this ctr is private to the returned function *)
  let ctr = { count = 0 } in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count

(* make one counter *)
let incr1 : unit -> int = mk_incr ()

(* make another counter *)
let incr2 : unit -> int = mk_incr ()
```

What number is printed by this program?

```
let mk_incr () : unit -> int =
  let ctr = { count = 0 } in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count

let incr1 = mk_incr () (* make one counter *)
let incr2 = mk_incr () (* and another *)

let _ = incr1 () in print_int (incr2 ())
```

1. 1
2. 2
3. 3
4. other

Running mk_incr

Workspace

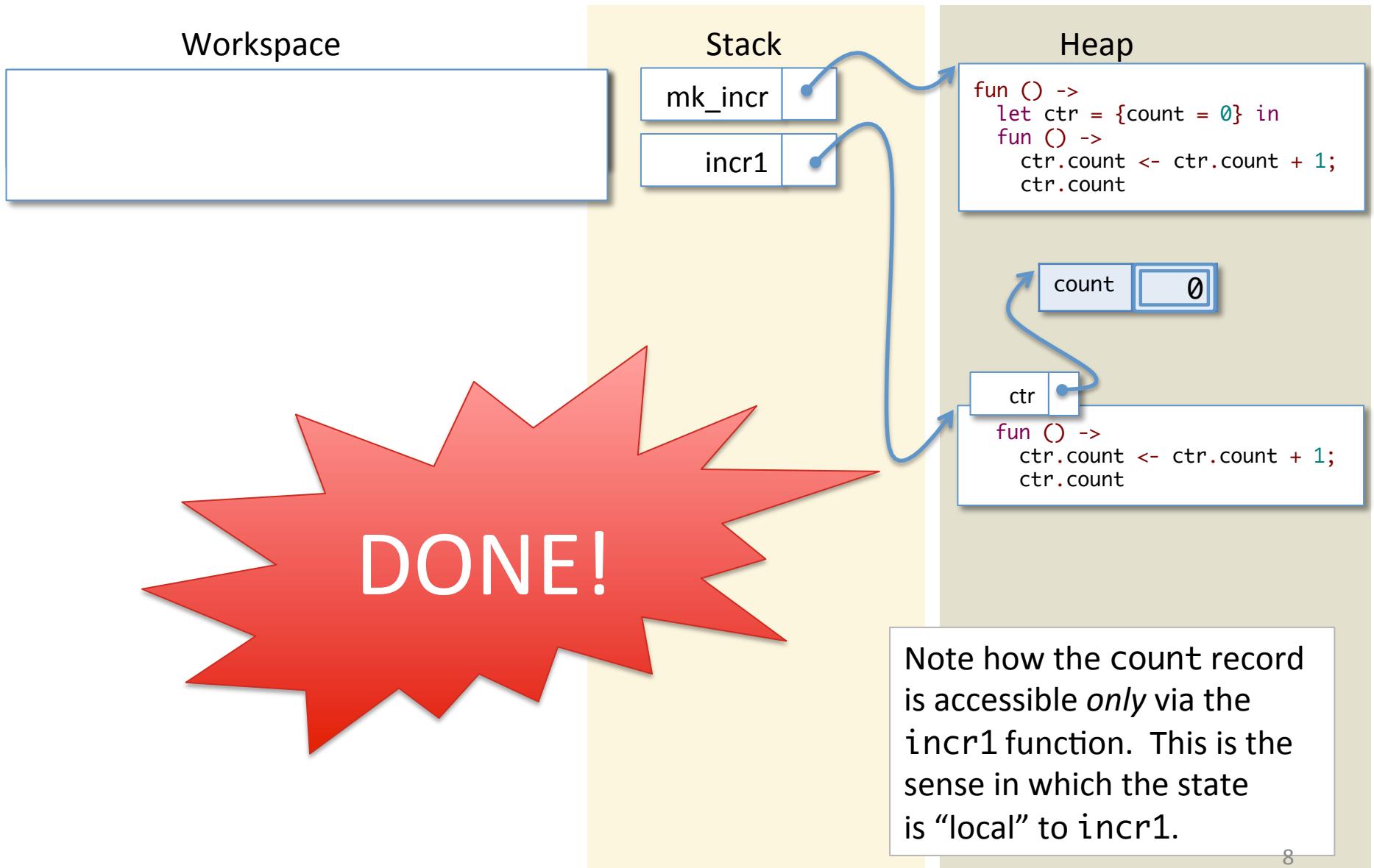
```
let mk_incr () : unit -> int =
  let ctr = {count = 0} in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count

let incr1 : unit -> int =
  mk_incr ()
```

Stack

Heap

Local Functions



Objects

One step further...

- `mk_incr` shows us how to create different instance of local state so that we can have several different counters.
- What if we want to bundle together *several* operations that share the same local state?
 - e.g. incr and decr operations that work on the same counter

Key Concept: *Object*

An object consists of:

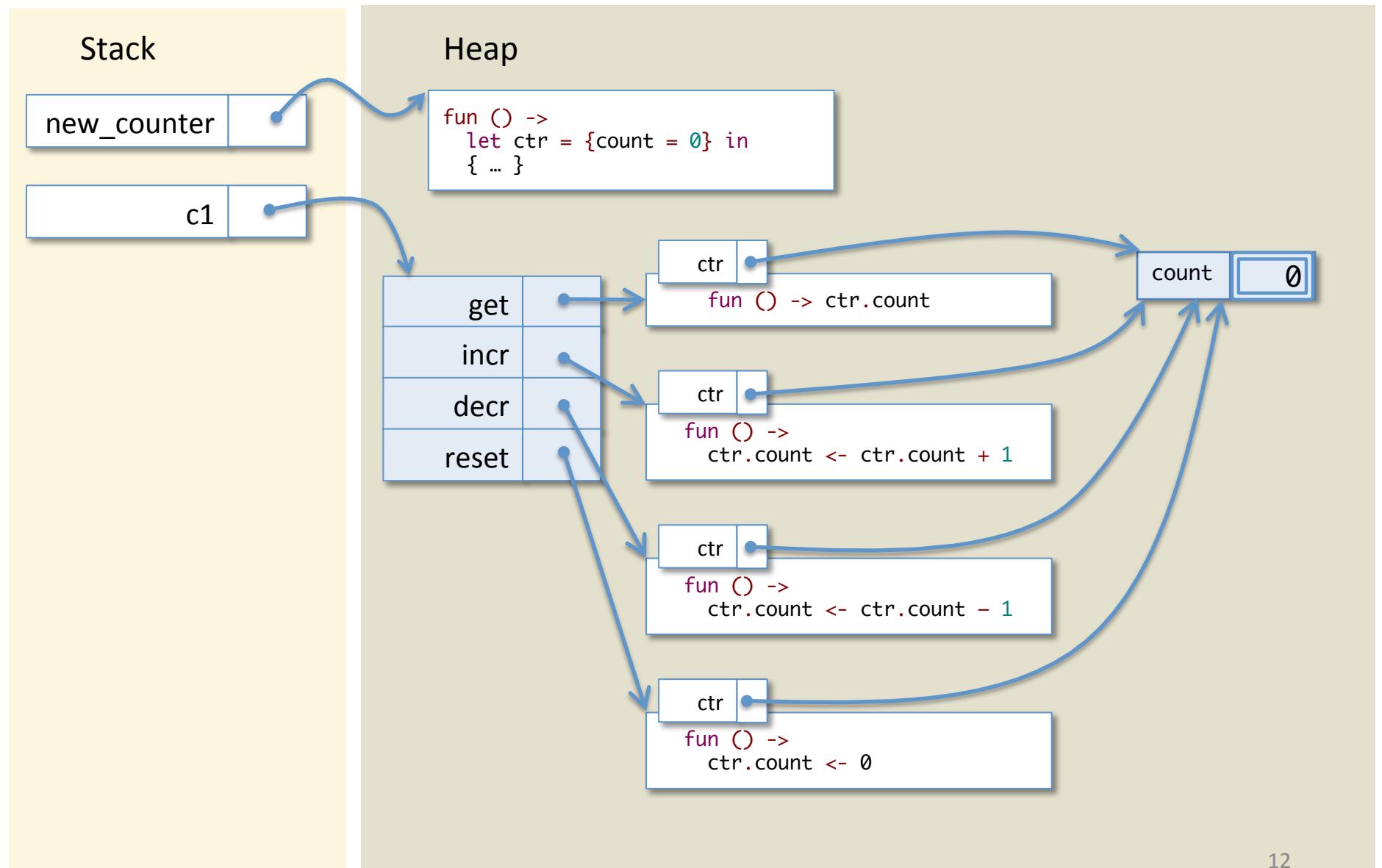
- some encapsulated mutable state (*fields*)
- a set of operations that manipulate that state (*methods*)

A Counter Object

```
(* The type of counter objects *)
type counter = {
    get   : unit -> int;
    incr  : unit -> unit;
    decr  : unit -> unit;
    reset : unit -> unit;
}

(* Create a fresh counter object with hidden state: *)
let new_counter () : counter =
    let ctr = {count = 0} in
    {
        get   = (fun () -> ctr.count) ;
        incr  = (fun () -> ctr.count <- ctr.count + 1) ;
        decr  = (fun () -> ctr.count <- ctr.count - 1) ;
        reset = (fun () -> ctr.count <- 0) ;
    }
```

let c1 = new_counter ()



Using Counter Objects

```
(* a helper function to create a nice string for printing *)
let ctr_string (s:string) (i:int) =
  s ^ ".ctr = " ^ (string_of_int i) ^ "\n"

let c1 = new_counter ()
let c2 = new_counter ()

;; print_string (ctr_string "c1" (c1.get ()))
;; c1.incr ()
;; c1.incr ()
;; print_string (ctr_string "c1" (c1.get ()))
;; c1.decr ()
;; print_string (ctr_string "c1" (c1.get ()))
;; c2.incr ()
;; print_string (ctr_string "c2" (c2.get ()))
;; c2.decr ()
;; print_string (ctr_string "c2" (c2.get ()))
```

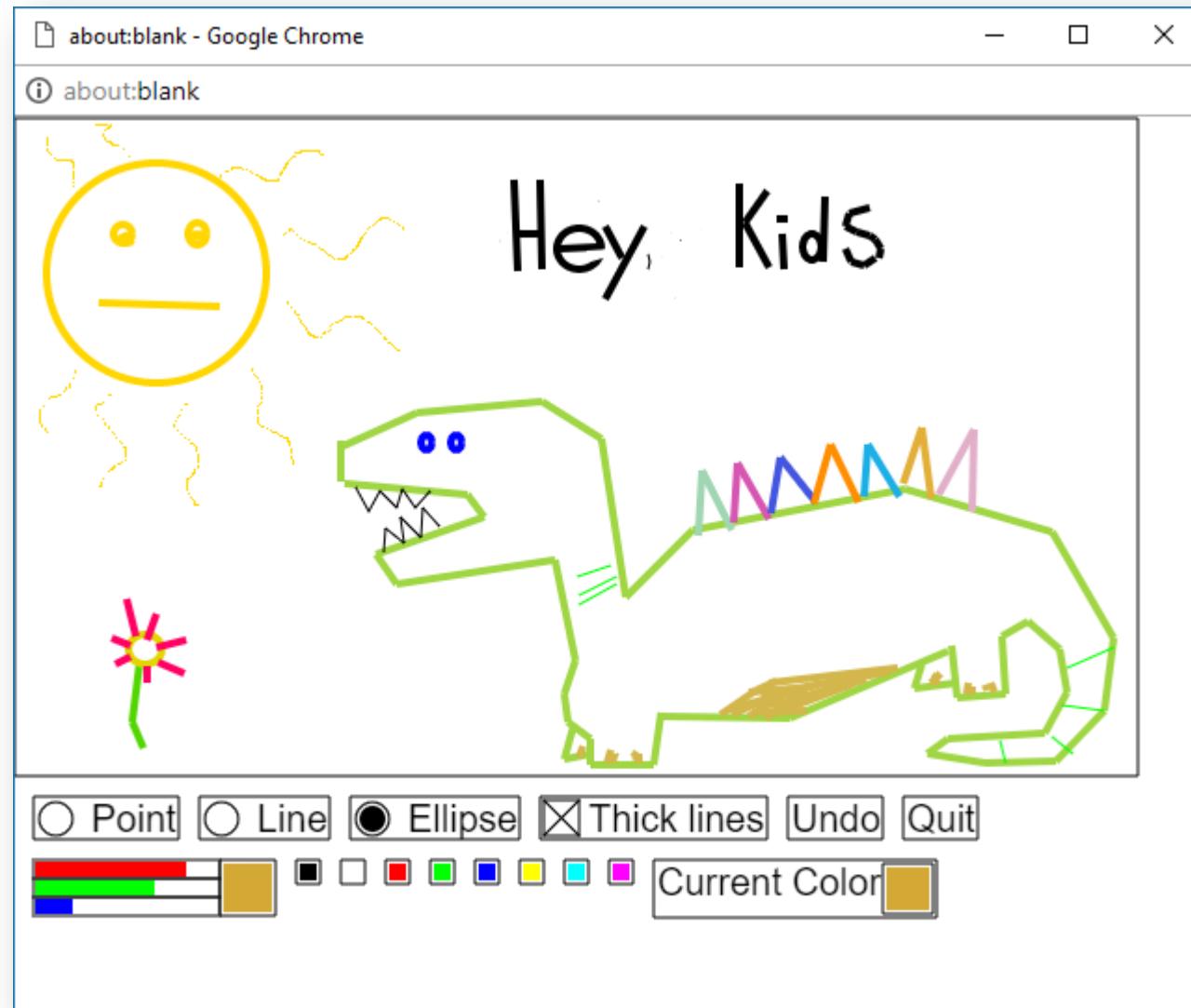
Objects and GUIs

Where we're going...

- HW 5: Build a GUI library and client application *from scratch* in OCaml
- Goals:
 - Practice with *first-class functions* and *hidden state*
 - Bridge to object-oriented programming in Java
 - Illustrate the *event-driven* programming model
 - Give you a feel for how GUI libraries (like Java's Swing) work
 - Apply everything we've seen so far to do some pretty serious programming

Have you ever used a GUI library (such as Java's Swing) to construct a user interface?

Building a GUI library & application



Step #1: Understand the Problem

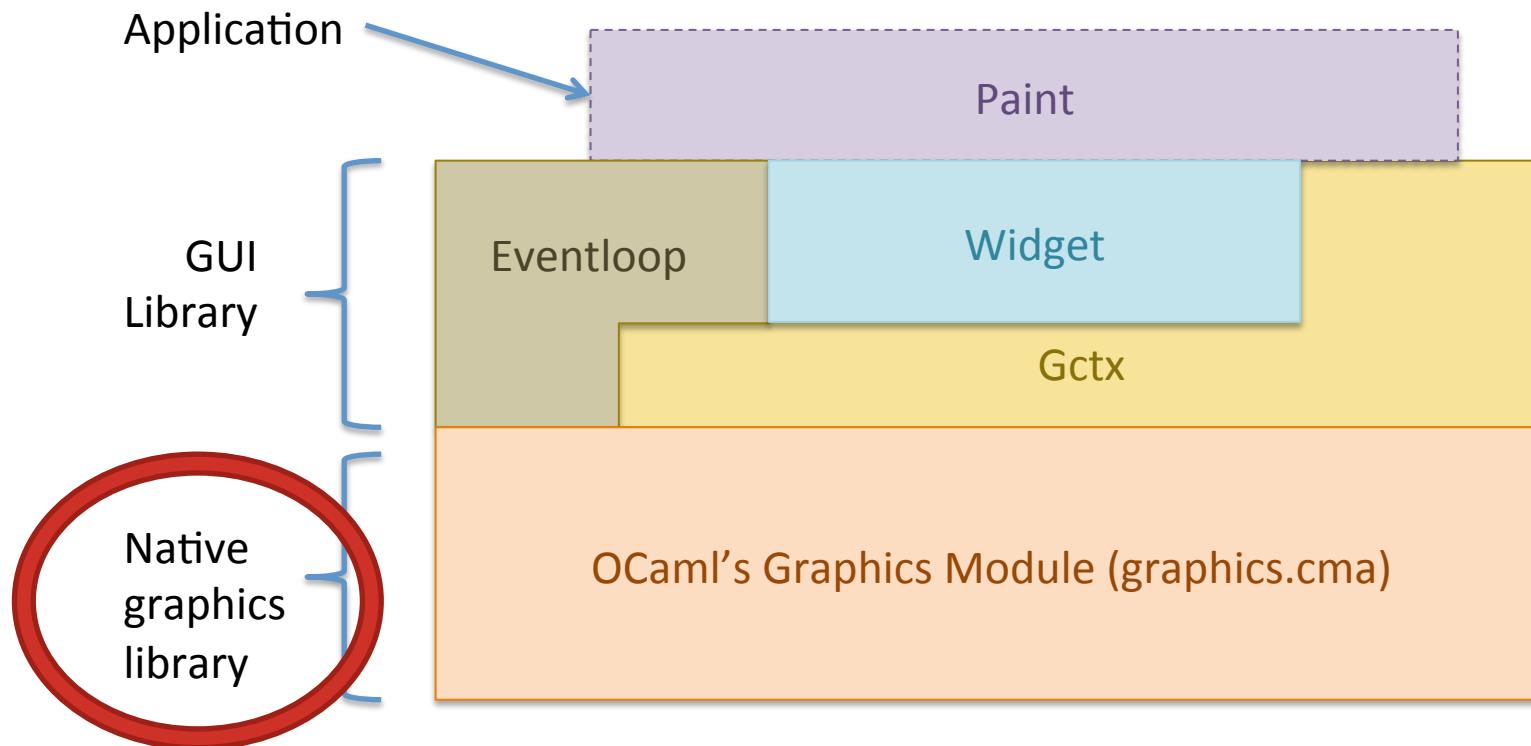
- We don't want to build just one graphical application: rather, as much as possible of our code should be *reusable*.
- What are the concepts involved in *GUI libraries* and how do they relate to each other?
- How can we separate the various concerns on the project?

GUI Library Design

putting objects to work

Step #2, Interfaces: Project Architecture*

*Subsequent program snippets will be color-coded according to this diagram



Goal of the GUI library: provide a consistent layer of abstraction *between* the application (Paint) and the Graphics module.

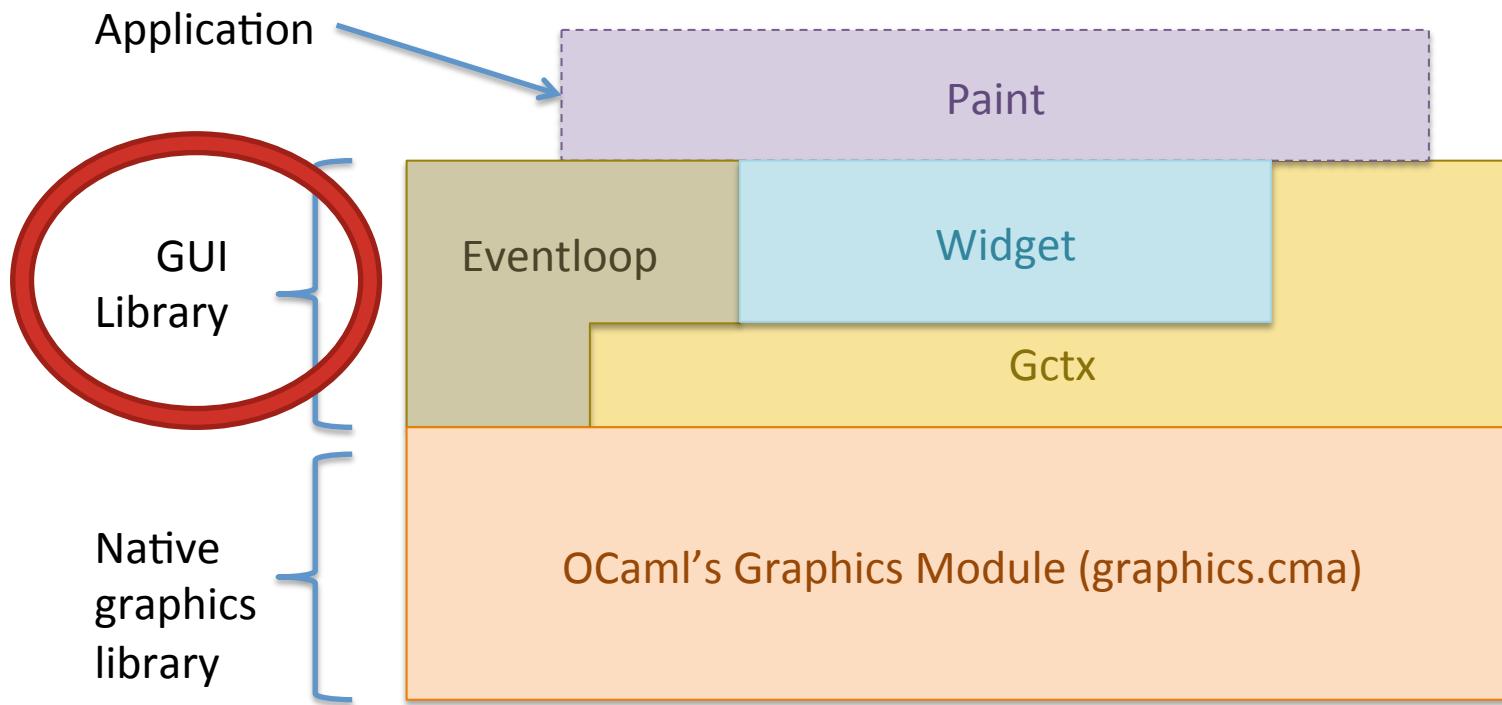
Starting point: The low-level Graphics module

- OCaml's `Graphics*` library provides *very basic* primitives for:
 - Creating a window
 - Drawing various shapes: points, lines, text, rectangles, circles, etc.
 - Getting the mouse position, whether the mouse button is pressed, what key is pressed, etc.
 - See: <http://caml.inria.fr/pub/docs/manual-ocaml/libref/Graphics.html>
- How do we go from that to a full-blown GUI library?

*Note: We actually have *two* Graphics libraries, one for running "natively" and one for running in the browser. We have configured the project so that you can refer to either one using the module alias `Graphics`.

For use within the browser, we use a tool called `js_of_ocaml` that translates OCaml-compiled bytecode into javascript. There are some rendering differences between the native and browser versions.

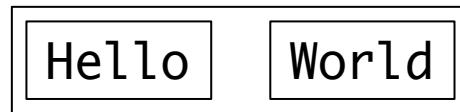
Interfaces: Project Architecture



Goal of the GUI library: provide a consistent layer of abstraction *between* the application (Paint) and the Graphics module.

GUI terminology – Widget*

- Basic element of GUIs: examples include buttons, checkboxes, windows, textboxes, canvases, scrollbars, labels
- Every widget
 - has a **position** on the screen
 - knows how to **display** itself
 - (knows how to react to events like mouse clicks)
- May be composed of other sub-widgets, for laying out complex interfaces

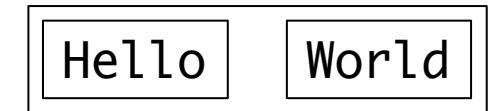
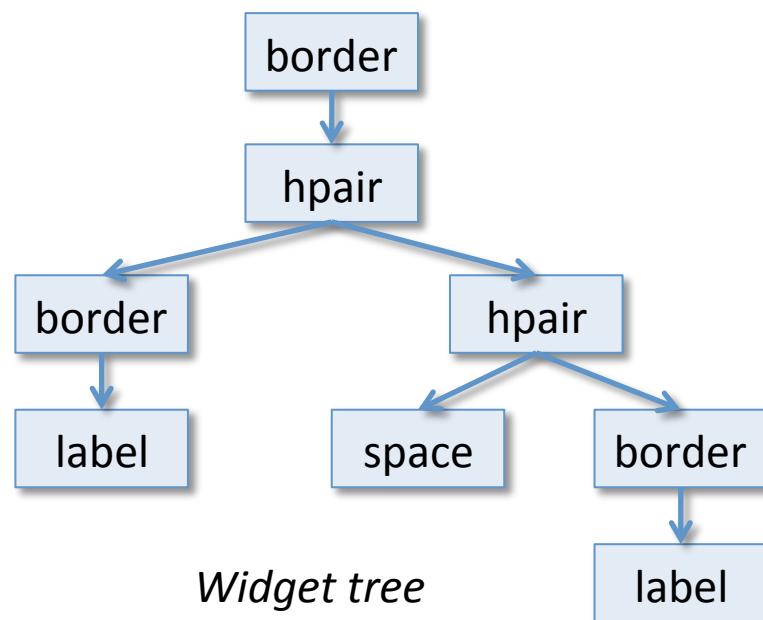


*Each GUI library uses its own naming convention for what we call “widgets.” Java Swing calls them “Components”; iOS UIKit calls them “UIViews”; WINAPI, GTK+, X11’s widgets, etc....

A “Hello World” application

```
(* Create some simple label widgets *)
let l1 = label "Hello"
let l2 = label "World"
(* Compose them horizontally, adding some borders *)
let h = border
    (hpair (border l1)
        (hpair (space (10,10)) (border l2)))
```

swdemo.ml



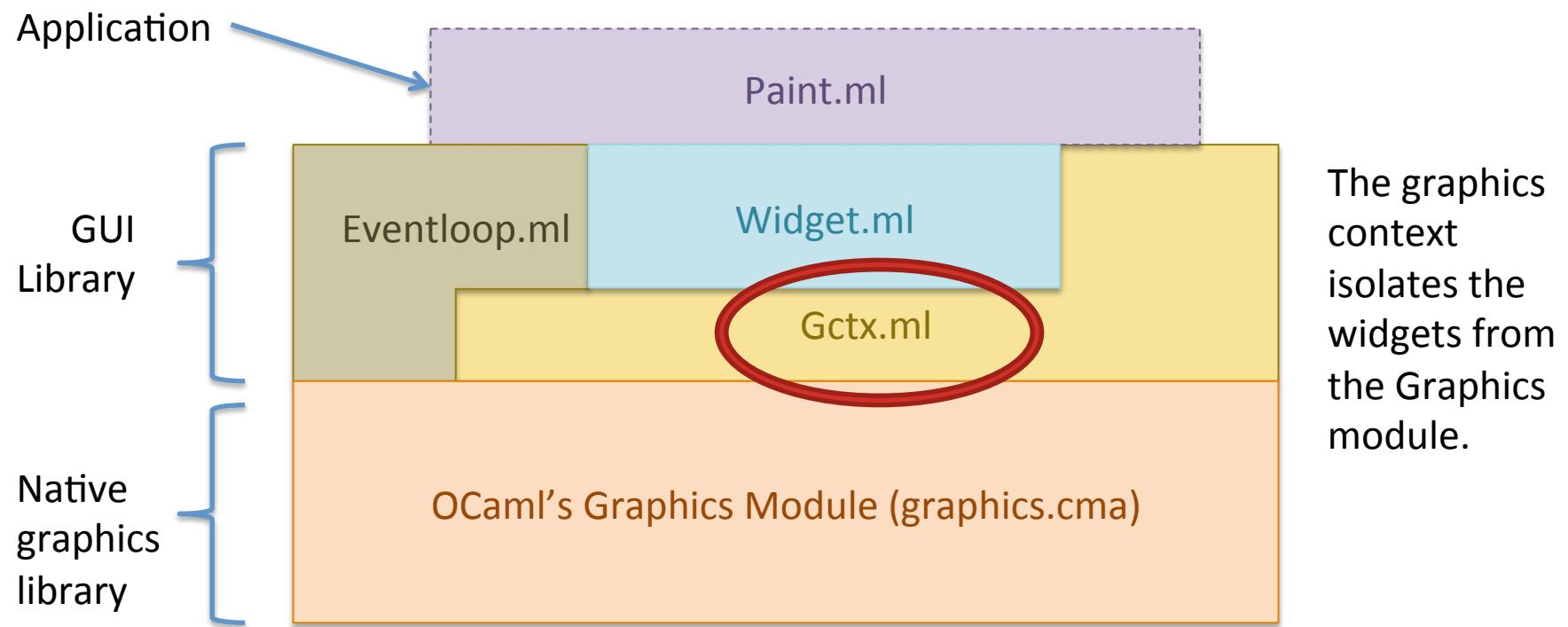
On the screen

Module: Gctx

Contextualizes graphics drawing operations

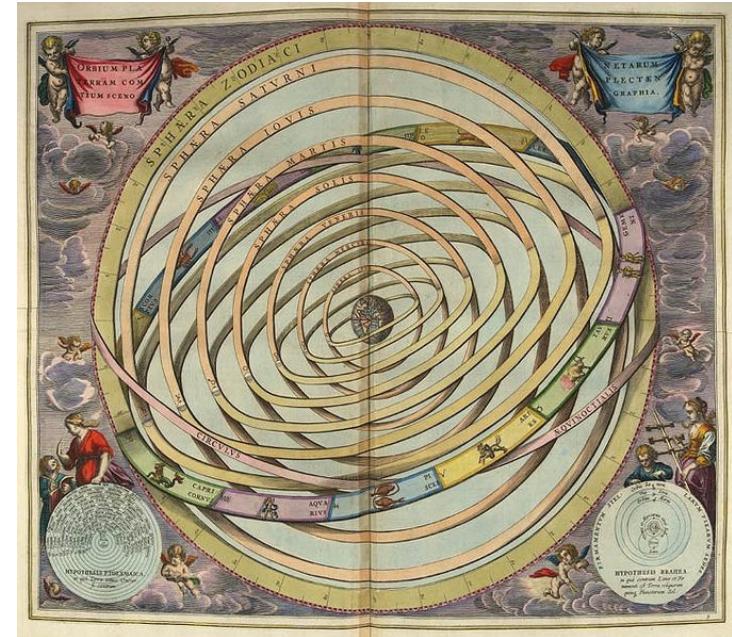
Challenge: Widget Layout

- Widgets are “things drawn on the screen”. How to make them location independent?
- Idea: Use a *graphics context* to make drawing *relative* to the widget’s current position



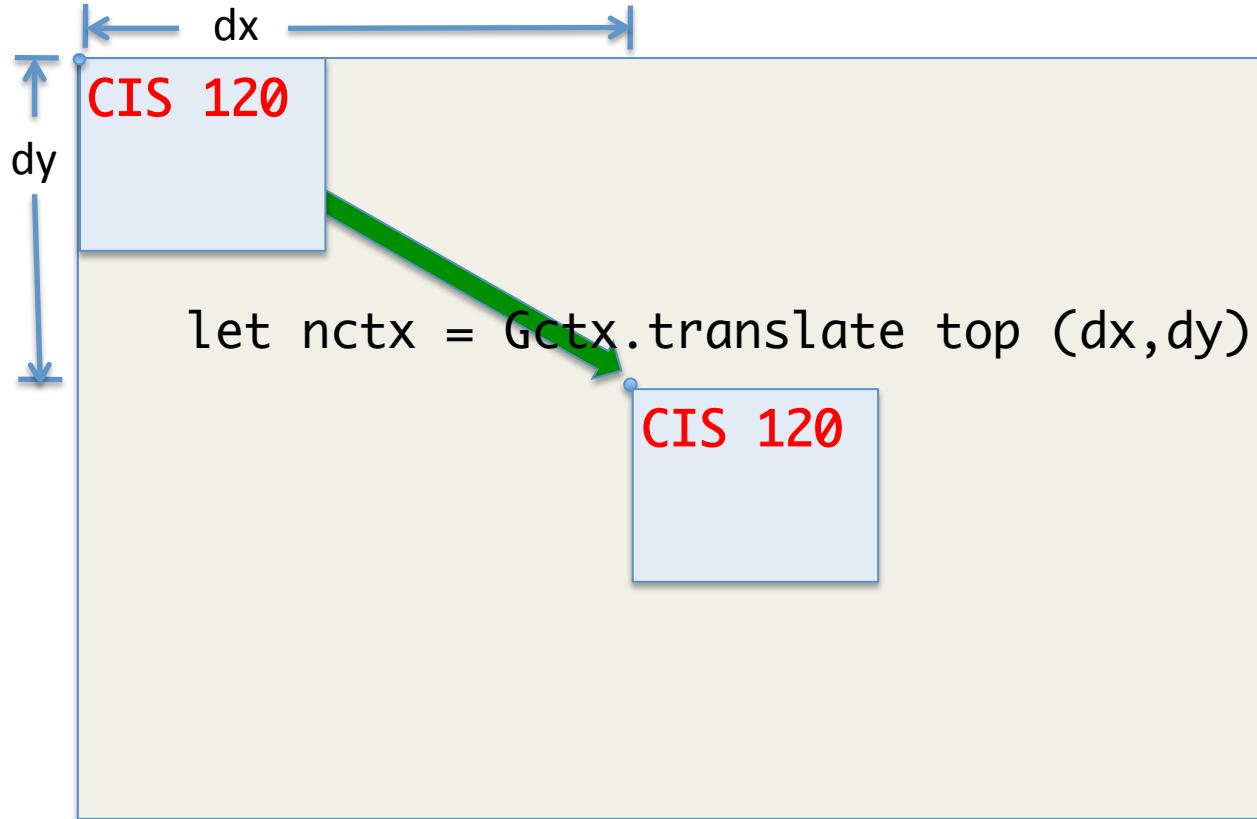
GUI terminology – Graphics Context

- Wraps OCaml Graphics library; puts drawing operations “in context”
- Translates coordinates
 - *Flips* between OCaml and “standard” coordinates so origin is top-left
 - *Translates* coordinates so all widgets can pretend that they are at the origin
- Also aggregates information about the way things are drawn
 - foreground color
 - line width



Graphics Contexts

```
let top = Gctx.top_level
```



```
draw_string top (0,10) "CIS 120";  
draw_string nctx (0,10) "CIS 120"
```

```
repaint = fun g -> draw_rect g (0,0) (20,20);  
                      draw_string g (0,10) "CIS 120"
```

Module Gctx

```
(** The main (abstract) type of graphics contexts. *)
type gctx

(** The top-level graphics context *)
val top_level : gctx

(** A widget-relative position *)
type position = int * int

(** Display text at the given (relative) position *)
val draw_string : gctx -> position -> string -> unit
(** Draw a line between the two specified positions *)
val draw_line : gctx -> position -> position -> unit

(** Produce a new gctx shifted by (dx,dy) *)
val translate : gctx -> int * int -> gctx
(** Produce a new gctx with a different pen color *)
val with_color : gctx -> color -> gctx
```

Event loop with graphics contexts

- Main loop of any GUI application:

```
let run (w:widget) : unit =
  let q = Gctx.top_level in
  ...do some graphics setup...
  let rec loop () : unit =
    Graphics.clear_graph ();
    w.repaint g;
    Graphics.synchronize ();
    ...wait for user input (mouse movement, key press)...
    ...inform w about the input so widgets can react to it...
    loop ()
  in
  loop ()
```

- Repaint is relative to a graphics context g

Programming Languages and Techniques (CIS120)

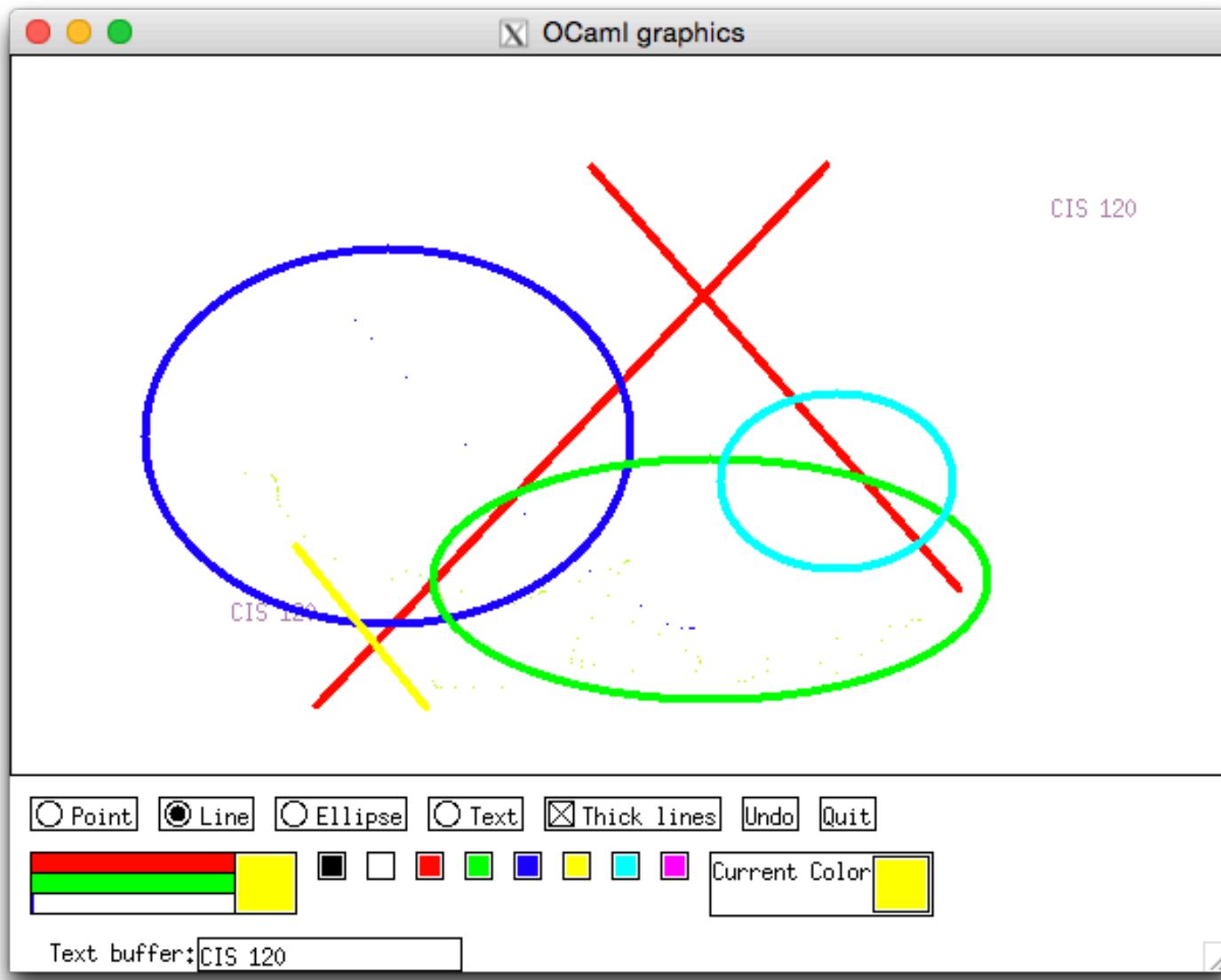
Lecture 19
October 18, 2017

GUI library: Simple Widgets

Announcements

- HW05: GUI programming
 - Due: **Tuesday, October 24** at 11:59:59pm
 - *Graded (mostly) manually*
 - *Submission checks for compilation, few auto tests*
 - *Only LAST submission will be graded*
 - *This project is challenging:*
 - Requires working with *multiple* levels of abstraction.
 - Managing state in the paint program is a bit tricky.
 - ***START IMMEDIATELY!!***

Building a GUI library & application

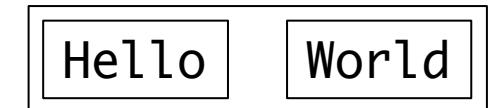
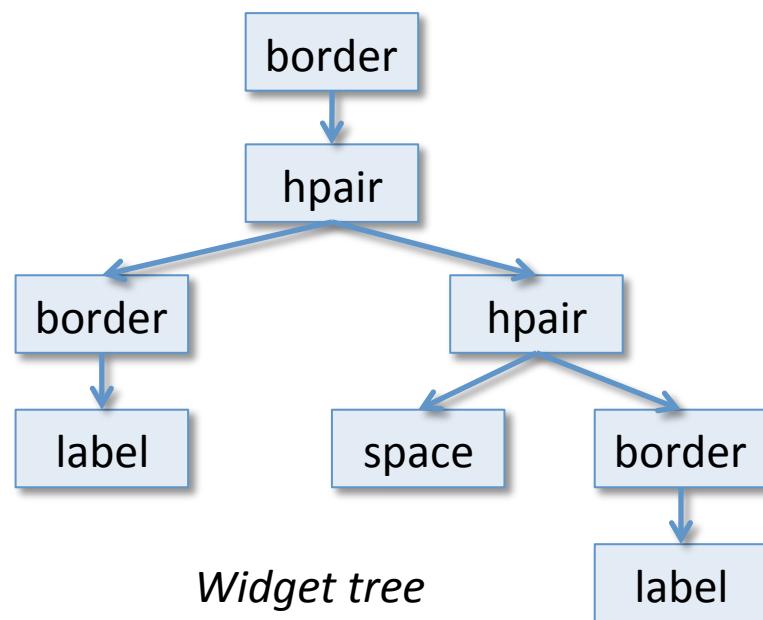


Review: A “Hello World” application

```
(* Create some simple label widgets *)
let l1 = label "Hello"
let l2 = label "World"

(* Compose them horizontally, adding some borders *)
let h = border
    (hpair (border l1)
        (hpair (space (10,10)) (border l2)))
```

swdemo.ml



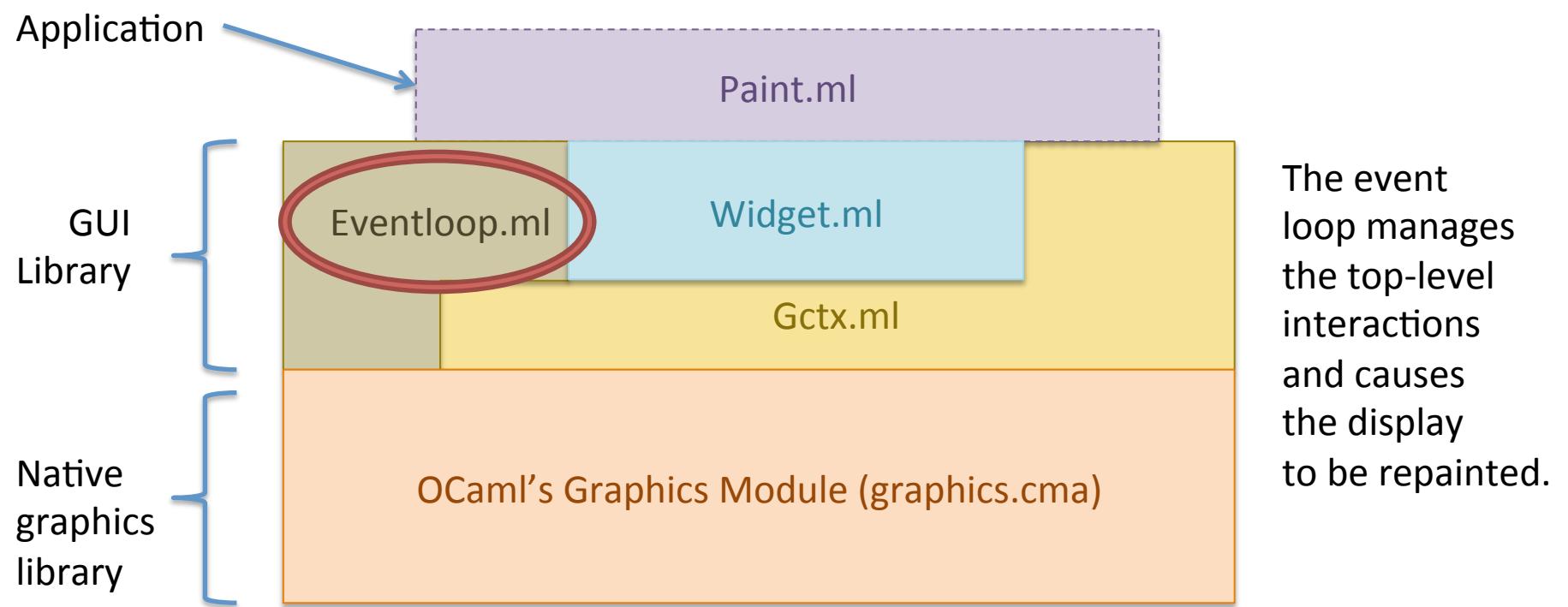
On the screen

Module: EventLoop

Top-level driver

GUI Architecture

- The eventloop is the main "driver" of a GUI application
 - For now: focus on how widgets are drawn on the screen
 - Later: deal with event handling



GUI terminology: “event loop”

- Main loop of any GUI application:

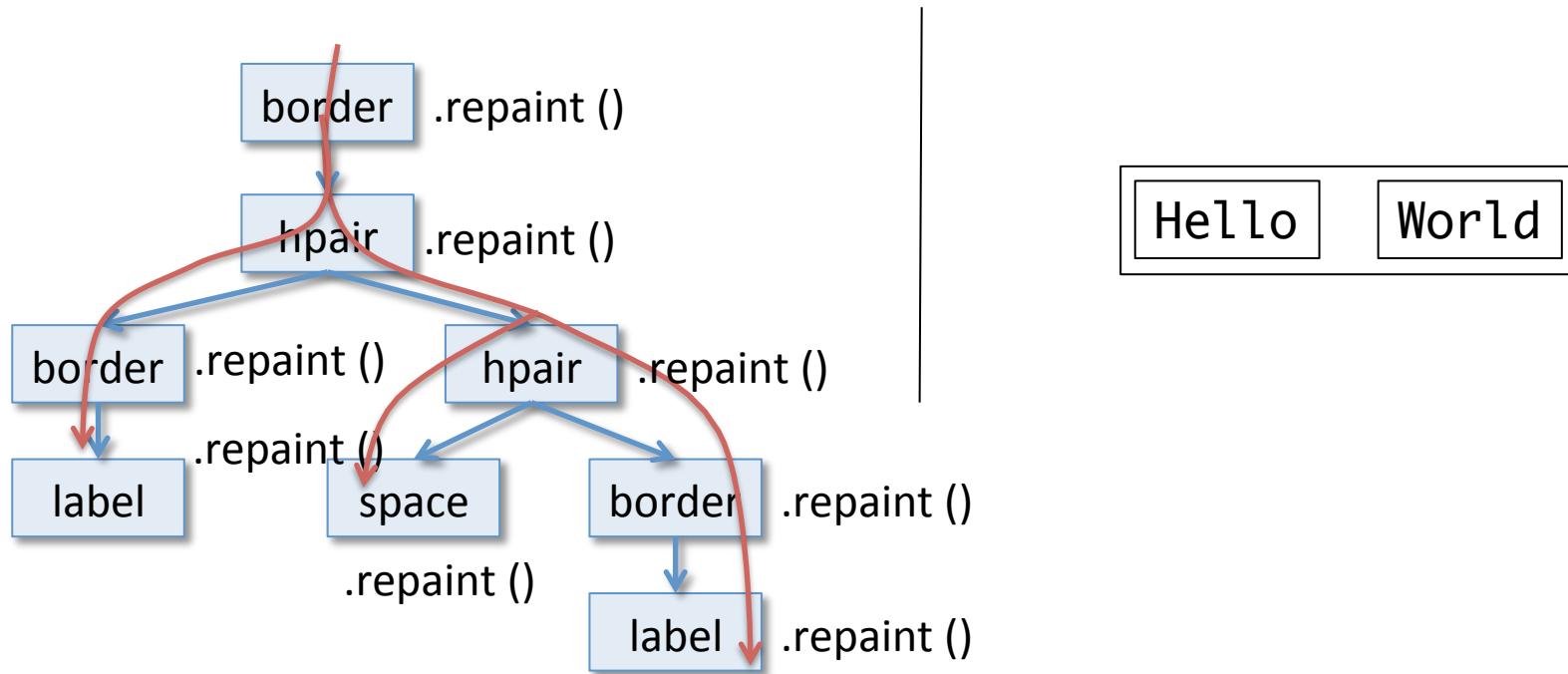
```
let run (w:widget) : unit =
    ...wait for user input (mouse movement, key press)...
Graphics.loop [Graphics.Mouse_motion; Graphics.Button_down;
               Graphics.Button_up; Graphics.Key_pressed]
(fun status ->
  clear_graph ();
  w.repaint ();   ...update the widget's appearance...
  ...inform w about the event so widgets can react to it...
)
```

