Programming Languages and Techniques (CIS120)

Lecture 17

February 22, 2016

Tail recursion

Chapter 16

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
```

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

Announcements

- Homework 4: Queues
  - Due: Tomorrow, February 23rd
- Homework 5: GUI Library & Paint
  - Available Wednesday
  - Due: Thursday, March 3rd
  - No class Friday, March 4th

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)

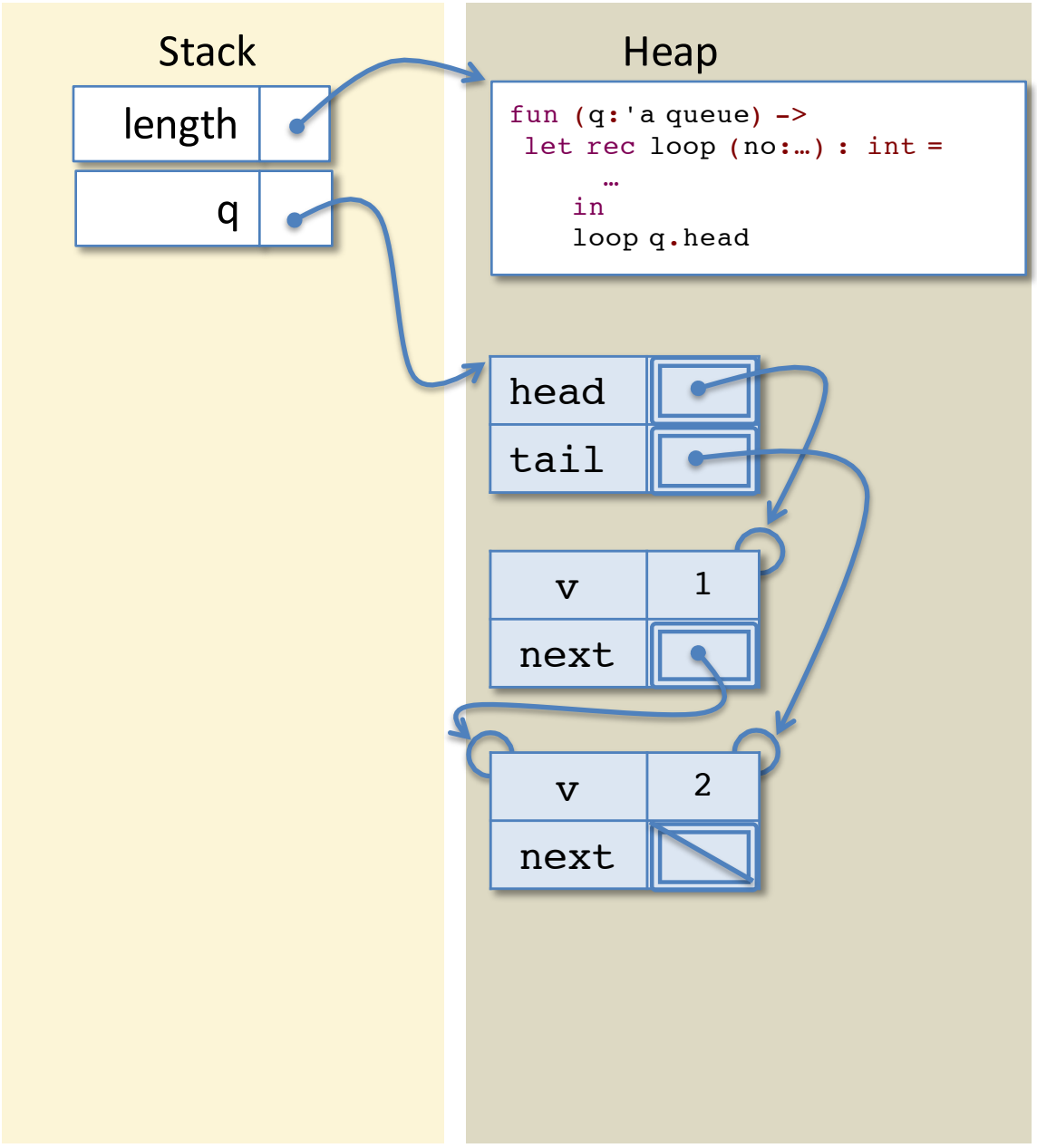
```
(* Calculate the length of the queue recursively *)
let length (q:'a queue) : int =
  let rec loop (no: 'a qnode option) : int =
    begin match no with
      | None -> 0
      | Some n -> 1 + (loop n.next)
    end
  in
  loop q.head
```

- 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

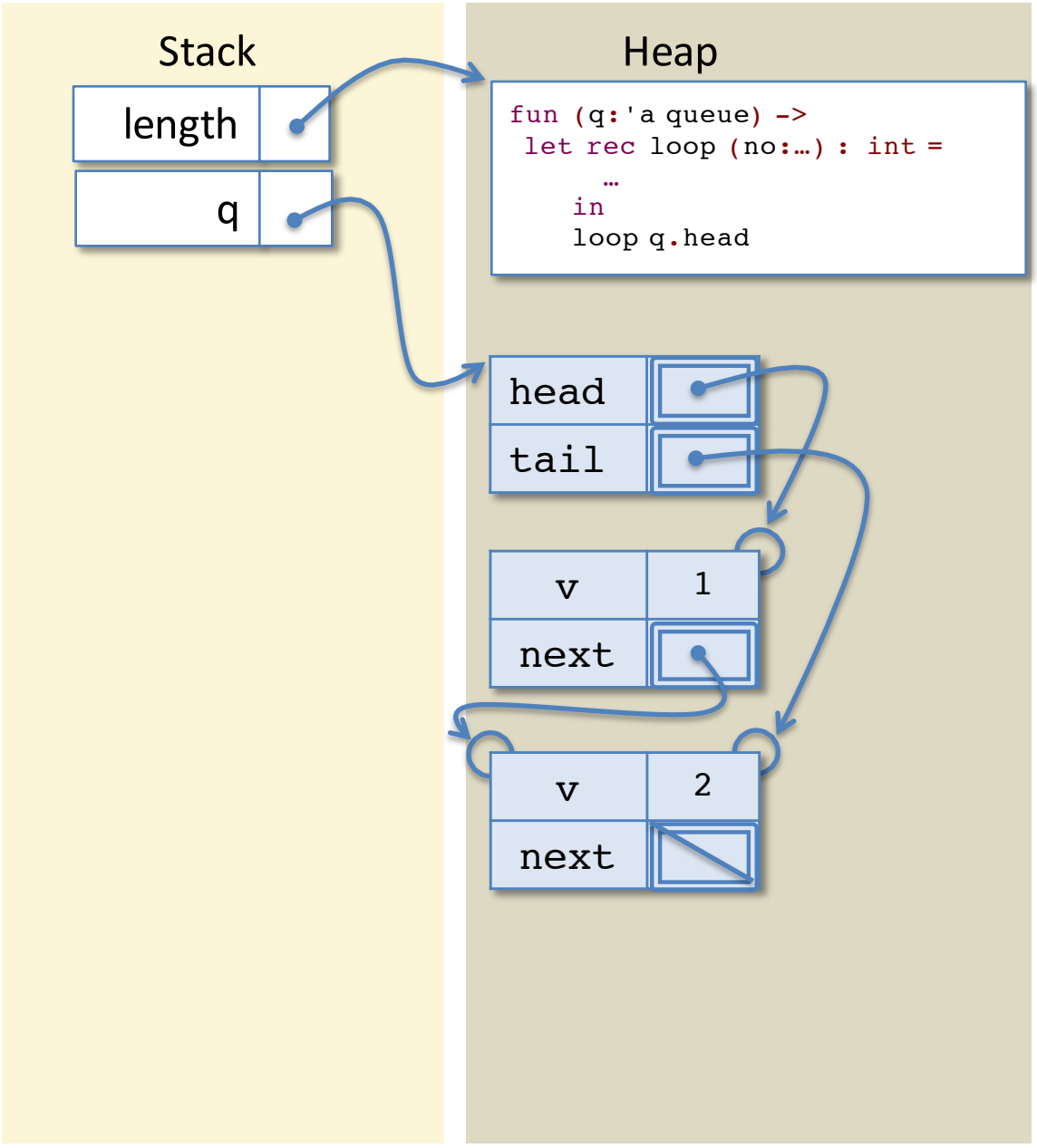
```
length q
```



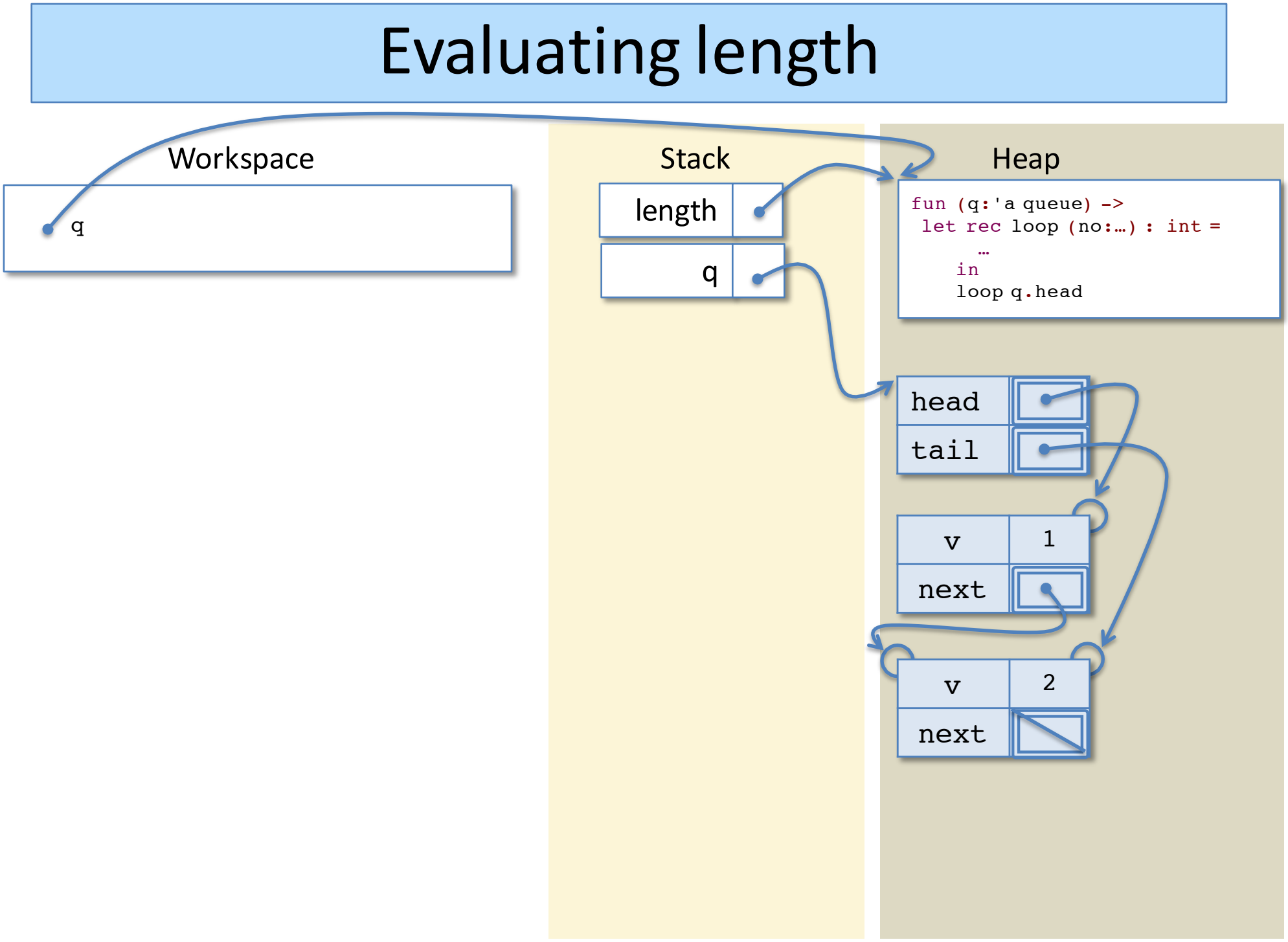
Evaluating length

Workspace

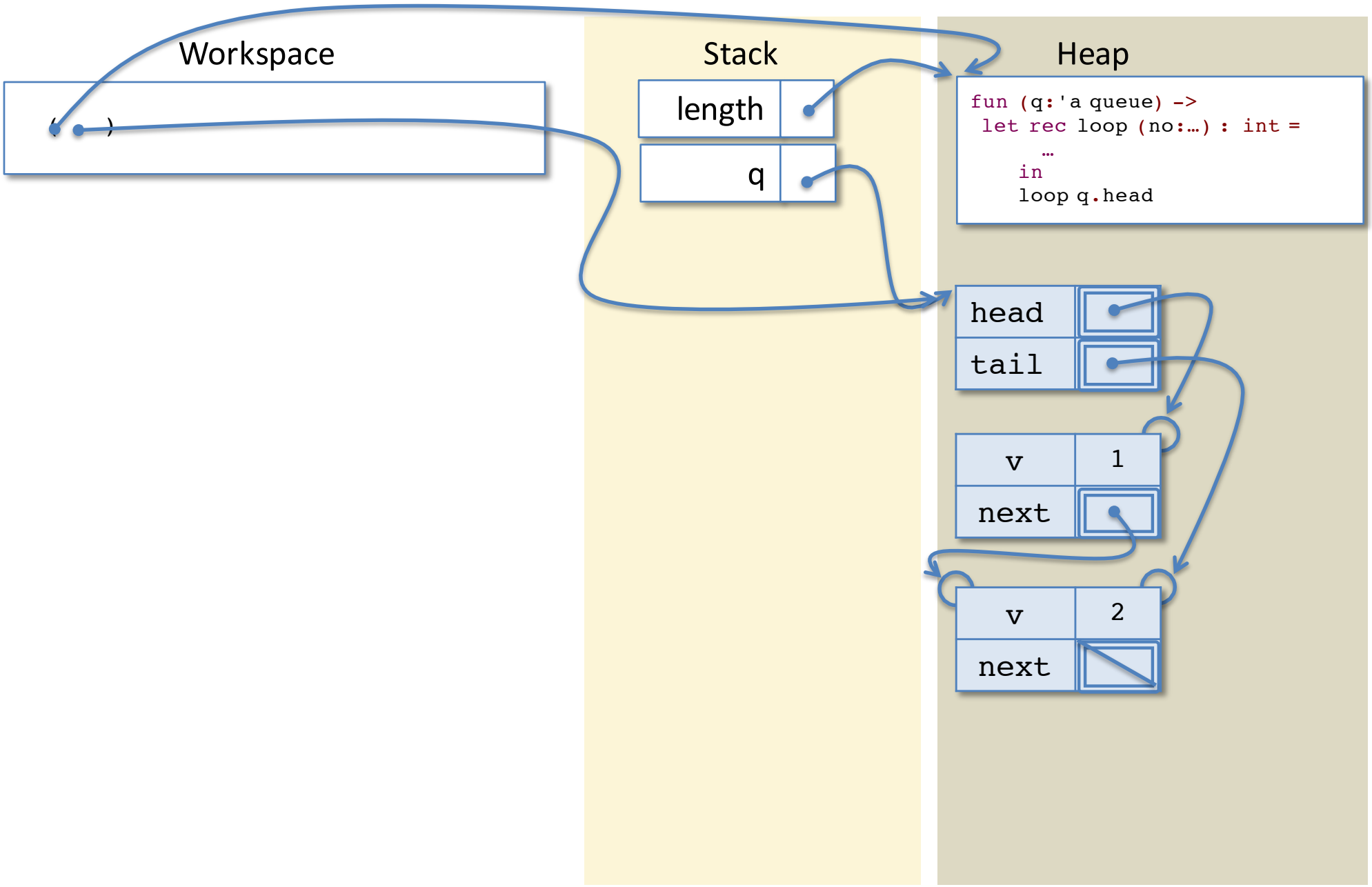
```
length q
```



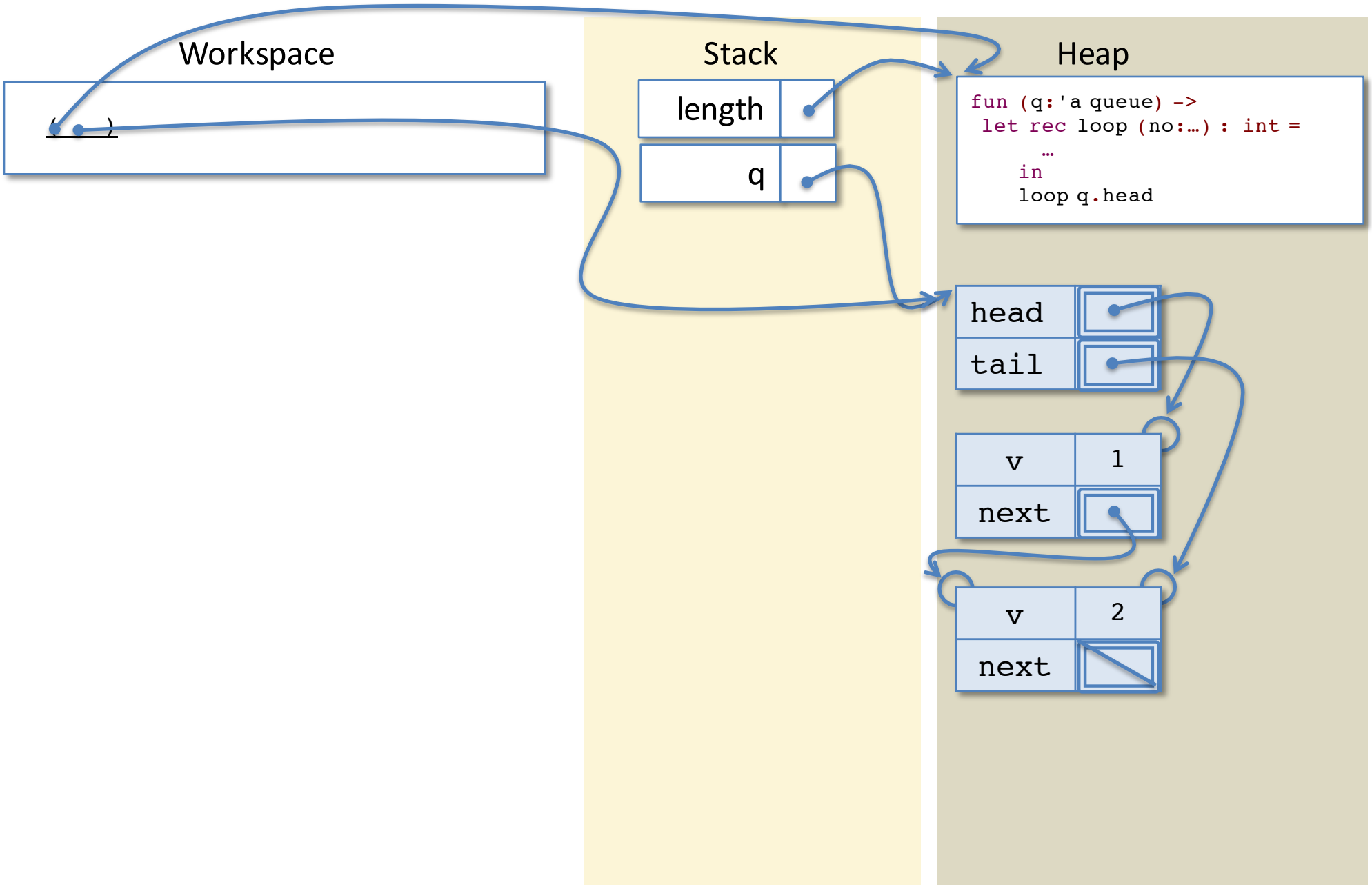
Evaluating length



Evaluating length



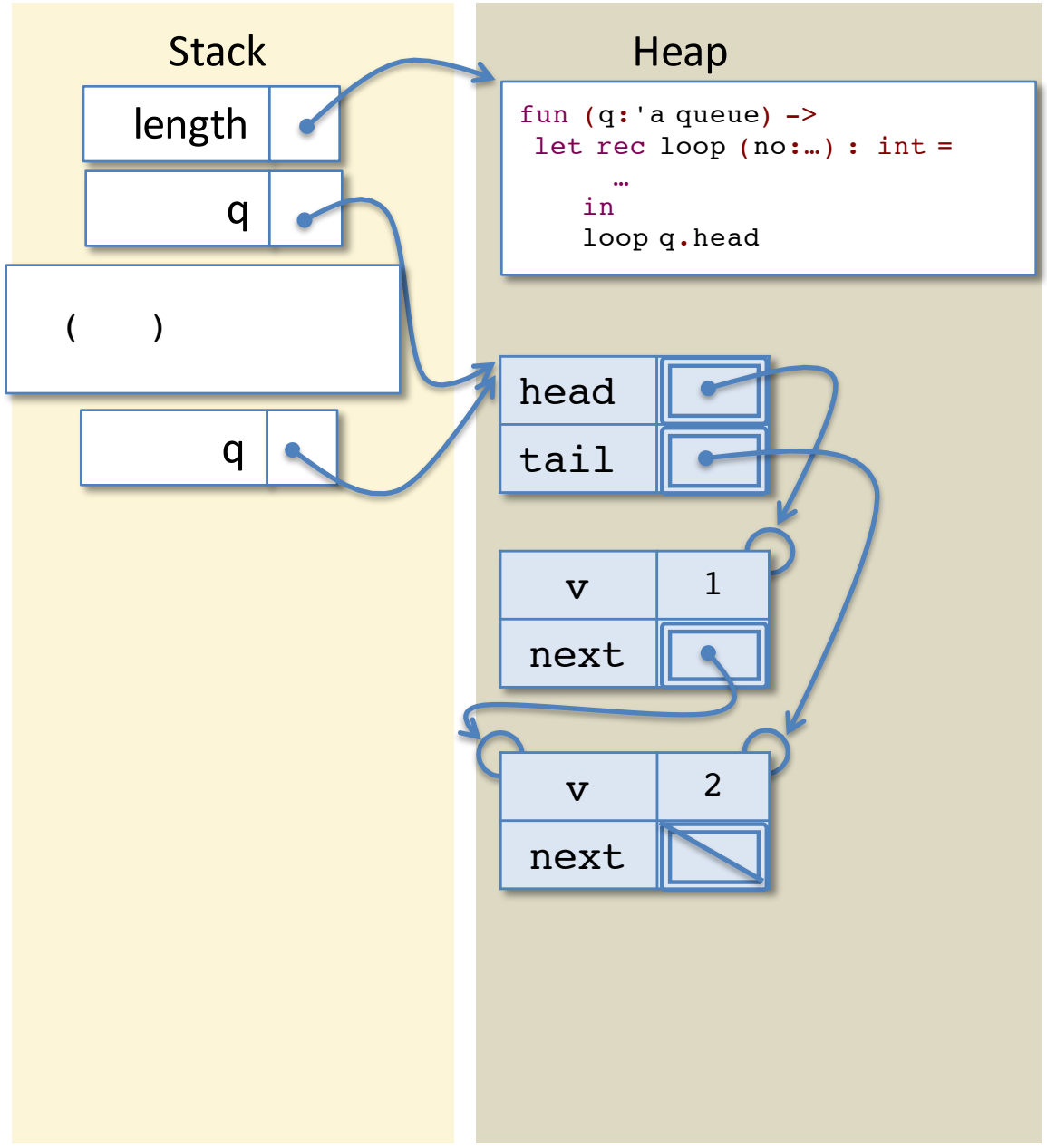
Evaluating length



Evaluating length

Workspace

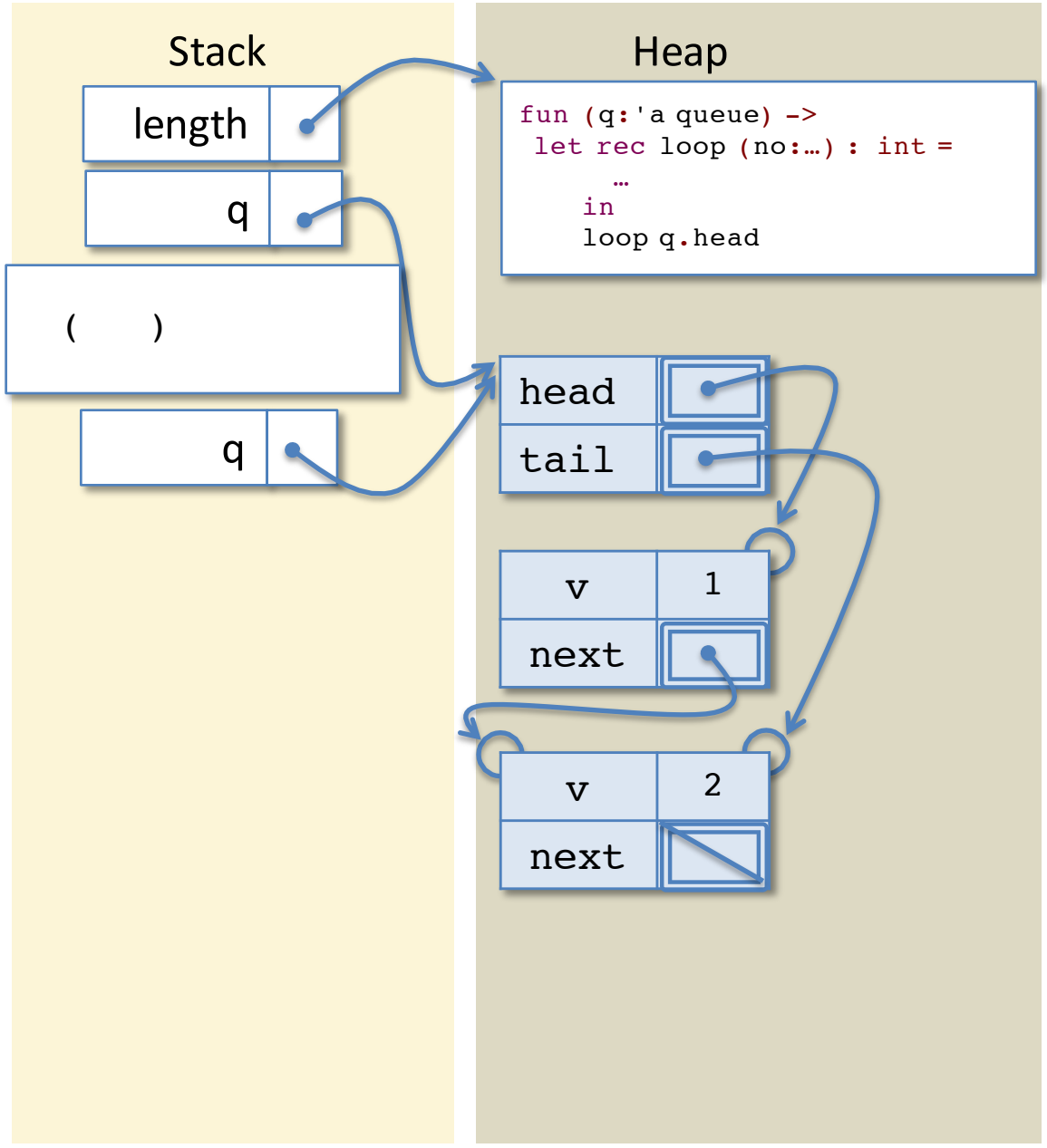
```
let rec loop (no: …) : int =
      begin match no with
         | None -> 0
         | Some n -> 1 + (loop n.next)
      end
    in
    loop q.head
```



Evaluating length

Workspace

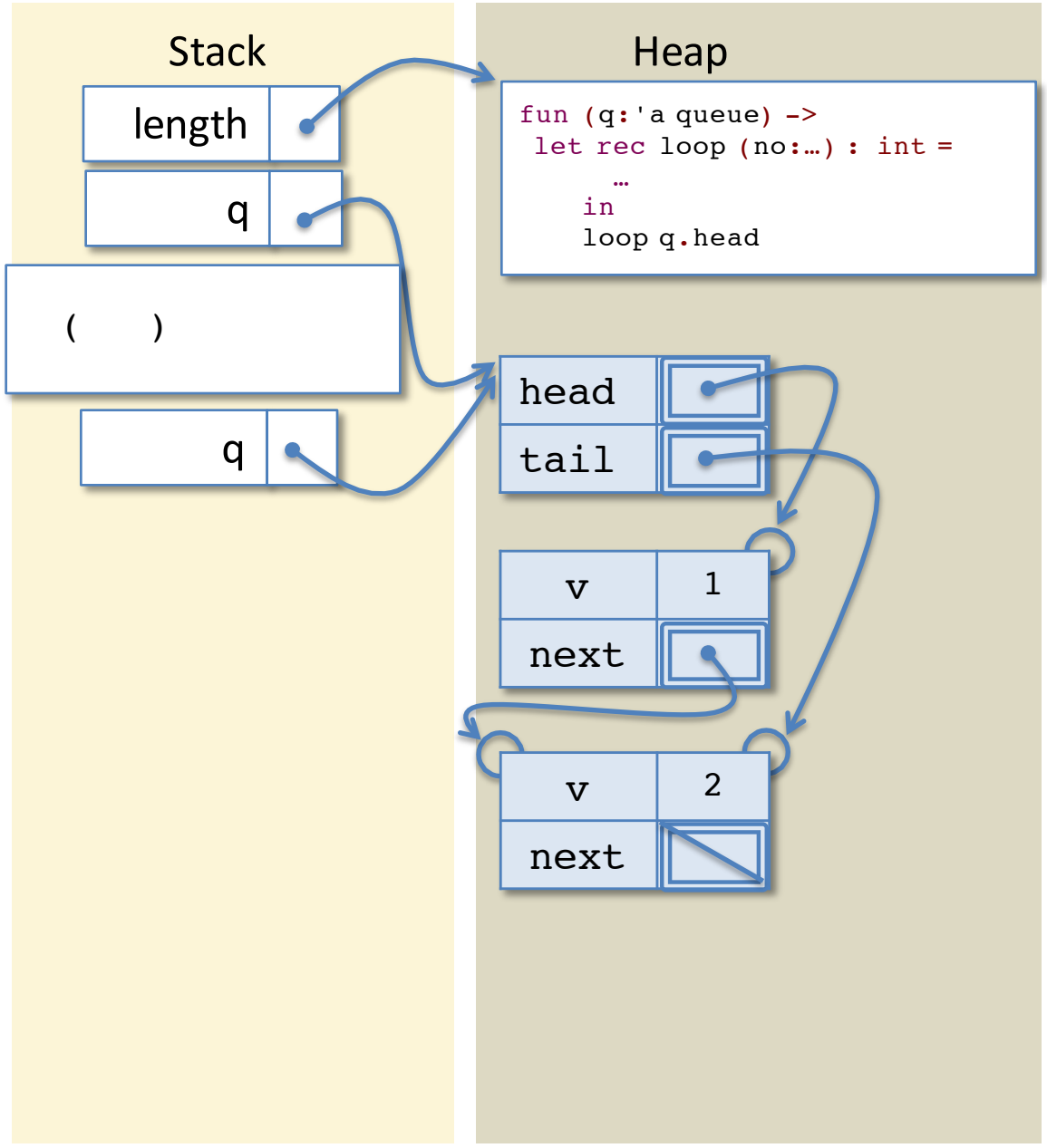
```
let loop = fun (no: …) ->
      begin match no with
         | None -> 0
         | Some n -> 1 + (loop n.next)
      end
    in
    loop q.head
```



Evaluating length

Workspace

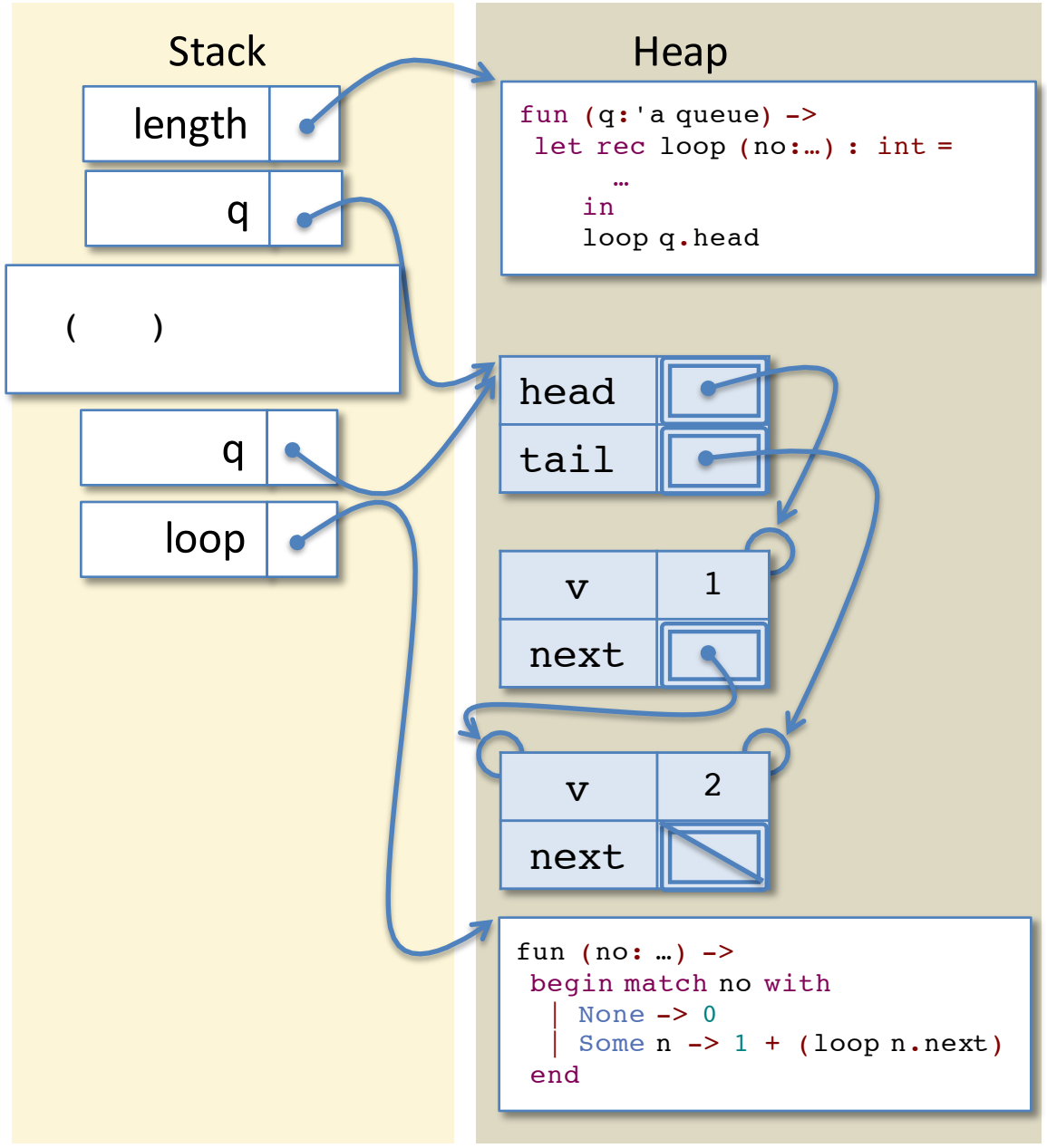
```
let loop = fun (no: …) ->
      begin match no with
          | None -> 0
          | Some n -> 1 + (loop n.next)
      end
    in
    loop q.head
```



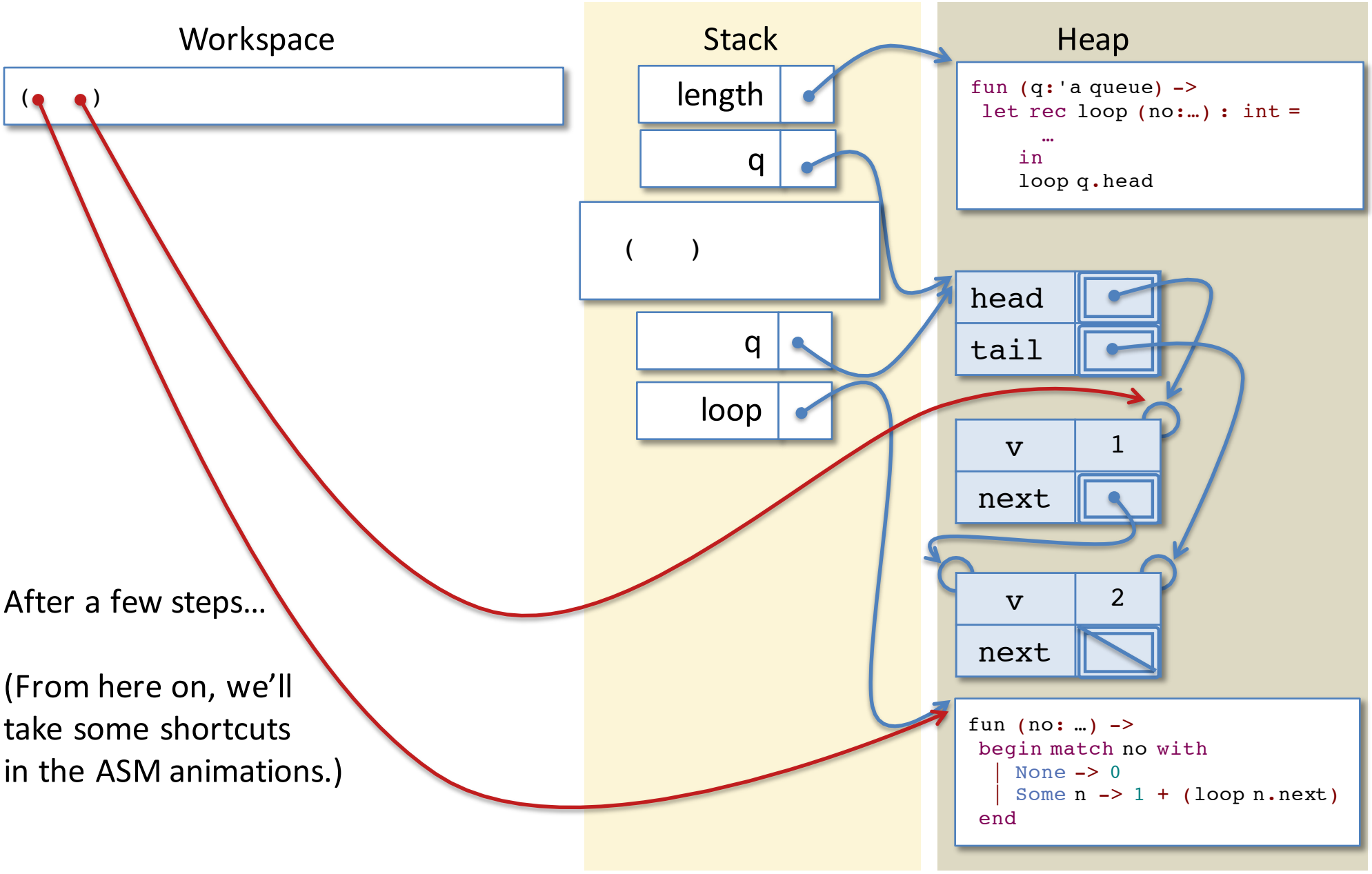
Evaluating length

Workspace

