Programming Languages and Techniques (CIS120)

Lecture 16

Iteration & Tail Recursion

Chapter 16

Announcements

- Homework 4: Mutable Queues
  - Due tomorrow at 11:59pm
- Homework 5: GUI Programming
  - Available before Fall Break
  - Due: Oct 22nd
- Fall Break:
  - No lab/recitation sections this week
  - No class on Friday

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.

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
```

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
```

3.
```
let q = create () in
enq 1 q;
ignore (deq q);
enq 2 q;
2 = deq q
```

ANSWER: 3

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
```

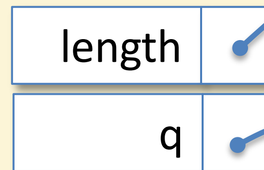
- This code for `length` uses a helper function, `loop`:
  - the correctness depends crucially on the queue invariant
  - (what happens if we pass in a bogus q that is cyclic?)
- The height of the ASM stack is proportional to the length of the queue
  - That seems inefficient… why should it take so much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

Evaluating length

Workspace

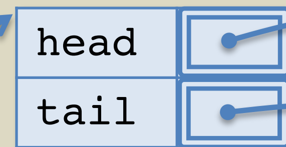
```
length q
```

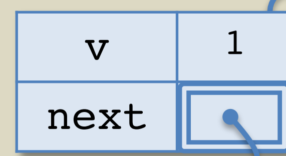
Stack

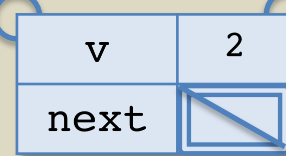


Heap

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





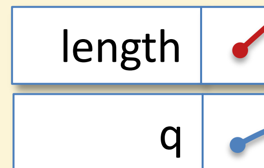


Evaluating length

Workspace

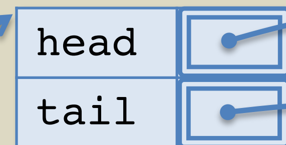
```
length q
```

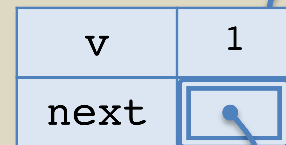
Stack

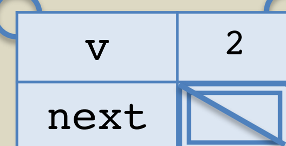


Heap

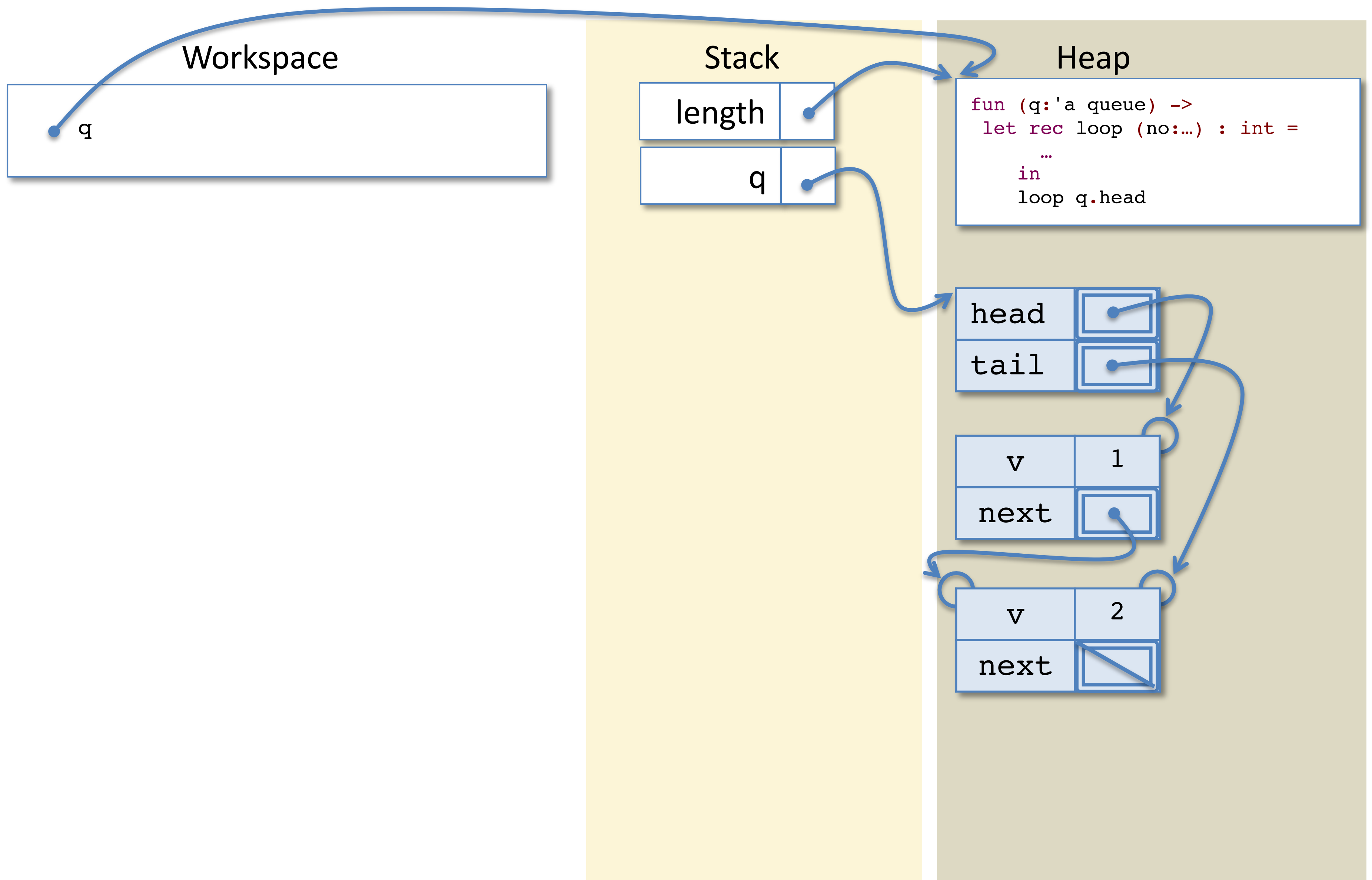
```
fun (q:'a queue) ->
  let rec loop (no:…) : int =
    …
  in
  loop q.head
```



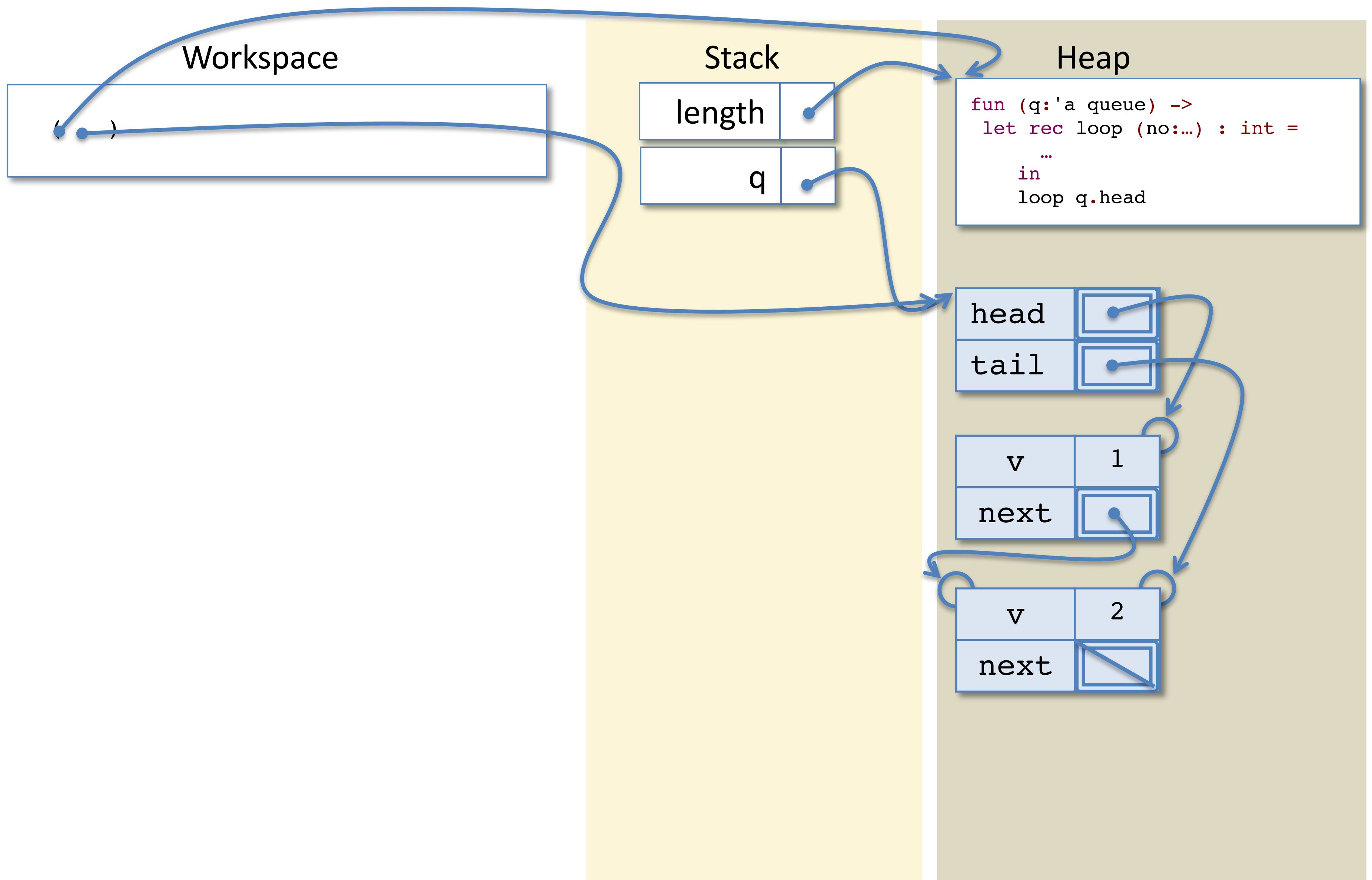




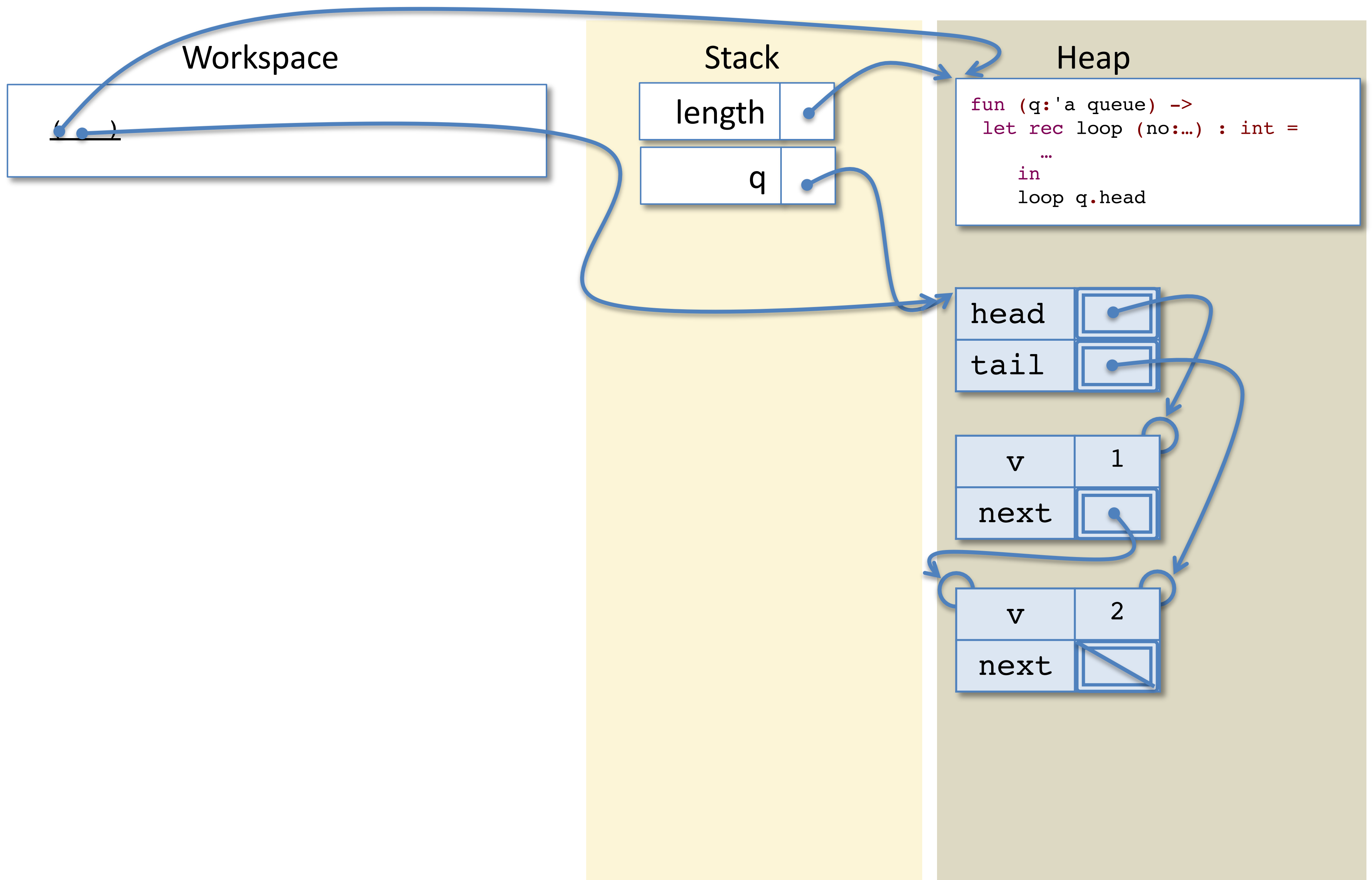
Evaluating length



Evaluating length



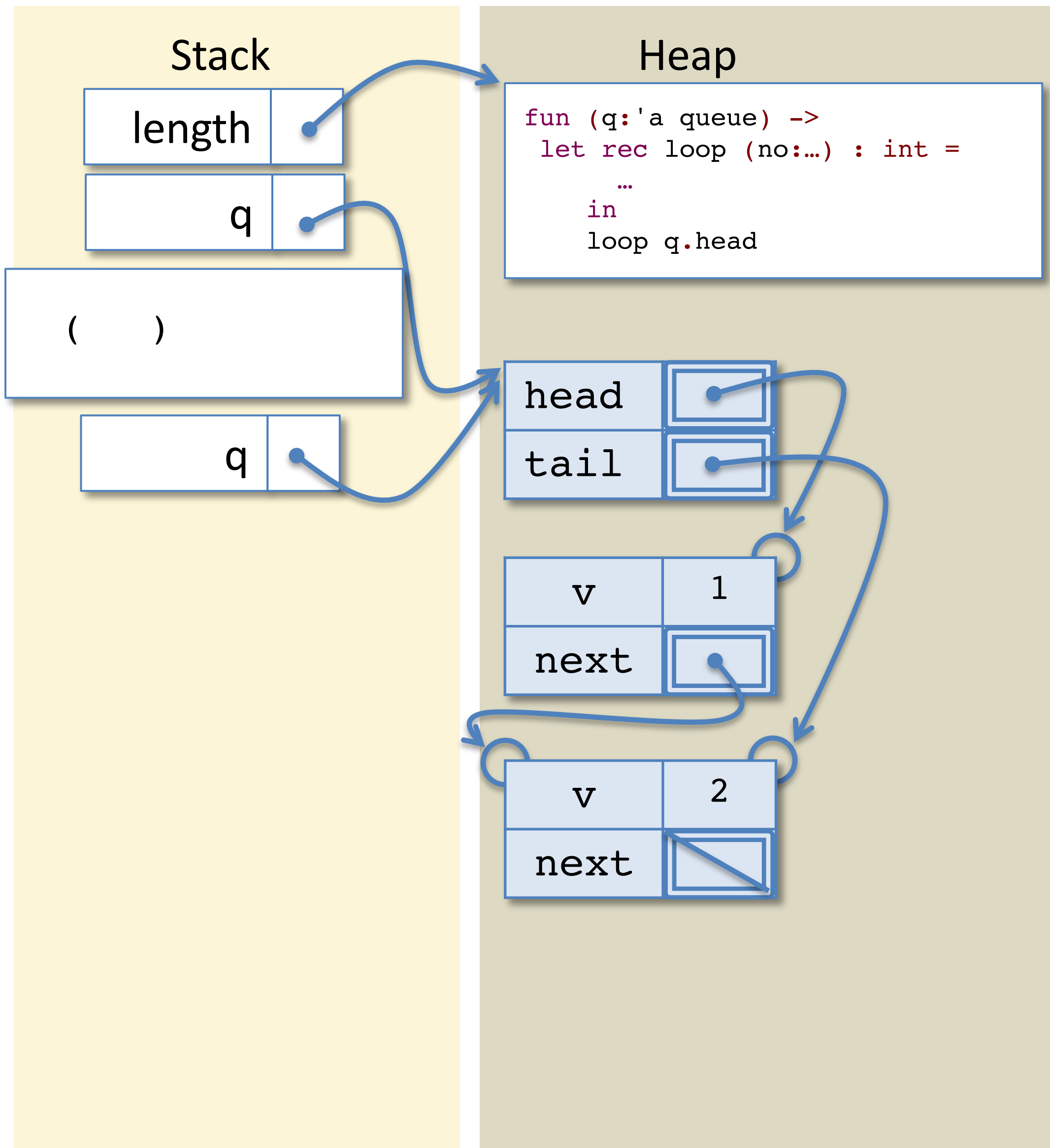
Evaluating length



Evaluating length

Workspace

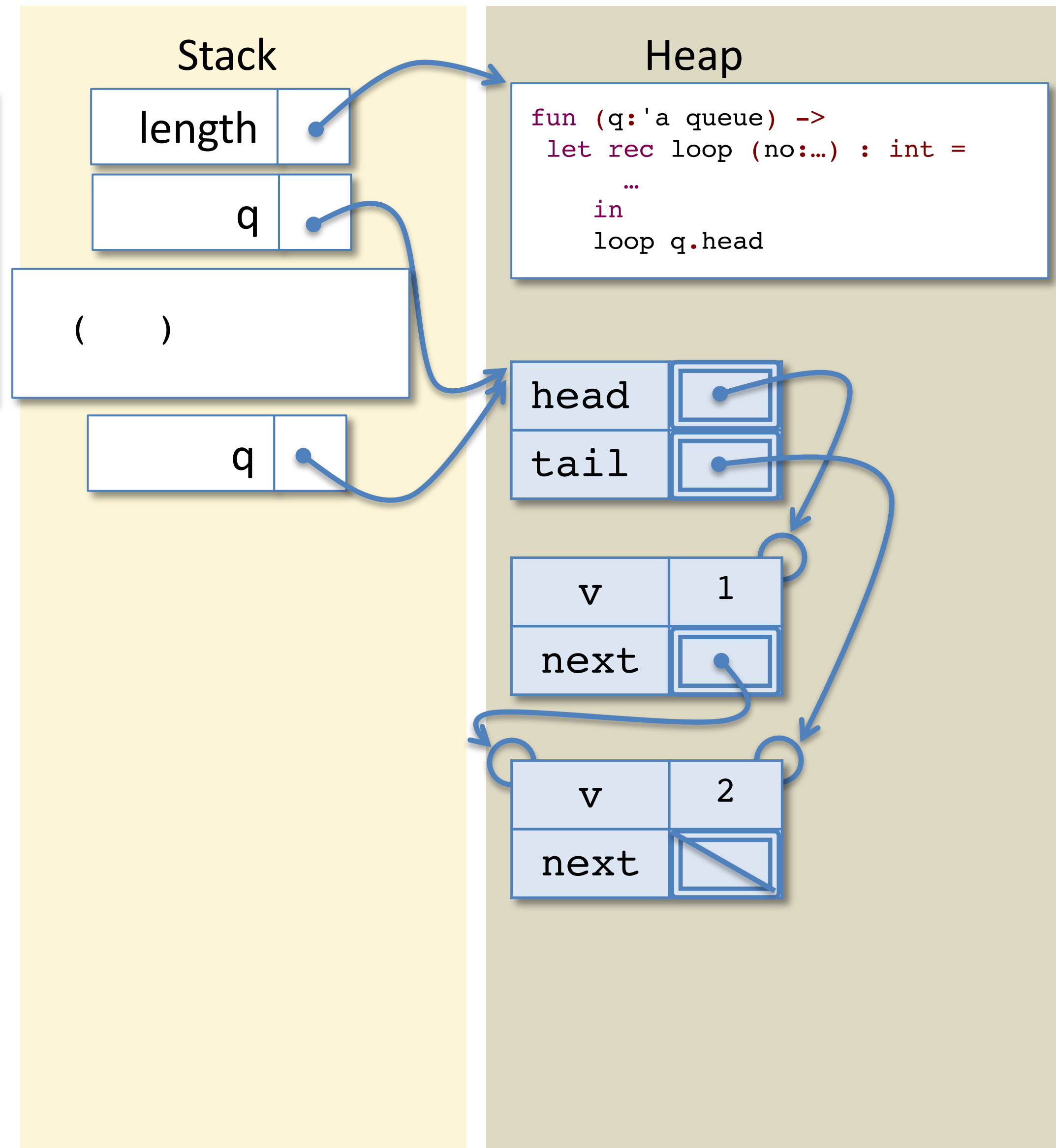
```
let rec loop (no: …) : int =
      begin match no with
        | None -> 0
        | Some n -> 1 + (loop n.next)
      end
    in
    loop q.head
```



Evaluating length

Workspace

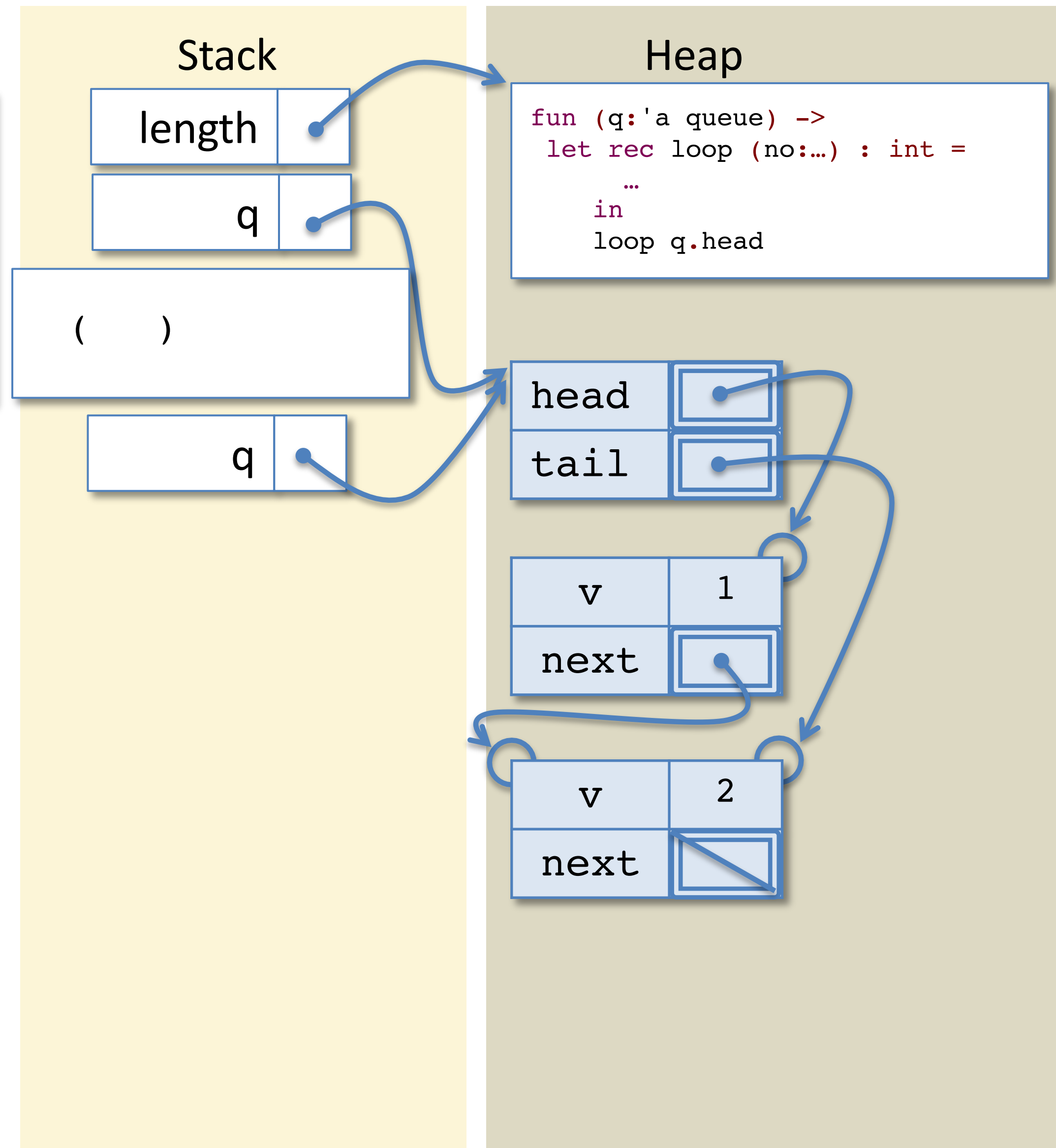
```
let rec loop = fun (no: …) ->
      begin match no with
        | None -> 0
        | Some n -> 1 + (loop n.next)
      end
    in
    loop q.head
```



Evaluating length

Workspace

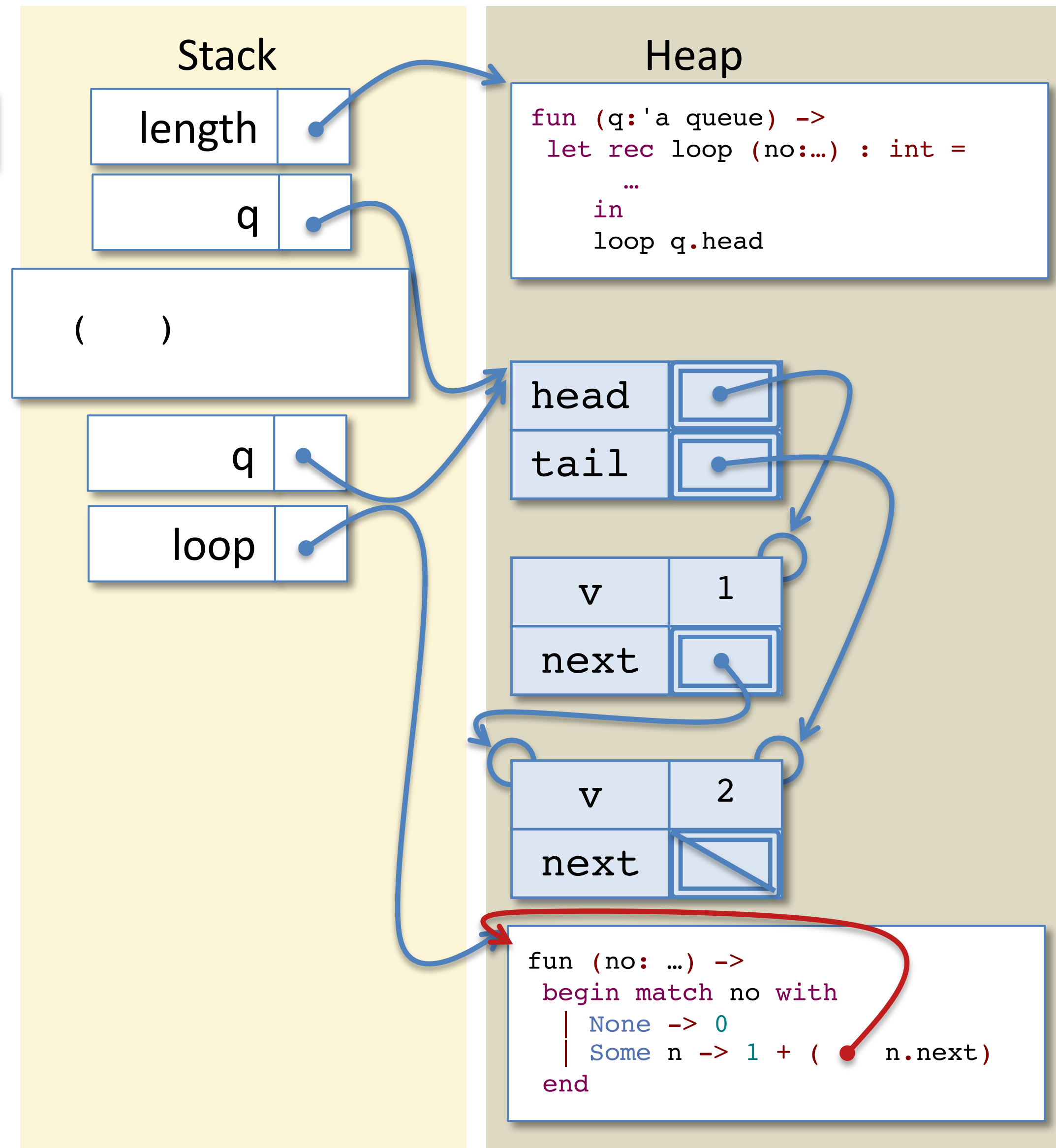
```
let loop = fun (no: …) ->
      begin match no with
          | None -> 0
          | Some n -> 1 + (loop n.next)
      end
    in
    loop q.head
```



Evaluating length

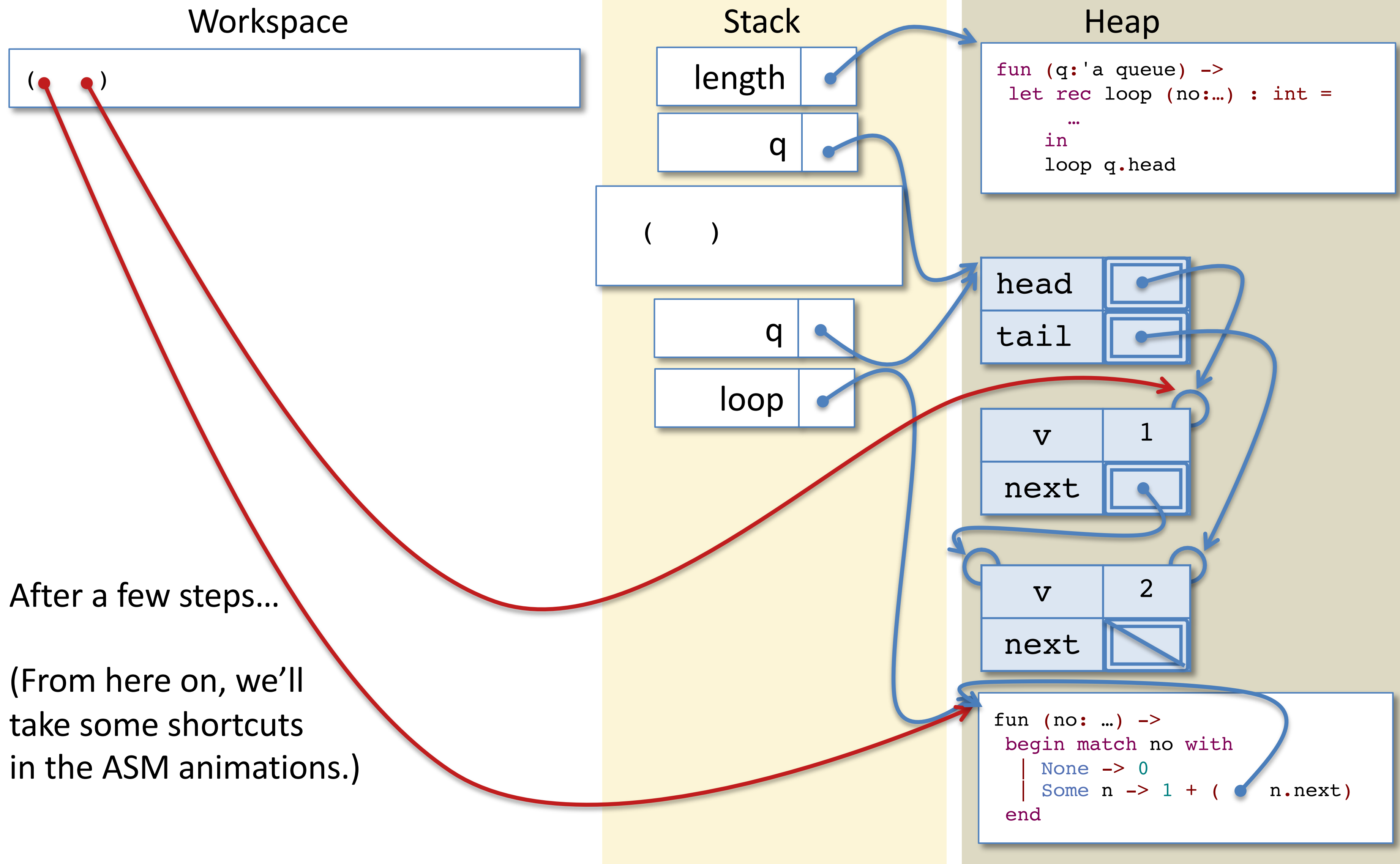
Workspace

