Programming Languages and Techniques (CIS120)

Lecture 16

Iteration & Tail Recursion

Chapter 16

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 }
```

- Mutable Queue *invariant*

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`

- Each queue operation may *assume* that the invariant holds of its inputs and must *ensure* that the invariant holds when it's done.

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.

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, then we must “patch up” the “next” link of the old tail node to maintain the queue invariant.

Challenge problem: discuss w/partner

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

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

```
(* BUGGY: delete element at head 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;
let _ = deq q in
2 = deq q
```

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

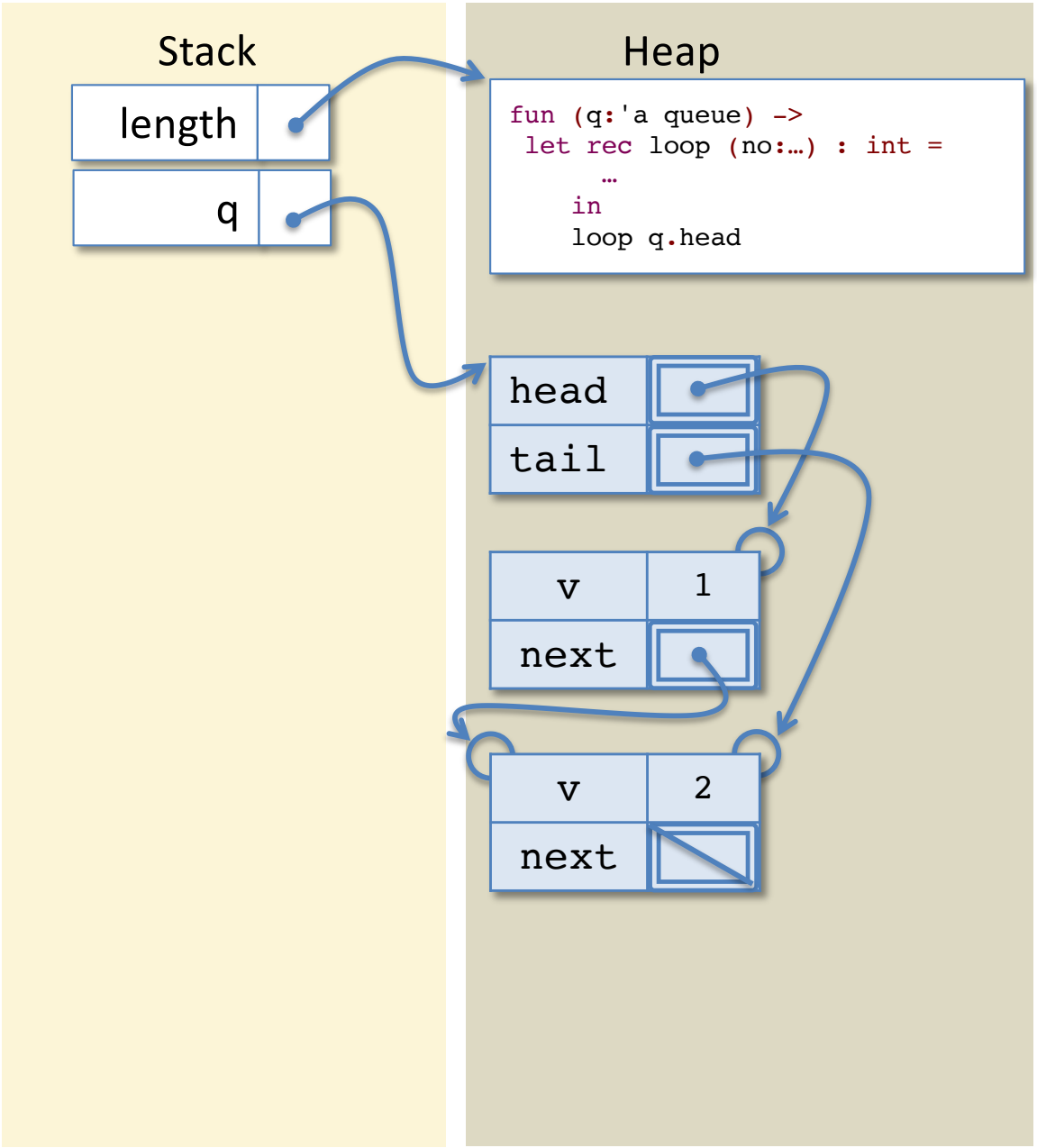
```
(* Calculate the length of the queue recursively *)
let length (q:'a queue) : int =