```
loop q.head
```



Evaluating length



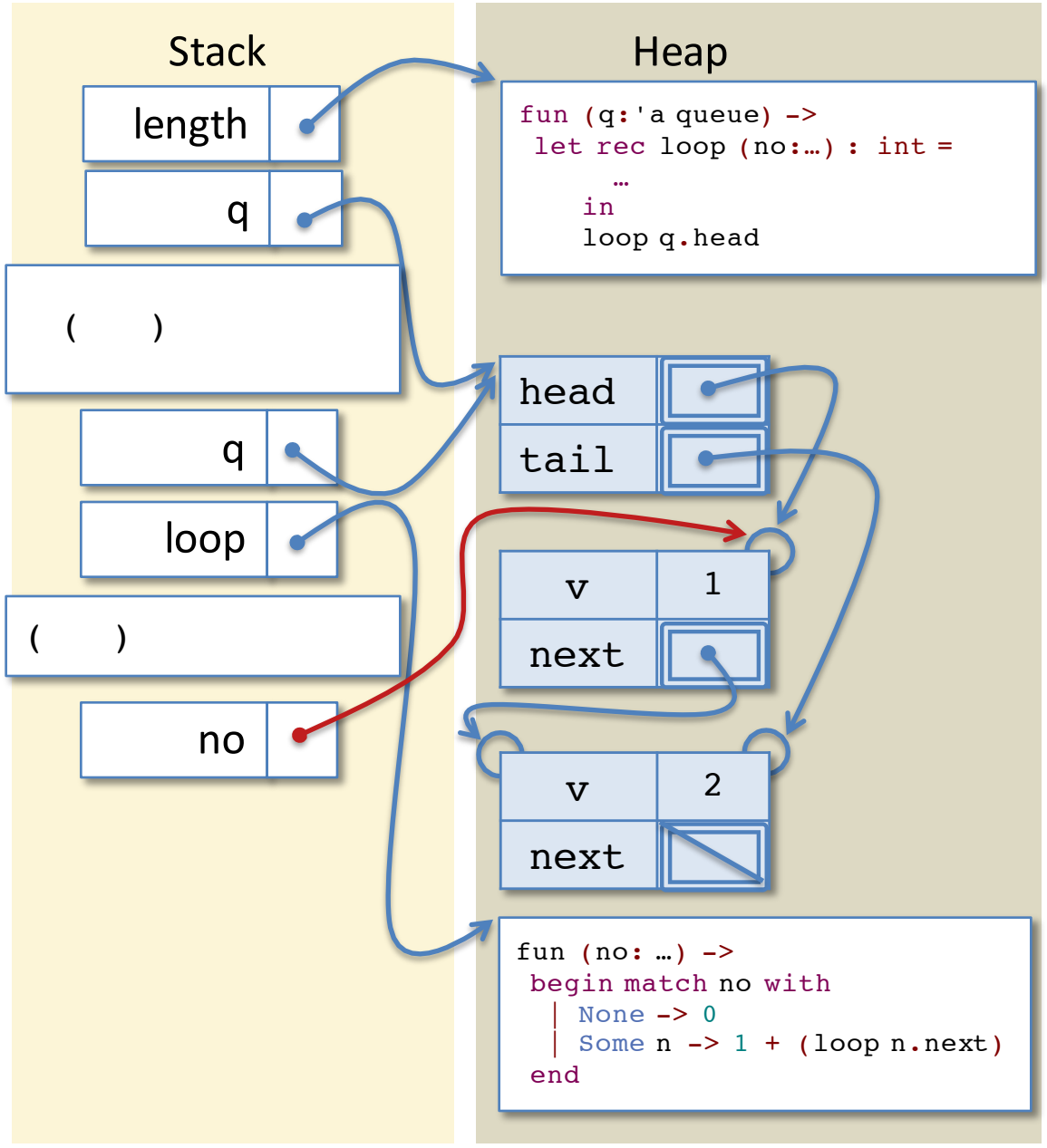
After a few steps…

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

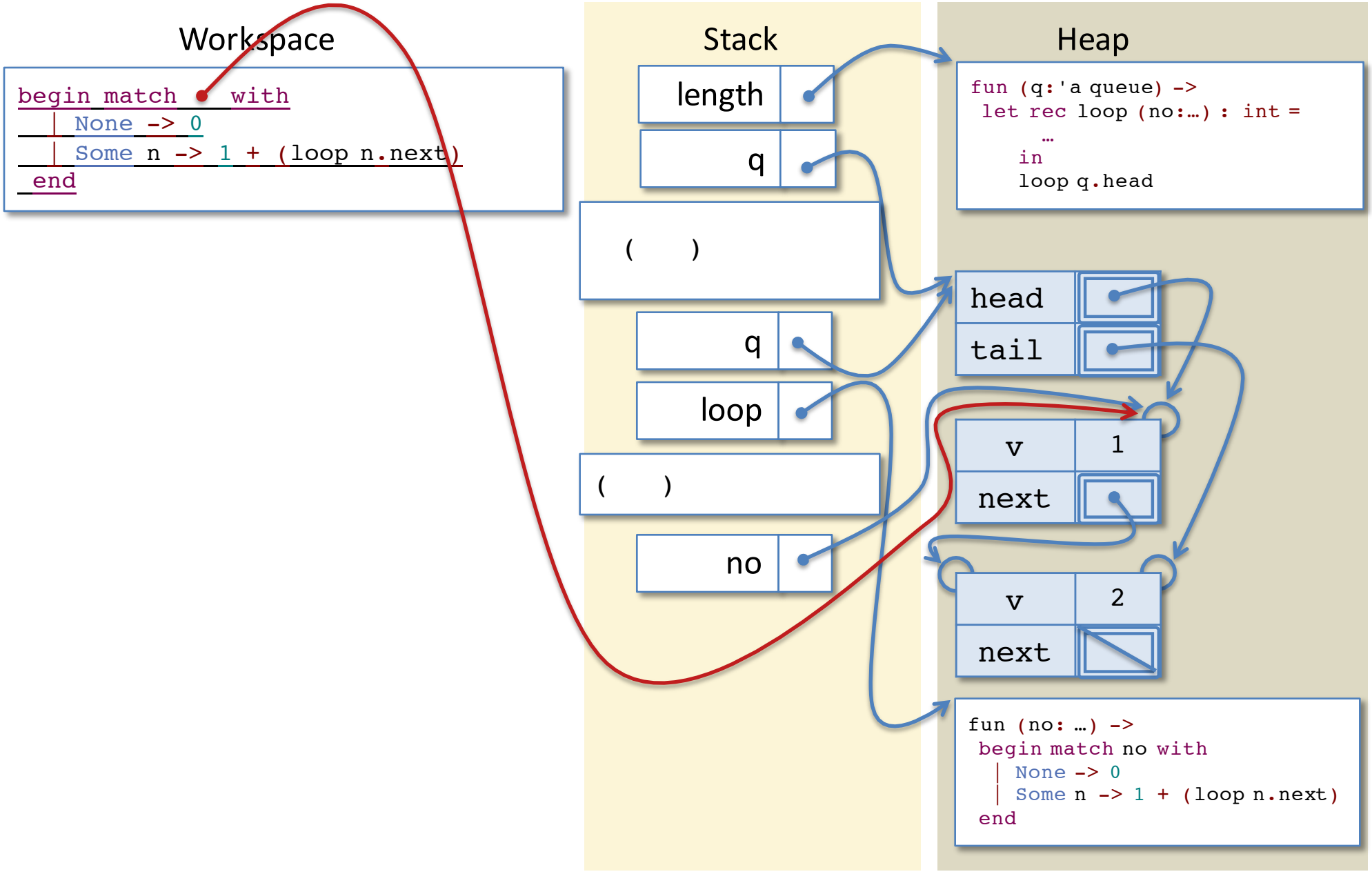
Evaluating length

Workspace

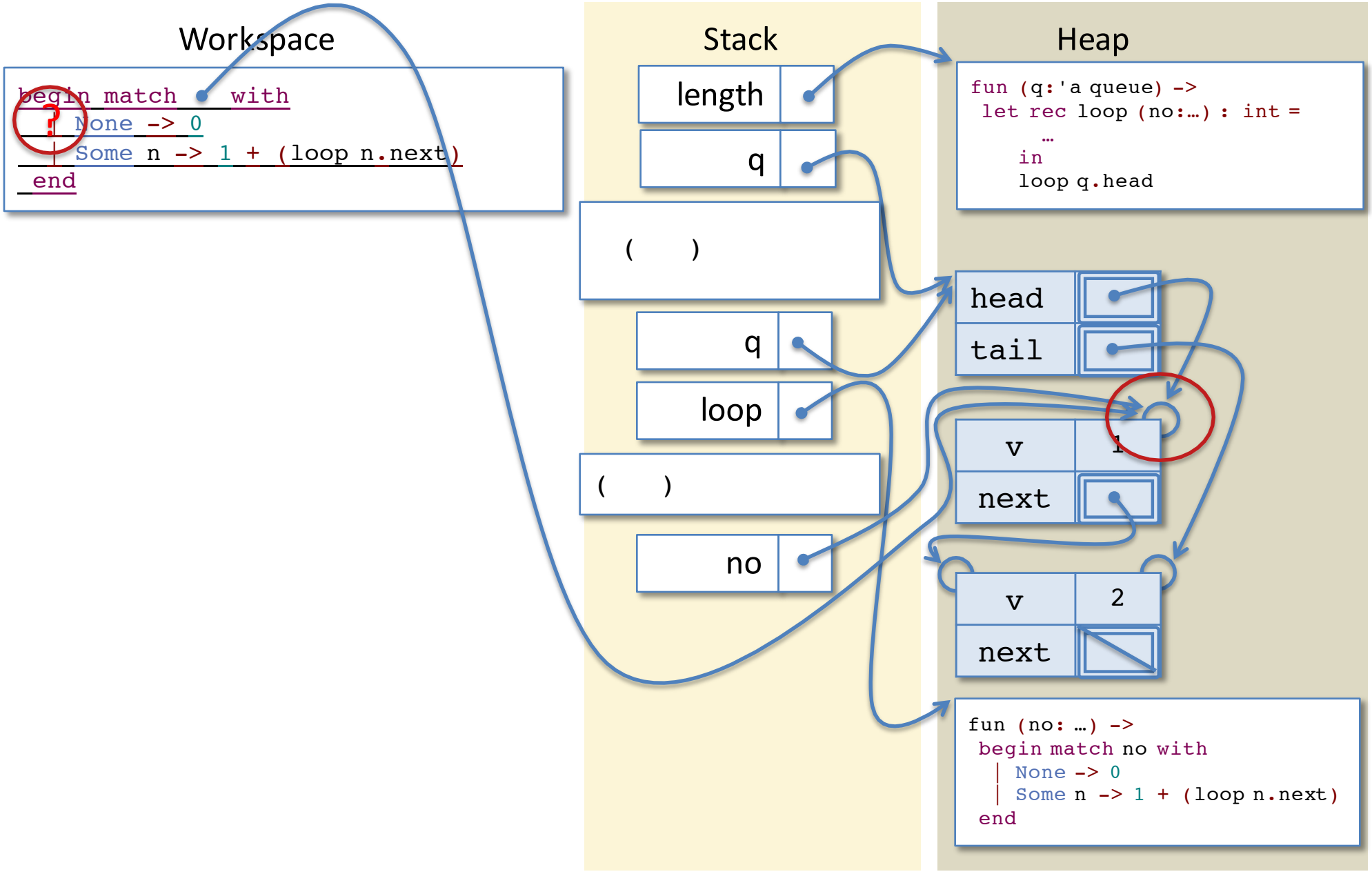
```
begin match no with
   | None -> 0
   | Some n -> 1 + (loop n.next)
 end
```



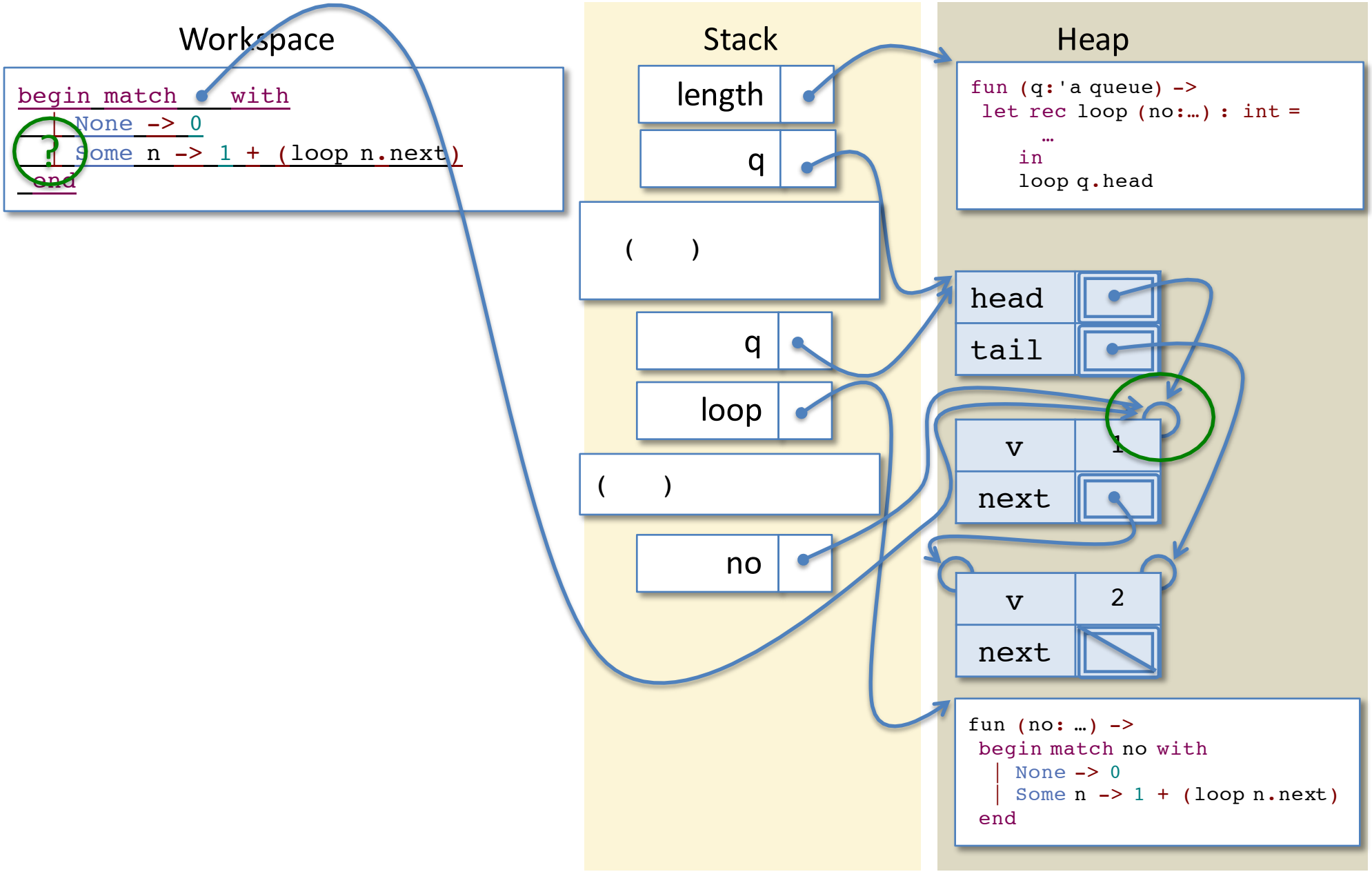
Evaluating length



Evaluating length



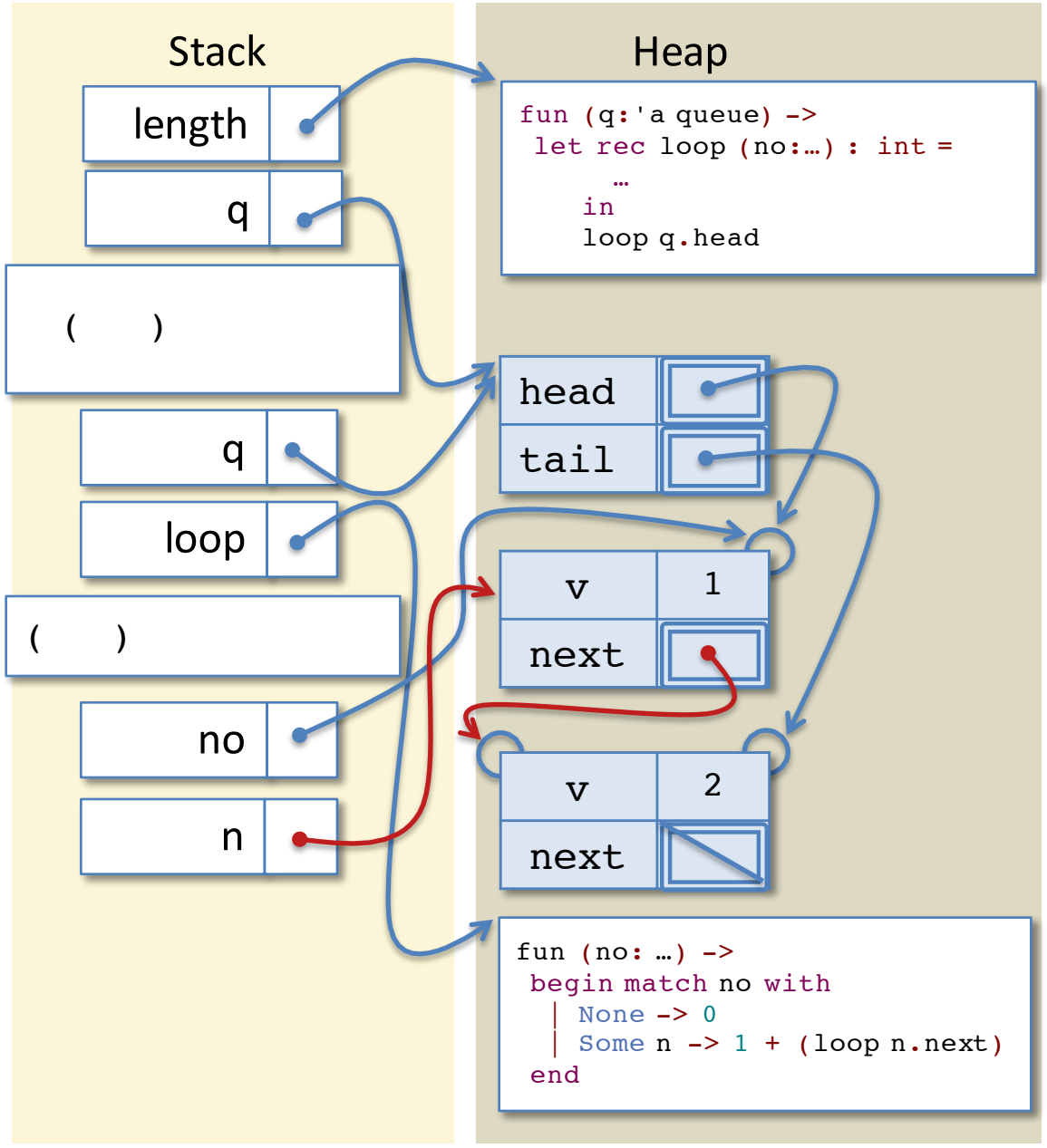
Evaluating length



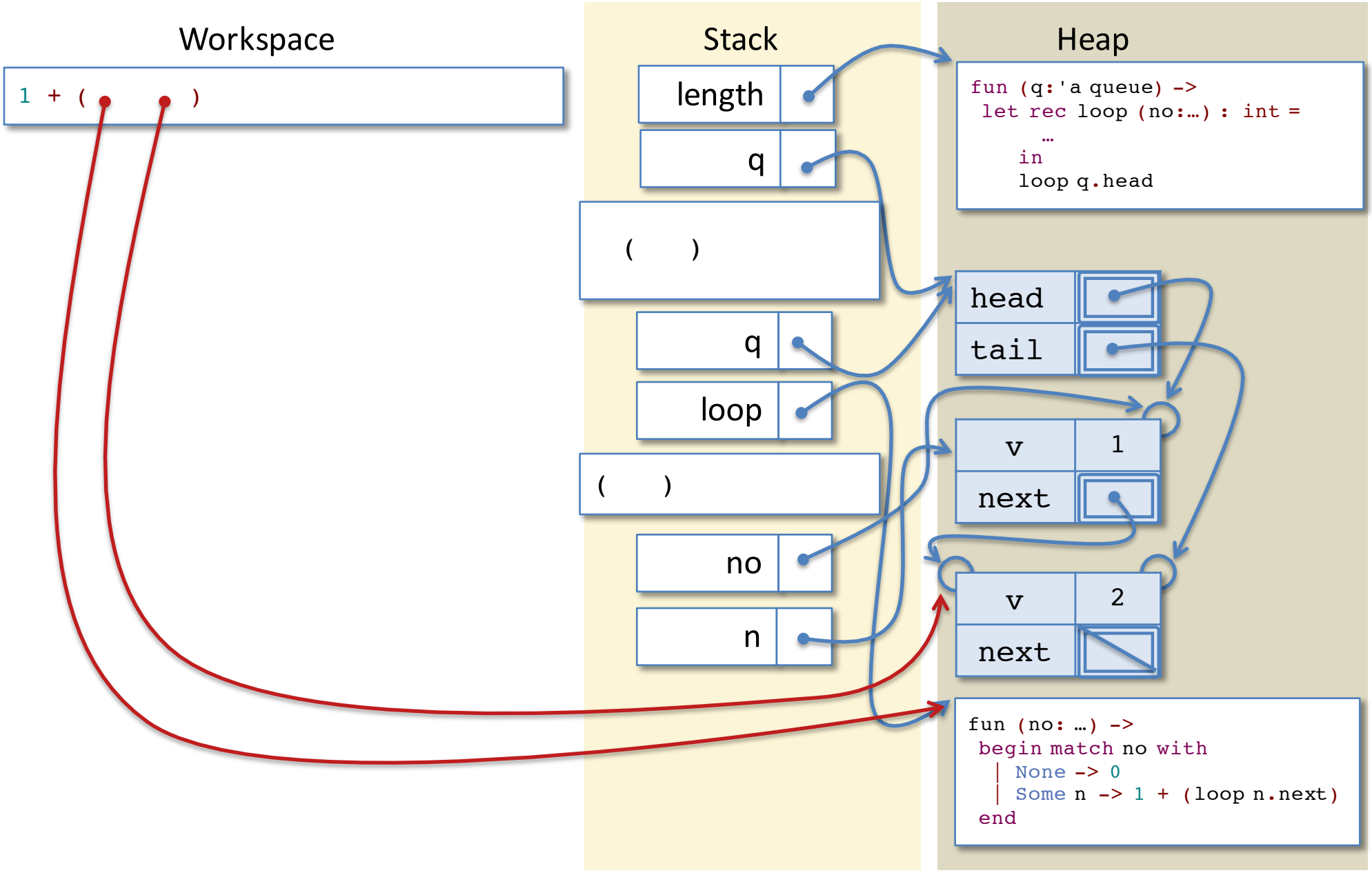
Evaluating length

Workspace