```
let rec loop (es:event list) (f:status -> unit) : unit =
  let status = wait_next_event es in
  (f status); loop es f
```

- Takes “top-level” widget w as argument. That widget *contains* all others in the application.

Drawing: Containers

Container widgets propagate repaint commands to their children:



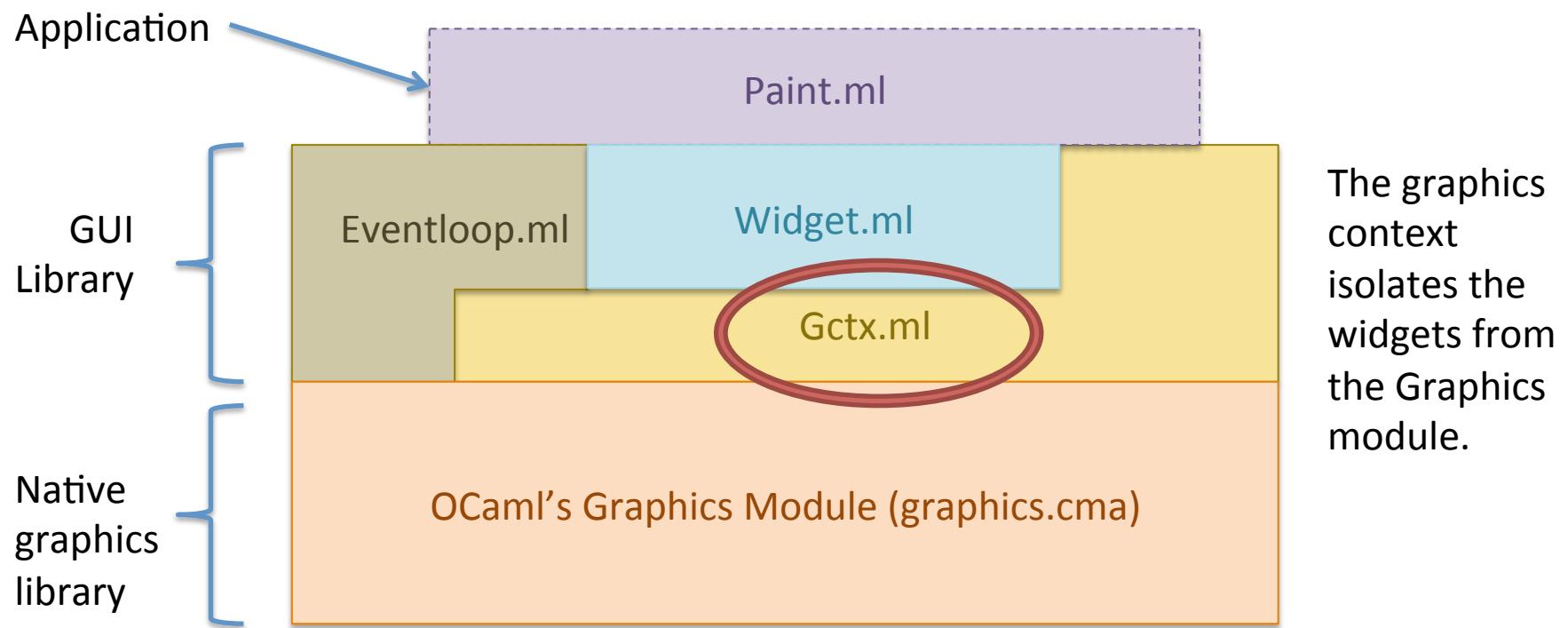
Challenge: How can we make it so that the functions that draw widgets in **different places** on the window are *location independent*?

Module: Gctx

Contextualizes graphics drawing operations

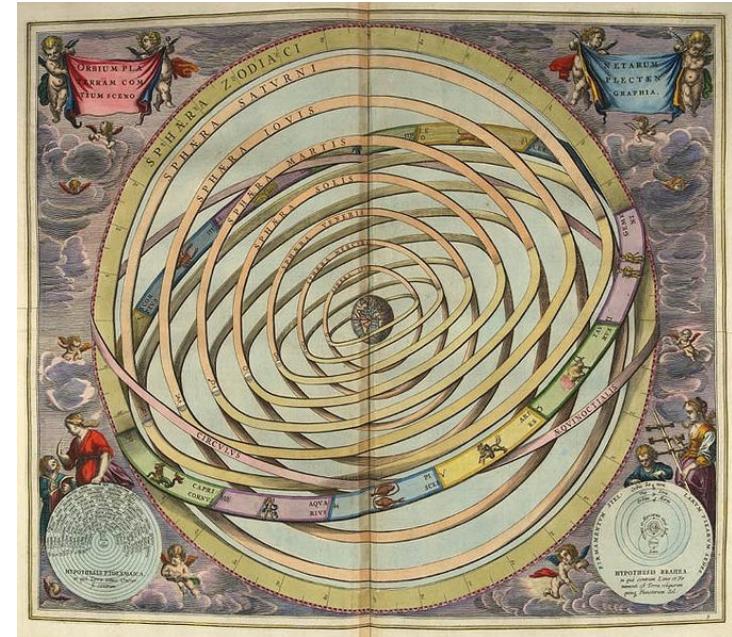
Challenge: Widget Layout

- Widgets are “things drawn on the screen”. How to make them location independent?
- Idea: Use a *graphics context* to make drawing *relative* to the widget’s current position



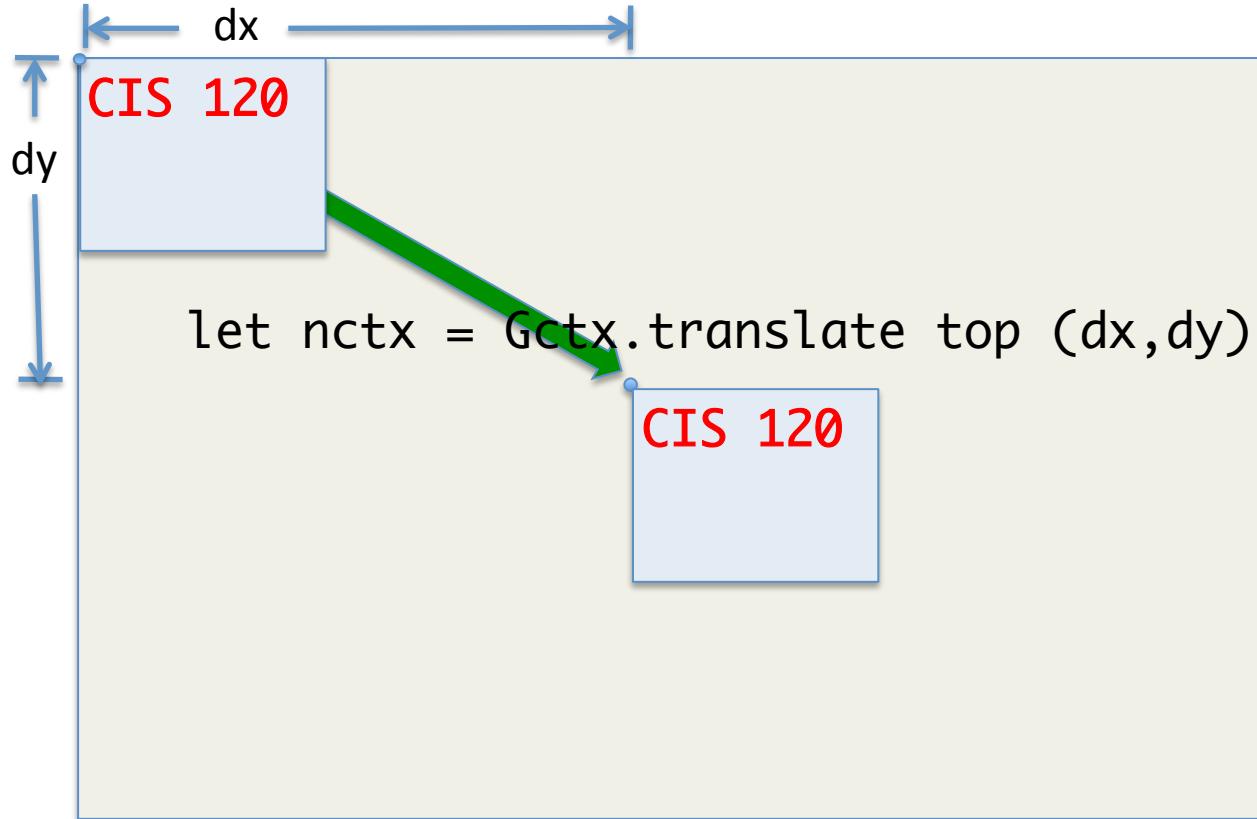
GUI terminology – Graphics Context

- Wraps OCaml Graphics library; puts drawing operations “in context”
- Translates coordinates
 - *Flips* between OCaml and “standard” coordinates so origin is top-left
 - *Translates* coordinates so all widgets can pretend that they are at the origin
- Also aggregates information about the way things are drawn
 - foreground color
 - line width



Graphics Contexts

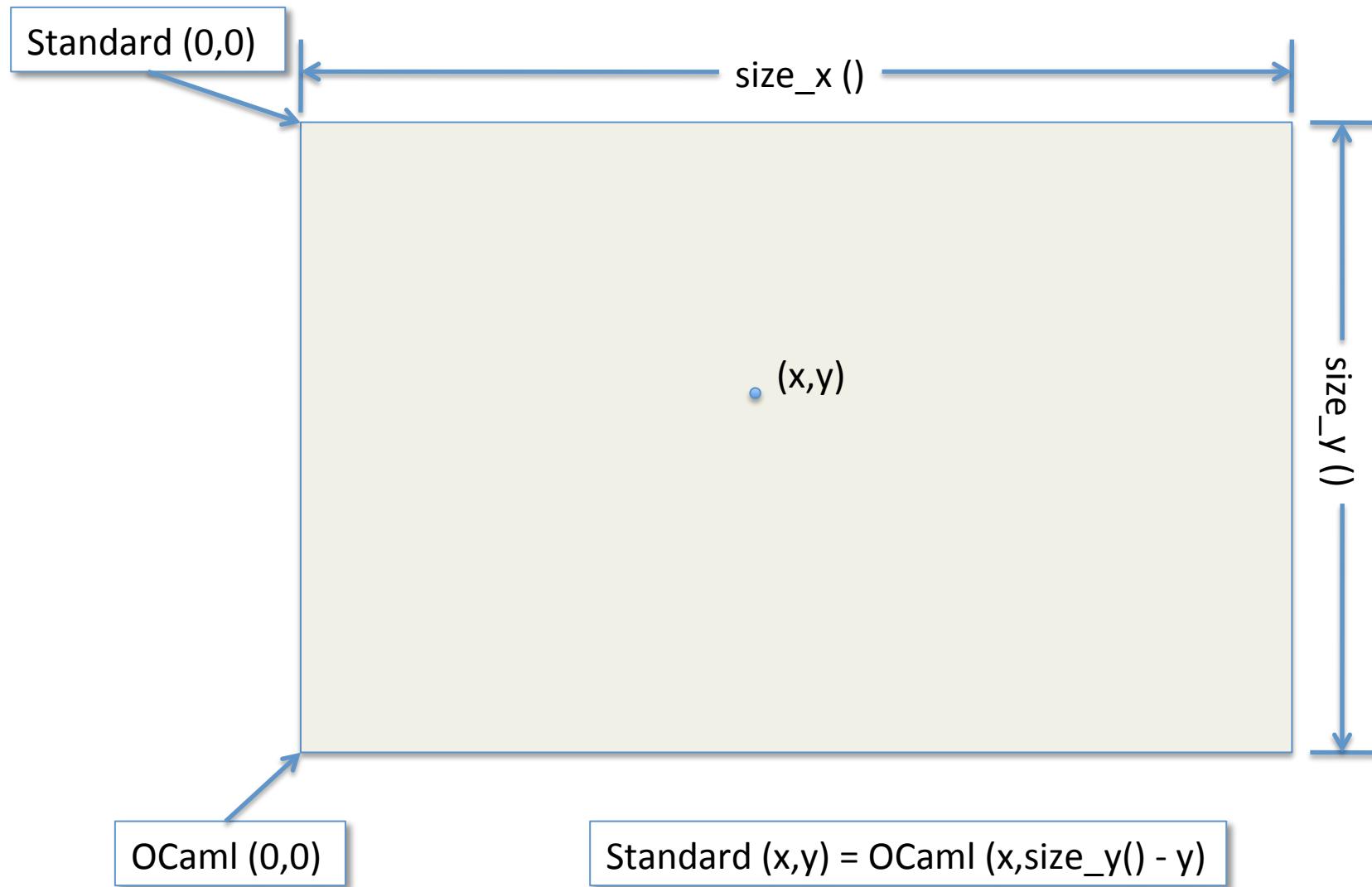
```
let top = Gctx.top_level
```



```
draw_string top (0,10) "CIS 120";  
draw_string nctx (0,10) "CIS 120"
```

```
repaint = fun g -> draw_rect g (0,0) (20,20);  
                      draw_string g (0,10) "CIS 120"
```

OCaml vs. Standard Coordinates



Module Gctx

```
(** The main (abstract) type of graphics contexts. *)
type gctx

(** The top-level graphics context *)
val top_level : gctx

(** A widget-relative position *)
type position = int * int

(** Display text at the given (relative) position *)
val draw_string : gctx -> position -> string -> unit
(** Draw a line between the two specified positions *)
val draw_line : gctx -> position -> position -> unit

(** Produce a new gctx shifted by (dx,dy) *)
val translate : gctx -> int * int -> gctx
(** Produce a new gctx with a different pen color *)
val with_color : gctx -> color -> gctx
```

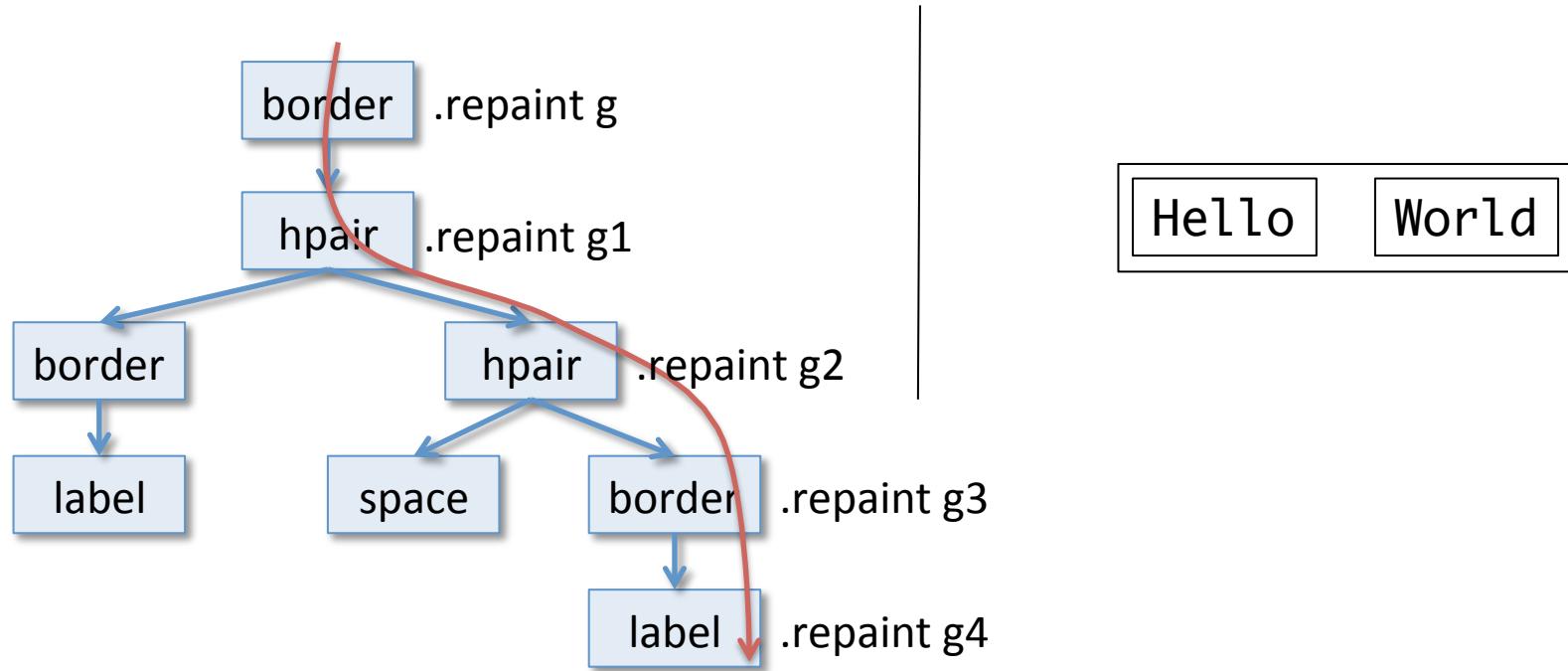
Event loop with Gctx

- Main loop of any GUI application:

```
let run (w:widget) : unit
  let g = Graphics.top in      ...create the initial gctx...
  Graphics.loop [Graphics.Mouse_motion; Graphics.Button_down;
                 Graphics.Button_up; Graphics.Key_pressed]
  (fun status ->
    clear_graph ();
    w.repaint g ();      ...repaint relative to g...
    ...inform w about the event so widgets can react to it...
  )
```

Drawing containers using graphics contexts

Container widgets propagate repaint commands to their children, with appropriately modified graphics contexts:



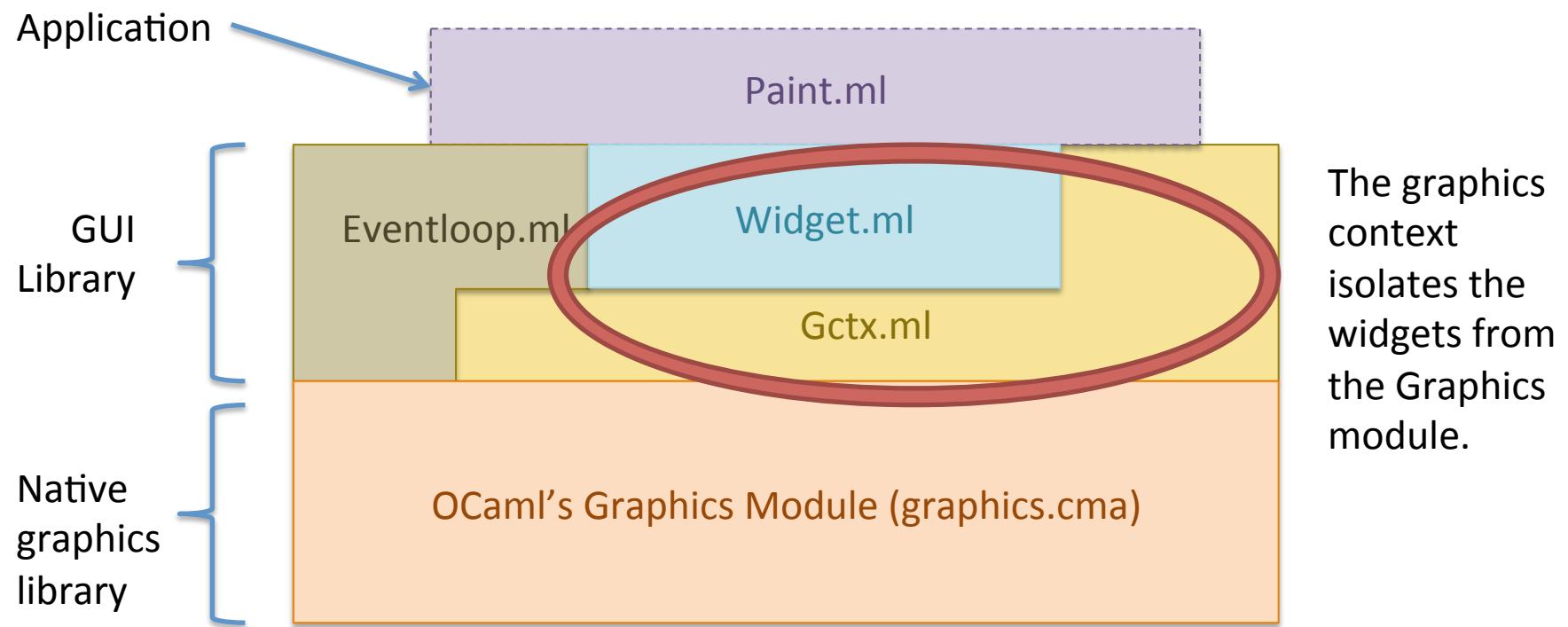
Module: Widgets

Building blocks of GUI applications

see simpleWidget.ml

Challenge: Widget Layout

- Widgets are “things drawn on the screen”. How to make them location independent?
- Idea: Use a *graphics context* to make drawing *relative* to the widget’s current position



Simple Widgets

simpleWidget.mli

```
(* An interface for simple GUI widgets *)
type widget = {
    repaint : Gctx.gctx -> unit;
    size    : unit -> (int * int)
}
val label  : string -> widget
val space  : int * int -> widget
val border : widget -> widget
val hpair  : widget -> widget -> widget
val canvas : int * int -> (Gctx.gctx -> unit) -> widget
```

- You can ask a simple widget to repaint itself.
- You can ask a simple widget to tell you its size.
- Both operations are relative to a graphics context

Widget Examples

simpleWidget.ml

```
(* A simple widget that puts some text on the screen *)
let label (s:string) : widget =
{
  repaint = (fun (g:Gctx.gctx) -> Gctx.draw_string g (0,0) s);
  size = (fun () -> Gctx.text_size s)
}
```

simpleWidget.ml

```
(* A "blank" area widget -- it just takes up space *)
let space ((x,y):int*int) : widget =
{
  repaint = (fun (_:Gctx.gctx) -> ());
  size = (fun () -> (x,y))
}
```

The canvas Widget

- Region of the screen that can be drawn upon
- Has a fixed width and height
- Parameterized by a repaint method
 - Use the Gctx drawing routines to draw on the canvas

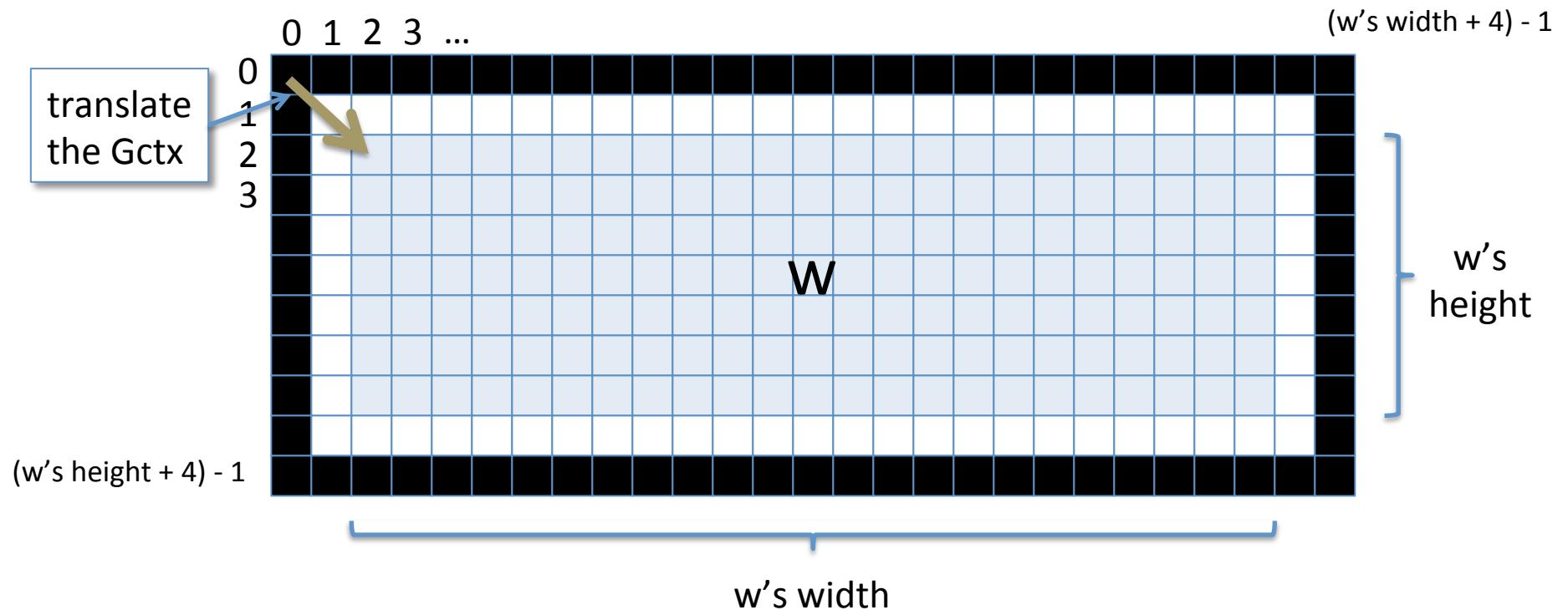
simpleWidget.ml

```
let canvas ((w,h):int*int) (repaint: Gctx.gctx -> unit) : widget =
{
  repaint = repaint;
  size = (fun () -> (w,h))
}
```

Nested Widgets

Containers and Composition

The Border Widget Container



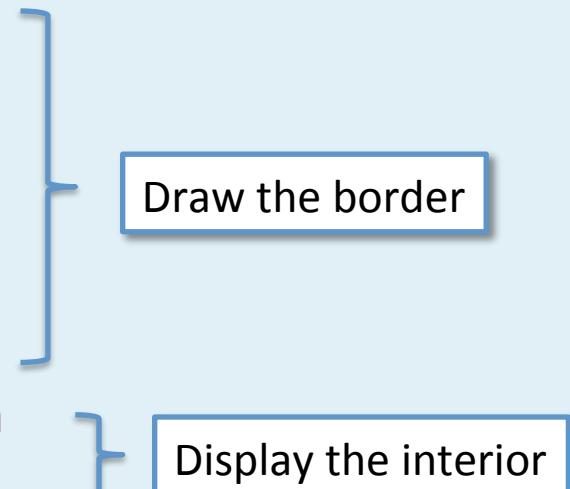
- `let b = border w`
- Draws a one-pixel wide border around contained widget W
- b's size is slightly larger than w's (+4 pixels in each dimension)
- b's repaint method must call w's repaint method
- When b asks w to repaint, b must *translate* the Gctx.t to (2,2) to account for the displacement of w from b's origin

The Border Widget

simpleWidget.ml

```
let border (w:widget):widget =
{
  repaint = (fun (g:Gctx.gctx) ->
    let (width,height) = w.size () in
    let x = width + 3 in
    let y = height + 3 in
    Gctx.draw_line g (0,0) (x,0);
    Gctx.draw_line g (0,0) (0,y);
    Gctx.draw_line g (x,0) (x,y);
    Gctx.draw_line g (0,y) (x,y);
    let gw = Gctx.translate g (2,2) in
    w.repaint gw);

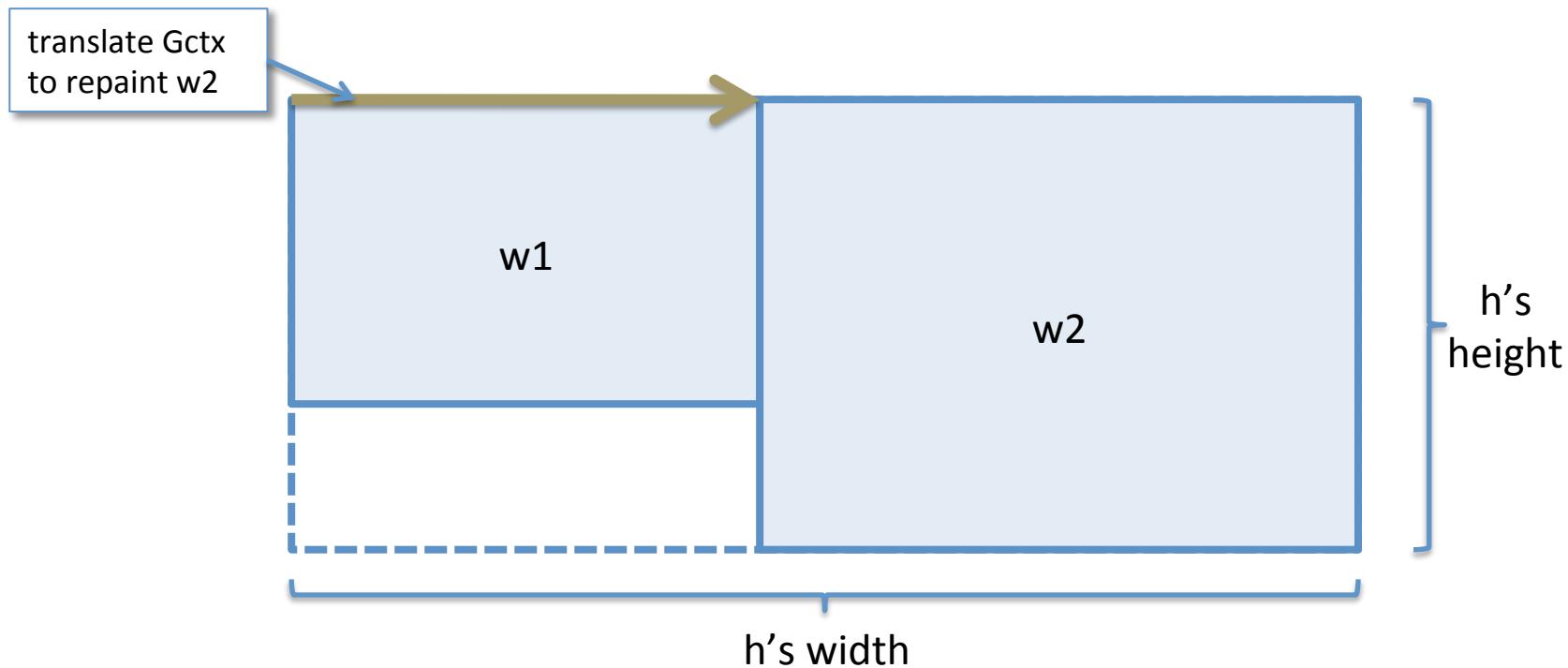
  size = (fun () ->
    let (width,height) = w.size () in
    (width+4, height+4))
}
```



Draw the border

Display the interior

The hpair Widget Container



- `let h = hpair w1 w2`
- Creates a horizontally adjacent pair of widgets
- Aligns them by their top edges
 - Must translate the Gctx when repainting w2
- Size is the *sum* of their widths and *max* of their heights

The hpair Widget

simpleWidget.ml

```
let hpair (w1: widget) (w2: widget) : widget =
{
  repaint = (fun (g: Gctx.gctx) ->
    let (x1, _) = w1.size () in begin
      w1.repaint g;
      w2.repaint (Gctx.translate g (x1, 0))
      (* Note translation of the Gctx *)
    end);

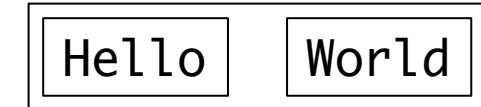
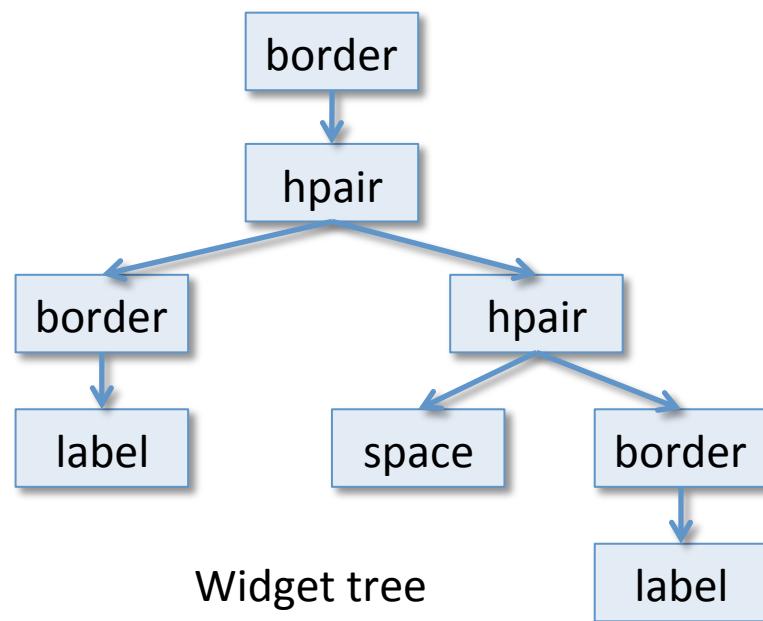
  size = (fun () ->
    let (x1, y1) = w1.size () in
    let (x2, y2) = w2.size () in
    (x1 + x2, max y1 y2))
}
```

Translate the Gctx
to shift w2's position
relative to widget-local
origin.

Widget Hierarchy Pictorially

```
(* Create some simple label widgets *)
let l1 = label "Hello"
let l2 = label "World"
(* Compose them horizontally, adding some borders *)
let h = border (hpair (border l1)
                  (hpair (space (10,10)) (border l2)))
```

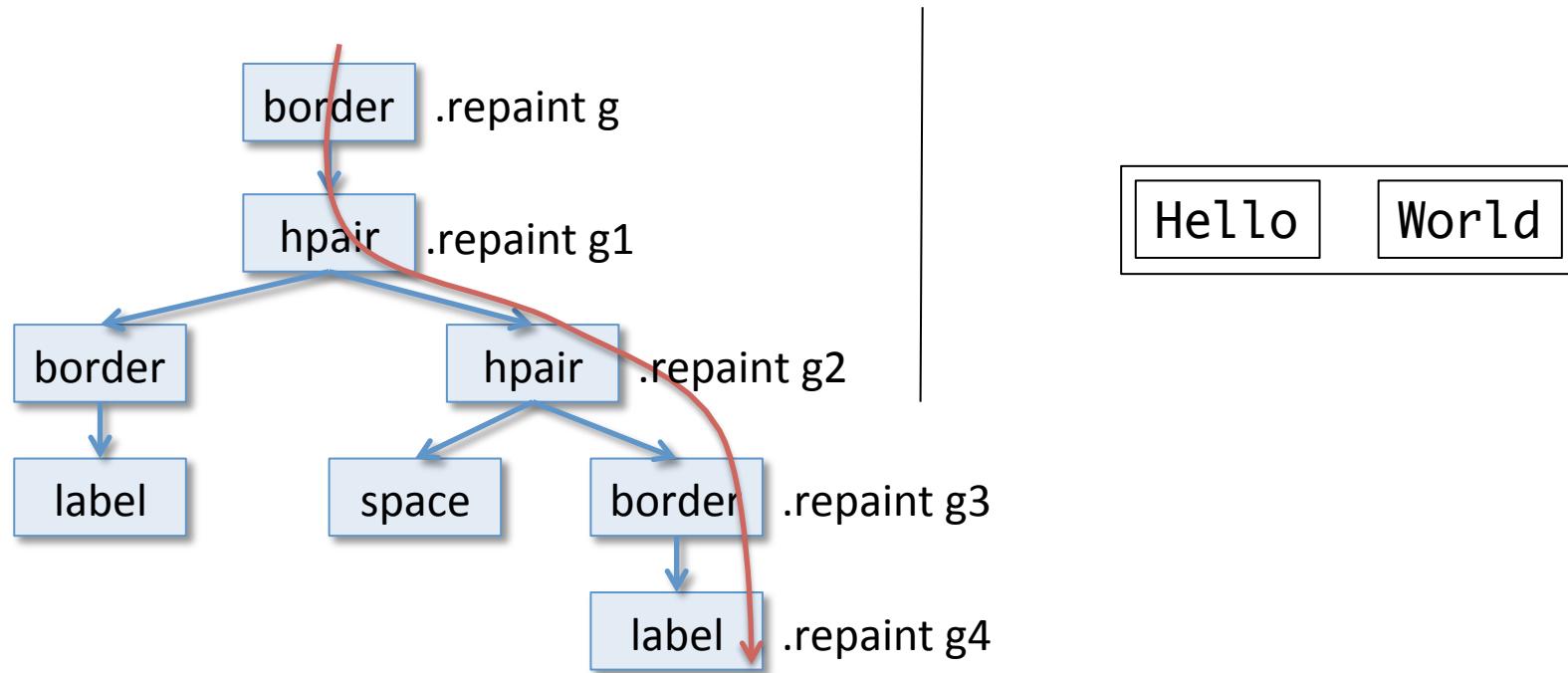
swdemo.ml



On the screen

Drawing: Containers

Container widgets propagate repaint commands to their children:

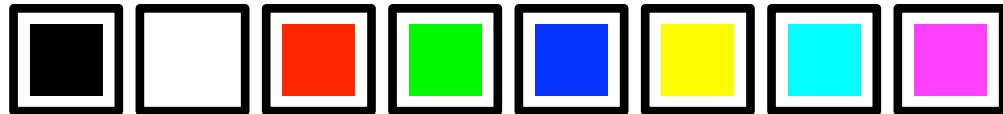


Widget tree

```
g1 = Gctx.translate g (2,2)
g2 = Gctx.translate g1 (hello_width,0)
g3 = Gctx.translate g2 (space_width,0)
g4 = Gctx.translate g3 (2,2)
```

On the screen

Container Widgets for layout



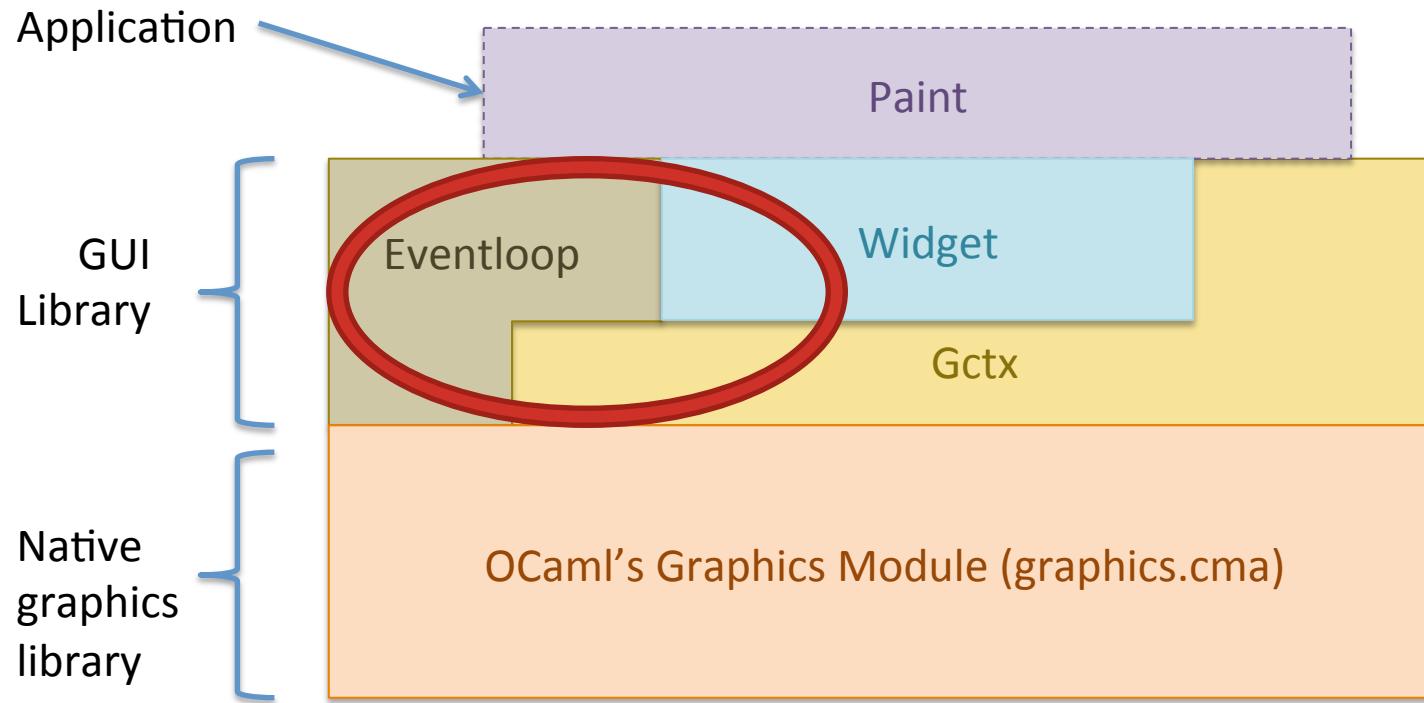
```
let color_toolbar : widget = hlist
  [ color_button black; spacer;
    color_button white; spacer;
    color_button red; spacer;
    color_button green; spacer;
    color_button blue; spacer;
    color_button yellow; spacer;
    color_button cyan; spacer;
    color_button magenta]
```

hlist is a container widget.
It takes a list of widgets and
turns them into a single one
by laying them out
horizontally (using hpair).

paint.ml

Events and Event Handling

Project Architecture

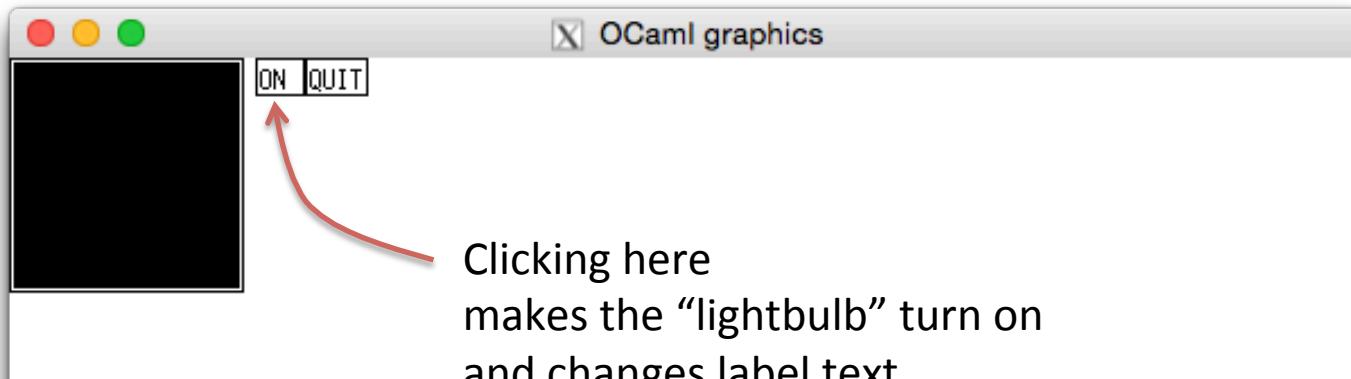


Goal of the GUI library: provide a consistent layer of abstraction *between* the application (Paint) and the Graphics module.

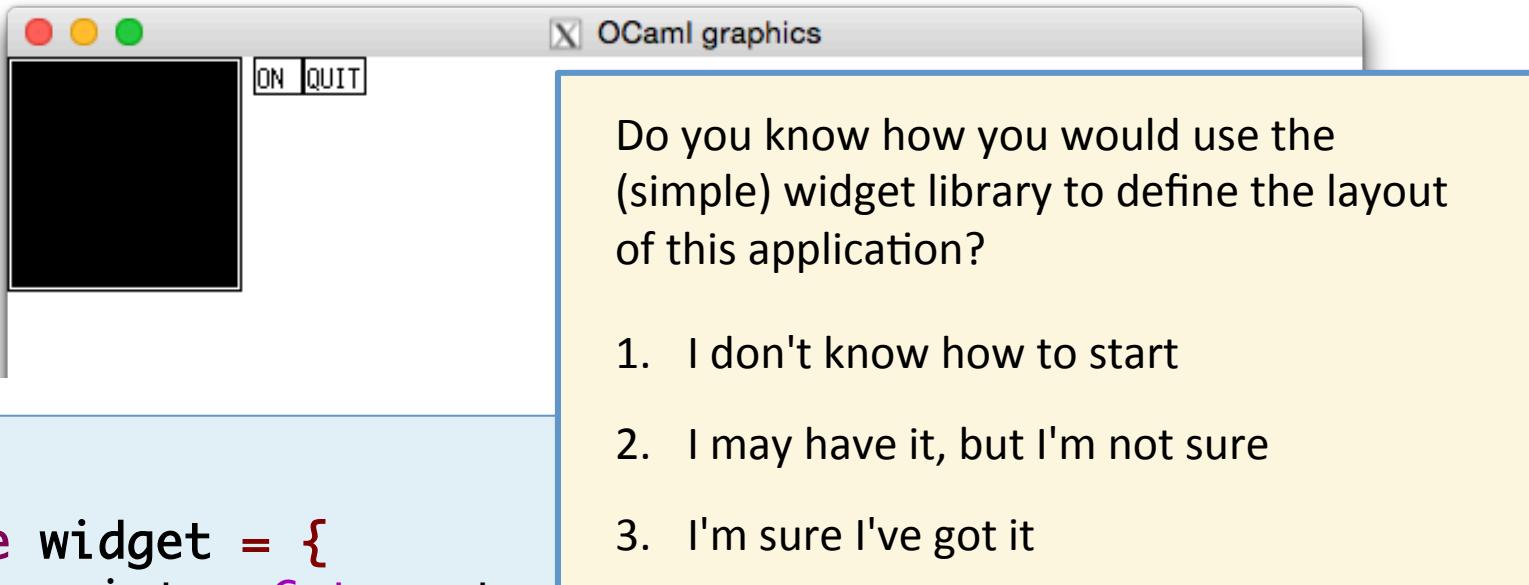
Demo: onoff.ml

Reacting to events

lightbulb demo

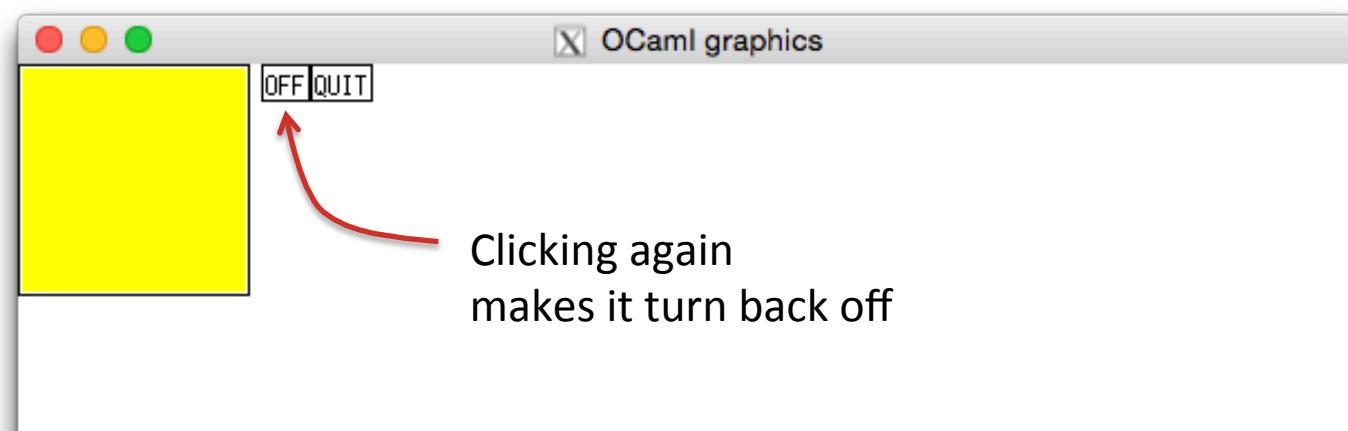
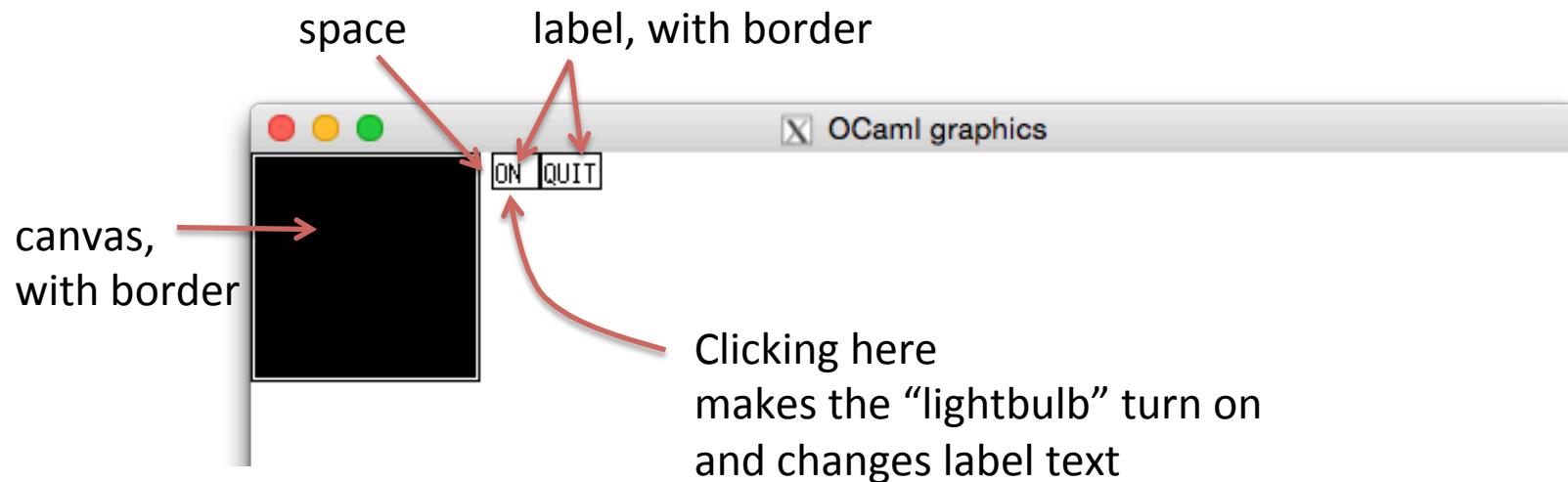


lightbulb demo



```
type widget = {  
    repaint : Gctx.gctx -> unit,  
    size     : unit -> (int * int)  
}  
val label  : string -> widget  
val space : int * int -> widget  
val border: widget -> widget  
val hpair : widget -> widget -> widget  
val canvas: int * int -> (Gctx.gctx -> unit) -> widget
```

lightbulb demo



User Interactions

- Problem: When a user moves the mouse, clicks the button, or presses a key, the application should react. How?

swdemo.ml

```
let run (w:widget) : unit =
  Gctx.open_graphics (); (* open graphics window *)
  let g = Gctx.top_level in
  w.repaint g; (* repaint the widget once *)
  Graphics.synchronize (); (* force window update *)
  ignore (Graphics.read_key ()); (* wait for a keypress *)
```

Handling Events

- Main loop of any GUI application:

eventloop.ml

```
let run (w:widget) : unit =
  let g = Graphics.top in      ...create the initial gctx...
  Graphics.loop [Graphics.Mouse_motion; Graphics.Button_down;
                 Graphics.Button_up; Graphics.Key_pressed]
  (fun status ->
    clear_graph ();
    w.repaint g ();      ...repaint relative to g...
    begin match event_of_status status with
    | None -> ()          ...spurious status update, do nothing...
    | Some e -> w.handle g e  ...let widget handle the event...
    end
  )
```

Events

gcxt.mli

```
type event

val wait_for_event : unit -> event

type event_type =
| KeyPress of char (* User pressed a key *)
| MouseDown      (* Mouse Button pressed, no movement *)
| MouseUp        (* Mouse button released, no movement *)
| MouseMove      (* Mouse moved with button up *)
| MouseDrag       (* Mouse moved with button down *)

val event_type : event -> event_type
val event_pos   : event -> gctx -> position
```

*The graphics context translates the location
of the event to widget-local coordinates*

Reactive Widgets

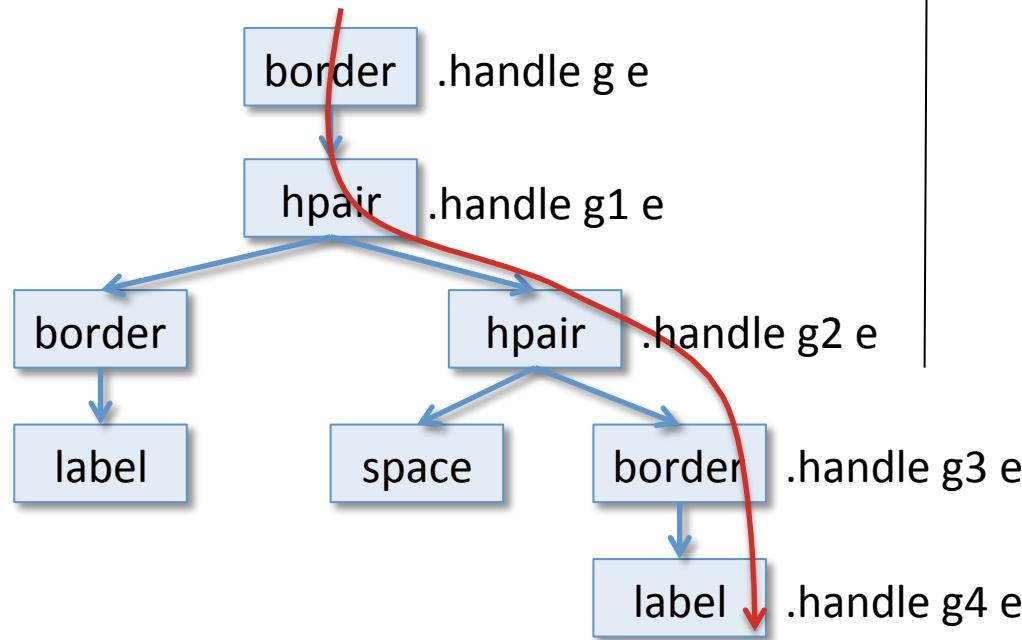
widget.mli

```
type t = {
  repaint : Gctx.gctx -> unit;
  size    : unit -> Gctx.dimension;
  handle  : Gctx.gctx -> Gctx.event -> unit (* NEW! *)
}
```

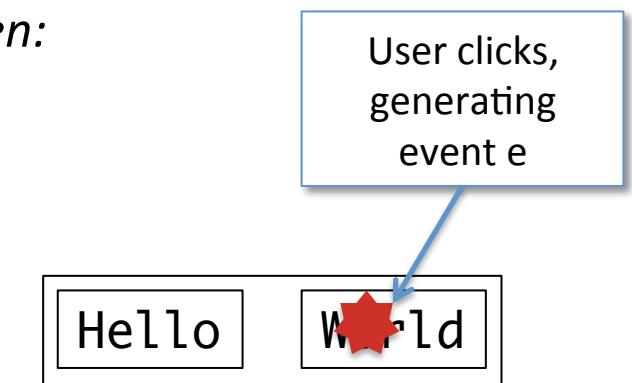
- Widgets now have a “method” for handling events
 - The eventloop waits for an event and then gives it to the root widget
 - The widgets forward the event down the tree, according to the position of the event

Event-handling: Containers

Container widgets propagate events to their children:



Widget tree



On the screen

Routing events through container widgets

Event Handling: Routing

- When a container widget handles an event, it passes the event to the appropriate child
- The Gctx.gctx must be translated so that the child can interpret the event in its own local coordinates.

widget.ml

```
let border (w:widget):widget =
  { repaint = ...;
    size = ...;
    handle = (fun (g:Gctx.gctx) (e:Gctx.event) ->
      w.handle (Gctx.translate g (2,2)) e);
  }
```

Consider routing an event through an hpair widget constructed by:

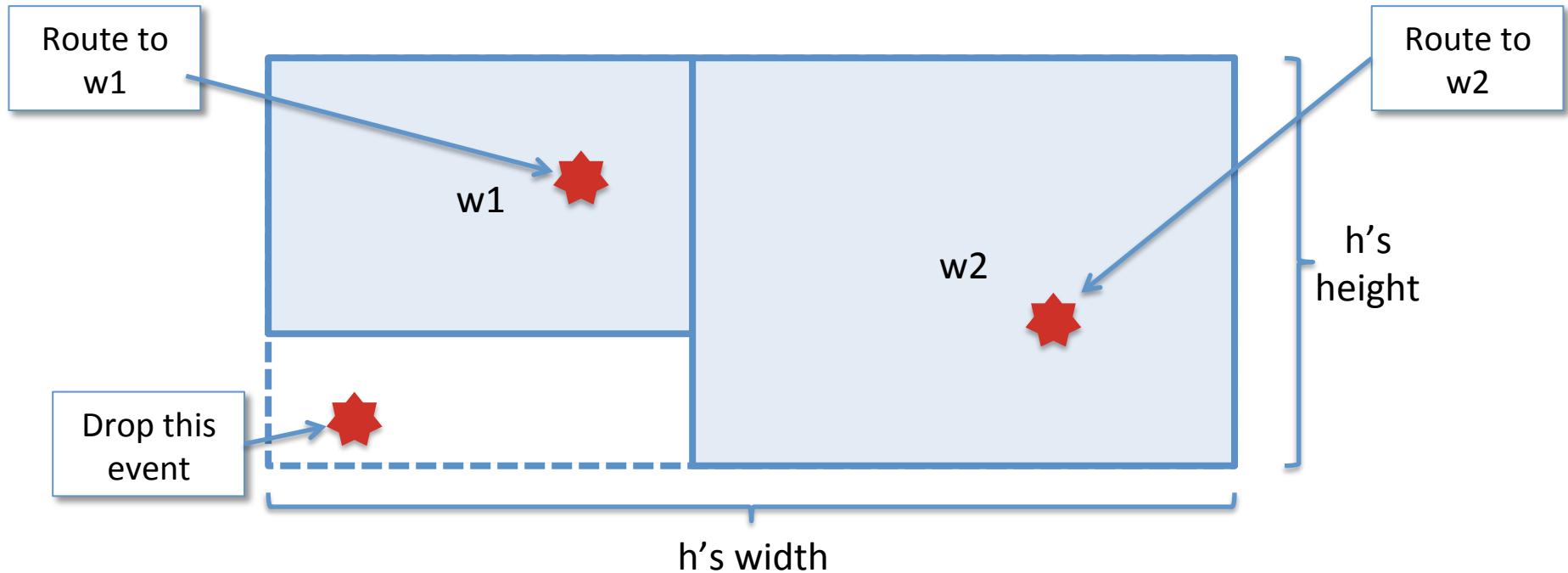
```
let hp = hpair w1 w2
```

The event will always be propagated either to w1 or w2.

1. True
2. False

Answer: False

Dropping Events in an HPair



- There are three cases for routing in an hpair.
- An event in the “empty area” should not be sent to either w1 or w2.

Routing events through hpair widgets

- The event handler of an hpair must check to see whether the event should be handled by the left or right widget.
 - Check the event's coordinates against the *size* of the left widget
 - If the event is within the left widget, let it handle the event
 - Otherwise check the event's coordinates against the right child's
 - If the right child gets the event, don't forget to translate its coordinates

```
handle =
(fun (g:Gctx.gctx) (e:Gctx.event) ->
  if event_within g e (w1.size g)
  then w1.handle g e
  else
    let g = (Gctx.translate g (fst (w1.size g), 0)) in
      if event_within g e (w2.size g)
      then w2.handle g e
      else ())
```

Stateful Widgets

How can widgets react to events?

A stateful label Widget

```
let label (s: string) : widget =
  let r = { contents = s } in
  { repaint =
    (fun (g: Gctx.gctx) ->
       Gctx.draw_string g (0,0) r.contents);
    handle = (fun _ _ -> ());
    size = (fun () ->
              Gctx.text_size r.contents)
  }
```

- The label “object” can make its string mutable. The “methods” can encapsulate that string.
- But what if the application wants to change this string in response to an event?

A stateful label Widget

widget.ml