```
loop q.head
```



Recall that the loop reference in its own definitions is backpatched….

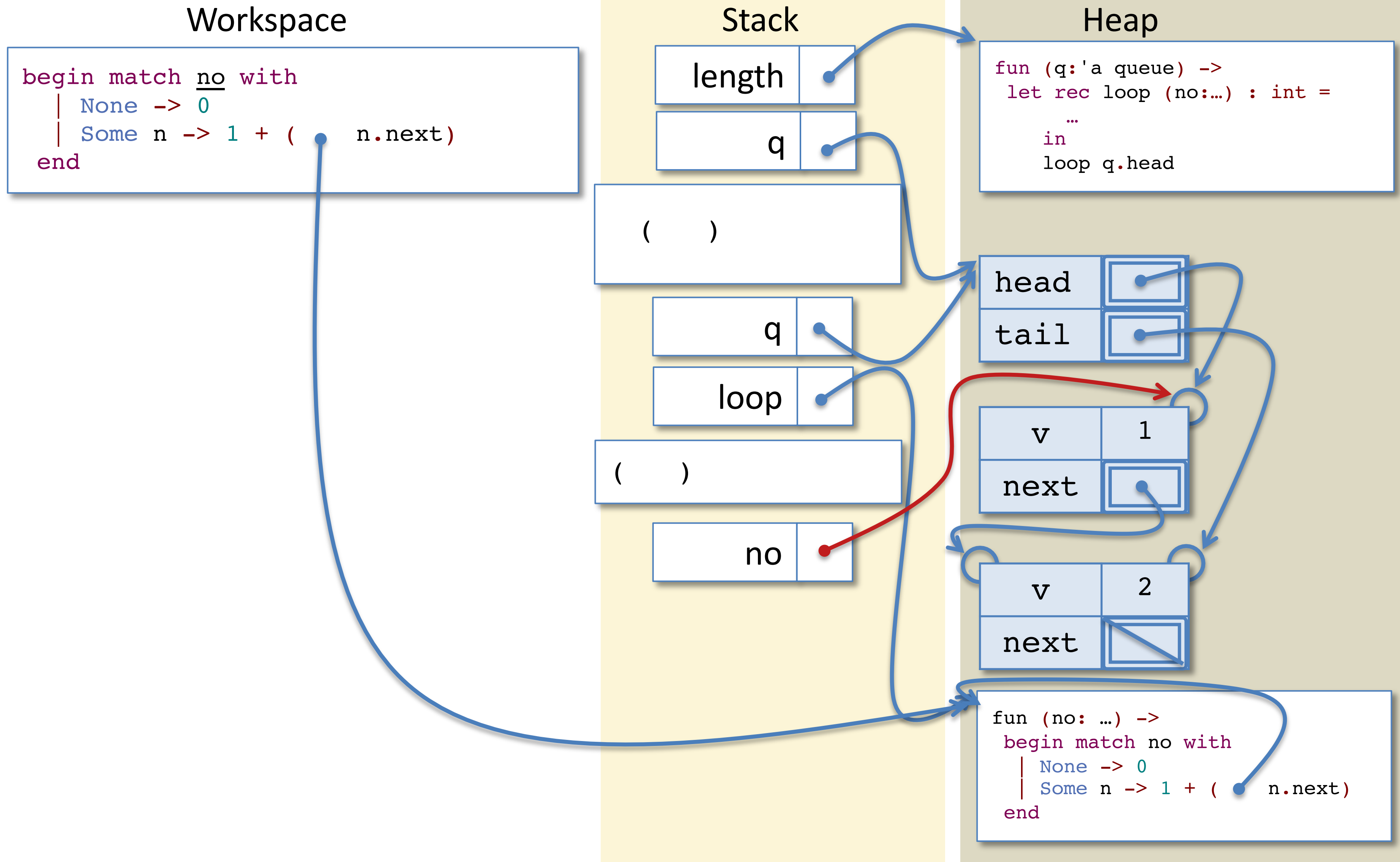
Evaluating length



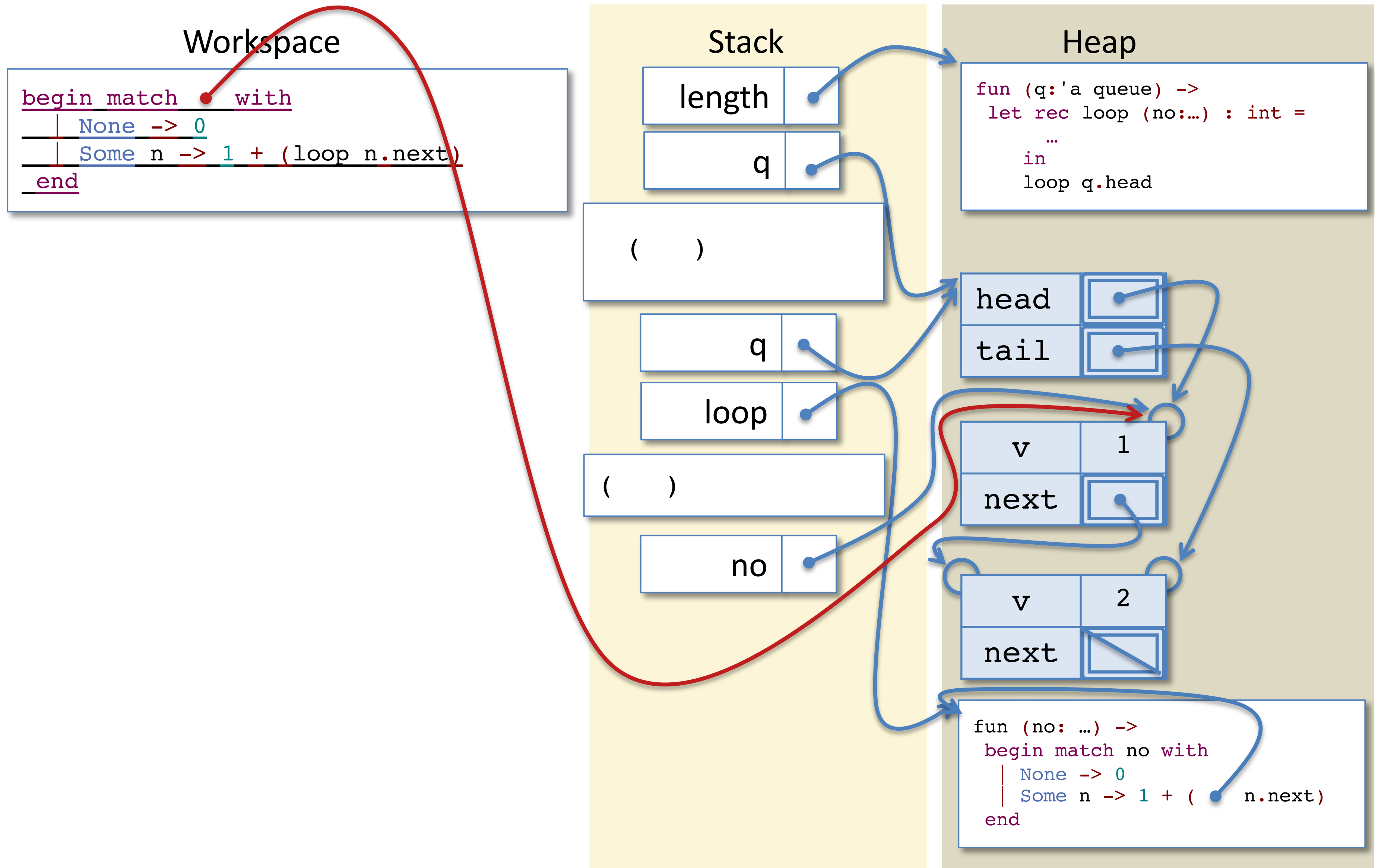
After a few steps…

(From here on, we'll take some shortcuts in the ASM animations.)

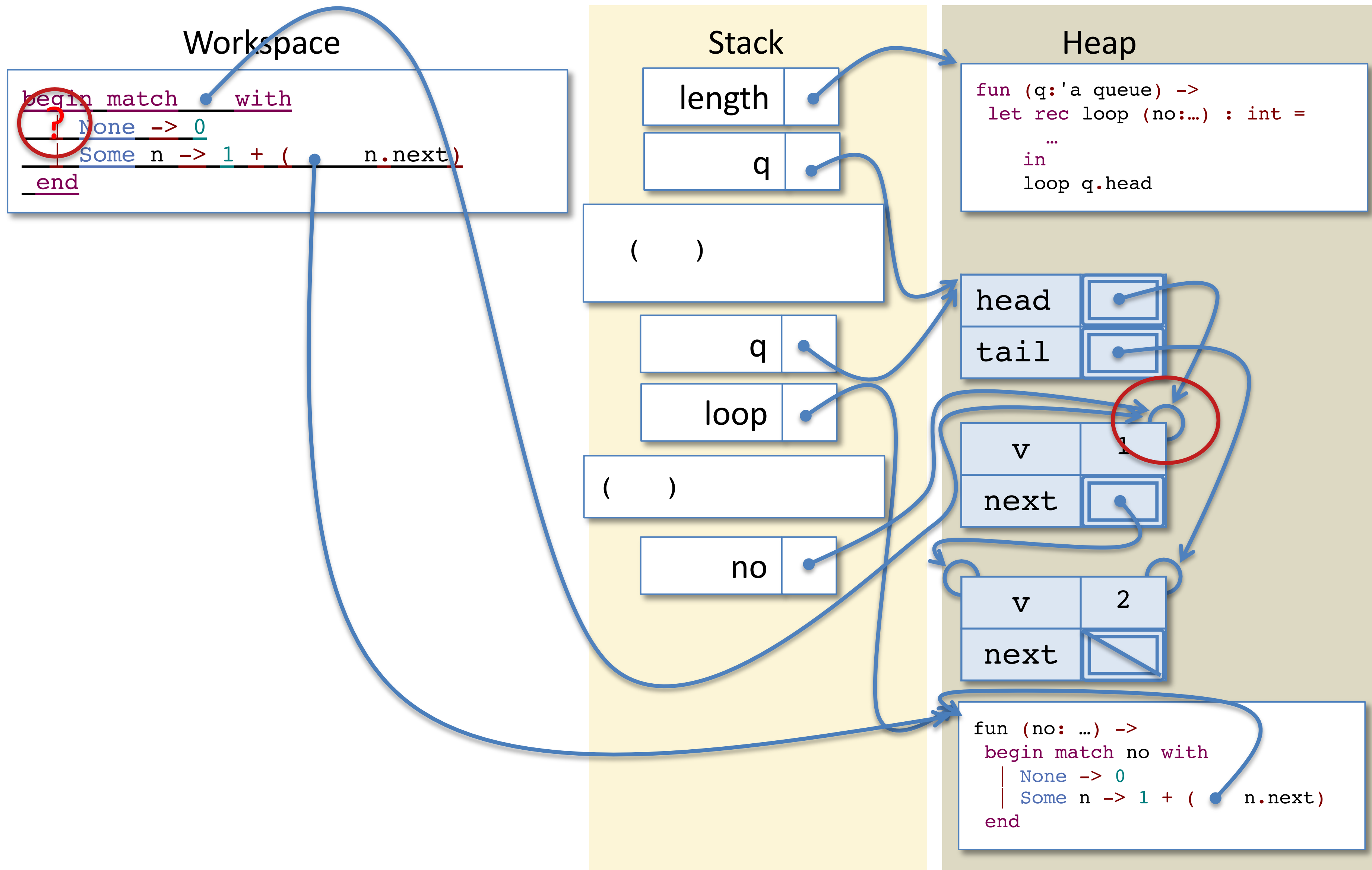
Evaluating length



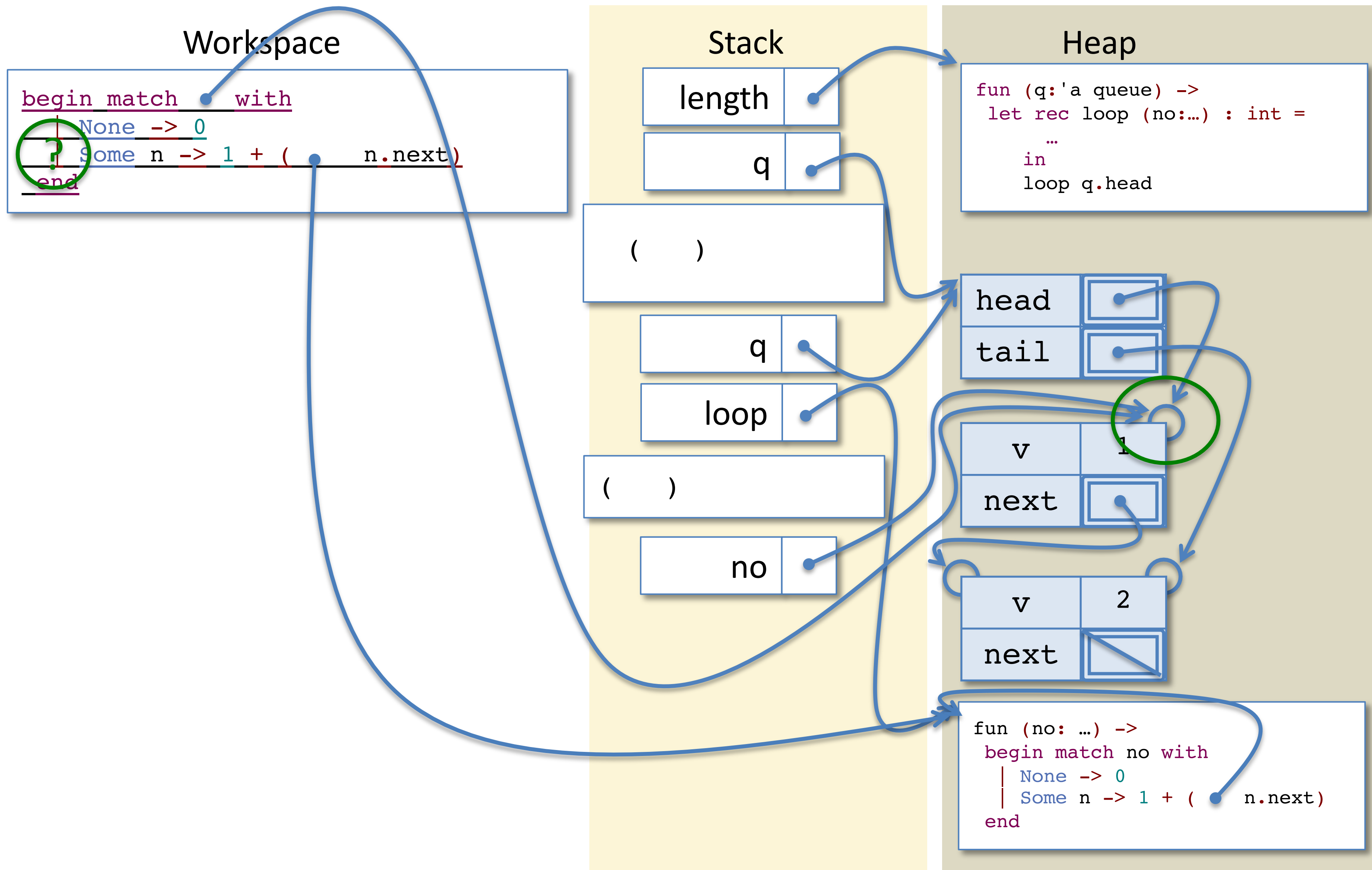
Evaluating length



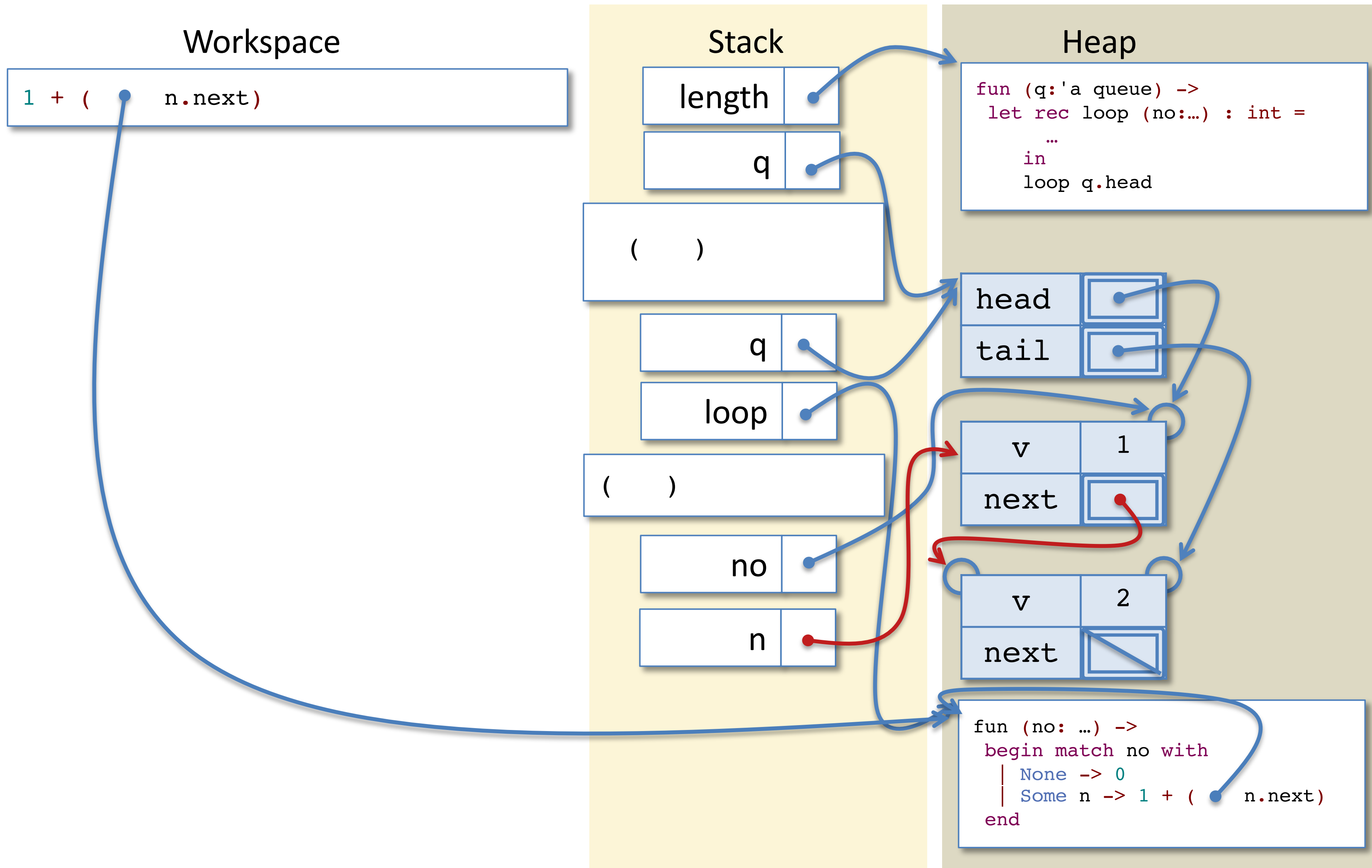
Evaluating length



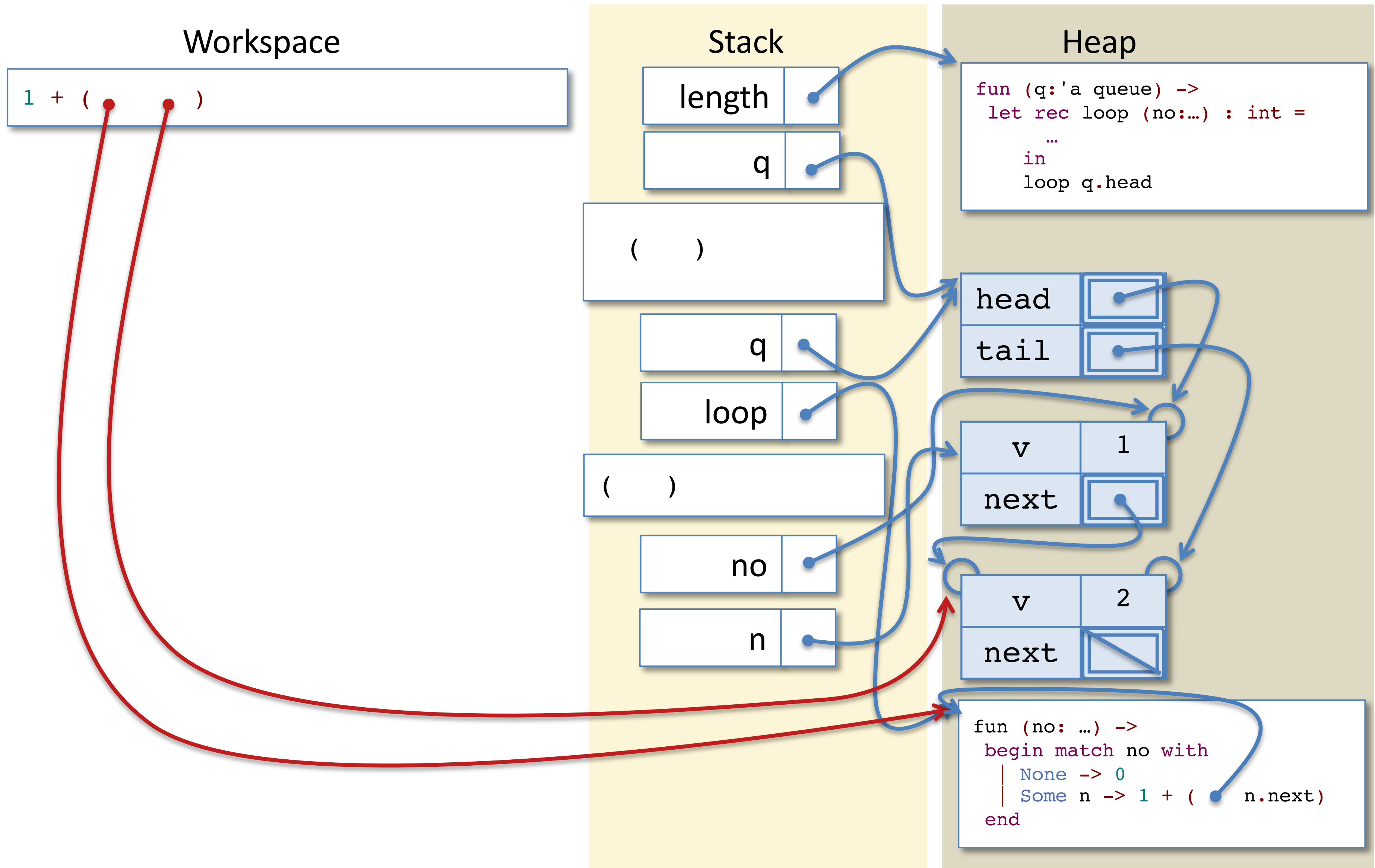
Evaluating length



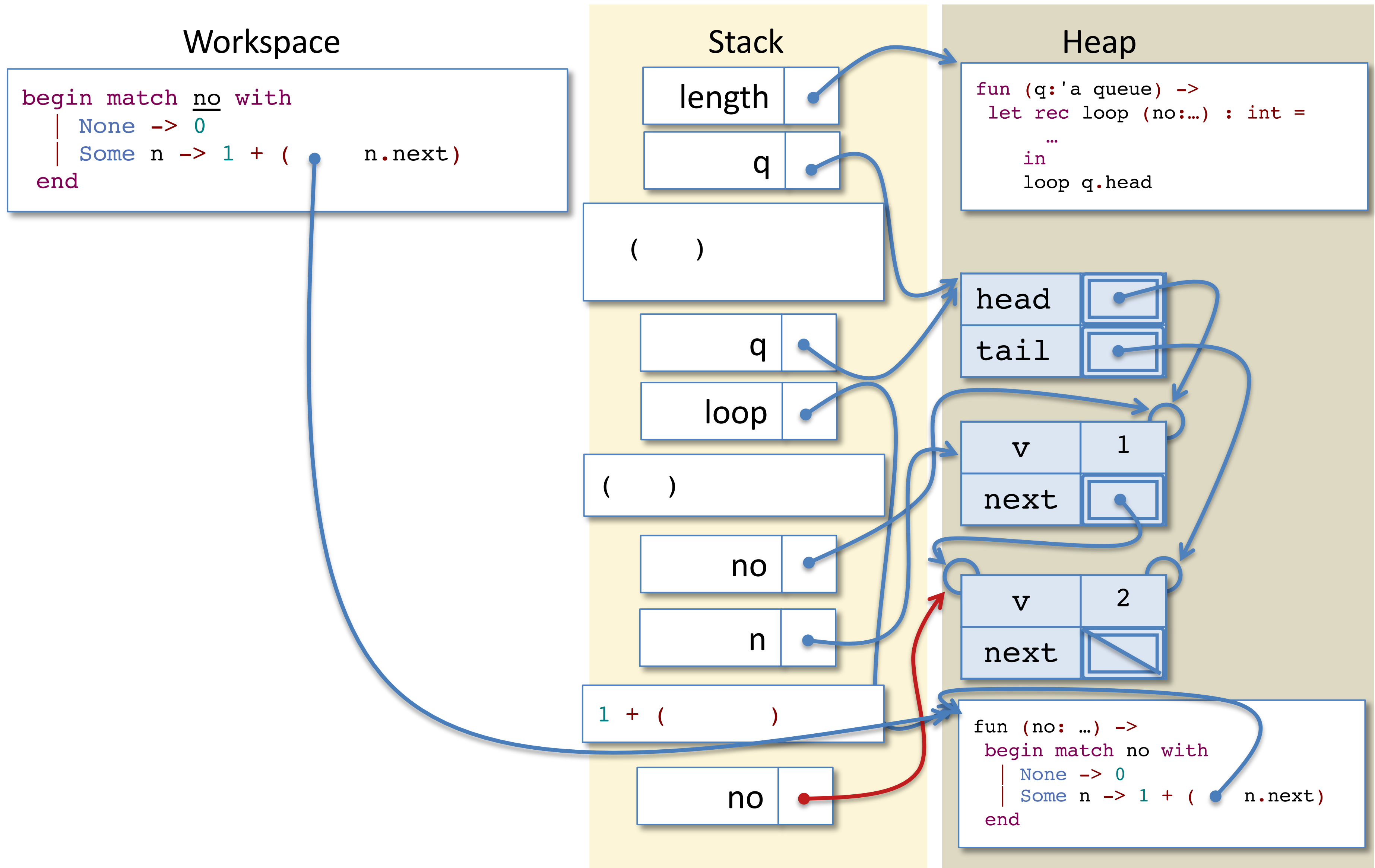
Evaluating length



Evaluating length

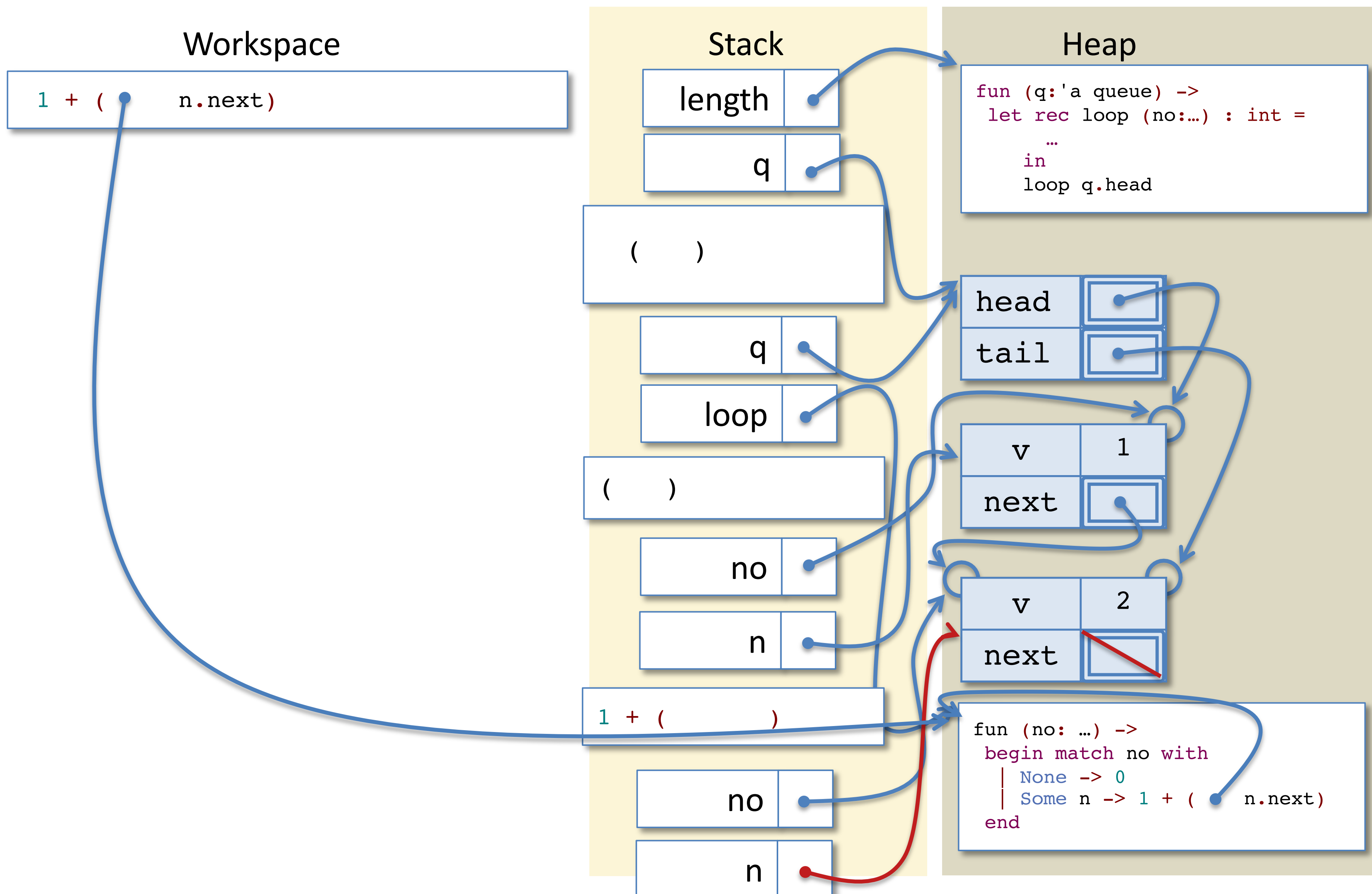


Evaluating length

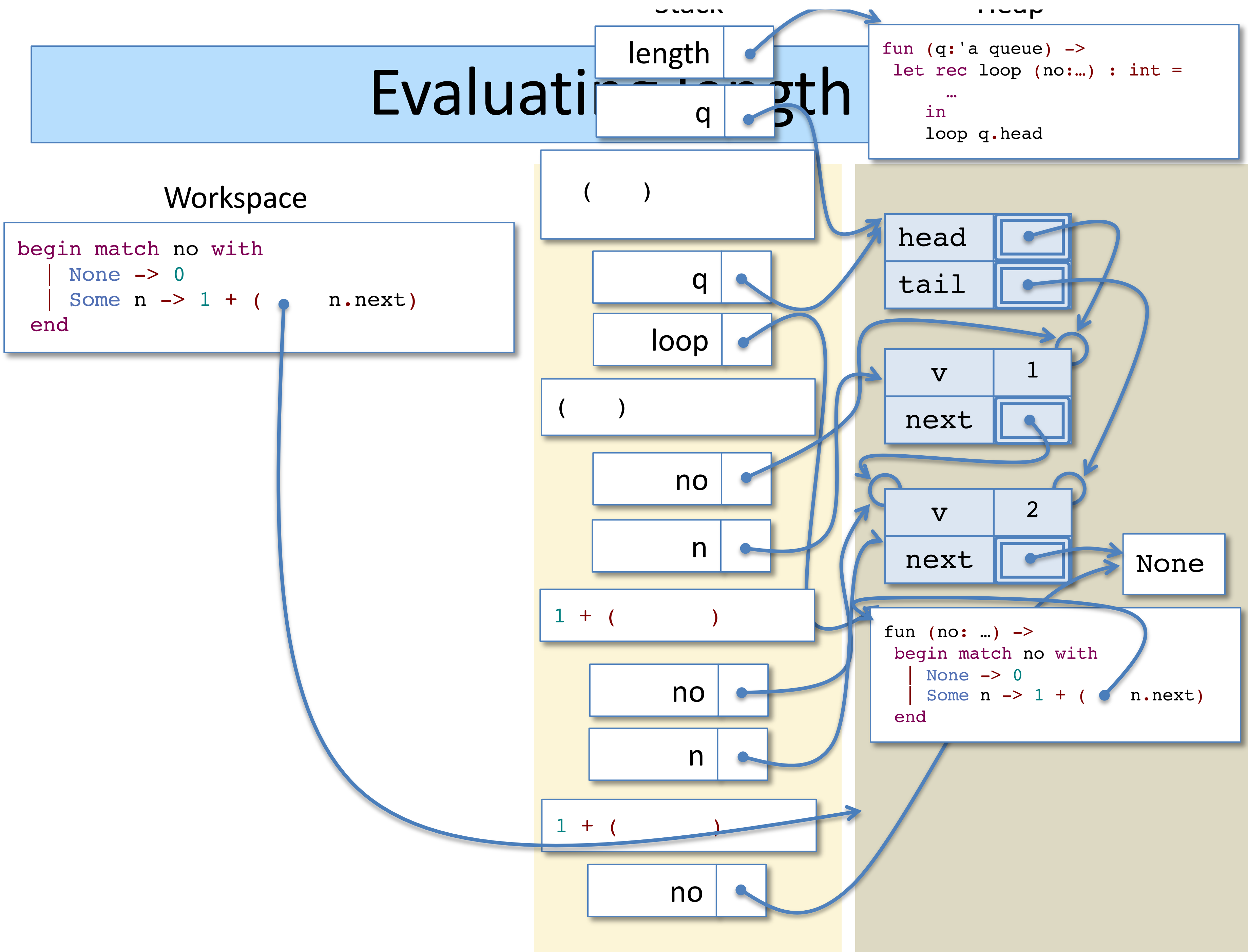


…after a few steps…

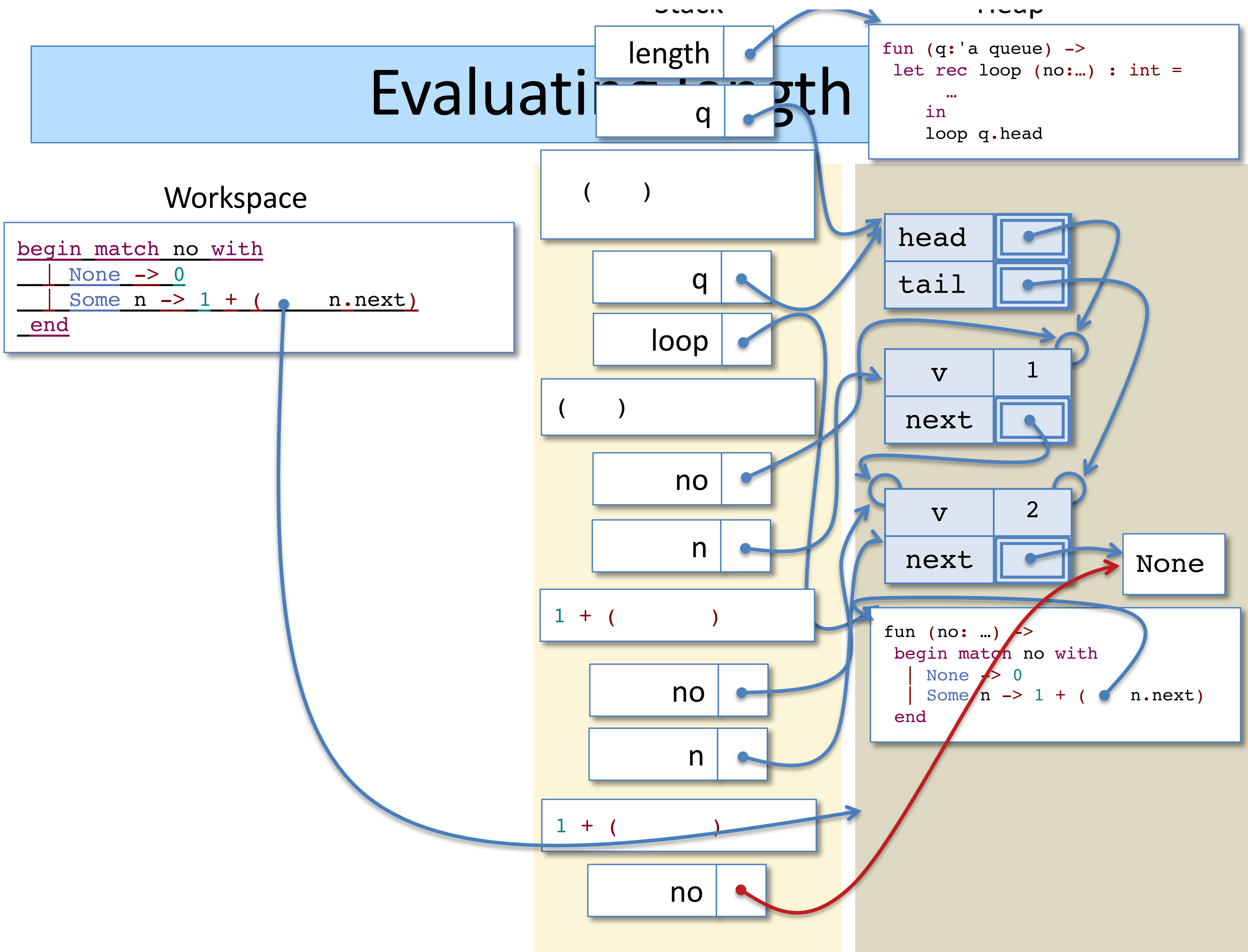
Evaluating length

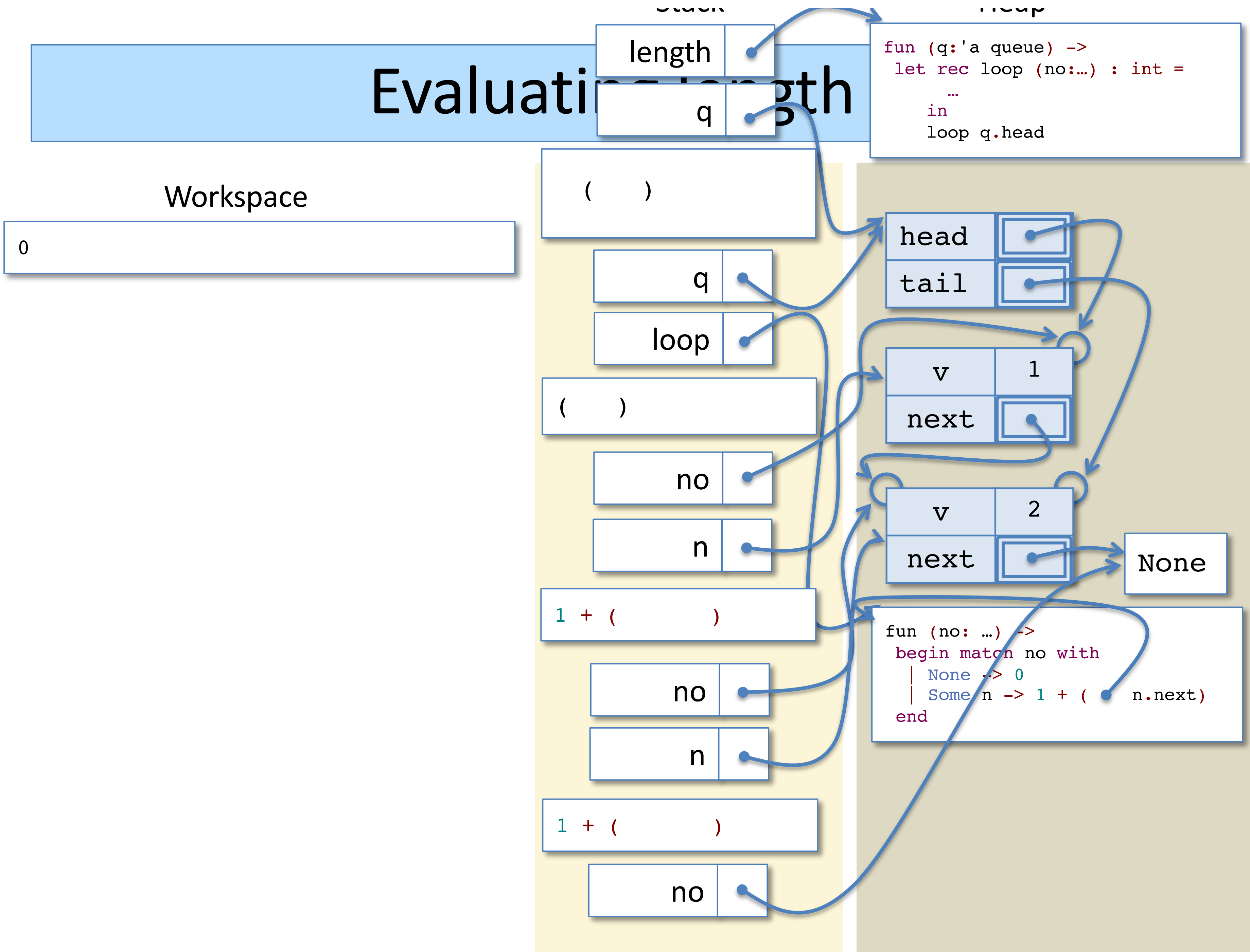


…after a few more steps…

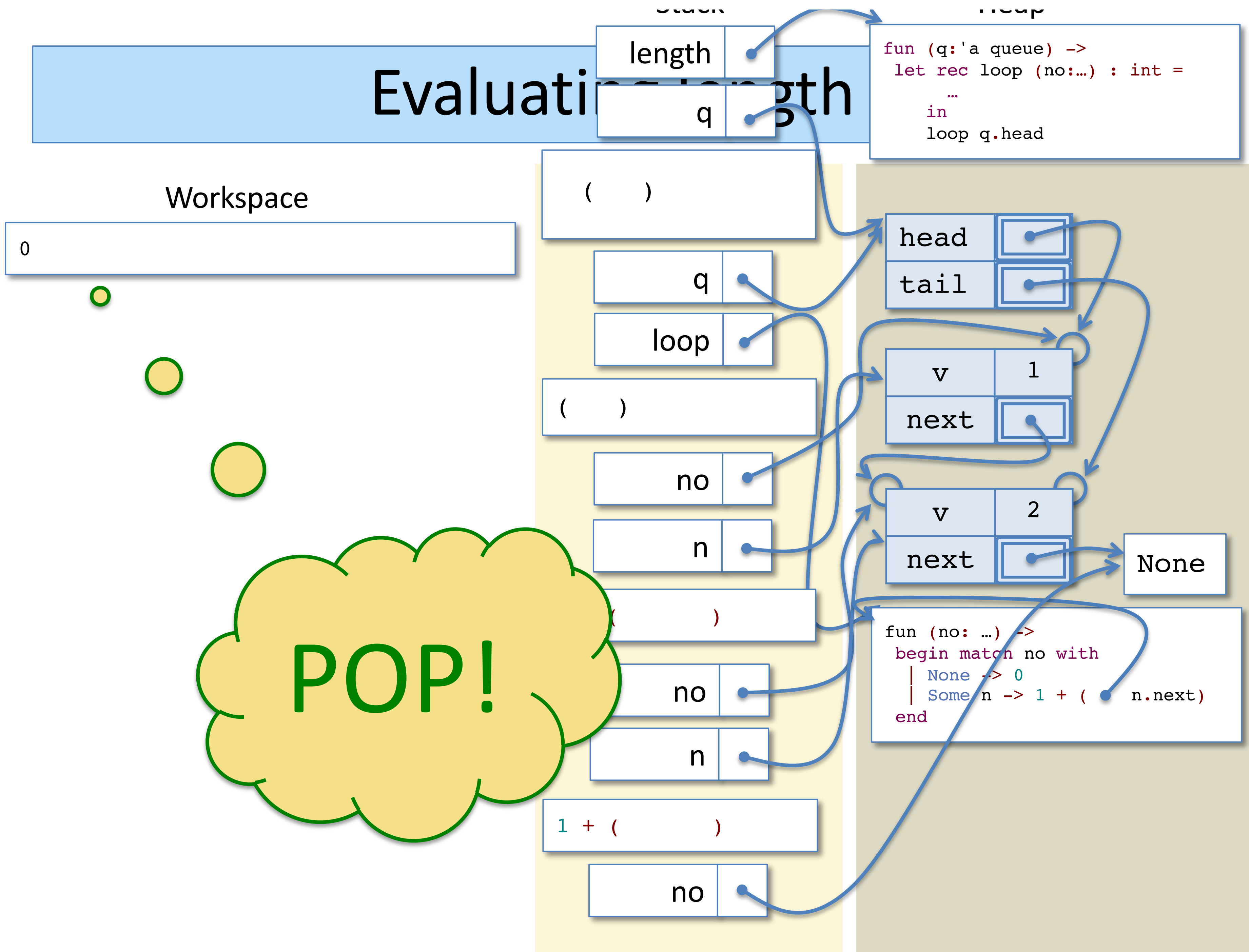


Evaluating length
Stack
Heap
Workspace
begin match no with
| None -> 0
| Some n -> 1 + (n.next)
end
length
q
( )
q
loop
( )
no
n
1 + ( )
no
n
1 + ( )
no
fun (q:'a queue) ->
let rec loop (no:…) : int =
…
in
loop q.head
head
tail
v
1
next
v
2
next
None
fun (no: …) ->
begin match no with
| None -> 0
| Some n -> 1 + (n.next)
end


Stack
Heap
Evaluating length
length
q
fun (q:'a queue) ->
let rec loop (no:…) : int =
…
in
loop q.head
Workspace
begin match no with
| None -> 0
| Some n -> 1 + (n.next)
end
( )
head
tail
q
loop
v
1
next
( )
no
n
v
2
next
None
1 + ( )
fun (no: …) ->
begin match no with
| None -> 0
| Some n -> 1 + (n.next)
end
no
n
1 + ( )
no



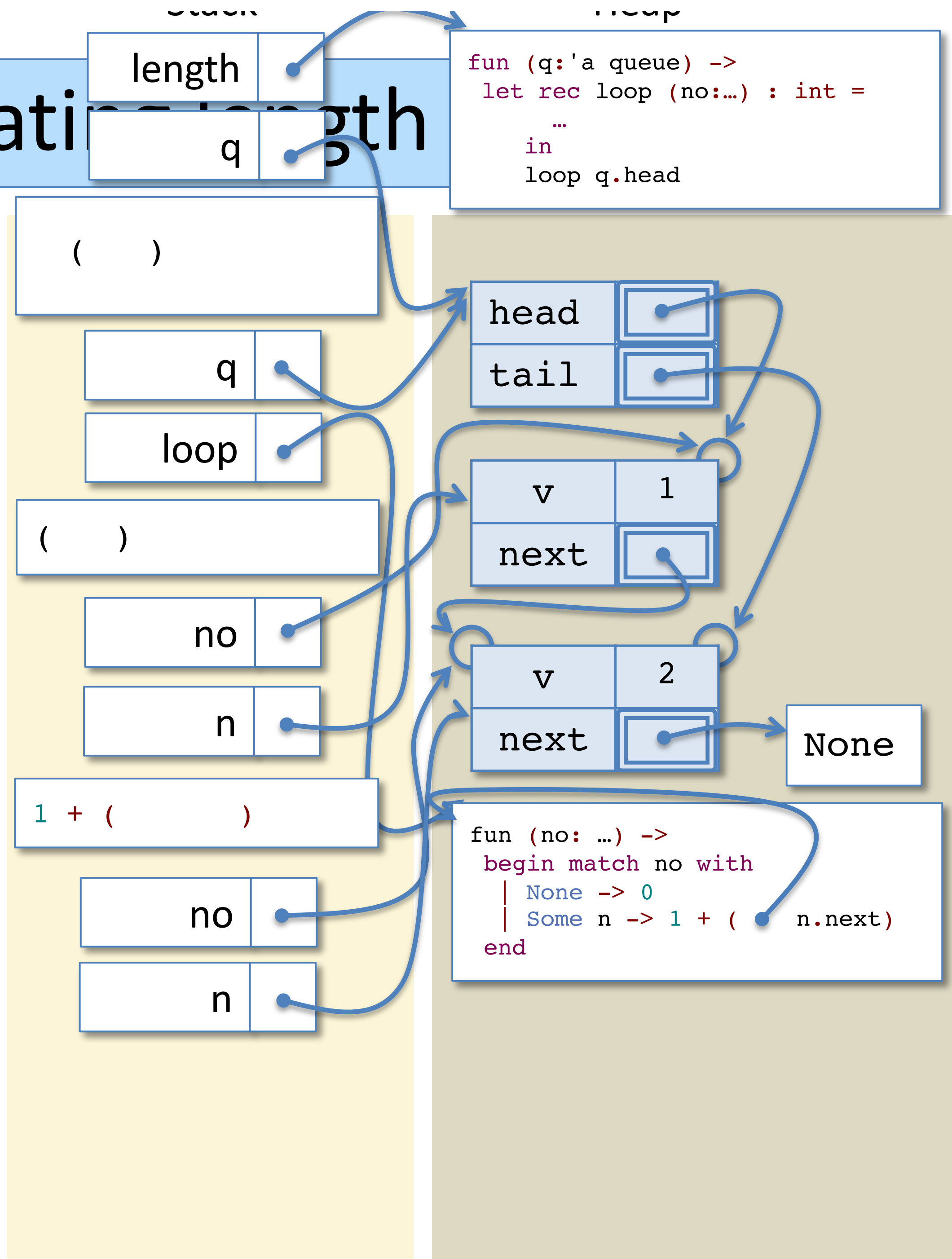
Stack
Heap
Evaluating length
length
q
fun (q:'a queue) ->
let rec loop (no:…) : int =
…
in
loop q.head
Workspace
0
( )
q
loop
( )
head
tail
v
1
next
no
n
v
2
next
None
1 + ( )
fun (no: …) ->
begin match no with
| None -> 0
| Some n -> 1 + (n.next)
end
no
n
1 + ( )
no


Evaluating length
Stack
Heap
length
q
fun (q:'a queue) ->
let rec loop (no:…) : int =
…
in
loop q.head
Workspace
0
( )
q
loop
( )
no
n
( )
no
n
1 + ( )
no
head
tail
v
1
next
v
2
next
None
fun (no: …) ->
begin match no with
| None -> 0
| Some n -> 1 + (n.next)
end
POP!

Evaluating length

Workspace

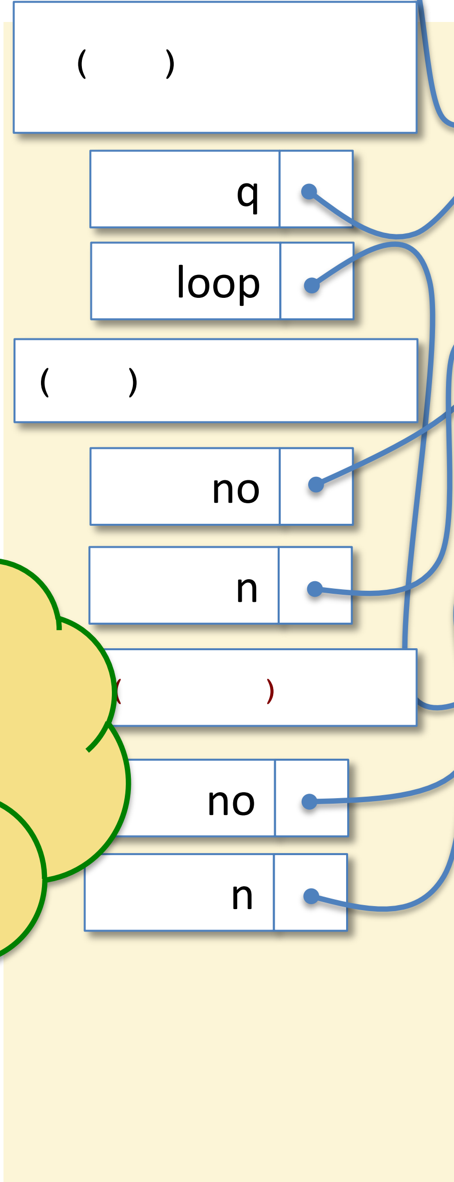
`1 + 0`



Evaluating length

Workspace