```
1 + (loop n.next)
```



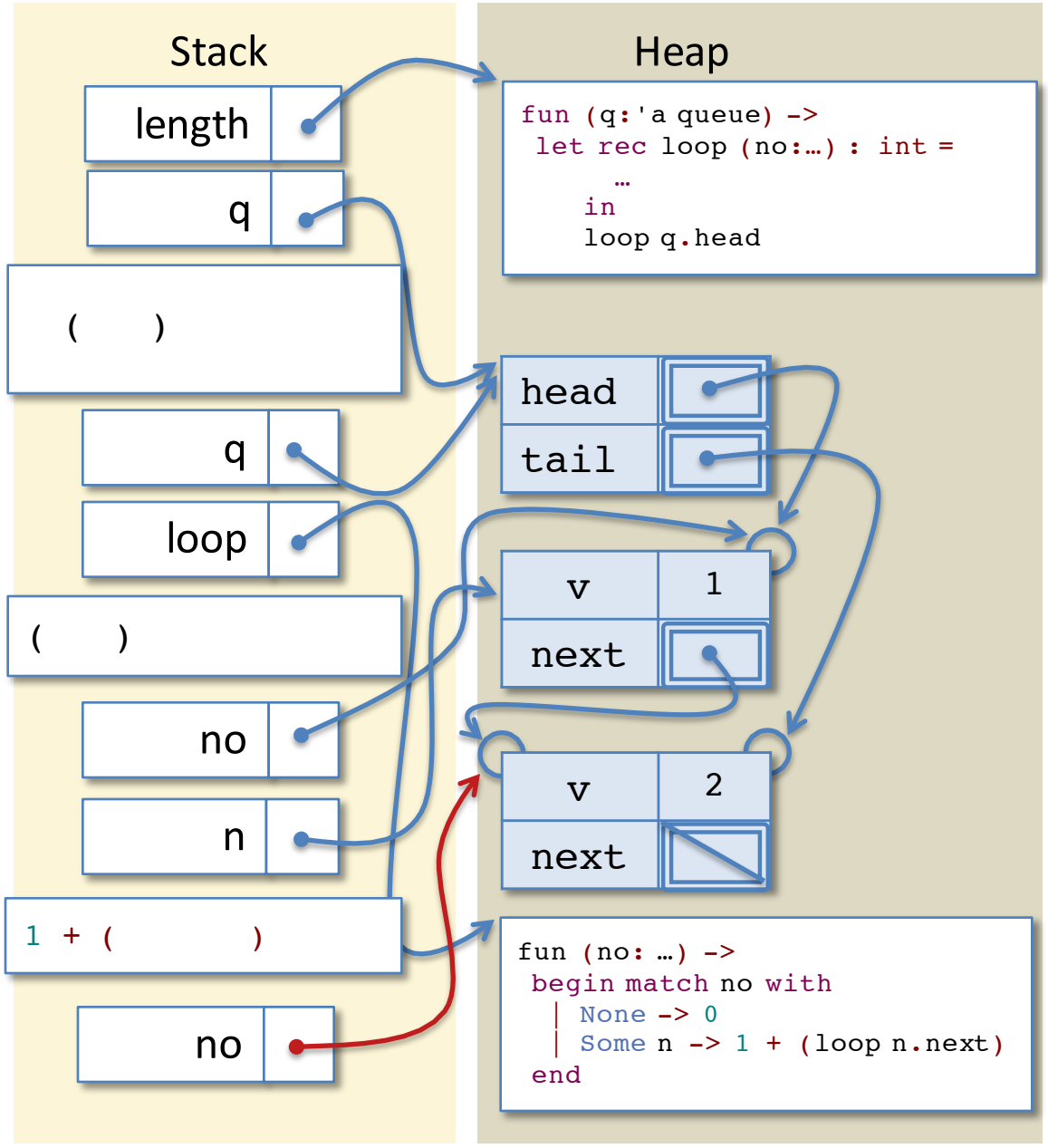
Evaluating length



Evaluating length

Workspace

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

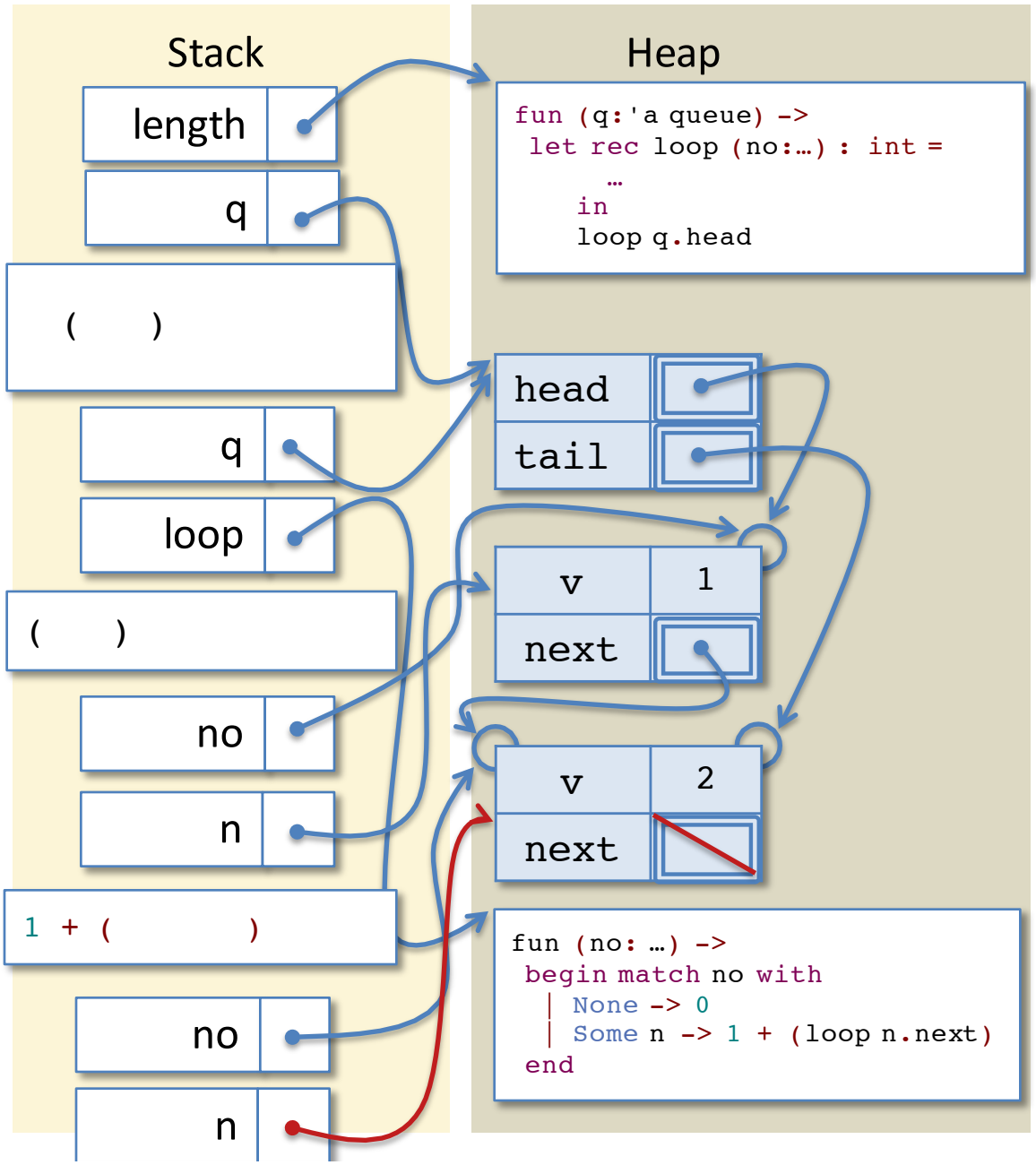


…after a few steps…

Evaluating length

Workspace

```
1 + (loop n.next)
```

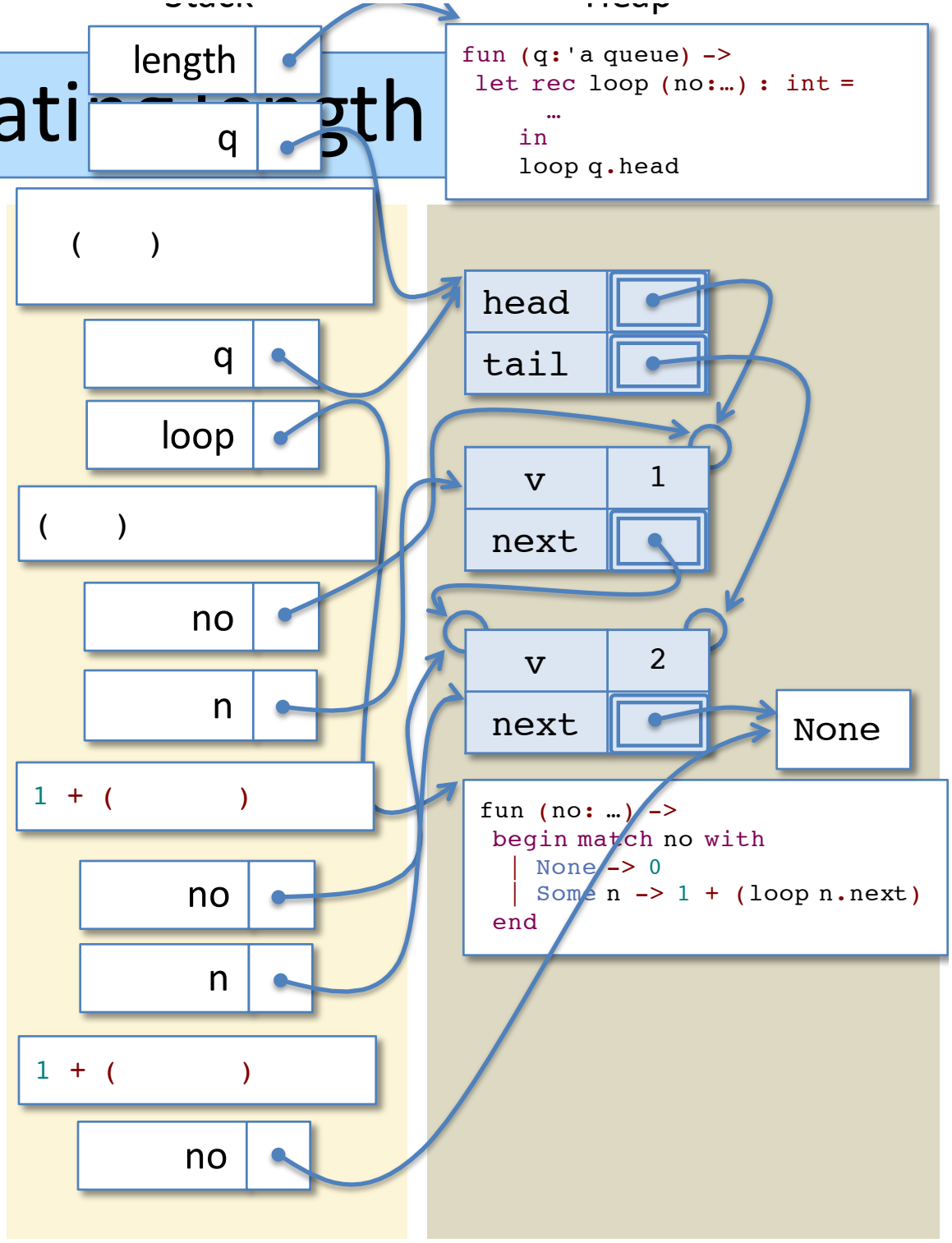


…after a few more steps…

Evaluating length

Workspace

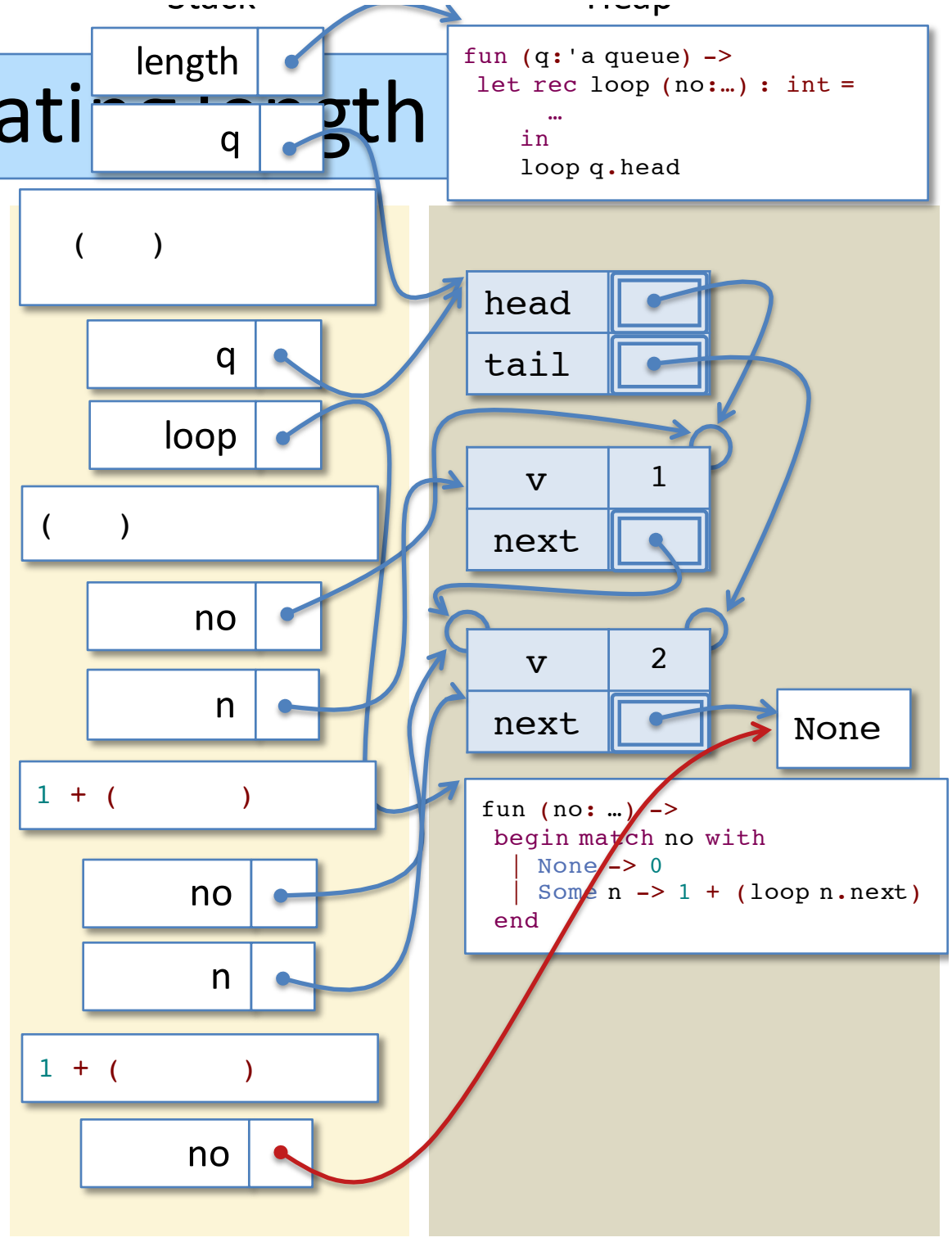
```
begin match no with
   | None -> 0
   | Some n -> 1 + (loop n.next)
 end
```



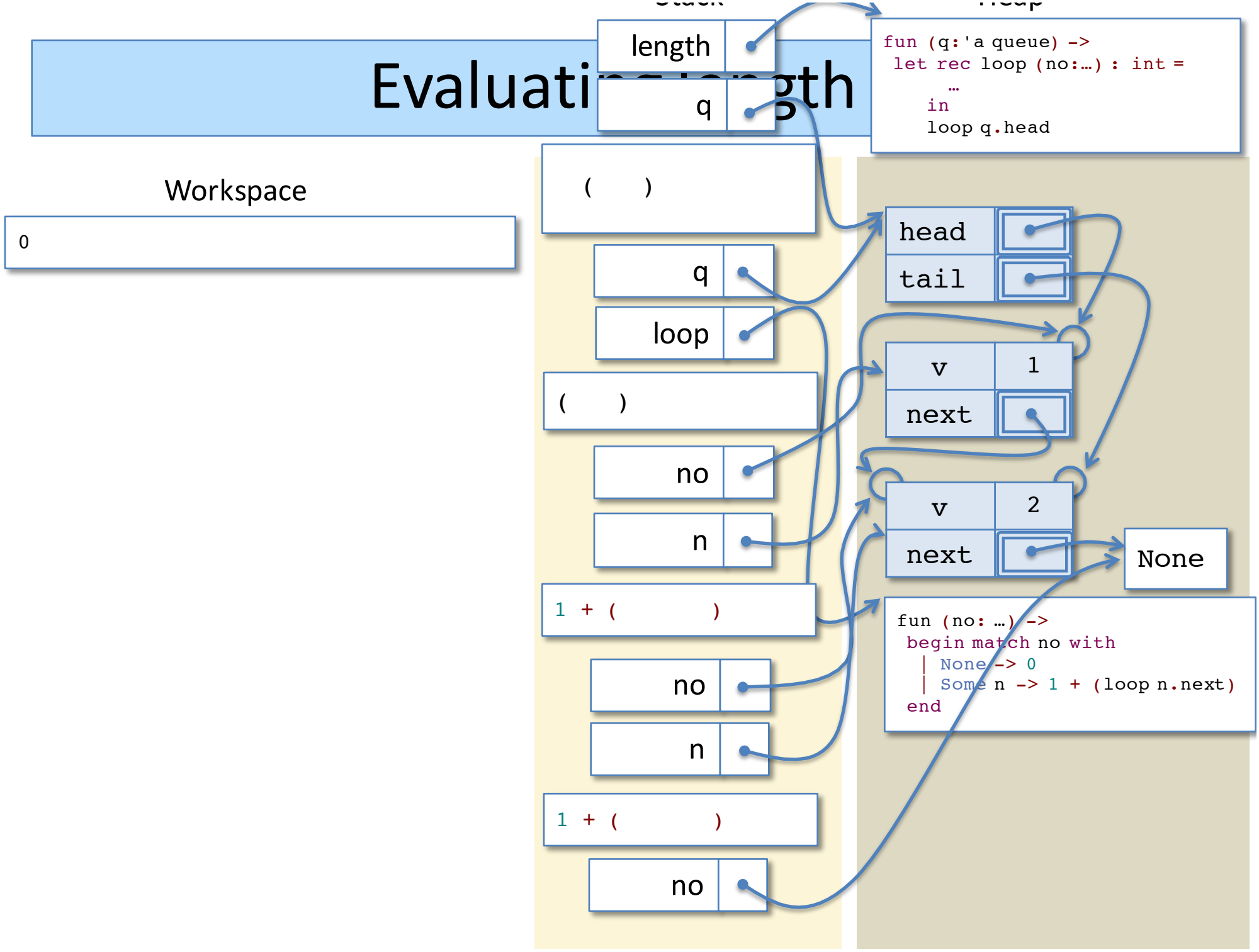
Evaluating length

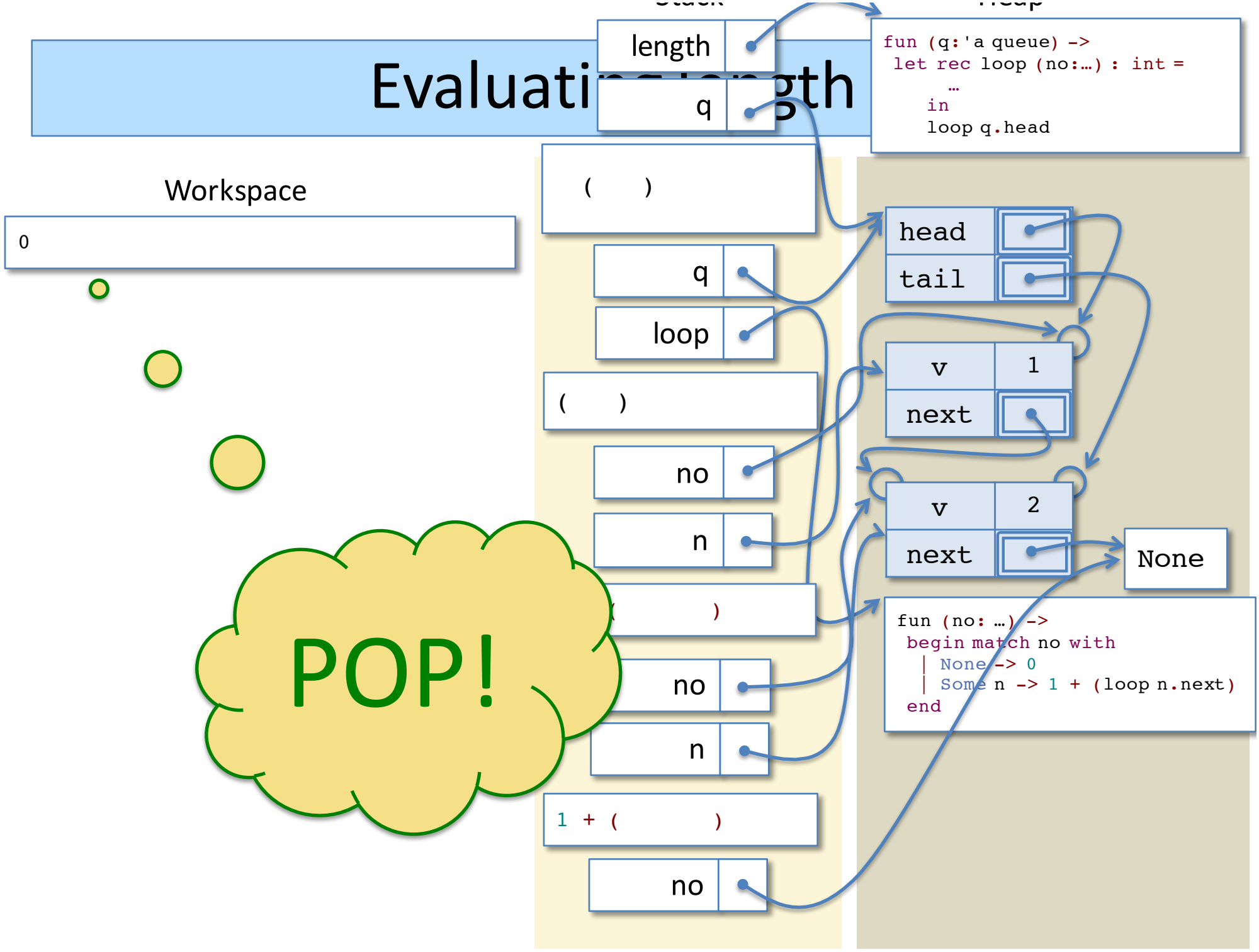
Workspace

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



Evaluating length

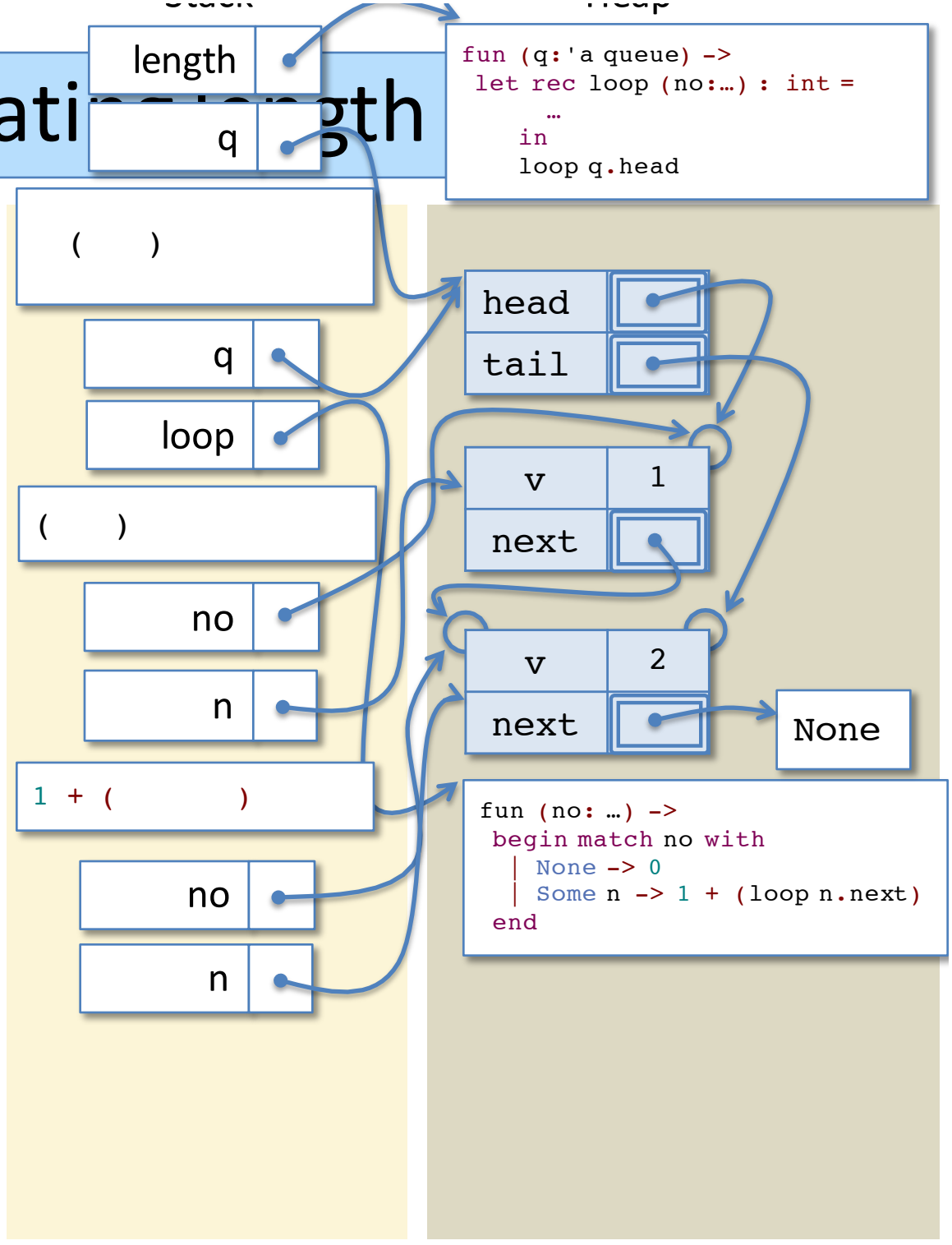


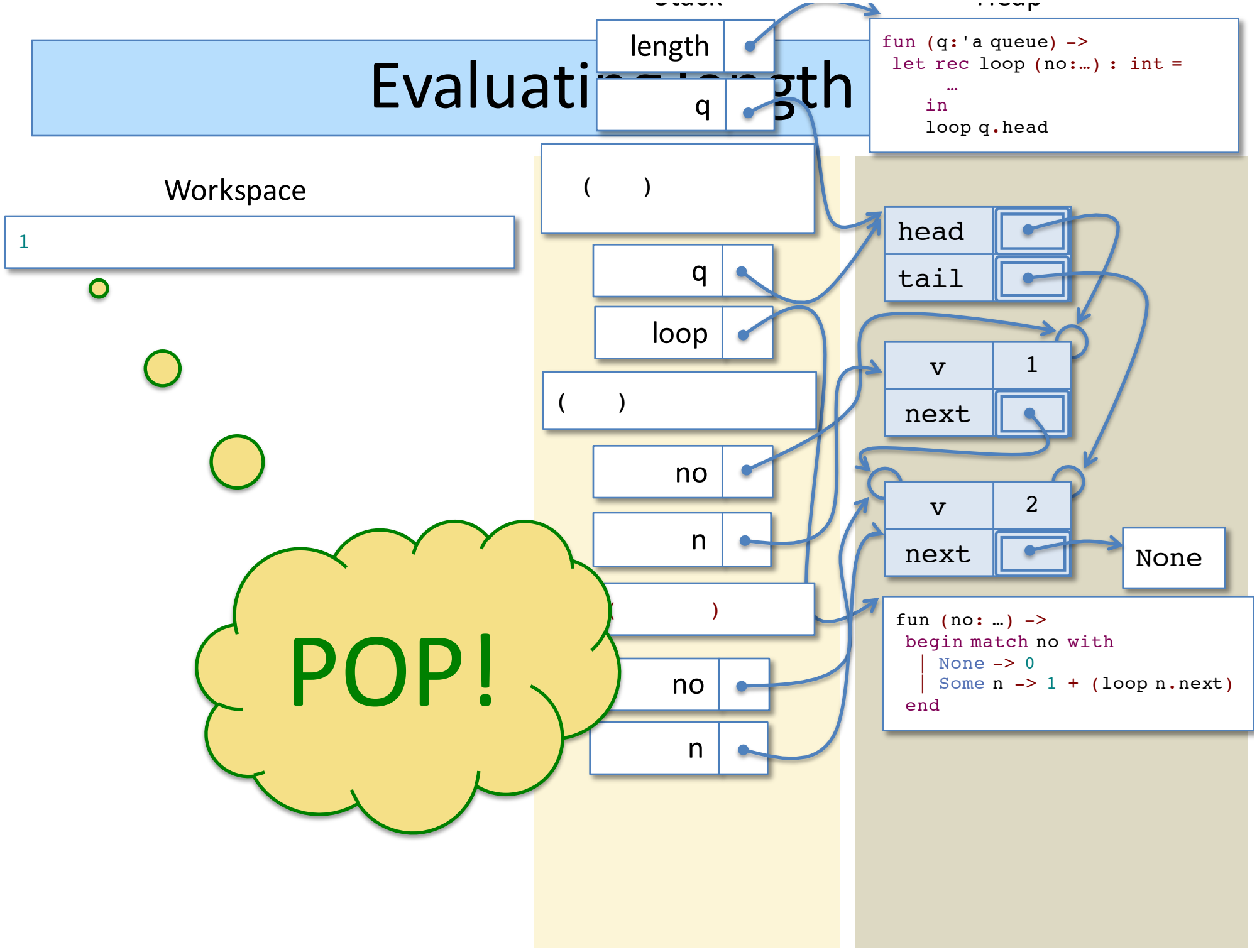

Evaluating length
length
q
fun (q:'a queue) ->
let rec loop (no:…) : int =
…
in
loop q.head
Workspace
0
( )
q
loop
( )
no
n
( )
no
n
1 + ( )
no
head
tail
v
1
next
v
2
next
None
fun (no: …) ->
begin match no with
| None -> 0
| Some n -> 1 + (loop n.next)
end
POP!

Evaluating length

Workspace