```
type label_controller = { set_label: string -> unit }

let label (s: string) : widget * label_controller =
  let r = { contents = s } in
  ({ repaint =
    (fun (g: Gctx.gctx) ->
       Gctx.draw_string g (0,0) r.contents);
    handle = (fun _ _ -> ());
    size = (fun () ->
              Gctx.text_size r.contents)
  },
  { set_label = fun (s: string) -> r.contents <- s })
```

- A *controller* gives access to the shared state.
 - e.g. the `label_controller` object provides a way to set the label

Demo: onoff.ml

Changing the label on a button click

When a widget's handle function receives an event, it should also call functions from the Gctx library to update the view of the widget.

1. True
2. False
3. Not sure

Answer: False

Programming Languages and Techniques (CIS120)

Lecture 20

October 20, 2017

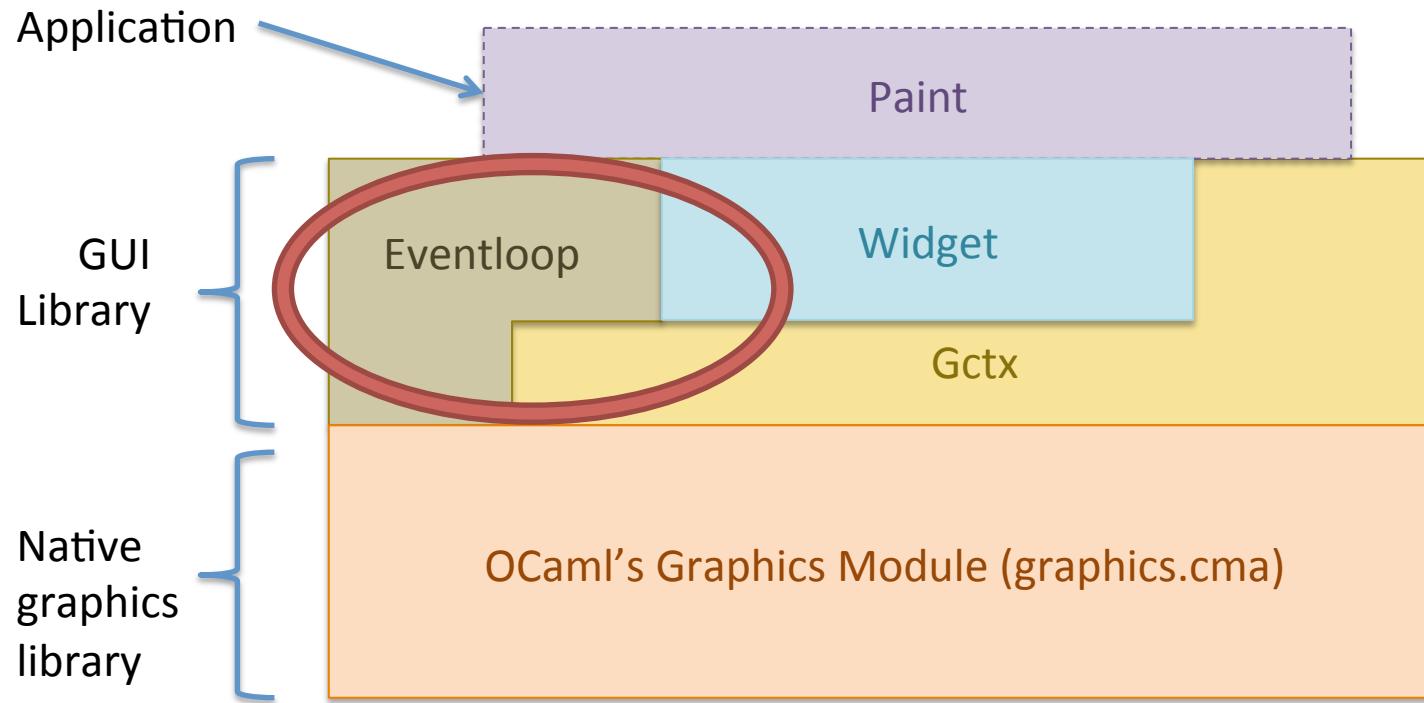
GUI: Events & State

Announcements

- Midterm 1 Grades & Solutions
 - will be released on Gradescope this afternoon
 - look for announcement on Piazza
- HW05: GUI programming
 - Due: **Tuesday, October 24** at 11:59:59pm
 - ***Graded (mostly) manually***
 - *Submission checks for compilation, few auto tests*
 - *Only LAST submission will be graded*
 - ***This project is challenging:***
 - ***IT IS NOW TOO LATE TO START EARLY!***

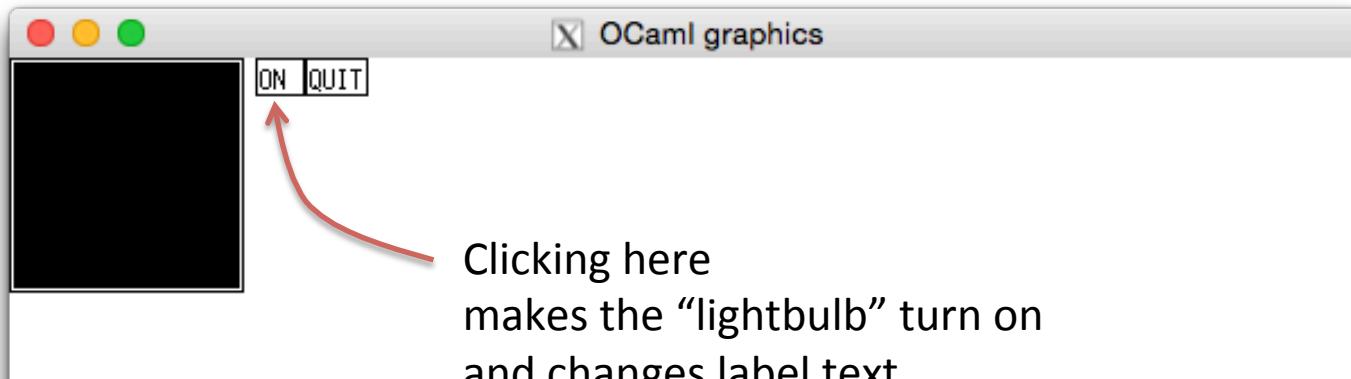
Events and Event Handling

Project Architecture



Goal of the GUI library: provide a consistent layer of abstraction *between* the application (Paint) and the Graphics module.

lightbulb demo



lightbulb demo

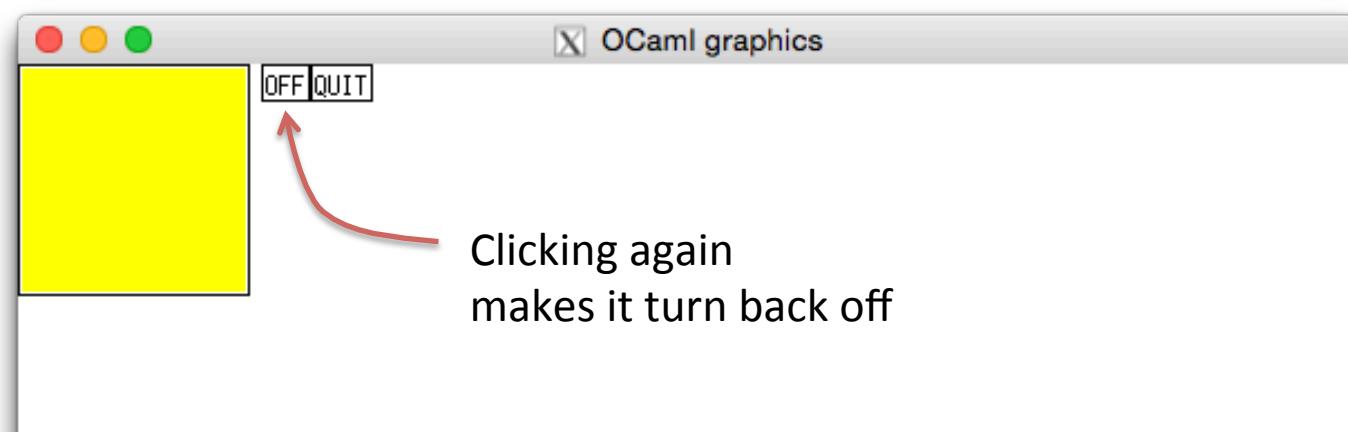
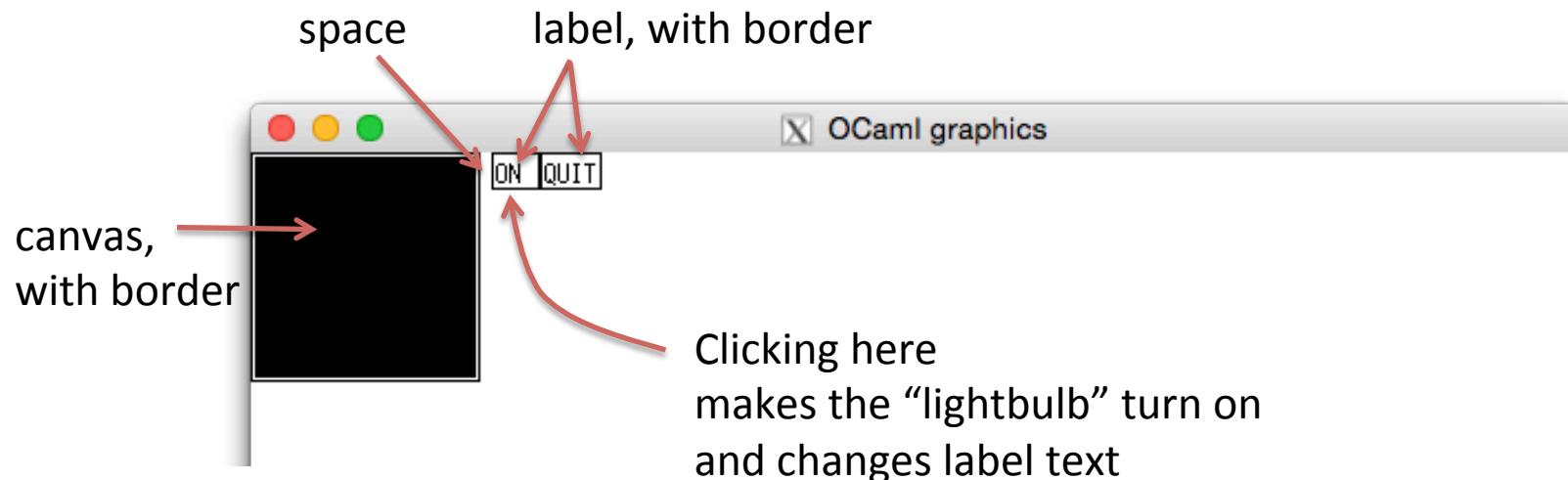


Do you know how you would use the (simple) widget library to define the layout of this application?

1. I don't know how to start
2. I may have it, but I'm not sure
3. I'm sure I've got it

```
type widget = {  
    repaint : Gctx.gctx -> unit,  
    size     : unit -> (int * int)  
}  
val label  : string -> widget  
val space : int * int -> widget  
val border: widget -> widget  
val hpair : widget -> widget -> widget  
val canvas: int * int -> (Gctx.gctx -> unit) -> widget
```

lightbulb demo



Handling Events

- Main loop of any GUI application:

eventloop.ml

```
let run (w:widget) : unit =
  let g = Graphics.top in      ...create the initial gctx...
  Graphics.loop [Graphics.Mouse_motion; Graphics.Button_down;
                 Graphics.Button_up; Graphics.Key_pressed]
  (fun status ->
    clear_graph ();
    w.repaint g ();      ...repaint relative to g...
    begin match event_of_status status with
    | None -> ()          ...spurious status update, do nothing...
    | Some e -> w.handle g e  ...let widget handle the event...
    end
  )
```

Events

gcxt.mli

```
type event

val wait_for_event : unit -> event

type event_type =
| KeyPress of char (* User pressed a key *)
| MouseDown      (* Mouse Button pressed, no movement *)
| MouseUp        (* Mouse button released, no movement *)
| MouseMove      (* Mouse moved with button up *)
| MouseDrag       (* Mouse moved with button down *)

val event_type : event -> event_type
val event_pos   : event -> gctx -> position
```

*The graphics context translates the location
of the event to widget-local coordinates*

Reactive Widgets

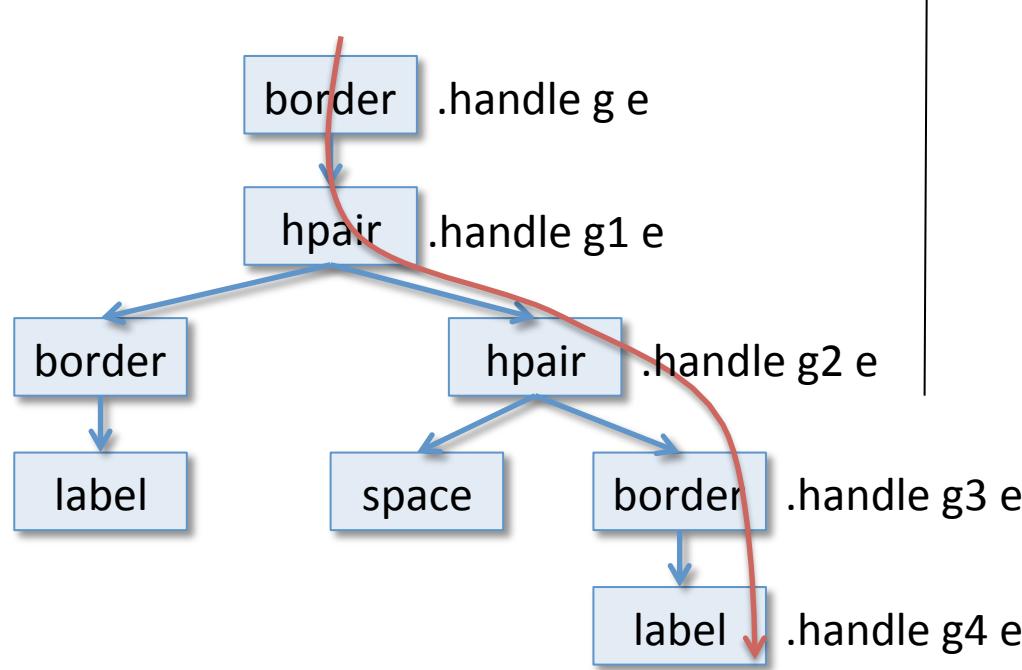
widget.mli

```
type t = {
  repaint : Gctx.gctx -> unit;
  size    : unit -> Gctx.dimension;
  handle  : Gctx.gctx -> Gctx.event -> unit (* NEW! *)
}
```

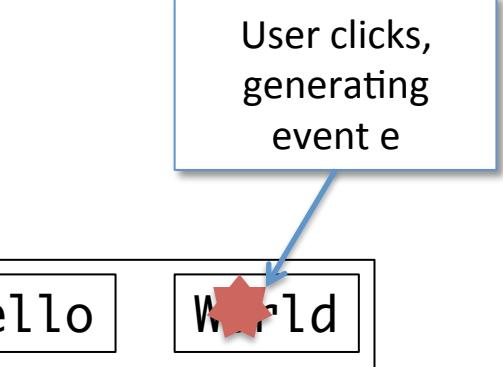
- Widgets now have a “method” for handling events
 - The eventloop waits for an event and then gives it to the root widget
 - The widgets forward the event down the tree, according to the position of the event

Event-handling: Containers

Container widgets propagate events to their children:



Widget tree



On the screen

Routing events through container widgets

Event Handling: Routing

- When a container widget handles an event, it passes the event to the appropriate child
- The Gctx.gctx must be translated so that the child can interpret the event in its own local coordinates.

widget.ml

```
let border (w:widget):widget =
  { repaint = ...;
    size = ...;
    handle = (fun (g:Gctx.gctx) (e:Gctx.event) ->
      w.handle (Gctx.translate g (2,2)) e);
  }
```

Consider routing an event through an hpair widget constructed by:

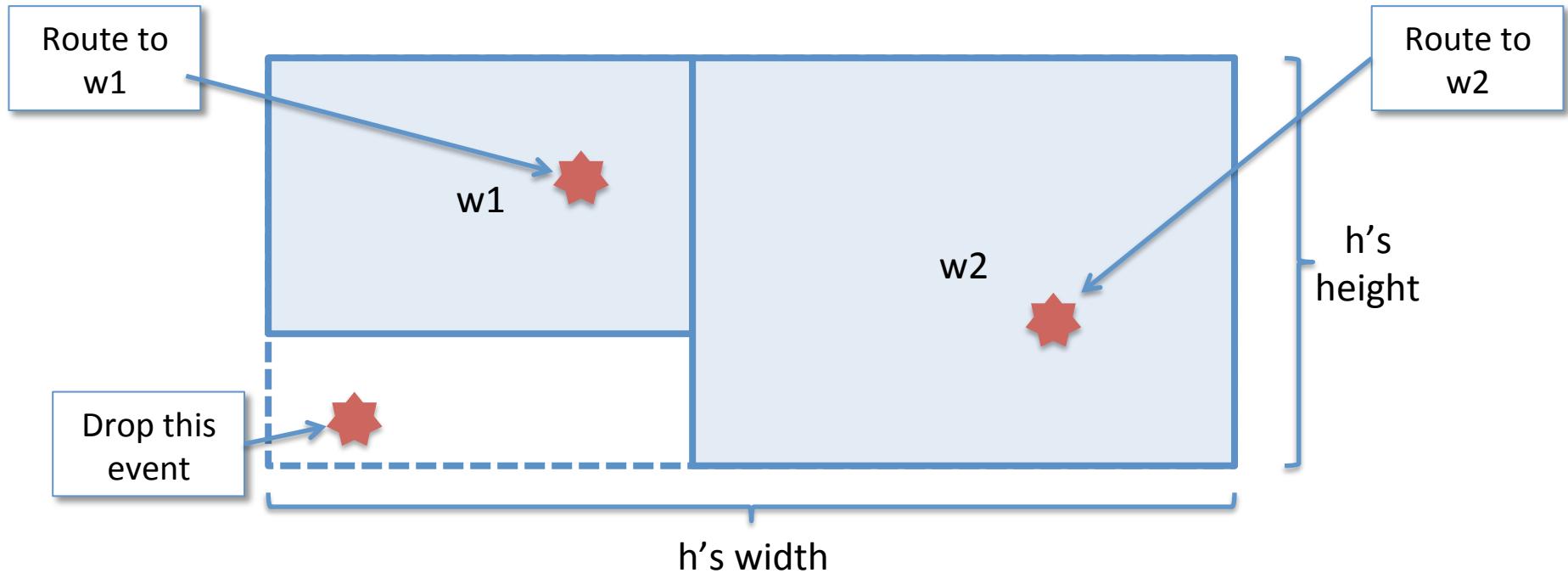
```
let hp = hpair w1 w2
```

The event will always be propagated either to w1 or w2.

1. True
2. False

Answer: False

Dropping Events in an HPair



- There are three cases for routing in an hpair.
- An event in the “empty area” should not be sent to either w1 or w2.

Routing events through hpair widgets

- The event handler of an hpair must check to see whether the event should be handled by the left or right widget.
 - Check the event's coordinates against the *size* of the left widget
 - If the event is within the left widget, let it handle the event
 - Otherwise check the event's coordinates against the right child's
 - If the right child gets the event, don't forget to translate its coordinates

```
handle =
(fun (g:Gctx.gctx) (e:Gctx.event) ->
  if event_within g e (w1.size ())
  then w1.handle g e
  else
    let g = (Gctx.translate g (fst (w1.size (), 0)) in
      if event_within g e (w2.size ())
      then w2.handle g e
      else ())
```

Stateful Widgets

How can widgets react to events?

A stateful label Widget

```
let label (s: string) : widget =
  let r = { contents = s } in
  { repaint = (fun (g: Gctx.gctx) ->
                Gctx.draw_string g (0,0) r.contents);
    handle   = (fun _ _ -> ());
    size     = (fun () -> Gctx.text_size r.contents)
  }
```

- The label “object” can make its string mutable. The “methods” can encapsulate that string.
- But what if the application wants to change this string in response to an event?

A stateful label Widget

widget.ml

```
type label_controller = { set_label: string -> unit }

let label (s: string) : widget * label_controller =
  let r = { contents = s } in
  ({ repaint = (fun (g: Gctx.gctx) ->
                 Gctx.draw_string g (0,0) r.contents);
    handle   = (fun _ _ -> ());
    size     = (fun () -> Gctx.text_size r.contents)
  }
  ,
  { set_label = fun (s: string) -> r.contents <- s })
```

- A *controller* gives access to the shared state.
 - e.g. the `label_controller` object provides a way to set the label

Event Listeners

How to react to events in a modular way?

Listeners

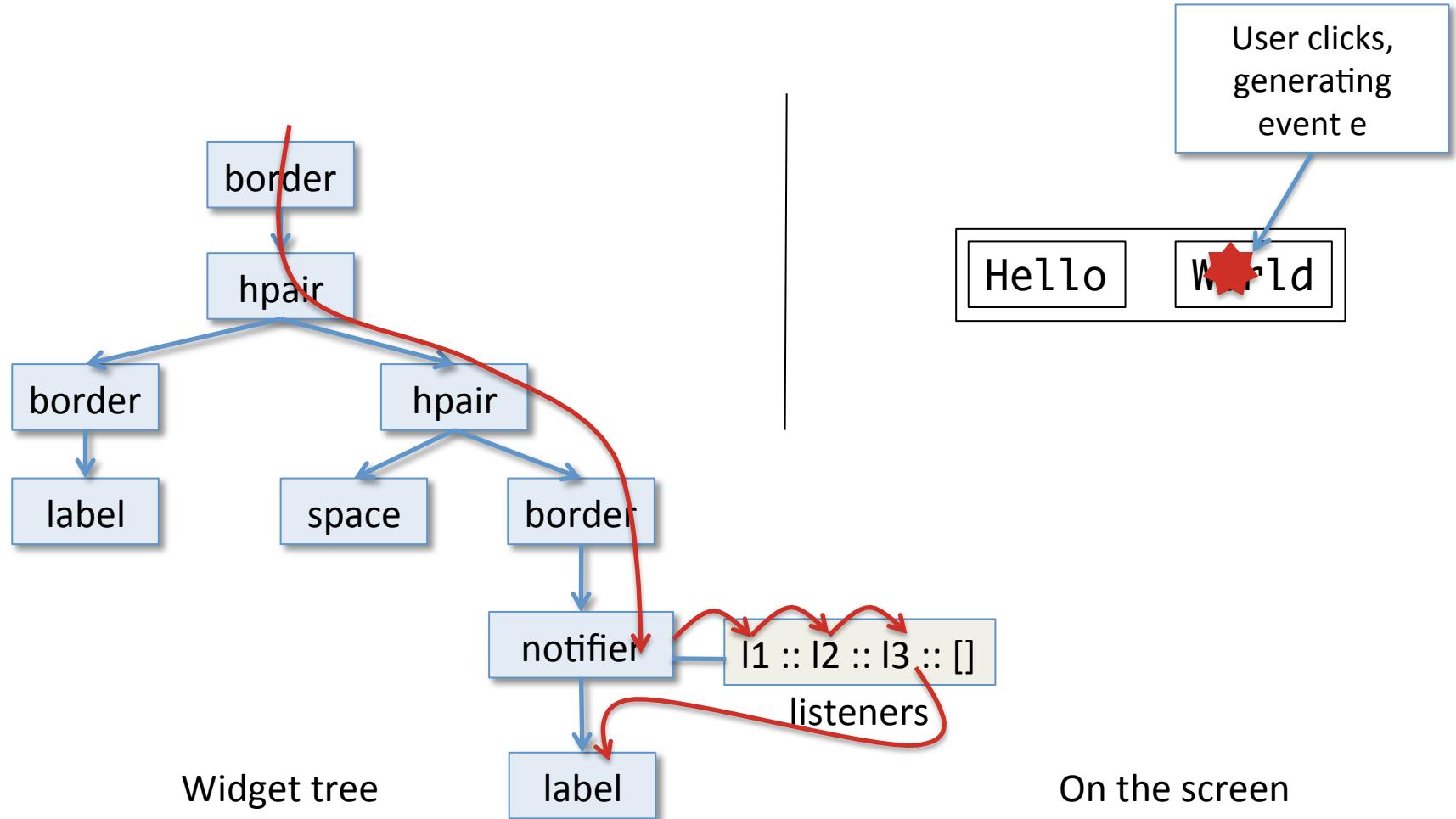
widget.ml

```
type event_listener = Gctx.gctx -> Gctx.event -> unit  
(* Performs an action upon receiving a mouse click. *)  
let mouseclick_listener (action: unit -> unit)  
    : event_listener =  
  fun (g:Gctx.gctx) (e: Gctx.event) ->  
    if Gctx.event_type e = Gctx.MouseDown  
    then action ()
```

Handling multiple event types

- Problem: *Widgets may want to react to many different sorts of events*
- Example: Button
 - button click: changes the state of the paint program and button label
 - mouse movement: tooltip? highlight?
 - key press: provide keyboard access to the button functionality?
- These reactions should be independent
 - Each sort of event handled by a different *event listener* (i.e. a first-class function)
 - Reactive widgets may have *several* listeners to handle a triggered event
 - Listeners react in sequence, all have a chance to see the event
- Solution: notifier

Listeners and Notifiers Pictorially



Notifiers

- A *notifier* is a container widget that adds event listeners to a node in the widget hierarchy
 - Note: this way of structuring event listeners is based on Java's Swing Library design (we use Swing terminology).
- *Event listeners* “eavesdrop” on the events flowing through the node
 - The event listeners are stored in a list
 - They react in order
 - Even if none of the listeners handle the event, then the event continues to the child widget
- List of event listeners can be updated by using a notifier_controller

Notifiers and Notifier Controllers

widget.ml

```
type notifier_controller =
  { add_listener : event_listener -> unit }

let notifier (w: widget) : widget * notifier_controller =
  let listeners = { contents = [] } in
  { repaint = w.repaint;
    handle =
      (fun (g: Gctx.gctx) (e: Gctx.event) ->
        List.iter (fun h -> h g e) listeners.contents;
        w.handle g e);
    size = w.size
  },
  { add_event_listener =
    fun (newl: event_listener) ->
      listeners.contents <- newl :: listeners.contents
  }
}
```

The notifier_controller allows new listeners to be added to the list.

Loop through the list of listeners, allowing each one to process the event. Then pass the event to the child.

Buttons (at last!)

widget.ml

```
(* A text button *)
let button (s: string) : widget
    (* label_controller
       * notifier_controller =
let (w, lc) = label s in
let (w', nc) = notifier w in
(w', lc, nc)
```

- A button widget is just a label wrapped in a notifier
- Add a mouseclick_listener to the button using the notifier_controller
- (For aesthetic purposes, you can put a border around the button widget.)

Demo: onoff.ml

Changing the label on a button click

Programming Languages and Techniques (CIS120)

Lecture 21

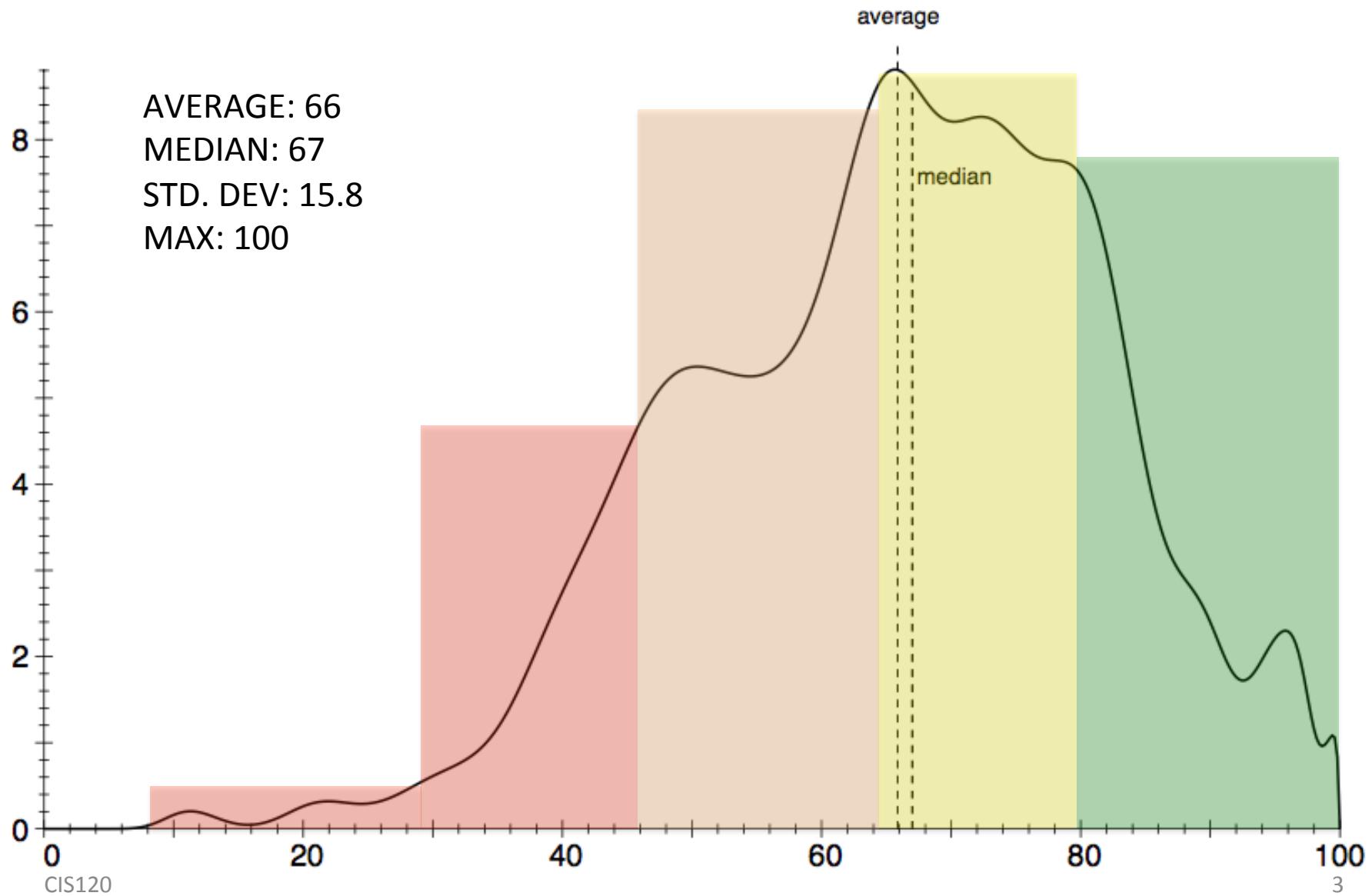
October 23, 2017

Transition to Java

Announcements

- HW05: GUI programming
 - Due: **TOMORROW** at 11:59:59pm
- HW06: Pennstagram
 - Available soon
 - Due: Tuesday, October 31st at 11:59:59pm
 - Java programming
 - Start Early to avoid Halloween conflict!

Midterm Exam Scores



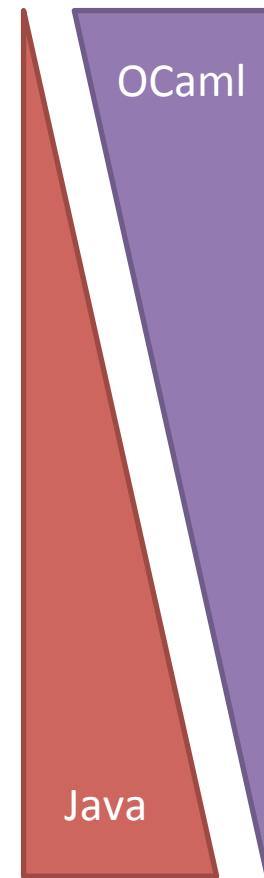
Goodbye OCaml...
...Hello Java!

Smoothing the transition

- Java Bootcamp Coming Soon
 - Look to Piazza for details
- General advice for the next few lectures: Ask questions, but don't stress about the details until you need them.
- Java resources:
 - Our lecture notes
 - CIS 110 website and textbook
 - Online Java textbook (<http://math.hws.edu/javanotes/>) linked from “CIS 120 Resources” on course website
 - Penn Library: Electronic access to “Java in a Nutshell” (and all other O'Reilly books)
 - Piazza

CIS 120 Overview

- Declarative (Functional) programming
 - *persistent* data structures
 - *recursion* is main control structure
 - frequent use of functions as data
- Imperative programming
 - *mutable* data structures (that can be modified “in place”)
 - *iteration* is main control structure
- Object-oriented (and reactive) programming
 - mutable data structures / iteration
 - heavy use of functions (objects) as data
 - pervasive “abstraction by default”



Java and OCaml together



Xavier Leroy, one of the principal designers of OCaml

Stephanie Weirich, Penn Prof. (CIS 120 co-developer, major contributor to Haskell)

Guy Steele, one of the principal designers of Java



Moral: Java and OCaml are not so far apart...

Recap: The Functional Style

- Core ideas:
 - immutable (persistent / declarative) data structures
 - recursion (and iteration) over tree structured data
 - functions as data
 - generic types for flexibility (i.e. ‘a list)
 - abstract types to preserve invariants (i.e. BSTs)
 - *simple model of computation (substitution)*
- Good for:
 - elegant descriptions of complex algorithms and/or data
 - small-scale compositional design
 - “symbol processing” programs (compilers, theorem provers, etc.)
 - parallelism, concurrency, and distribution

Functional programming



- Immutable lists primitive, tail recursion
- Datatypes and pattern matching for tree structured data
- First-class functions, transform and fold
- Generic types
- Abstract types through module signatures



- No primitive data structures, no tail recursion
- Trees must be encoded by objects, mutable by default
- First-class functions less common*
- Generic types
- Abstract types through public/private modifiers

*completely unsupported until recently (Java 8)

OCaml vs. Java for FP



```
type 'a tree =
| Empty
| Node of ('a tree) * 'a * ('a tree)

let is_empty (t:'a tree) : bool =
begin match t with
| Empty -> true
| _      -> false
end

let t : int tree = Node(Empty,3,Empty)
let ans : bool = is_empty t
```

OCaml provides a succinct, clean notation for working with generic, immutable, tree-structured data. Java requires a lot more "boilerplate".



```
interface Tree<A> {
    public boolean isEmpty();
}

class Empty<A> implements Tree<A> {
    public boolean isEmpty() {
        return true;
    }
}

class Node<A> implements Tree<A> {
    private final A v;
    private final Tree<A> lt;
    private final Tree<A> rt;

    Node(Tree<A> lt, A v, Tree<A> rt) {
        this.lt = lt; this.rt = rt; this.v = v;
    }

    public boolean isEmpty() {
        return false;
    }
}

class Program {
    public static void main(String[] args) {
        Tree<Integer> t =
            new Node<Integer>(new Empty<Integer>(),
                3, new Empty<Integer>());
        boolean ans = t.isEmpty();
    }
}
```

Other Popular Functional Languages



F#: Most similar to OCaml,
Shares libraries with C#



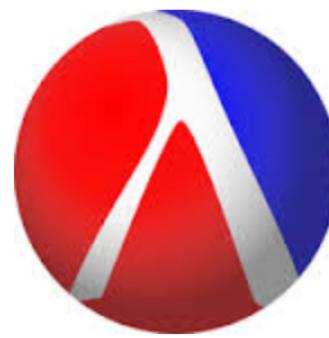
Haskell (CIS 552)
Purity + laziness



Swift
iOS programming



Clojure
Dynamically typed
Runs on JVM



Racket: LISP descendant;
widely used in education



Scala
Java / OCaml hybrid

Recap: The imperative style

- Core ideas:
 - computation as change of state over time
 - distinction between primitive and reference values
 - aliasing
 - linked data-structures and iteration control structure
 - generic types for flexibility (i.e. ‘a queue)
 - abstract types to preserve invariants (i.e. queue invariant)
 - *Abstract Stack Machine model of computation*
- Good for:
 - numerical simulations
 - implicit coordination between components (queues, GUI)

Imperative programming



- No null. Partiality must be made explicit with **options**.
- Code is an **expression** that has a value. Sometimes computing that value has other effects.
- References are **immutable** by default, must be explicitly declared to be mutable



- Most types have a **null** element. Partial functions can return **null**.
- Code is a sequence of **statements** that have effects, sometimes using expressions to compute values.
- References are **mutable** by default, must be explicitly declared to be constant

Explicit vs. Implicit Partiality



OCaml identifiers

- Cannot be changed once created; only mutable fields can change

```
type 'a ref = { mutable contents: 'a }
let x = { contents = counter () }
;; x.contents <- counter ()
```

- Cannot be null, must use options

```
let y = { contents = Some (counter ()) }
;; y.contents <- None
```

- Accessing the value requires pattern matching

```
;; match y.contents with
| None -> failwith "NPE"
| Some c -> c.inc ()
```



Java variables

- Can be assigned to after initialization

```
Counter x = new Counter ();
x = new Counter ();
```

- Can always be null

```
Counter y = new Counter ();
y = null;
```

- Check for null is implicit whenever a variable is used

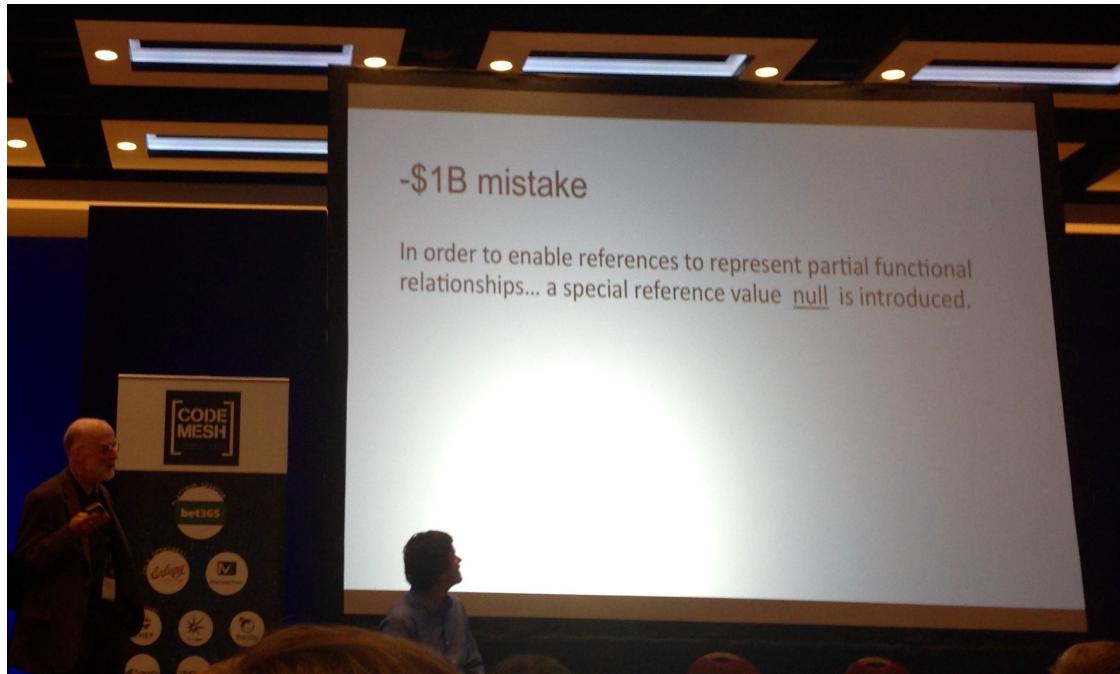
```
y.inc();
```

- If null is used as if it were an object (i.e. for a method call) then a **NullPointerException** occurs

The Billion Dollar Mistake

"I call it my billion-dollar mistake. It was the invention of the null reference in 1965. ... This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years. "

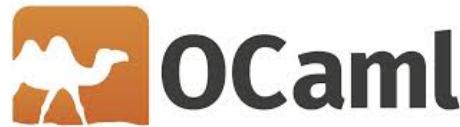
Sir Tony Hoare, QCon, London 2009



Java Core Language

differences between OCaml and Java

Structure of a Program



- All code lives in (perhaps implicitly named) **modules**.
- Modules may contain multiple **type definitions**, **let-bound value declarations**, and top-level **expressions** that are executed in the order they are encountered.



- All code lives in explicitly named **classes**.
- Classes are themselves types.
- Classes contain **field declarations** and **method definitions**.
- There is a single "entry point" of the program where it starts running, which must be a method called `main`.

Expressions vs. Statements

- OCaml is an *expression language*
 - Every program phrase is an expression (and returns a value)
 - The special value () of type unit is used as the result of expressions that are evaluated only for their side effects
 - Semicolon is an *operator* that combines two expressions (where the left-hand one returns type unit)
- Java is a *statement language*
 - Two sorts of program phrases: expressions (which compute values) and statements (which don't)
 - Statements are *terminated* by semicolons
 - Any expression can be used as a statement (but not vice-versa)



Types

- As in OCaml, every Java *expression* has a type
- The type describes the value that an expression computes

| Expression form | Example | Type |
|--------------------|----------------|-------------------------------------|
| Variable reference | x | Declared type of variable |
| Object creation | new Counter () | Class of the object |
| Method call | c.inc() | Return type of method |
| Equality test | x == y | boolean |
| Assignment | x = 5 | <i>don't use as an expression!!</i> |

Type System Organization

| | OCaml | Java |
|--|--|---|
| <i>primitive types</i> (values stored “directly” in the stack) | int, float, char, bool, ... | int, float, double, char, boolean, ... |
| structured types (a.k.a. <i>reference types</i> — values stored in the heap) | tuples, datatypes, records, functions, arrays <i>(objects encoded as records of functions)</i> | objects, arrays <i>(records, tuples, datatypes, strings, first-class functions are special cases of classes)</i> |
| generics | 'a list | List<A> |
| abstract types | module types (signatures) | interfaces public/private modifiers |

Arithmetic & Logical Operators

| OCaml | Java | |
|--------------|--------------|-------------------------------------|
| =, == | == | equality test |
| <>, != | != | inequality |
| >, >=, <, <= | >, >=, <, <= | comparisons |
| + | + | addition (and string concatenation) |
| - | - | subtraction (and unary minus) |
| * | * | multiplication |
| / | / | division |
| mod | % | remainder (modulus) |
| not | ! | logical “not” |
| && | && | logical “and” (short-circuiting) |
| | | logical “or” (short-circuiting) |

Java: Operator Overloading

- The *meaning* of an operator in Java is determined by the *types* of the values it operates on:
 - Integer division
 $4/3 \Rightarrow 1$
 - Floating point division
 $4.0/3.0 \Rightarrow 1.3333333333333333$
 - Automatic conversion from int to float
 $4/3.0 \Rightarrow 1.3333333333333333$
- Overloading is a general mechanism in Java
 - we'll see more of it later

Equality

- like OCaml, Java has two ways of testing reference types for equality:
 - “pointer equality”
 $o1 == o2$
 - “deep equality”
 $o1.equals(o2)$
- Normally, you should use `==` to compare primitive types and `.equals` to compare objects

every object provides an “equals” method that should “do the right thing” depending on the class of the object

Strings

- `String` is a *built in* Java class
- Strings are sequences of (unicode) characters
 - "" "Java" "3 Stooges" "富士山"
- + means String concatenation (overloaded)
 - "3" + " " + "Stooges" \Rightarrow "3 Stooges"
- Text in a String is immutable (like OCaml)
 - but variables that store strings are not
 - `String x = "OCaml";`
 - `String y = x;`
 - Can't do anything to x so that y changes
- The `.equals` method returns true when two strings contain the same sequence of characters

What is the value of ans at the end of this program?

```
String x = "CIS 120";
String z = "CIS 120";
boolean ans = x.equals(z);
```

1. true
2. false
3. NullPointerException

Answer: true

This is the preferred method of comparing strings!

What is the value of ans at the end of this program?

```
String x1 = "CIS ";
String x2 = "120";
String x = x1 + x2;
String z = "CIS 120";
boolean ans = (x == z);
```

1. true
2. false
3. NullPointerException

Answer: false

Even though x and z both contain the characters “CIS 120”,
they are stored in two different locations in the heap.

What is the value of ans at the end of this program?

```
String x = "CIS 120";
String z = "CIS 120";
boolean ans = (x == z);
```

1. true
2. false
3. NullPointerException

Answer: true(!)

Why? Since strings are immutable, two identical strings that are known when the program is compiled can be aliased by the compiler (to save space).

Moral

Always use `s1.equals(s2)` to compare strings!

Compare strings with respect to their content, not where they happen to be allocated in memory...

Object Oriented Programming

Recap: The OO Style

- Core ideas:
 - objects (state encapsulated with operations)
 - dynamic dispatch (“receiver” of method call determines behavior)
 - classes (“templates” for object creation)
 - subtyping (grouping object types by common functionality)
 - inheritance (creating new classes from existing ones)
- Good for:
 - GUIs!
 - complex software systems that include many different implementations of the same “interface” (set of operations) with different behaviors
 - Simulations
 - designs with an explicit correspondence between “objects” in the computer and things in the real world

OO programming



OCaml (part we've seen)

- Explicitly create objects using a record of higher order functions and hidden state
- Flexibility through ***composition***: objects can only implement one interface

```
type button =  
    widget *  
    label_controller *  
    notifier_controller
```



Java

(and C, C++, C#)

- Primitive notion of object creation (classes, with fields, methods and constructors)
- Flexibility through ***extension***: ***Subtyping*** allows related objects to share a common interface

```
class Button extends Widget {  
    /* Button is a subtype  
       of Widget */  
}
```

OO terminology

- **Object**: a structured collection of encapsulated *fields* (aka *instance variables*) and *methods*
- **Class**: a template for creating objects
- The class of an object specifies...
 - the types and initial values of its local state (fields)
 - the set of operations that can be performed on the object (methods)
 - one or more **constructors**: code that is executed when the object is created (optional)
- Every (Java) object is an *instance* of some class

Objects in Java

```
public class Counter {    class name  
    private int r;        instance variable  
  
    public Counter () {  
        r = 0;  
    }  
  
    public int inc () {  
        r = r + 1;  
        return r;  
    }  
  
    public int dec () {  
        r = r - 1;  
        return r;  
    }  
}
```

class declaration

constructor

methods

```
public class Main {
```

```
    public static void  
        main (String[] args) {    constructor  
            invocation
```

```
            Counter c = new Counter();
```

```
            System.out.println( c.inc() );
```

method call

Encapsulating local state

```
public class Counter {  
    private int r;  
  
    public Counter () {  
        r = 0;  
    }  
  
    public int inc () {  
        r = r + 1;  
        return r;  
    }  
  
    public int dec () {  
        r = r - 1;  
        return r;  
    }  
}
```

constructor and
methods can
refer to r

r is *private*

```
public class Main {  
    public static void main (String[] args) {  
  
        Counter c = new Counter();  
  
        System.out.println( c.inc() );  
    }  
}
```

other parts of the
program can only access
public members

method call

Encapsulating local state

- Visibility modifiers make the state local by controlling access
- Basically:
 - `public` : accessible from anywhere in the program
 - `private` : only accessible inside the class
- Design pattern — first cut:
 - Make *all* fields private
 - Make constructors and non-helper methods public

(Java offers a couple of other protection levels — “protected” and “package protected”. The details are not important at this point.)

Which programming style is the best?

1. Functional
2. Imperative
3. Object-oriented
4. To every thing there is a season...

What is the value of ans at the end of this program?

```
Counter x;  
x.inc();  
int ans = x.inc();
```

1. 1
2. 2
3. 3
4. Raises NullPointerException

```
public class Counter {  
  
    private int r;  
  
    public Counter () {  
        r = 0;  
    }  
  
    public int inc () {  
        r = r + 1;  
        return r;  
    }  
}
```

Answer: NullPointerException

What is the value of ans at the end of this program?

```
Counter x = new Counter();
x.inc();
Counter y = x;
y.inc();
int ans = x.inc();
```

1. 1
2. 2
3. 3
4. NullPointerException

```
public class Counter {
    private int r;
    public Counter () {
        r = 0;
    }
    public int inc () {
        r = r + 1;
        return r;
    }
}
```

Answer: 3

Interfaces

Working with objects abstractly

“Objects” in OCaml vs. Java

```
(* The type of “objects” *)
type point = {
    getX : unit -> int;
    getY : unit -> int;
    move : int*int -> unit;
}

(* Create an "object" with
   hidden state: *)
type position =
{ mutable x: int;
  mutable y: int; }

let new_point () : point =
let r = {x = 0; y=0} in {
    getX = (fun () -> r.x);
    getY = (fun () -> r.y);
    move = (fun (dx,dy) -
              r.x <- r.x + dx;
              r.y <- r.y + dy)
}
```

Type is separate
from the implementation

```
public class Point {

    private int x;
    private int y;

    public Point () {
        x = 0;
        y = 0;
    }
    public int getX () {
        return x;
    }
    public int getY () {
        return y;
    }
    public void move
        (int dx, int dy) {
        x = x + dx;
        x = x + dx;
    }
}
```

Class specifies both type and
implementation of object values

Interfaces

- Give a type for an object based on what it *does*, not on how it was constructed
- Describe a contract that objects must satisfy
- Example: Interface for objects that have a position and can be moved

```
public interface Displaceable {  
    public int getX();  
    public int getY();  
    public void move(int dx, int dy);  
}
```

No fields, no constructors, no method bodies!

Implementing the interface

- A class that implements an interface provides appropriate definitions for the methods specified in the interface
- That class fulfills the contract implicit in the interface

methods required to satisfy contract

```
public class Point implements Displaceable {  
    private int x, y;  
    public Point(int x0, int y0) {  
        x = x0;  
        y = y0;  
    }  
    public int getX() { return x; }  
    public int getY() { return y; }  
    public void move(int dx, int dy) {  
        x = x + dx;  
        y = y + dy;  
    }  
}
```



interfaces implemented

Another implementation

```
public class Circle implements Displaceable {  
    private Point center;  
    private int radius;  
    public Circle(Point initCenter, int initRadius) {  
        center = initCenter;  
        radius = initRadius;  
    }  
    public int getX() { return center.getX(); }  
    public int getY() { return center.getY(); }  
    public void move(int dx, int dy) {  
        center.move(dx, dy);  
    }  
}
```

Objects with different local state can satisfy the same interface

Delegation: move the circle by moving the center

Another implementation

```
class ColoredPoint implements Displaceable {  
    private Point p;  
    private Color c;  
    ColoredPoint (int x0, int y0, Color c0) {  
        p = new Point(x0,y0);  
        c = c0;  
    }  
    public void move(int dx, int dy) {  
        p.move(dx, dy);  
    }  
    public int getX() { return p.getX(); }  
    public int getY() { return p.getY(); }  
    public Color getColor() { return c; }  
}
```

Flexibility: Classes
may contain more
methods than
interface requires

Interfaces are types

- Can declare variables of interface type

```
void m(Displaceable d) { ... }
```

- Can call m with any Displaceable argument...

```
obj.m(new Point(3,4));  
obj.m(new ColoredPoint(1,2,Color.Black));
```

- ... but m can only operate on d according to the interface

```
d.move(-1,1);  
...  
... d.getX() ...      ⇒ 0.0  
... d.getY() ...      ⇒ 3.0
```

Using interface types

- Interface variables can refer (during execution) to objects of any class implementing the interface
- Point, Circle, and ColoredPoint are all *subtypes* of Displaceable

```
Displaceable d0, d1, d2;  
d0 = new Point(1, 2);  
d1 = new Circle(new Point(2,3), 1);  
d2 = new ColoredPoint(-1,1, red);  
d0.move(-2,0);  
d1.move(-2,0);  
d2.move(-2,0);  
...  
... d0.getX() ...     ⇒ -1.0  
... d1.getX() ...     ⇒  0.0  
... d2.getX() ...     ⇒ -3.0
```

Class that created the object value determines what move function is called.

Abstraction

- The interface gives us a single name for all the possible kinds of “moveable things.” This allows us to write code that manipulates arbitrary Displaceable objects, without caring whether it’s dealing with points or circles.