```
1
```

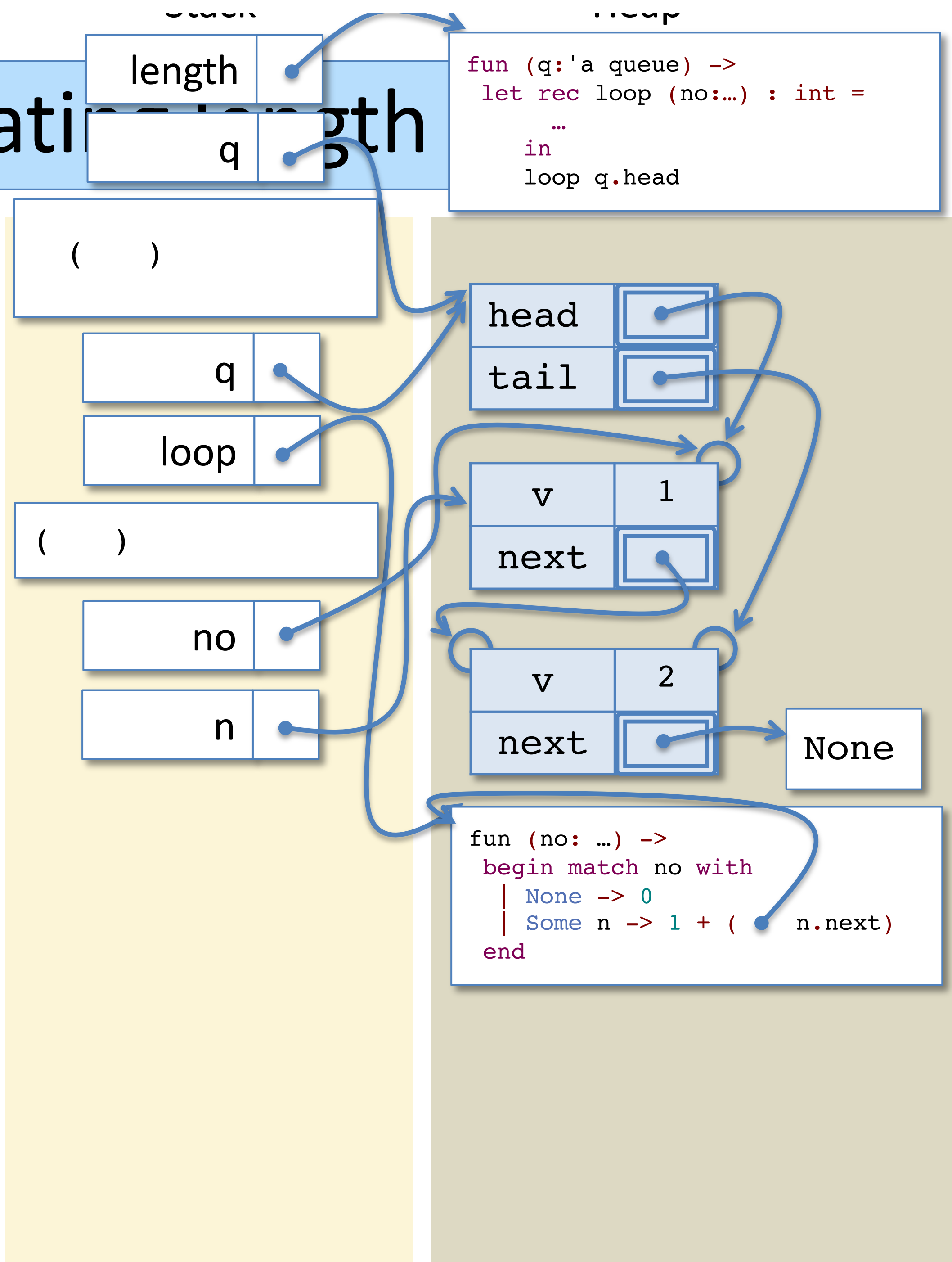


POP!

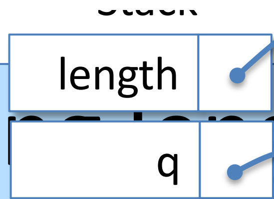
Evaluating length

Workspace

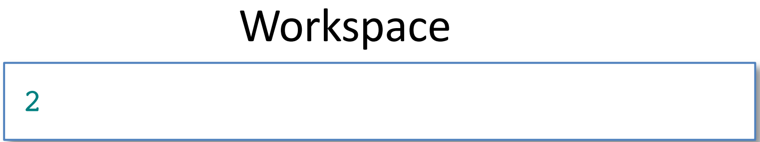
`1 + 1`

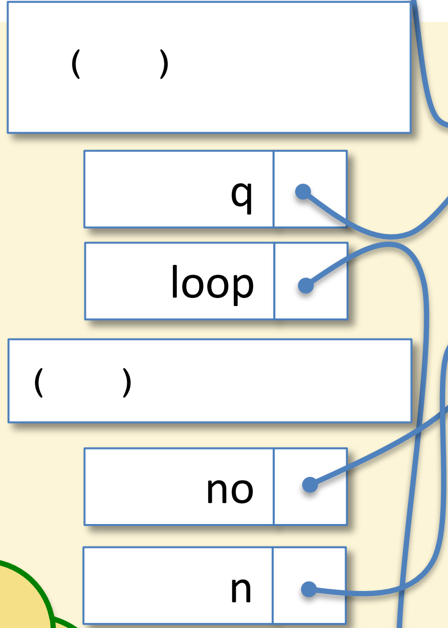


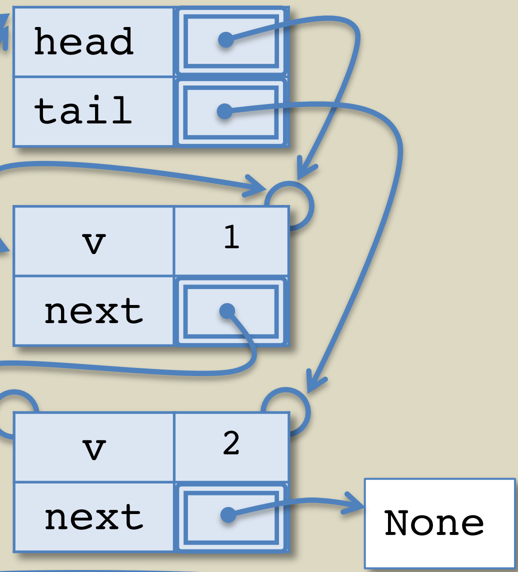
Evaluating length




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








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

POP!

Evaluating length

Workspace