  let rec loop (no: 'a qnode option) : int =
    begin match no with
      | None -> 0
      | Some n -> 1 + (loop n.next)
    end
  in
  loop q.head
```

- 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

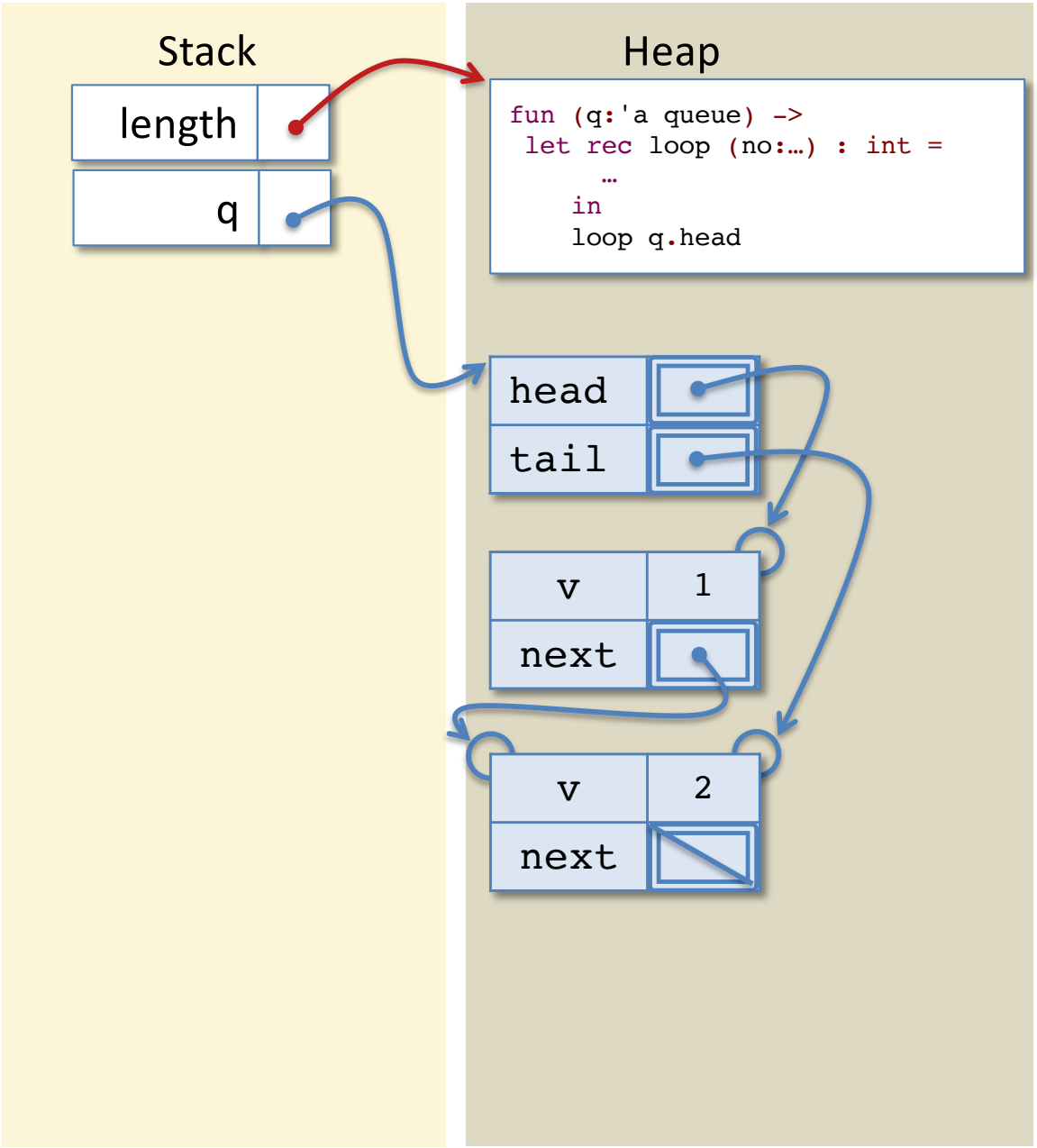
```