```
class DoStuff {  
    public void moveItALot (Displaceable s) {  
        s.move(3,3);  
        s.move(100,1000);  
        s.move(1000,234651);  
    }  
  
    public void dostuff () {  
        Displaceable s1 = new Point(5,5);  
        Displaceable s2 = new Circle(new Point(0,0),100);  
        moveItALot(s1);  
        moveItALot(s2);  
    }  
}
```

Recap

- **Object:** A collection of related *fields* (or *instance variables*)
- **Class:** A template for creating objects, specifying
 - types and initial values of fields
 - code for methods
 - optionally, a *constructor* that is run each time a new object is created from the class
- **Interface:** A “signature” for objects, describing a collection of methods that must be provided by classes that *implement* the interface
- **Object Type:** Either a class or an interface (meaning “this object was created from a class that implements this interface”)

“Datatypes” in Java

OCaml

```
type shape =
| Point of ...
| Circle of ...

let draw_shape (s:shape) =
begin match s with
| Point ... -> ...
| Circle ... -> ...
end
```

Java

```
interface Shape {
    public void draw();
}

class Point implements Shape {
    ...
    public void draw() {
        ...
    }
}

class Circle implements Shape {
    ...
    public void draw() {
        ...
    }
}
```

Multiple interfaces

- An interface represents a point of view
...but there can be multiple valid points of view
- Example: Geometric objects
 - All can move (all are Displaceable)
 - Some have Color (are Colored)

Colored interface

- Contract for objects that have a color
 - Circles and Points don't implement Colored
 - ColoredPoints do

```
public interface Colored {  
    public Color getColor();  
}
```

ColoredPoints

```
public class ColoredPoint
    implements Displaceable, Colored {
    Point center;
    private Color color;
    public Color getColor() {
        return color;
    }
    ...
}
```

Programming Languages and Techniques (CIS120)

Lecture 22
October 25, 2017

Java: Objects, Interfaces, Static Members
Chapters 19 & 20

Announcements

- Java Bootcamp
 - Tonight! October, 25th 6:00-8:00PM Towne 100
- HW06: Pennstagram
 - Available soon
 - Due: Tuesday, October 31st at 11:59:59pm
 - Java programming
 - Start Early to avoid Halloween conflict!

Object Oriented Programming

Recap: The OO Style

- Core ideas:
 - objects (state encapsulated with operations)
 - dynamic dispatch (“receiver” of method call determines behavior)
 - classes (“templates” for object creation)
 - subtyping (grouping object types by common functionality)
 - inheritance (creating new classes from existing ones)
- Good for:
 - GUIs!
 - complex software systems that include many different implementations of the same “interface” (set of operations) with different behaviors
 - Simulations
 - designs with an explicit correspondence between “objects” in the computer and things in the real world

OO programming



OCaml (part we've seen)

- Explicitly create objects using a record of higher order functions and hidden state
- Flexibility through ***composition***: objects can only implement one interface

```
type button =  
    widget *  
    label_controller *  
    notifier_controller
```



Java

(and C, C++, C#)

- Primitive notion of object creation (classes, with fields, methods and constructors)
- Flexibility through ***extension***: ***Subtyping*** allows related objects to share a common interface

```
class Button extends Widget {  
    /* Button is a subtype  
       of Widget */  
}
```

OO terminology

- **Object**: a structured collection of encapsulated *fields* (aka *instance variables*) and *methods*
- **Class**: a template for creating objects
- The class of an object specifies...
 - the types and initial values of its local state (*fields*)
 - the set of operations that can be performed on the object (*methods*)
 - one or more *constructors*: create new objects by (1) allocating heap space, and (2) running code to initialize the object (optional, but default provided)
- Every (Java) object is an *instance* of some class
 - Instances are created by invoking a constructor with the **new** keyword

Objects in Java

```
public class Counter {    class name  
    private int r;        instance variable  
  
    public Counter () {  
        r = 0;  
    }  
  
    public int inc () {  
        r = r + 1;  
        return r;  
    }  
  
    public int dec () {  
        r = r - 1;  
        return r;  
    }  
}
```

class declaration

constructor

methods

```
public class Main {
```

```
    public static void  
        main (String[] args) {    constructor  
            invocation
```

```
            Counter c = new Counter();
```

```
            System.out.println( c.inc() );
```

method call

Encapsulating local state

```
public class Counter {  
    private int r;  
  
    public Counter () {  
        r = 0;  
    }  
  
    public int inc () {  
        r = r + 1;  
        return r;  
    }  
  
    public int dec () {  
        r = r - 1;  
        return r;  
    }  
}
```

constructor and
methods can
refer to r

r is *private*

```
public class Main {  
    public static void main (String[] args) {  
  
        Counter c = new Counter();  
  
        System.out.println( c.inc() );  
    }  
}
```

other parts of the
program can only access
public members

method call

Encapsulating local state

- Visibility modifiers make the state local by controlling access
- Basically:
 - `public` : accessible from anywhere in the program
 - `private` : only accessible inside the class
- Design pattern — first cut:
 - Make *all* fields private
 - Make constructors and non-helper methods public

(Java offers a couple of other protection levels — “protected” and “package protected”. The details are not important at this point.)

What is the value of ans at the end of this program?

```
Counter x;  
x.inc();  
int ans = x.inc();
```

1. 1
2. 2
3. 3
4. Raises NullPointerException

```
public class Counter {  
  
    private int r;  
  
    public Counter () {  
        r = 0;  
    }  
  
    public int inc () {  
        r = r + 1;  
        return r;  
    }  
}
```

Answer: NullPointerException

What is the value of ans at the end of this program?

```
Counter x = new Counter();
x.inc();
Counter y = x;
y.inc();
int ans = x.inc();
```

1. 1
2. 2
3. 3
4. NullPointerException

```
public class Counter {
    private int r;
    public Counter () {
        r = 0;
    }
    public int inc () {
        r = r + 1;
        return r;
    }
}
```

Answer: 3

“Objects” in OCaml vs. Java

```
(* The type of “objects” *)
type point = {
    getX : unit -> int;
    getY : unit -> int;
    move : int*int -> unit;
}

(* Create an "object" with
   hidden state: *)
type position =
{ mutable x: int;
  mutable y: int; }

let new_point () : point =
let r = {x = 0; y=0} in {
    getX = (fun () -> r.x);
    getY = (fun () -> r.y);
    move = (fun (dx,dy) -
              r.x <- r.x + dx;
              r.y <- r.y + dy)
}
```

Type is separate
from the implementation

```
public class Point {

    private int x;
    private int y;

    public Point () {
        x = 0;
        y = 0;
    }
    public int getX () {
        return x;
    }
    public int getY () {
        return y;
    }
    public void move
        (int dx, int dy) {
        x = x + dx;
        x = x + dx;
    }
}
```

Class specifies both type and
implementation of object values

Which programming style is the best?

1. Functional
2. Imperative
3. Object-oriented
4. To every thing there is a season...

Interfaces

Working with objects abstractly

Interfaces

- Give a type for an object based on how it can be *used*, not on how it was *constructed*
- Describe a contract that objects must satisfy
- Example: Interface for objects that have a position and can be moved

```
public interface Displaceable {  
    public int getX();  
    public int getY();  
    public void move(int dx, int dy);  
}
```

No fields, no constructors, no method bodies!

Implementing the interface

- A class that *implements* an interface provides appropriate definitions for the methods specified in the interface
- The class fulfills the contract implicit in the interface

methods required to satisfy contract

```
public class Point implements Displaceable {  
    private int x, y;  
    public Point(int x0, int y0) {  
        x = x0;  
        y = y0;  
    }  
    public int getX() { return x; }  
    public int getY() { return y; }  
    public void move(int dx, int dy) {  
        x = x + dx;  
        y = y + dy;  
    }  
}
```



interfaces implemented

Another implementation

```
public class Circle implements Displaceable {  
    private Point center;  
    private int radius;  
    public Circle(Point initCenter, int initRadius) {  
        center = initCenter;  
        radius = initRadius;  
    }  
    public int getX() { return center.getX(); }  
    public int getY() { return center.getY(); }  
    public void move(int dx, int dy) {  
        center.move(dx, dy);  
    }  
}
```

Objects with different local state can satisfy the same interface

Delegation: move the circle by moving the center

Another implementation

```
class ColoredPoint implements Displaceable {  
    private Point p;  
    private Color c;  
    ColoredPoint (int x0, int y0, Color c0) {  
        p = new Point(x0,y0);  
        c = c0;  
    }  
    public void move(int dx, int dy) {  
        p.move(dx, dy);  
    }  
    public int getX() { return p.getX(); }  
    public int getY() { return p.getY(); }  
    public Color getColor() { return c; }  
}
```

Flexibility: Classes
may contain more
methods than
interface requires

Interfaces are types

- Can declare variables of interface type

```
void m(Displaceable d) { ... }
```

- Can call m with any Displaceable argument...

```
obj.m(new Point(3,4));  
obj.m(new ColoredPoint(1,2,Color.Black));
```

- ... but m can only operate on d according to the interface

```
d.move(-1,1);  
...  
... d.getX() ...      ⇒ 0  
... d.getY() ...      ⇒ 3
```

Using interface types

- Interface variables can refer *dynamically*, i.e. during execution, to objects of any class implementing the interface
- Point, Circle, and ColoredPoint are all *subtypes* of Displaceable

```
Displaceable d0, d1, d2;  
d0 = new Point(1, 2);  
d1 = new Circle(new Point(2,3), 1);  
d2 = new ColoredPoint(-1,1, red);  
d0.move(-2,0);  
d1.move(-2,0);  
d2.move(-2,0);  
...  
... d0.getX() ...     ⇒ -1.0  
... d1.getX() ...     ⇒  0.0  
... d2.getX() ...     ⇒ -3.0
```

The class that created the object value determines which move code is executed:
dynamic dispatch

Abstraction

- The interface gives us a single name for all the possible kinds of “moveable things.” This allows us to write code that manipulates arbitrary Displaceable objects, without caring whether it’s dealing with points or circles.

```
class DoStuff {  
    public void moveItALot (Displaceable s) {  
        s.move(3,3);  
        s.move(100,1000);  
        s.move(1000,234651);  
    }  
  
    public void dostuff () {  
        Displaceable s1 = new Point(5,5);  
        Displaceable s2 = new Circle(new Point(0,0),100);  
        moveItALot(s1);  
        moveItALot(s2);  
    }  
}
```

Recap

- **Object:** A collection of related *fields* (or *instance variables*)
- **Class:** A template for creating objects, specifying
 - types and initial values of fields
 - code for methods
 - optionally, a *constructor* that is run each time a new object is created from the class
- **Interface:** A “signature” for objects, describing a collection of methods that must be provided by classes that *implement* the interface
- **Object Type:** Either a class or an interface (meaning “this object was created from a class that implements this interface”)

Multiple interfaces

- An interface represents a point of view
...but there can be multiple valid points of view
- Example: Geometric objects
 - All can move (all are Displaceable)
 - Some have Color (are Colored)

Colored interface

- Contract for objects that have a color
 - Circles and Points don't implement Colored
 - ColoredPoints do

```
public interface Colored {  
    public Color getColor();  
}
```

ColoredPoints

```
public class ColoredPoint
    implements Displaceable, Colored {
    Point center;
    private Color color;
    public Color getColor() {
        return color;
    }
    ...
}
```

“Datatypes” in Java

OCaml

```
type shape =
| Point of ...
| Circle of ...

let draw_shape (s:shape) =
begin match s with
| Point ... -> ...
| Circle ... -> ...
end
```

Java

```
interface Shape {
    public void draw();
}

class Point implements Shape {
    ...
    public void draw() {
        ...
    }
}

class Circle implements Shape {
    ...
    public void draw() {
        ...
    }
}
```

Static Methods and Fields

functions and global state

Java Main Entry Point

```
class MainClass {  
    public static void main (String[] args) {  
        ...  
    }  
}
```

- Program starts running at `main`
 - `args` is an array of `Strings` (passed in from the command line)
 - must be `public`
 - returns `void` (i.e. is a command)
- What does *static* mean?

How familiar are you with the idea of "static" methods and fields?

1. I haven't heard of the idea of "static".
2. I've used "static" before without really understanding what it means
3. I have some familiarity with the difference between "static" and "dynamic"
4. I totally get it.

Static method example

```
public class Max {  
    public static int max (int x, int y) {  
        if (x > y) {  
            return x;  
        } else {  
            return y;  
        }  
    }  
}
```

```
public static int max3(int x, int y, int z) {  
    return max(max(x,y), z);  
}
```

Internally (within the same class), call with just the method name

main method must be static; it is invoked to start the program running

closest analogue of top-level functions in OCaml, but must be a member of some class

```
public class Main {  
    public static void main (String[] args) {  
        System.out.println(Max.max(3,4));  
        return;  
    }  
}
```

Externally, prefix with name of the class

mantra

Static == Decided at *Compile* Time
Dynamic == Decided at *Run* Time

Static vs. Dynamic Methods

- Static Methods are *independent* of object values
 - Similar to OCaml functions
 - Cannot refer to the local state of objects (fields or normal methods)
- Use static methods for:
 - Non-OO programming
 - Programming with primitive types: Math.sin(60), Integer.toString(3), Boolean.valueOf("true")
 - “public static void main”
- “Normal” methods are *dynamic*
 - Need access to the local state of the particular object on which they are invoked
 - We only know at *runtime* which method will get called

```
void moveTwice (Displaceable o) {  
    o.move (1,1); o.move(1,1);  
}
```

Method call examples

- Calling a (dynamic) method of an object (o) that returns a number:

```
x = o.m() + 5;
```

- Calling a static method of a class (C) that returns a number:

```
x = C.m() + 5;
```

- Calling a method that returns void:

Static

```
C.m();
```

Dynamic

```
o.m();
```

- Calling a static or dynamic method in a method of the same class:

Either

```
m();
```

Static

```
C.m();
```

Dynamic

```
this.m();
```

- Calling (dynamic) methods that return objects:

```
x = o.m().n();
x = o.m().n().x().y().z().a().b().c().d().e();
```

Which **static** method can we add to this class?

```
public class Counter {  
  
    private int r;  
  
    public Counter () {  
        r = 0;  
    }  
  
    public int inc () {  
        r = r + 1;  
        return r;  
    }  
  
    // 1,2, or 3 here ?  
}
```

1.

```
public static int dec () {  
    r = r - 1;  
    return r;  
}
```

2.

```
public static int inc2 () {  
    inc();  
    return inc();  
}
```

3.

```
public static int getInitialVal () {  
    return 0;  
}
```

Answer: 3

Static vs. Dynamic Class Members

```
public class FancyCounter {  
    private int c = 0;  
    private static int total = 0;  
  
    public int inc () {  
        c += 1;  
        total += 1;  
        return c;  
    }  
  
    public static int getTotal () {  
        return total;  
    }  
}
```

```
FancyCounter c1 = new FancyCounter();  
FancyCounter c2 = new FancyCounter();  
int v1 = c1.inc();  
int v2 = c2.inc();  
int v3 = c1.getTotal();  
System.out.println(v1 + " " + v2 + " " + v3);
```

Static Class Members

- Static methods can depend *only* on other static things
 - Static fields and methods, from the same or other classes
- Static methods *can* create *new* objects and use them
 - This is typically how `main` works
- `public static` fields are the "global" state of the program
 - Mutable global state should generally be avoided
 - Immutable global fields are useful: for constants like pi

```
public static final double PI = 3.14159265359793238462643383279;
```

Style: naming conventions

| Kind | Part-of-speech | Example |
|-----------------------------------|----------------|------------------|
| class | noun | RacingCar |
| field / variable | noun | initialSpeed |
| static final field (constants) | noun | MILES_PER_GALLON |
| method | verb | shiftGear |

- Identifiers consist of alphanumeric characters and `_` and cannot start with a digit
- The larger the scope, the more *informative* the name should be
- Conventions are important: variables, methods and classes can have the same name

Why naming conventions matter

```
public class Turtle {  
    private Turtle Turtle;  
    public Turtle() { }  
  
    public Turtle Turtle (Turtle Turtle) {  
        return Turtle;  
    }  
}
```

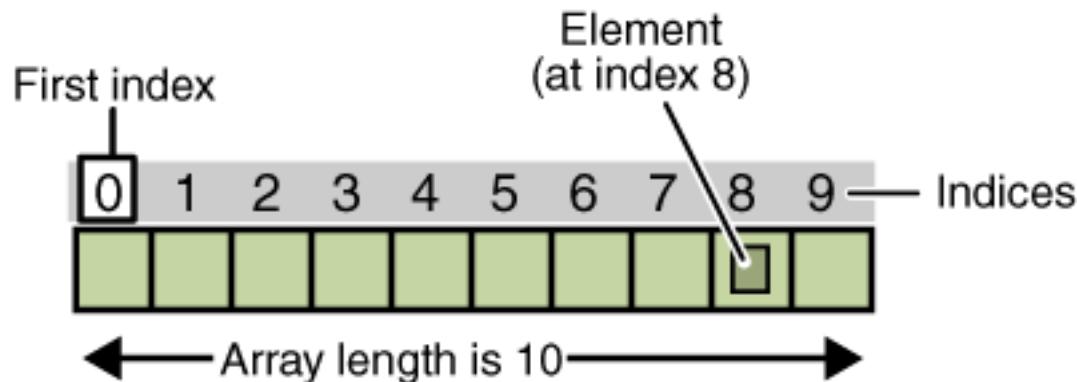
Many more details on good Java style here: http://www.seas.upenn.edu/~cis120/current/java_style.shtml

Java arrays

Working with static methods

Java Arrays: Indexing

- An array is a sequentially ordered collection of values that can be indexed in *constant* time.
- Index elements from 0

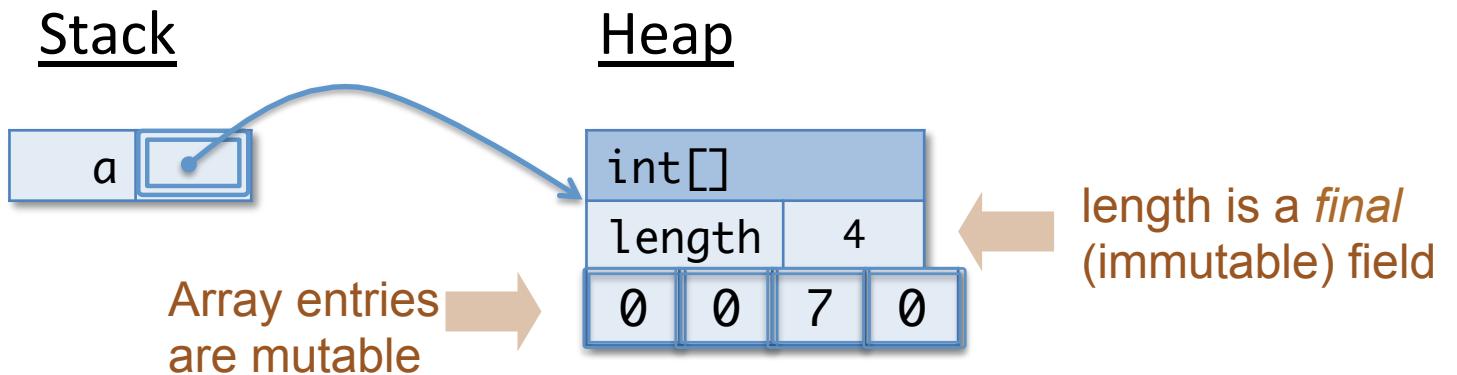


- Basic array expression forms
 - $a[i]$ access element of array a at index i
 - $a[i] = e$ assign e to element of array a at index i
 - $a.length$ get the number of elements in a

Java Arrays: Dynamic Creation

- Create an array a of size n with elements of type C
 $C[] a = new C[n];$
- Arrays live in the heap; values with array type are mutable references

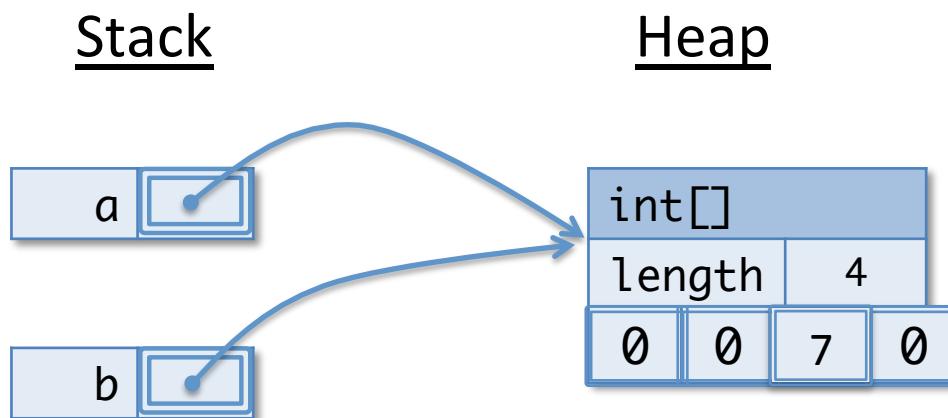
```
int[] a = new int[4];
a[2] = 7;
```



Java Arrays: Aliasing

- Variables of array type are references and can be aliases

```
int[] a = new int[4];
int[] b = a;
a[2] = 7;
int ans = b[2];
```



What is the value of ans at the end of this program?

```
int[] a = {1, 2, 3, 4};  
int ans = a[0];
```

1. 1
2. 2
3. 3
4. 4
5. NullPointerException
6. ArrayIndexOutOfBoundsException

What is the value of ans at the end of this program?

```
int[] a = {1, 2, 3, 4};  
int ans = a[a.length];
```

- 1. 1
- 2. 2
- 3. 3
- 4. 4
- 5. NullPointerException
- 6. ArrayIndexOutOfBoundsException

What is the value of ans at the end of this program?

```
int[] a = null;  
int ans = a.length;
```

1. 1
2. 2
3. 3
4. 0
5. NullPointerException
6. ArrayIndexOutOfBoundsException

What is the value of ans at the end of this program?

```
int[] a = {};
int ans = a.length;
```

1. 1
2. 2
3. 3
4. 0
5. NullPointerException
6. ArrayIndexOutOfBoundsException

What is the value of ans at the end of this program?

```
int[] a = {1, 2, 3, 4};  
int[] b = a;  
b[0] = 0;  
int ans = a[0];
```

1. 1
2. 2
3. 3
4. 0
5. NullPointerException
6. ArrayIndexOutOfBoundsException

What is the value of ans at the end of this program?

```
Counter[] a = { new Counter(), new Counter() };
Counter[] b = a;
a[0].inc();
b[0].inc();
int ans = a[0].inc();
```

1. 1
2. 2
3. 3
4. 0
5. NullPointerException
6. ArrayIndexOutOfBoundsException

```
public class Counter {

    private int r;

    public Counter () {
        r = 0;
    }

    public int inc () {
        r = r + 1;
        return r;
    }
}
```

What is the value of ans at the end of this program?

```
Counter[] a = { new Counter(), new Counter() };
Counter[] b = { new Counter(), new Counter() };
a[0].inc();
b[0].inc();
int ans = a[0].inc();
```

1. 1
2. 2
3. 3
4. 0
5. NullPointerException
6. ArrayIndexOutOfBoundsException

```
public class Counter {

    private int r;

    public Counter () {
        r = 0;
    }

    public int inc () {
        r = r + 1;
        return r;
    }

}
```

What is the value of ans at the end of this program?

```
Counter[] a = { new Counter(), new Counter() };
Counter[] b = { a[0], a[1] };
a[0].inc();
b[0].inc();
int ans = a[0].inc();
```

1. 1
2. 2
3. 3
4. 0
5. NullPointerException
6. ArrayIndexOutOfBoundsException

```
public class Counter {

    private int r;

    public Counter () {
        r = 0;
    }

    public int inc () {
        r = r + 1;
        return r;
    }

}
```

Array Iteration

For loops

```
initialization      loop condition      update  
for (int i = 0; i < a.length; i++) {  
    total += a[i];  
}  
                                ← loop body
```

```
static double sum(double[] a) {  
    double total = 0;  
    for (int i = 0; i < a.length; i++) {  
        total += a[i];  
    }  
    return total;  
}
```

General pattern for computing info about an array

Programming Languages and Techniques (CIS120)

Lecture 23
October 27, 2017

Java Arrays
Chapters 20 & 21

Announcements

- HW6: Java Programming (Pennstagram)
 - Due: Tuesday the 31st at 11:59pm
 - Note: Run JUnit tests (under Tools menu)
- Reminder: please complete mid-semester survey
 - See post on Piazza
- Upcoming: Midterm 2
 - Friday, November 10th in class
 - Coverage: mutable state, queues, deques, GUI, Java

mantra

Static == Decided at *Compile* Time
Dynamic == Decided at *Run* Time

Compile Time:

- javac Foo.java
- checks for *syntax* errors – is it possibly legal code
- typechecking – do the types make sense?
- compilation – produce Foo.class
- in Codio: "Build Project"

Run Time:

- java Foo arg1 arg2 arg3
- instantiates the abstract stack machine
- executes the code in Foo.class (assuming main exists)

Static method example

```
public class Max {  
    public static int max (int x, int y) {  
        if (x > y) {  
            return x;  
        } else {  
            return y;  
        }  
    }  
}
```

```
public static int max3(int x, int y, int z) {  
    return max(max(x,y), z);  
}
```

Internally (within the same class), call with just the method name

main method must be static; it is invoked to start the program running

closest analogue of top-level functions in OCaml, but must be a member of some class

```
public class Main {  
    public static void main (String[] args) {  
        System.out.println(Max.max(3,4));  
        return;  
    }  
}
```

Externally, prefix with name of the class

Static vs. Dynamic Methods

- Static Methods are *independent* of object values
 - Similar to OCaml functions
 - Cannot refer to the local state of objects (fields or normal methods)
- Use static methods for:
 - Non-OO programming
 - Programming with primitive types: Math.sin(60), Integer.toString(3), Boolean.valueOf("true")
 - “public static void main”
- “Normal” methods are *dynamic*
 - Need access to the local state of the particular object on which they are invoked
 - We only know at *runtime* which method will get called

```
void moveTwice (Displaceable o) {  
    o.move (1,1); o.move(1,1);  
}
```

Method call examples

- Calling a (dynamic) method of an object (o) that returns a number:

```
x = o.m() + 5;
```

- Calling a static method of a class (C) that returns a number:

```
x = C.m() + 5;
```

- Calling a method that returns void:

Static

```
C.m();
```

Dynamic

```
o.m();
```

- Calling a static or dynamic method in a method of the same class:

Either

```
m();
```

Static

```
C.m();
```

Dynamic

```
this.m();
```

- Calling (dynamic) methods that return objects:

```
x = o.m().n();
x = o.m().n().x().y().z().a().b().c().d().e();
```

Which **static** method can we add to this class?

```
public class Counter {  
  
    private int r;  
  
    public Counter () {  
        r = 0;  
    }  
  
    public int inc () {  
        r = r + 1;  
        return r;  
    }  
  
    // 1,2, or 3 here ?  
}
```

1.

```
public static int dec () {  
    r = r - 1;  
    return r;  
}
```

2.

```
public static int inc2 () {  
    inc();  
    return inc();  
}
```

3.

```
public static int getInitialVal () {  
    return 0;  
}
```

Answer: 3

Recap: Static vs. Dynamic Class Members

```
public class FancyCounter {  
    private int c = 0;  
    private static int total = 0;  
  
    public int inc () {  
        c += 1;  
        total += 1;  
        return c;  
    }  
  
    public static int getTotal () {  
        return total;  
    }  
}
```

```
FancyCounter c1 = new FancyCounter();  
FancyCounter c2 = new FancyCounter();  
int v1 = c1.inc();  
int v2 = c2.inc();  
int v3 = c1.getTotal();  
System.out.println(v1 + " " + v2 + " " + v3);
```

Static Class Members

- Static methods can depend *only* on other static things
 - Static fields and methods, from the same or other classes
- Static methods *can* create *new* objects and use them
 - This is typically how `main` works
- `public static` fields are the "global" state of the program
 - Mutable global state should generally be avoided
 - Immutable global fields are useful: for constants like pi

```
public static final double PI = 3.14159265359793238462643383279;
```

Style: naming conventions

| Kind | Part-of-speech | Example |
|-----------------------------------|----------------|------------------|
| class | noun | RacingCar |
| field / variable | noun | initialSpeed |
| static final field (constants) | noun | MILES_PER_GALLON |
| method | verb | shiftGear |

- Identifiers consist of alphanumeric characters and `_` and cannot start with a digit
- The larger the scope, the more *informative* the name should be
- Conventions are important: variables, methods and classes can have the same name

Why naming conventions matter

```
public class Turtle {  
    private Turtle Turtle;  
    public Turtle() { }  
  
    public Turtle Turtle (Turtle Turtle) {  
        return Turtle;  
    }  
}
```

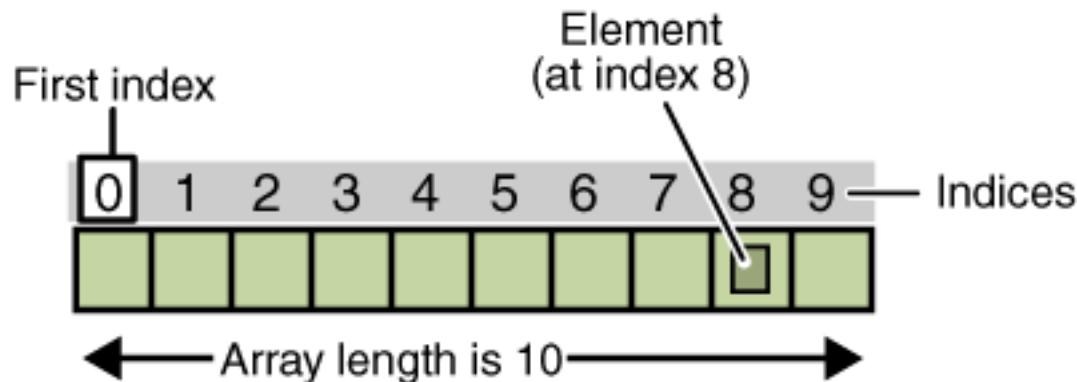
Many more details on good Java style here: http://www.seas.upenn.edu/~cis120/current/java_style.shtml

Java arrays

Working with static methods

Java Arrays: Indexing

- An array is a sequentially ordered collection of values that can be indexed in *constant* time.
- Index elements from 0

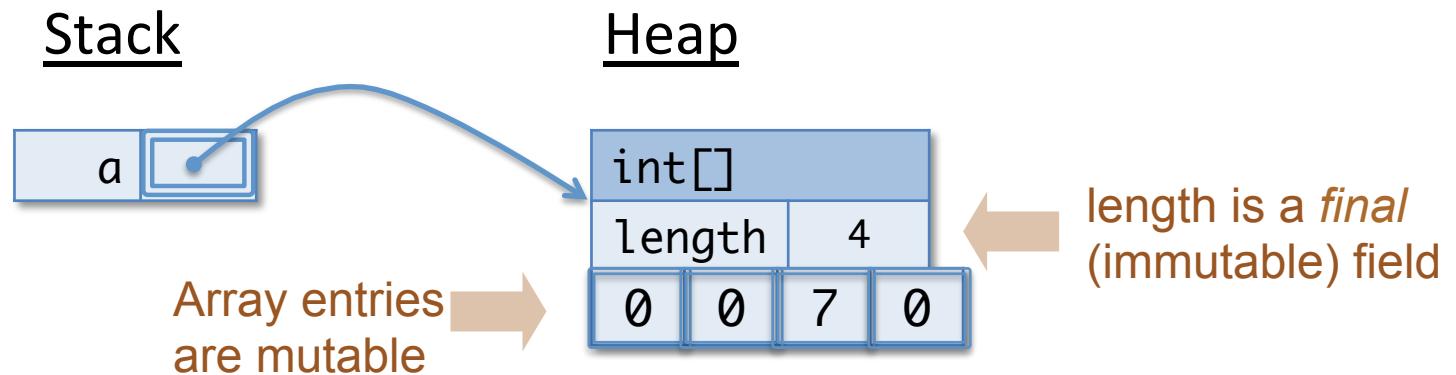


- Basic array expression forms
 - $a[i]$ access element of array a at index i
 - $a[i] = e$ assign e to element of array a at index i
 - $a.length$ get the number of elements in a

Java Arrays: Dynamic Creation

- Create an array `a` of size `n` with elements of type `C`
`C[] a = new C[n];`
- Create an array of four integers, initialized as given:
`int[] x = {1; 2; 3; 4};`
- Arrays live in the heap; values with array type are mutable references:

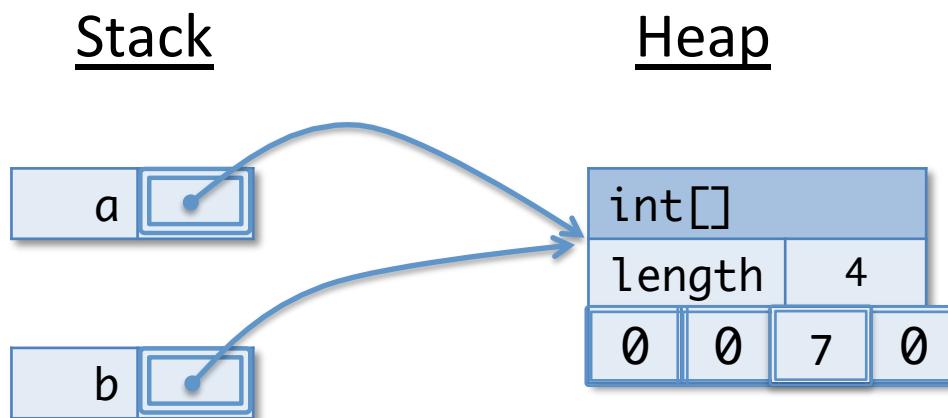
```
int[] a = new int[4];
a[2] = 7;
```



Java Arrays: Aliasing

- Variables of array type are references and can be aliases

```
int[] a = new int[4];
int[] b = a;
a[2] = 7;
int ans = b[2];
```



What is the value of ans at the end of this program?

```
int[] a = {1, 2, 3, 4};  
int ans = a[a.length];
```

- 1. 1
- 2. 2
- 3. 3
- 4. 4
- 5. NullPointerException
- 6. ArrayIndexOutOfBoundsException

Answer: ArrayIndexOutOfBoundsException

What is the value of ans at the end of this program?

```
int[] a = null;  
int ans = a.length;
```

- 1. 1
- 2. 2
- 3. 3
- 4. 0
- 5. NullPointerException
- 6. ArrayIndexOutOfBoundsException

Answer: NullPointerException

What is the value of ans at the end of this program?

```
int[] a = {};
int ans = a.length;
```

- 1. 1
- 2. 2
- 3. 3
- 4. 0
- 5. NullPointerException
- 6. ArrayIndexOutOfBoundsException

Answer: 0

What is the value of ans at the end of this program?

```
int[] a = {1, 2, 3, 4};  
int[] b = a;  
b[0] = 0;  
int ans = a[0];
```

1. 1
2. 2
3. 3
4. 0
5. NullPointerException
6. ArrayIndexOutOfBoundsException

Answer: 0

Array Iteration

For loops

```
initialization      loop condition      update  
for (int i = 0; i < a.length; i++) {  
    total += a[i];  
}  
          ← loop body
```

```
static double sum(double[] a) {  
    double total = 0;  
    for (int i = 0; i < a.length; i++) {  
        total += a[i];  
    }  
    return total;  
}
```

General pattern for computing info about an array

Multidimensional Arrays

Multi-Dimensional Arrays

A 2-d array is just an array of arrays...

```
String[][] names = {{"Mr. ", "Mrs. ", "Ms. "},  
                     {"Smith", "Jones"}};  
  
System.out.println(names[0][0] + names[1][0]);  
    // --> Mr. Smith  
System.out.println(names[0][2] + names[1][1]);  
    // --> Ms. Jones
```

String[] [] just means (String[])[]
names[1][1] just means (names[1])[1]
More brackets → more dimensions

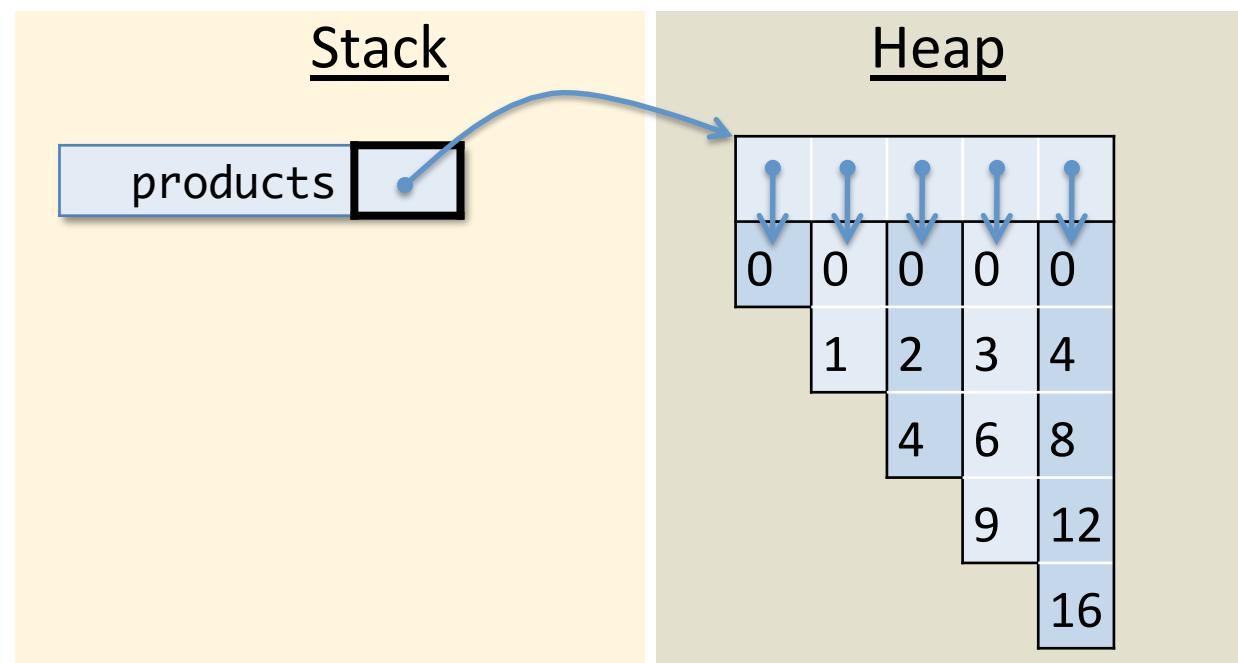
Multi-Dimensional Arrays

```
int[][] products = new int[5][];  
for(int col = 0; col < 5; col++) {  
    products[col] = new int[col+1];  
    for(int row = 0; row <= col; row++) {  
        products[col][row] = col * row;  
    }  
}
```

What would a “Java ASM”
stack and heap look like
after running this program?

Multi-Dimensional Arrays

```
int[][] products = new int[5][];
for(int col = 0; col < 5; col++) {
    products[col] = new int[col+1];
    for(int row = 0; row <= col; row++) {
        products[col][row] = col * row;
    }
}
```



Demo

ArrayDemo.java

Design Exercise: Resizable Arrays

Arrays that grow without bound.

Resizable Arrays

```
public class ResArray {  
  
    /** Constructor, takes no arguments. */  
    public ResArray() { ... }  
  
    /** Access the array at position i. If position i has not yet  
     * been initialized, return 0.  
     */  
    public int get(int i) { ... }  
  
    /** Modify the array at position i to contain the value v. */  
    public void set(int i, int v) { ... }  
  
    /** Return the extent of the array. */  
    public int getExtent() { ... }  
  
}
```

Object Invariant: extent is 1 past
the last nonzero value in data
(can be 0 if the array is all zeros)

Programming Languages and Techniques (CIS120)

Lecture 24

October 30, 2017

Arrays, Java ASM

Announcements

- HW6: Java Programming (Pennstagram)
 - Due: TOMORROW at 11:59pm
 - Note: Run JUnit tests (under Tools menu)
- Reminder: please complete mid-semester survey
 - See post on Piazza
- Upcoming: Midterm 2
 - Friday, November 10th in class
 - Coverage: mutable state, queues, deques, GUI, Java

Design Exercise: Resizable Arrays

Arrays that grow without bound.

Resizable Arrays

```
public class ResArray {  
  
    /** Constructor, takes no arguments. */  
    public ResArray() { ... }  
  
    /** Access the array at position i. If position i has not yet  
     * been initialized, return 0.  
     */  
    public int get(int i) { ... }  
  
    /** Modify the array at position i to contain the value v. */  
    public void set(int i, int v) { ... }  
  
    /** Return the extent of the array. */  
    public int getExtent() { ... }  
  
}
```

Object Invariant: extent is 1 past
the last nonzero value in data
(can be 0 if the array is all zeros)

ResArray (review)

```
public class ResArray {  
  
    private int[] data = {};  
  
    /** Constructor, takes no arguments. */  
    public ResArray() {  
    }  
  
    /** Access the array at position i.  
     * If position i has not yet been initialized, return 0.  
     */  
    public int get(int idx) {  
        if (idx >= data.length) {  
            return 0;  
        } else {  
            return data[idx];  
        }  
    }  
...  
}
```

ResArray (review)

```
public class ResArray {  
    ...  
    private int[] data = {};  
  
    /** Modify the array at position i to contain the value v. */  
    public void set(int idx, int val) {  
        if (idx >= data.length) {  
            int[] newdata = new int[idx+1]  
            for (int i=0; i < data.length; i++) {  
                newdata[i] = data[i];  
            }  
            data = newdata;  
        }  
        data[idx] = val;  
    }  
  
    public int[] values() {  
        return data;  
    }  
}
```

ResArray (review)

```
public class ResArray {  
    ...  
    private int[] data = {};  
  
    /** Modify the array at position idx to contain the value val. */  
    public void set(int idx, int val) {  
        if (idx >= data.length) {  
            int[] newdata = new int[Math.max(idx+1, data.length*2)]  
            for (int i=0; i < data.length; i++) {  
                newdata[i] = data[i];  
            }  
            data = newdata;  
        }  
        data[idx] = val;  
    }  
  
    public int[] values() {  
        return data;  
    }  
}
```

Adding extent

```
private int extent = 0;
/* INVARIANT: extent = 1+index of last nonzero
 * element, or 0 if all elements are 0. */

/** Modify the array at position i to contain the value v. */
public void set(int idx, int val) {
    if (idx < 0) {
        throw new IllegalArgumentException();
    }
    grow(idx);
    data[idx] = val;
    if (val != 0 && idx+1 > extent) {
        extent = idx+1;
    }
    if (val == 0 && idx+1 == extent) {
        while (extent > 0 && data[extent-1] == 0) {
            extent--;
        }
    }
}

/** Return the extent of the array. */
public int getExtent() {
    return extent;
}
```

Revenge of the Son of the Abstract Stack Machine

The Java Abstract Stack Machine

Objects, Arrays, and Static Methods

Java Abstract Stack Machine

- Similar to OCaml Abstract Stack Machine
 - Workspace
 - Contains the currently executing code
 - Stack
 - Remembers the values of local variables and "what to do next" after function/method calls
 - Heap
 - Stores reference types: objects and arrays
- Key differences:
 - Everything, including stack slots, is mutable by default
 - Objects store *what class was used to create them*
 - Arrays store *type information and length*
 - New component: *Class table (coming soon)*

Java Primitive Values

- The values of these data types occupy (less than) one machine word and are stored directly in the stack slots.

| Type | Description | Values |
|---------|----------------------------|---------------------------|
| byte | 8-bit | -128 to 127 |
| short | 16-bit integer | -32768 to 32767 |
| int | 32-bit integer | -2^{31} to $2^{31} - 1$ |
| long | 64-bit integer | -2^{63} to $2^{63} - 1$ |
| float | 32-bit IEEE floating point | |
| double | 64-bit IEEE floating point | |
| boolean | true or false | true false |
| char | 16-bit unicode character | 'a' 'b' '\u0000' |

Heap Reference Values

Arrays

- Type of values that it stores
- Length
- Values for all of the array elements

```
int [] a =  
    { 0, 0, 7, 0 };
```

| int[] | |
|--------|---|
| length | 4 |
| 0 | 0 |

length *never* mutable;
elements *always* mutable

Objects

- Name of the class that constructed it
- Values for all of the fields

```
class Node {  
    private int elt;  
    private Node next;
```

...

}

| Node | |
|------|------|
| elt | 1 |
| next | null |

fields may or may not be mutable
public/private not tracked by ASM

ResArray ASM

Workspace

```
ResArray x = new ResArray();
x.set(3,2);
x.set(4,1);
x.set(4,0);
```

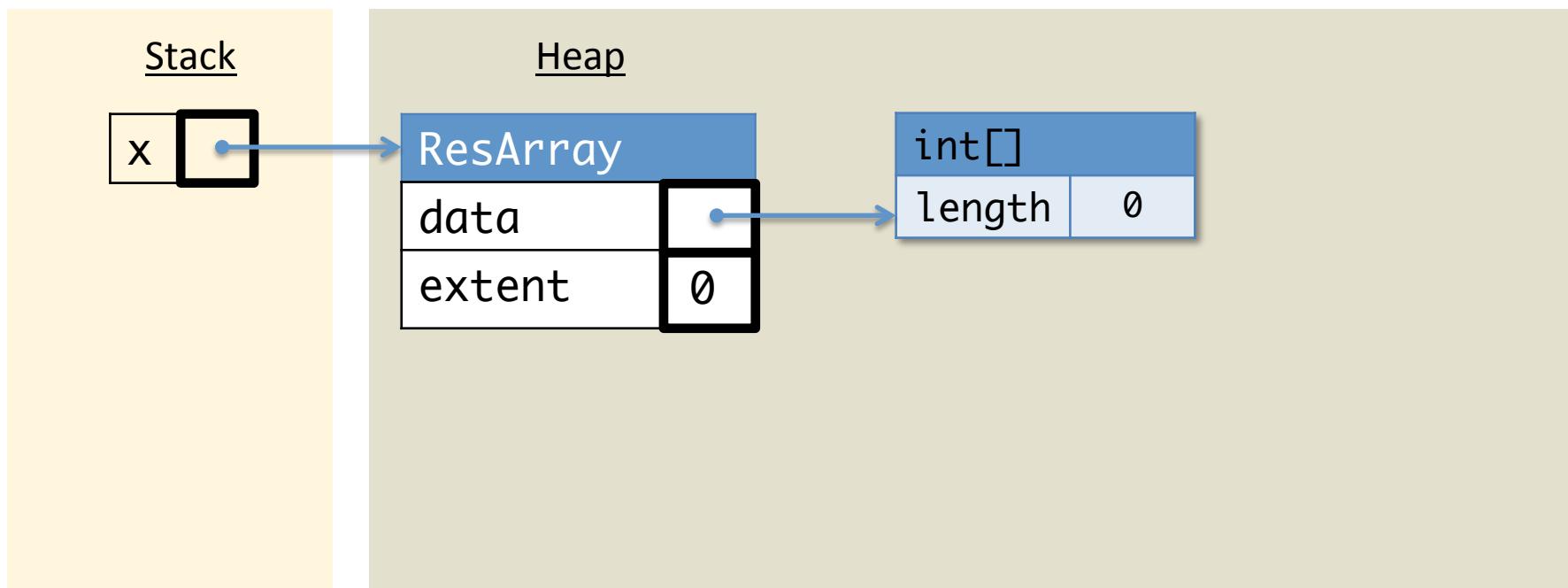
Stack

Heap

ResArray ASM

Workspace

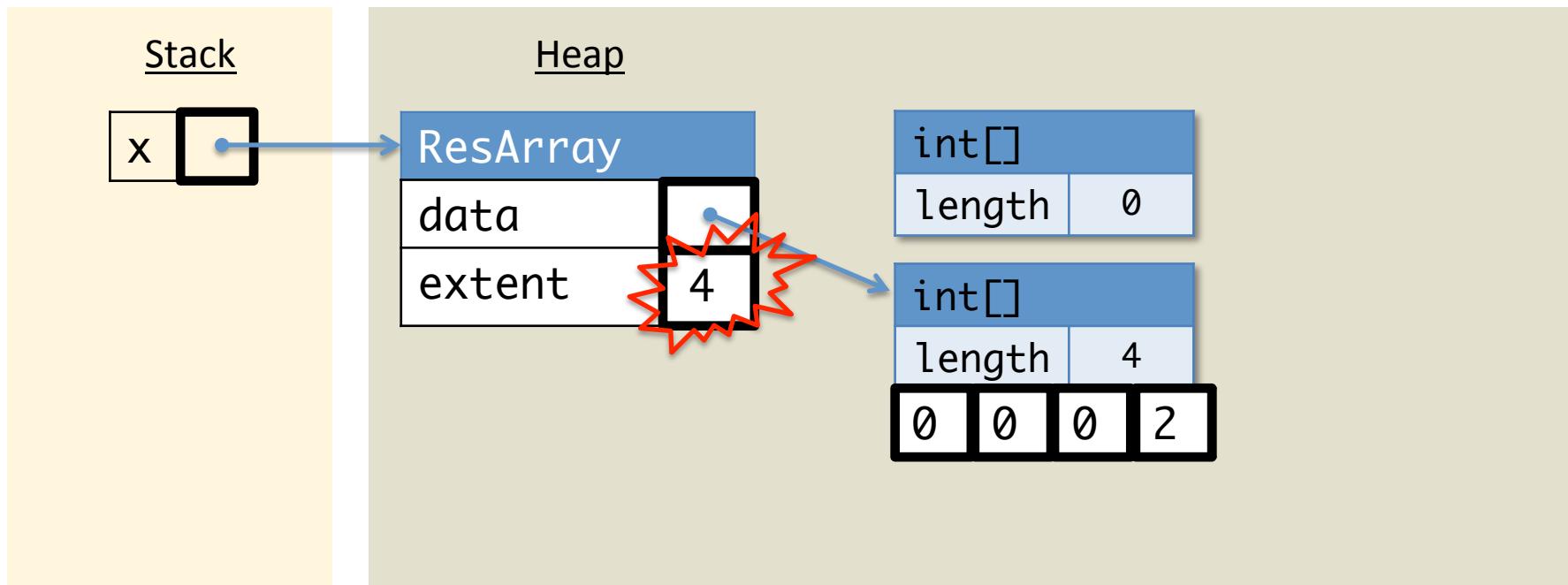
```
ResArray x = new ResArray();  
x.set(3,2);  
x.set(4,1);  
x.set(4,0);
```



ResArray ASM

Workspace

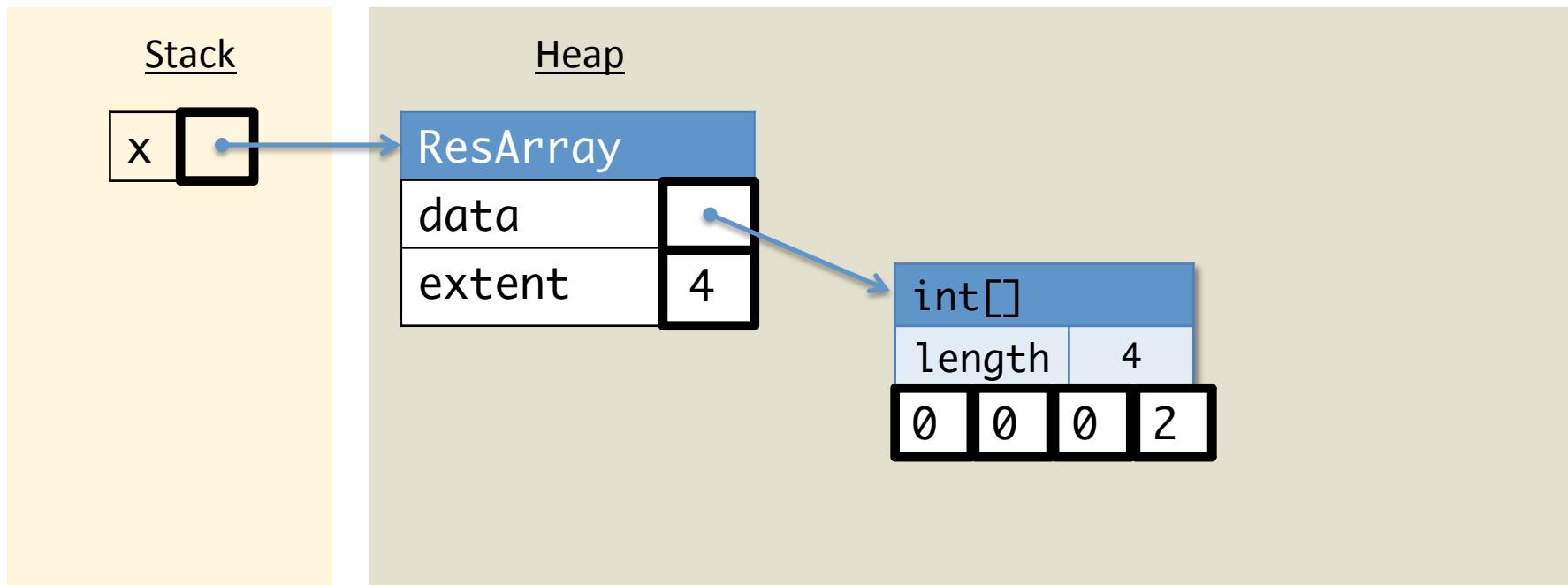
```
ResArray x = new ResArray();  
x.set(3,2);  
x.set(4,1);  
x.set(4,0);
```



ResArray ASM

Workspace

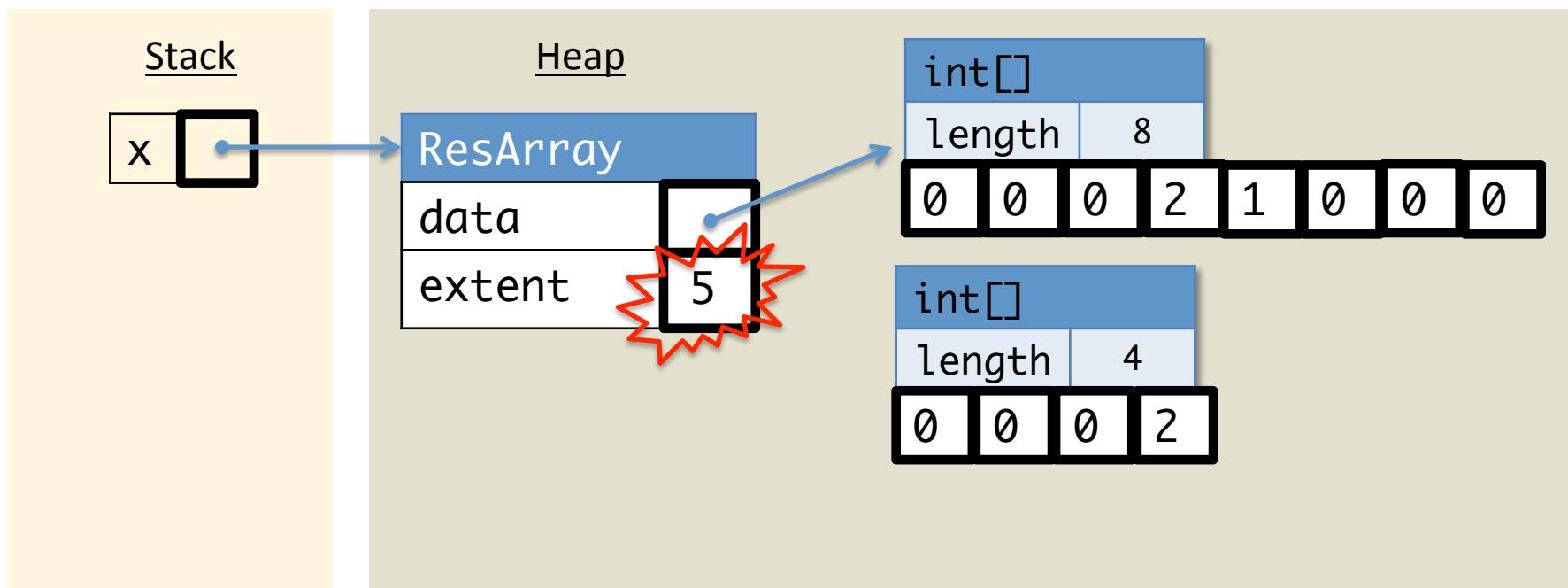
```
ResArray x = new ResArray();  
x.set(3,2);  
x.set(4,1);  
x.set(4,0);
```



ResArray ASM

Workspace

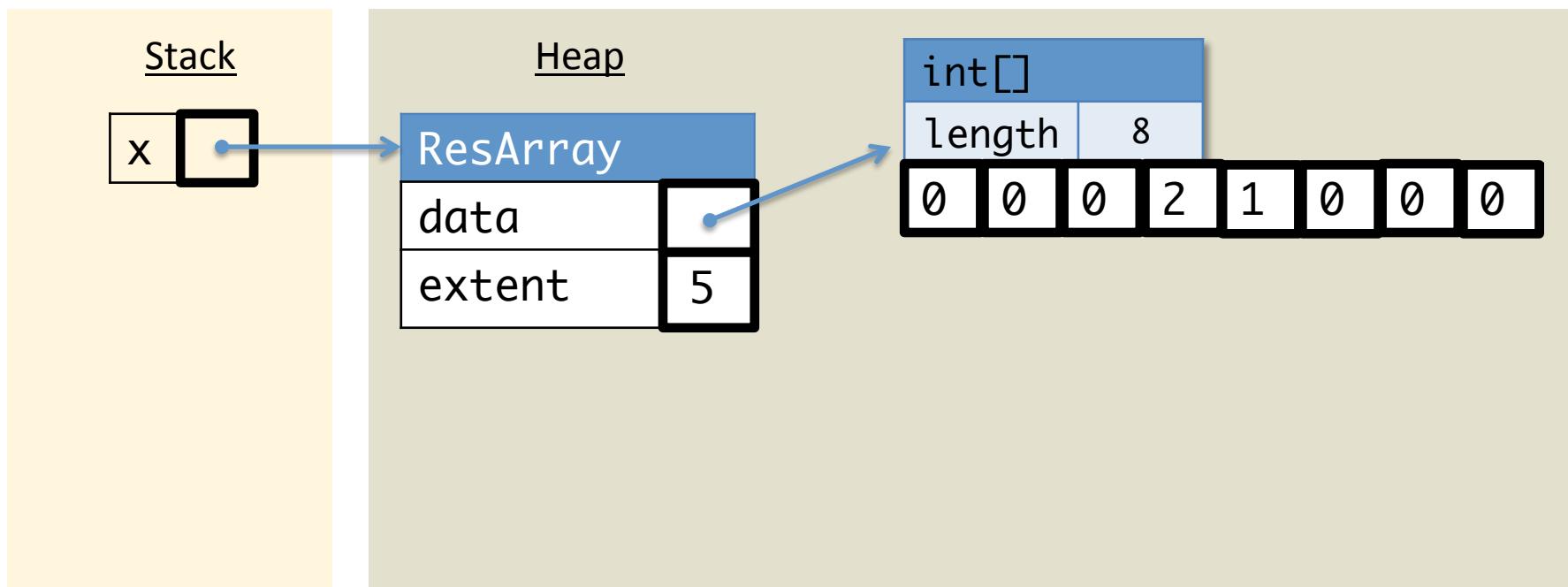
```
ResArray x = new ResArray();  
x.set(3,2);  
x.set(4,1);  
x.set(4,0);
```



ResArray ASM

Workspace

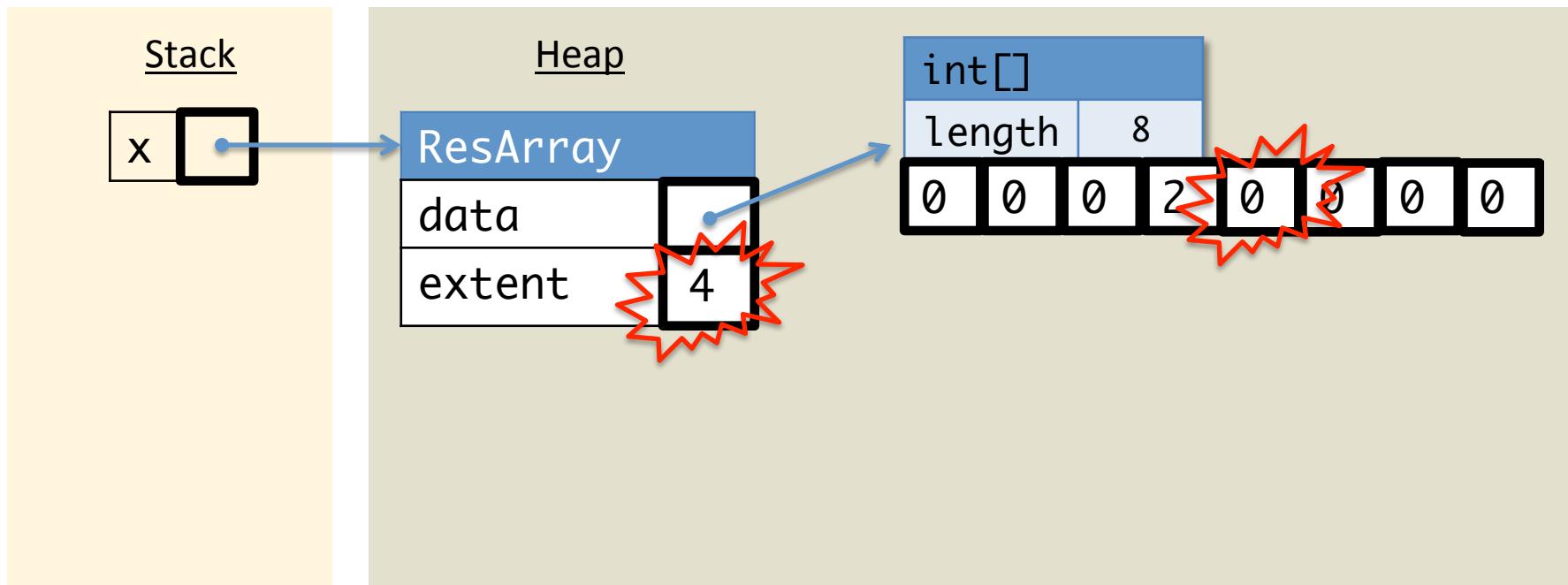
```
ResArray x = new ResArray();  
x.set(3,2);  
x.set(4,1);  
x.set(4,0);
```



ResArray ASM

Workspace

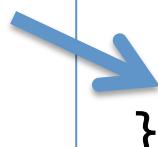
```
ResArray x = new ResArray();  
x.set(3,2);  
x.set(4,1);  
x.set(4,0);
```



Resizable Arrays

```
public class ResArray {  
  
    /** Constructor, takes no arguments. */  
    public ResArray() { ... }  
  
    /** Access the array at position i. If position i has not yet  
     * been initialized, return 0.  
     */  
    public int get(int i) { ... }  
  
    /** Modify the array at position i to contain the value v. */  
    public void set(int i, int v) { ... }  
  
    /** Return the extent of the array. */  
    public int getExtent() { ... }  
  
    /** The smallest prefix of the ResArray  
     * that contains all of the nonzero values, as a normal array.  
     */  
    public int[] values() { ... }  
}
```

Object Invariant: extent is always 1 past the last nonzero value in data (or 0 if the array is all zeros)



Values Method

```
public int[] values() {  
    int[] values = new int[extent];  
    for (int i=0; i<extent; i++) {  
        values[i] = data[i];  
    }  
    return values;  
}
```

Or maybe we can do it more straightforwardly? ...