```
2
```

Heap

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

head	
tail	

v	1
next	

v	2
next	

None

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

POP!

Evaluating length

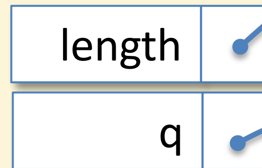
Workspace

```
2
```

Stack

length

q

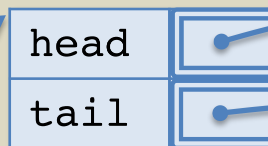


Heap

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

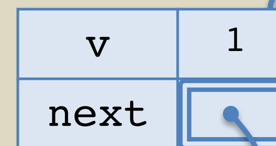
head

tail



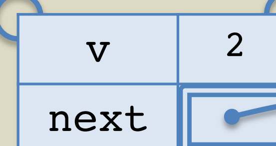
v 1

next



v 2

next



None

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

DONE!

Iteration

Using tail calls for loops

`length` (using iteration)

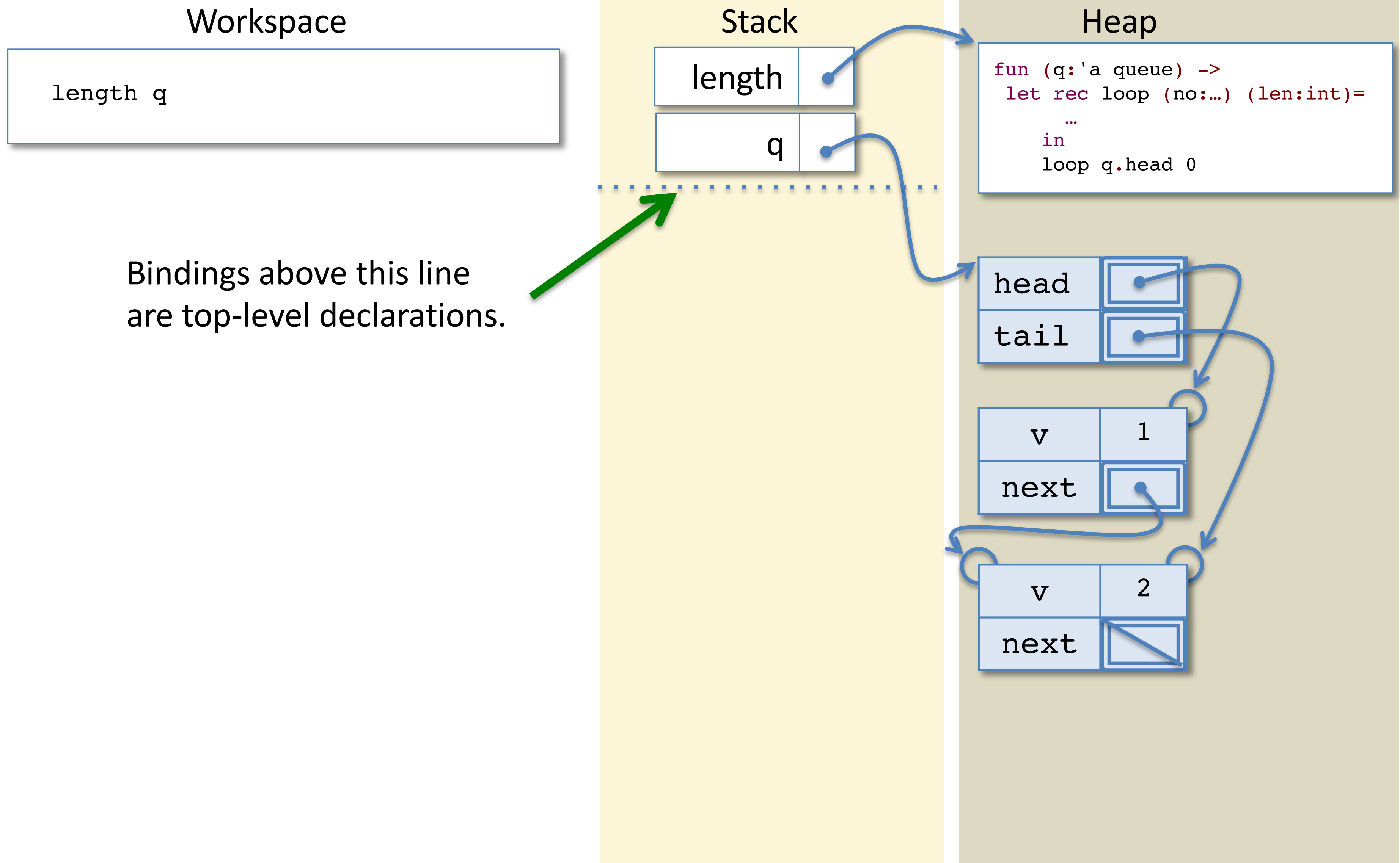
```
(* Calculate the length of the list using iteration *)
let length (q:'a queue) : int =
  let rec loop (no:'a qnode option) (len:int) : int =
      begin match no with
        | None -> len
        | Some n -> loop n.next (1+len)
      end
  in
    loop q.head 0
```

- This implementation of `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 otherwise-empty workspace—the results of the call to `loop` never need to be used to compute another expression. In contrast, we had (1 + (`loop` …)) in the recursive version.

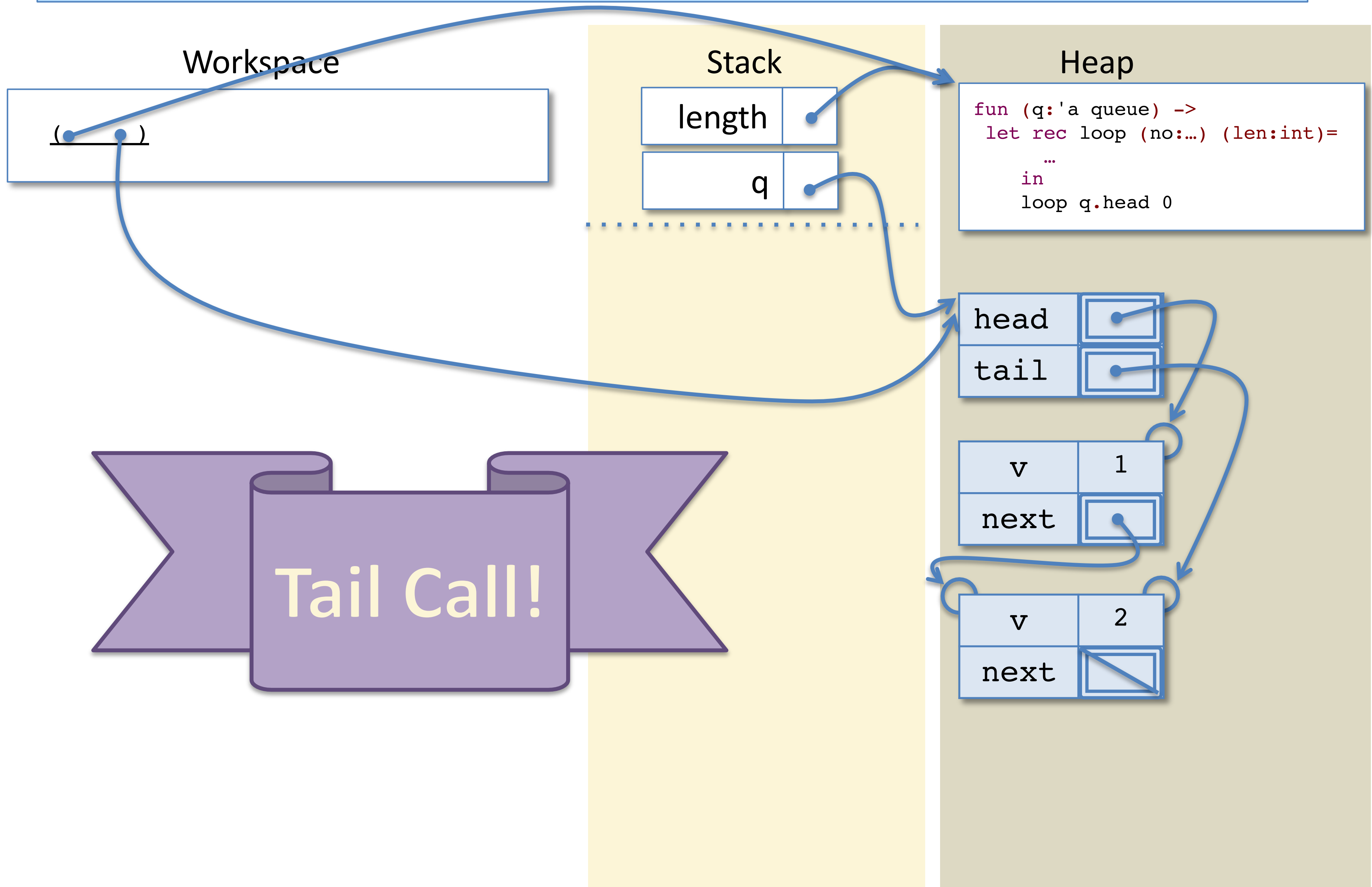
Tail Call Optimization

- 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 during evaluation of the current workspace 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

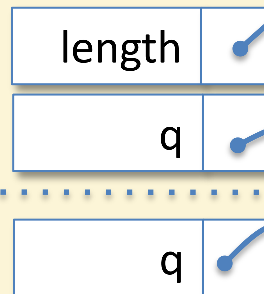


Tail Calls and Iterative length

Workspace

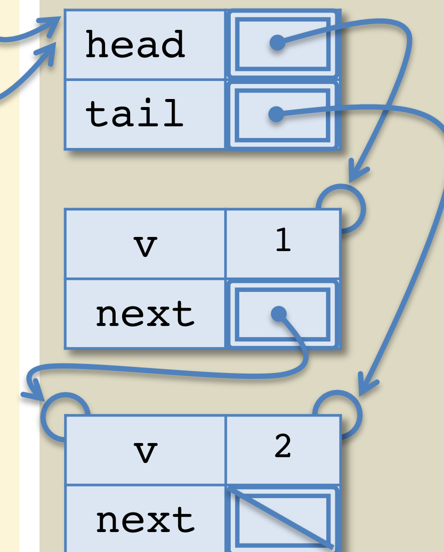
```
let rec loop (no:'a qnode option)
(len:int) : int =
      begin match no with
        | None -> len
        | Some n -> loop n.next (1+len)
      end
    in
    loop q.head 0
```

Stack



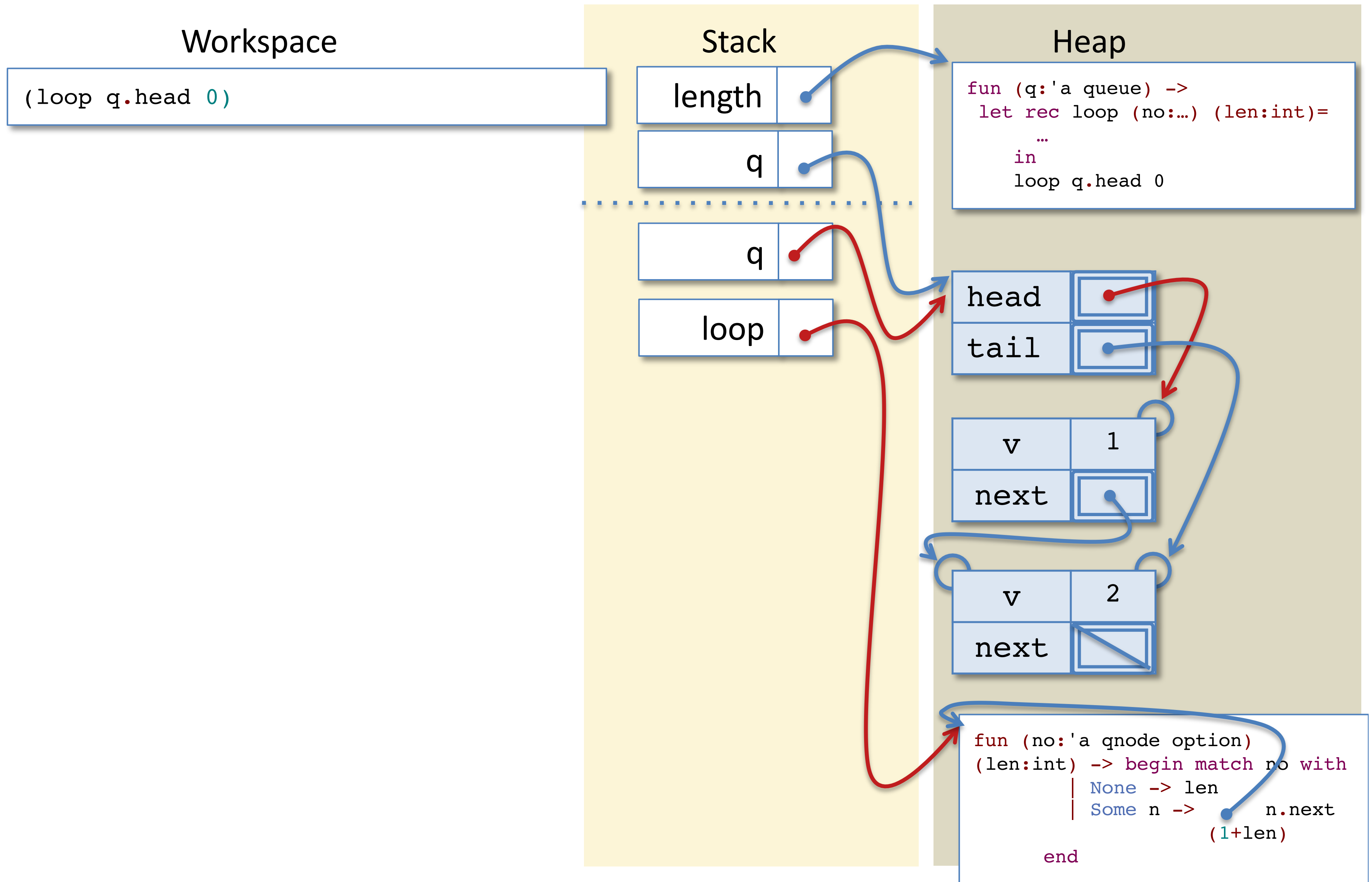
Heap