length q
```



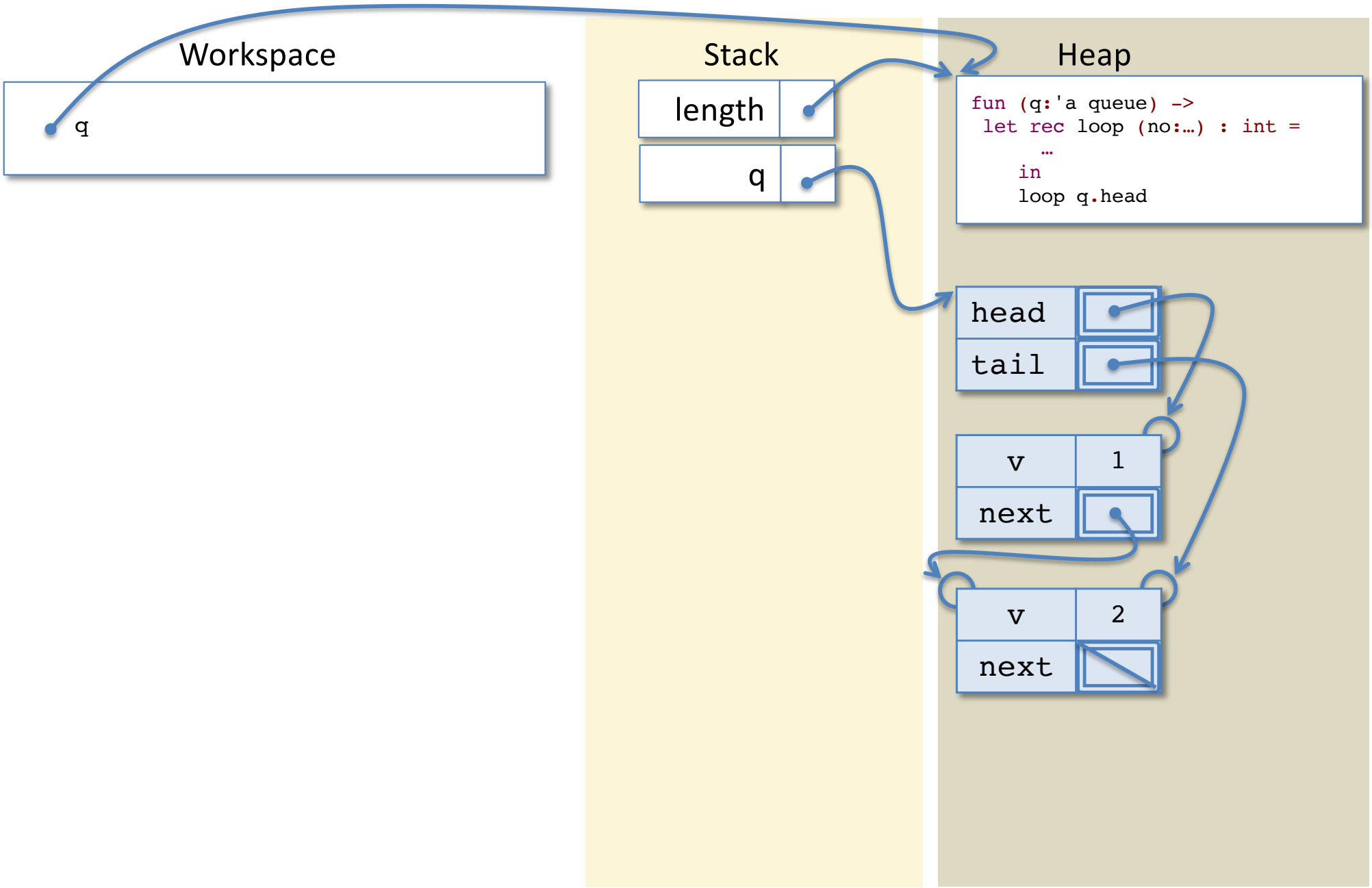
Evaluating length

Workspace

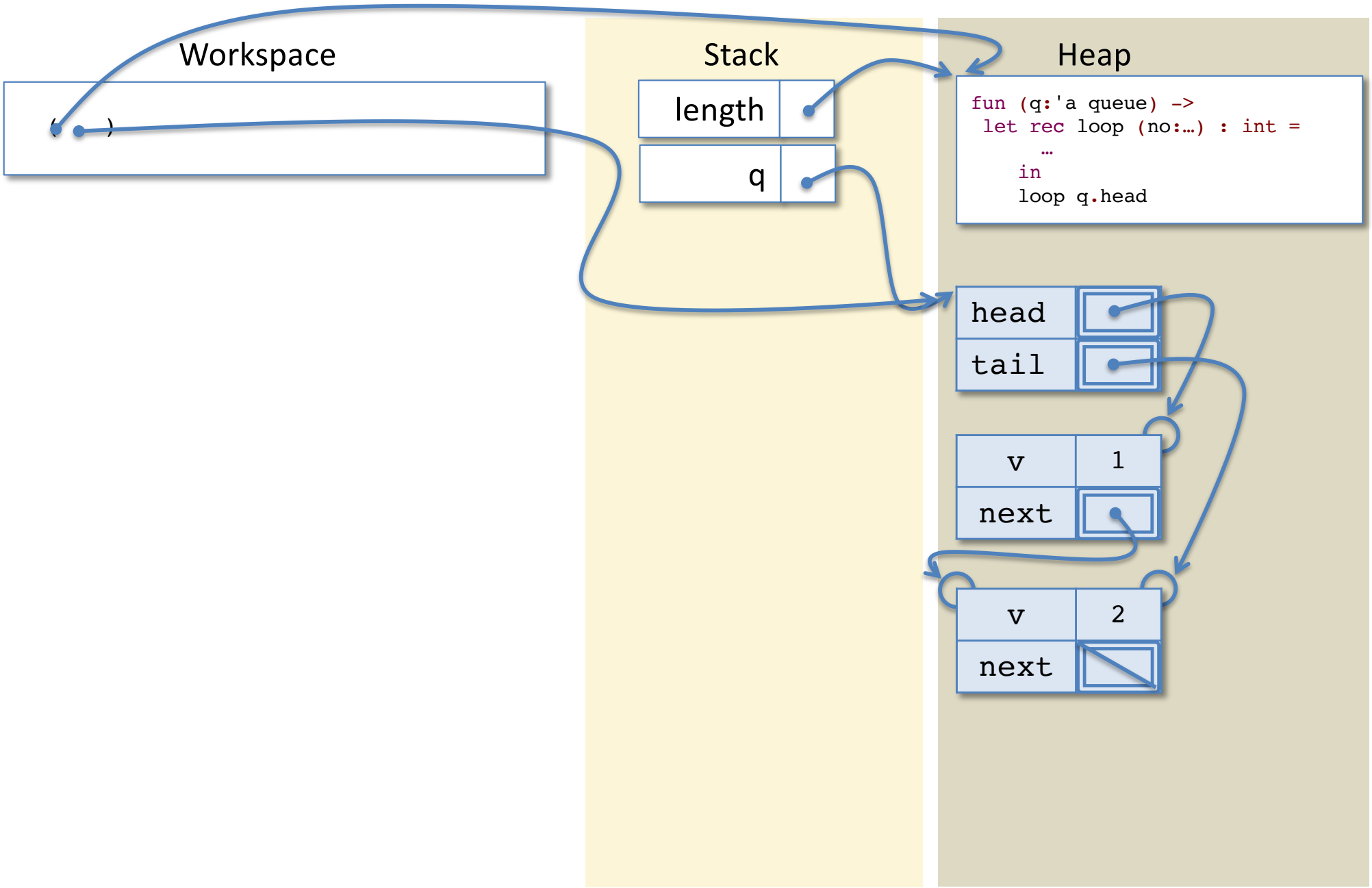
```
length q
```



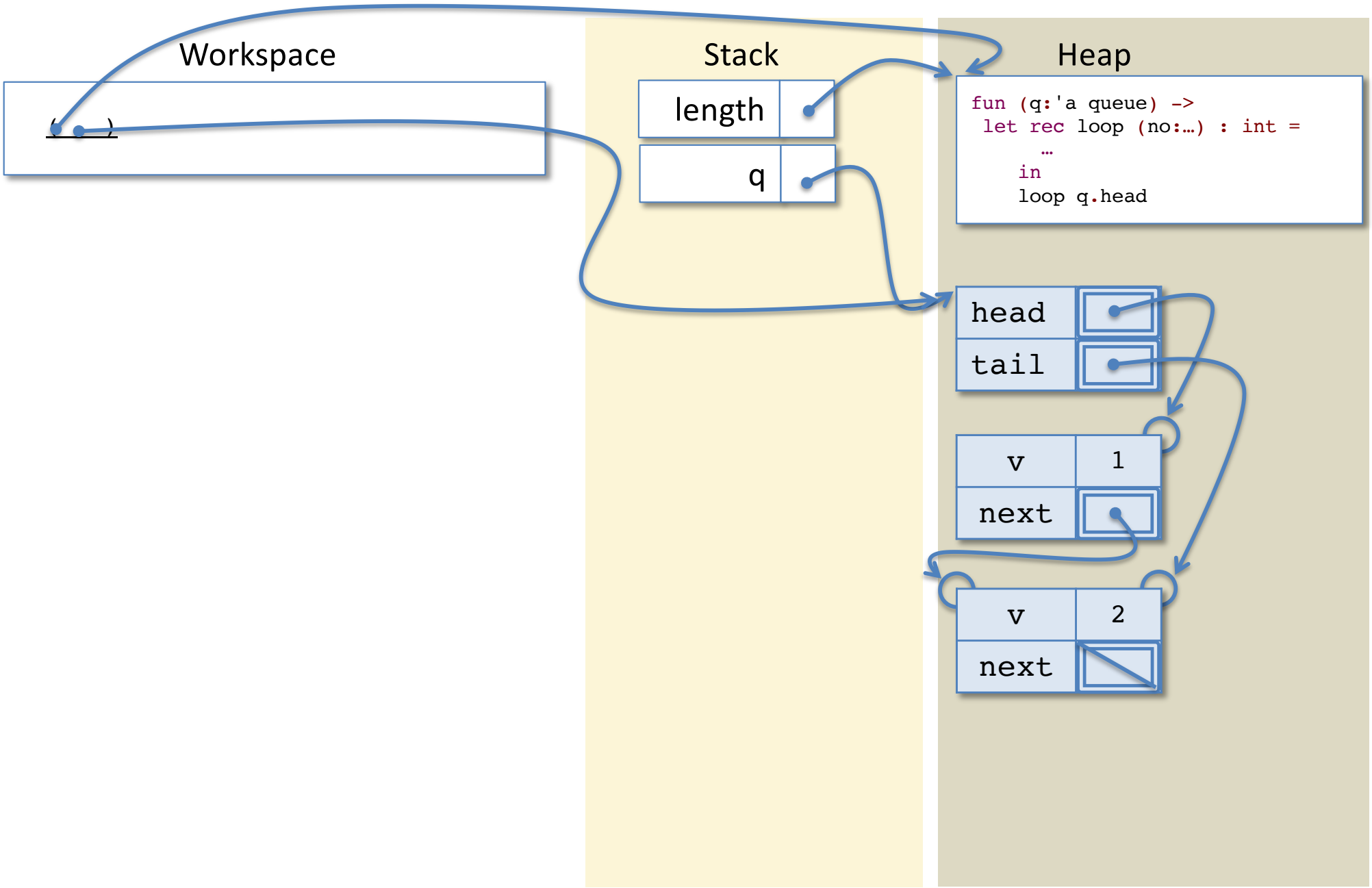
Evaluating length



Evaluating length



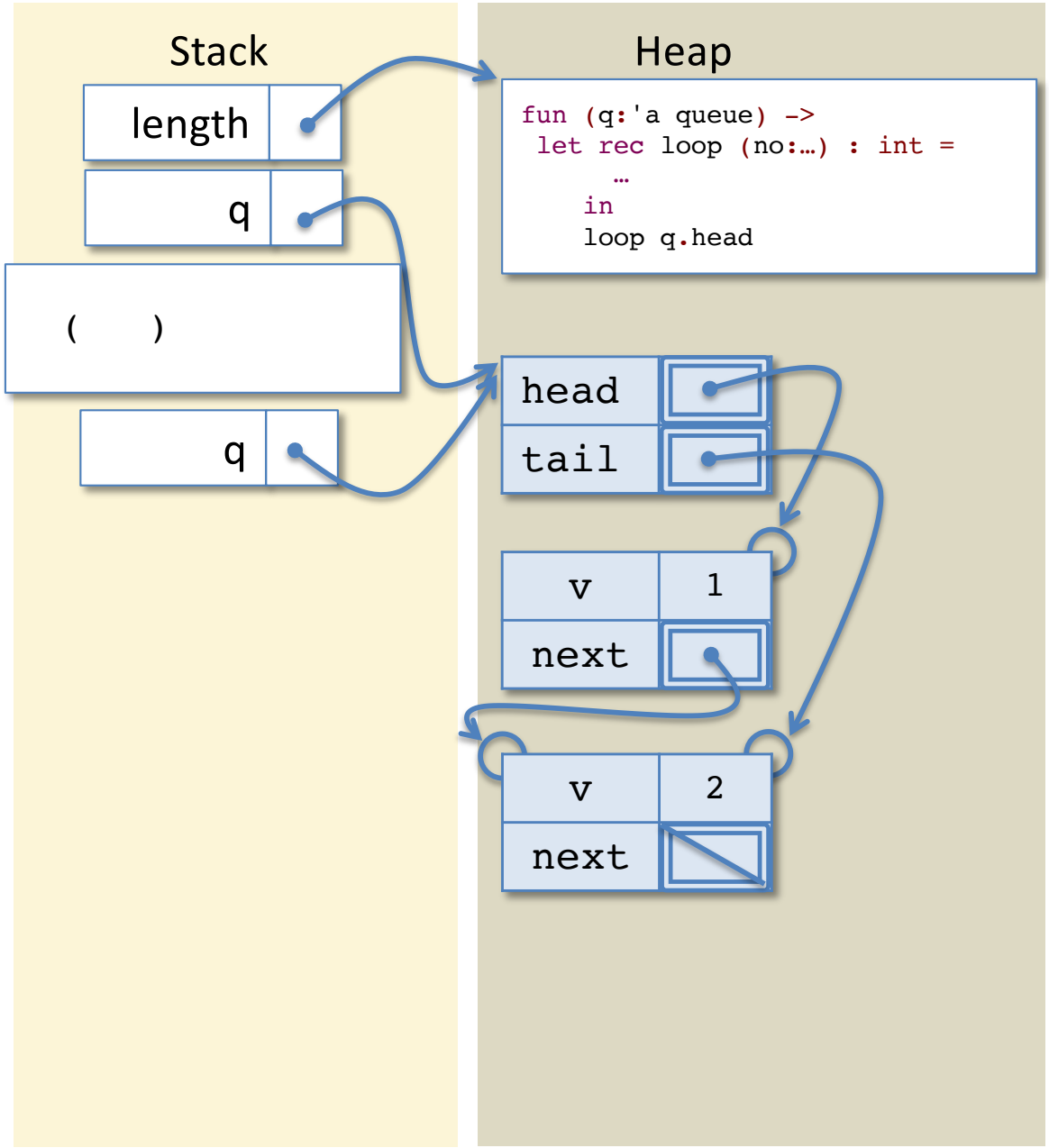