```
public int[] values() {  
    return data;  
}
```

This optimized implementation of values correctly encapsulates the state of the ResArray object.

```
public int[] values() {  
    return data;  
}
```

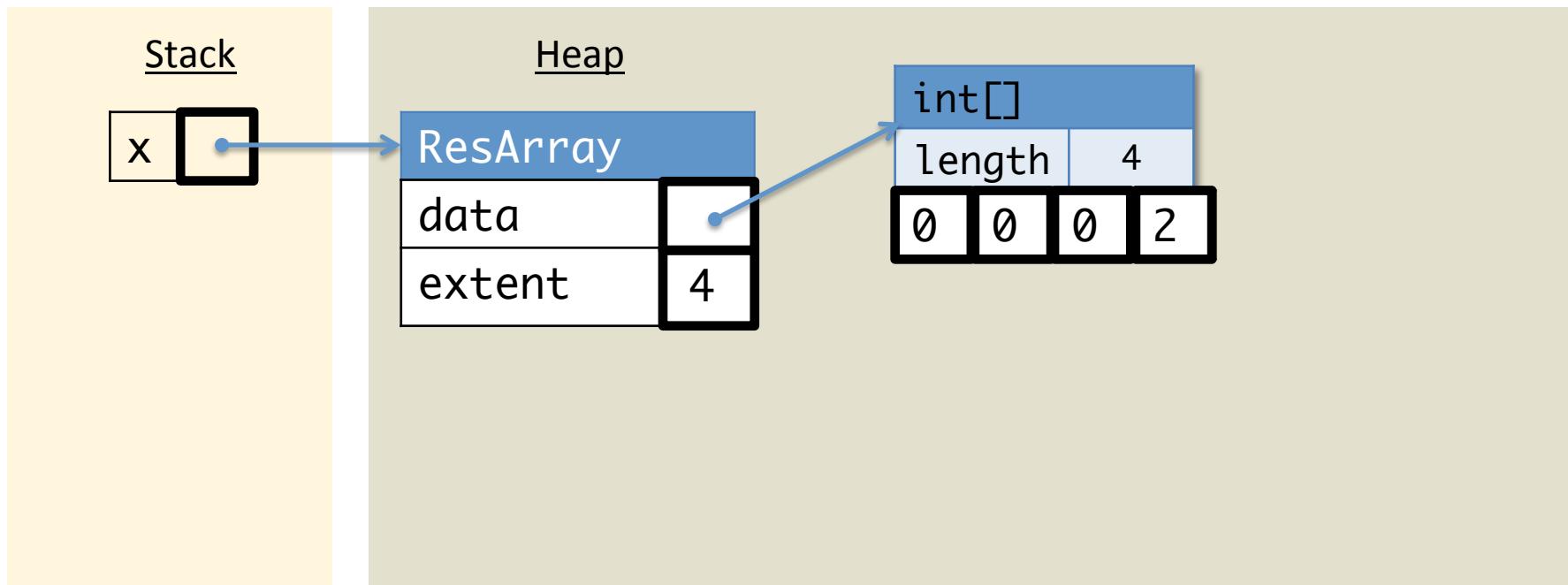
1. True
2. False

Answer: False

ResArray ASM

Workspace

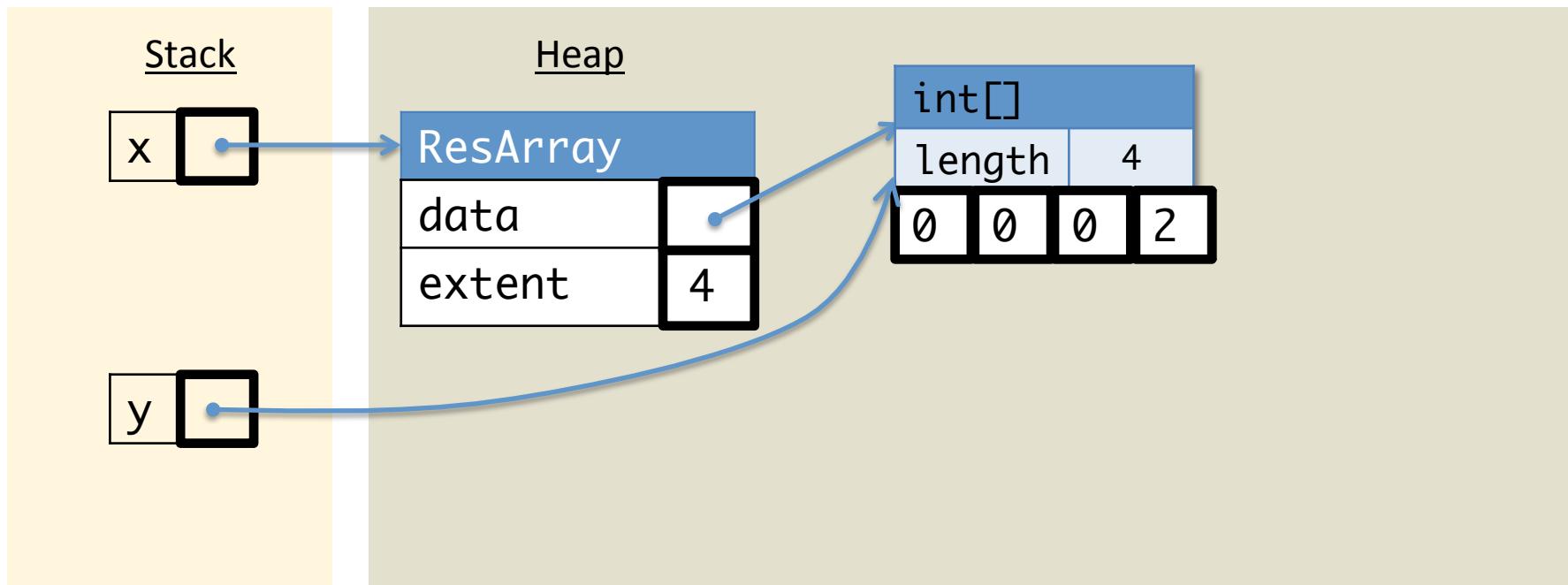
```
ResArray x = new ResArray();  
x.set(3,2);  
int[] y = x.values();  
y[3] = 0;
```



ResArray ASM

Workspace

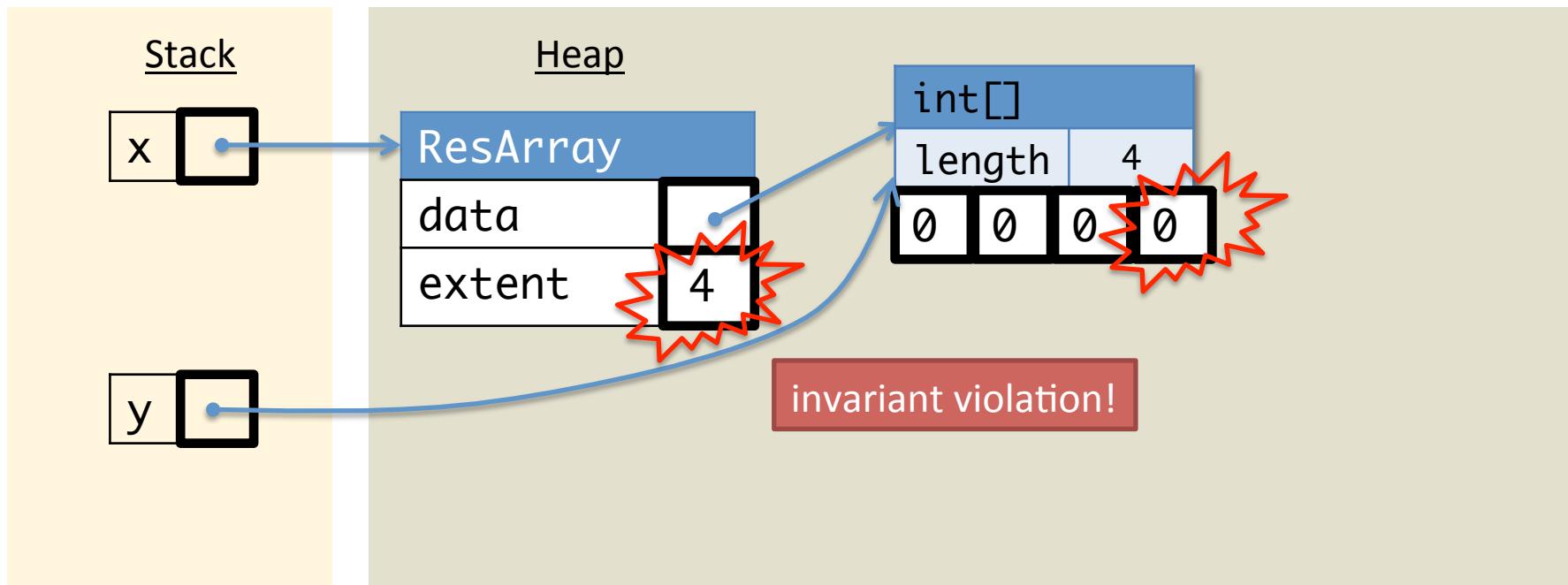
```
ResArray x = new ResArray();  
x.set(3,2);  
int[] y = x.values();  
y[3] = 0;
```



ResArray ASM

Workspace

```
ResArray x = new ResArray();  
x.set(3,2);  
int[] y = x.values();  
y[3] = 0;
```



Object encapsulation

- *All modification to the state of the object must be done using the object's own methods.*
- Use encapsulation to preserve invariants about the state of the object.
- Enforce encapsulation by not returning aliases from methods.

Objects on the ASM

What does the heap look like at the end of this program?

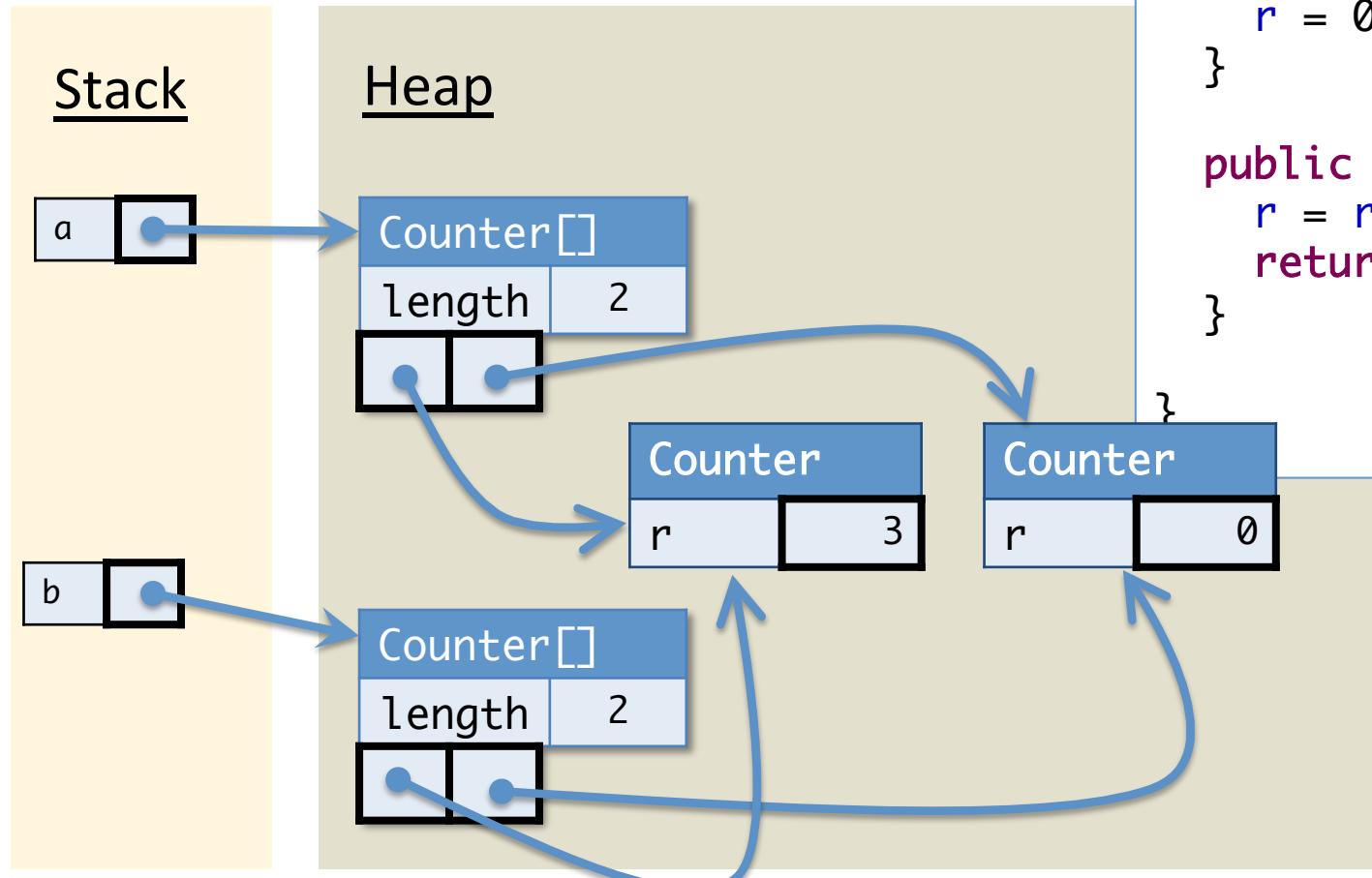
```
Counter[] a = { new Counter(), new Counter() };
Counter[] b = { a[0], a[1] };
a[0].inc();
b[0].inc();
int ans = a[0].inc();
```

```
public class Counter {
    private int r;
    public Counter() {
        r = 0;
    }
    public int inc() {
        r = r + 1;
        return r;
    }
}
```

What does the ASM look like at the end of this program?

```
Counter[] a = { new Counter(), new Counter() };
Counter[] b = { a[0], a[1] };
a[0].inc();
b[0].inc();
int ans = a[0].inc();
```

```
public class Counter {
    private int r;
    public Counter() {
        r = 0;
    }
    public int inc() {
        r = r + 1;
        return r;
    }
}
```



What does the following program print?

1 – 9

or 0 for "NullPointerException"

```
public class Node {  
    public int elt;  
    public Node next;  
    public Node(int e0, Node n0) {  
        elt = e0;  
        next = n0;  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
        Node n1 = new Node(1, null);  
        Node n2 = new Node(2, n1);  
        Node n3 = n2;  
        n3.next.next = n2;  
        Node n4 = new Node(4, n1.next);  
        n2.next.elt = 9;  
        System.out.println(n1.elt);  
    }  
}
```

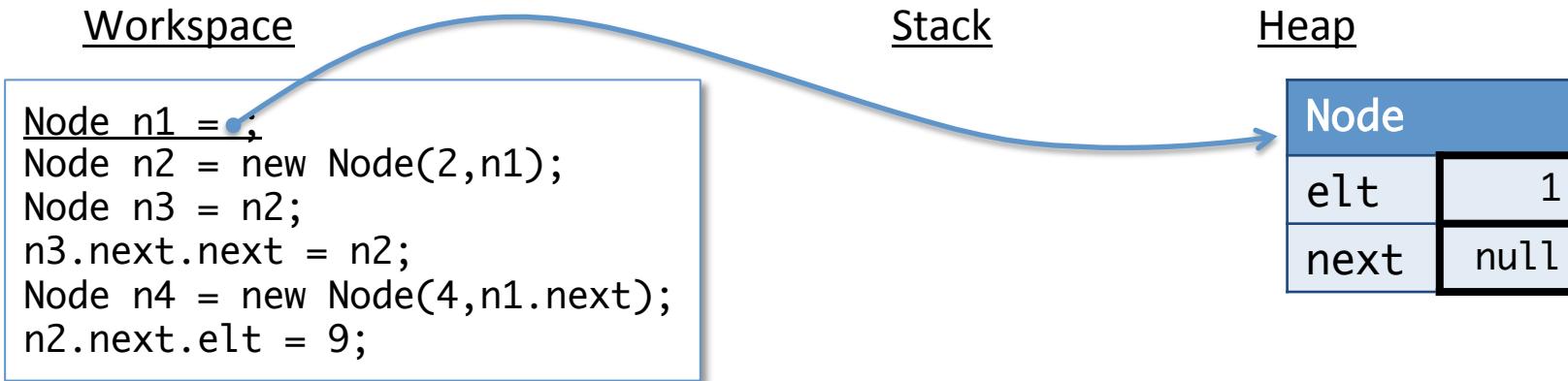
Answer: 9

Workspace

```
Node n1 = new Node(1,null);
Node n2 = new Node(2,n1);
Node n3 = n2;
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 9;
```

Stack

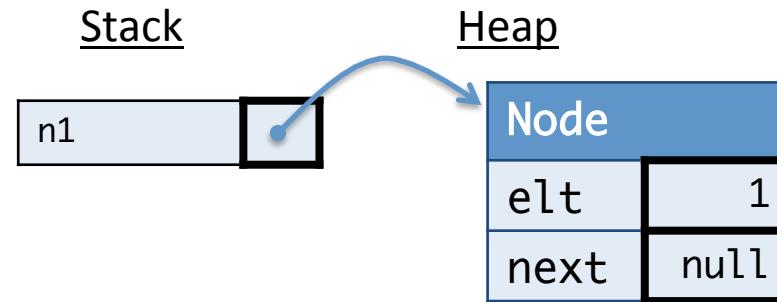
Heap



Note: we're skipping details here about how the constructor works. We'll fill them in next week. For now, assume the constructor allocates and initializes the object in one step.

Workspace

```
Node n2 = new Node(2,n1);
Node n3 = n2;
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next_elt = 9;
```



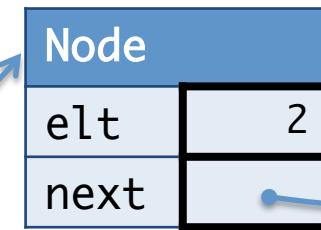
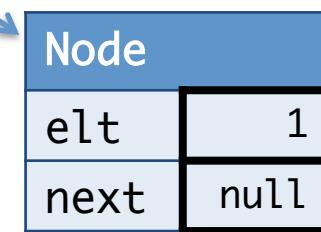
Workspace

```
Node n2 = ;  
Node n3 = n2;  
n3.next.next = n2;  
Node n4 = new Node(4,n1.next);  
n2.next_elt = 9;
```

Stack

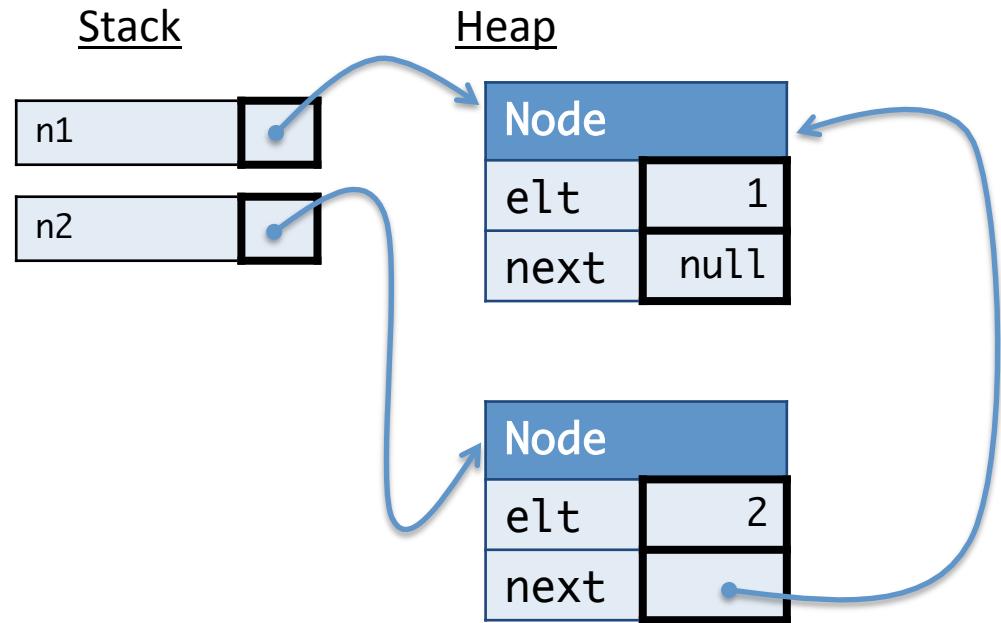


Heap



Workspace

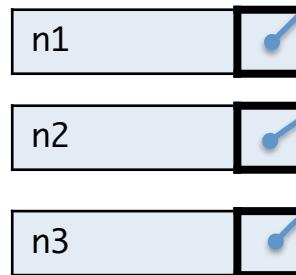
```
Node n3 = n2;
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next_elt = 9;
```



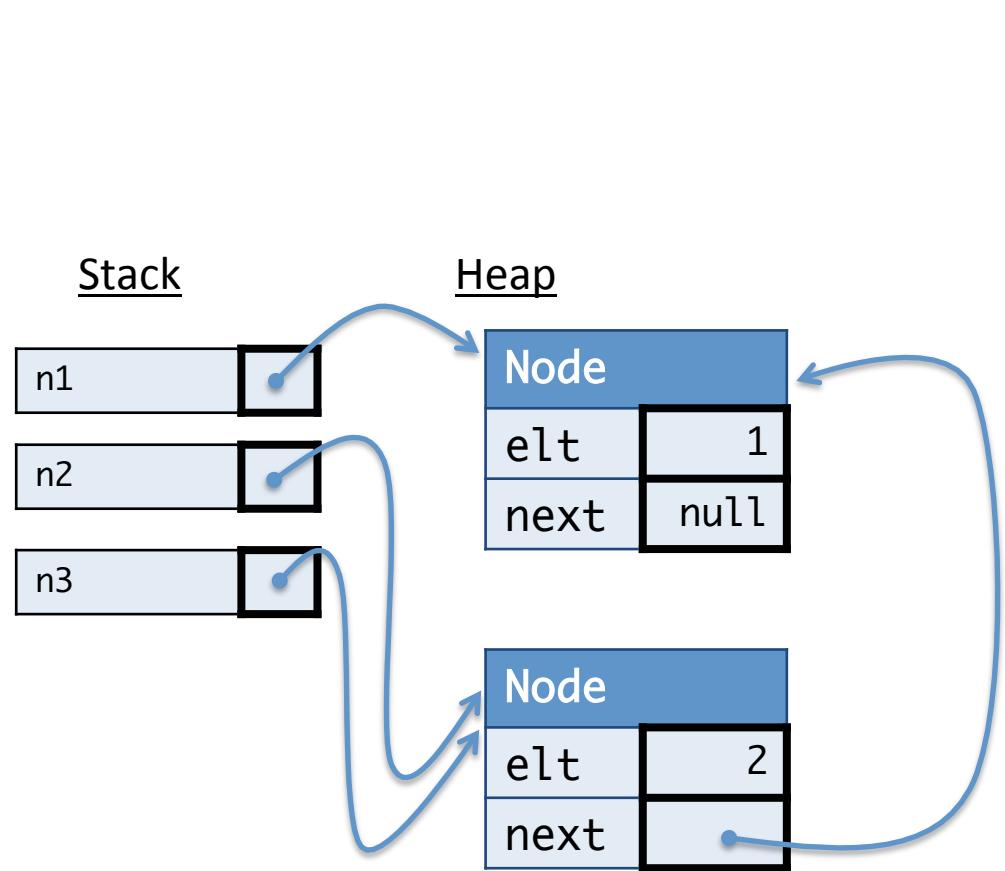
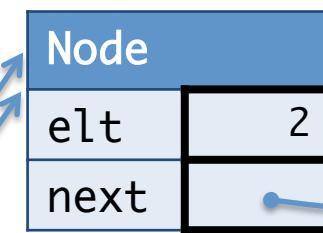
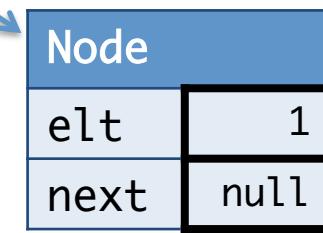
Workspace

```
n3.next.next = n2;  
Node n4 = new Node(4,n1.next);  
n2.next.elt = 9;
```

Stack



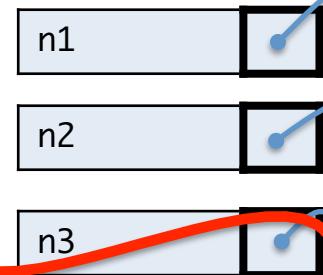
Heap



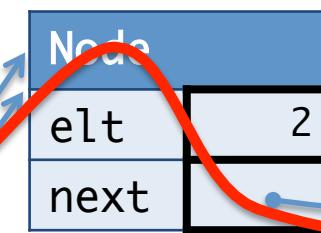
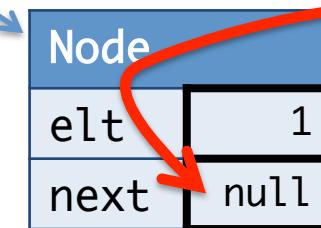
Workspace

```
n3.next.next = n2;  
Node n4 = new Node(4,n1.next);  
n2.next.elt = 9;
```

Stack



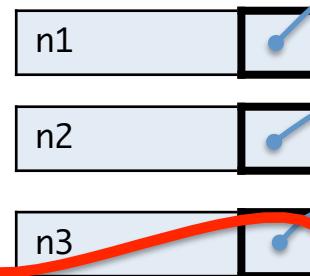
Heap



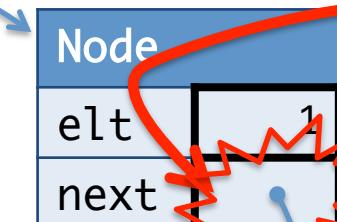
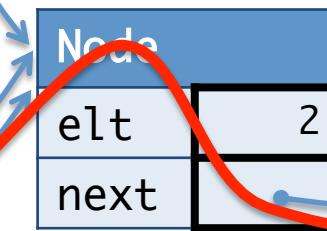
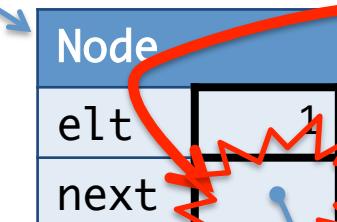
Workspace

```
n3.next.next = n2;  
Node n4 = new Node(4,n1.next);  
n2.next_elt = 9;
```

Stack

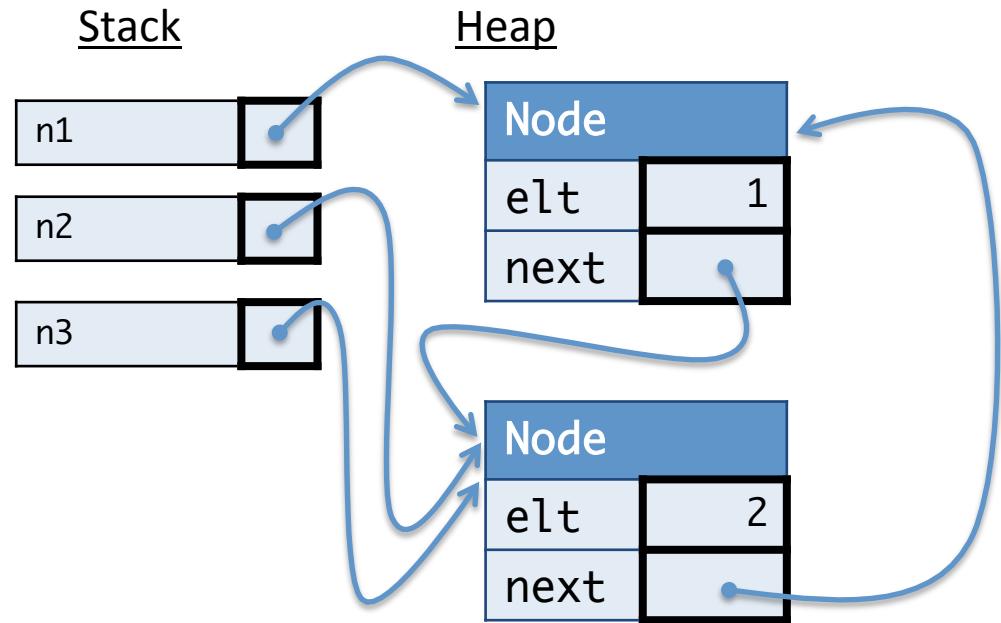


Heap



Workspace

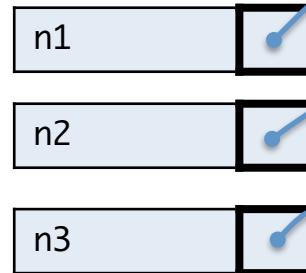
```
Node n4 = new Node(4,n1.next);  
n2.next.elt = 9;
```



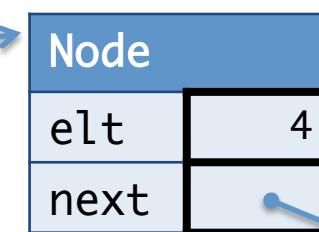
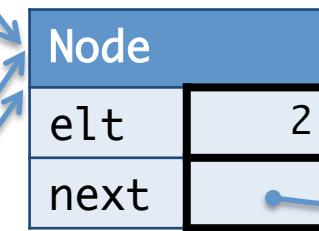
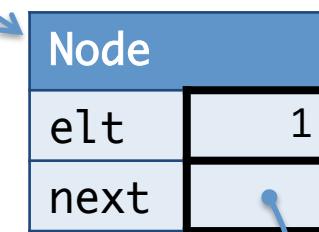
Workspace

```
Node n4 = ;  
n2.next.elt = 9;
```

Stack



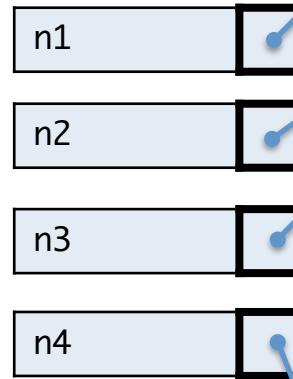
Heap



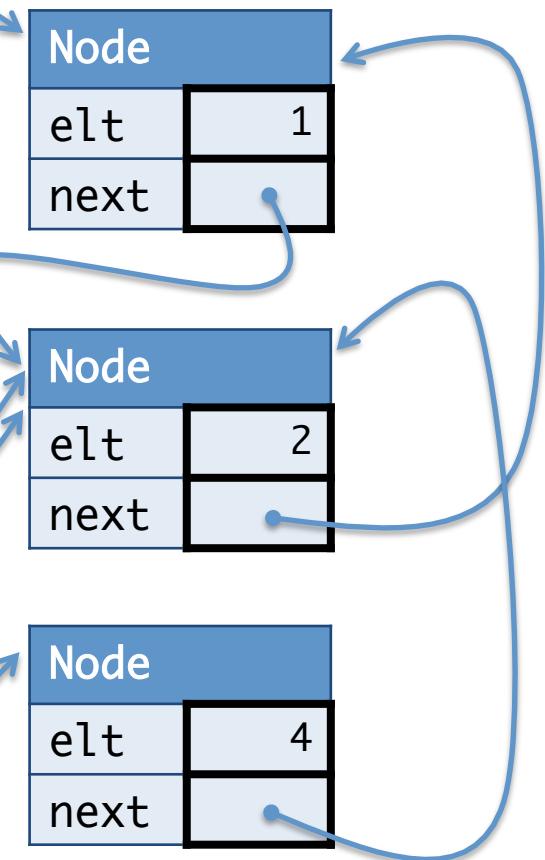
Workspace

```
n2.next_elt = 9;
```

Stack



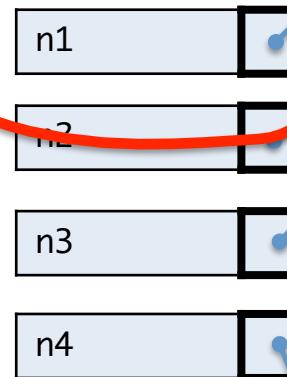
Heap



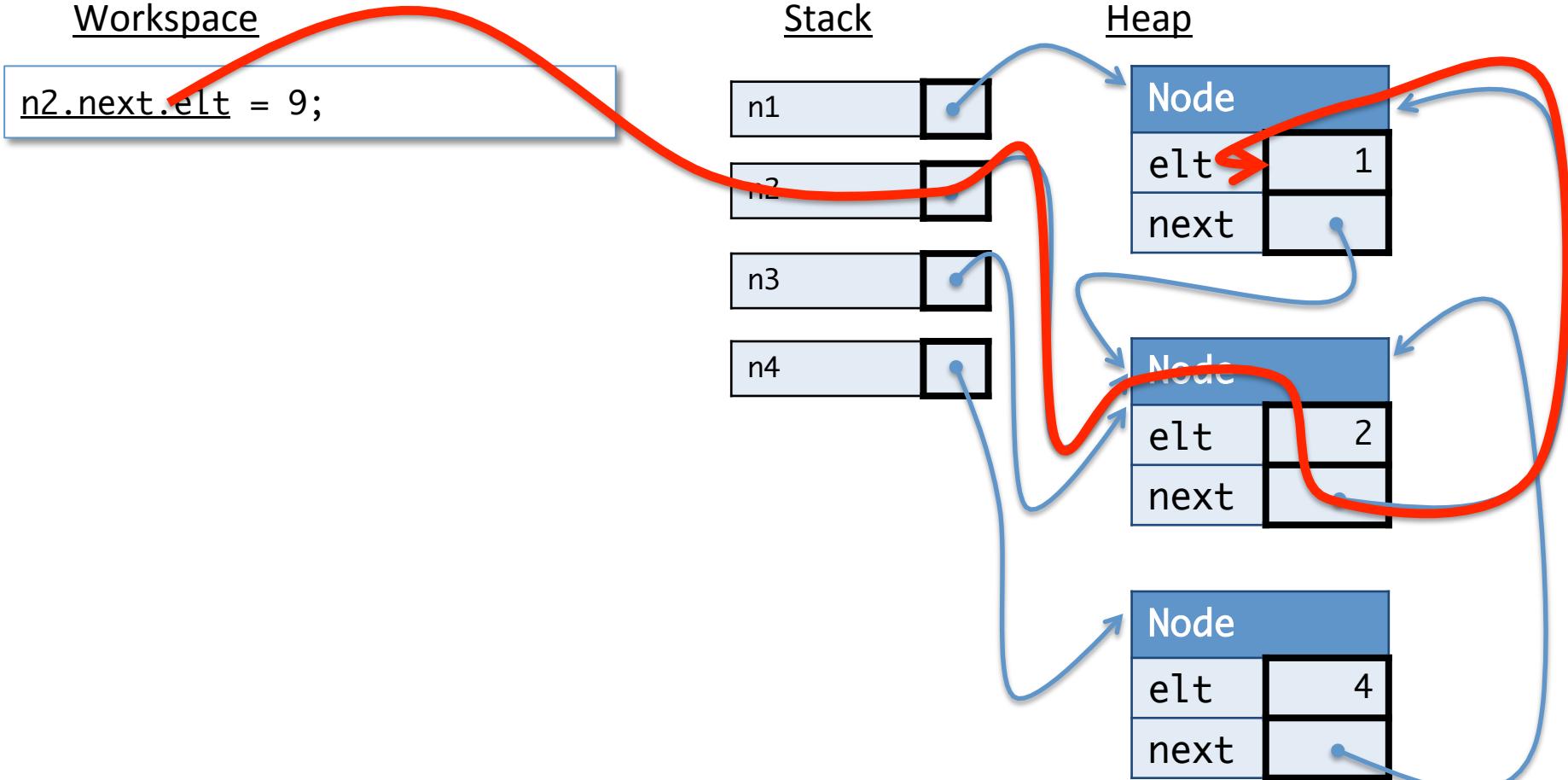
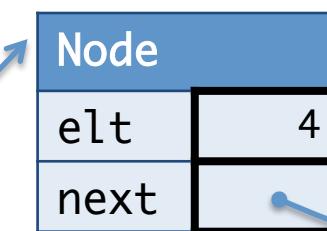
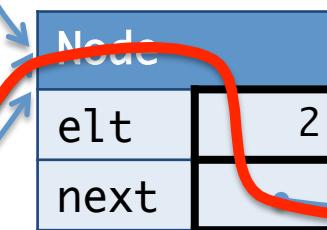
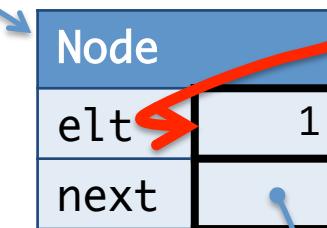
Workspace

```
n2.next.elt = 9;
```

Stack



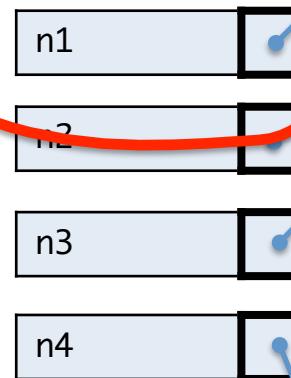
Heap



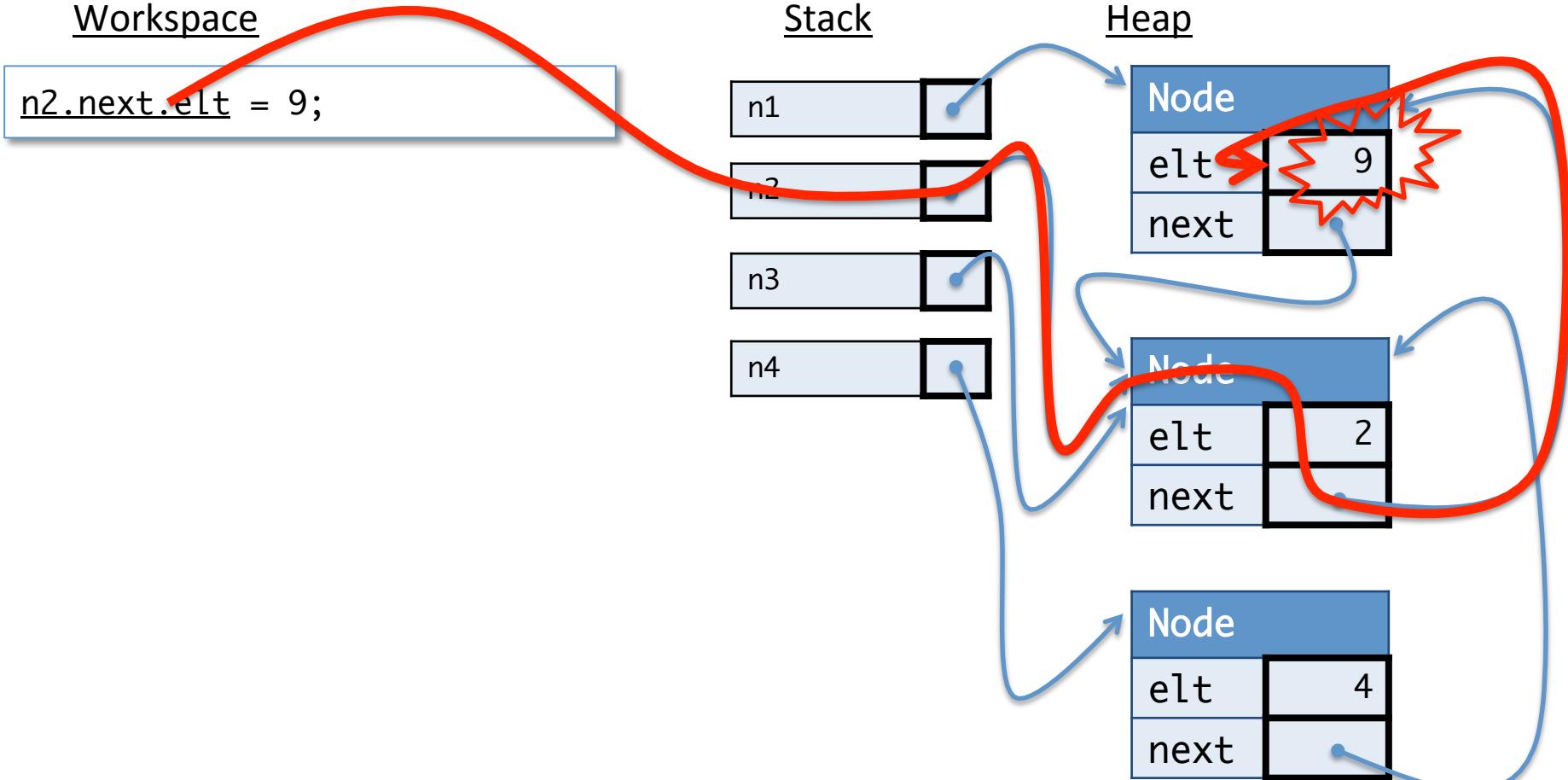
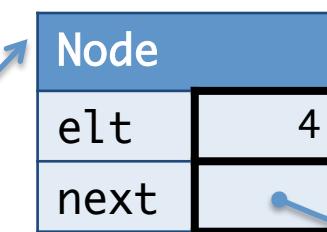
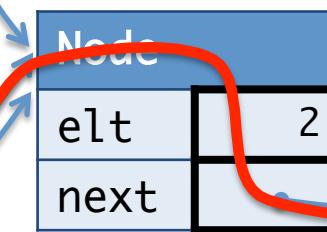
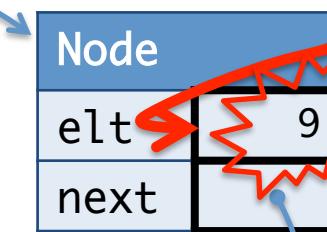
Workspace

```
n2.next.elt = 9;
```

Stack



Heap





OOoooooo programming



OO
Subtypes

Interfaces

- Give a type for an object based on what it *does*, not on how it was constructed
- Describes a contract that objects must satisfy
- Example: Interface for objects that have a position and can be moved

```
public interface Displaceable {  
    public int getX();  
    public int getY();  
    public void move(int dx, int dy);  
}
```

keyword

No fields, no constructors, no method bodies!

Implementing the interface

- A class that implements an interface provides appropriate definitions for the methods specified in the interface
- That class fulfills the contract implicit in the interface

methods required to satisfy contract

```
public class Point implements Displaceable {  
    private int x, y;  
    public Point(int x0, int y0) {  
        x = x0;  
        y = y0;  
    }  
    public int getX() { return x; }  
    public int getY() { return y; }  
    public void move(int dx, int dy) {  
        x = x + dx;  
        y = y + dy;  
    }  
}
```



interfaces implemented

Another implementation

```
public class Circle implements Displaceable {  
    private Point center;  
    private int radius;  
    public Circle(Point initCenter, int initRadius) {  
        center = initCenter;  
        radius = initRadius;  
    }  
    public int getX() { return center.getX(); }  
    public int getY() { return center.getY(); }  
    public void move(int dx, int dy) {  
        center.move(dx, dy);  
    }  
}
```

Objects with different local state can satisfy the same interface

Delegation: move the circle by moving the center

The following snippet of code typechecks:

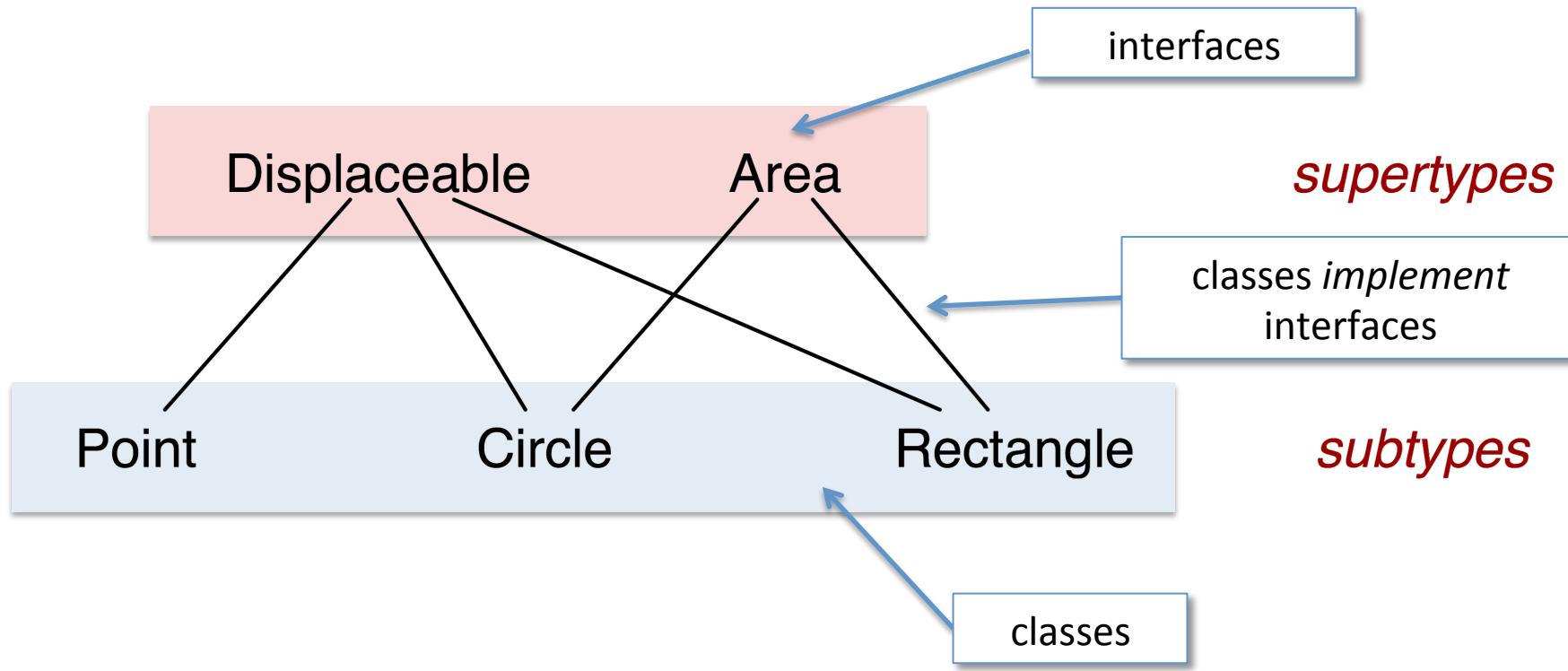
```
public void moveItALot (Displaceable s) {  
    ... //omitted  
}  
  
... // elsewhere  
Circle c = new Circle(10,10,10);  
moveItAlot(c);
```

1. True
2. False

Answer: True

Subtypes and Supertypes

- An interface represents a *point of view* about an object
- Classes can implement *multiple* interfaces



Types can have many different supertypes / subtypes

Extension

1. Interface extension
2. Class extension (Simple inheritance)

Interface Extension

- Build richer interface hierarchies by *extending* existing interfaces.