```
fun (q:'a queue) ->
 let rec loop (no:…) (len:int)=
    …
   in
   loop q.head 0
```

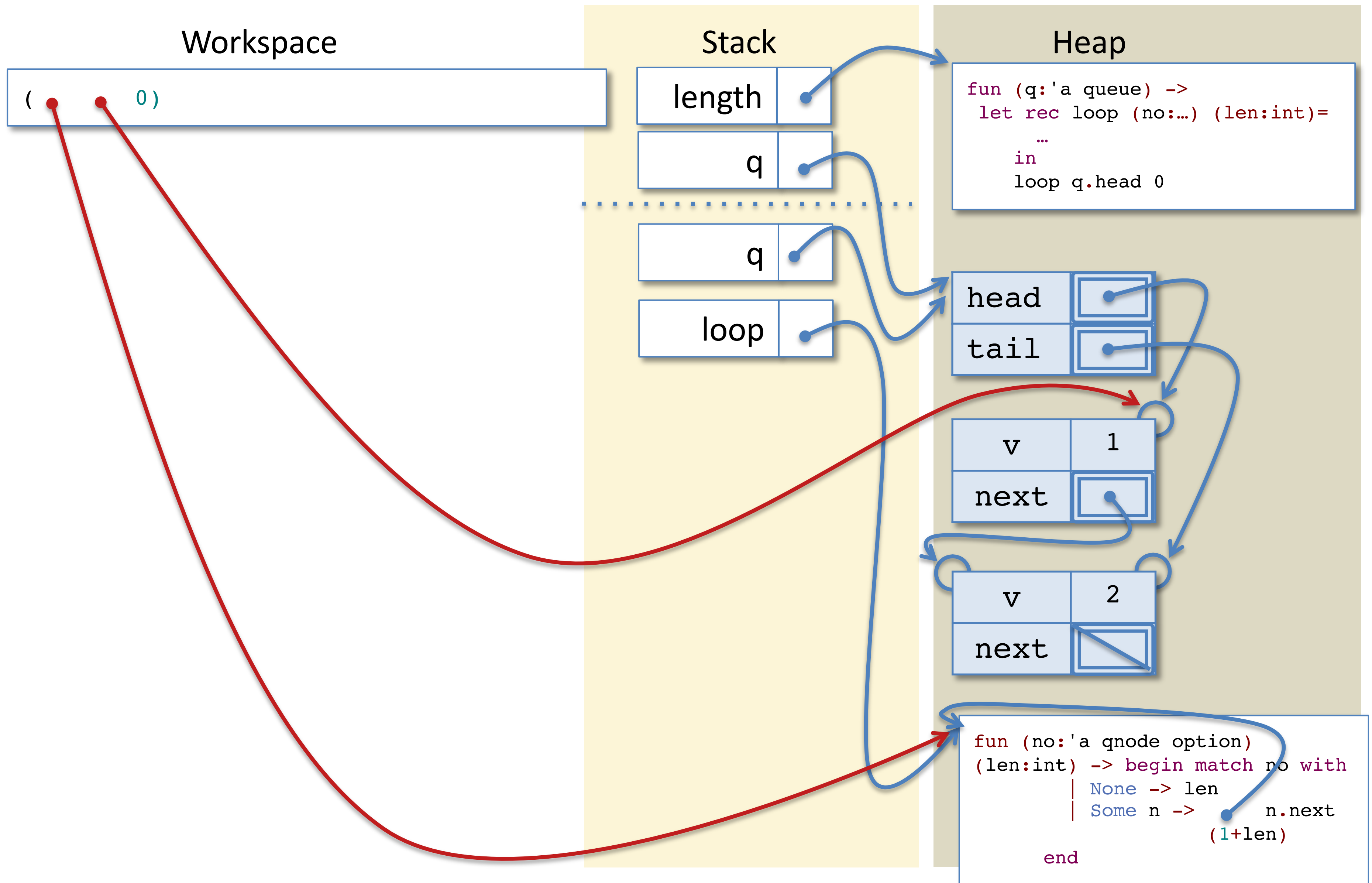


Note:
(1) No workspace is saved – there is no need do to that for tail calls
(2) We pop all the locals, up to the last saved workspace.
 (In this case, there weren't any.)

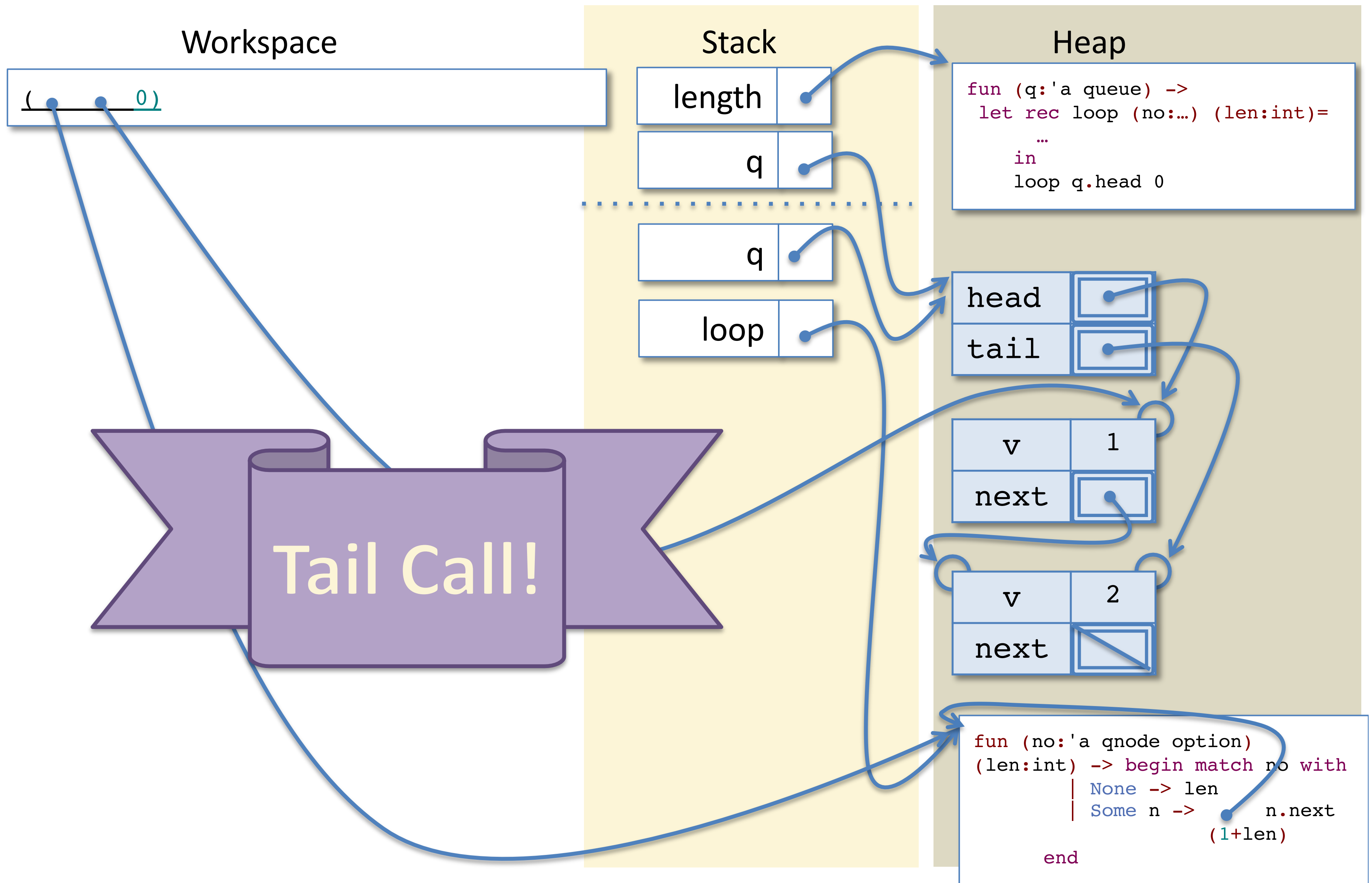
Tail Calls and Iterative length



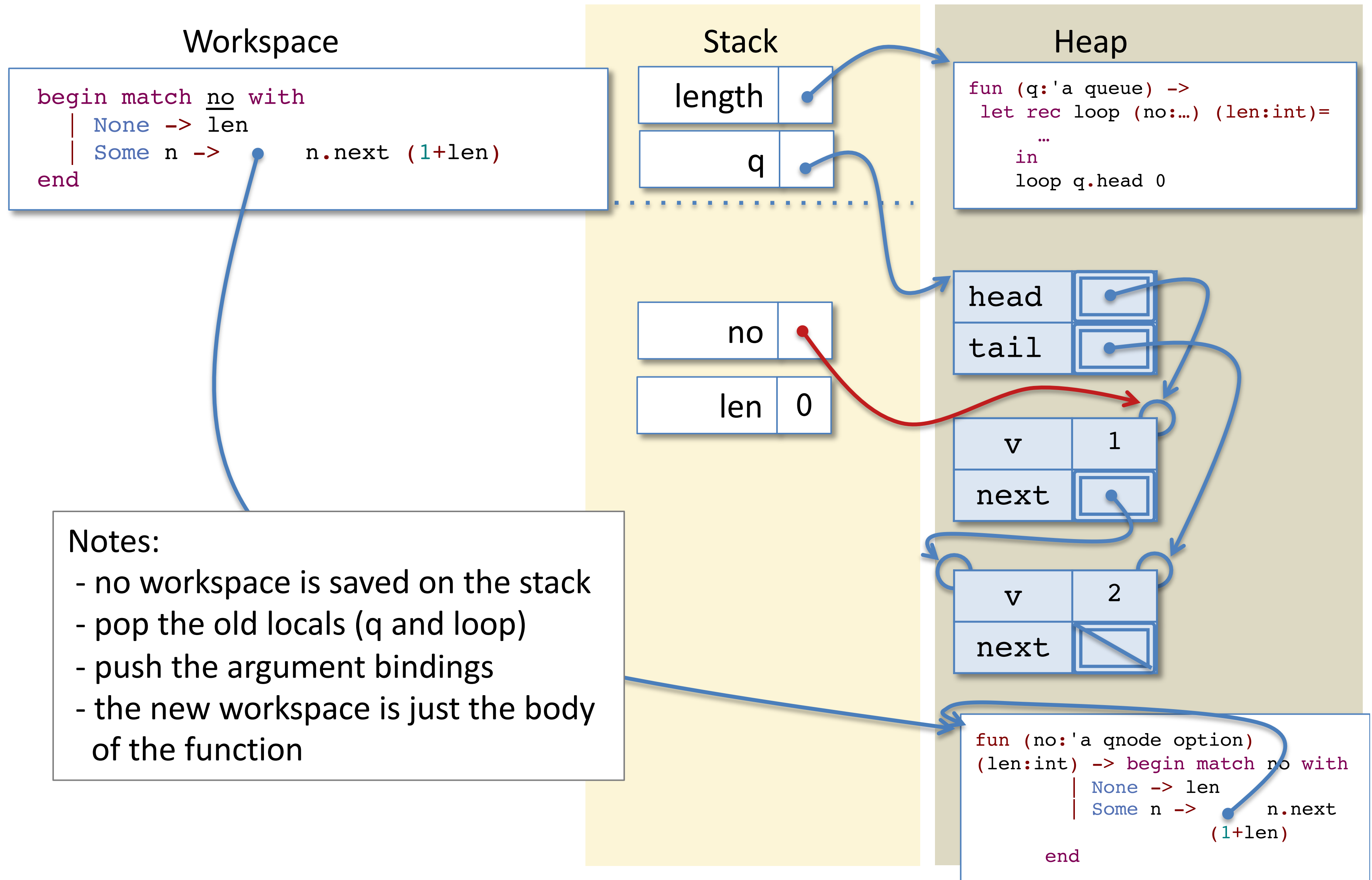
Tail Calls and Iterative length



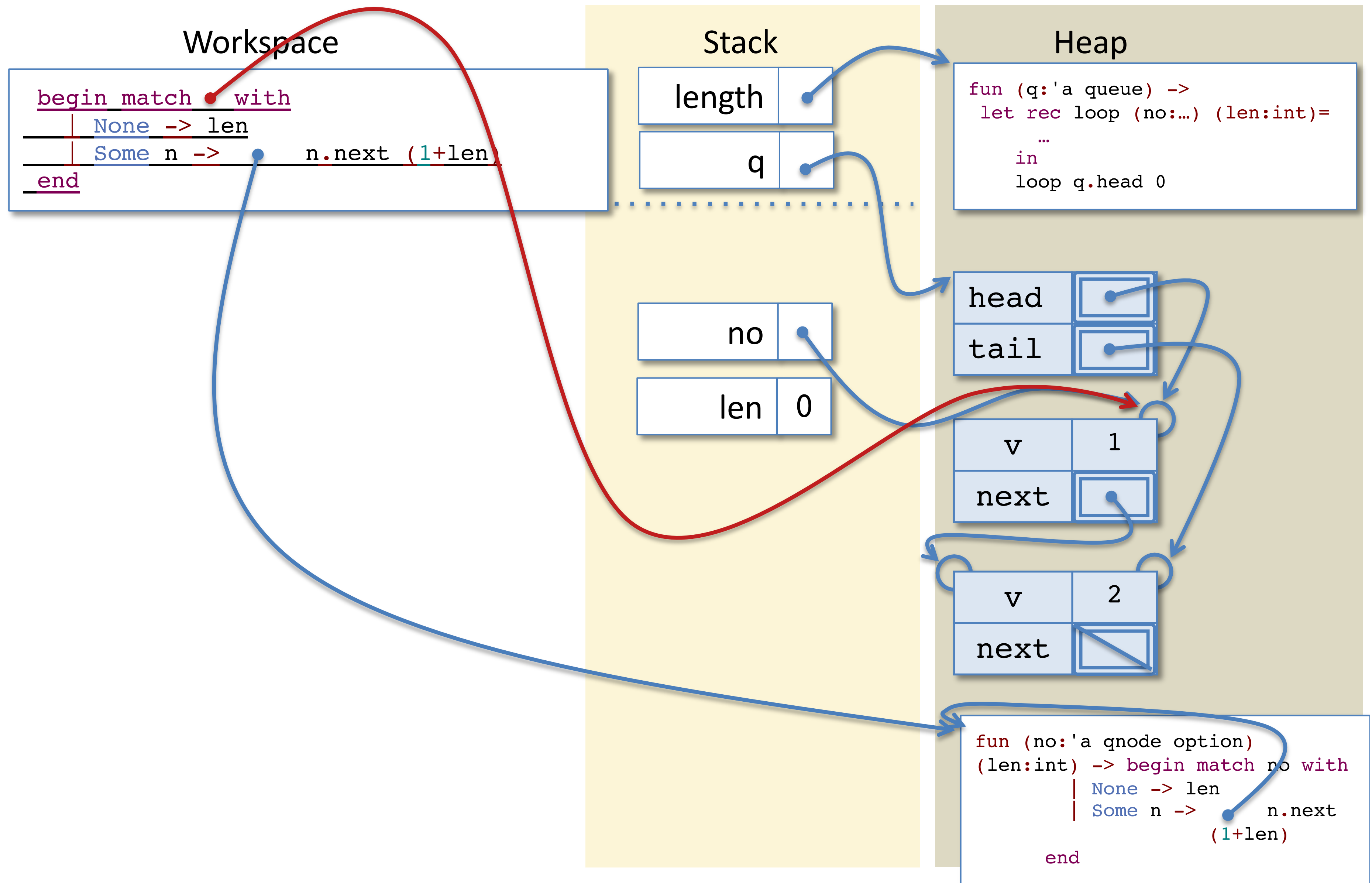
Tail Calls and Iterative length



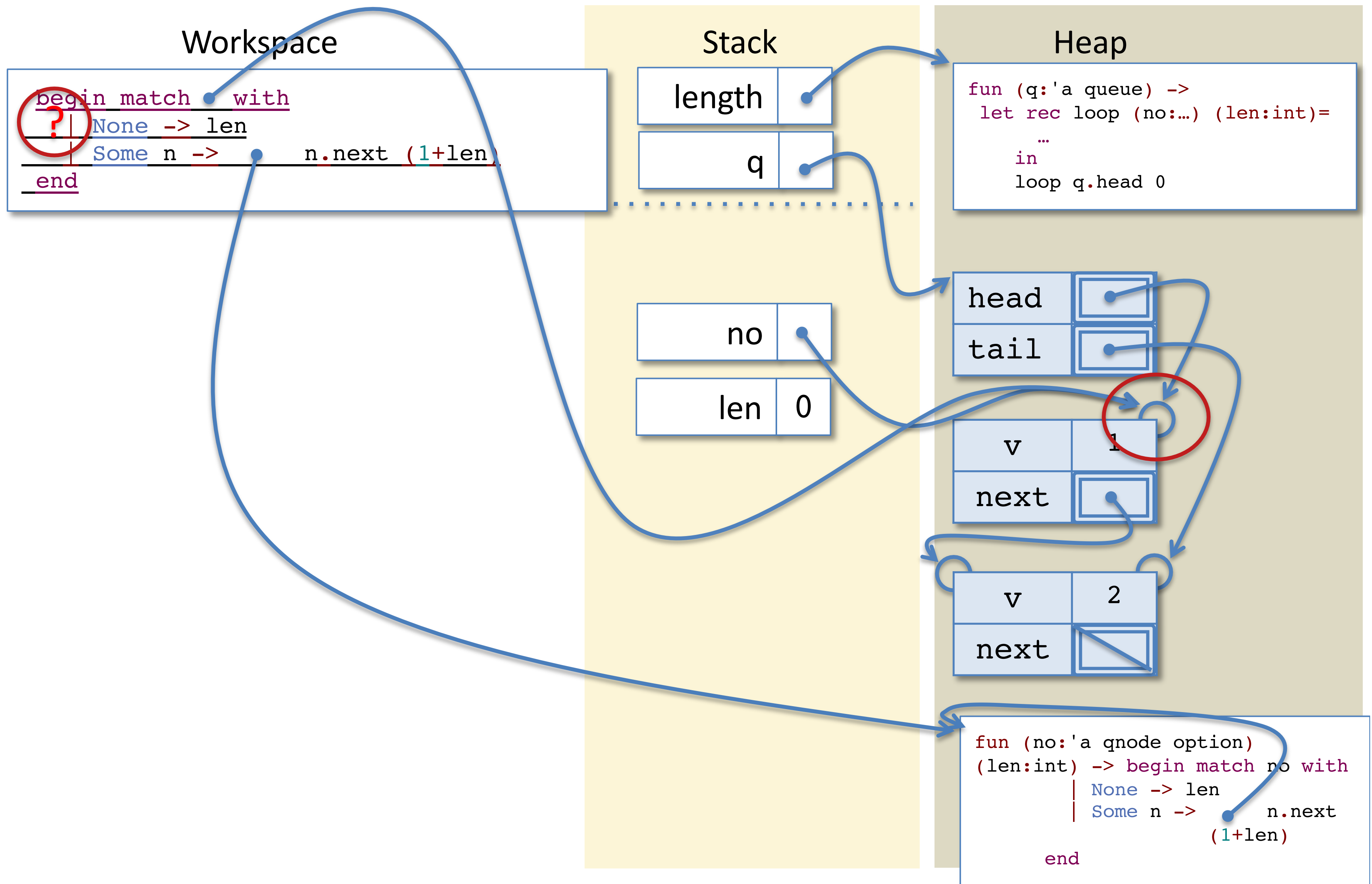
Tail Calls and Iterative length



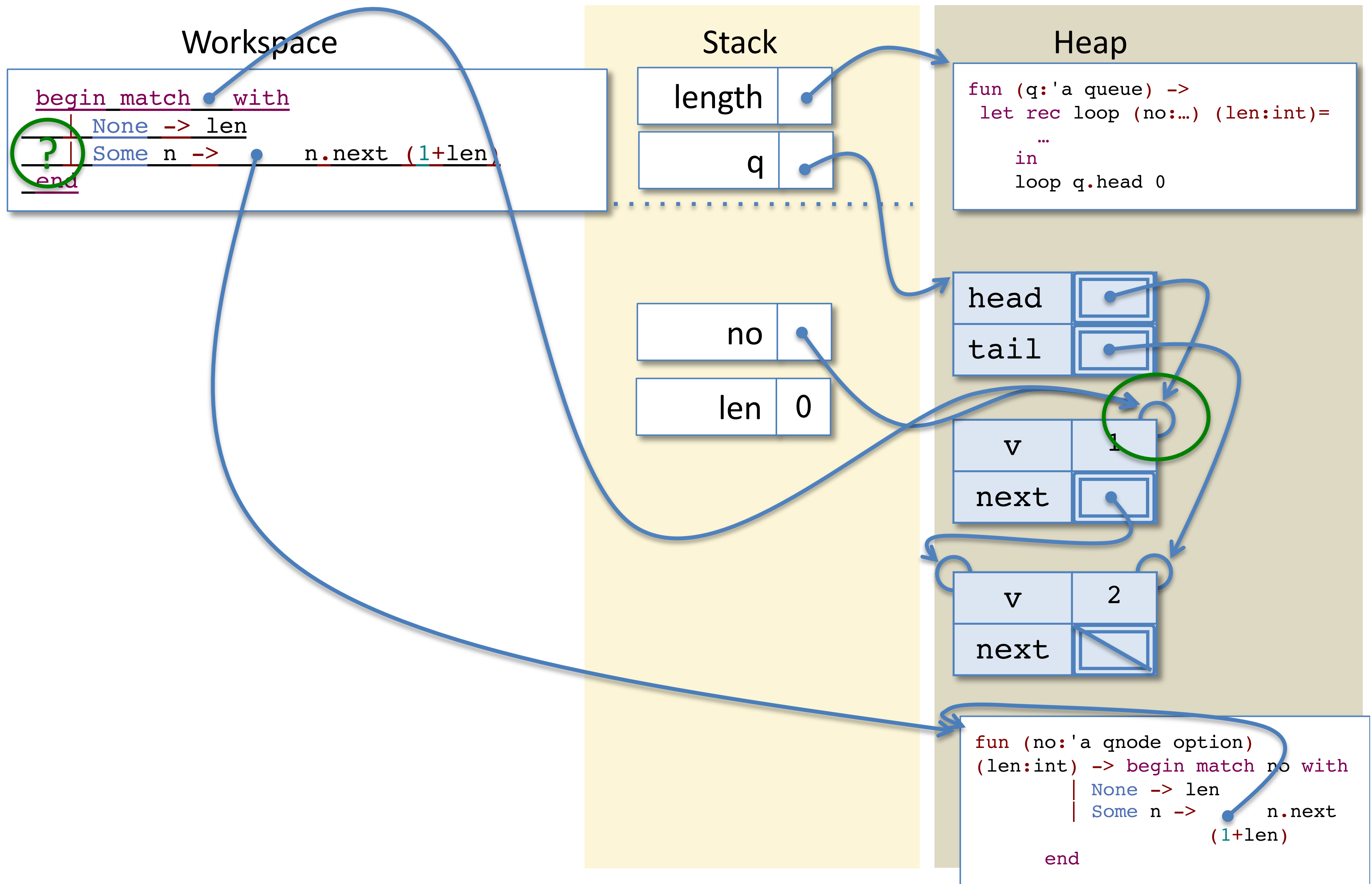
Tail Calls and Iterative length



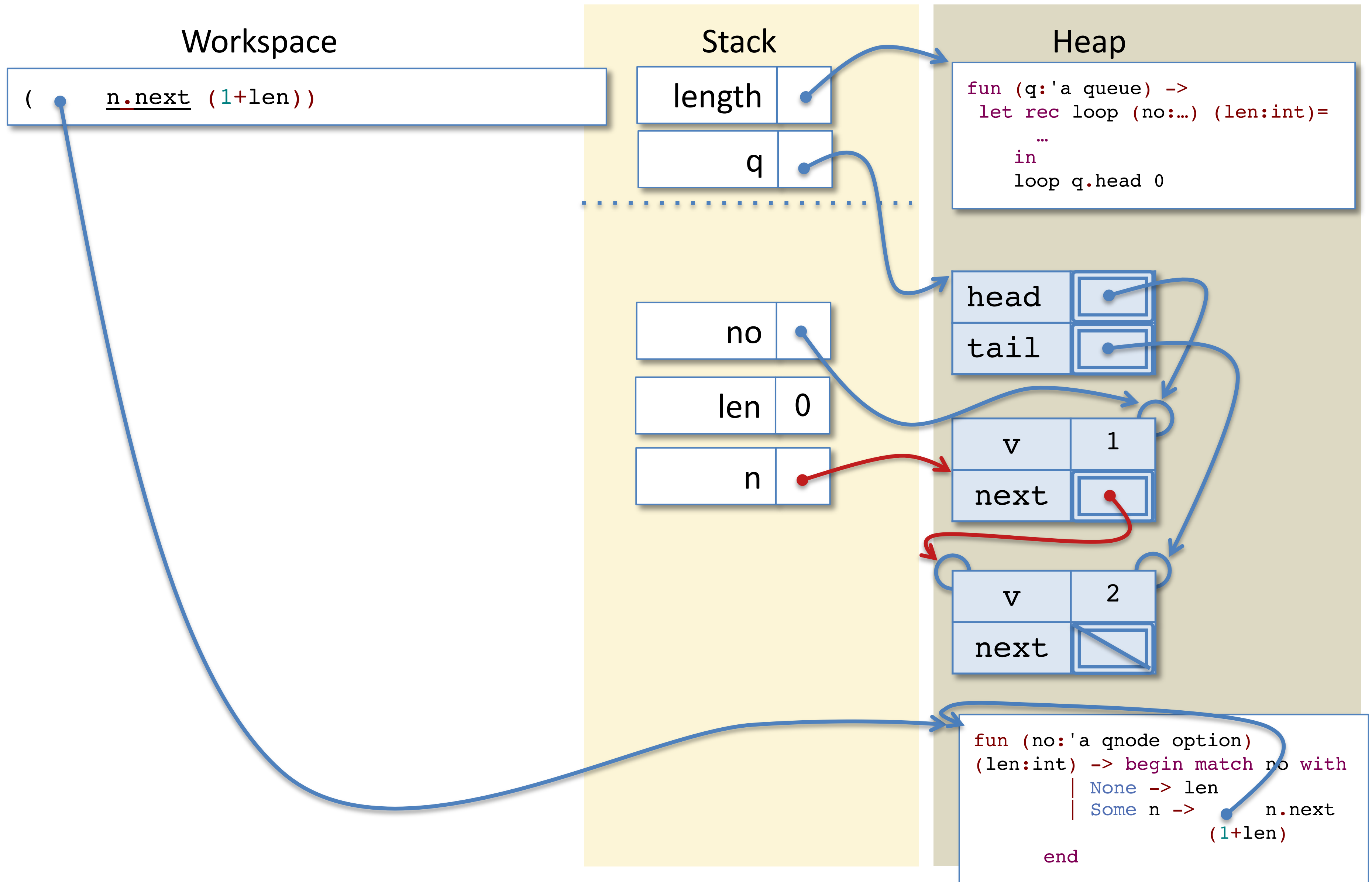
Tail Calls and Iterative length



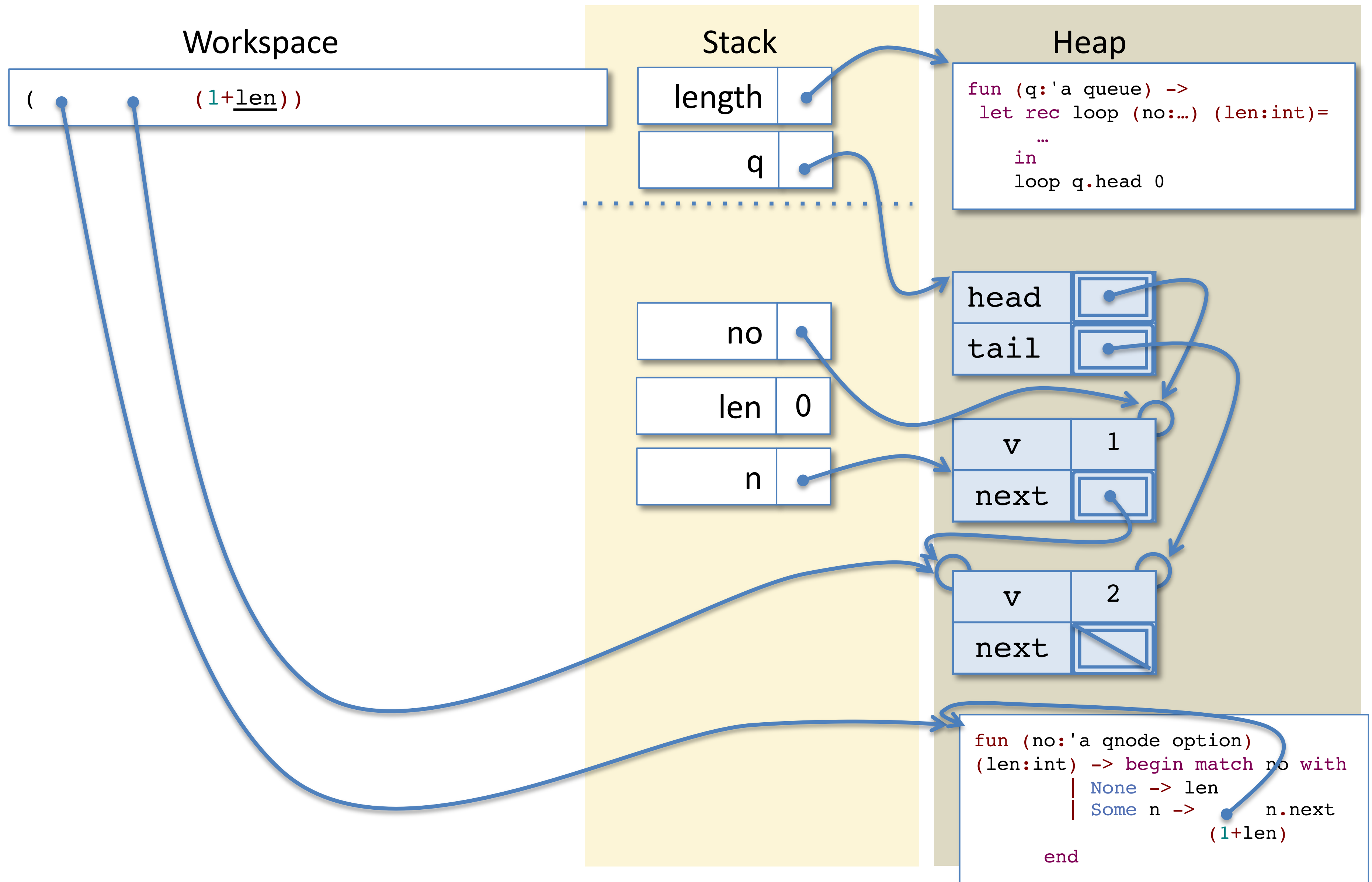
Tail Calls and Iterative length



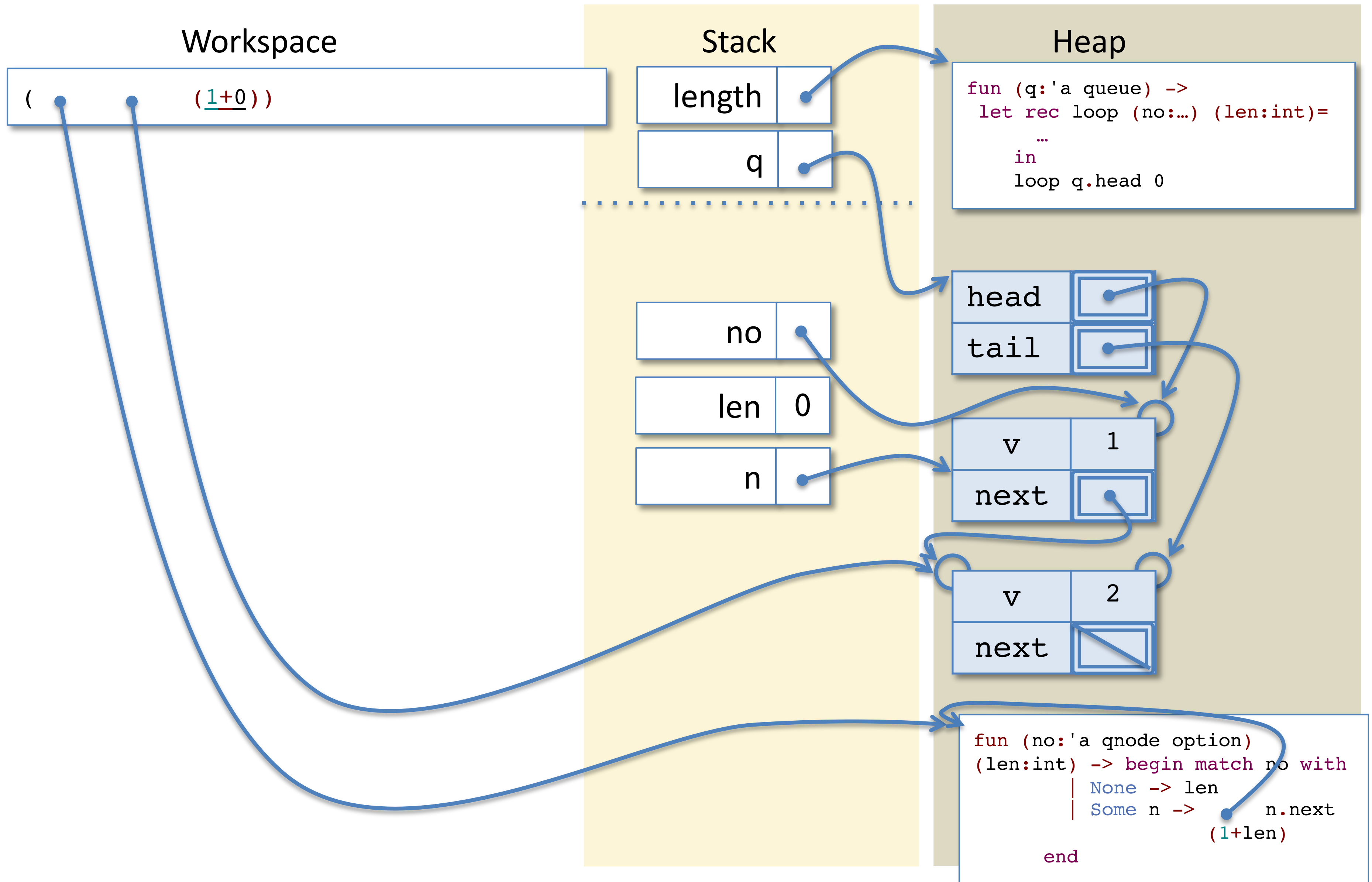
Tail Calls and Iterative length



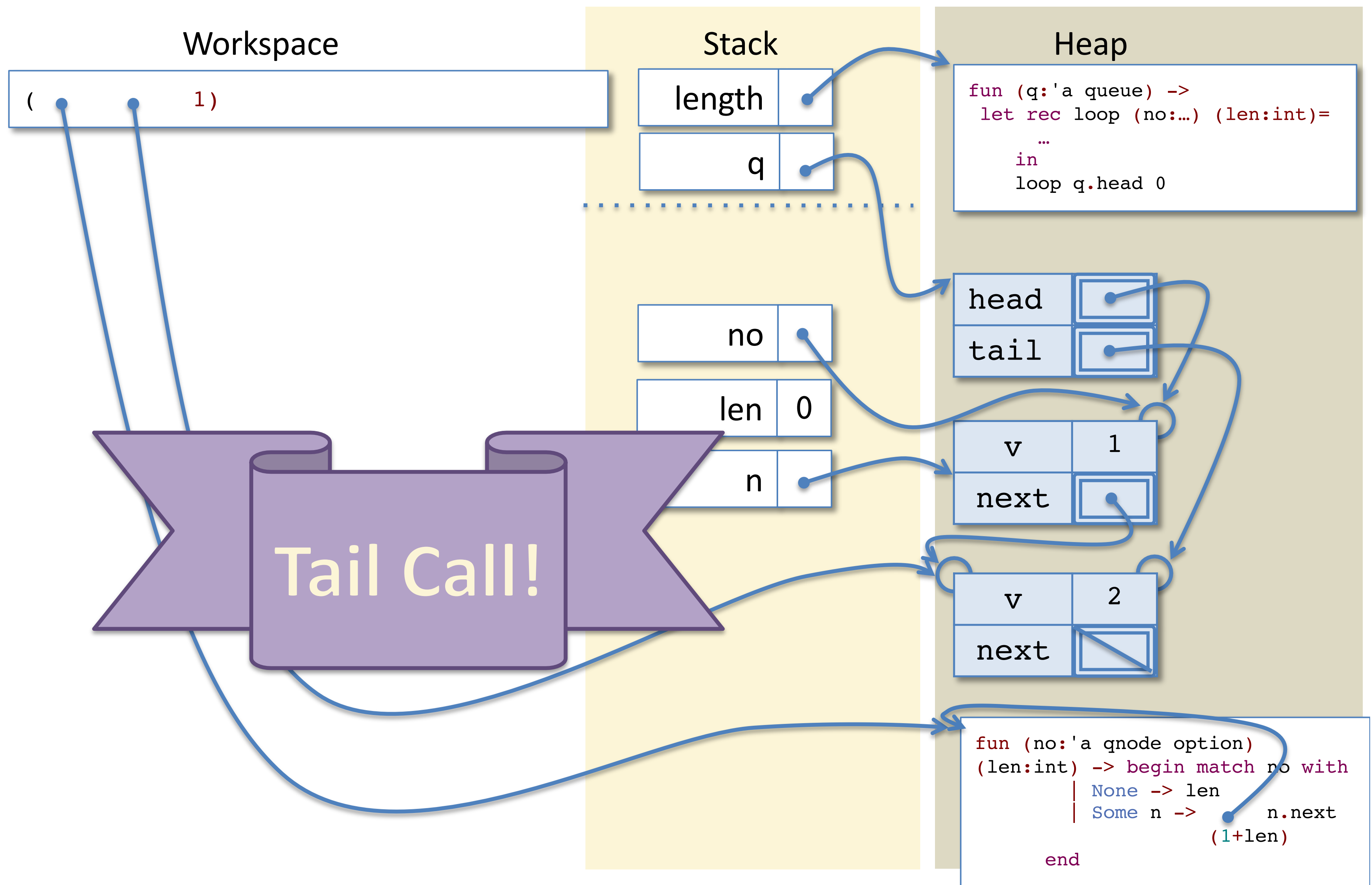
Tail Calls and Iterative length



Tail Calls and Iterative length



Tail Calls and Iterative length

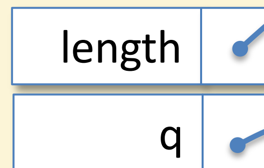


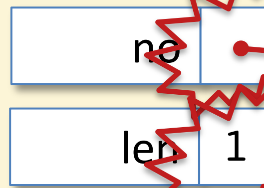
Tail Calls and Iterative length

Workspace