Evaluating length



Evaluating length

Workspace

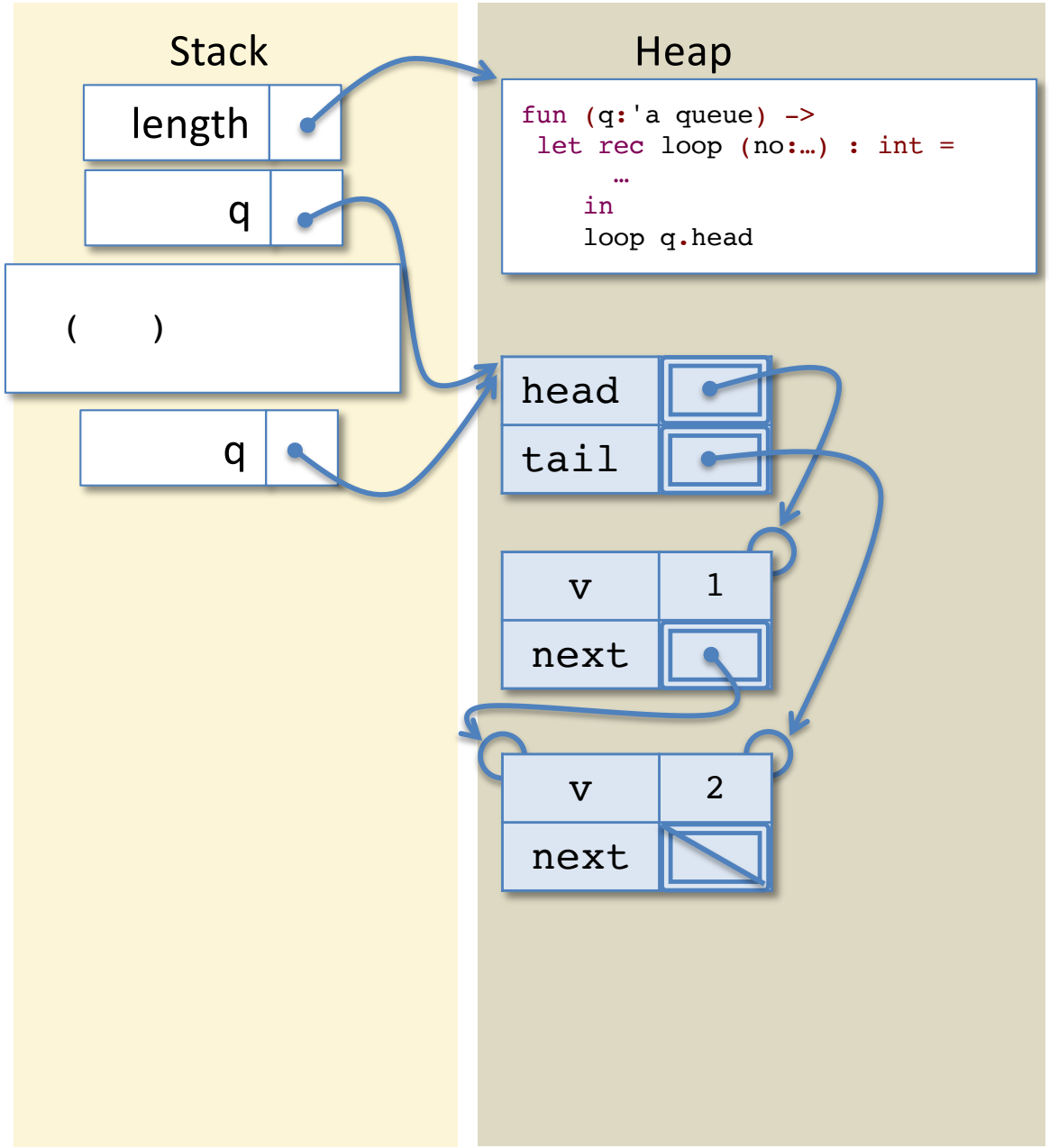
```
let rec loop (no: …) : int =
      begin match no with
         | None -> 0
         | Some n -> 1 + (loop n.next)
      end
    in
    loop q.head
```