```
public interface Displaceable {  
    double getX();  
    double getY();  
    void move(double dx, double dy);  
}
```

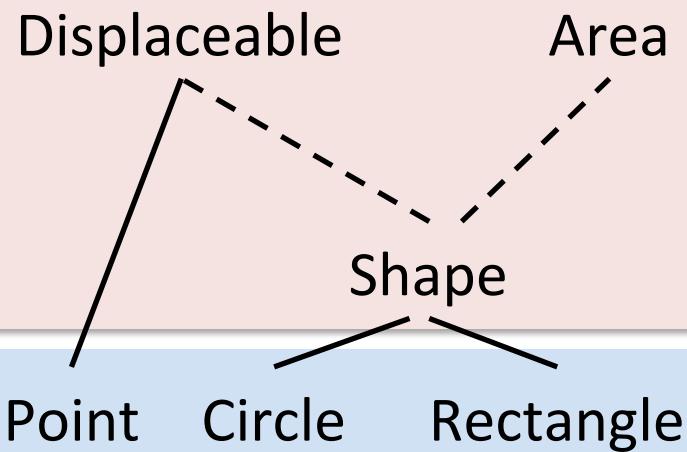
```
public interface Area {  
    double getArea();  
}
```

```
public interface Shape extends Displaceable, Area {  
    Rectangle getBoundingBox();  
}
```

The Shape type includes all the methods of Displaceable and Area, plus the new getBoundingBox method.

Note the use of the “extends” keyword.

Interface Hierarchy



```
class Point implements Displaceable {
    ... // omitted
}

class Circle implements Shape {
    ... // omitted
}

class Rectangle implements Shape {
    ... // omitted
}
```

- Shape is a *subtype* of both Displaceable and Area.
- Circle and Rectangle are both subtypes of Shape, and, by *transitivity*, both are also subtypes of Displaceable and Area.
- Note that one interface may extend *several* others.
 - Interfaces do not necessarily form a tree, but the hierarchy has no cycles.

Class Extension: Inheritance

- Classes, like interfaces, can also extend one another.
 - Unlike interfaces, a class can extend only *one* other class.
- The extending class *inherits* all of the fields and methods of its *superclass*, may include additional fields or methods.
 - This captures the “is a” relationship between objects (e.g. a Car is a Vehicle).
 - Class extension should *never* be used when “is a” does not relate the subtype to the supertype.

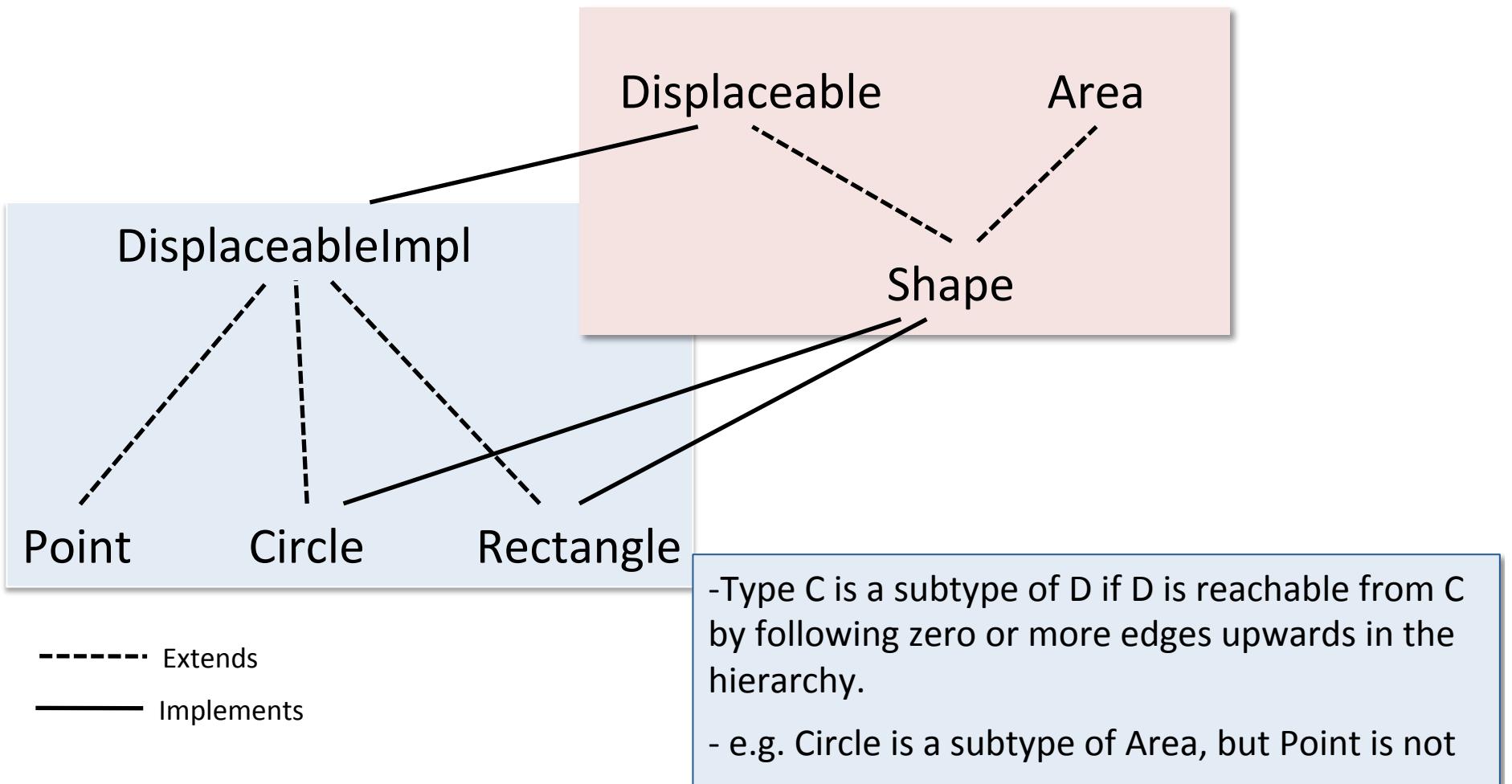
Class Extension: Inheritance

```
public class DisplaceableImpl implements Displaceable {  
    private Point pt;  
    public DisplaceableImpl(int x, int y) {  
        this.pt = new Point(x,y);  
    }  
    public int getX() { return pt.getX(); }  
    public int getY() { return pt.getY(); }  
    public void move(int dx, int dy) {  
        pt.move(dx, dy);}  
    }  
  
public class Circle extends DisplaceableImpl {  
    private int radius;  
  
    public Circle(int x, int y, int radius) {  
        super(x,y);  
        this.radius = radius;  
    }  
  
    public int getRadius() { return radius; }  
}
```

Simple Inheritance

- In *simple inheritance*, the subclass only *adds* new fields or methods.
- Use simple inheritance to *share common code* among related classes.
- Example: Circle, and Rectangle have *identical* code for getX(), getY(), and move() methods when implementing Displaceable.

Subtyping with Inheritance



Example of Simple Inheritance

See: Main2.java

Inheritance: Constructors

- Constructors *cannot* be inherited
 - They have the wrong names!
 - A subclass invokes the constructor of its super class using the keyword `super`.
 - `Super` *must* be the first line of the subclass constructor
 - if the parent class constructor takes no arguments, it is OK to omit the call to `super`.
 - It is then called implicitly.

Class Extension: Inheritance

```
public class DisplaceableImpl implements Displaceable {  
    private Point pt;  
    public DisplaceableImpl(int x, int y) {  
        this.pt = new Point(x,y);  
    }  
    public int getX() { return pt.getX(); }  
    public int getY() { return pt.getY(); }  
    public void move(int dx, int dy) {  
        pt.move(dx, dy);}  
    }  
  
public class Circle extends DisplaceableImpl {  
    private int radius;  
  
    public Circle(int x, int y, int radius) {  
        super(x,y);  
        this.radius = radius;  
    }  
  
    public int getRadius() { return radius; }  
}
```

Other forms of inheritance

- Java has other features related to inheritance (some of which we will discuss later in the course):
 - A subclass might *override* (re-implement) a method already found in the superclass.
 - A class might be *abstract* – i.e. it does not provide implementations for all of its methods (its subclasses must provide them instead)
- These features are hard to use properly, and the need for them arises only in somewhat special cases
 - Making reusable libraries
 - Special methods: equals and toString
- We recommend avoiding *all* forms of inheritance (even “simple inheritance”) when possible – prefer interfaces and composition.

Especially: avoid overriding.

Subtype Polymorphism*

- Main idea:

Anywhere an object of type A is needed, an object that is a subtype of A can be provided.

```
// in class C
public static void leapIt(DisplaceableImpl c) {
    c.move(1000,1000);
}
// somewhere else
C.leapIt(new Circle (10));
```

- If B is a subtype of A, it provides all of A's (public) methods.
- Due to dynamic dispatch, the behavior of a method depends on B's implementation.
 - Simple inheritance means B's method is inherited from A
 - Otherwise, behavior of B should be "compatible" with A's behavior

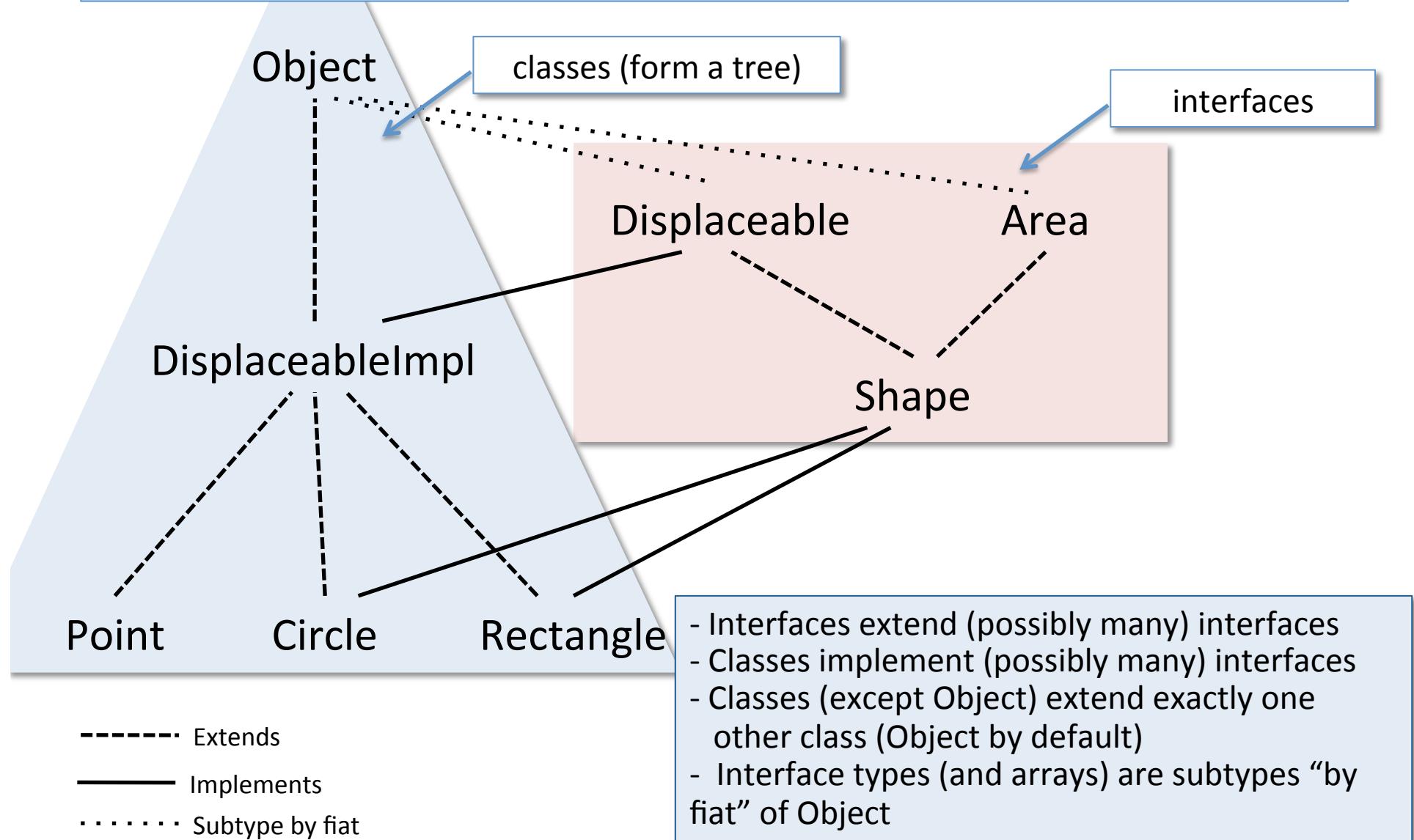
*polymorphism = many shapes

Object

```
public class Object {  
    boolean equals(Object o) {  
        ... // test for equality  
    }  
    String toString() {  
        ... // return a string representation  
    }  
    ... // other methods omitted  
}
```

- Object is the root of the class tree.
 - Classes that leave off the “extends” clause *implicitly* extend Object
 - Arrays also implement the methods of Object
 - This class provides methods useful for *all* objects to support
- Object is the highest type in the subtyping hierarchy.

Recap: Subtyping



Programming Languages and Techniques (CIS120)

Lecture 25

November 1, 2017

Inheritance and Dynamic Dispatch
(Chapter 24)

Announcements

- HW7: Chat Client
 - Available Soon
 - Due: Tuesday, November 14th at 11:59pm
- Upcoming: Midterm 2
 - Friday, November 10th in class
- Covers:
 - Mutable state (in OCaml and Java)
 - Objects (in OCaml and Java)
 - ASM (in OCaml and Java)
 - Reactive programming (in OCaml)
 - Arrays (in Java)
 - Subtyping & Simple Extension (in Java)



OOoooooo programming



OO
Subtypes

Interfaces

- Give a type for an object based on what it *does*, not on how it was constructed
- Describes a contract that objects must satisfy
- Example: Interface for objects that have a position and can be moved

```
public interface Displaceable {  
    public int getX();  
    public int getY();  
    public void move(int dx, int dy);  
}
```

keyword

No fields, no constructors, no method bodies!

Implementing the interface

- A class that implements an interface provides appropriate definitions for the methods specified in the interface
- That class fulfills the contract implicit in the interface

methods required to satisfy contract

```
public class Point implements Displaceable {  
    private int x, y;  
    public Point(int x0, int y0) {  
        x = x0;  
        y = y0;  
    }  
    public int getX() { return x; }  
    public int getY() { return y; }  
    public void move(int dx, int dy) {  
        x = x + dx;  
        y = y + dy;  
    }  
}
```



interfaces implemented

Another implementation

```
public class Circle implements Displaceable {  
    private Point center;  
    private int radius;  
    public Circle(Point initCenter, int initRadius) {  
        center = initCenter;  
        radius = initRadius;  
    }  
    public int getX() { return center.getX(); }  
    public int getY() { return center.getY(); }  
    public void move(int dx, int dy) {  
        center.move(dx, dy);  
    }  
}
```

Objects with different local state can satisfy the same interface

Delegation: move the circle by moving the center

The following snippet of code typechecks:

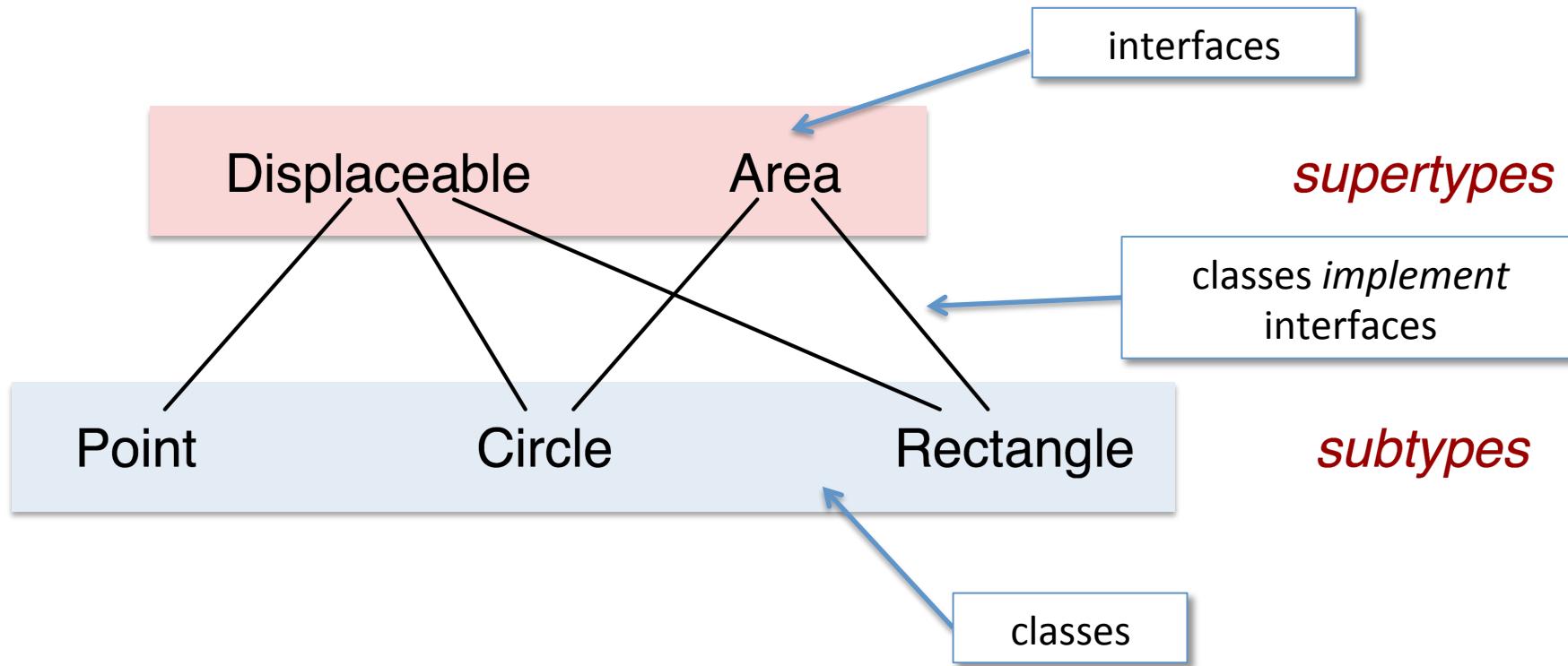
```
public void moveItALot (Displaceable s) {  
    ... //omitted  
}  
  
... // elsewhere  
Circle c = new Circle(10,10,10);  
moveItAlot(c);
```

1. True
2. False

Answer: True

Subtypes and Supertypes

- An interface represents a *point of view* about an object
- Classes can implement *multiple* interfaces



Types can have many different supertypes / subtypes

Extension

1. Interface extension
2. Class extension (Simple inheritance)

Class Extension: Inheritance

```
public class DisplaceableImpl implements Displaceable {  
    private double x;  
    private double y;  
    public DisplaceableImpl(double x, double y) {  
        this.x = x; this.y = y;  
    }  
    public double getX() { return x; }  
    public double getY() { return y; }  
    public void move(double dx, double dy) {  
        x = x + dx; y = y + dy;  
    }  
}  
  
public class Circle extends DisplaceableImpl {  
    private int radius;  
  
    public Circle(int x, int y, int radius) {  
        super(x,y);  
        this.radius = radius;  
    }  
  
    public int getRadius() { return radius; }  
}
```

Interface Extension

- Build richer interface hierarchies by *extending* existing interfaces.

```
public interface Displaceable {  
    double getX();  
    double getY();  
    void move(double dx, double dy);  
}
```

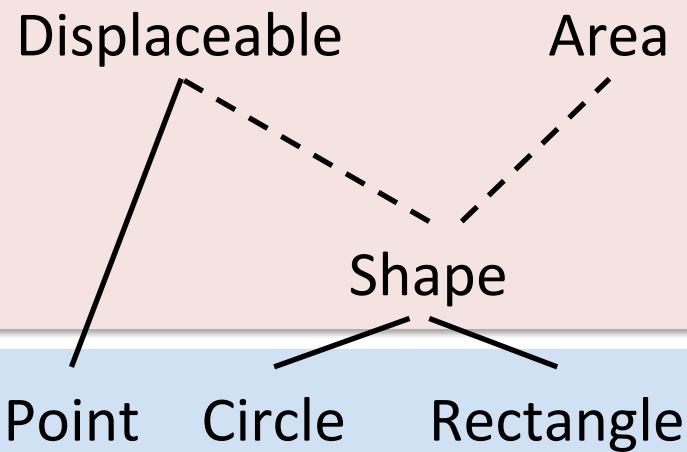
```
public interface Area {  
    double getArea();  
}
```

```
public interface Shape extends Displaceable, Area {  
    Rectangle getBoundingBox();  
}
```

The Shape type includes all the methods of Displaceable and Area, plus the new getBoundingBox method.

Note the use of the “extends” keyword.

Interface Hierarchy



```
class Point implements Displaceable {
    ... // omitted
}

class Circle implements Shape {
    ... // omitted
}

class Rectangle implements Shape {
    ... // omitted
}
```

- Shape is a *subtype* of both Displaceable and Area.
- Circle and Rectangle are both subtypes of Shape, and, by *transitivity*, both are also subtypes of Displaceable and Area.
- Note that one interface may extend *several* others.
 - Interfaces do not necessarily form a tree, but the hierarchy has no cycles.

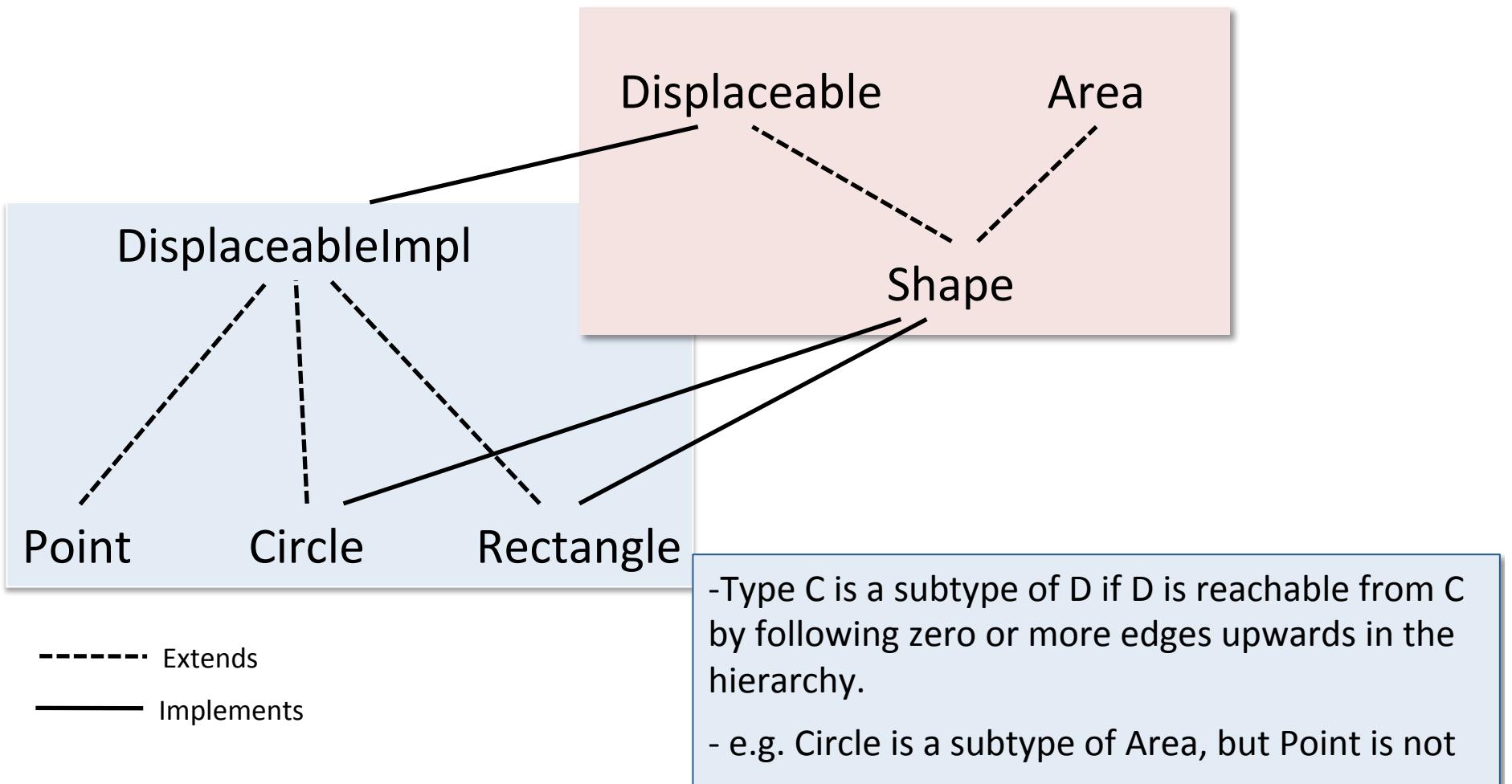
Class Extension: Inheritance

- Classes, like interfaces, can also extend one another.
 - Unlike interfaces, a class can extend only *one* other class.
- The extending class *inherits* all of the fields and methods of its *superclass*, may include additional fields or methods.
 - This captures the “is a” relationship between objects (e.g. a Car is a Vehicle).
 - Class extension should *never* be used when “is a” does not relate the subtype to the supertype.

Simple Inheritance

- In *simple inheritance*, the subclass only *adds* new fields or methods.
- Use simple inheritance to *share common code* among related classes.
- Example: Circle, and Rectangle have *identical* code for getX(), getY(), and move() methods when implementing Displaceable.

Subtyping with Inheritance



Example of Simple Inheritance

See: Shapes.zip

Inheritance: Constructors

- Constructors *cannot* be inherited
 - They have the wrong names!
 - A subclass invokes the constructor of its super class using the keyword `super`.
 - `Super` *must* be the first line of the subclass constructor
 - if the parent class constructor takes no arguments, it is OK to omit the call to `super`.
 - It is then called implicitly.

Class Extension: Inheritance

```
public class DisplaceableImpl implements Displaceable {  
    private double x;  
    private double y;  
    public DisplaceableImpl(double x, double y) {  
        this.x = x; this.y = y;  
    }  
    public double getX() { return x; }  
    public double getY() { return y; }  
    public void move(double dx, double dy) {  
        x = x + dx; y = y + dy;  
    }  
}  
  
public class Circle extends DisplaceableImpl {  
    private int radius;  
  
    public Circle(int x, int y, int radius) {  
        super(x,y);  
        this.radius = radius;  
    }  
  
    public int getRadius() { return radius; }  
}
```

Other forms of inheritance

- Java has other features related to inheritance (some of which we will discuss later in the course):
 - A subclass might *override* (re-implement) a method already found in the superclass.
 - A class might be *abstract* – i.e. it does not provide implementations for all of its methods (its subclasses must provide them instead)
- These features are hard to use properly, and the need for them arises only in somewhat special cases
 - Making reusable libraries
 - Special methods: equals and toString
- We recommend avoiding *all* forms of inheritance (even “simple inheritance”) when possible – prefer interfaces and composition.

Especially: avoid overriding.

Subtype Polymorphism*

- Main idea:

Anywhere an object of type A is needed, an object that is a subtype of A can be provided.

```
// in class C
public static void leapIt(DisplaceableImpl c) {
    c.move(1000,1000);
}
// somewhere else
C.leapIt(new Circle (10));
```

- If B is a subtype of A, it provides all of A's (public) methods.
- Due to dynamic dispatch, the behavior of a method depends on B's implementation.
 - Simple inheritance means B's method is inherited from A
 - Otherwise, behavior of B should be "compatible" with A's behavior

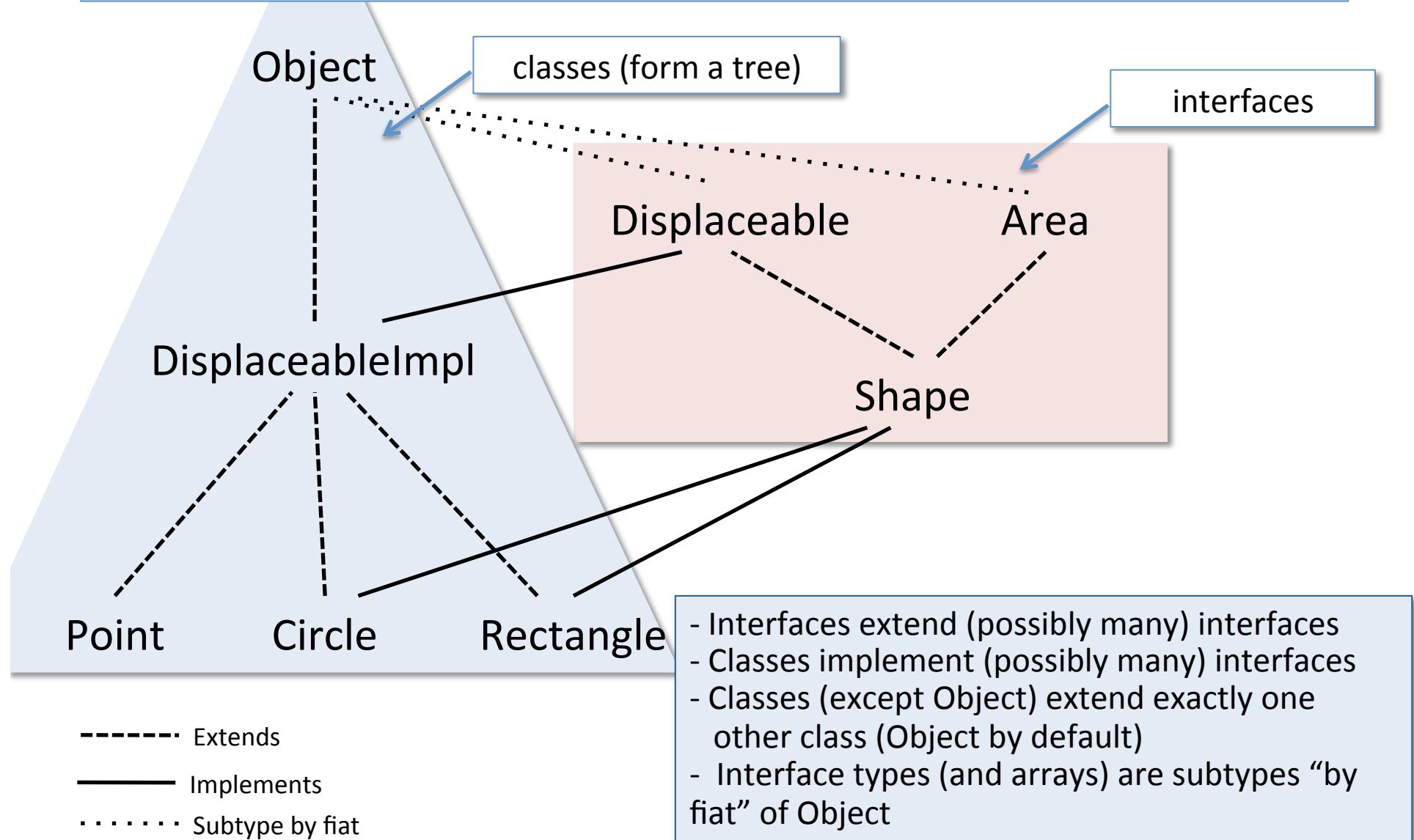
*polymorphism = many shapes

Object

```
public class Object {  
    boolean equals(Object o) {  
        ... // test for equality  
    }  
    String toString() {  
        ... // return a string representation  
    }  
    ... // other methods omitted  
}
```

- Object is the root of the class tree.
 - Classes that leave off the “extends” clause *implicitly* extend Object
 - Arrays also implement the methods of Object
 - This class provides methods useful for *all* objects to support
- Object is the highest type in the subtyping hierarchy.

Recap: Subtyping



When do constructors execute?
How are fields accessed?
What code runs in a method call?

How do method calls work?

- What code gets run in a method invocation?
`o.move(3,4);`
- When that code is running, how does it access the fields of the object that invoked it?
`x = x + dx;`
- When does the code in a constructor get executed?
- What if the method was inherited from a superclass?

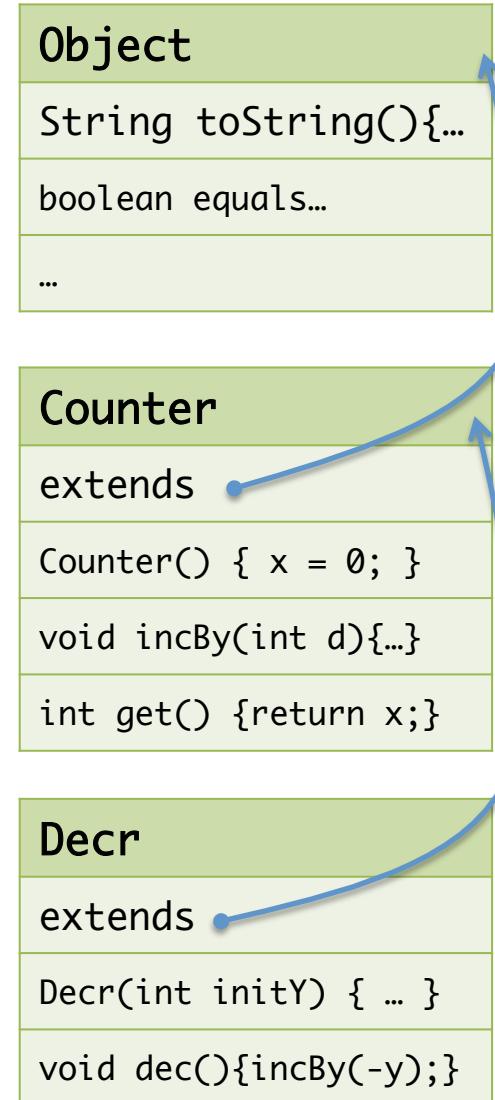
ASM refinement: The Class Table

```
public class Counter {  
    private int x;  
    public Counter () { x = 0; }  
    public void incBy(int d) { x = x + d; }  
    public int get() { return x; }  
}  
  
public class Decr extends Counter {  
    private int y;  
    public Decr (int initY) { y = initY; }  
    public void dec() { incBy(-y); }  
}
```

The class table contains:

- the code for each method,
- references to each class's parent, and
- the class's static members.

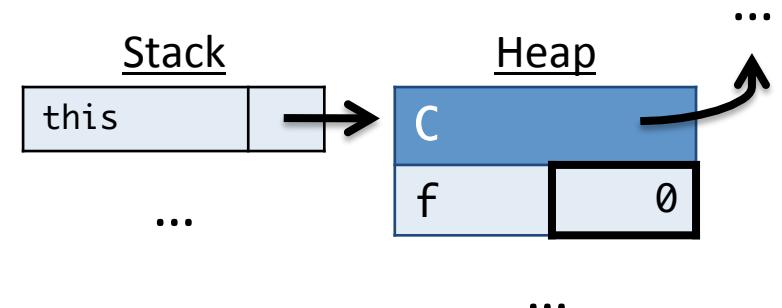
Class Table



this

- Inside a non-static method, the variable `this` is a reference to the object on which the method was invoked.
- References to local fields and methods have an implicit “`this.`” in front of them.

```
class C {  
    private int f;  
  
    public void copyF(C other) {  
        this.f = other.f;  
    }  
}
```



An Example

```
public class Counter {  
    private int x;  
    public Counter () { x = 0; }  
    public void incBy(int d) { x = x + d; }  
    public int get() { return x; }  
}  
  
public class Decr extends Counter {  
    private int y;  
    public Decr (int initY) { y = initY; }  
    public void dec() { incBy(-y); }  
}  
  
// ... somewhere in main:  
Decr d = new Decr(2);  
d.dec();  
int x = d.get();
```

...with Explicit `this` and `super`

```
public class Counter extends Object {  
    private int x;  
    public Counter () { super(); this.x = 0; }  
    public void incBy(int d) { this.x = this.x + d; }  
    public int get() { return this.x; }  
}  
  
public class Decr extends Counter {  
    private int y;  
    public Decr (int initY) { super(); this.y = initY; }  
    public void dec() { this.incBy(-this.y); }  
}  
  
// ... somewhere in main:  
Decr d = new Decr(2);  
d.dec();  
int x = d.get();
```

Constructing an Object

Workspace

```
Decr d = new Decr(2);  
d.dec();  
int x = d.get();
```

Stack

Heap

Class Table

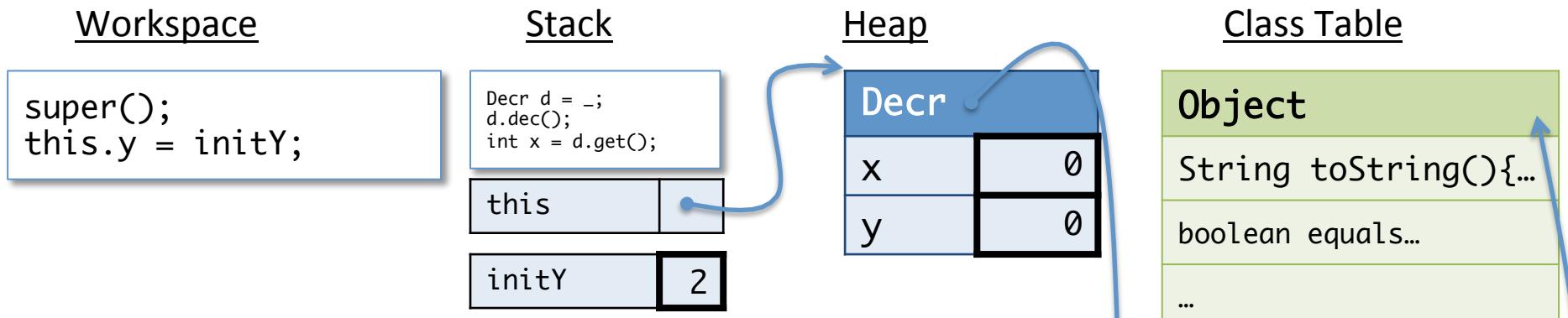
| |
|------------------------|
| Object |
| String toString(){...} |
| boolean equals... |
| ... |

| |
|------------------------|
| Counter |
| extends |
| Counter() { x = 0; } |
| void incBy(int d){...} |
| int get() {return x;} |

| |
|-------------------------|
| Decr |
| extends |
| Decr(int initY) { ... } |
| void dec(){incBy(-y);} |



Allocating Space on the Heap



Invoking a constructor:

- allocates space for a new object in the heap
- includes slots for *all* fields of *all* ancestors in the class tree (here: *x and y*)
- creates a pointer to the class – this is the object's dynamic type
- runs the constructor body after pushing parameters and *this* onto the stack

Note: fields start with a “sensible” default

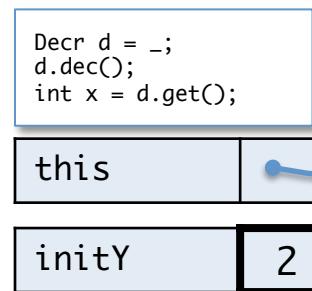
- 0 for numeric values
- null for references

Calling super

Workspace

```
super();  
this.y = initY;
```

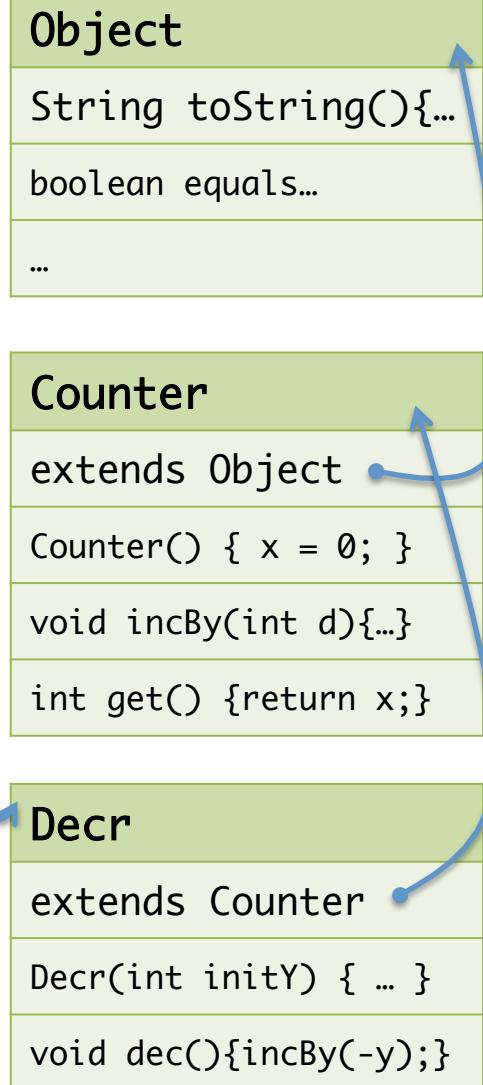
Stack



Heap



Class Table



Call to super:

- The constructor (implicitly) calls the super constructor
- Invoking a method/constructor pushes the saved workspace, the method params (none here) and a new `this` pointer.

Abstract Stack Machine

Workspace

```
super();  
this.x = 0;
```

Stack

```
Decr d = ...;  
d.dec();  
int x = d.get();  
  
this  
initY 2  
  
;  
this.y = initY;  
  
this
```

Heap

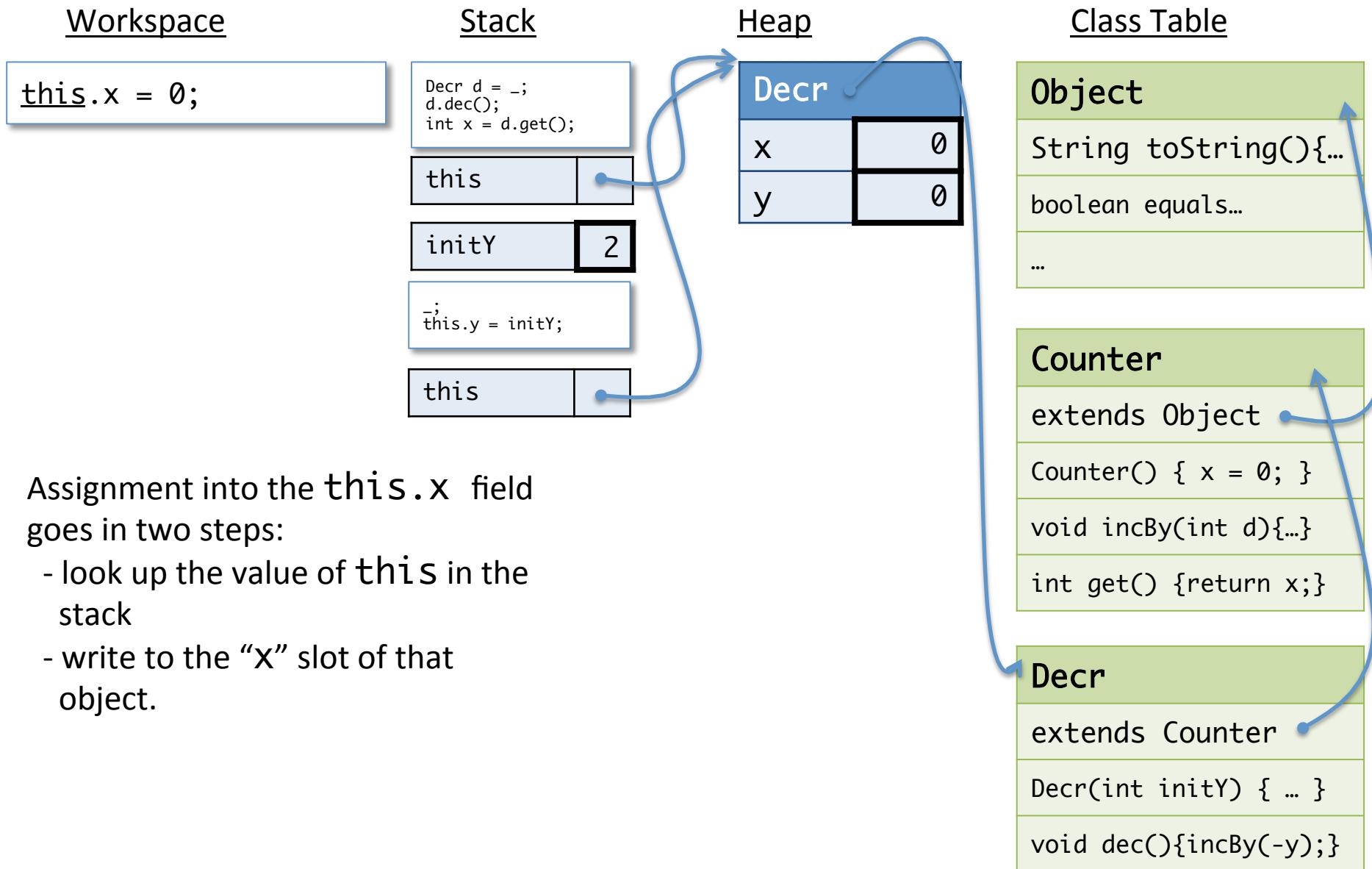
| | |
|------|---|
| Decr | |
| x | 0 |
| y | 0 |

Class Table

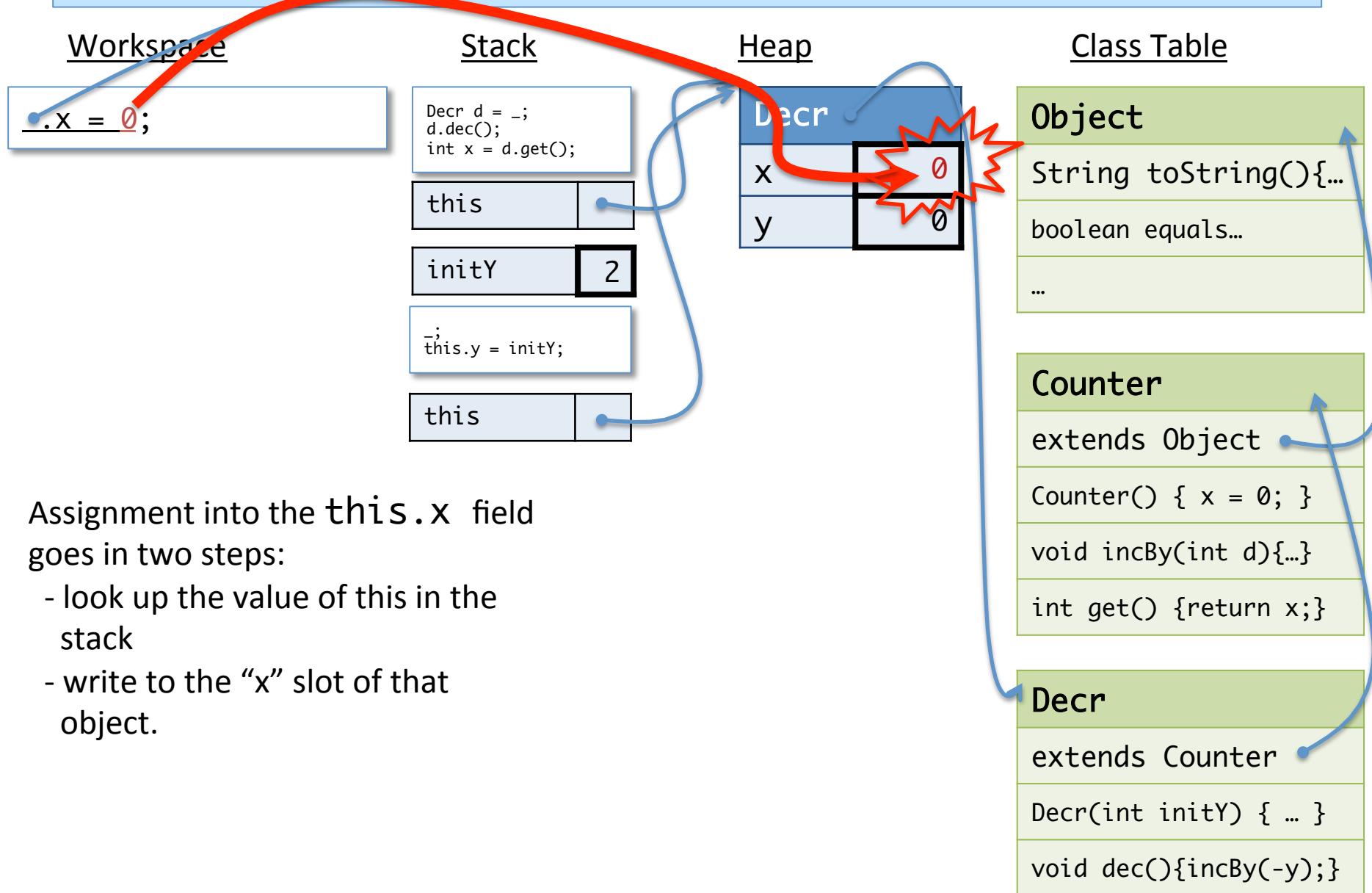
| | |
|-----------|-------------------|
| Object | |
| String | toString()... |
| boolean | equals... |
| ... | |
| Counter | |
| extends | Object |
| Counter() | { x = 0; } |
| void | incBy(int d){...} |
| int | get() {return x;} |
| Decr | |
| extends | Counter |
| Decr(int | initY) { ... } |
| void | dec(){incBy(-y);} |

(Running Object's default constructor omitted.)

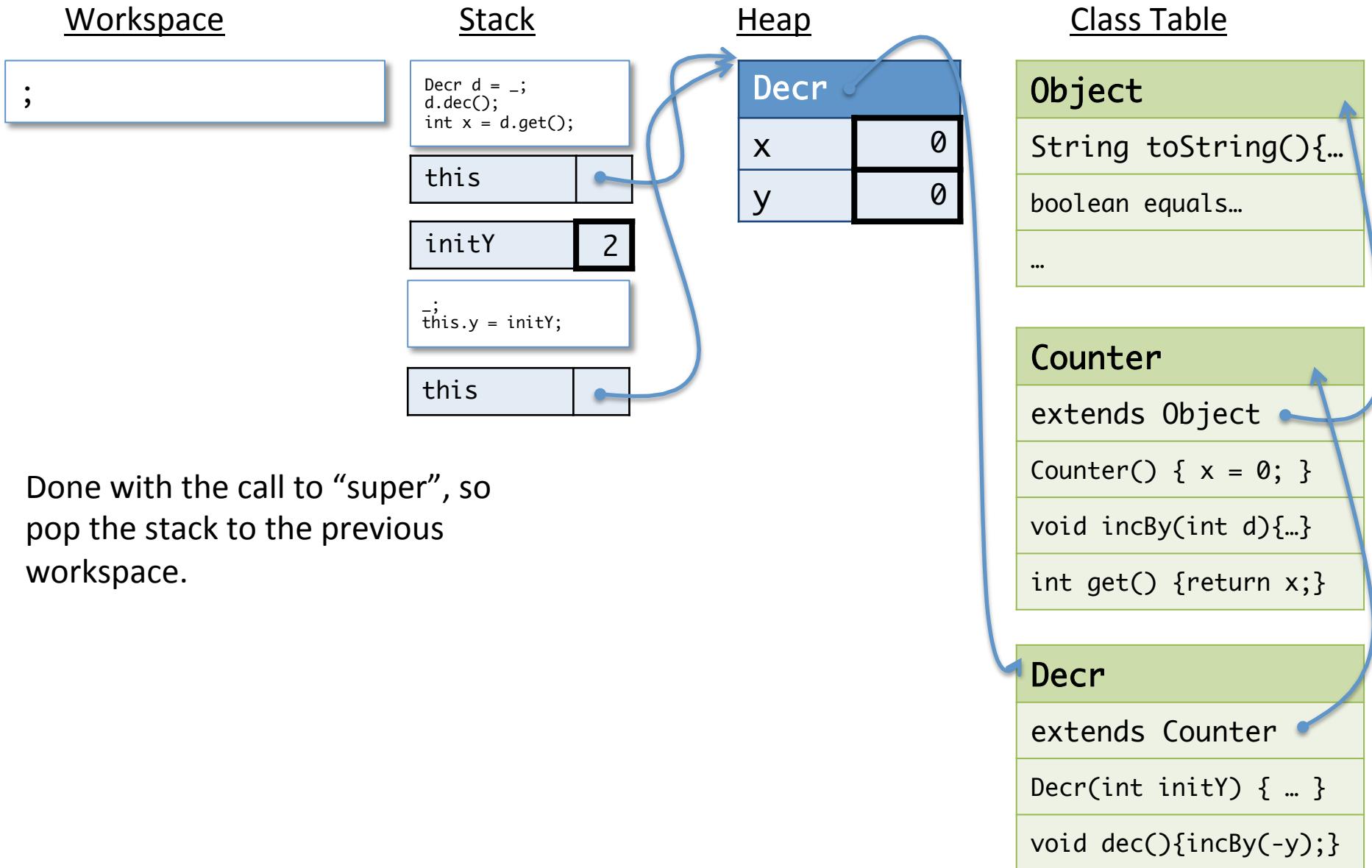
Assigning to a Field



Assigning to a Field



Done with the call



Continuing

Workspace

```
this.y = initY;
```

Stack

```
Decr d = ...;
d.dec();
int x = d.get();
```

| | |
|-------|---|
| this | |
| initY | 2 |

Heap

| | |
|------|---|
| Decr | |
| x | 0 |
| y | 0 |

Class Table

Object