```
begin match no with
    | None -> len
    | Some n ->      n.next (1+len)
 end
```

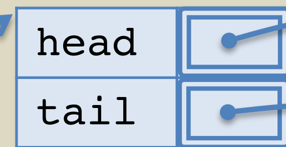
Stack

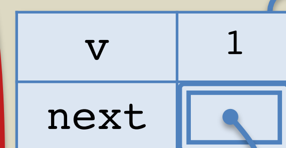


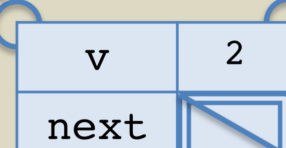


Heap

```
fun (q:'a queue) ->
 let rec loop (no:…) (len:int)=
      …
    in
    loop q.head 0
```







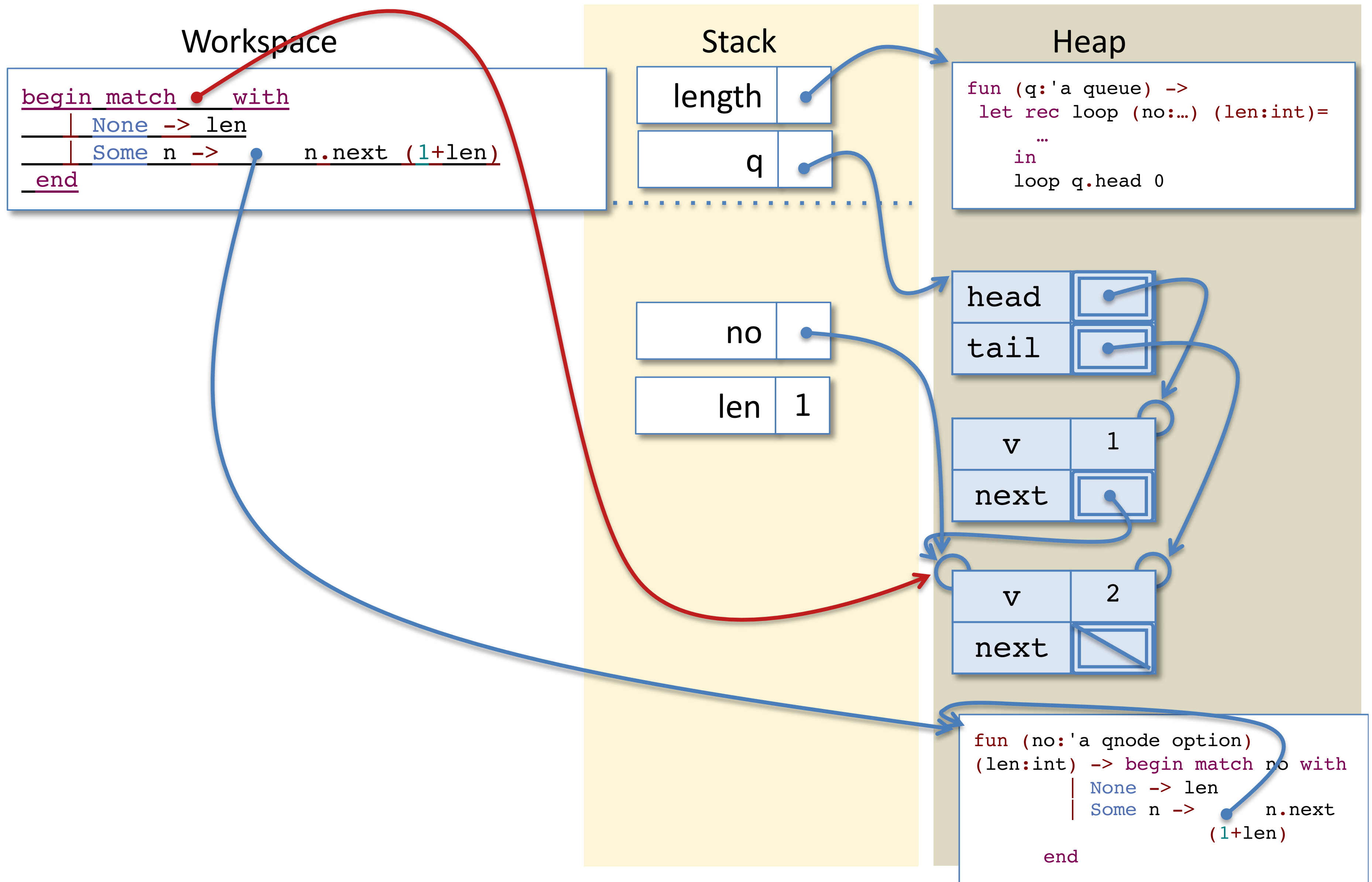

```
fun (no:'a qnode option)
(len:int) -> begin match no with
           | None -> len
           | Some n ->      n.next
                           (1+len)
       end
```

Note: we *popped* the old values of no, len, and n when we did the tail call. Then we *pushed* the new values of no, and len.

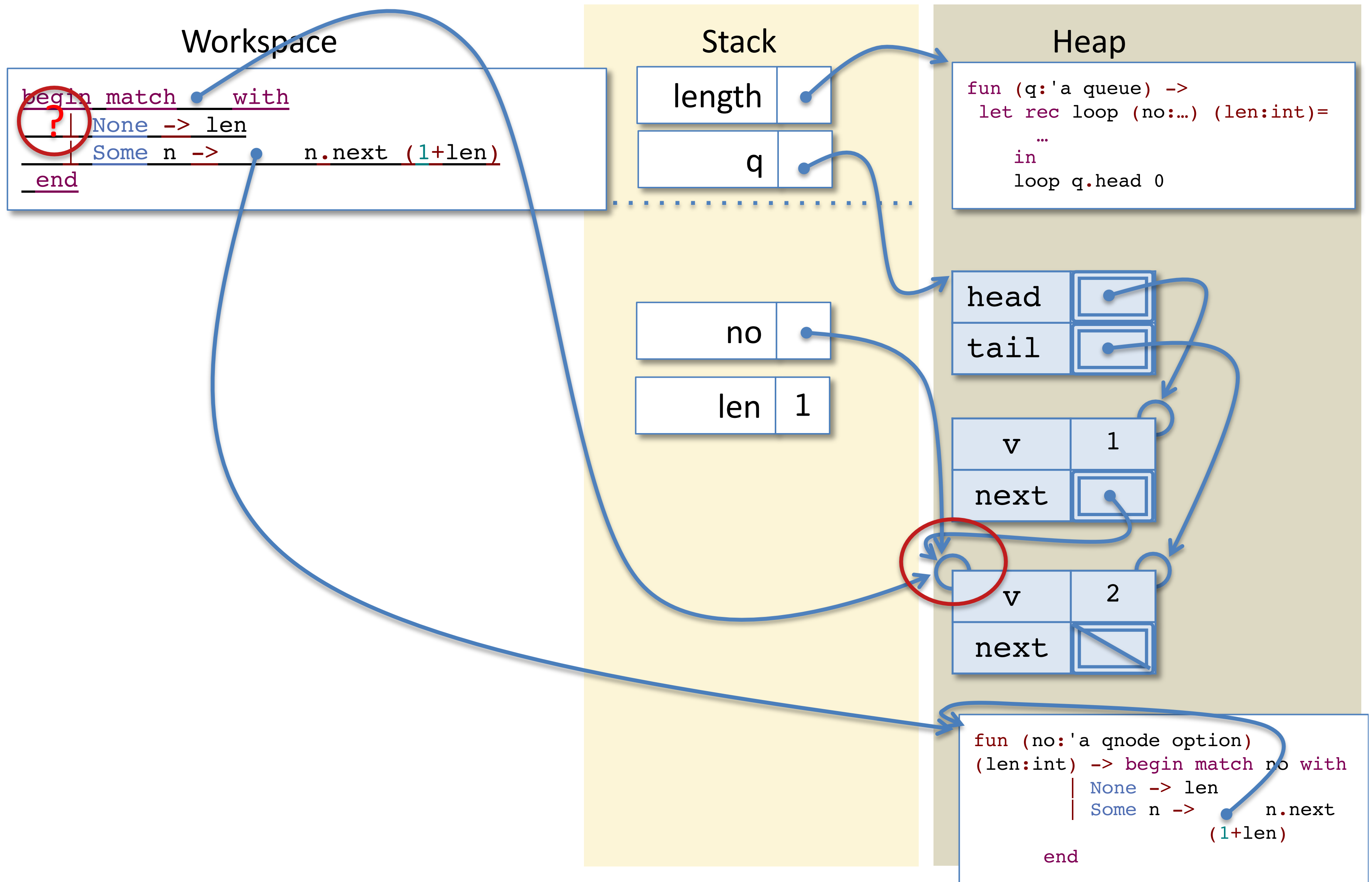
This leaves a stack with exactly the same shape as when we first called loop.

In effect, we have *updated* the stack slots for no and len.

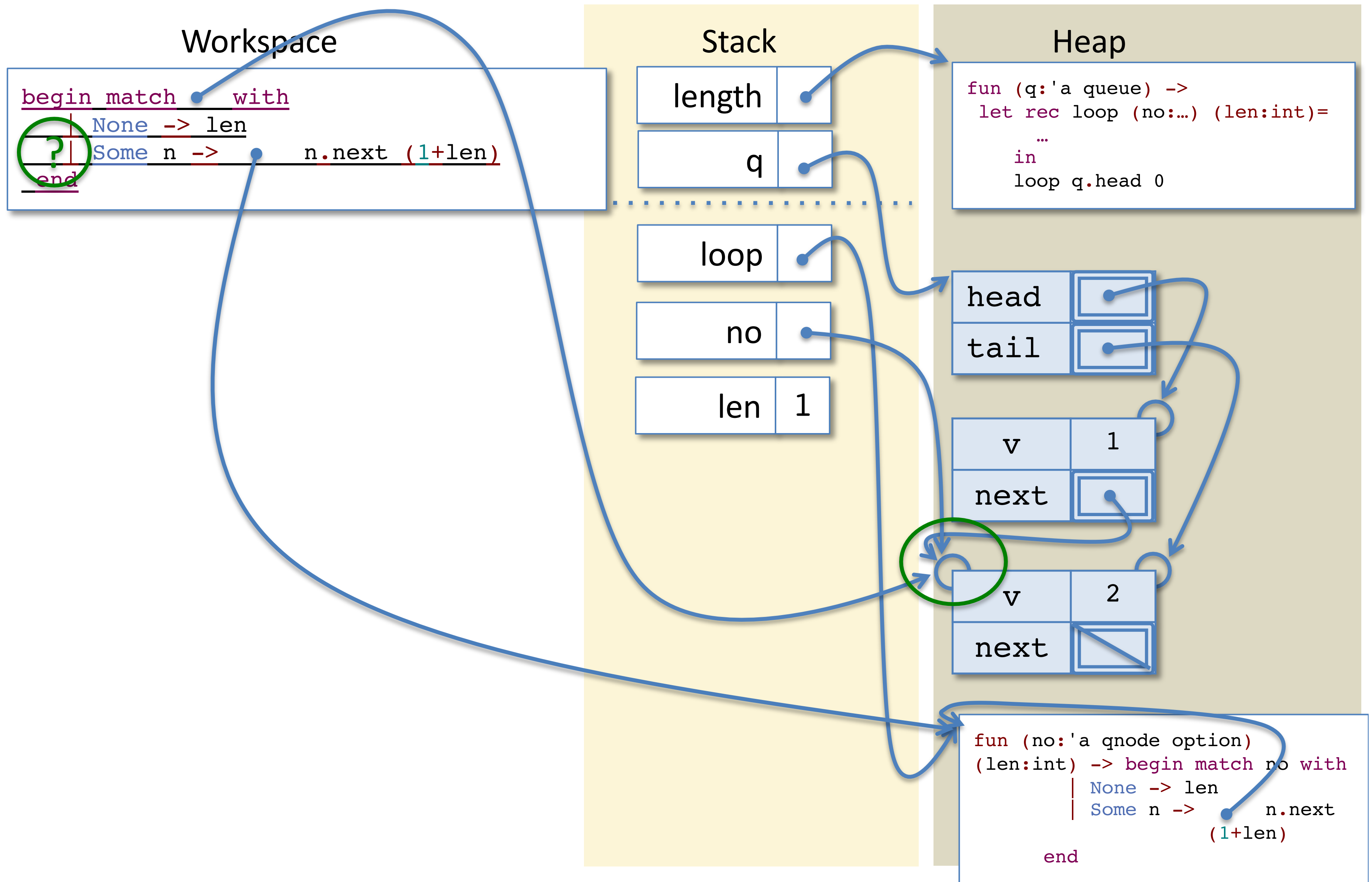
Tail Calls and Iterative length



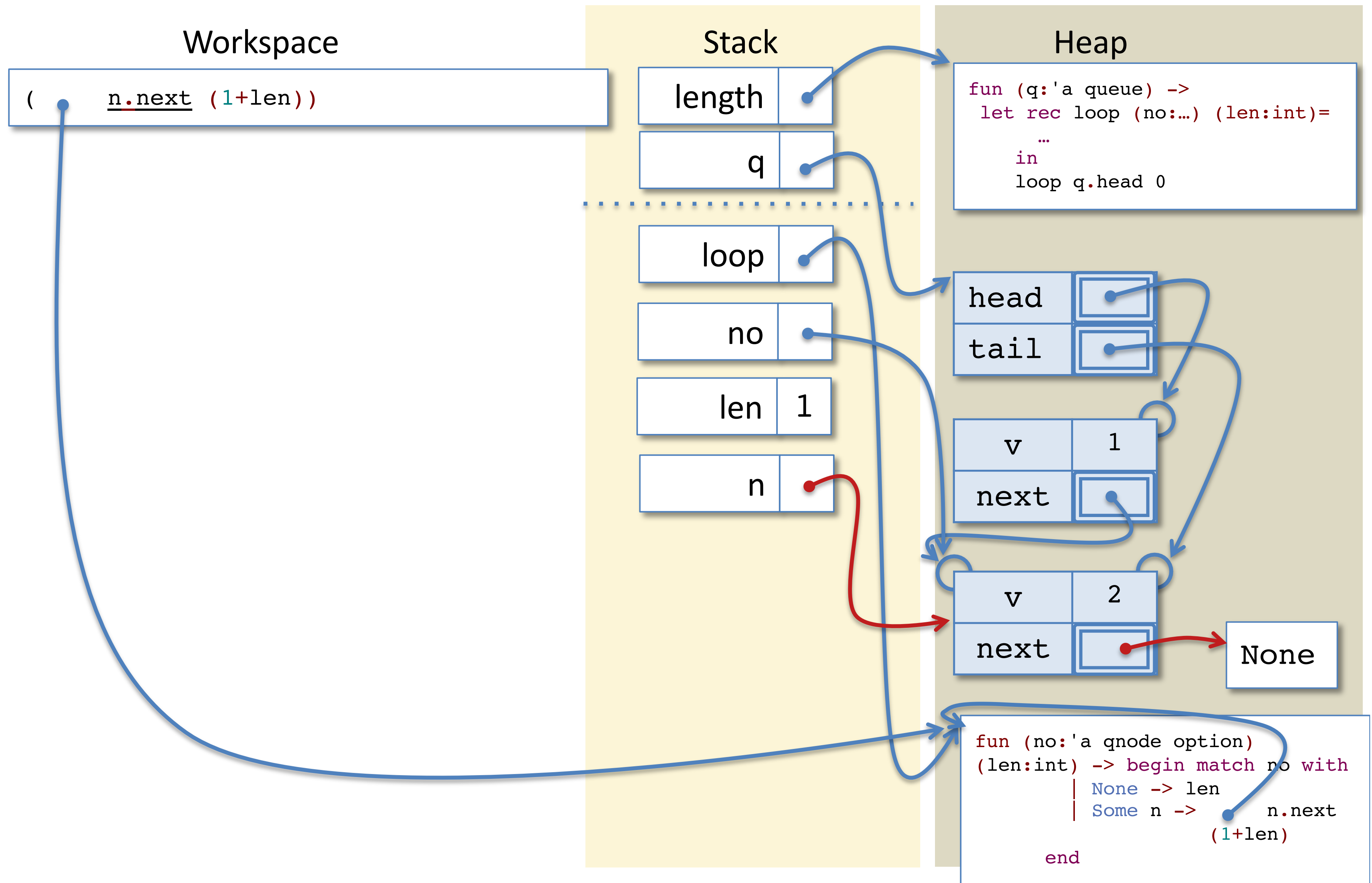
Tail Calls and Iterative length



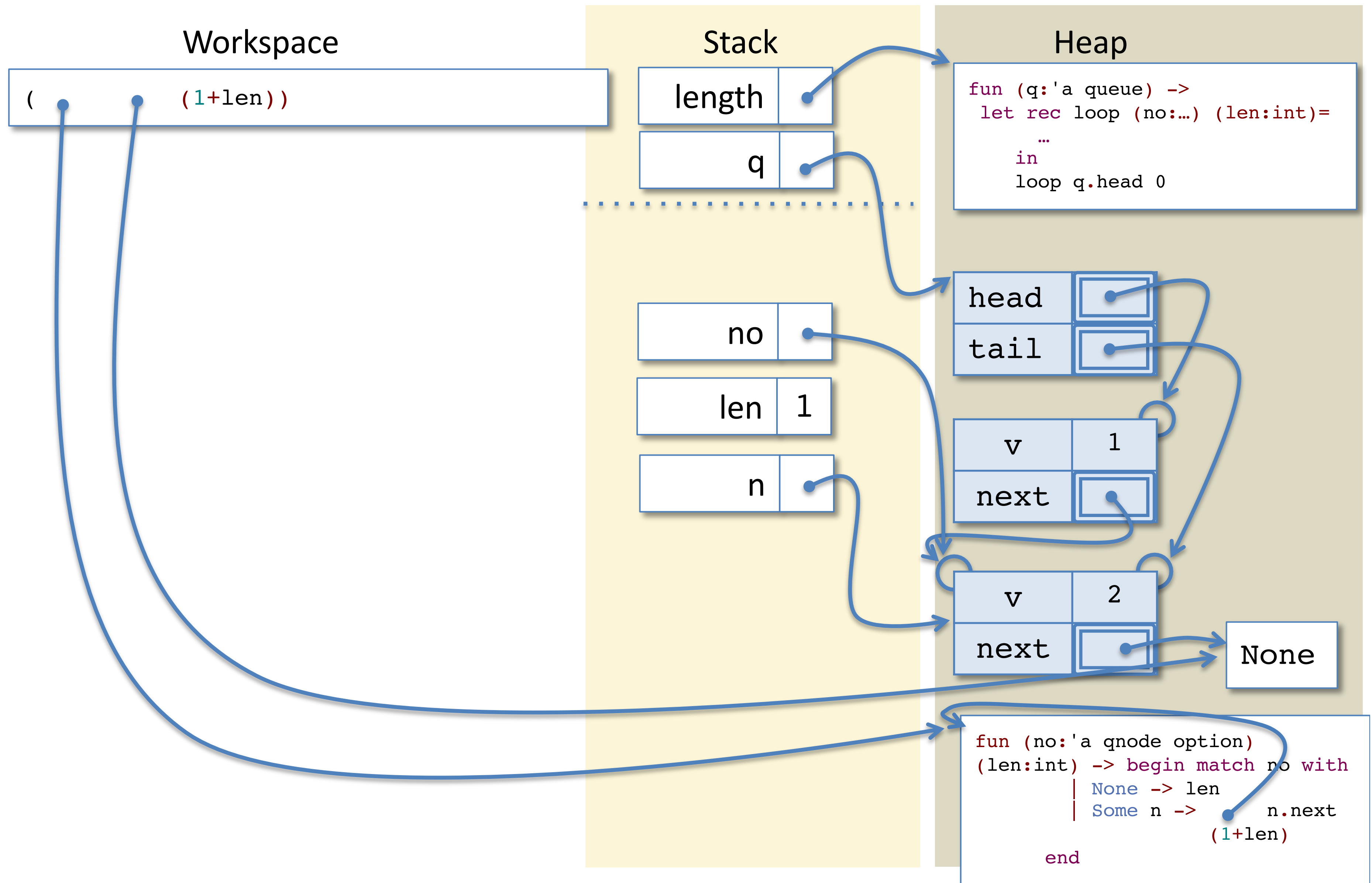
Tail Calls and Iterative length



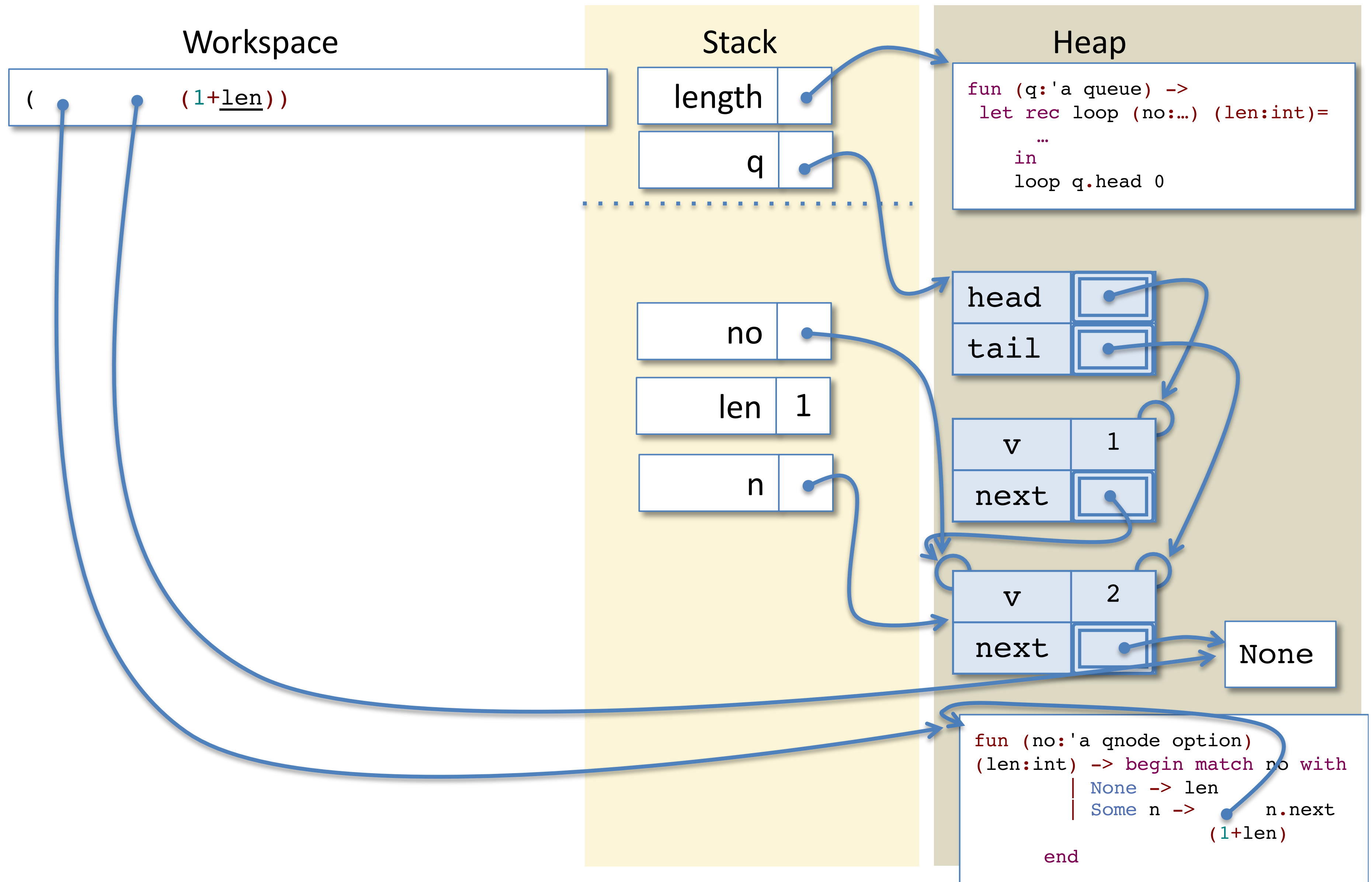
Tail Calls and Iterative length



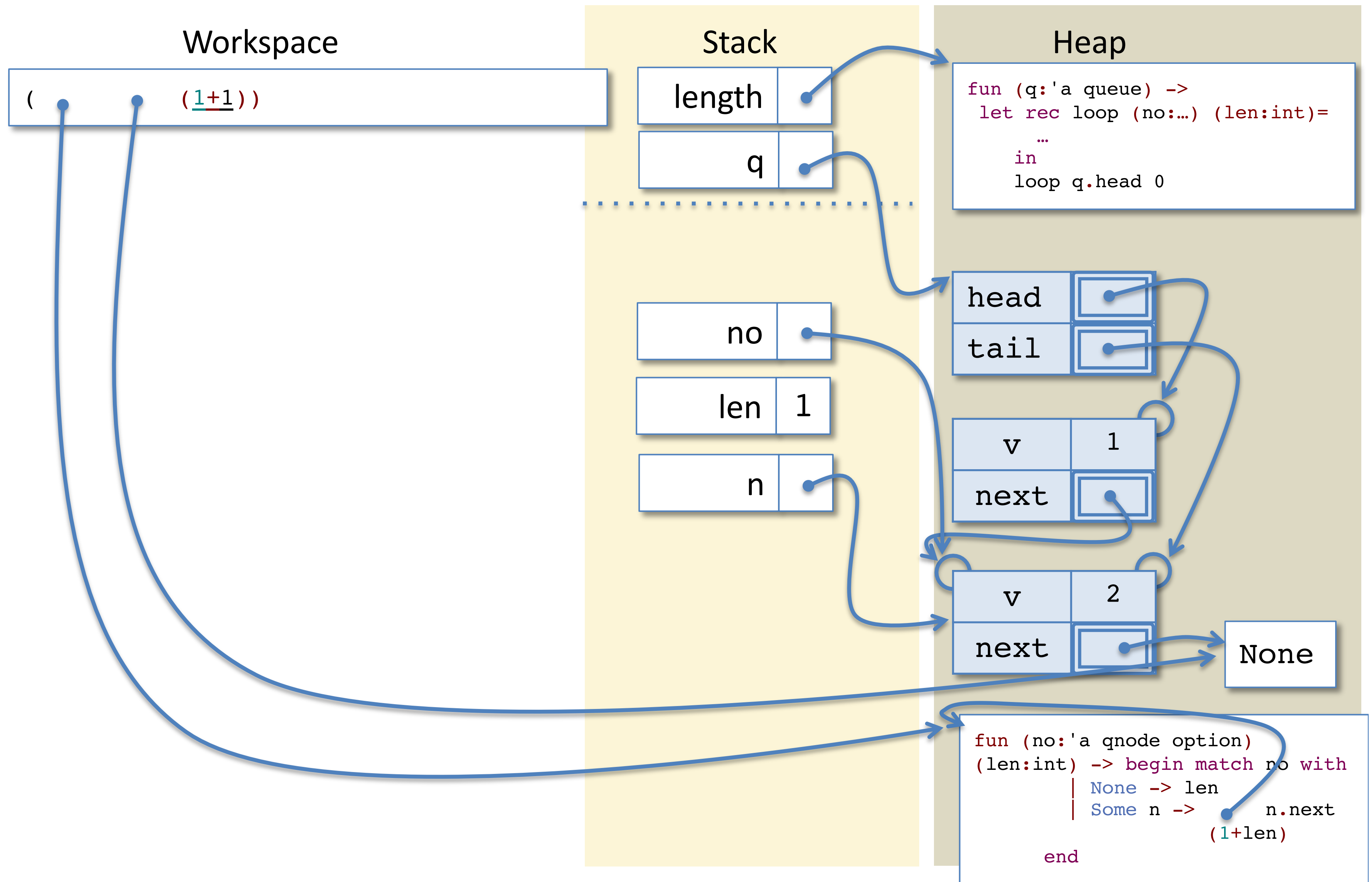
Tail Calls and Iterative length



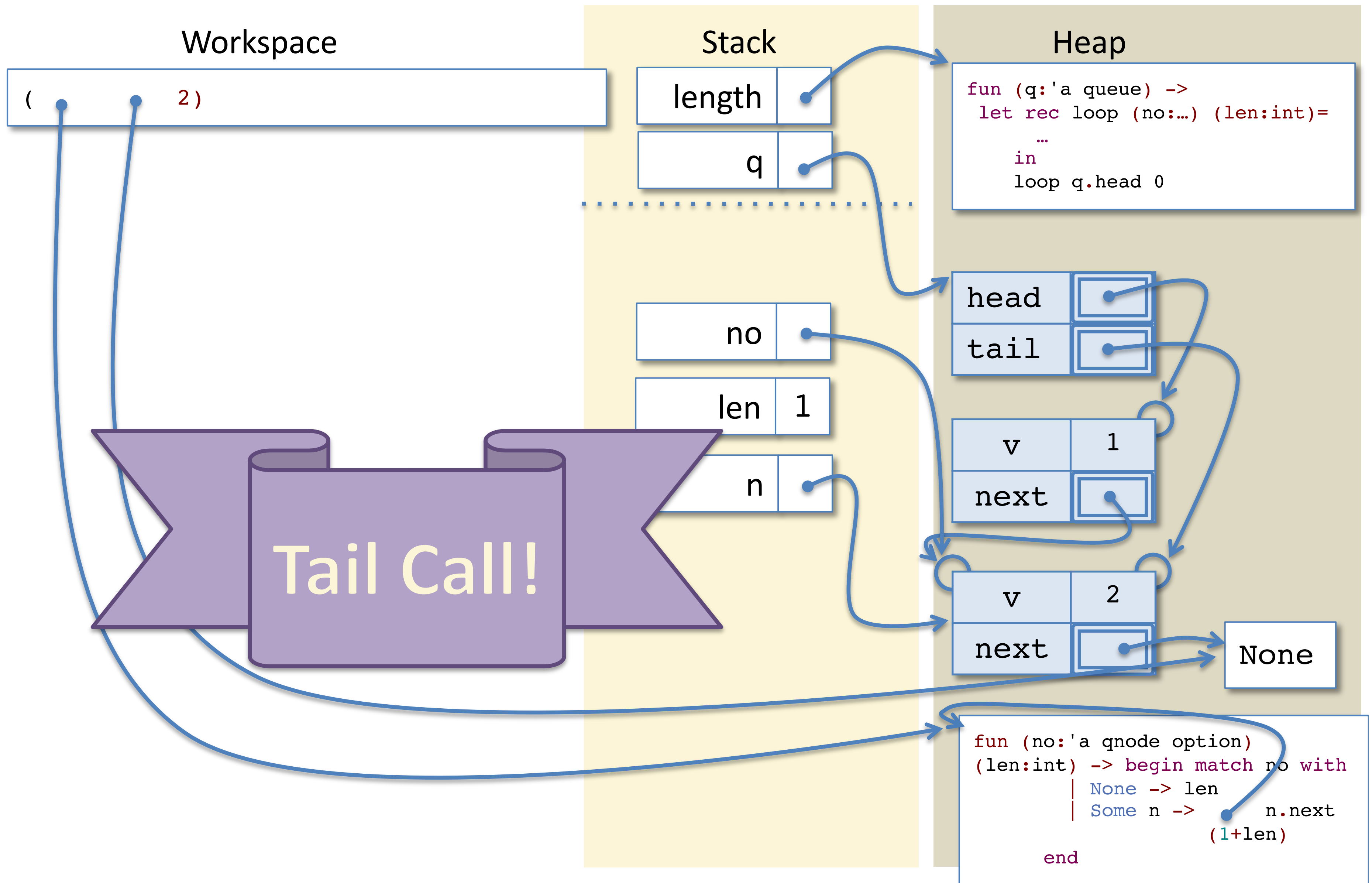
Tail Calls and Iterative length



Tail Calls and Iterative length



Tail Calls and Iterative length

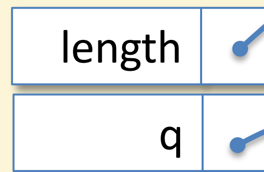


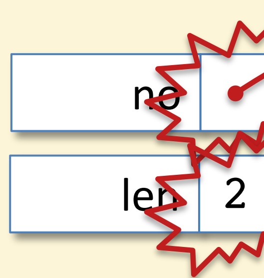
Tail Calls and Iterative length

Workspace