```
1 + 0
```

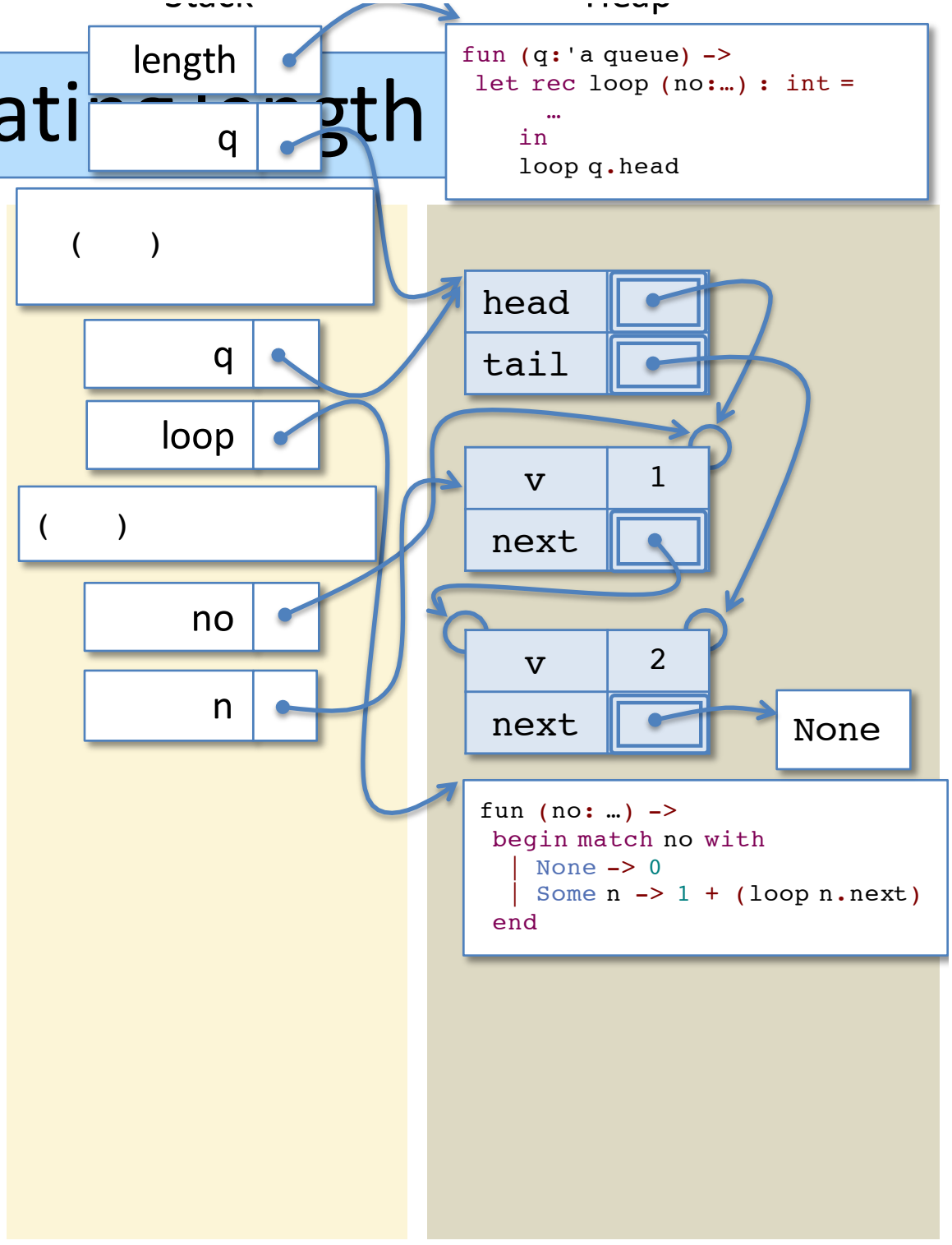



Evaluating length
Workspace
1
length
q
fun (q:'a queue) ->
let rec loop (no:…) : int =
…
in
loop q.head
( )
q
loop
( )
no
n
( )
no
n
head
tail
v
1
next
v
2
next
None
fun (no: …) ->
begin match no with
| None -> 0
| Some n -> 1 + (loop n.next)
end
POP!

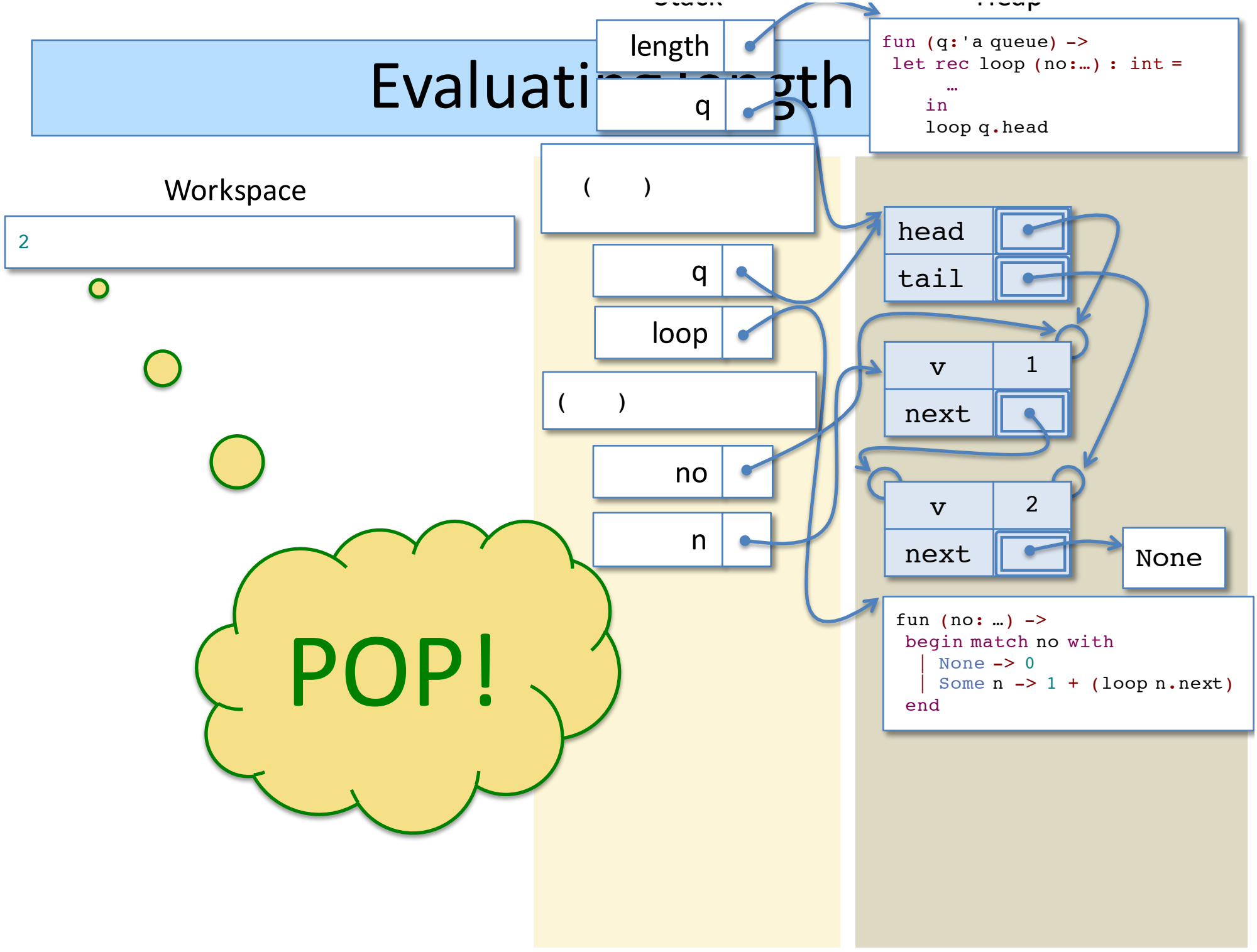
Evaluating length

Workspace

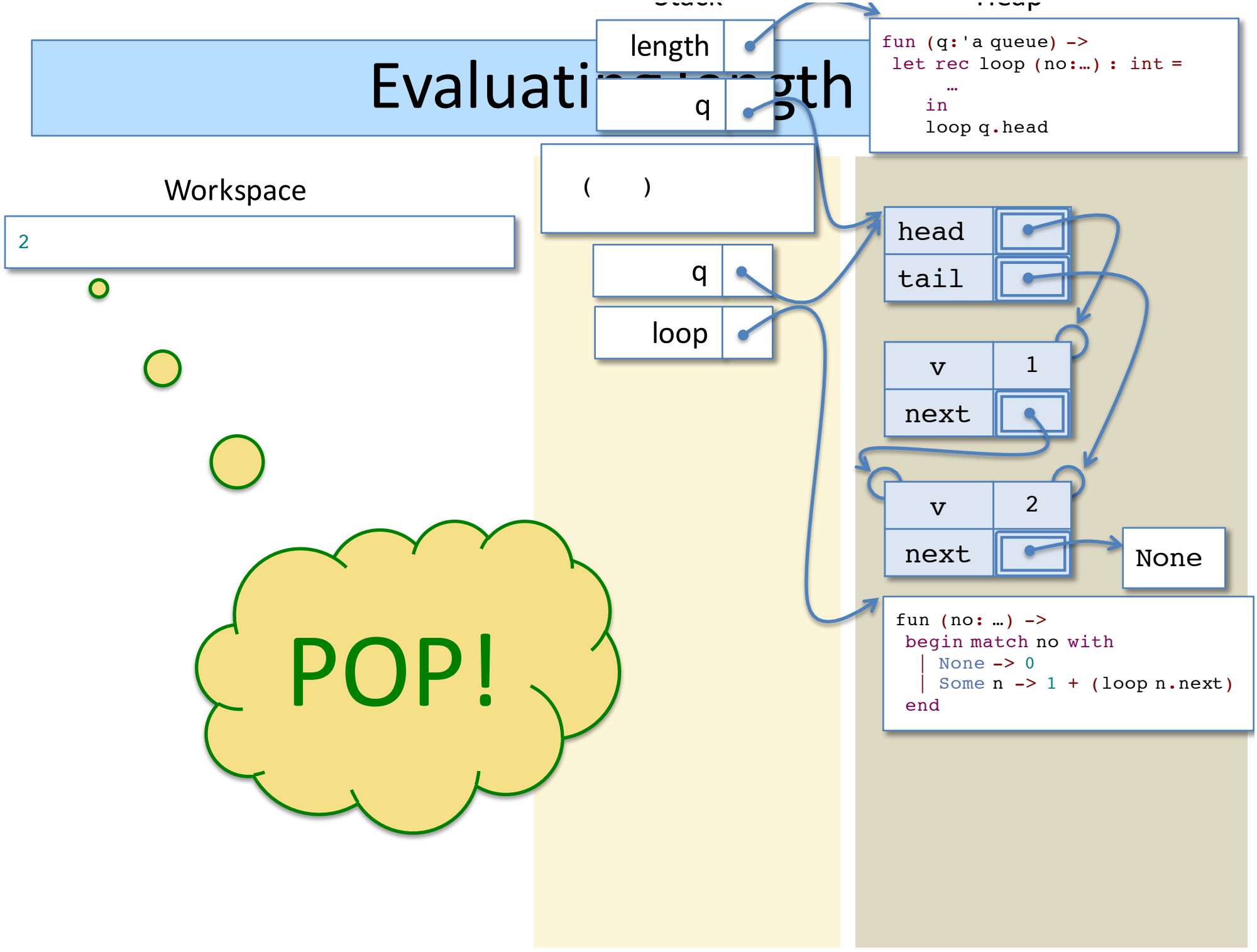
```
1 + 1
```



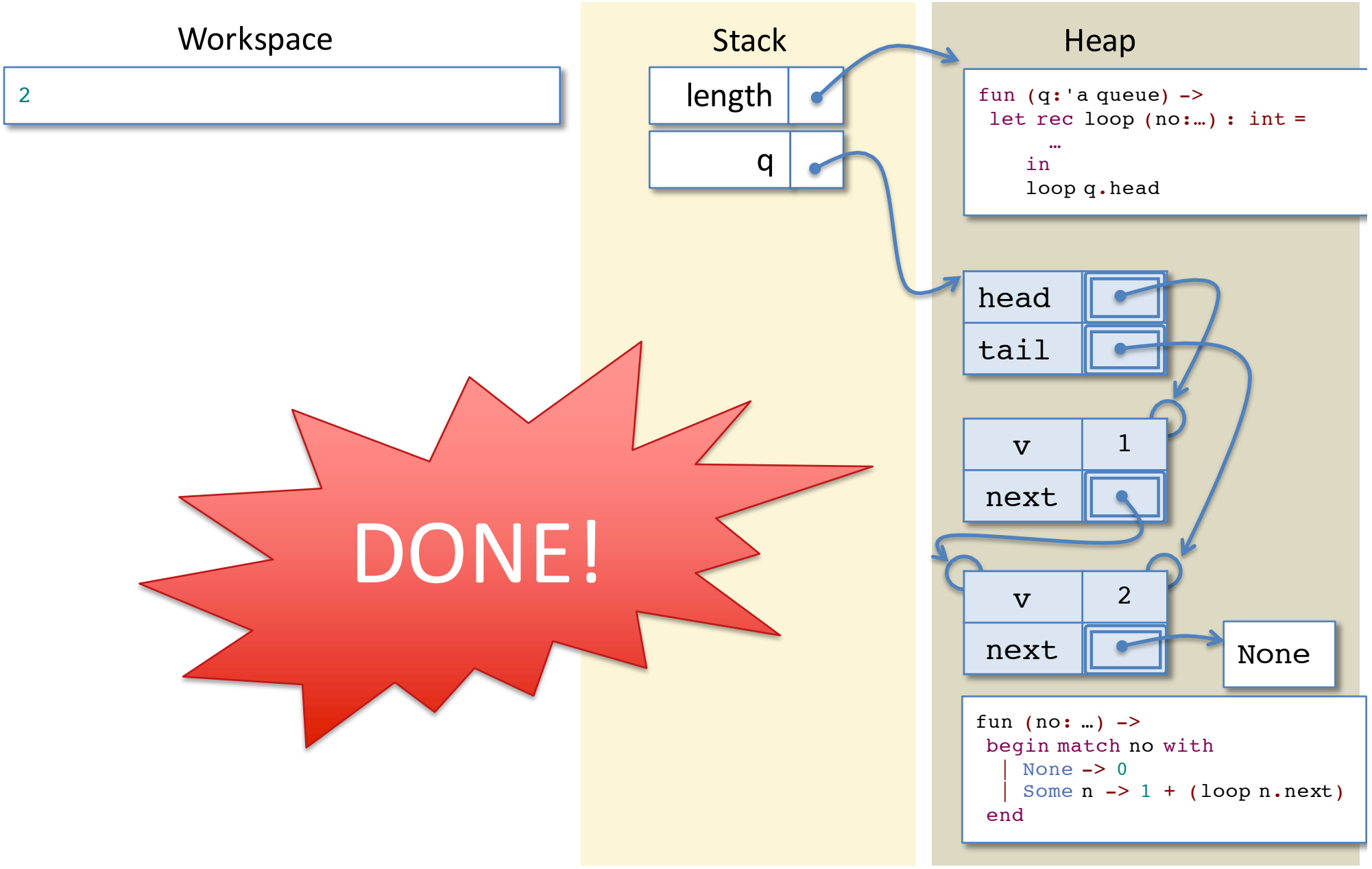
Evaluating length



Evaluating length



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 + (`loop` …)) in the recursive version.

Tail Call Optimization

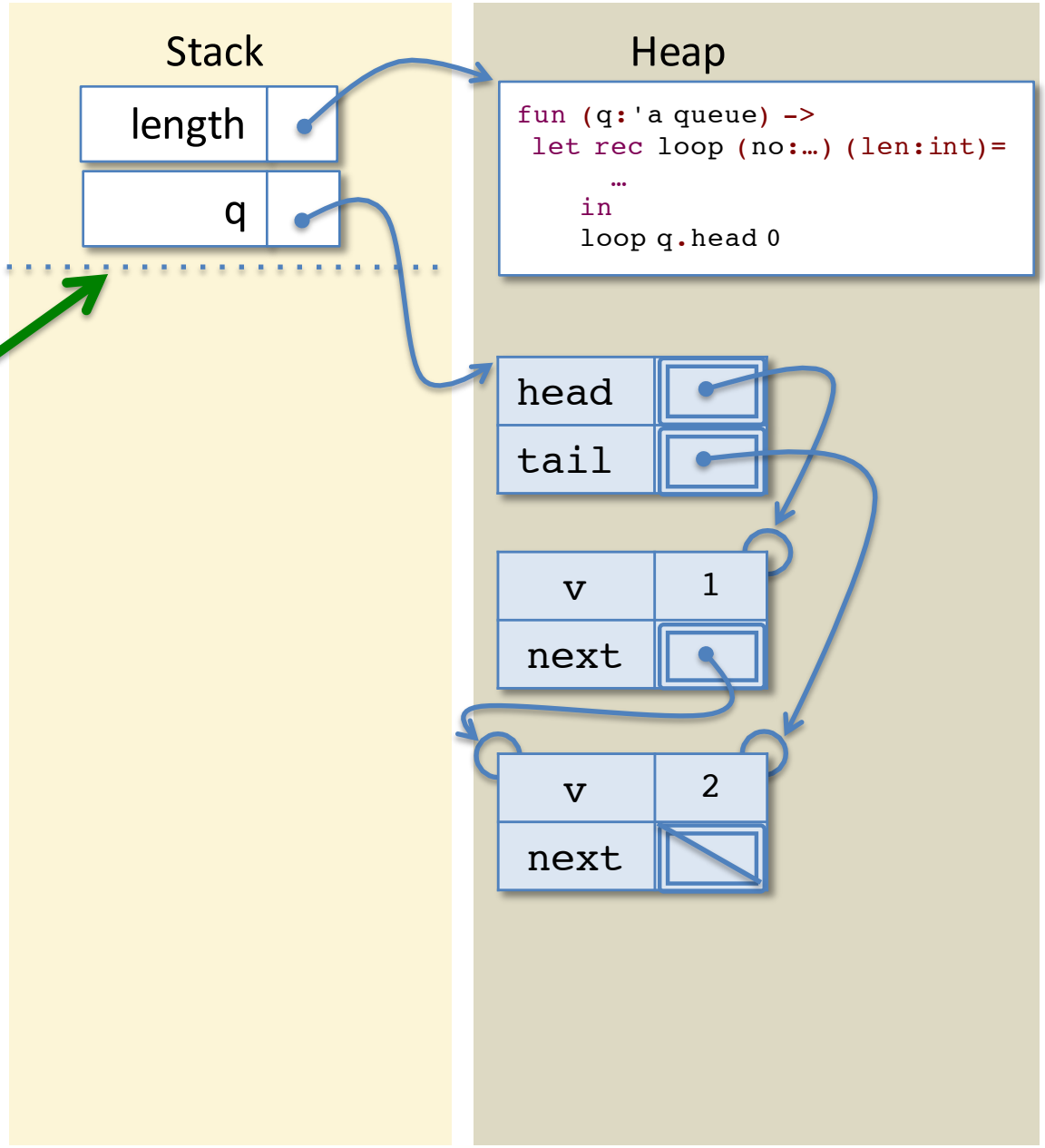
- Why does it matter that 'loop' is only called in an empty workspace?
- We can *optimize* the abstract stack machine:
  - The workspace pushed onto the stack tells us "what to do" when the function call returns.
  - If the pushed workspace is empty, we will always 'pop' immediately after the function call returns.
  - So there is no need to save the empty workspace on the stack!
  - Moreover, any local variables that were pushed 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

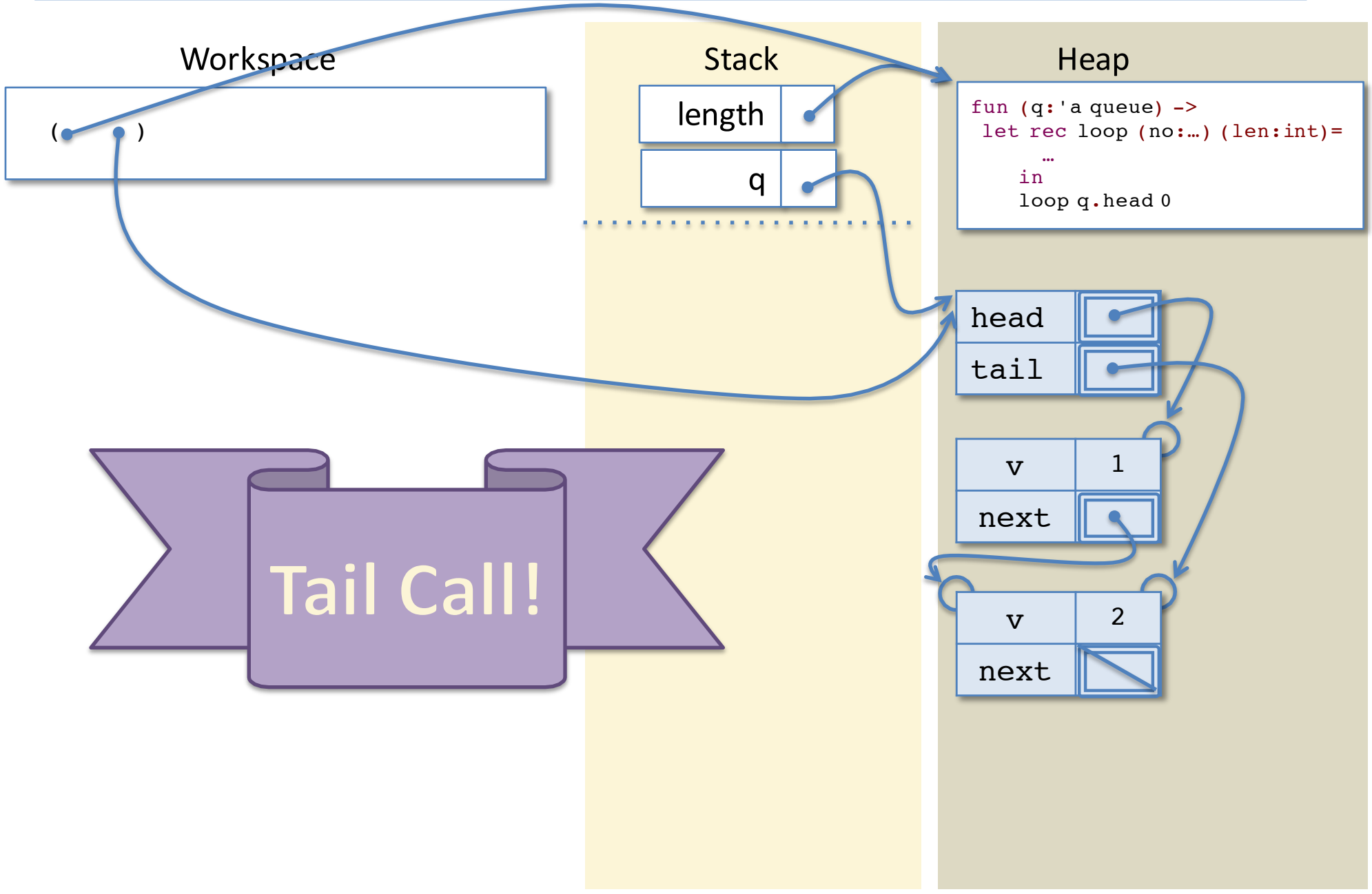
Workspace