```
String toString() { ... }
```

```
boolean equals...()
```

```
...
```

Counter

```
extends Object
```

```
Counter() { x = 0; }
```

```
void incBy(int d) { ... }
```

```
int get() { return x; }
```

Decr

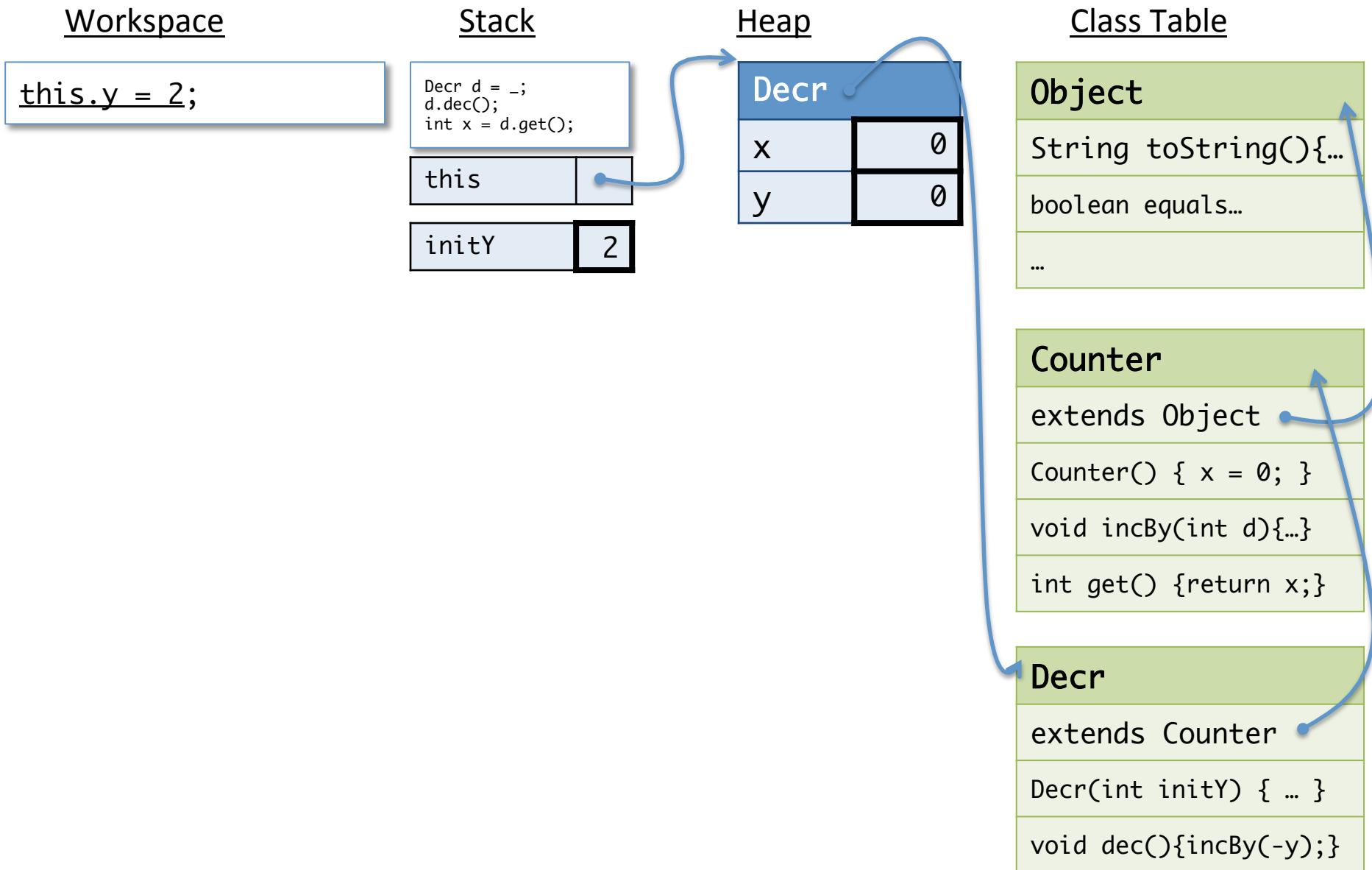
```
extends Counter
```

```
Decr(int initY) { ... }
```

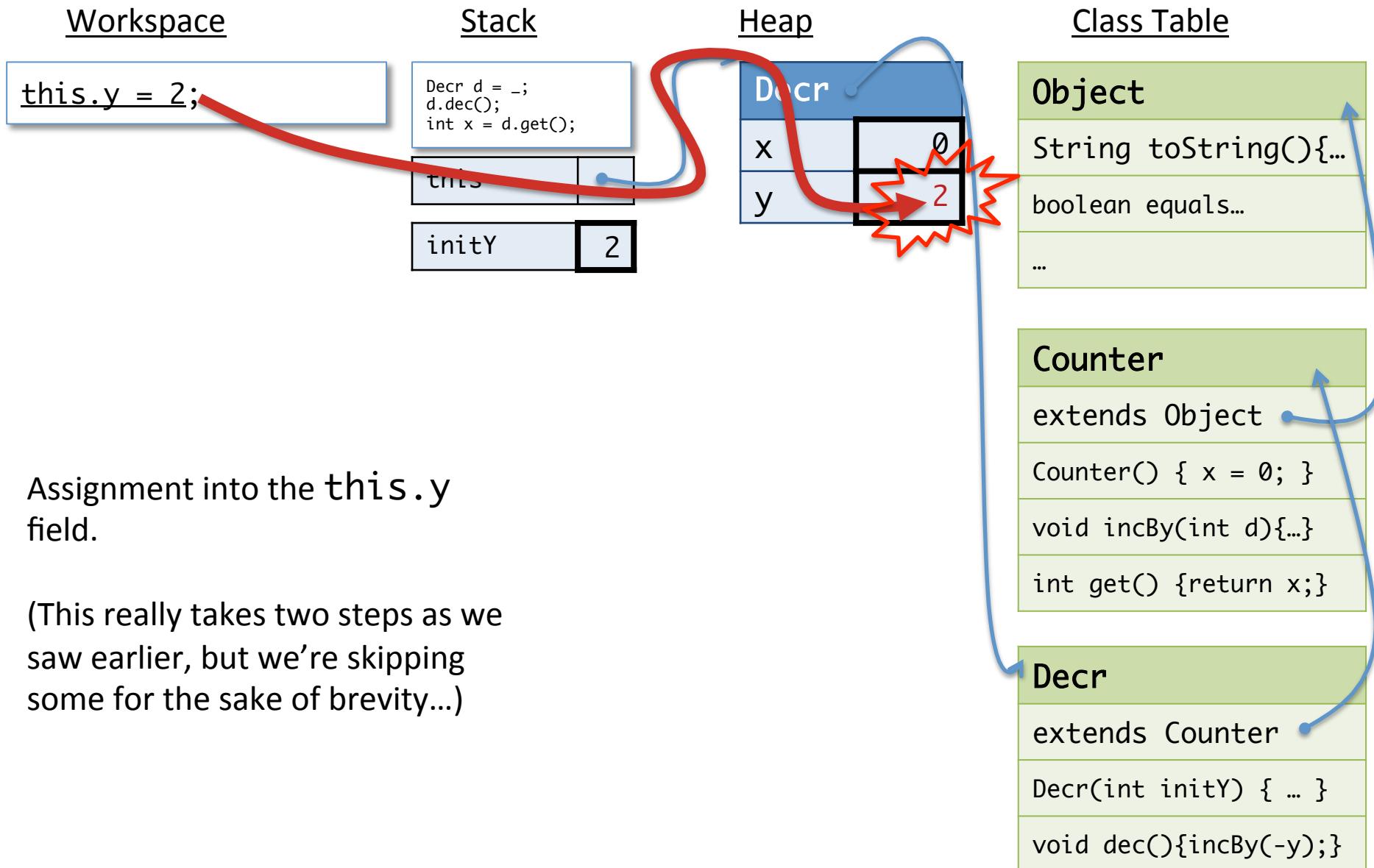
```
void dec() { incBy(-y); }
```

Continue in the Decr class's constructor.

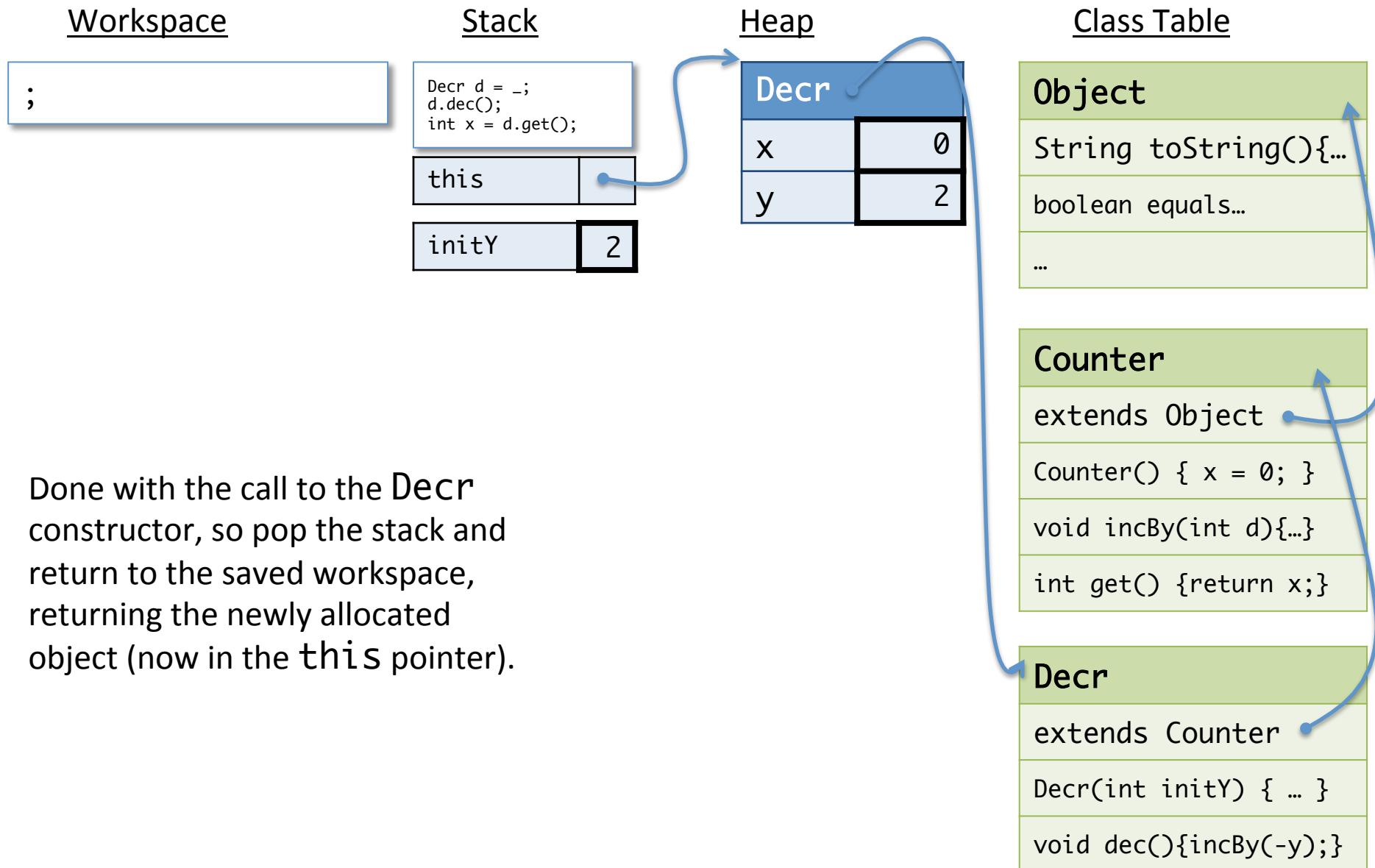
Abstract Stack Machine



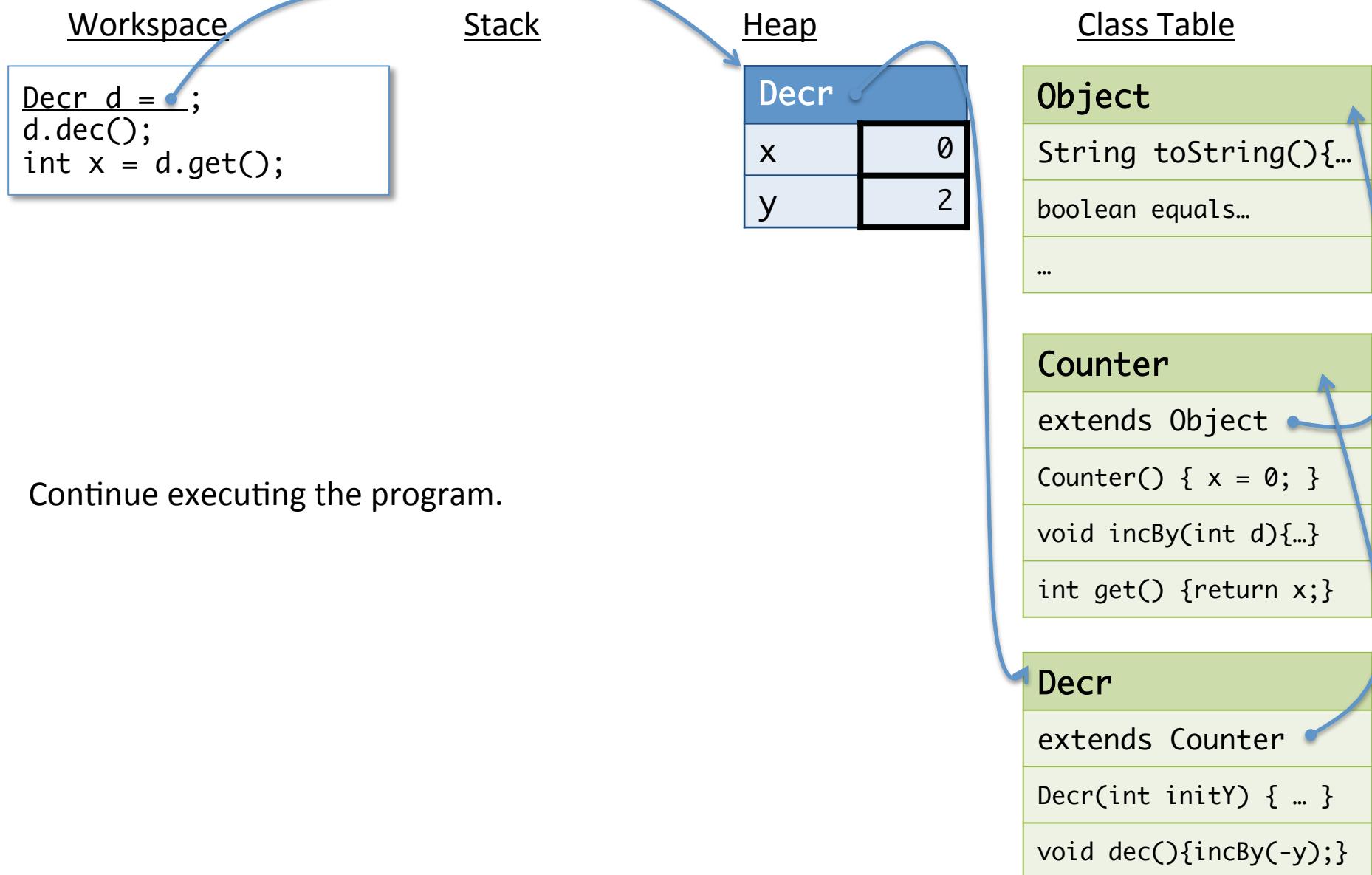
Assigning to a field



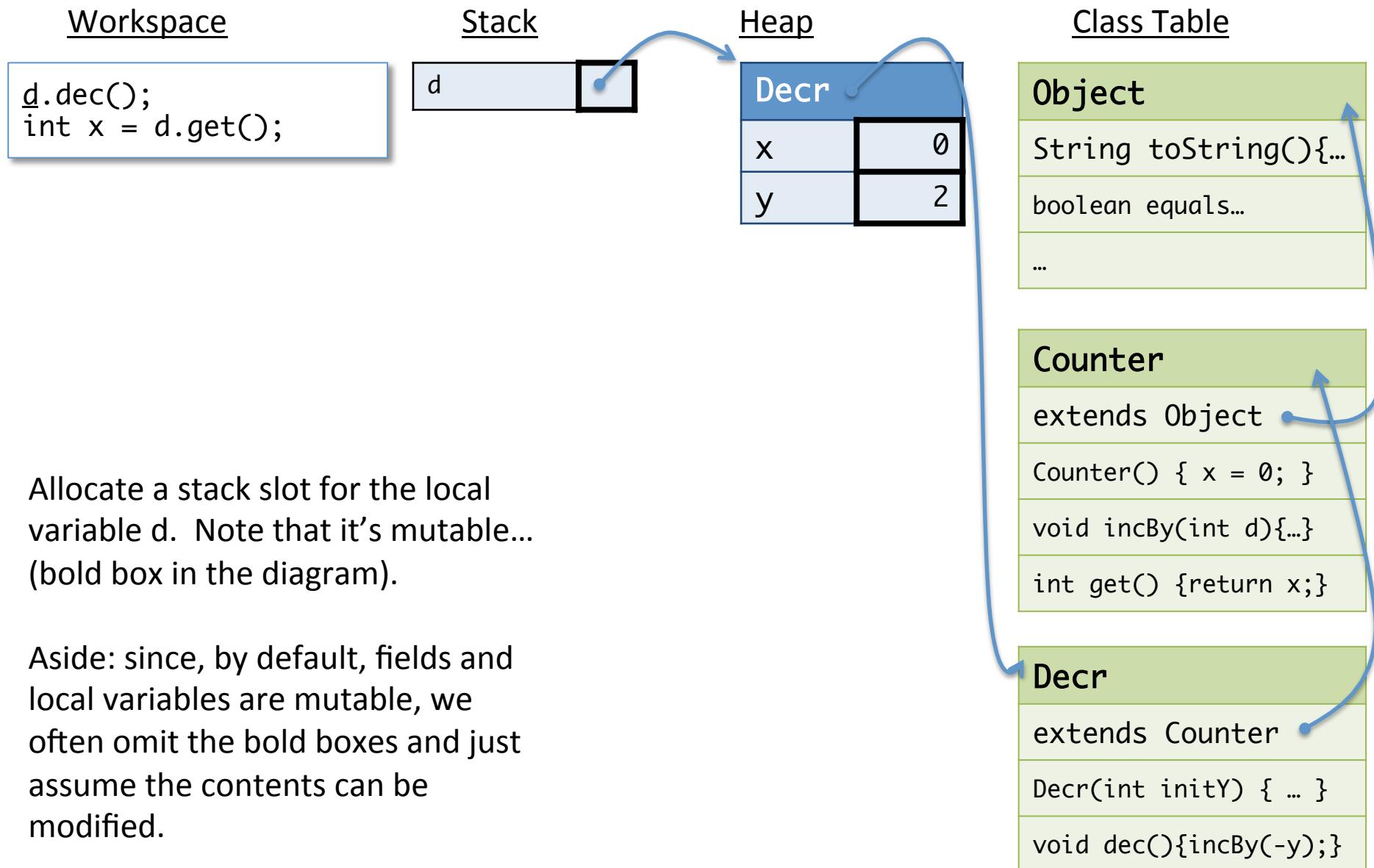
Done with the call



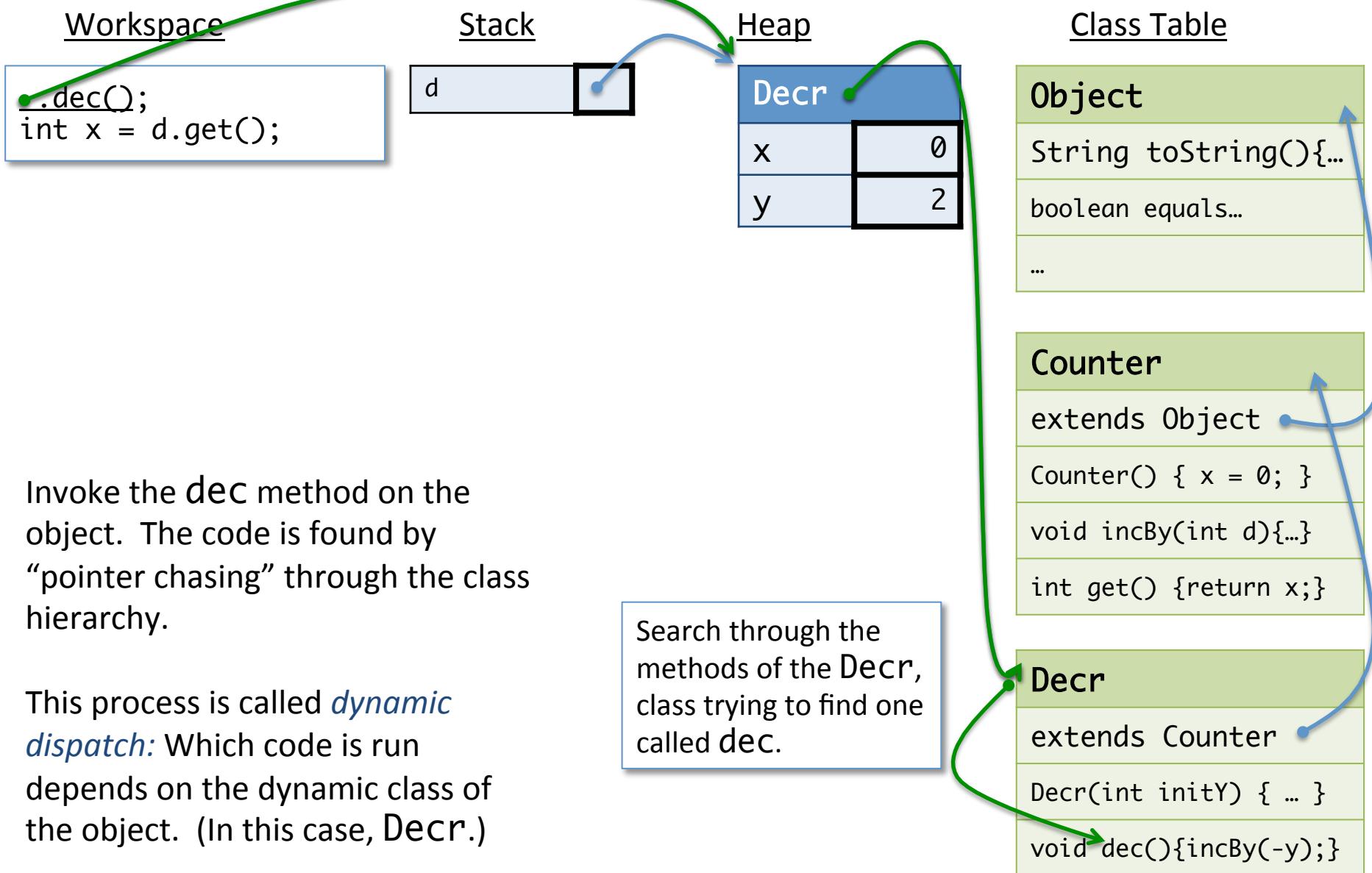
Returning the Newly Constructed Object



Allocating a local variable



Dynamic Dispatch: Finding the Code

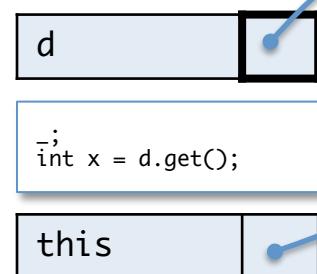


Dynamic Dispatch: Finding the Code

Workspace

```
this.incBy(-this.y);
```

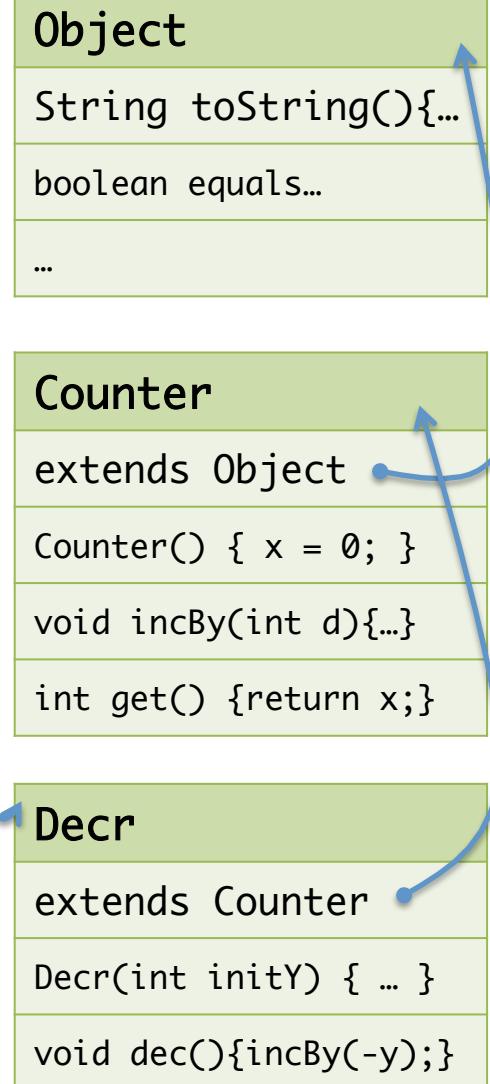
Stack



Heap

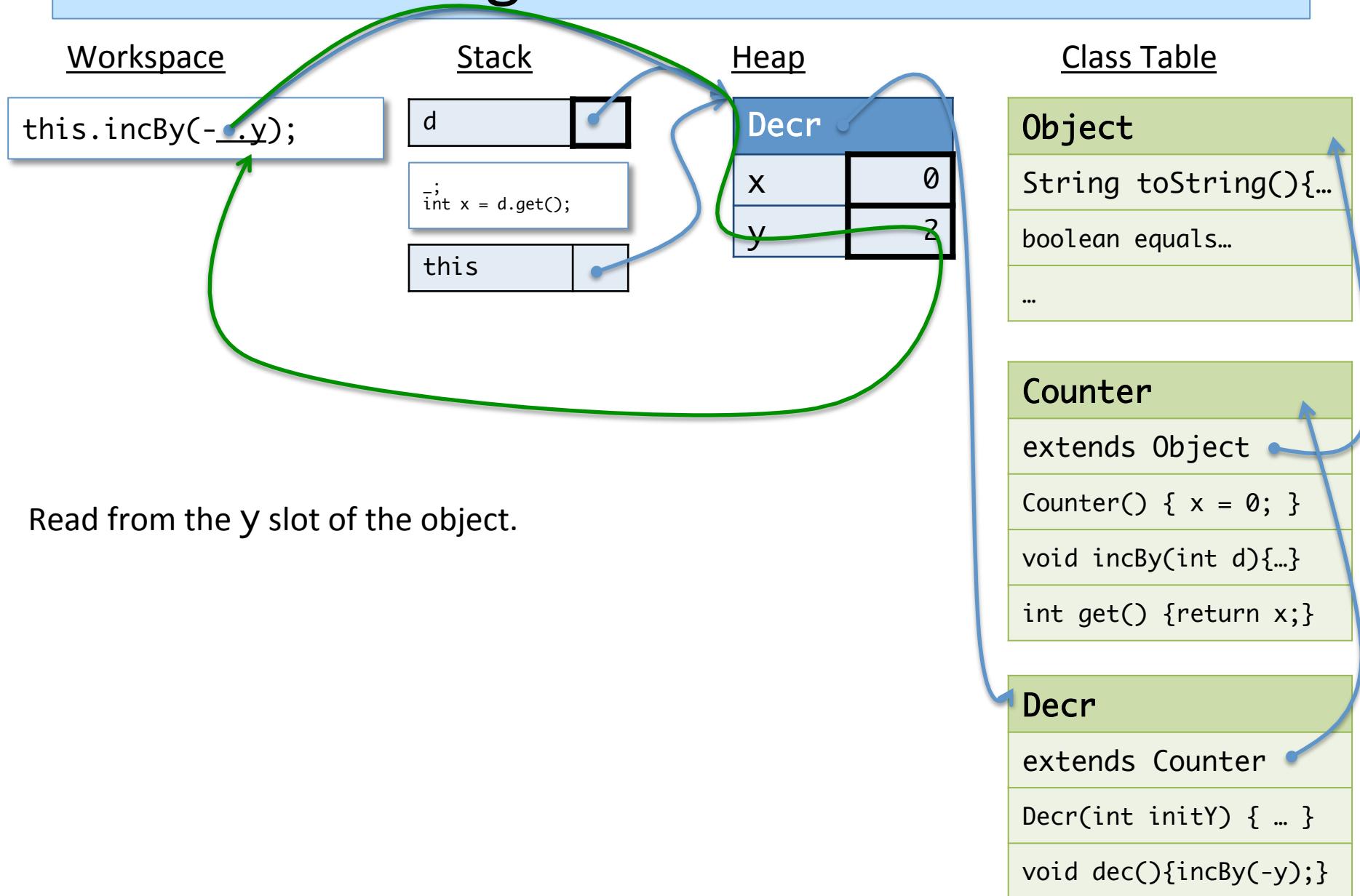
| | |
|------|---|
| Decr | |
| x | 0 |
| y | 2 |

Class Table

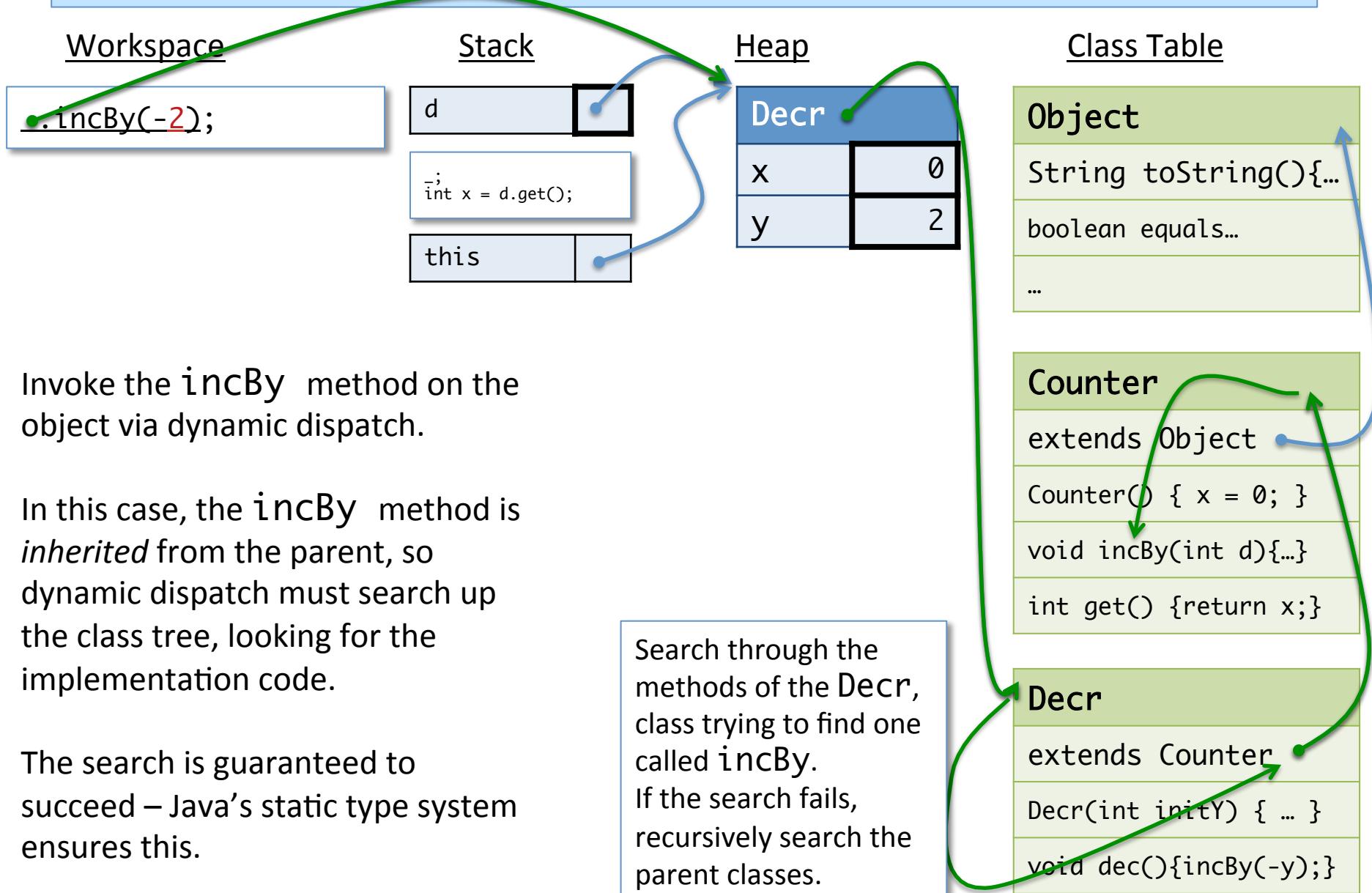


Call the method, remembering the current workspace and pushing the `this` pointer and any arguments (none in this case).

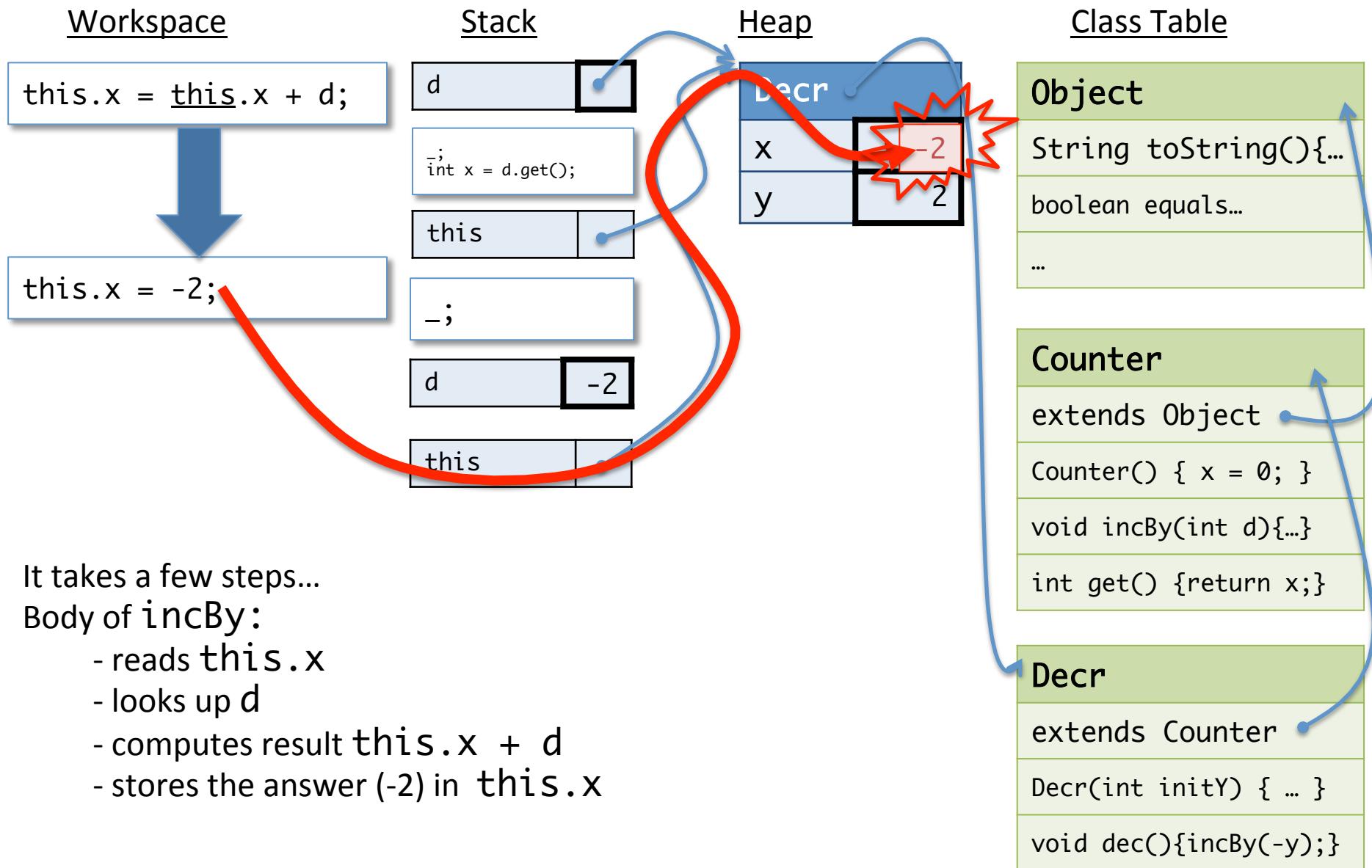
Reading A Field's Contents



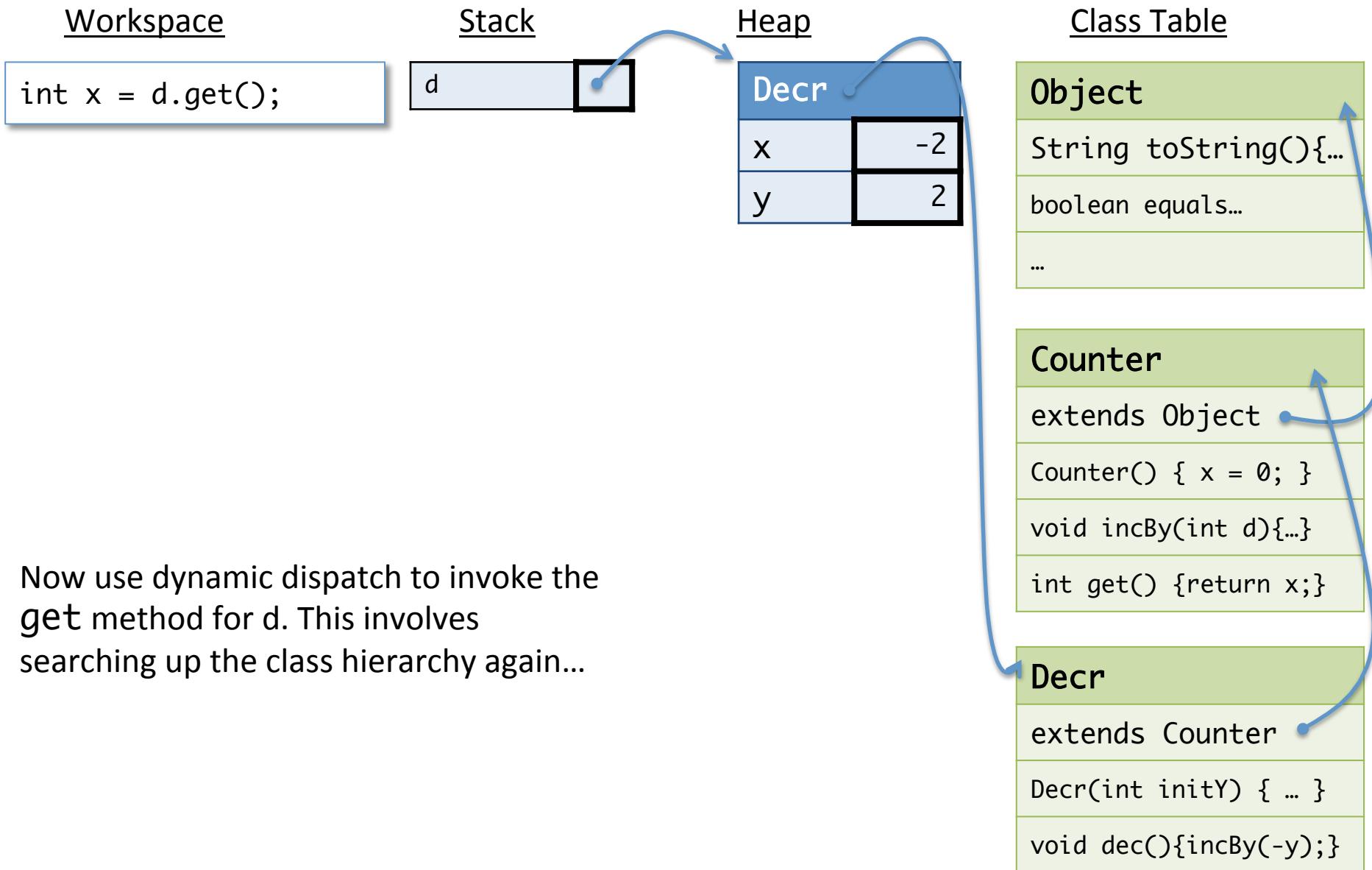
Dynamic Dispatch, Again



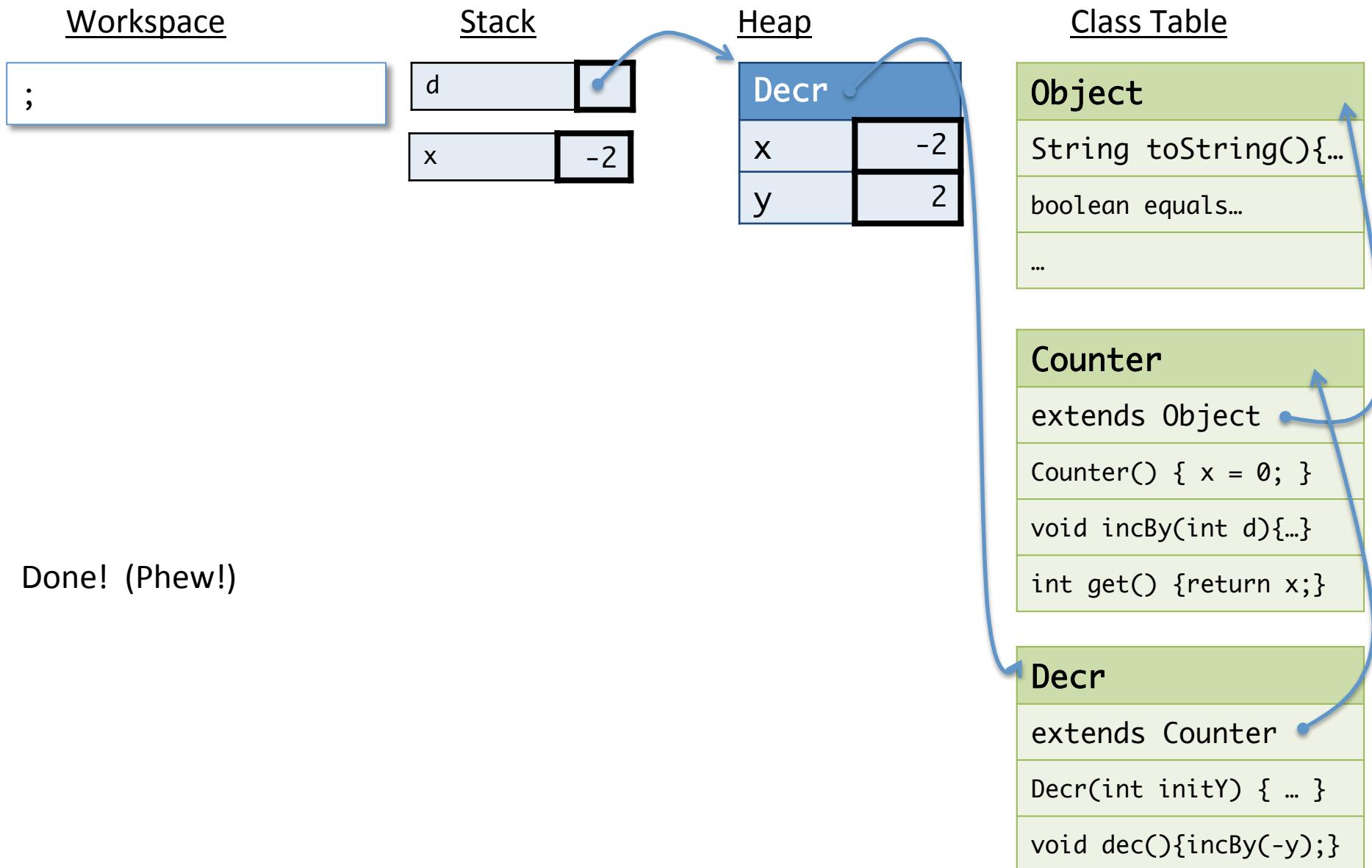
Running the body of incBy



After a few more steps...



After yet a few more steps...



Summary: `this` and dynamic dispatch

- When object's method is invoked, as in `o.m()`, the code that runs is determined by `o`'s *dynamic* class.
 - The dynamic class, represented as a pointer into the class table, is included in the object structure in the heap
 - If the method is inherited from a superclass, determining the code for `m` might require searching up the class hierarchy via pointers in the class table
 - This process of *dynamic dispatch* is the heart of OOP!
- Once the code for `m` has been determined, a binding for `this` is pushed onto the stack.
 - The `this` pointer is used to resolve field accesses and method invocations inside the code.

Refinements to the Stack Machine

- Code is stored in a *class table*, which is a special part of the heap:
 - When a program starts, the JVM initializes the class table
 - Each class has a pointer to its (unique) parent in the class tree
 - A class stores the constructor and method code for its instances
 - The class also stores *static* members
- Constructors:
 - Allocate space in the heap
 - (Implicitly) invoke the superclass constructor, then run the constructor body
- Objects and their methods:
 - Each object in the heap has a pointer to the class table of its dynamic type (the one it was created with via `new`).
 - A method invocation “`o.m(...)`” uses `o`’s class table to “dispatch” to the appropriate method code (might involve searching up the class hierarchy).
 - Methods and constructors take an implicit “`this`” parameter, which is a pointer to the object whose method was invoked. Fields& methods are accessed with `this`.

Programming Languages and Techniques (CIS120)

Lecture 26

November 3, 2017

The Java ASM, Java Generics
Chapter 24

Announcements

- HW7: Chat Server
 - Available on Codio / Instructions on the web site
 - Due Tuesday, November 14th at 11:59pm
- *Midterm 2 is next Friday, in class*
 - Last names A – M Leidy Labs 10 (here)
 - Last names N – Z Meyerson B1
- *Review Session: Wednesday November 8th at 6:00pm*
- Coverage:
 - Mutable state (in OCaml and Java)
 - Objects (in OCaml and Java)
 - ASM (in OCaml and Java)
 - Reactive programming (in Ocaml)
 - Arrays (in Java)
 - Subtyping, Simple Extension, Dynamic Dispatch (in Java)
- Sample exams from recent years posted on course web page
- Makeup exam request form: on the course web pages

When do constructors execute?

How are fields accessed?

What code runs in a method call?

What is 'this'?

ASM refinement: The Class Table

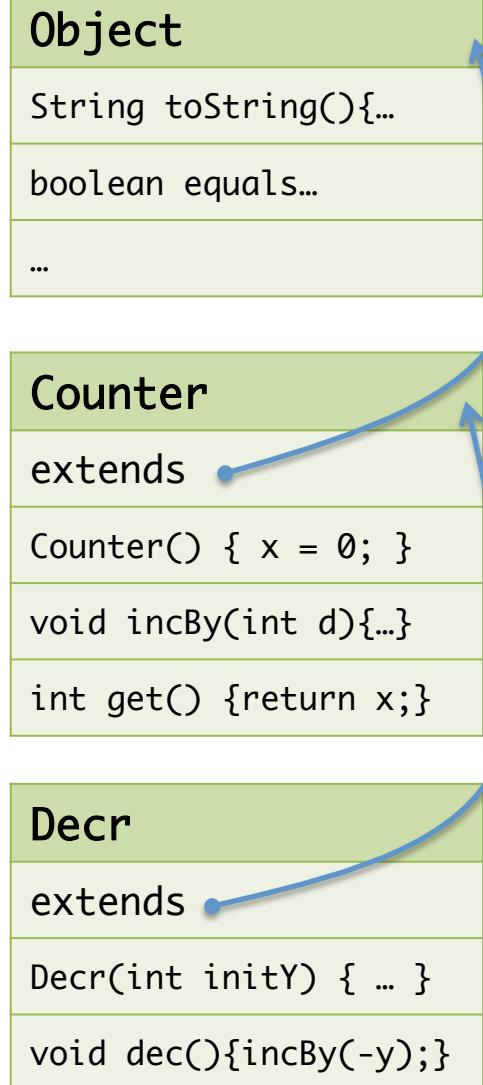
```
class Counter {  
    private int x;  
    public Counter () { x = 0; }  
    public void incBy(int d) { x = x + d; }  
    public int get() { return x; }  
}
```

```
class Decr extends Counter {  
    private int y;  
    public Decr (int initY) { y = initY; }  
    public void dec() { incBy(-y); }  
}
```

The class table contains:

- the code for each method,
- references to each class's parent, and
- the class's static members.

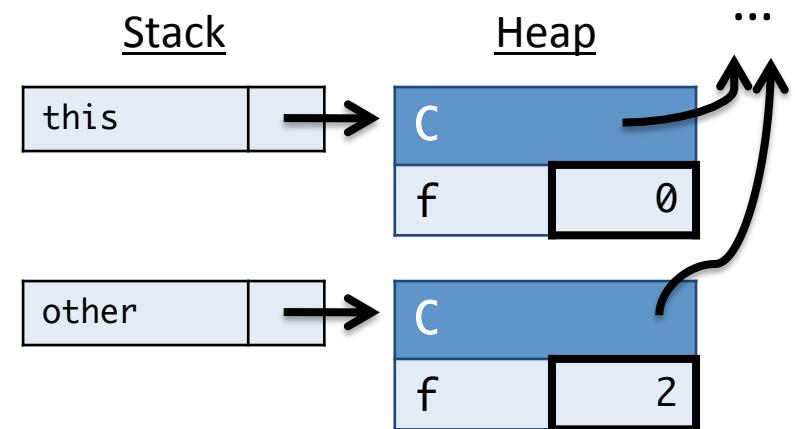
Class Table



this

- Inside a non-static method, the variable **this** refers to the object on which the method was invoked.
- References to fields and methods in the same class as the currently running method have an implicit “**this.**” in front of them.