```
begin match no with
    | None -> len
    | Some n ->        n.next (1+len)
 end
```

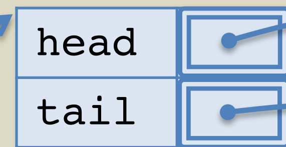
Stack

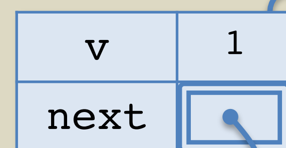


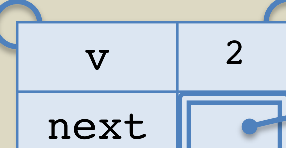


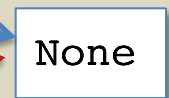
Heap

```
fun (q:'a queue) ->
 let rec loop (no:…) (len:int)=
     …
   in
   loop q.head 0
```







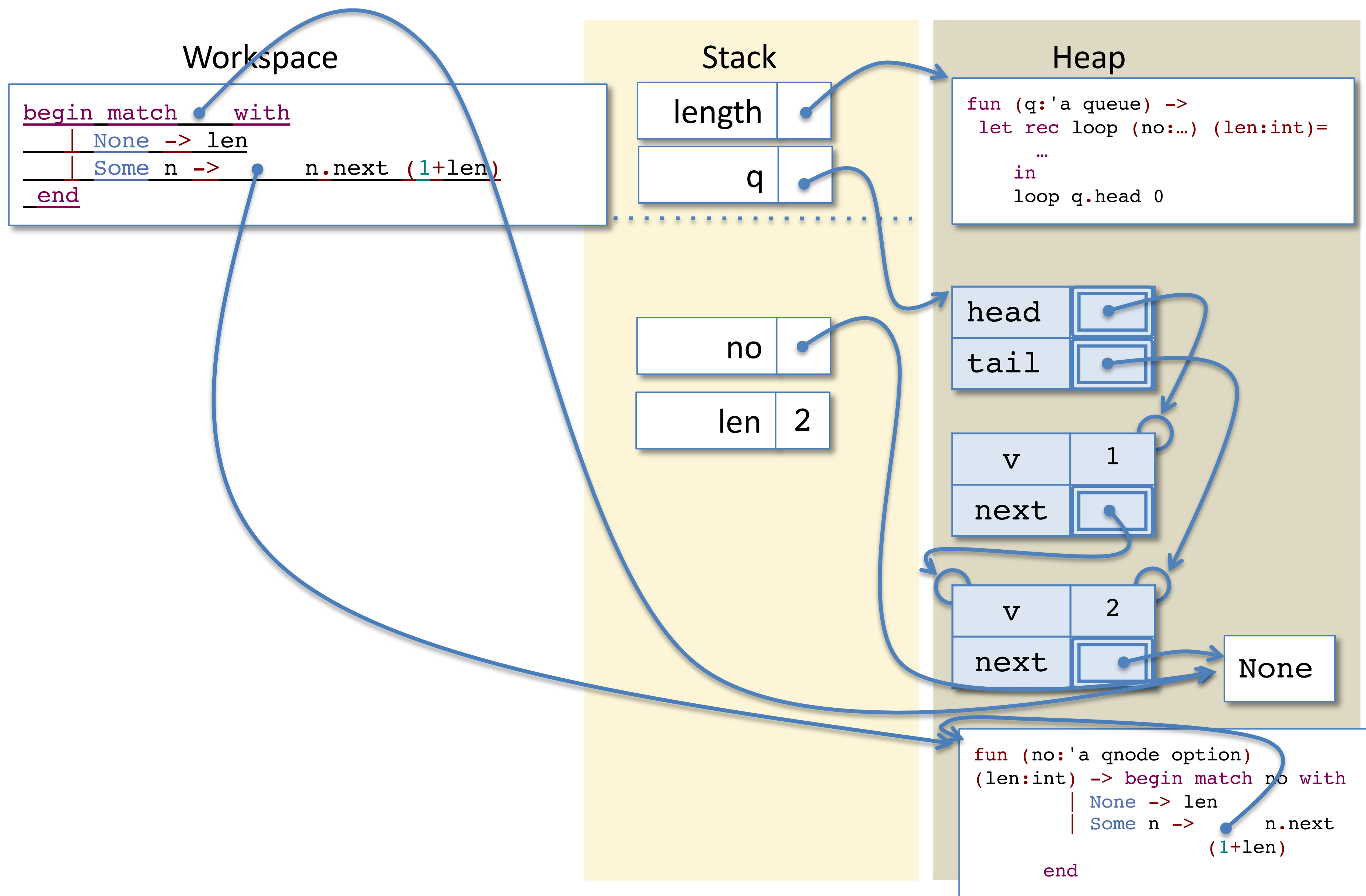



```
fun (no:'a qnode option)
(len:int) -> begin match no with
           | None -> len
           | Some n ->      n.next
                        (1+len)
       end
```

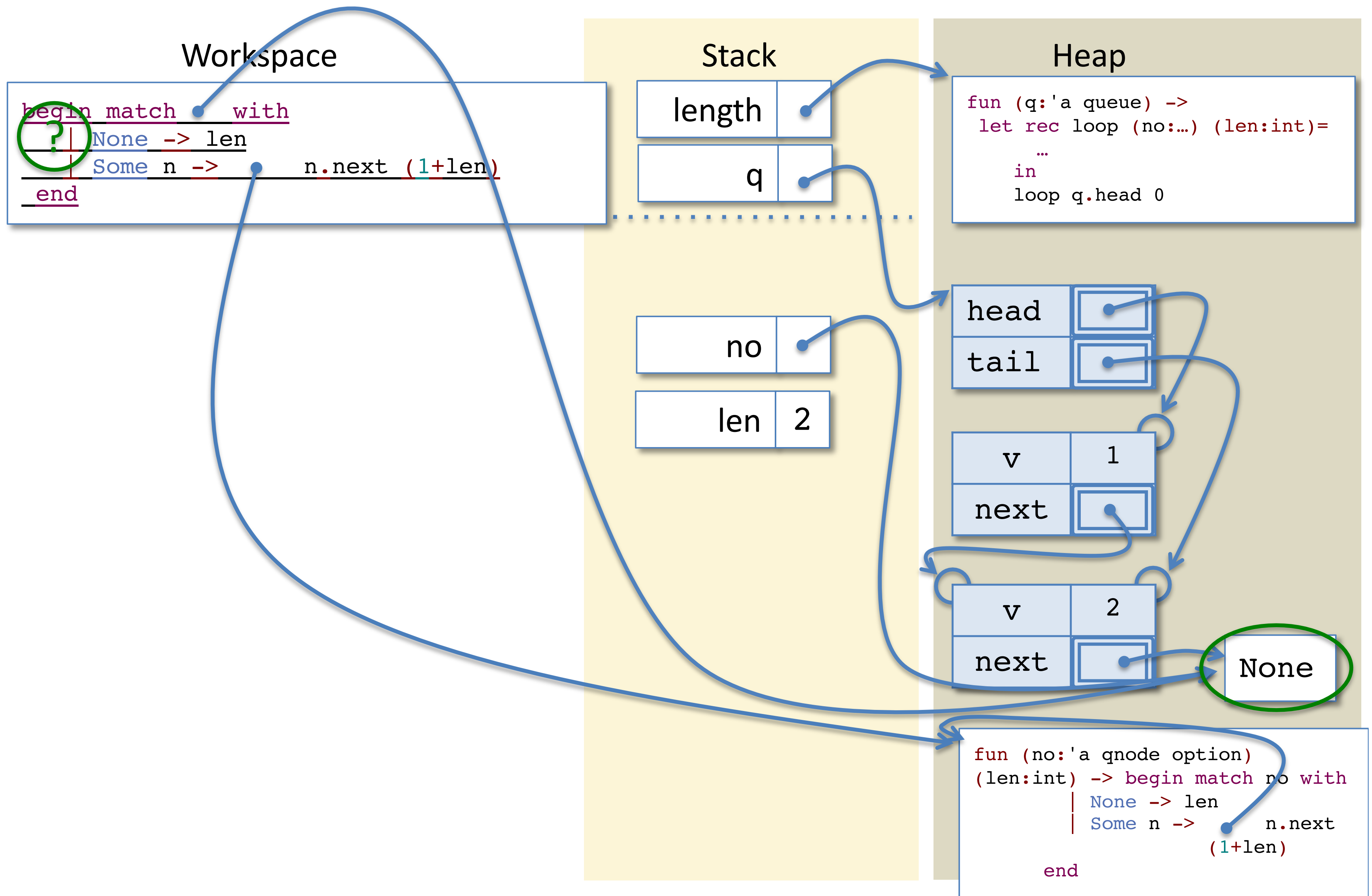
Note: Again, the tail call leaves the stack as before, but effectively updates the values of no and len.

We can think of this as an in-place update of the stack, even though technically these bindings are not mutable!

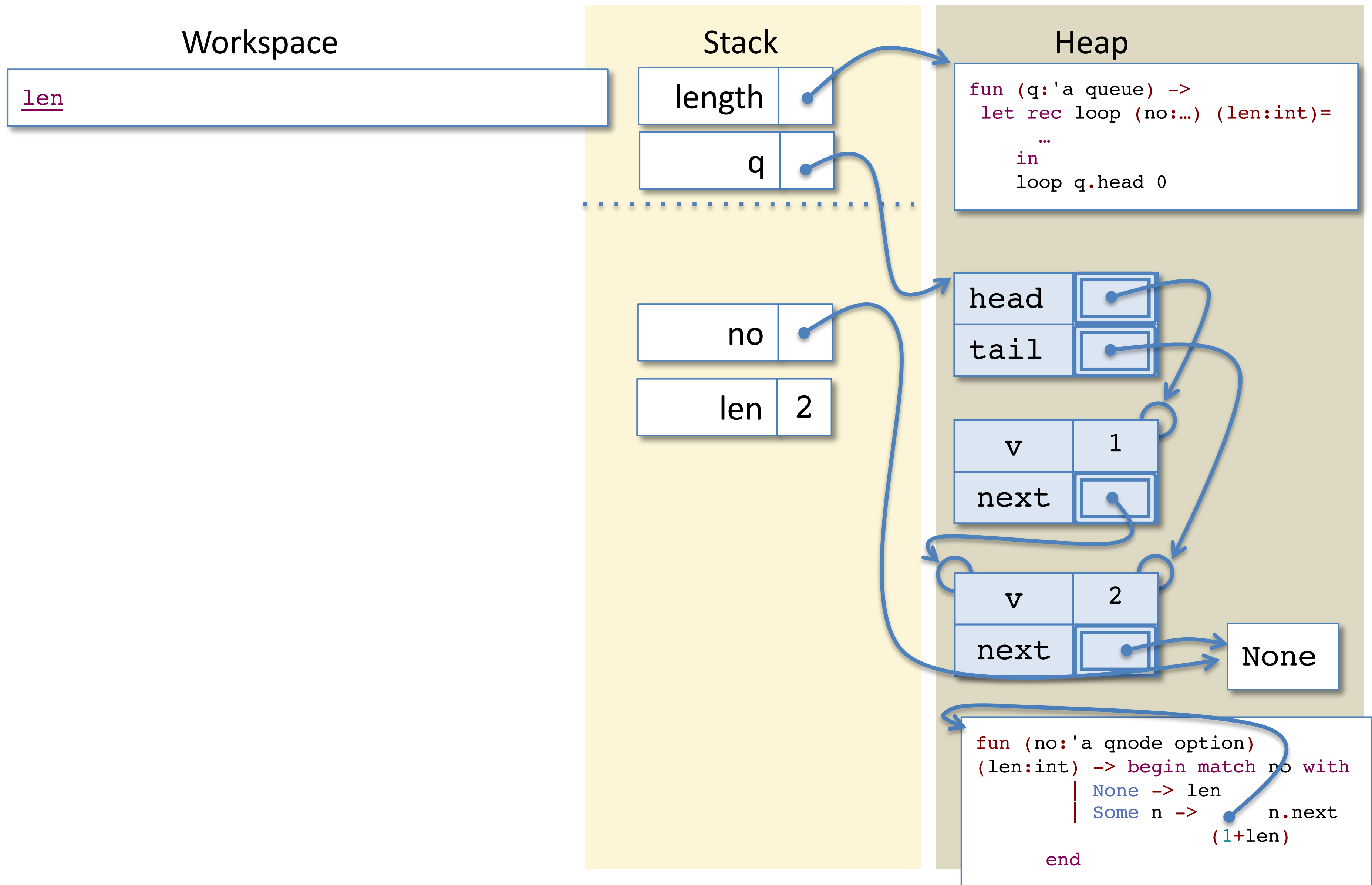
Tail Calls and Iterative length



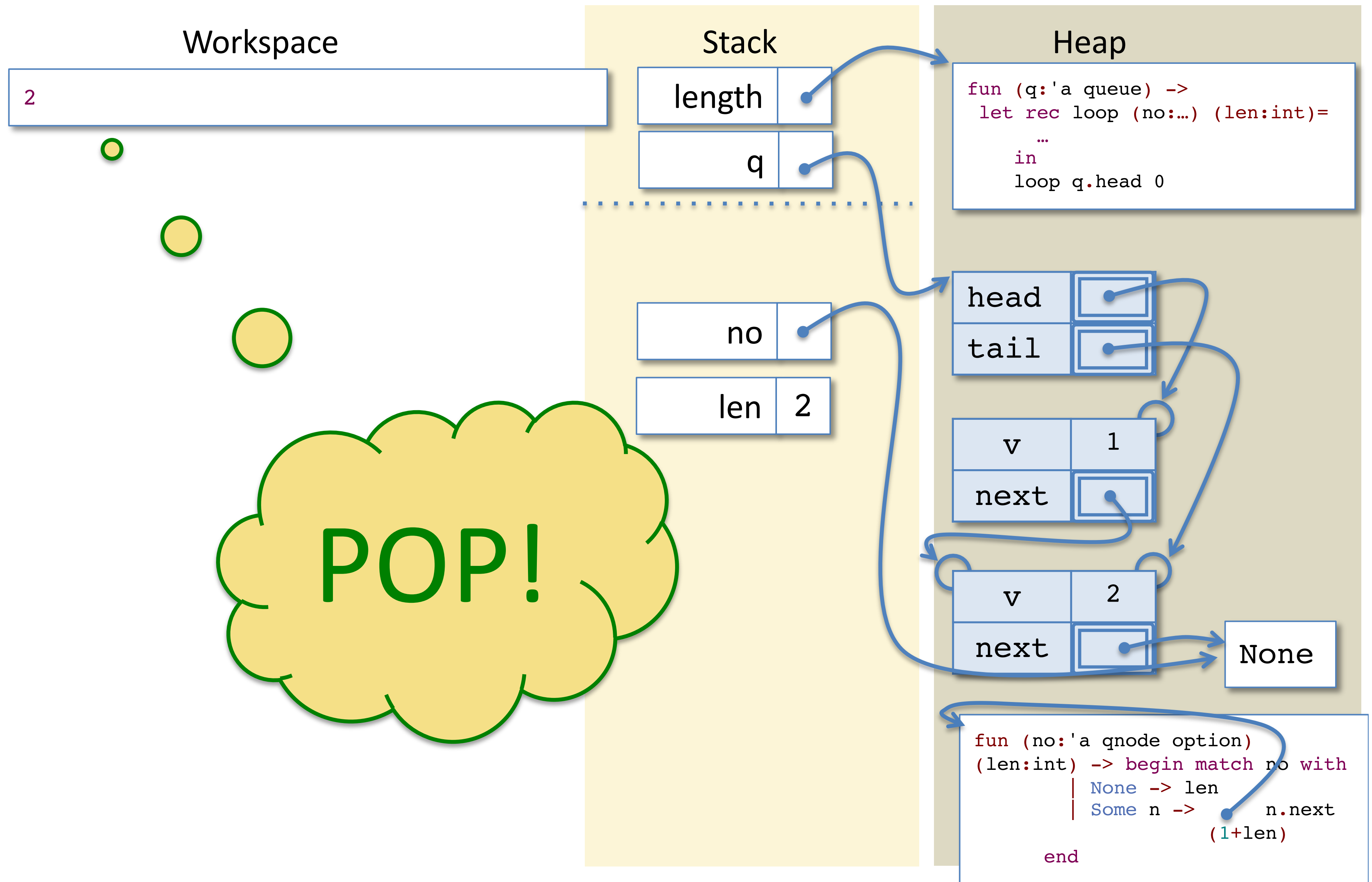
Tail Calls and Iterative length



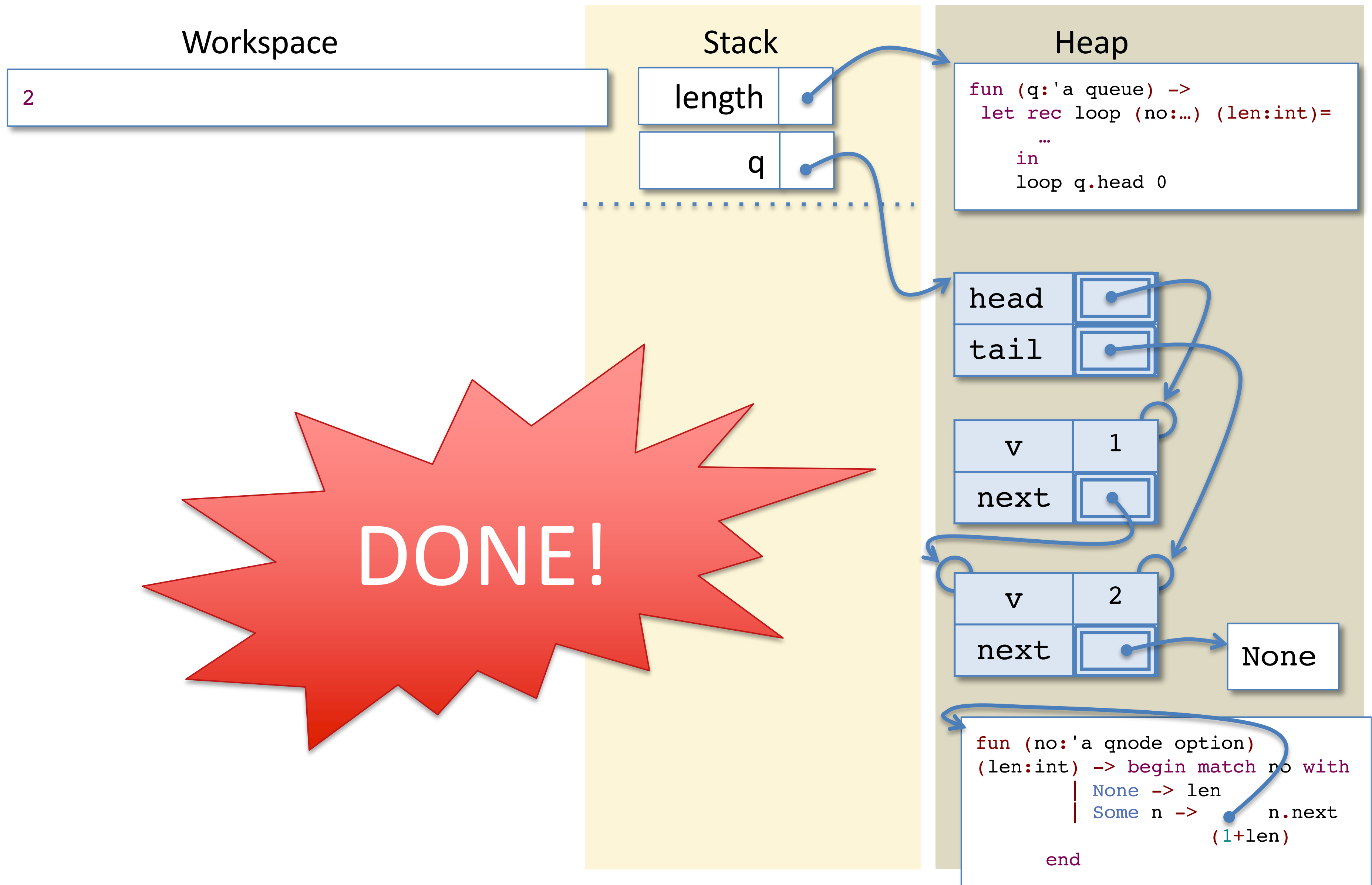
Tail Calls and Iterative length



Tail Calls and Iterative length



Tail Calls and Iterative length



Crucial Observations

- Tail call optimization lets the stack take only a *fixed amount of space*.

- The recursive call to `loop` effectively updates 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
```