```
length q
```

Bindings above this line are top-level declarations.



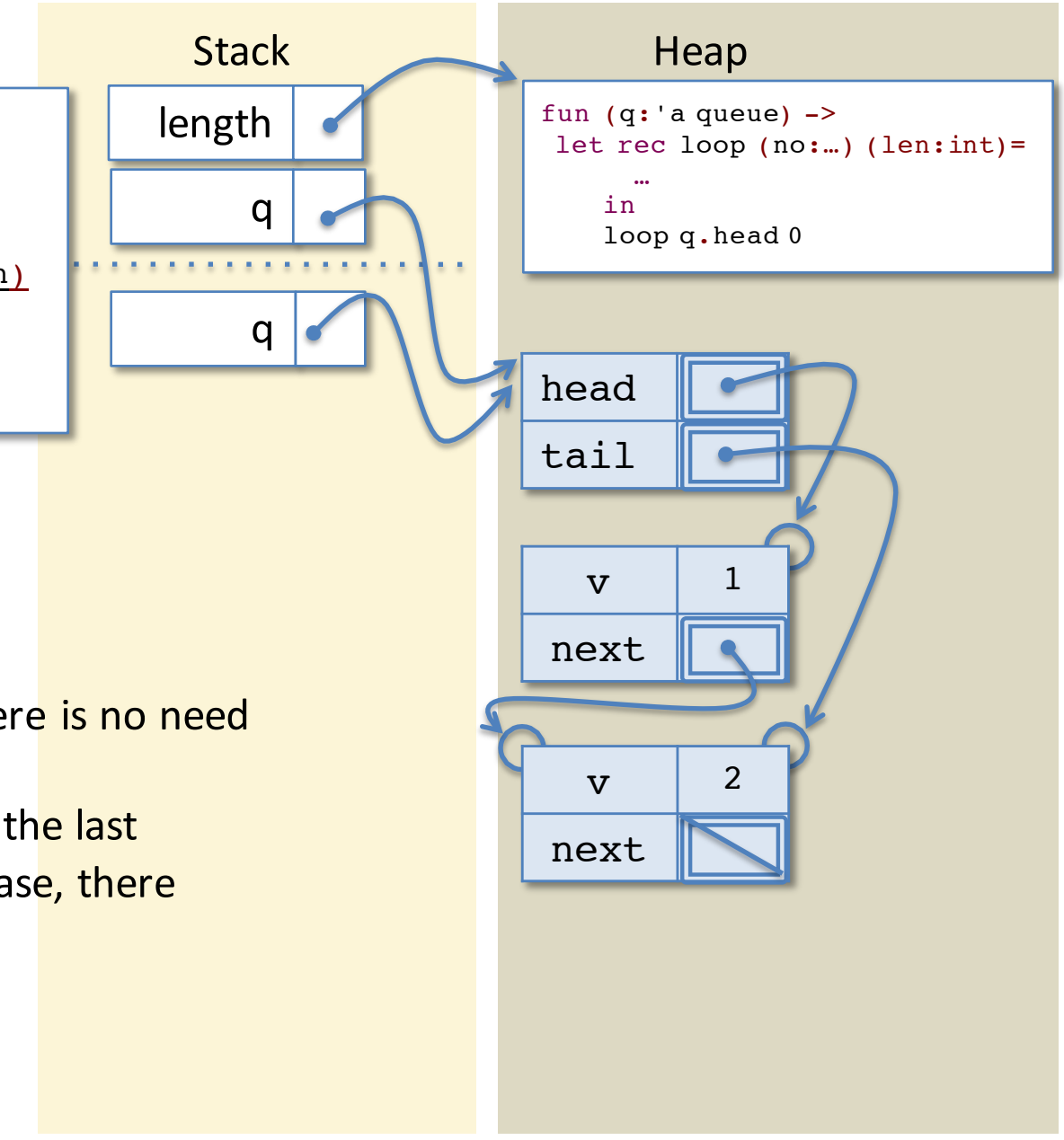
Tail Calls and Iterative length



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
```



Note:

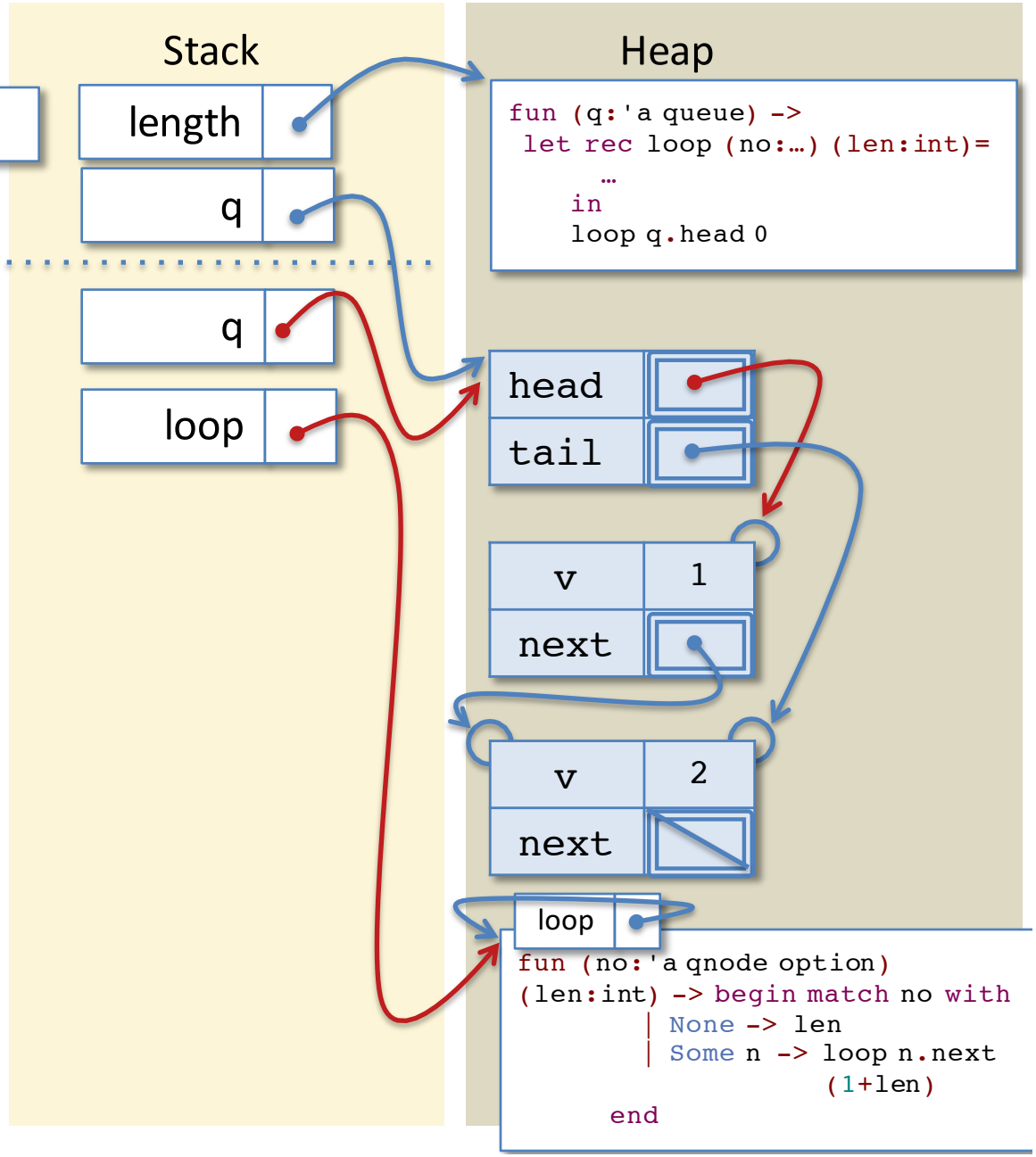
(1) No workspace is saved – there is no need do to that for tail calls

(2) We pop all the locals, up to the last saved workspace. (In this case, there weren't any anyway.)

Tail Calls and Iterative length

Workspace

```
(loop q.head 0)
```

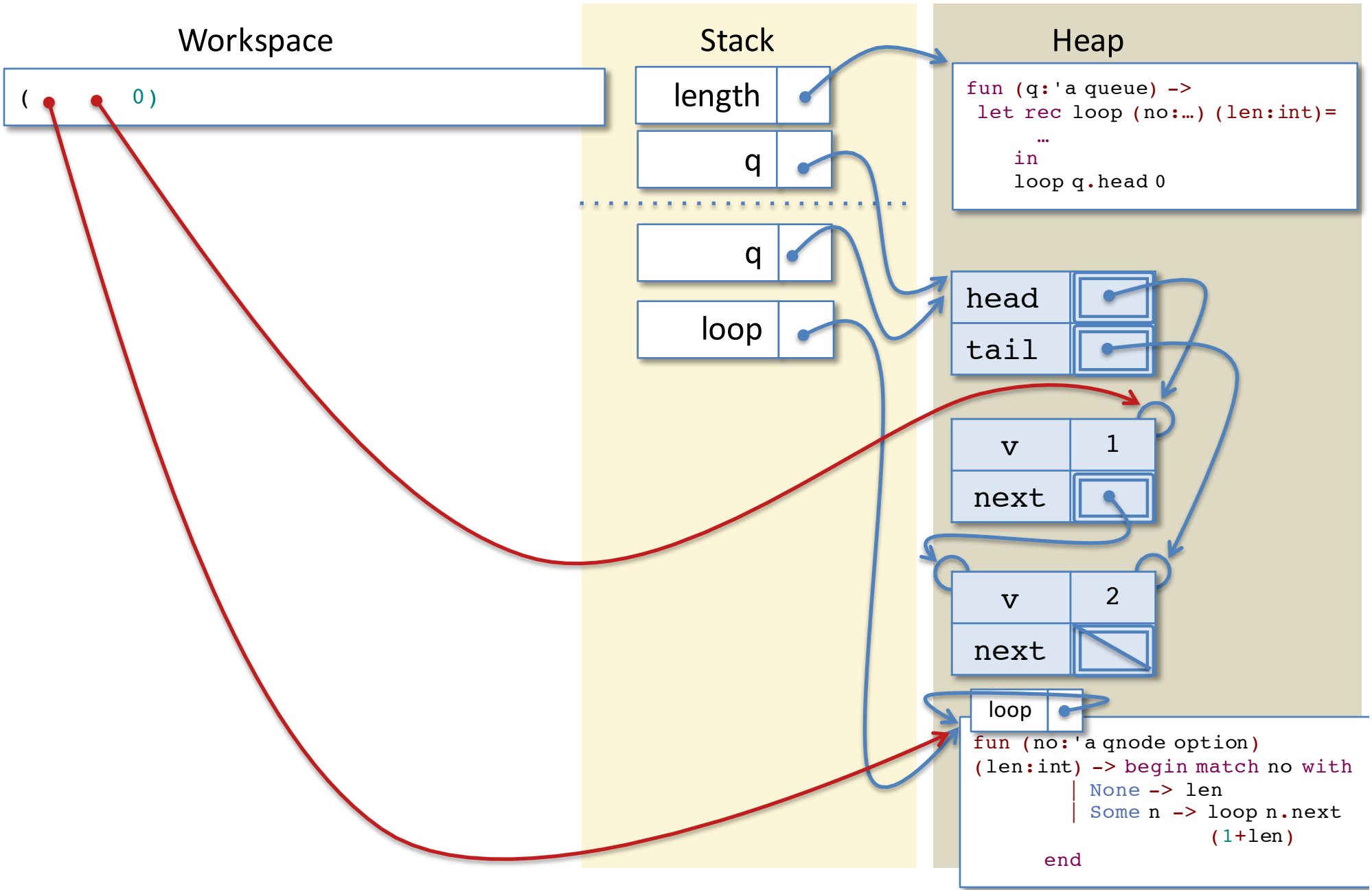


A detail we've been sweeping under the rug until now:

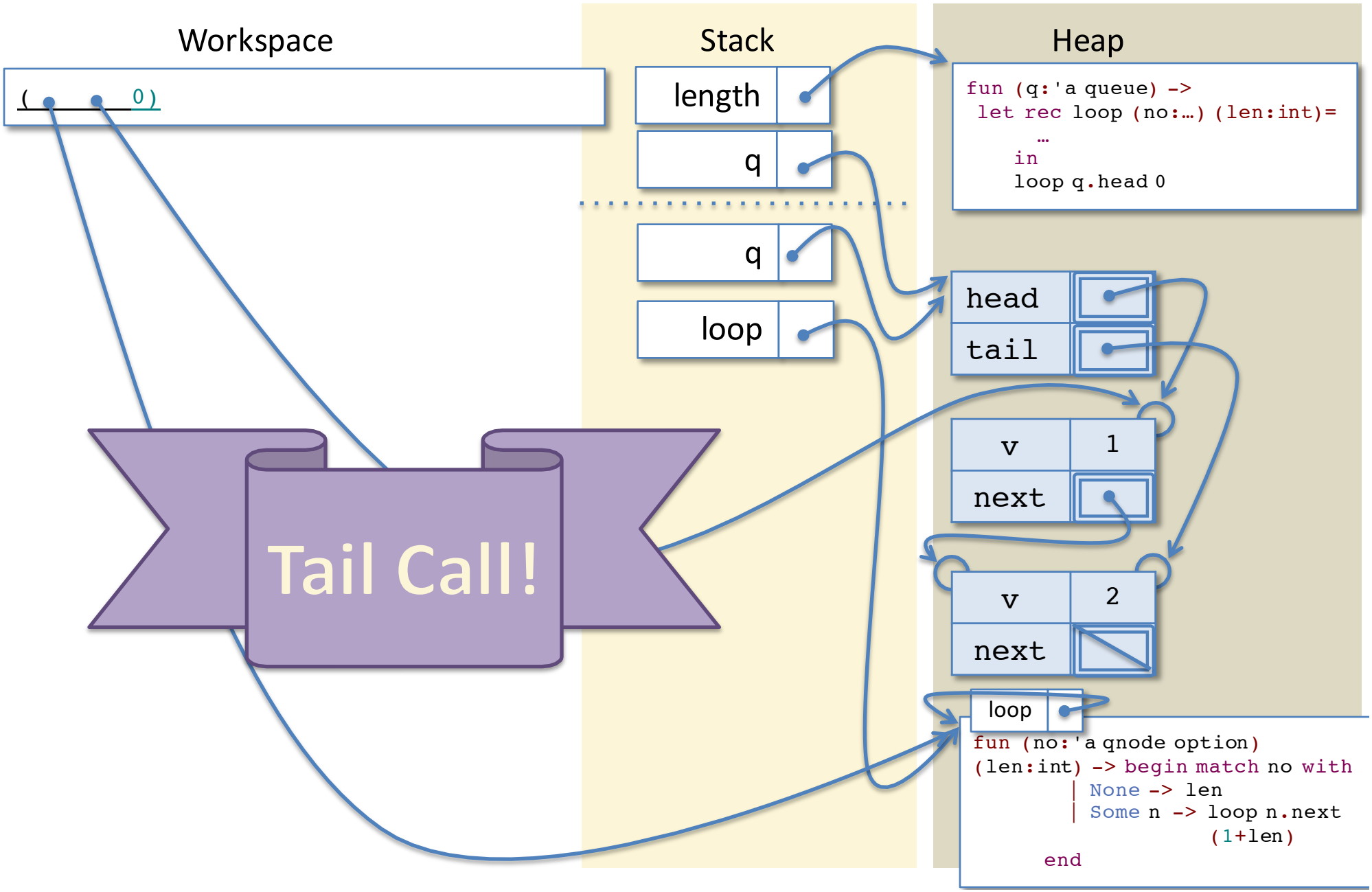
The *closure* of the local recursive function `loop` includes a binding for the loop function itself!

Why? The loop body mentions the loop identifier.

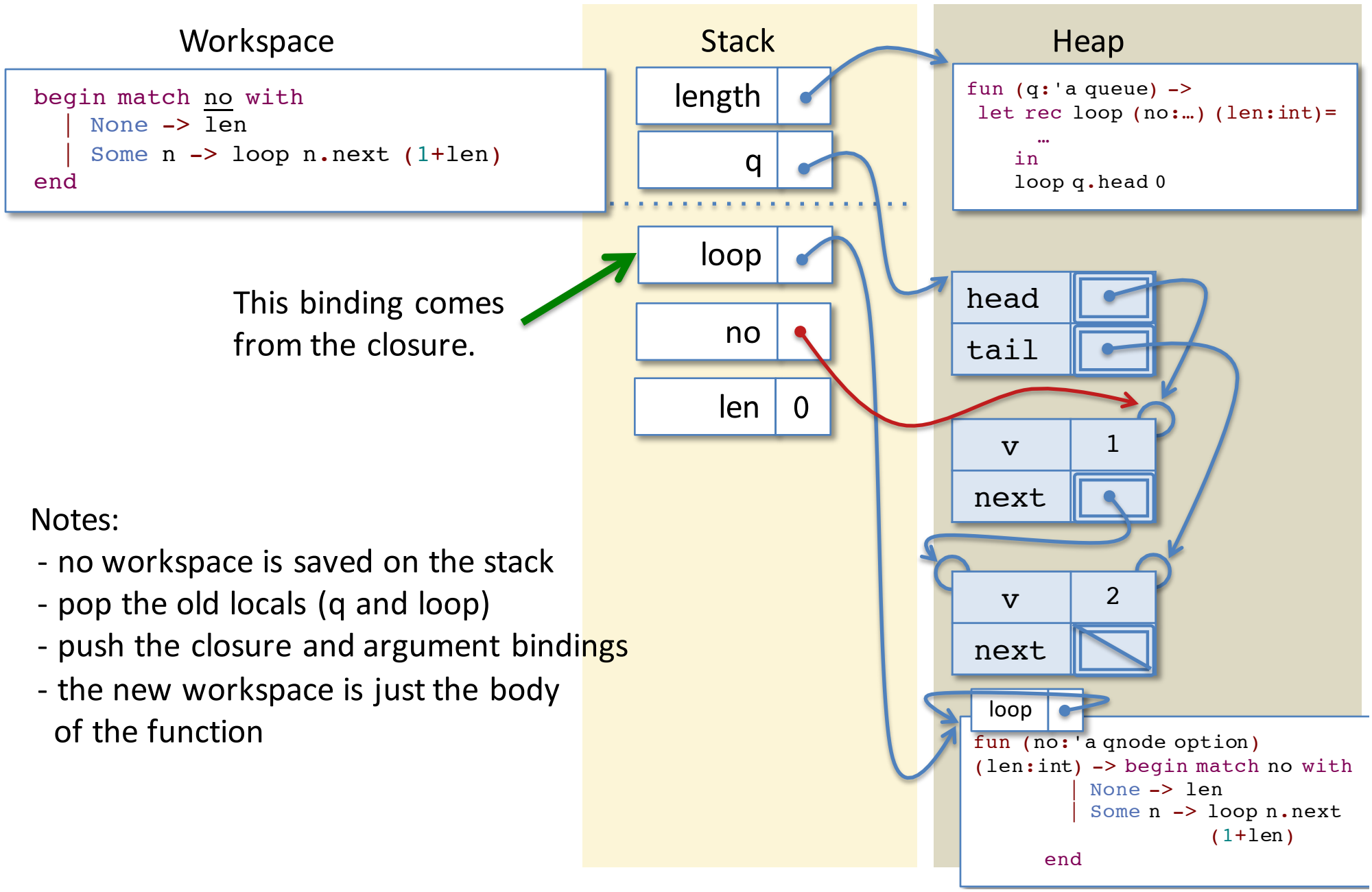
Tail Calls and Iterative length



Tail Calls and Iterative length



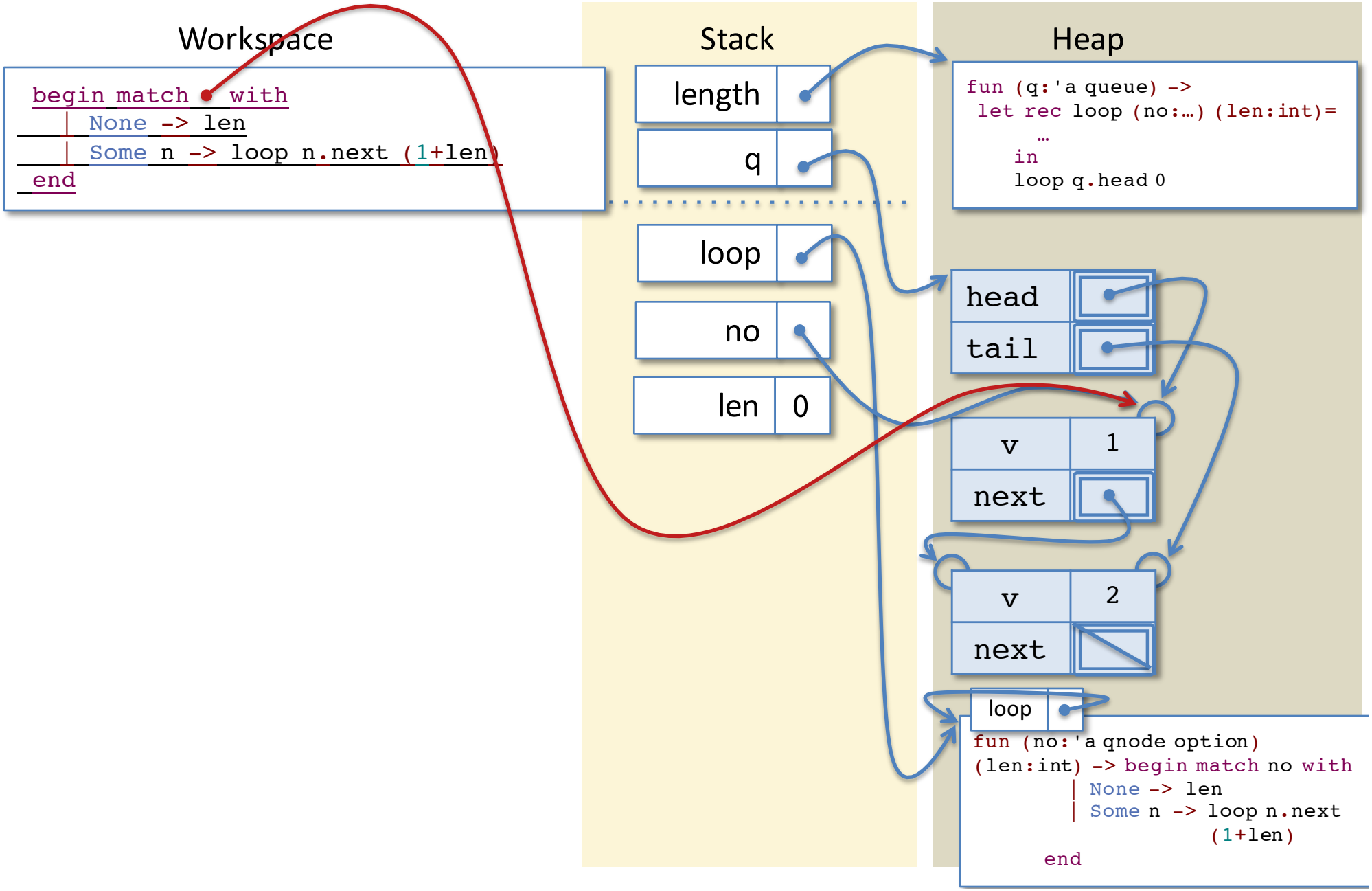
Tail Calls and Iterative length



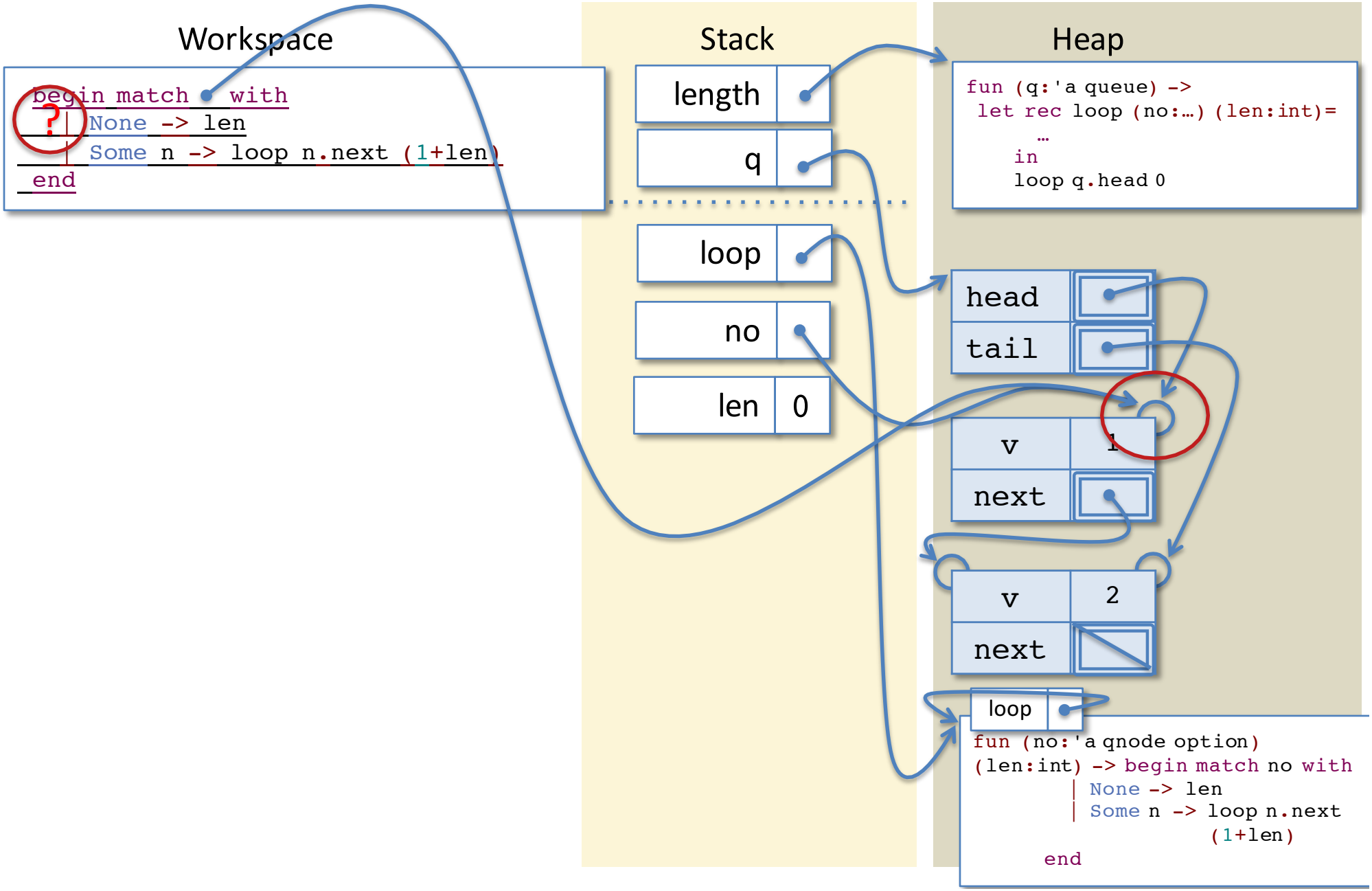
Notes:

- no workspace is saved on the stack
- pop the old locals (q and loop)
- push the closure and argument bindings
- the new workspace is just the body of the function

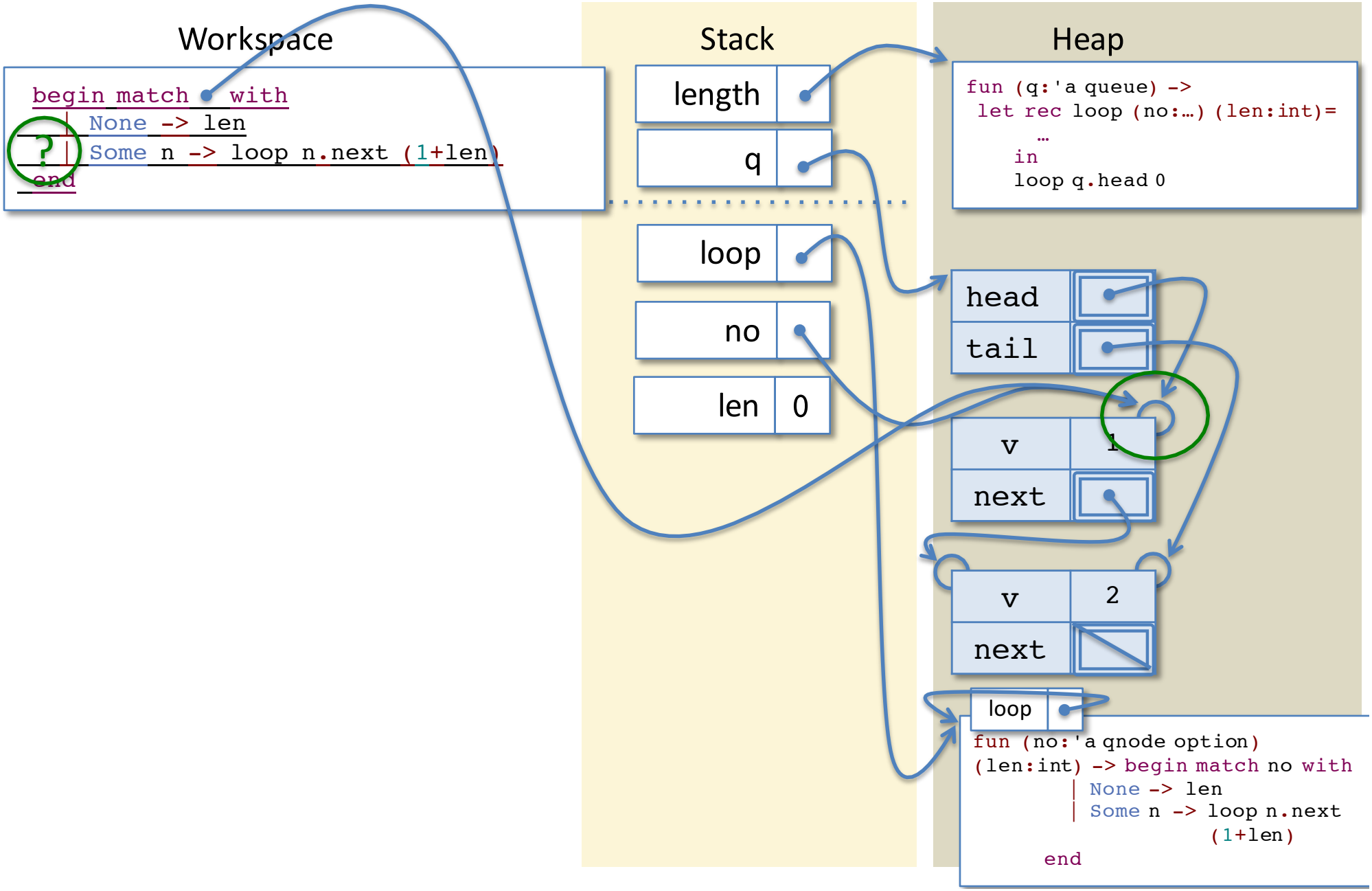
Tail Calls and Iterative length



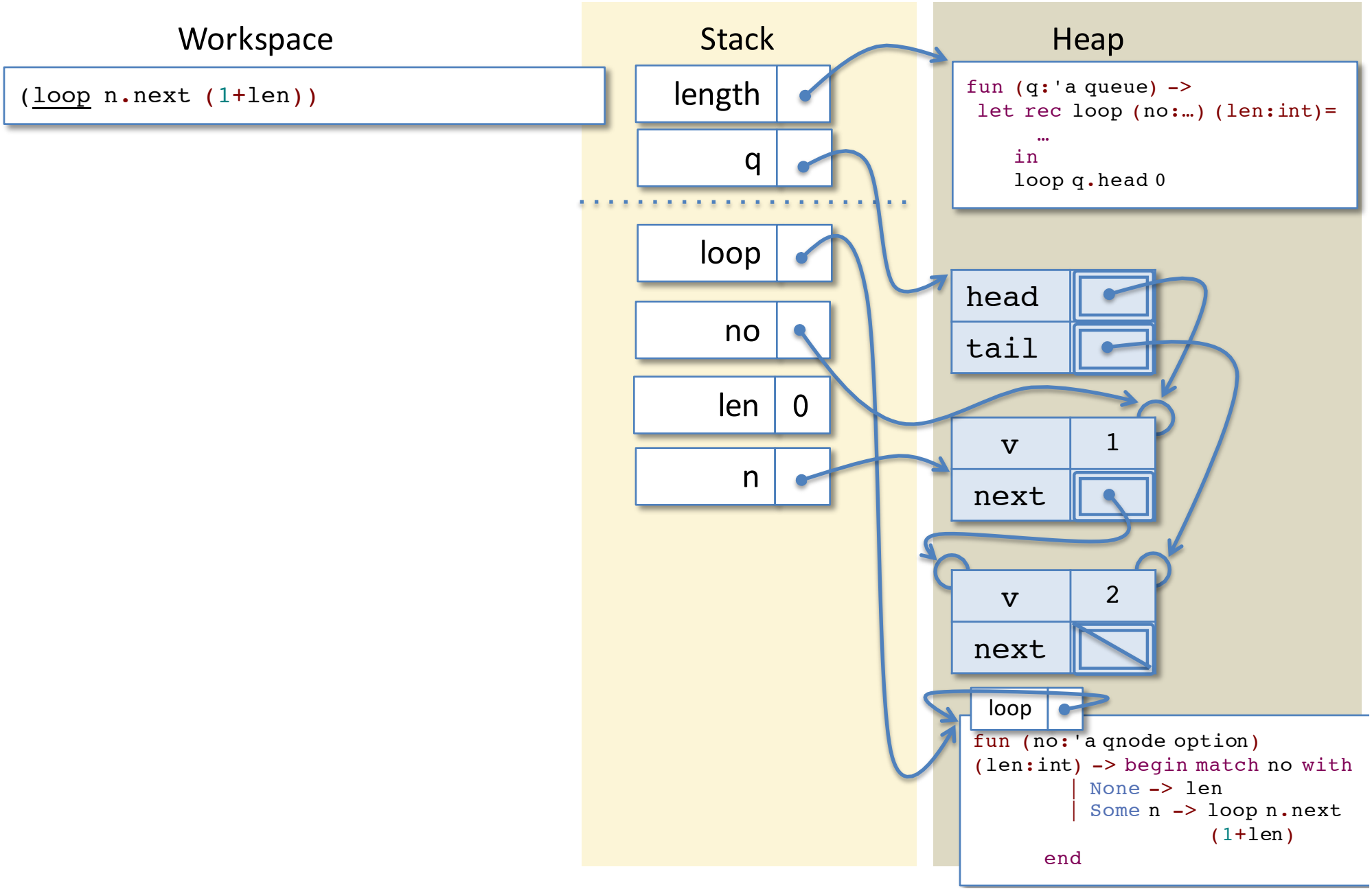
Tail Calls and Iterative length



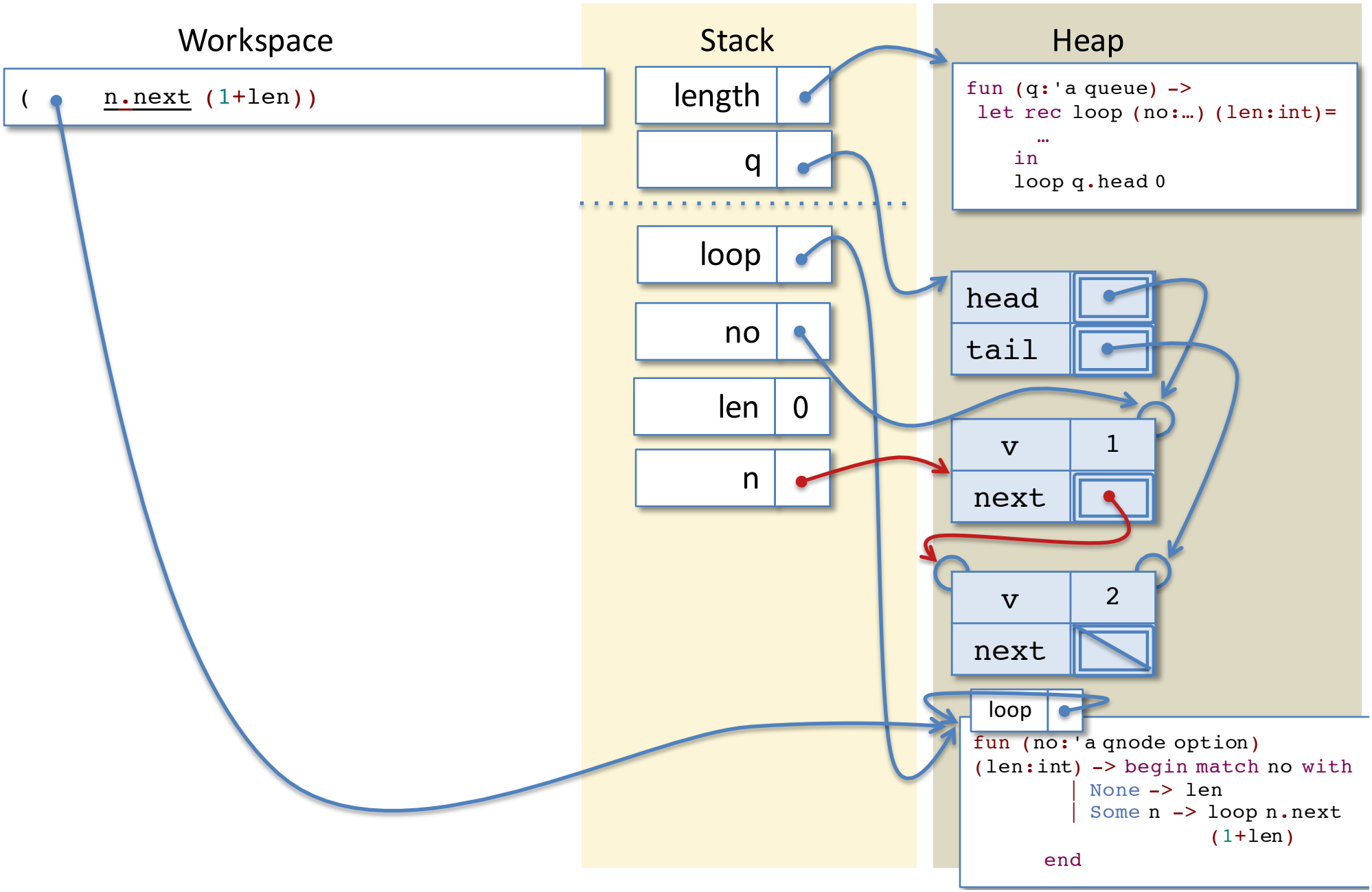
Tail Calls and Iterative length



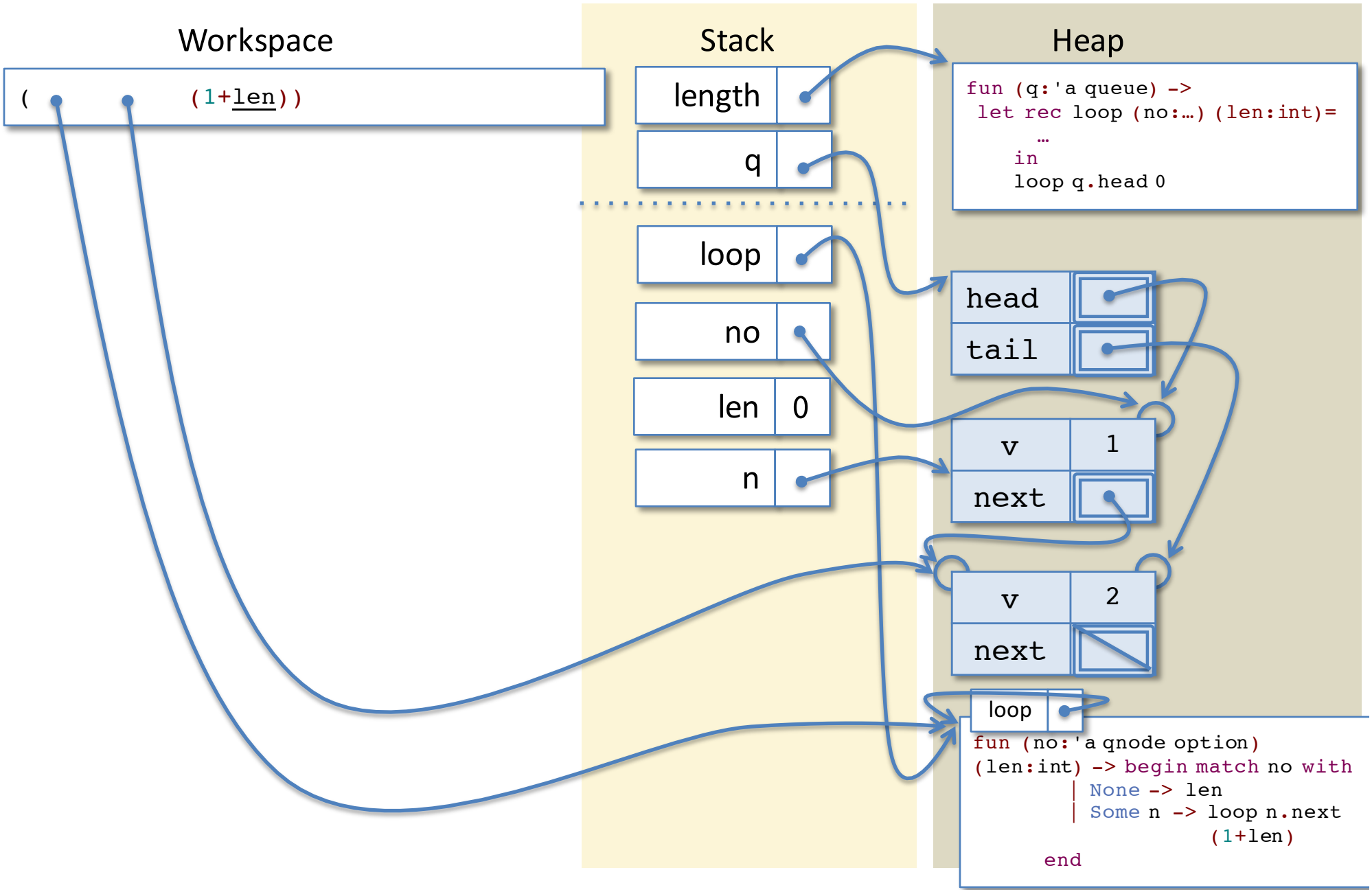
Tail Calls and Iterative length



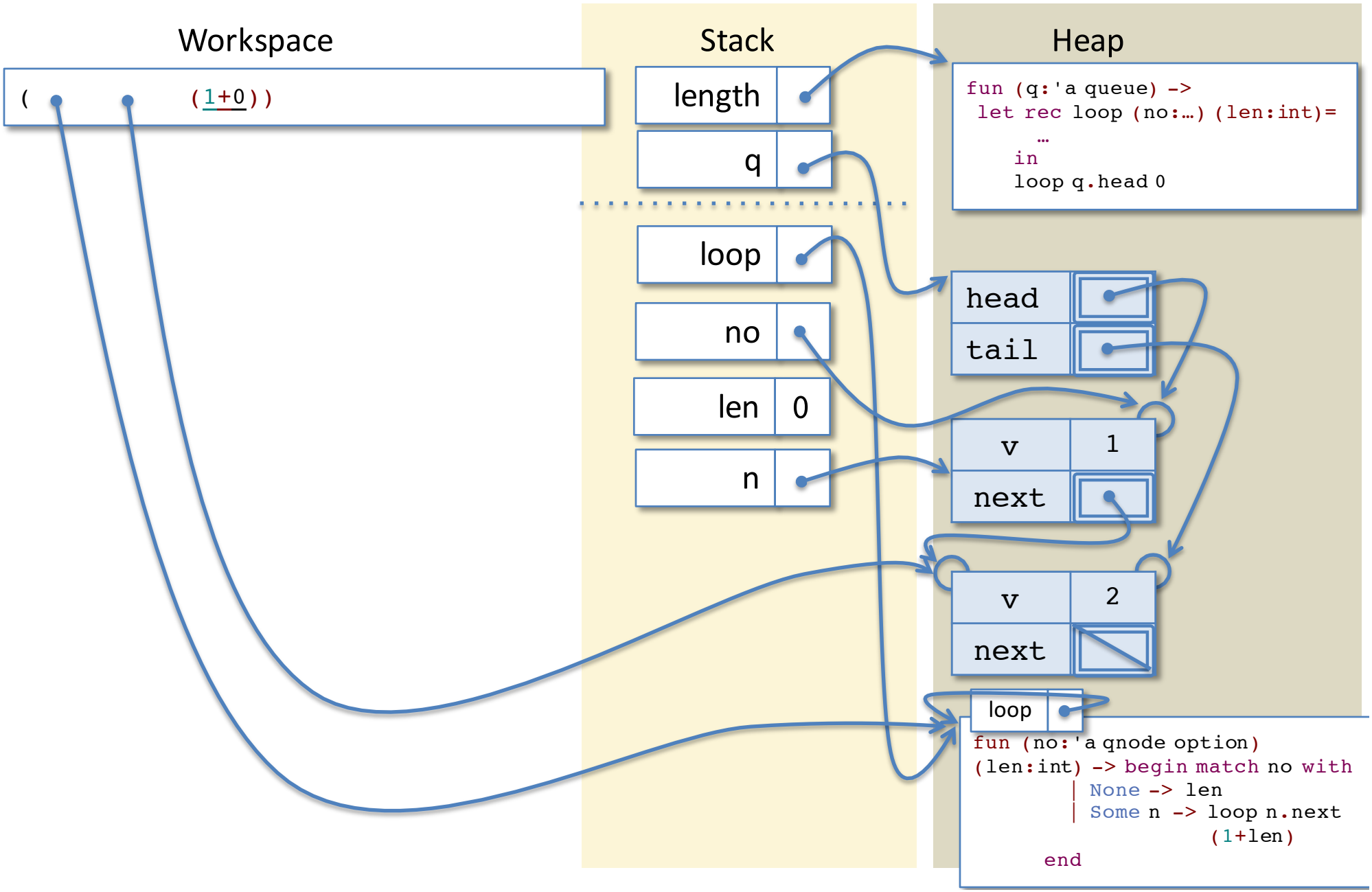
Tail Calls and Iterative length



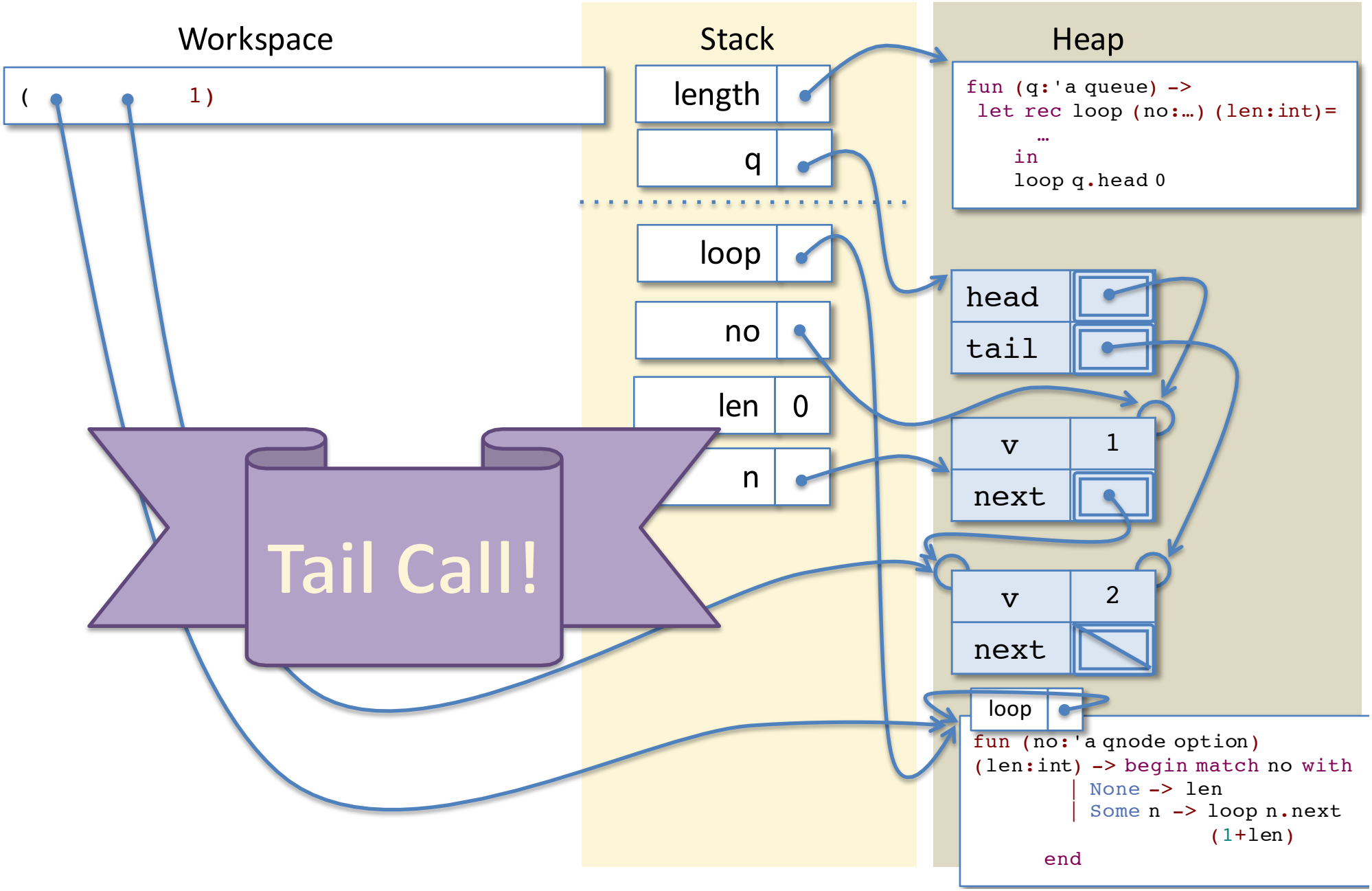
Tail Calls and Iterative length



Tail Calls and Iterative length



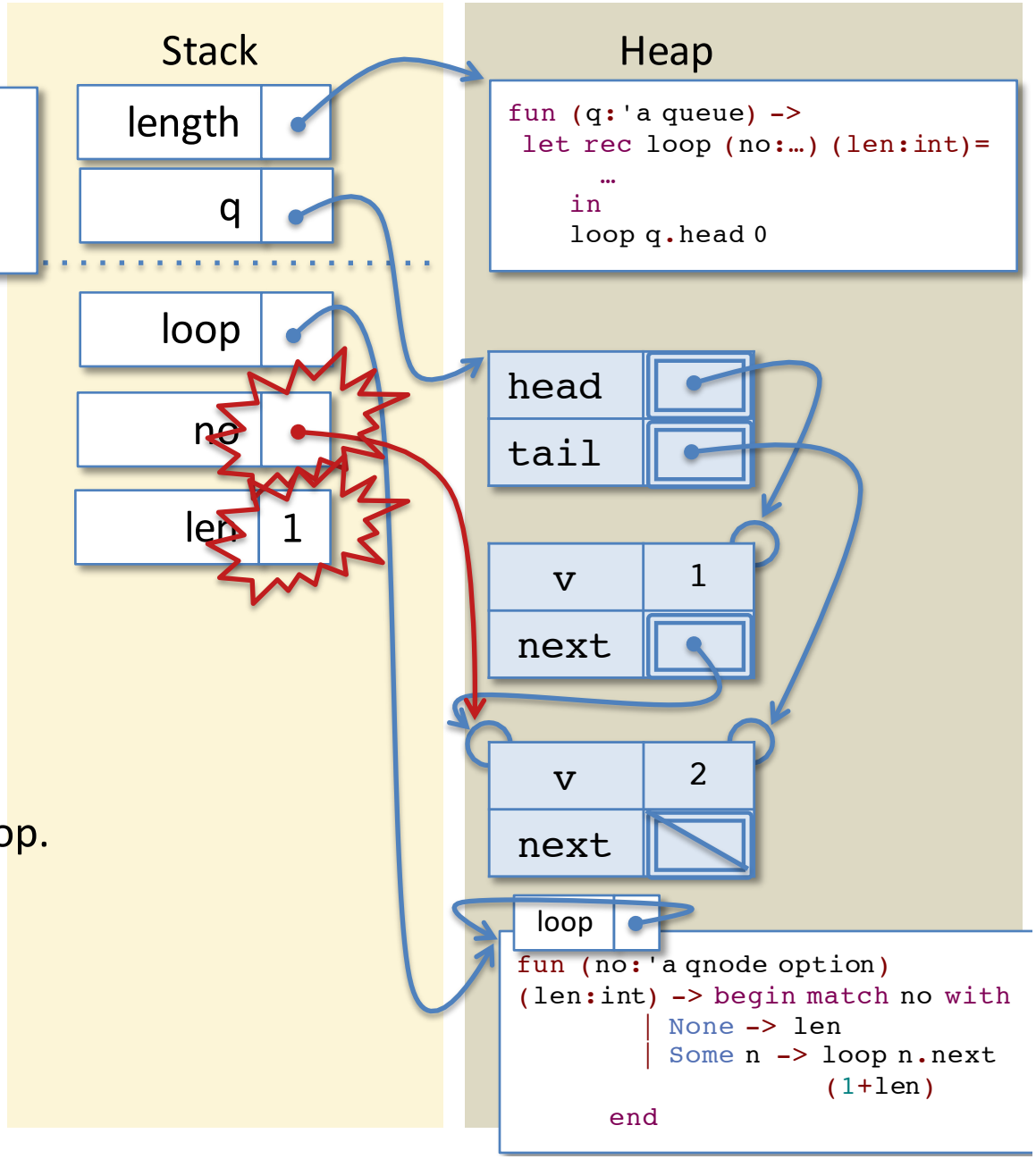
Tail Calls and Iterative length



Tail Calls and Iterative length

Workspace

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

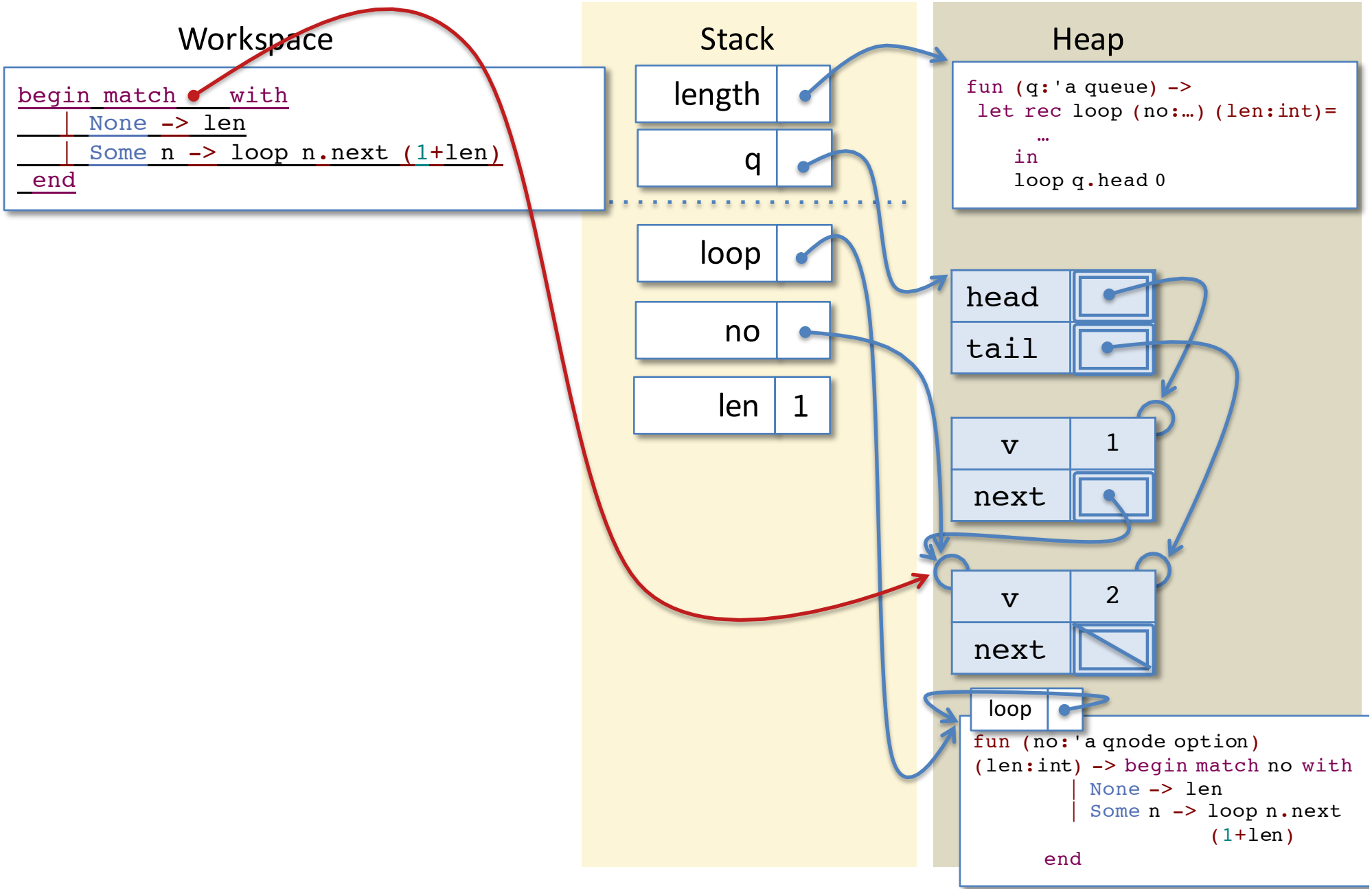


Note: we popped the old values of loop, no, len, and n when we did the tail call. Then we pushed the new values of loop, no, and len.

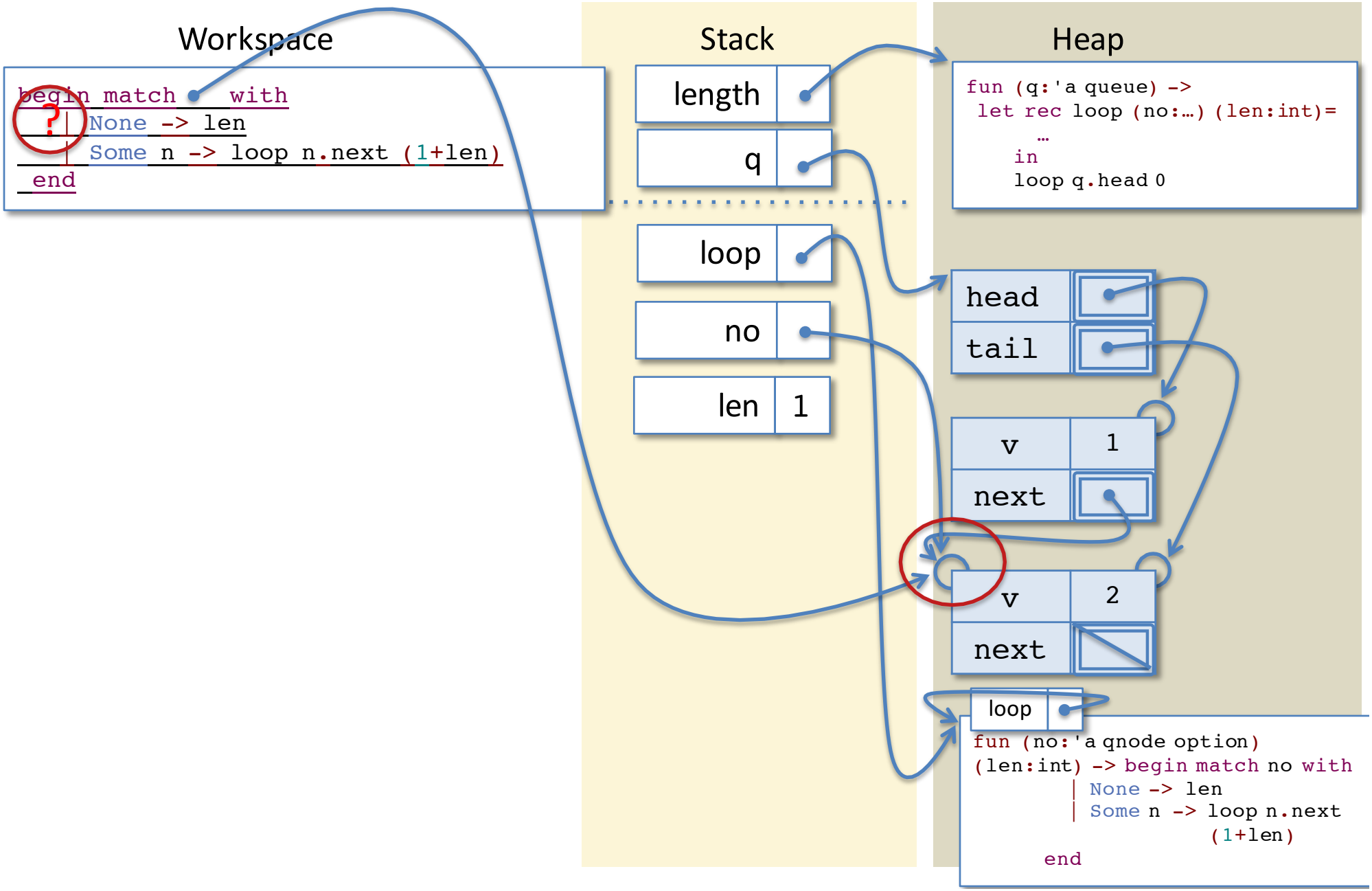
This leaves the stack in almost the same shape as when we first called loop.