```
class C {  
    private int f;  
  
    public void copyF(C other) {  
        this.f = other.f;  
    }  
}
```



Example

```
class Counter {  
    private int x;  
    public Counter () { x = 0; }  
    public void incBy(int d) { x = x + d; }  
    public int get() { return x; }  
}  
  
class Decr extends Counter {  
    private int y;  
    public Decr (int initY) { y = initY; }  
    public void dec() { incBy(-y); }  
}  
  
// ... somewhere in main:  
Decr d = new Decr(2);  
d.dec();  
int x = d.get();
```

...with Explicit this and super

```
class Counter extends Object {  
    private int x;  
    public Counter () { super(); this.x = 0; }  
    public void incBy(int d) { this.x = this.x + d; }  
    public int get() { return this.x; }  
}  
  
class Decr extends Counter {  
    private int y;  
    public Decr (int initY) { super(); this.y = initY; }  
    public void dec() { this.incBy(-this.y); }  
}  
  
// ... somewhere in main:  
Decr d = new Decr(2);  
d.dec();  
int x = d.get();
```

Constructing an Object

Workspace

```
Decr d = new Decr(2);  
d.dec();  
int x = d.get();
```

Stack

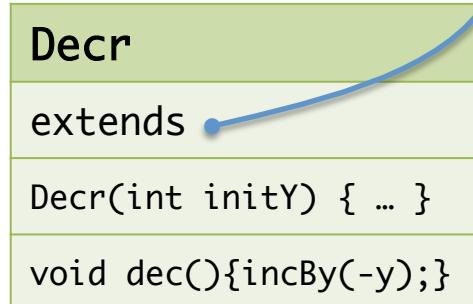
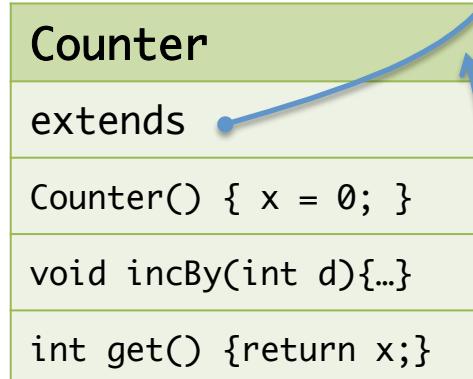
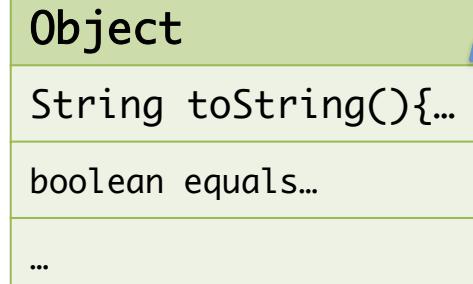
Heap

Class Table

| |
|--------------------------------------|
| Object |
| String <code>toString()</code> {...} |
| boolean <code>equals...</code> |
| ... |

| |
|--------------------------------------|
| Counter |
| extends |
| Counter() { x = 0; } |
| void <code>incBy(int d)</code> {...} |
| int <code>get()</code> {return x;} |

| |
|--|
| Decr |
| extends |
| Decr(int initY) { ... } |
| void <code>dec()</code> { <code>incBy(-y)</code> ;}} |



Allocating Space on the Heap

Workspace

```
super();  
this.y = initY;
```

Stack

```
Decr d = ...;  
d.dec();  
int x = d.get();  
  
this  
initY
```

Heap

| Decr | |
|------|---|
| x | 0 |
| y | 0 |

Class Table

Object

```
String toString(){...}
```

```
boolean equals...
```

```
...
```

Counter

```
extends Object
```

```
Counter() { x = 0; }
```

```
void incBy(int d){...}
```

```
int get() {return x;}
```

Decr

```
extends Counter
```

```
Decr(int initY) { ... }
```

```
void dec(){incBy(-y);}
```

Invoking a constructor:

- allocates space for a new object in the heap
- includes slots for *all* fields of *all* ancestors in the class tree (here: *x and y*)
- creates a pointer to the class – this is the object's dynamic type
- runs the constructor body after pushing parameters and *this* onto the stack

Note: fields start with a “sensible” default

- 0 for numeric values
- null for references

Calling super

Workspace

```
super();  
this.y = initY;
```

Stack

```
Decr d = ...;  
d.dec();  
int x = d.get();  
  
this  
initY
```

Heap

| | |
|------|---|
| Decr | |
| x | 0 |
| y | 0 |

Class Table

Object

```
String toString(){...}  
boolean equals...  
...
```

Counter

```
extends Object  
Counter() { x = 0; }  
void incBy(int d){...}  
int get() {return x;}
```

Decr

```
extends Counter  
Decr(int initY) { ... }  
void dec(){incBy(-y);}
```

Call to super:

- The constructor (implicitly) calls the super constructor
- Invoking a method or constructor pushes the saved workspace, the method params (none here) and a new this pointer.

Abstract Stack Machine

Workspace

```
super();  
this.x = 0;
```

Stack

```
Decr d = ...;  
d.dec();  
int x = d.get();
```

```
this
```

```
initY 2
```

```
; this.y = initY;
```

```
this
```

(Running Object's default constructor omitted.)

Heap

```
Decr
```

| | |
|---|---|
| x | 0 |
|---|---|

| | |
|---|---|
| y | 0 |
|---|---|

Class Table

```
Object
```

```
String toString(){...}
```

```
boolean equals...
```

```
...
```

```
Counter
```

```
extends Object
```

```
Counter() { x = 0; }
```

```
void incBy(int d){...}
```

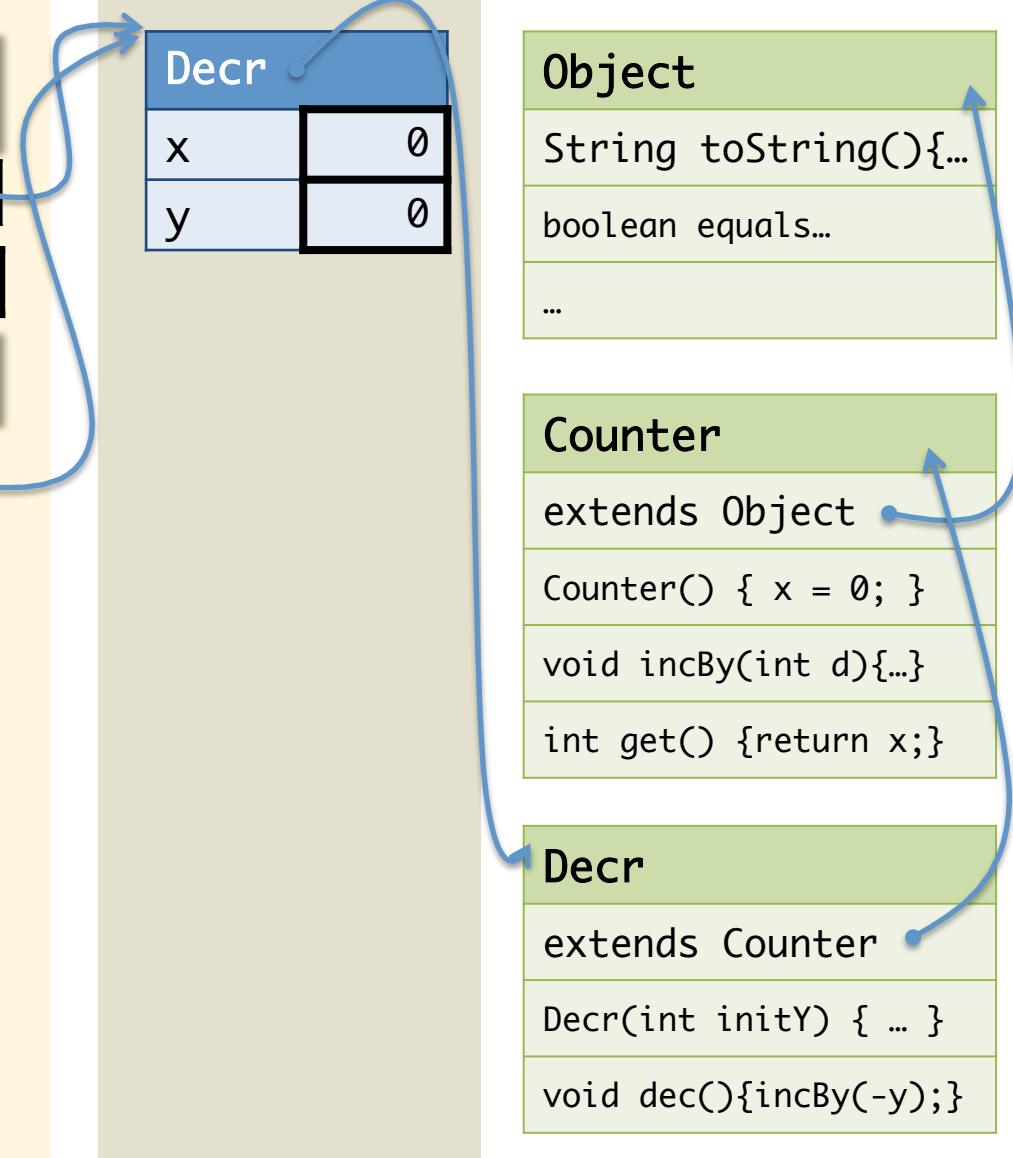
```
int get() {return x;}
```

```
Decr
```

```
extends Counter
```

```
Decr(int initY) { ... }
```

```
void dec(){incBy(-y);}
```



Assigning to a Field

Workspace

```
this.x = 0;
```

Stack

```
Decr d = ...;  
d.dec();  
int x = d.get();
```

| | |
|------|---|
| this | |
| | ↳ |

| | |
|-------|---|
| initY | 2 |
| | ↳ |

```
...;  
this.y = initY;
```

| | |
|------|---|
| this | |
| | ↳ |

Assignment into the `this.x` field goes in two steps:

- look up the value of `this` in the stack
- write to the “X” slot of that object.

Heap

| | |
|------|---|
| Decr | |
| x | 0 |
| y | 0 |

Class Table

Object

```
String toString(){...}
```

```
boolean equals...
```

```
...
```

Counter

```
extends Object
```

```
Counter() { x = 0; }
```

```
void incBy(int d){...}
```

```
int get() {return x;}
```

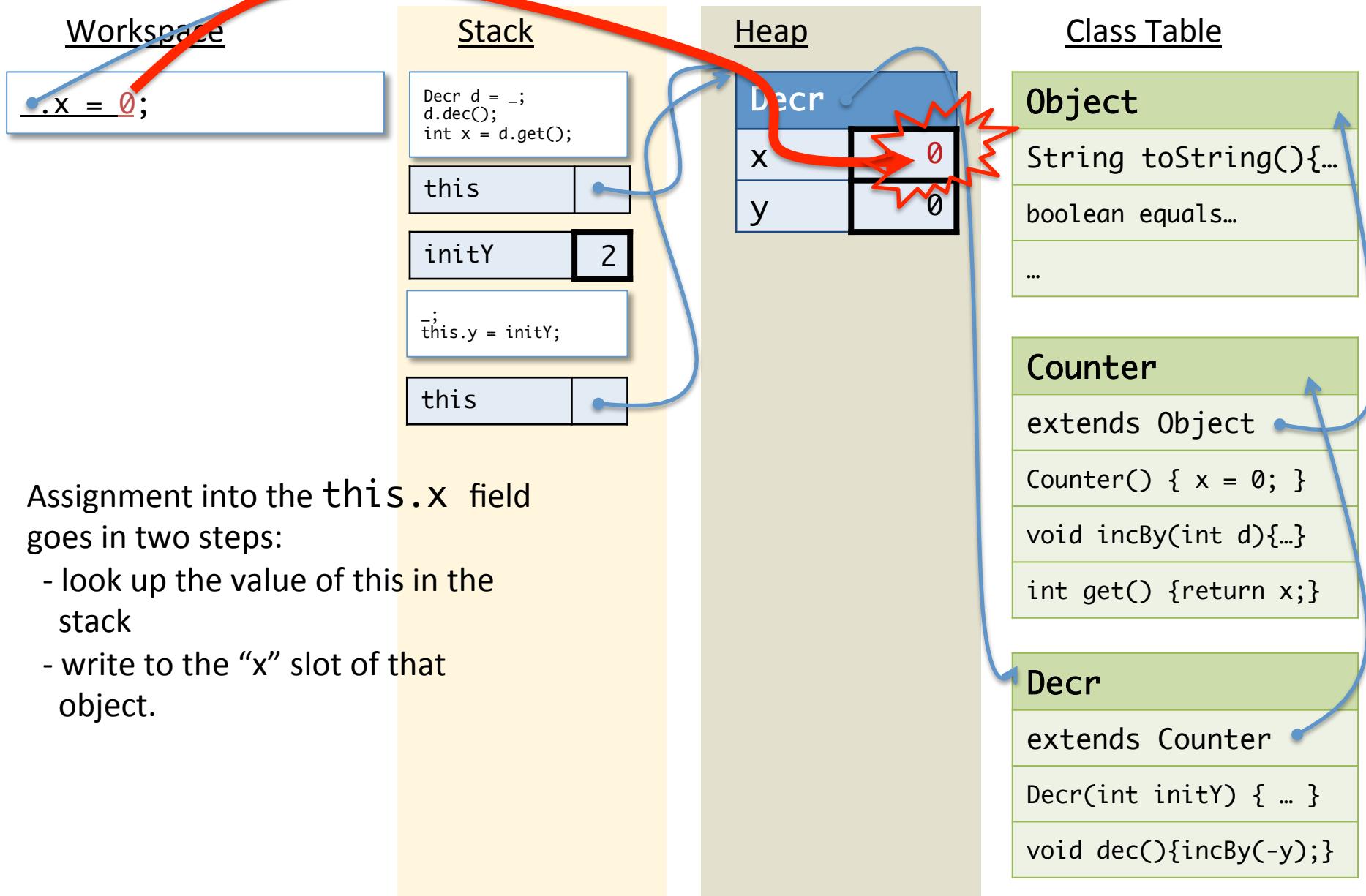
Decr

```
extends Counter
```

```
Decr(int initY) { ... }
```

```
void dec(){incBy(-y);}
```

Assigning to a Field



Done with the call

Workspace

```
;
```

Stack

```
Decr d = ...;  
d.dec();  
int x = d.get();
```

| | |
|------|--|
| this | |
|------|--|

| | |
|-------|---|
| initY | 2 |
|-------|---|

```
...  
this.y = initY;
```

| | |
|------|--|
| this | |
|------|--|

Done with the call to “super”, so
pop the stack to the previous
workspace.

Heap

Decr

| | |
|---|---|
| x | 0 |
| y | 0 |

Class Table

Object

```
String toString(){...}
```

```
boolean equals...
```

```
...
```

Counter

```
extends Object
```

```
Counter() { x = 0; }
```

```
void incBy(int d){...}
```

```
int get() {return x;}
```

Decr

```
extends Counter
```

```
Decr(int initY) { ... }
```

```
void dec(){incBy(-y);}
```

Continuing

Workspace

```
this.y = initY;
```

Continue in the `Decr` class's constructor.

Stack

```
Decr d = ...;  
d.dec();  
int x = d.get();
```

| | |
|-------|---|
| this | |
| initY | 2 |

Heap

Decr

| | |
|---|---|
| x | 0 |
| y | 0 |

Class Table

Object

String `toString()`{...}

boolean `equals...`

...

Counter

extends Object

`Counter()` { `x = 0;` }

`void incBy(int d){...}`

`int get() {return x;}`

Decr

extends Counter

`Decr(int initY) { ... }`

`void dec(){incBy(-y);}`

Workspace

```
this.y = 2;
```

Stack

```
Decr d = ...;  
d.dec();  
int x = d.get();
```

| | |
|------|--|
| this | |
|------|--|

| | |
|-------|---|
| initY | 2 |
|-------|---|

Heap

Decr

| | |
|---|---|
| x | 0 |
| y | 0 |

Class Table

Object

String toString(){...}

boolean equals...

...

Counter

extends Object

Counter() { x = 0; }

void incBy(int d){...}

int get() {return x;}

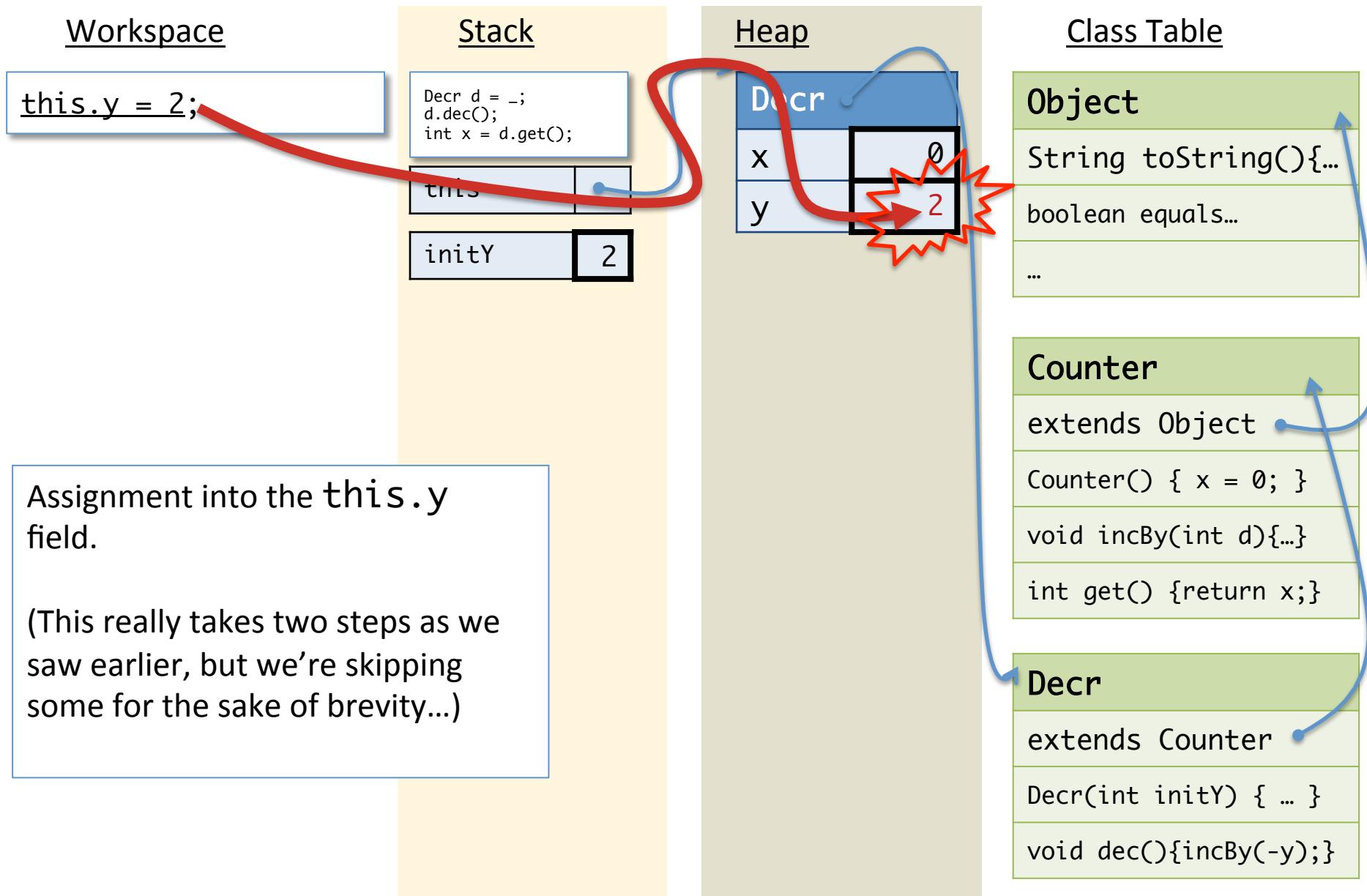
Decr

extends Counter

Decr(int initY) { ... }

void dec(){incBy(-y);}

Assigning to a field



Done with the call

Workspace

```
;
```

Stack

```
Decr d = _;
d.dec();
int x = d.get();
```

| | |
|------|--|
| this | |
|------|--|

| | |
|-------|---|
| initY | 2 |
|-------|---|

Heap

```
Decr
```

| | |
|---|---|
| x | 0 |
| y | 2 |

Class Table

```
Object
```

```
String toString(){...}
```

```
boolean equals...
```

```
...
```

```
Counter
```

```
extends Object
```

```
Counter() { x = 0; }
```

```
void incBy(int d){...}
```

```
int get() {return x;}
```

```
Decr
```

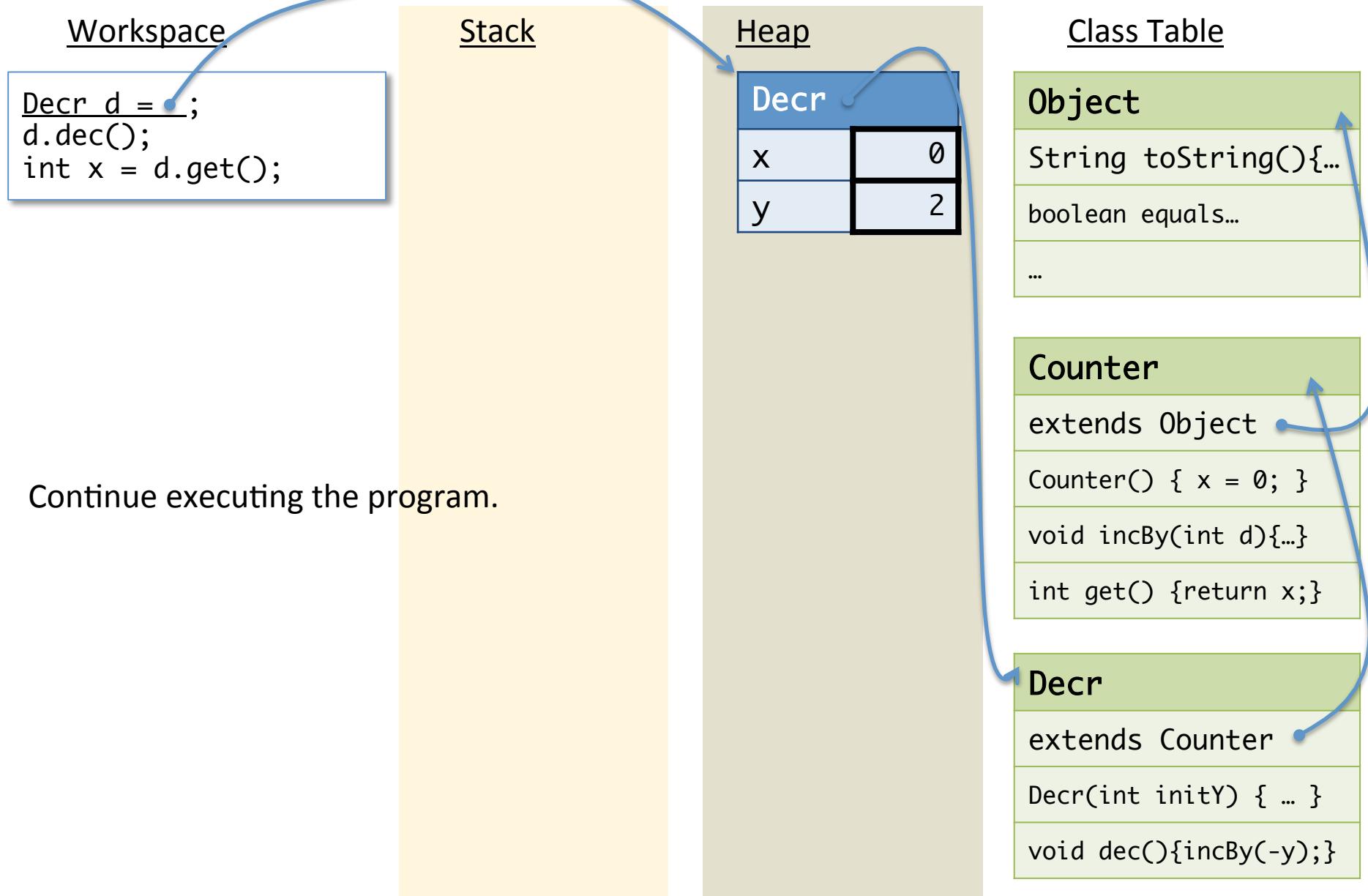
```
extends Counter
```

```
Decr(int initY) { ... }
```

```
void dec(){incBy(-y);}
```

Done with the call to the Decr constructor, so pop the stack and return to the saved workspace, returning the newly allocated object (now in the this pointer).

Returning the Newly Constructed Object



Allocating a local variable

Workspace

```
d.dec();  
int x = d.get();
```

Stack



Heap



Class Table

Object

```
String toString(){...}
```

```
boolean equals...
```

```
...
```

Counter

```
extends Object
```

```
Counter() { x = 0; }
```

```
void incBy(int d){...}
```

```
int get() {return x;}
```

Decr

```
extends Counter
```

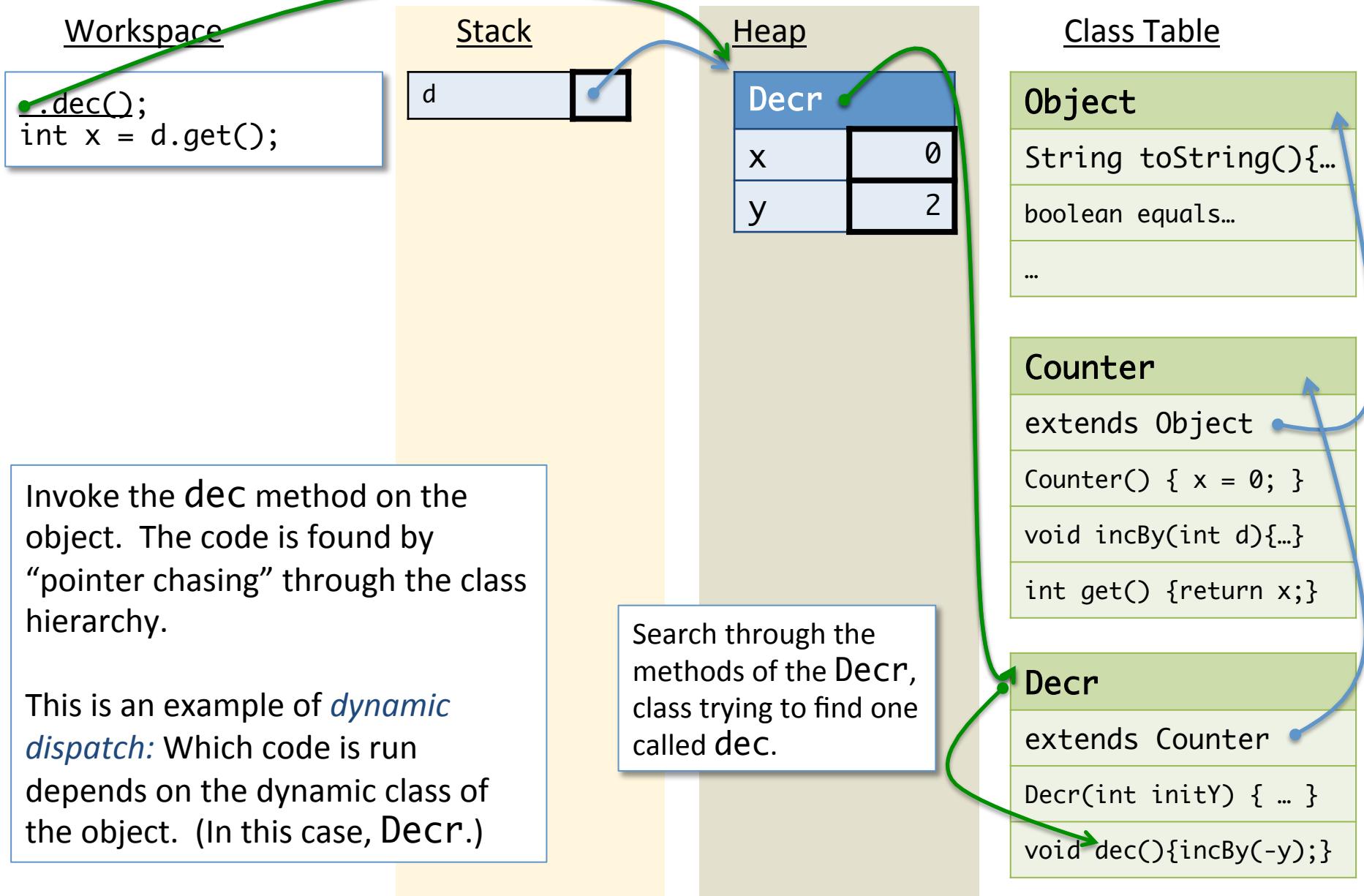
```
Decr(int initY) { ... }
```

```
void dec(){incBy(-y);}
```

Allocate a stack slot for the local variable `d`. Note that it's mutable... (bold box in the diagram).

Aside: since, by default, fields and local variables are mutable, we often omit the bold boxes and just assume the contents can be modified.

Dynamic Dispatch: Finding the Code



Dynamic Dispatch: Finding the Code

Workspace

```
this.incBy(-this.y);
```

Stack

| | |
|------|------------------|
| d | [] |
| -; | int x = d.get(); |
| this | [] |

Heap

| | |
|------|-----|
| Decr | [] |
| x | 0 |
| y | 2 |

Class Table

Object

```
String toString(){...}
```

```
boolean equals...
```

```
...
```

Counter

```
extends Object
```

```
Counter() { x = 0; }
```

```
void incBy(int d){...}
```

```
int get() {return x;}
```

Decr

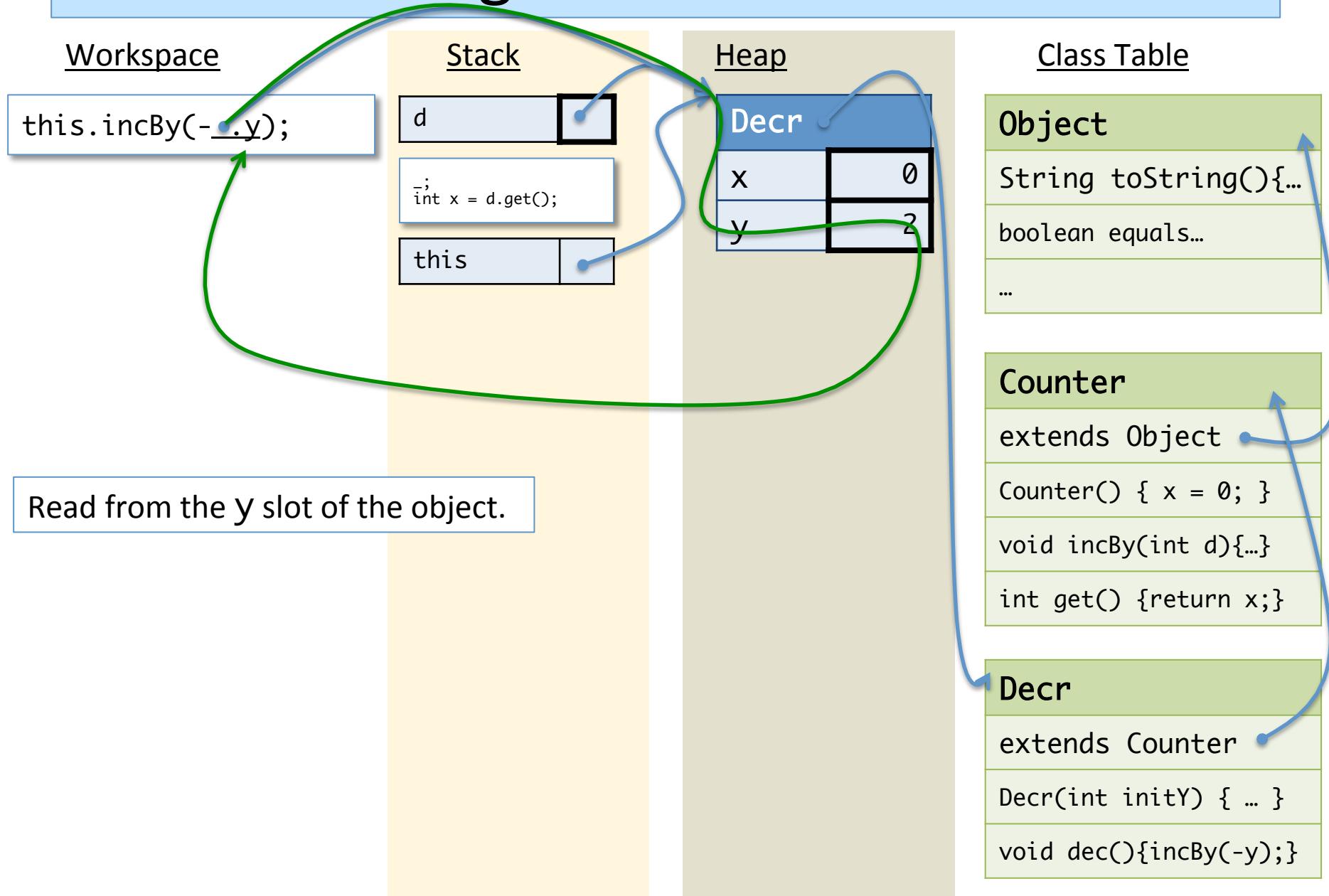
```
extends Counter
```

```
Decr(int initY) { ... }
```

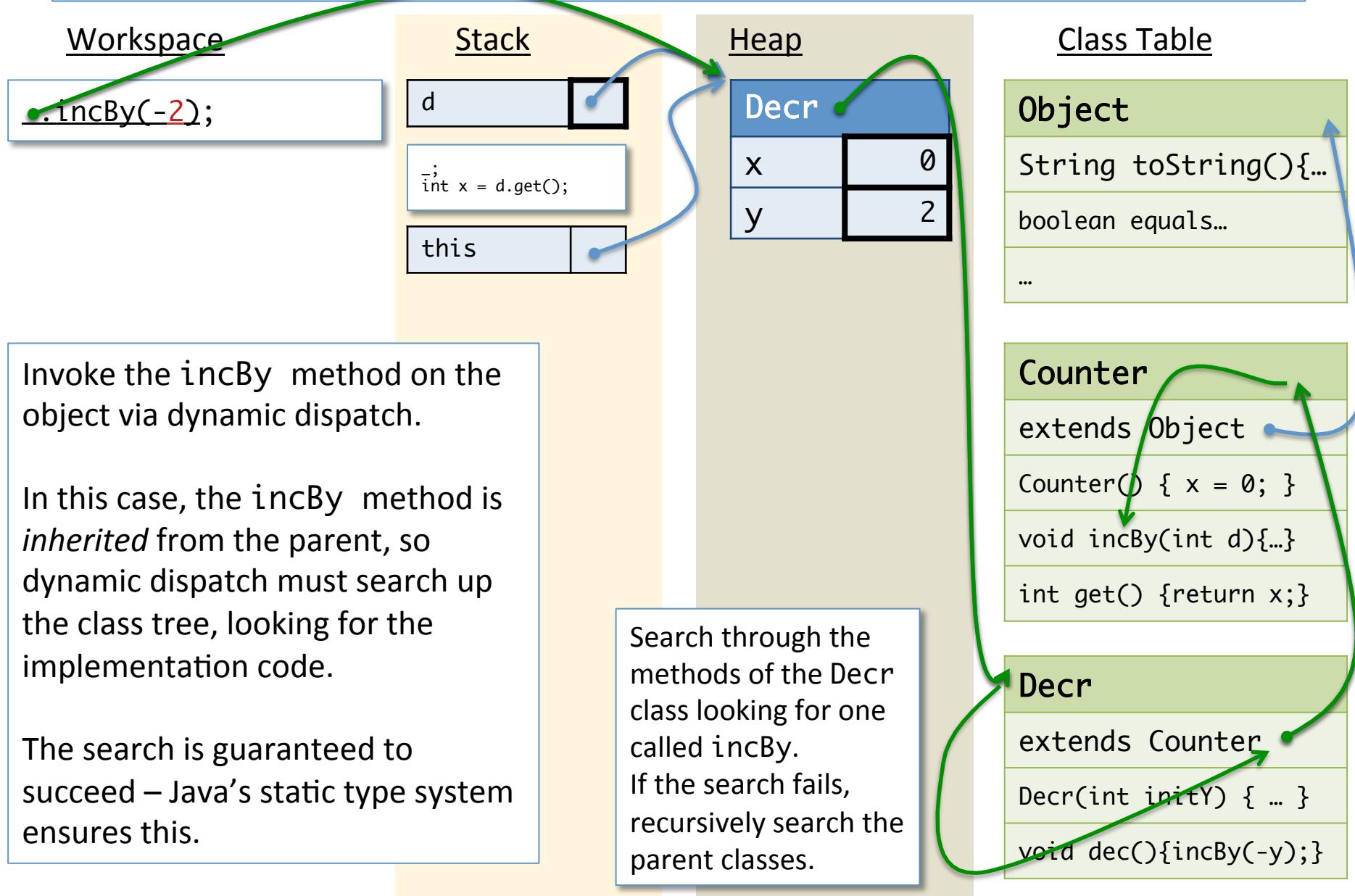
```
void dec(){incBy(-y);}
```

Call the method, remembering the current workspace and pushing the `this` pointer and any arguments (none in this case).

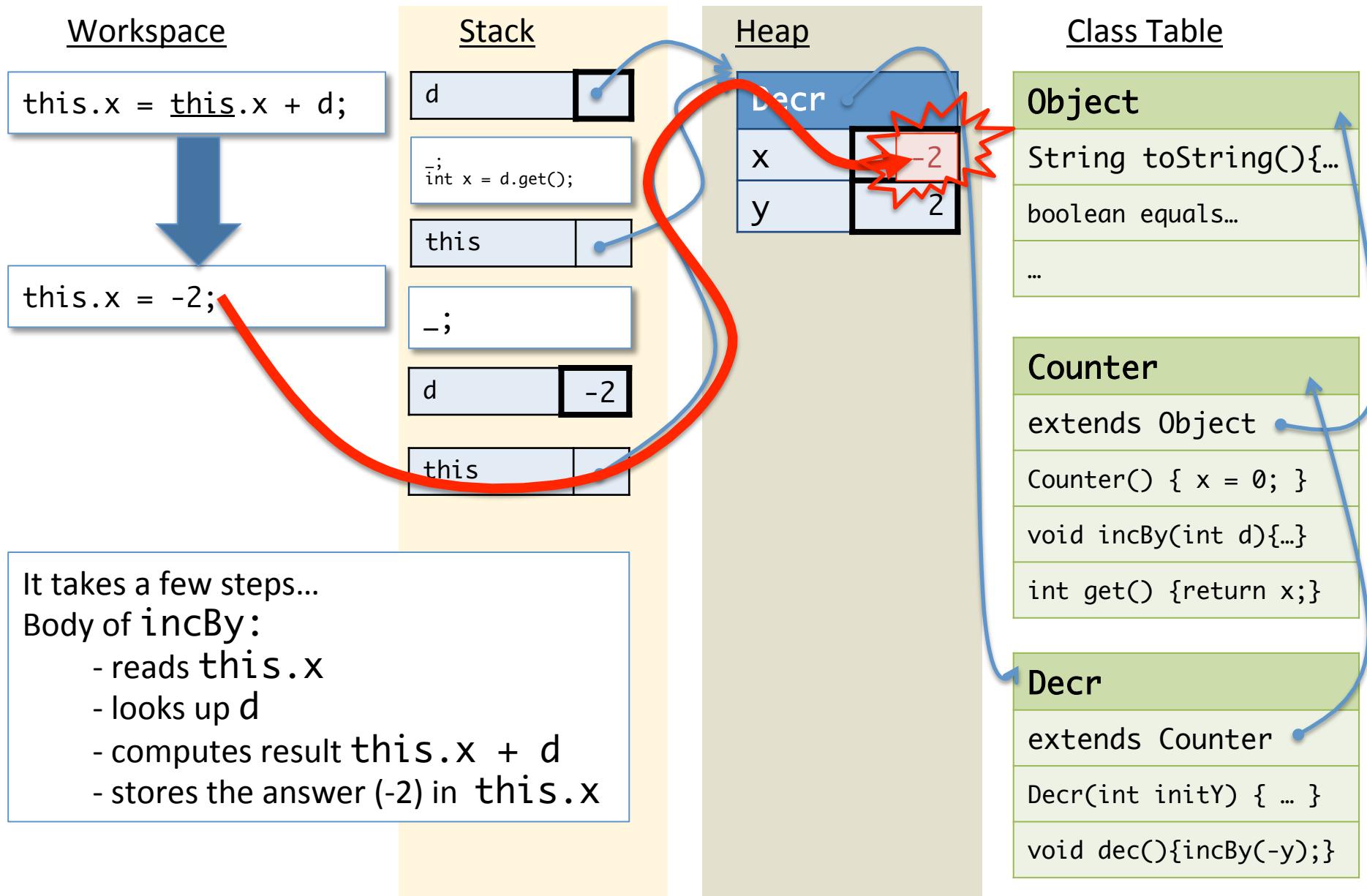
Reading a Field's Contents



Dynamic Dispatch, Again



Running the body of incBy



After a few more steps...

Workspace

```
int x = d.get();
```

Stack



Heap

| Decr | |
|------|----|
| x | -2 |
| y | 2 |

Class Table

Object

String `toString()`{...}

boolean `equals...`

...

Counter

extends Object

`Counter()` { `x = 0;` }

`void incBy(int d){...}`

`int get() {return x;}`

Decr

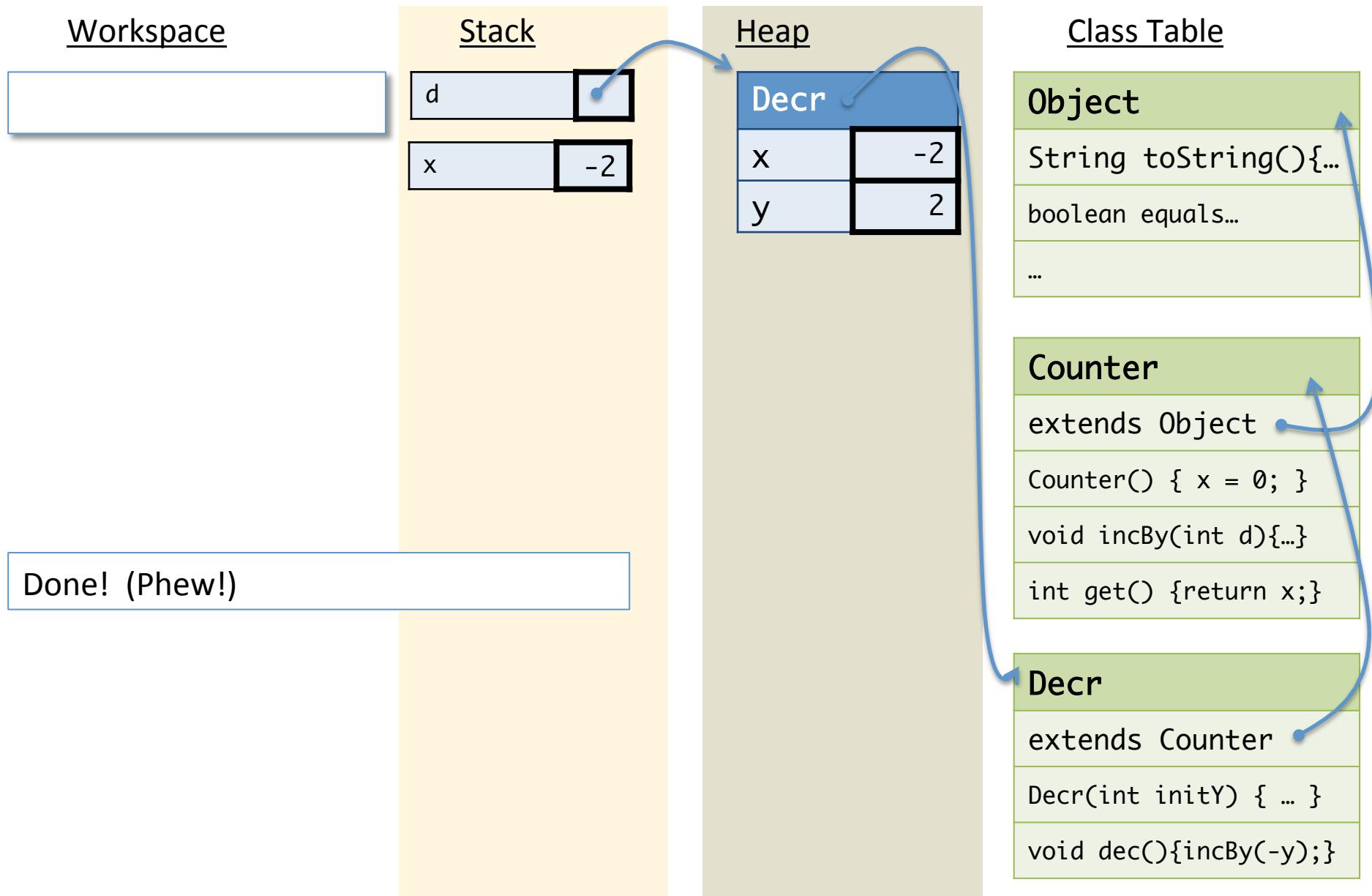
extends Counter

`Decr(int initY) { ... }`

`void dec(){incBy(-y);}`

Now use dynamic dispatch to invoke the `get` method for `d`. This involves searching up the class hierarchy again...

After yet a few more steps...



Summary: `this` and dynamic dispatch

- When object's method is invoked, as in `o.m()`, the code that runs is determined by `o`'s *dynamic* class.
 - The dynamic class, represented as a pointer into the class table, is included in the object structure in the heap
 - If the method is inherited from a superclass, determining the code for `m` might require searching up the class hierarchy via pointers in the class table
 - This process of *dynamic dispatch* is the heart of OOP!
- Once the code for `m` has been determined, a binding for `this` is pushed onto the stack.
 - The `this` pointer is used to resolve field accesses and method invocations inside the code.

Inheritance Example

```
public class Counter {  
    private int x;  
    public Counter () { x = 0; }  
    public void incBy(int d) { x = x + d; }  
    public int get() { return x; }  
}  
  
class Decr extends Counter {  
    private int y;  
    public Decr (int initY) { y = initY; }  
    public void dec() { incBy(-y); }  
}  
  
// ... somewhere in main:  
Decr d = new Decr(2);  
d.dec();  
int x = d.get();
```

What is the value of x at the end of this computation?

1. -2
2. -1
3. 0
4. 1
5. 2
6. NPE
7. Doesn't type check

Answer: -2

Static members and the Java ASM

Static Members

- Classes in Java can also act as *containers* for code and data.
- The modifier **static** means that the field or method is associated with the class and *not* instances of the class.

```
class C {  
    public static int x = 23;  
    public static int someMethod(int y) { return C.x + y; }  
    public static void main(String args[]) {  
        ...  
    }  
}  
  
// Elsewhere:  
C.x = C.x + 1;  
C.someMethod(17);
```

You can do a static assignment to initialize a static field.

Access to the static member uses the class name C.x or C.foo()

Based on your understanding of 'this', is it possible to refer to 'this' in a static method?

1. No
2. Yes
3. I'm not sure

Class Table Associated with C

- The class table entry for C has a field slot for x.
- Updates to C.x modify the contents of this slot: C.x = 17;



| | |
|---------------------------------|----|
| C | 23 |
| extends Object | |
| static x | |
| static int someMethod(int y) | |
| { return x + y; } | |
| static void main(String args[]) | |
| {...} | |

- A static field is a *global* variable
 - There is only one heap location for it (in the class table)
 - Modifications to such a field are visible everywhere the field is
 - if the field is public, this means *everywhere*
 - Use with care!

Static Methods (Details)

- Static methods do *not* have access to a `this` pointer
 - Why? There isn't an instance to dispatch through!
 - Therefore, static methods may only directly call other static methods.
 - Similarly, static methods can only directly read/write static fields.
 - Of course a static method can create instance of objects (via `new`) and then invoke methods on those objects.
- Gotcha: It is possible (but confusing) to invoke a static method as though it belongs to an object instance.
 - e.g. `o.someMethod(17)` where `someMethod` is static
 - Eclipse will issue a warning if you try to do this.

Java Generics

Subtype Polymorphism

vs.

Parametric Polymorphism

Subtype Polymorphism*

- Main idea:

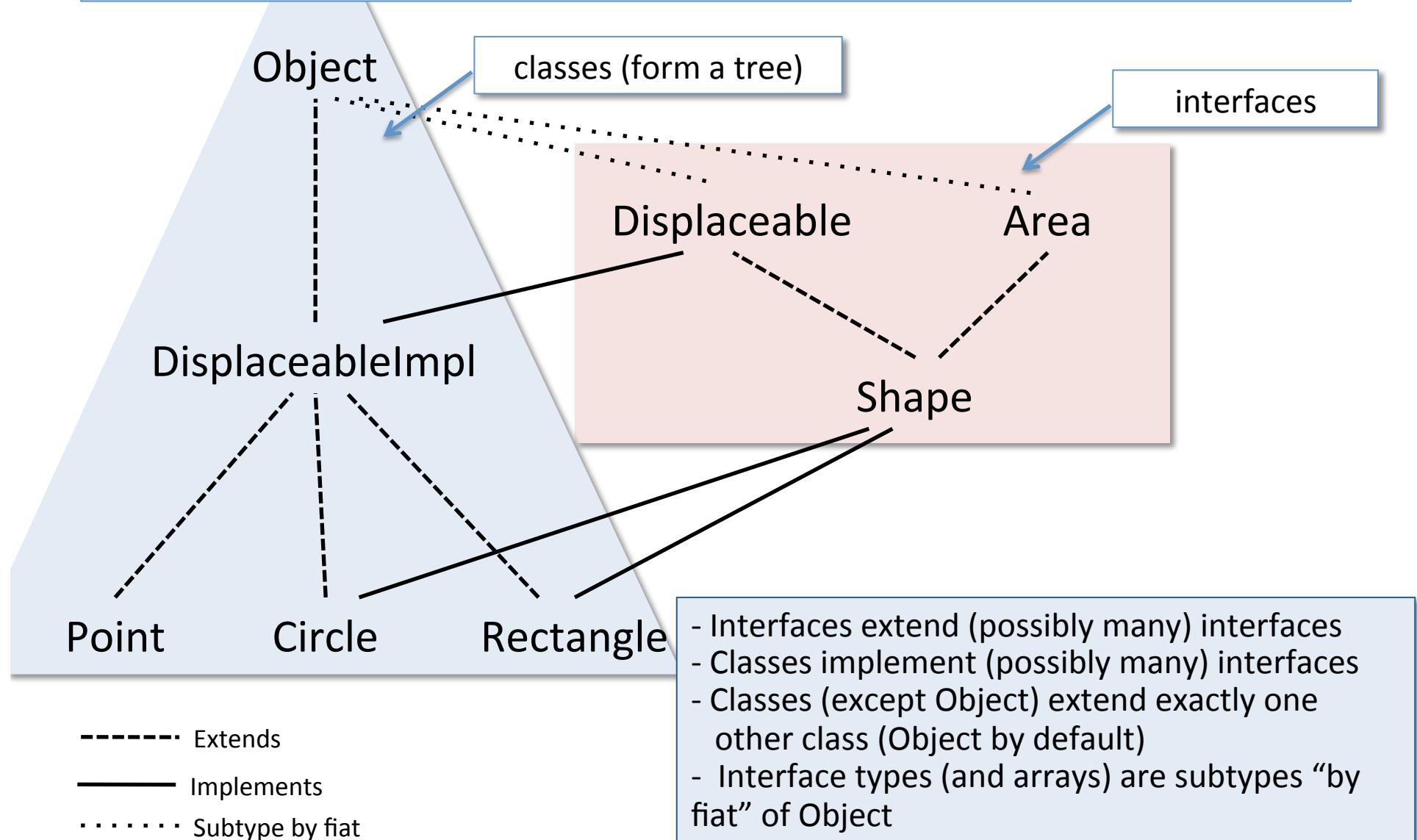
Anywhere an object of type A is needed, an object that is a subtype of A can be provided.

```
// in class C
public static void times2(Counter c) {
    c.incBy(c.get());
}
// somewhere else, Decr extends Counter
C.times2(new Decr(3));
```

- If B is a subtype of A, it provides all of A's (public) methods.

*polymorphism = many shapes

Recap: Subtyping



Is subtyping enough?

Mutable Queue ML Interface

```
module type QUEUE =
sig
  (* type of the data structure *)
  type 'a queue

  (* Make a new, empty queue *)
  val create : unit -> 'a queue

  (* Add a value to the end of the queue *)
  val enq : 'a -> 'a queue -> unit

  (* Remove the front value and return it (if any) *)
  val deq : 'a queue -> 'a

  (* Determine if the queue is empty *)
  val is_empty : 'a queue -> bool
end
```

How can we
translate this
interface to Java?

Java Interface

```
module type QUEUE =
sig
  type 'a queue
  val create : unit -> 'a queue
  val enq : 'a -> 'a queue -> unit
  val deq : 'a queue -> 'a
  val is_empty : 'a queue -> bool
end
```

```
interface ObjQueue {
  // no constructors
  // in an interface
  public void enq(Object elt);
  public Object deq();
  public boolean isEmpty();
}
```

Subtype Polymorphism

```
interface ObjQueue {  
    public void enq(Object elt);  
    public Object deq();  
    public boolean isEmpty();  
}
```

```
ObjQueue q = ...;  
  
q.enq(" CIS 120 ");  
__A__ x = q.deq();
```

What type for A?

1. String
2. Object
3. ObjQueue
4. None of the above

Subtype Polymorphism

```
interface ObjQueue {  
    public void enq(Object elt);  
    public Object deq();  
    public boolean isEmpty();  
}
```

```
ObjQueue q = ...;  
  
q.enq(" CIS 120 ");  
Object x = q.deq();  
System.out.println(x.trim());
```

← Does this line type check

1. Yes
2. No
3. It depends

Subtype Polymorphism

```
interface ObjQueue {  
    public void enq(Object elt);  
    public Object deq();  
    public boolean isEmpty();  
}
```

```
ObjQueue q = ...;  
  
q.enq(" CIS 120 ");  
Object x = q.deq();  
//System.out.println(x.trim());  
q.enq(new Point(0.0,0.0));  
---B--- y = q.deq();
```

What type for B?

1. Point
2. Object
3. ObjQueue
4. None of the above

Parametric Polymorphism (a.k.a. Generics)

- Big idea:

Parameterize a type (i.e. interface or class) by another type.

```
public interface Queue<E> {  
    public void enq(E o);  
    public E deq();  
    public boolean isEmpty();  
}
```

- The implementation of a parametric polymorphic interface cannot depend on the implementation details of the parameter.
 - e.g. the implementation of enq cannot invoke any methods on ‘o’

Generics (Parametric Polymorphism)

```
public interface Queue<E> {  
    public void enq(E o);  
    public E deq();  
    public boolean isEmpty();  
    ...  
}
```

```
Queue<String> q = ...;  
  
q.enq(" CIS 120 ");  
String x = q.deq();           // What type of x? String  
System.out.println(x.trim()); // Is this valid? Yes!  
q.enq(new Point(0.0,0.0));   // Is this valid? No!
```

Subtyping and Generics

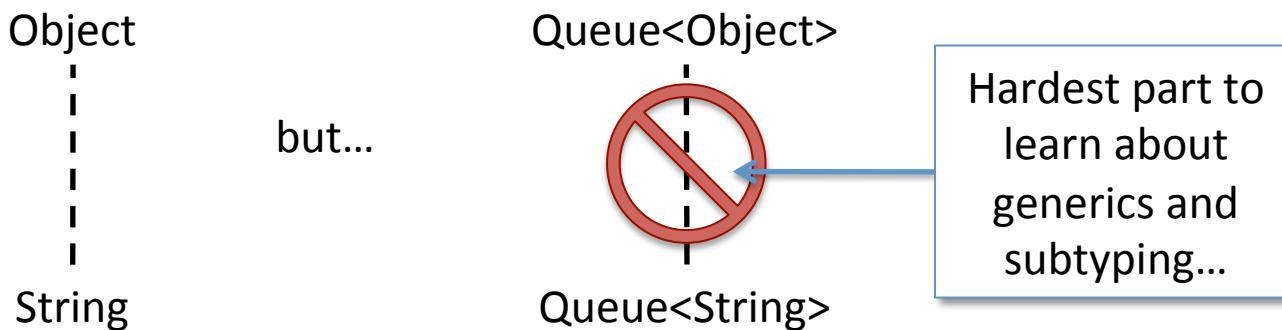
Subtyping and Generics*

```
Queue<String> qs = new QueueImpl<String>();  
Queue<Object> qo = qs;  
  
qo.enq(new Object());  
String s = qs.deq();
```

0k? Sure!
0k? Let's see...

0k? I guess
0k? Noooo!

- Java generics are *invariant*:
 - Subtyping of *arguments* to generic types does not imply subtyping between the instantiations:



* Subtyping and generics interact in other ways too. Java supports *bounded polymorphism* and *wildcard types*, but those are beyond the scope of CIS 120.

Subtyping and Generics

Which of these are true, assuming that class `QueueImpl<E>` implements interface `Queue<E>`?

1. `QueueImpl<Queue<String>>` is a subtype of `Queue<Queue<String>>`
2. `Queue<QueueImpl<String>>` is a subtype of `Queue<Queue<String>>`
3. Both
4. Neither

One Subtlety

- Unlike OCaml, Java classes and methods can be generic only with respect to *reference* types.
 - Not possible to do: Queue<int>
 - Must instead do: Queue<Integer>
- Java Arrays cannot be generic:
 - Not possible to do:

```
class C<E> {  
    E[] genericArray;  
    public C() {  
        genericArray = new E[];  
    }  
}
```