The value 2 is returned

The value 0 is returned

StackOverflow

Your program hangs

Start the presentation to see live content. Still no live content? Install the app or get help at **PollEv.com/app**

Total Results

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
```

The value 2 is returned

The value 0 is returned

StackOverflow

Your program hangs

Start the presentation to see live content. Still no live content? Install the app or get help at **PollEv.com/app**

Total Results

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 current with
      | None -> …  (* iteration complete, produce result *)

      | Some n ->  … (* do something with n,
                        create new loop state *)
                   loop n.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)

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 counter, we need to build another counter_state value (say, ctr2) and another incrementing function (incr2)
  - *No encapsulation*: Code anywhere in the rest of the program can modify count

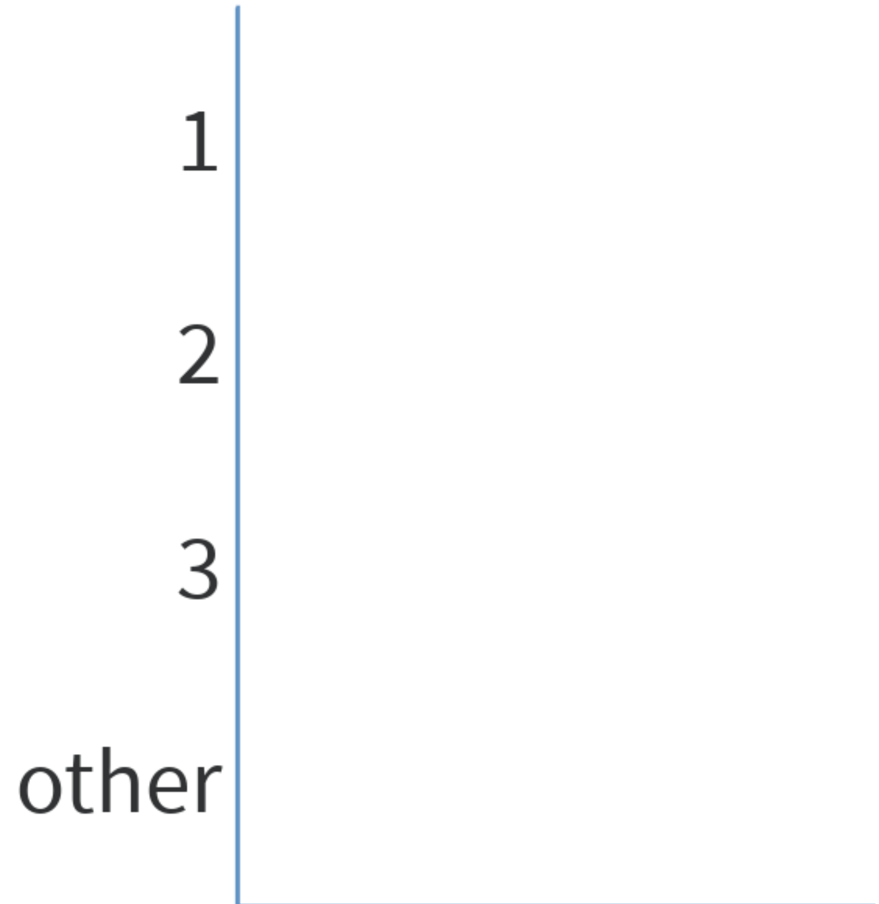
Using Hidden State

- Better: 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 ())
```



Start the presentation to see live content. Still no live content? Install the app or get help at **PollEv.com/app**

Total Results

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

Answer: 1

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

Workspace

```
let mk_incr : unit -> unit ->
int =

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

Stack

Heap

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

Running mk_incr

Workspace

```
let mk_incr : unit -> unit ->
int = .

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

Stack

Heap

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

Running mk_incr

Workspace

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

Stack

Heap

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

Running mk_incr

Workspace

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

Stack

Heap

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

Running mk_incr

Workspace

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

Stack

mk_incr

Heap

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

Running mk_incr

Workspace

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

Stack

mk_incr	

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

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 = • 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
```

count	0

Running mk_incr

Workspace

```
let ctr = • 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
```

count	0

Running mk_incr

Workspace

```
fun () ->
  ctr.count <- ctr.count + 1;
  ctr.count
```

Stack

mk_incr

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

ctr

Heap

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

count 0

Running mk_incr

Workspace

```
fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count
```

Stack

mk_incr

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

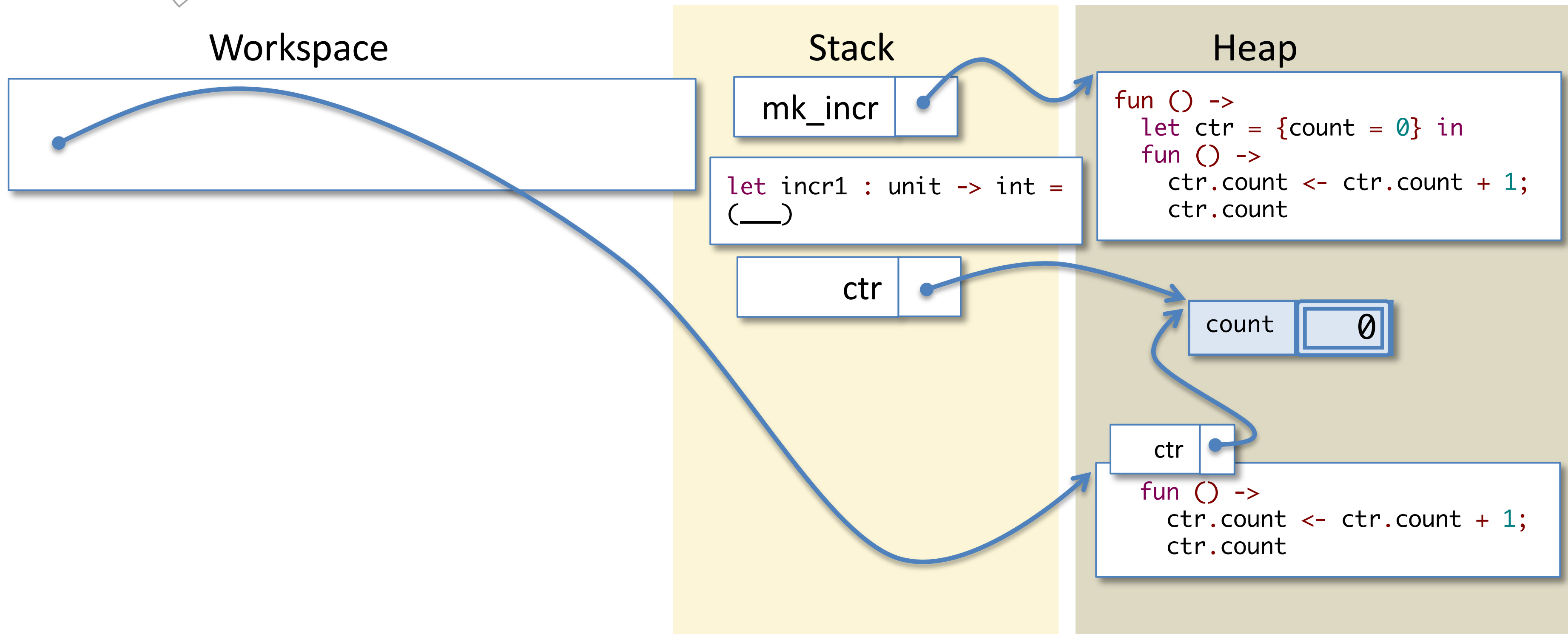
ctr

Heap

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

count	0

Local Functions



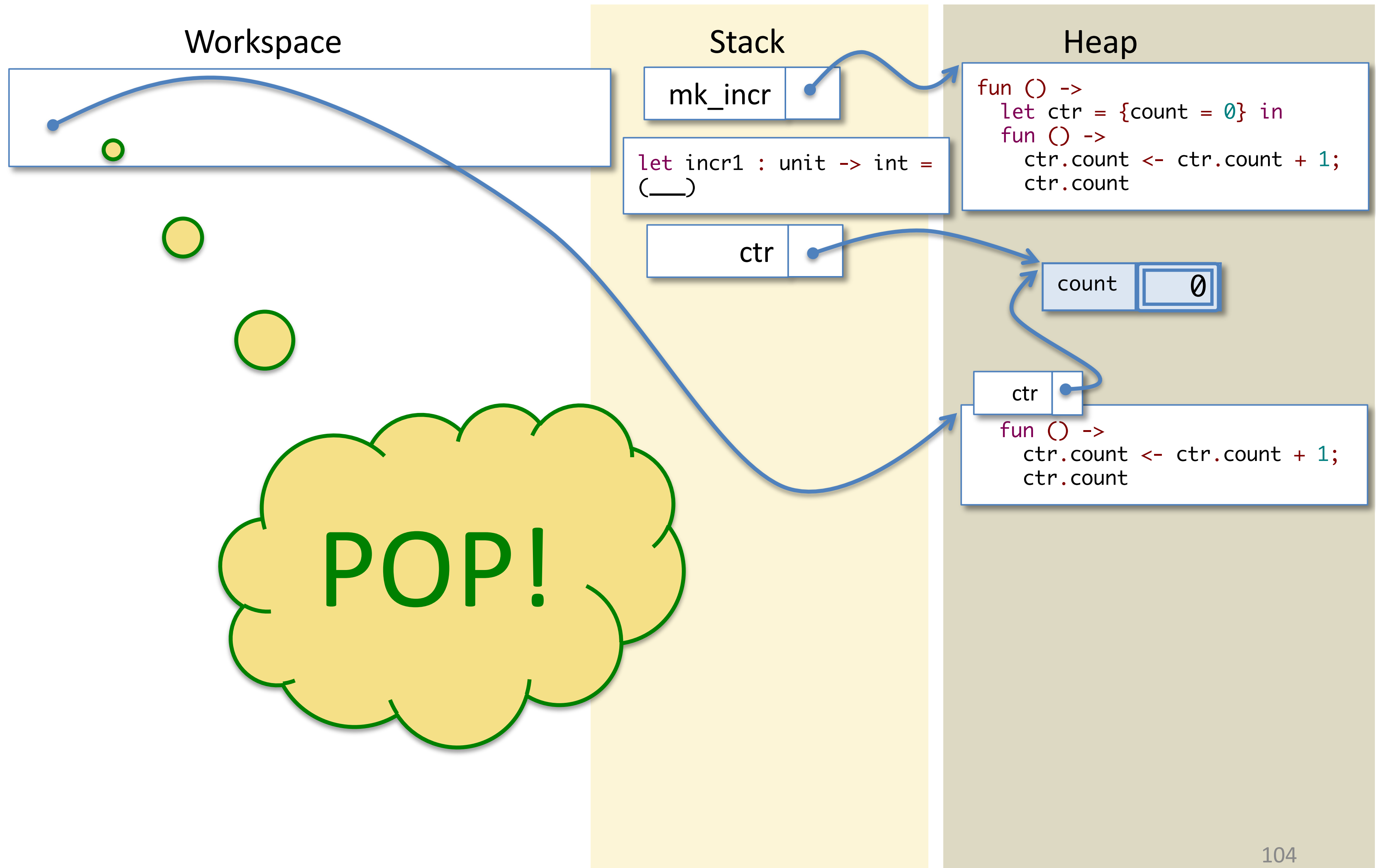
Note: We need one refinement of the ASM model that we've explained so far. Why?

The function body mentions "ctr", which is on the stack at the moment *but about to be popped off…*

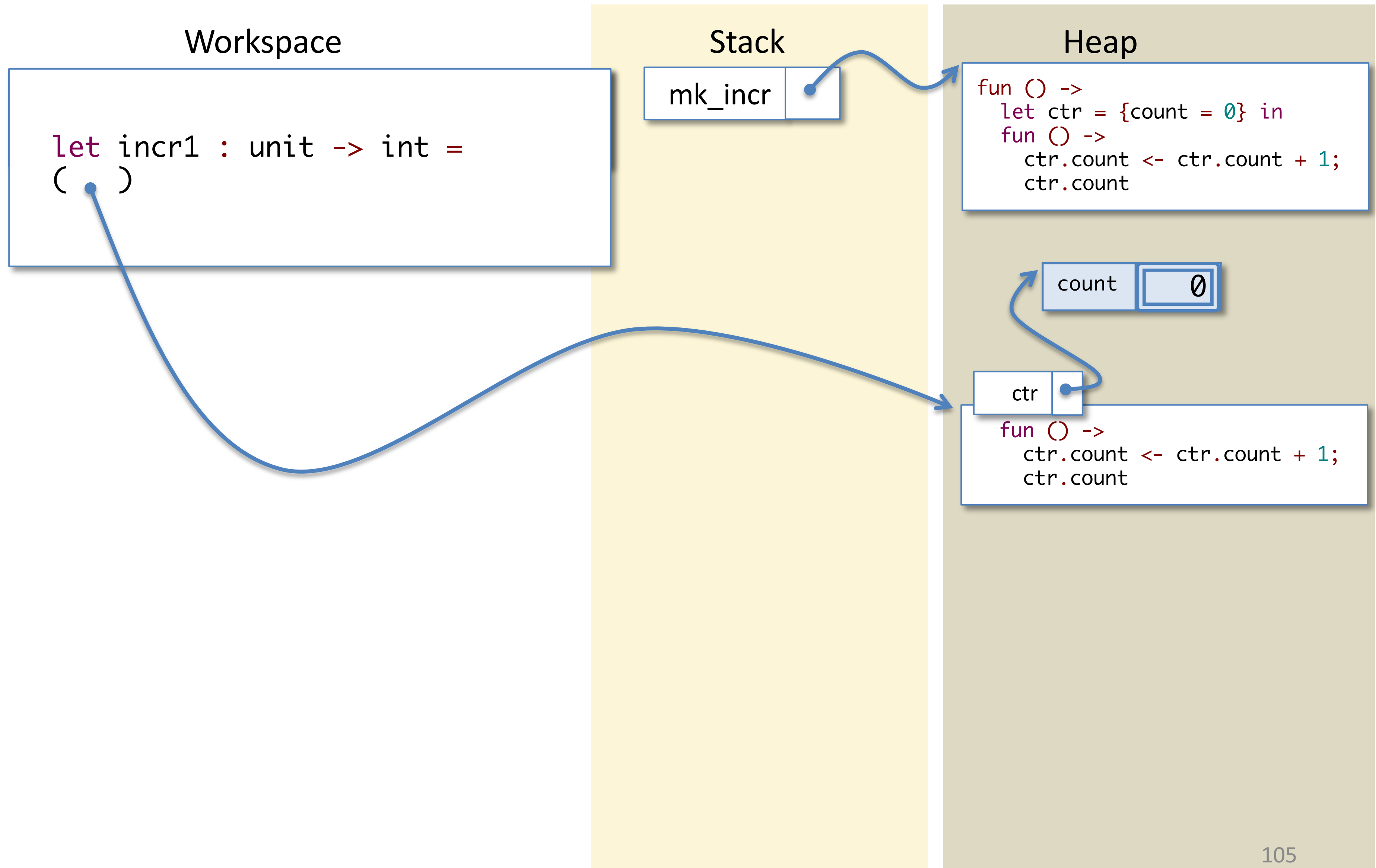
…so we save a copy of the needed stack bindings with the function itself.

This package of "function body plus needed bindings" is called a *closure*…

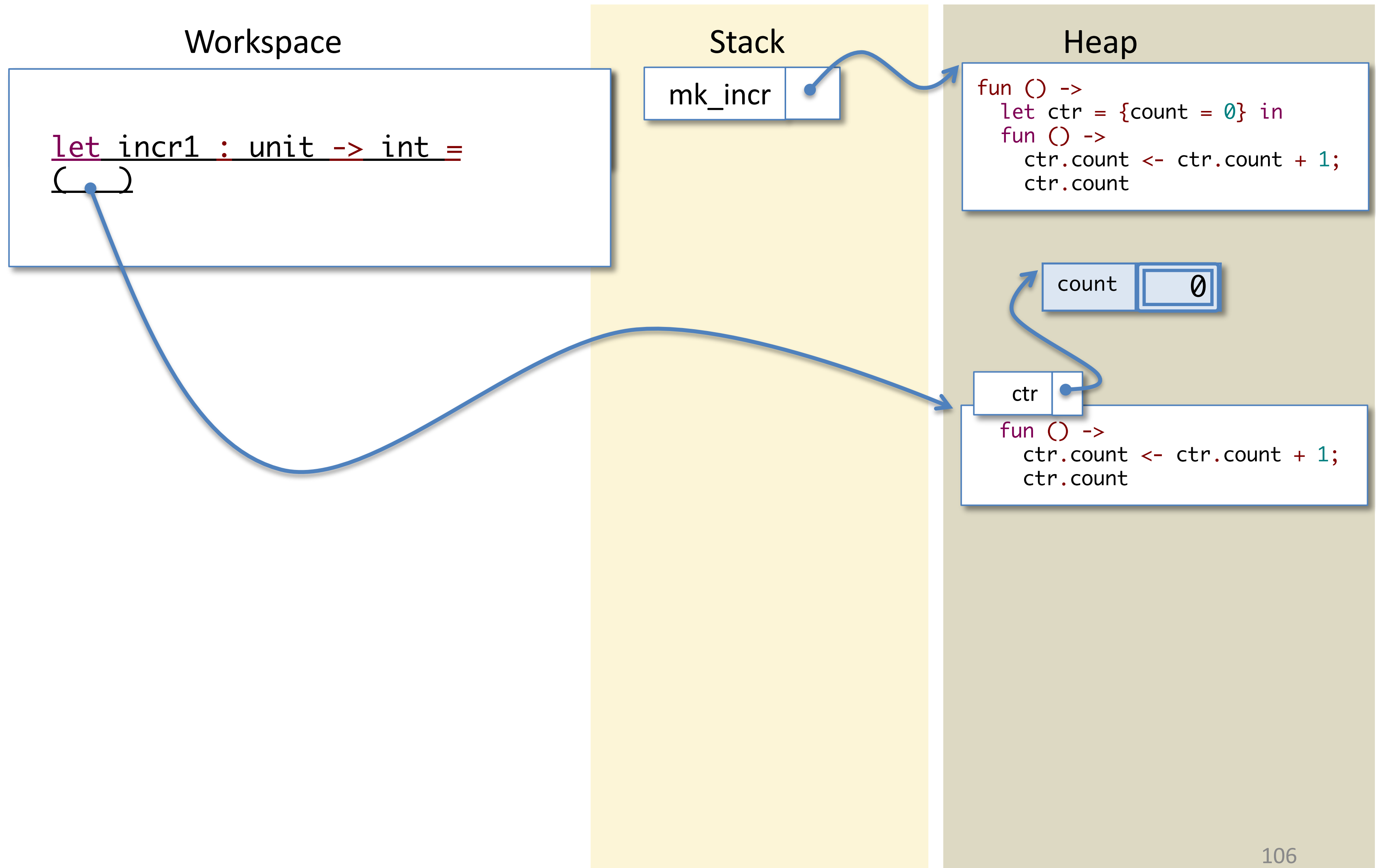
Local Functions



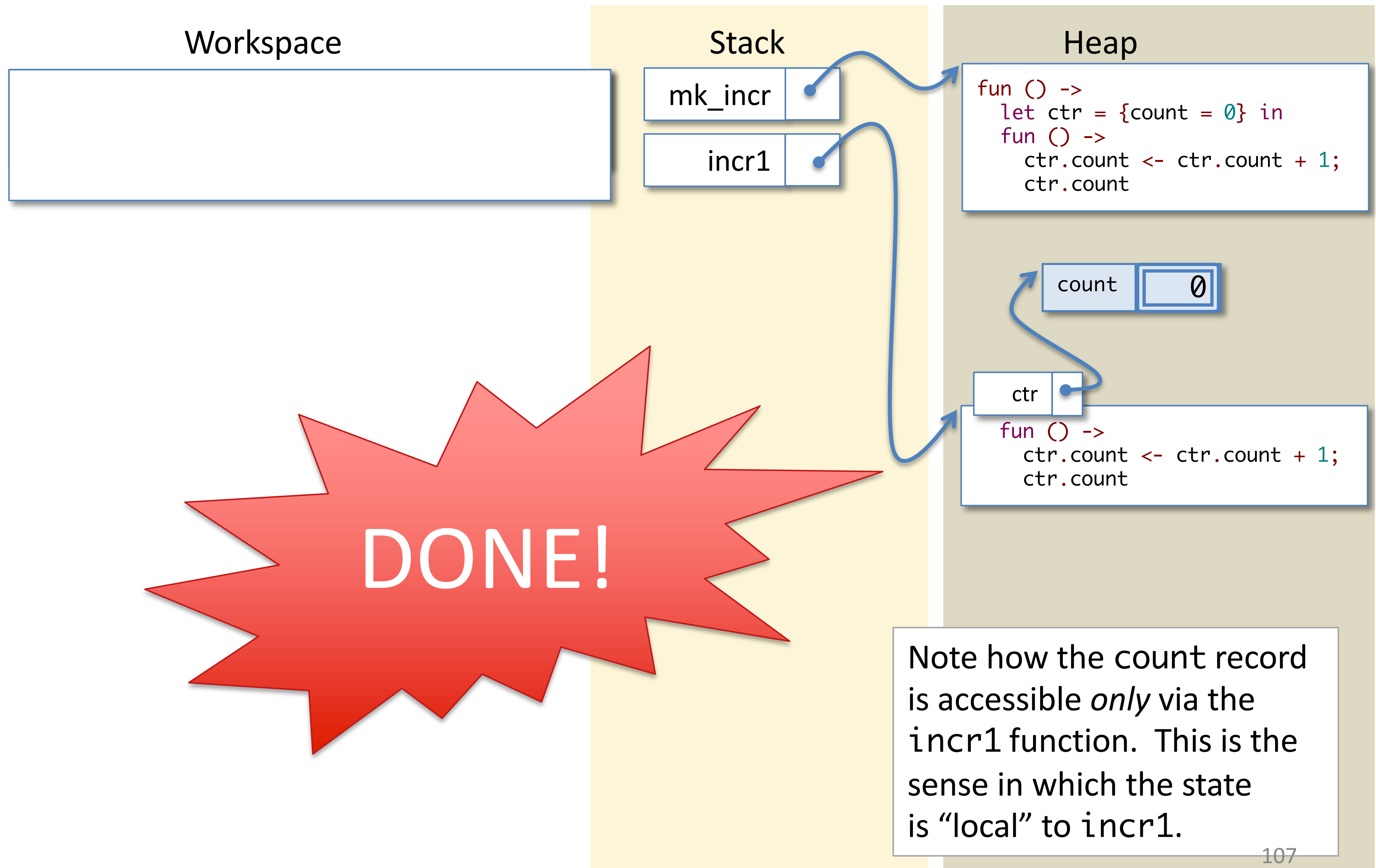
Local Functions



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`.