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

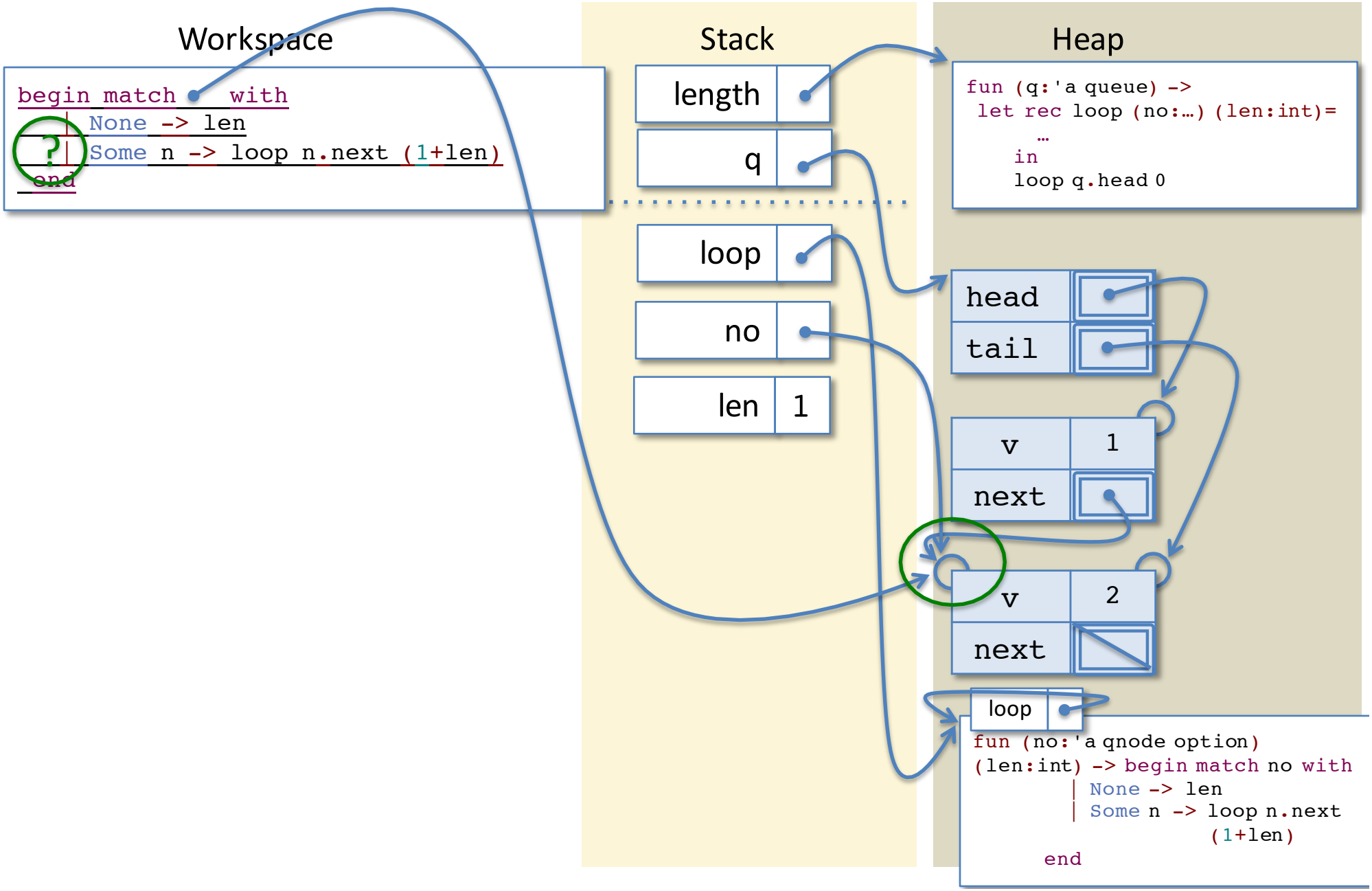
Tail Calls and Iterative length



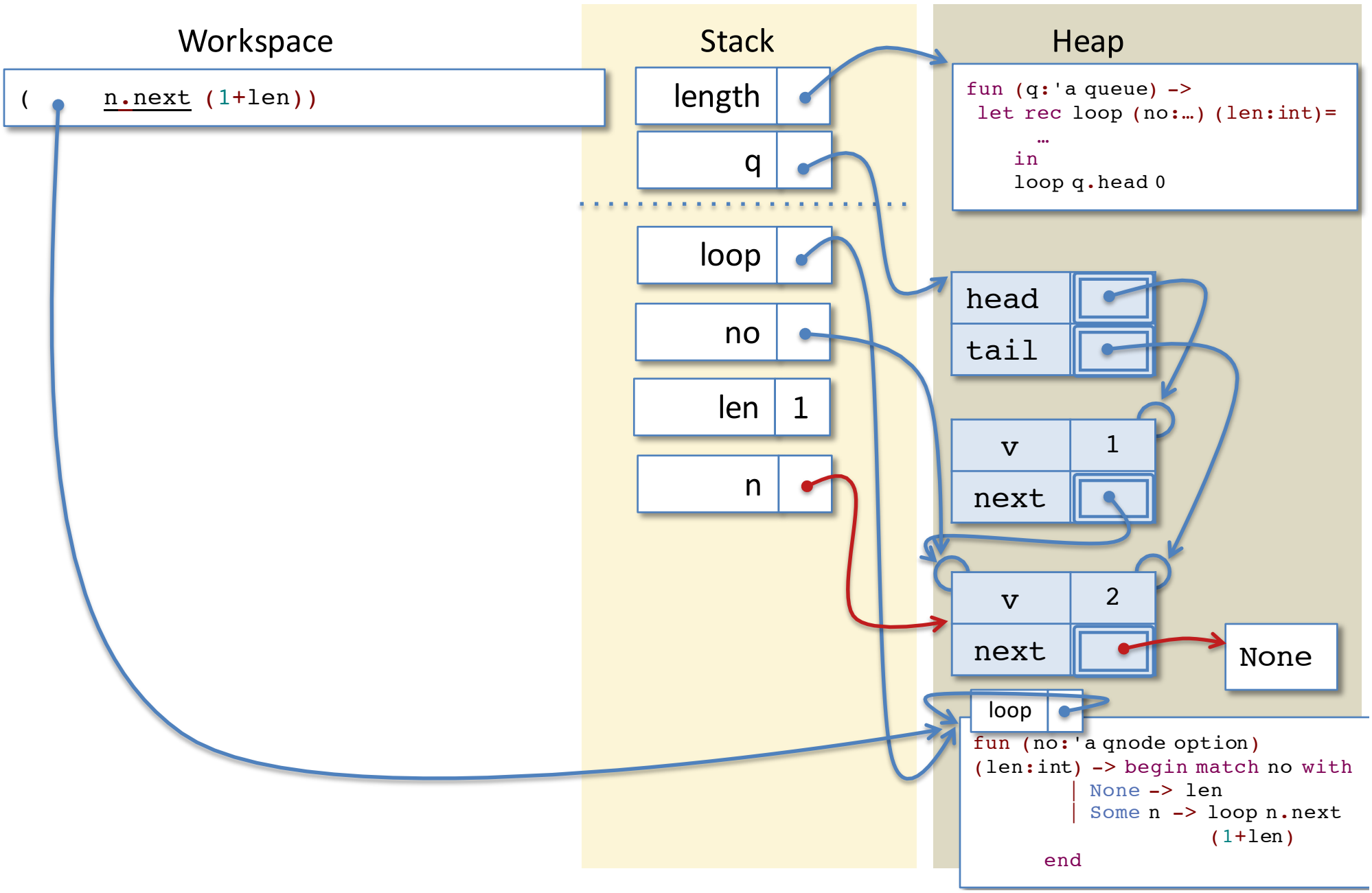
Tail Calls and Iterative length



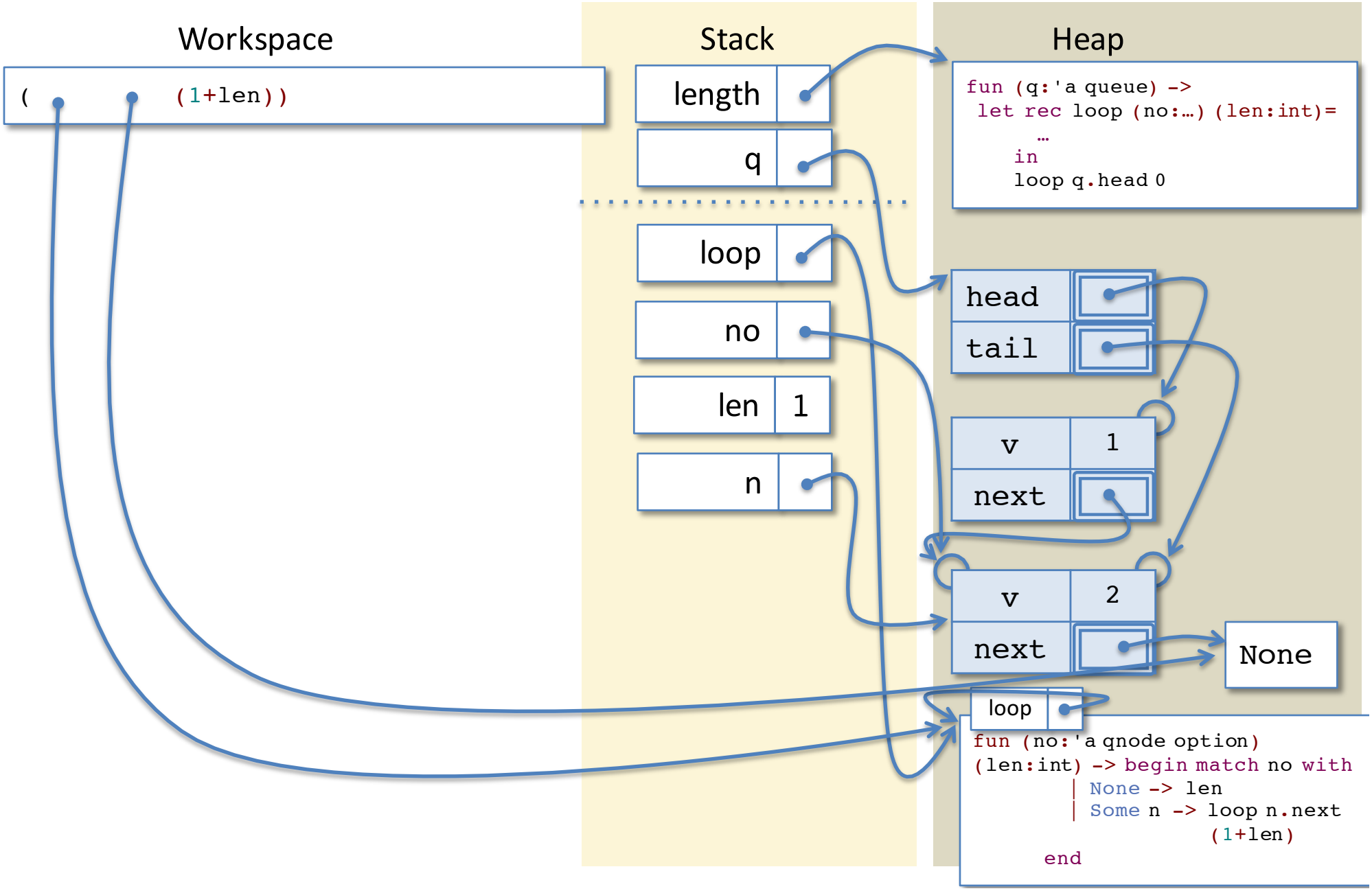
Tail Calls and Iterative length



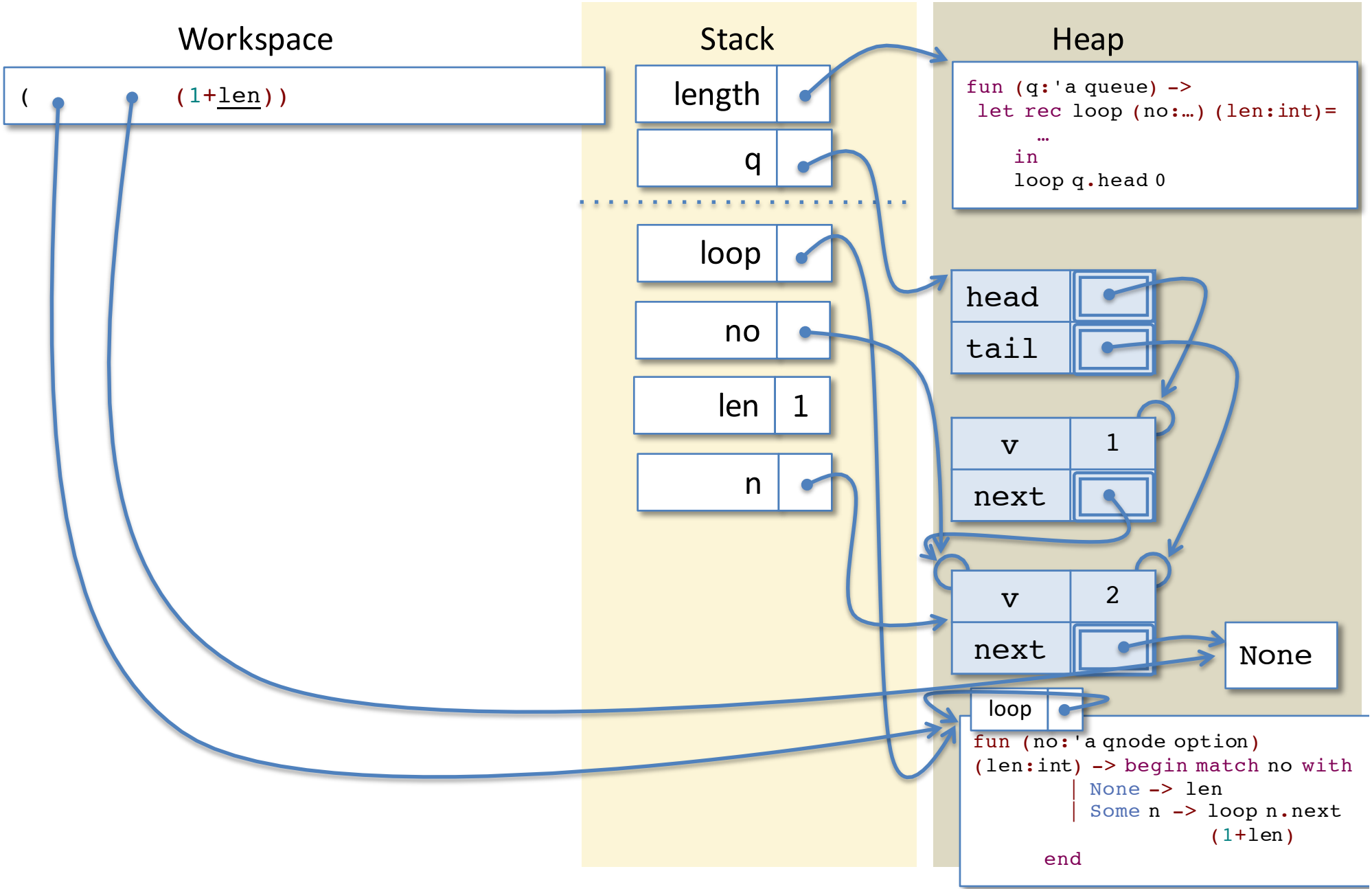
Tail Calls and Iterative length



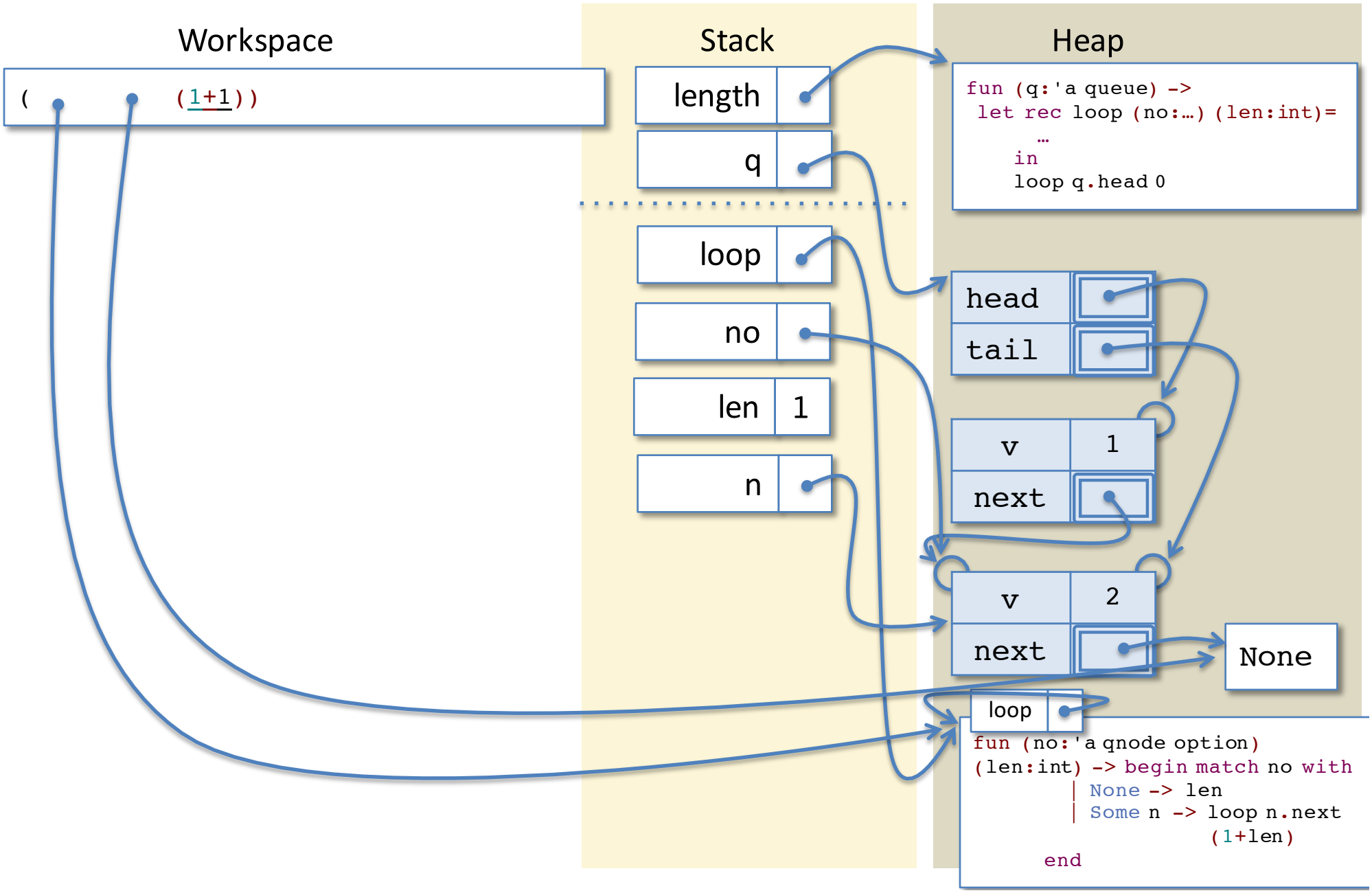
Tail Calls and Iterative length



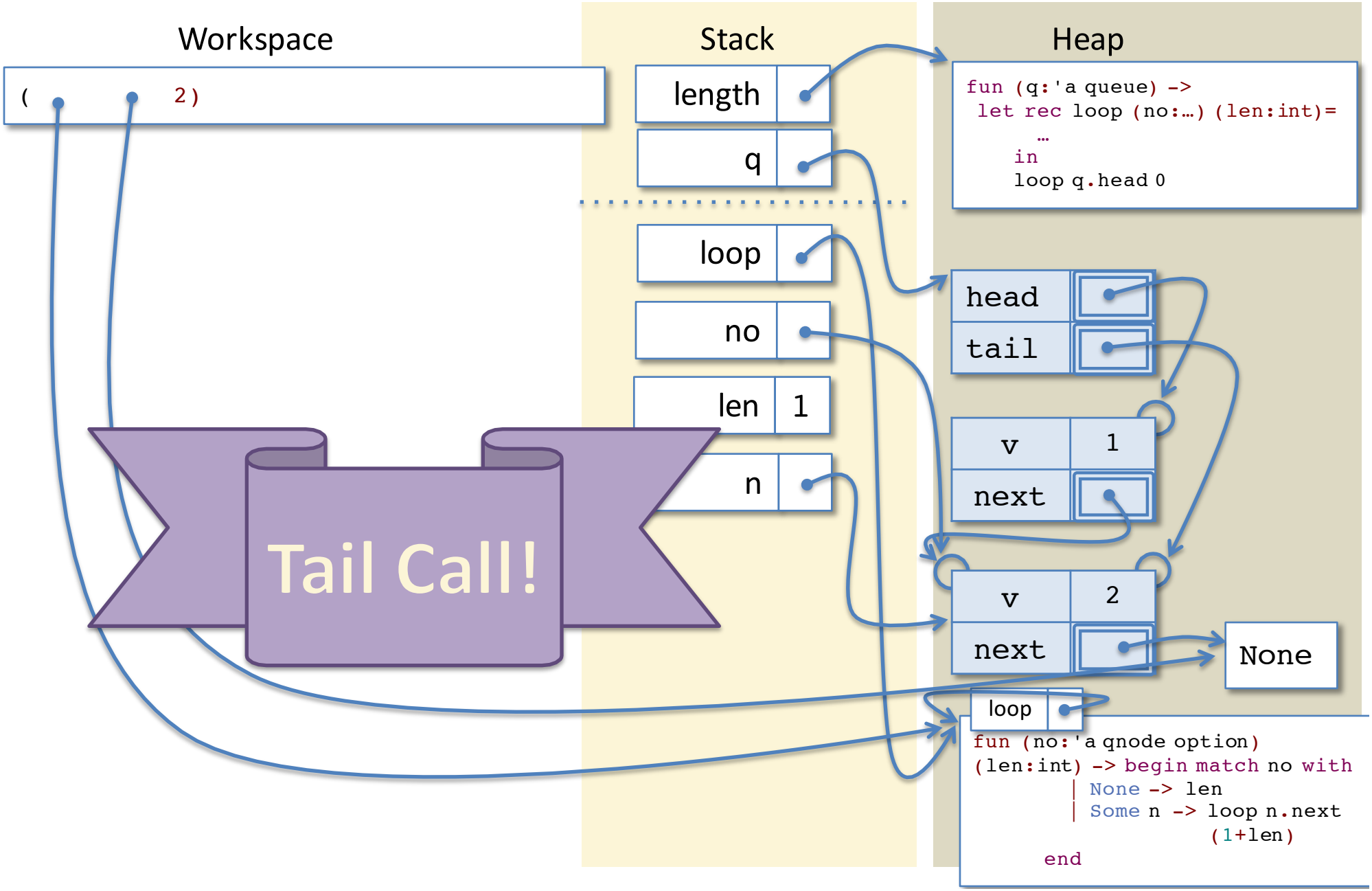
Tail Calls and Iterative length



Tail Calls and Iterative length



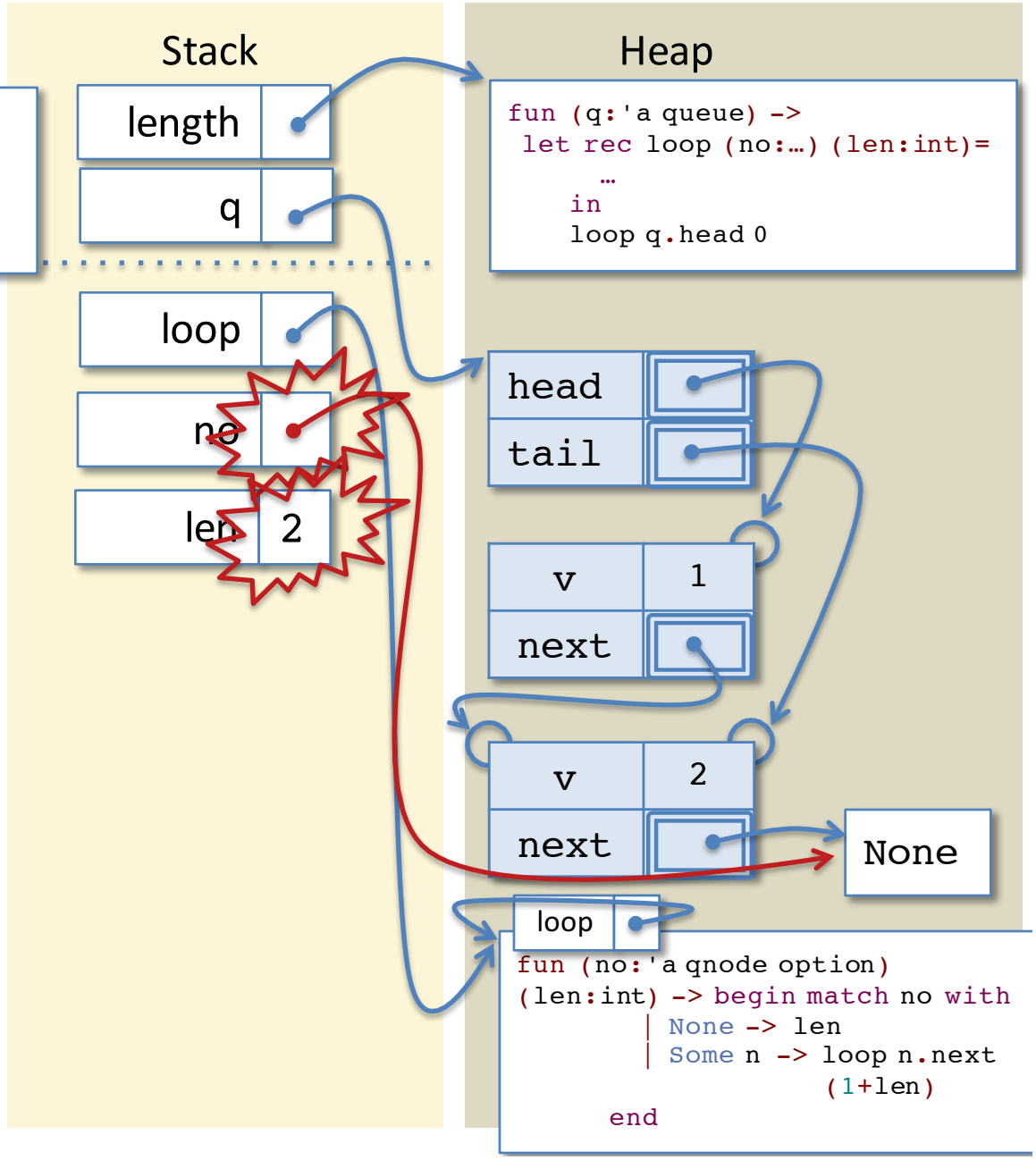
Tail Calls and Iterative length



Tail Calls and Iterative length

Workspace

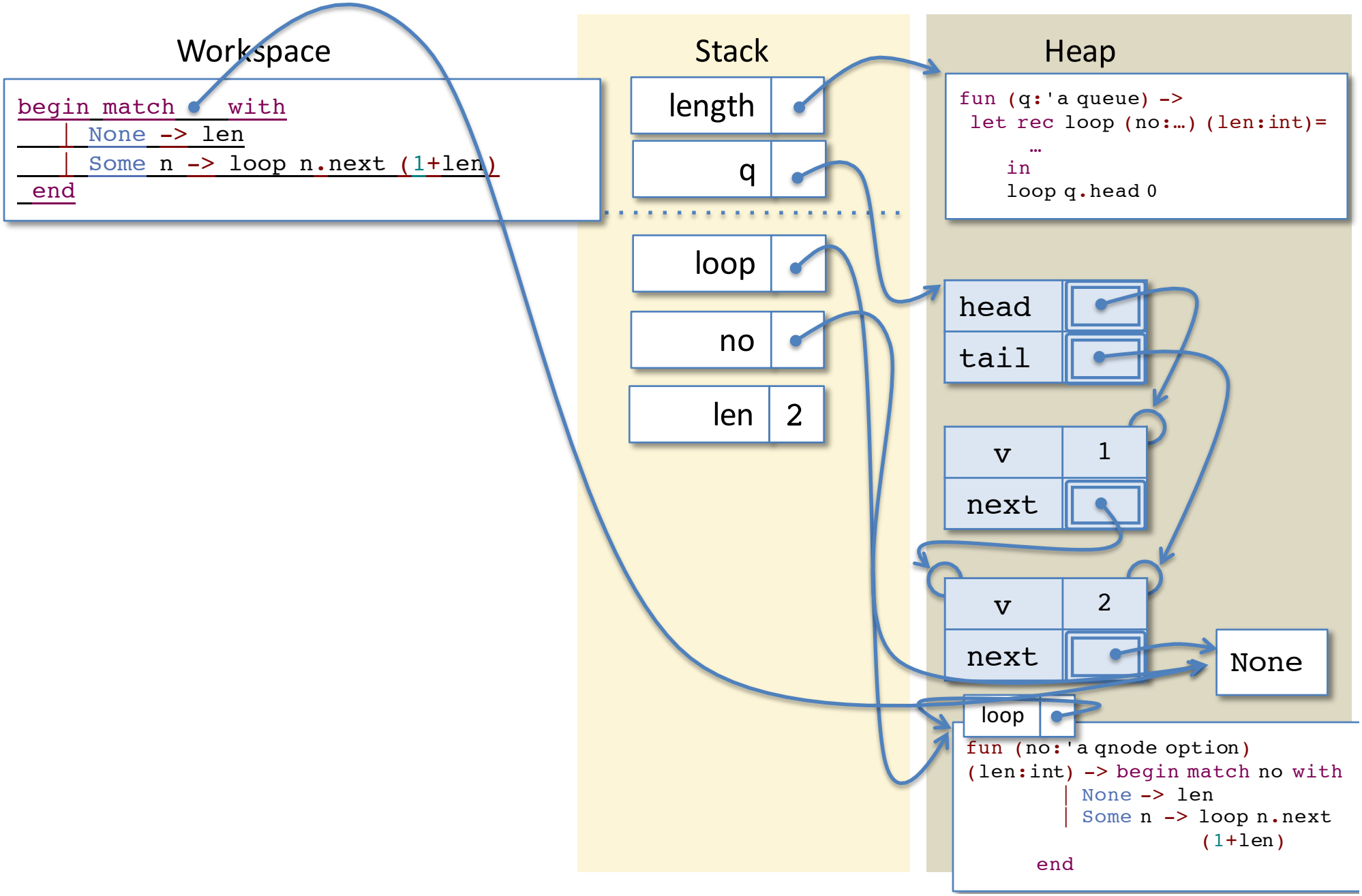
```
begin match no with
    | None -> len
    | Some n -> loop n.next (1+len)
 end
```



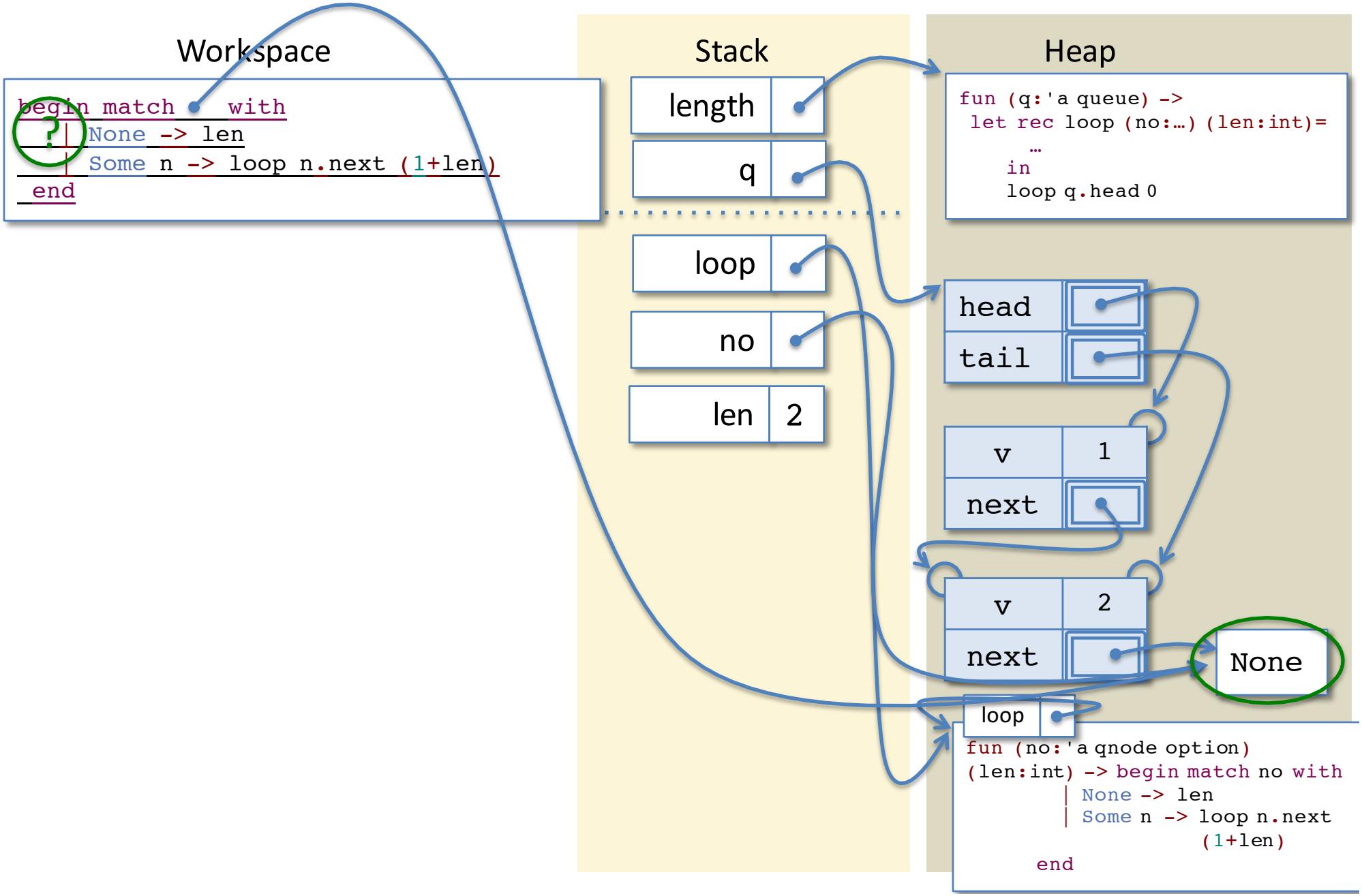
Note: Again, the tail call leaves the stack as before, but effectively updates the values of no and len.

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

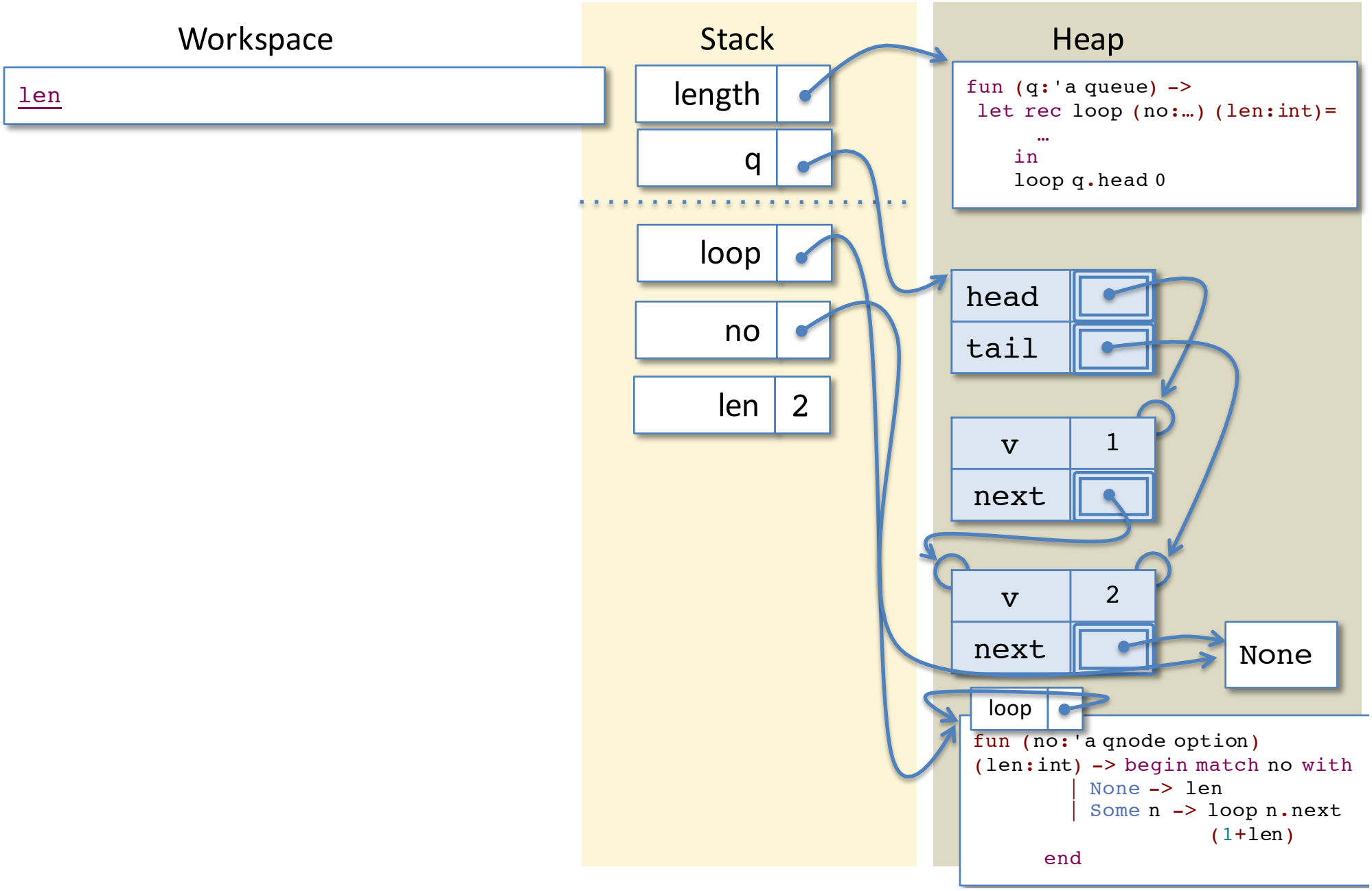
Tail Calls and Iterative length



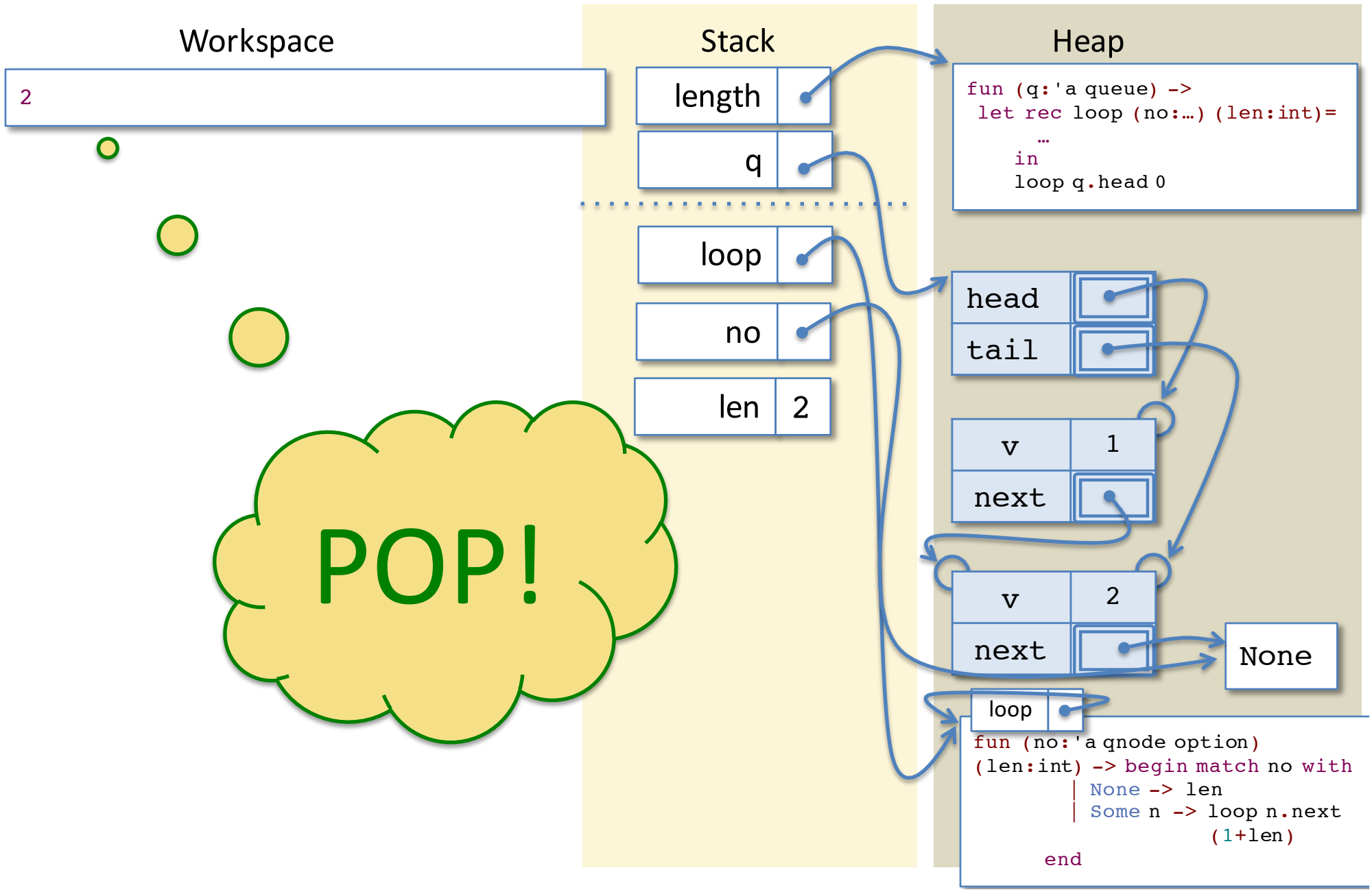
Tail Calls and Iterative length



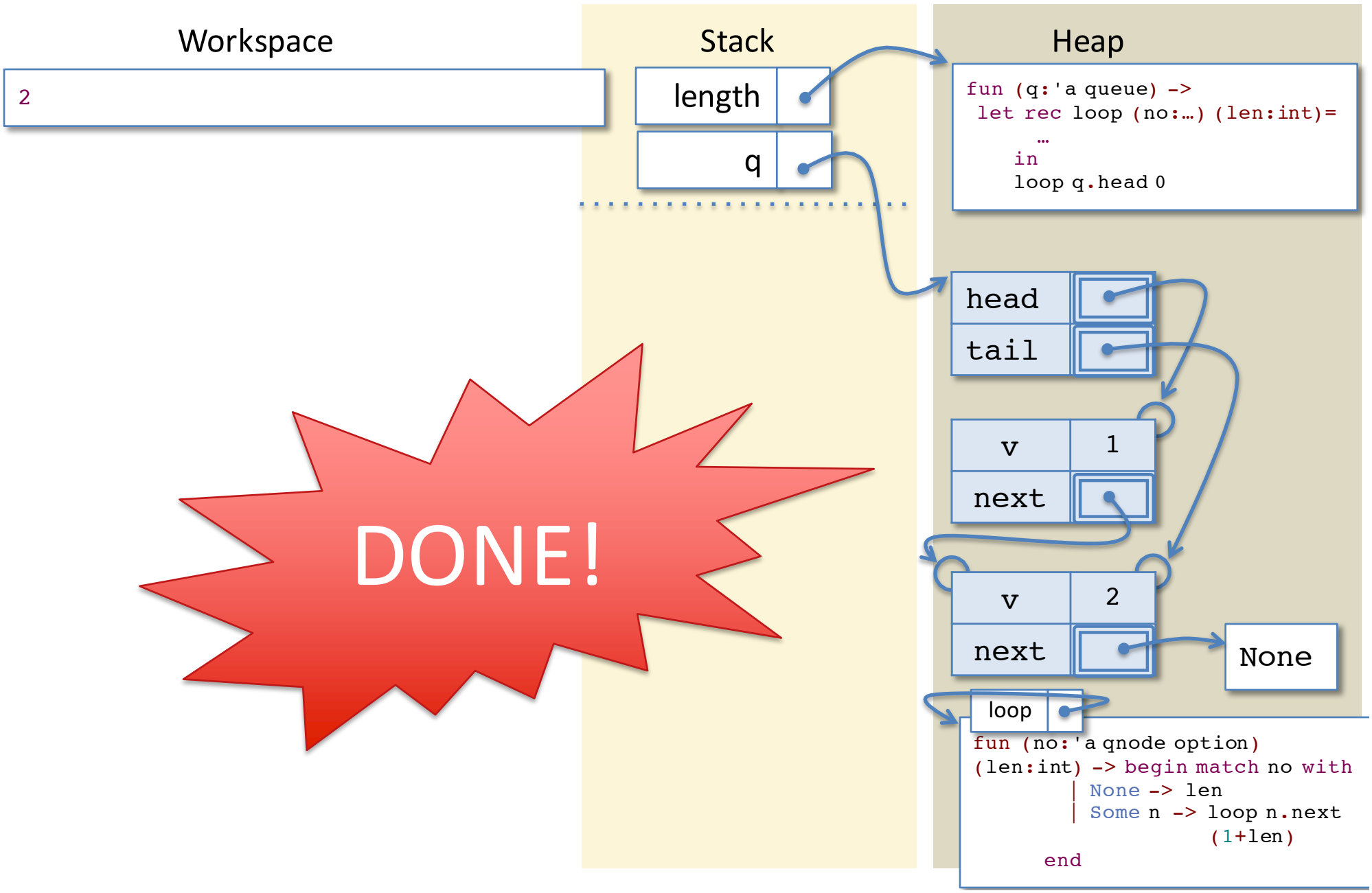
Tail Calls and Iterative length



Tail Calls and Iterative length



Tail Calls and Iterative length



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

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!