Evaluating length

Workspace

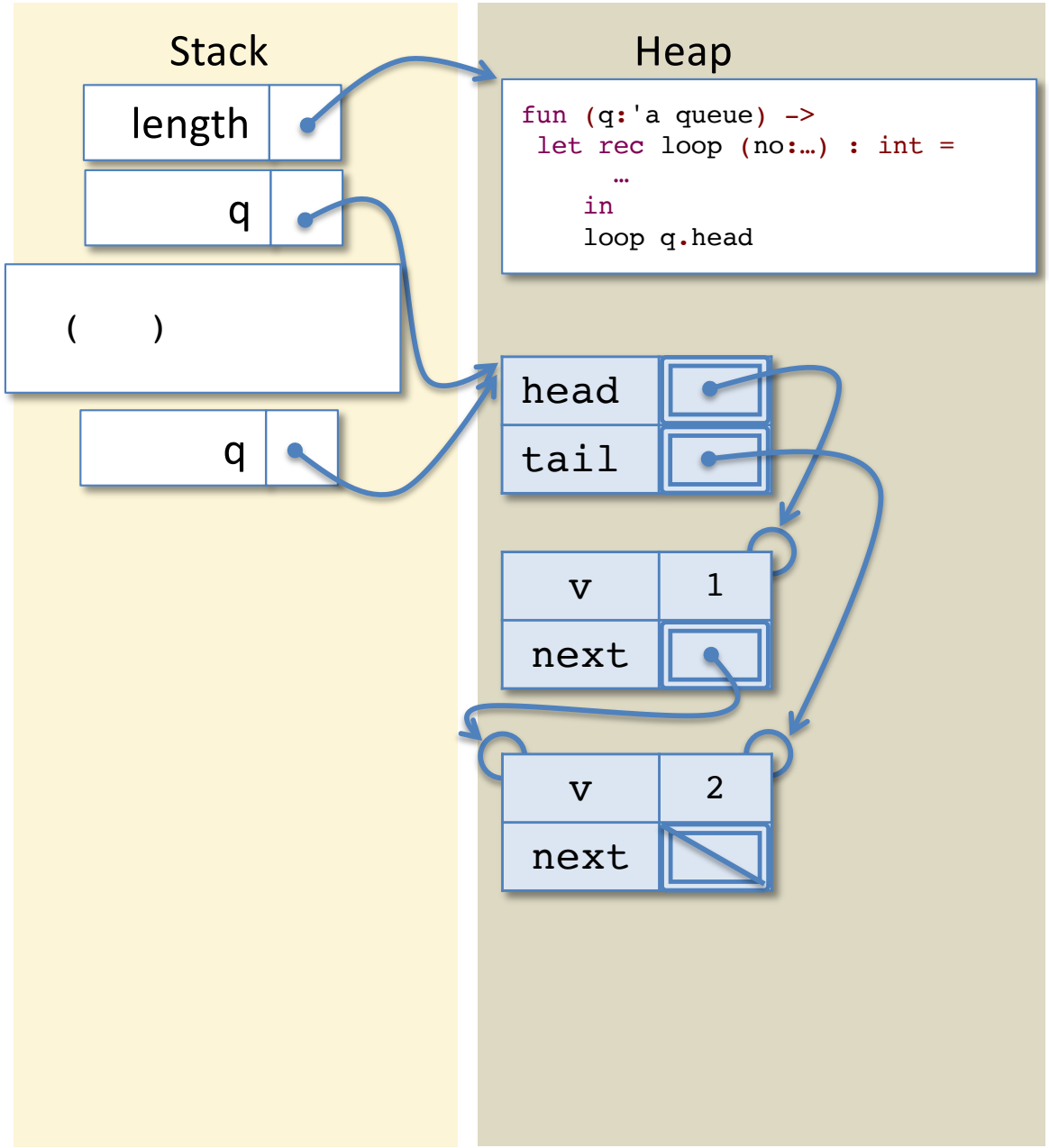
```
let rec loop = fun (no: …) ->
      begin match no with
        | None -> 0
        | Some n -> 1 + (loop n.next)
      end
    in
    loop q.head
```



Evaluating length

Workspace

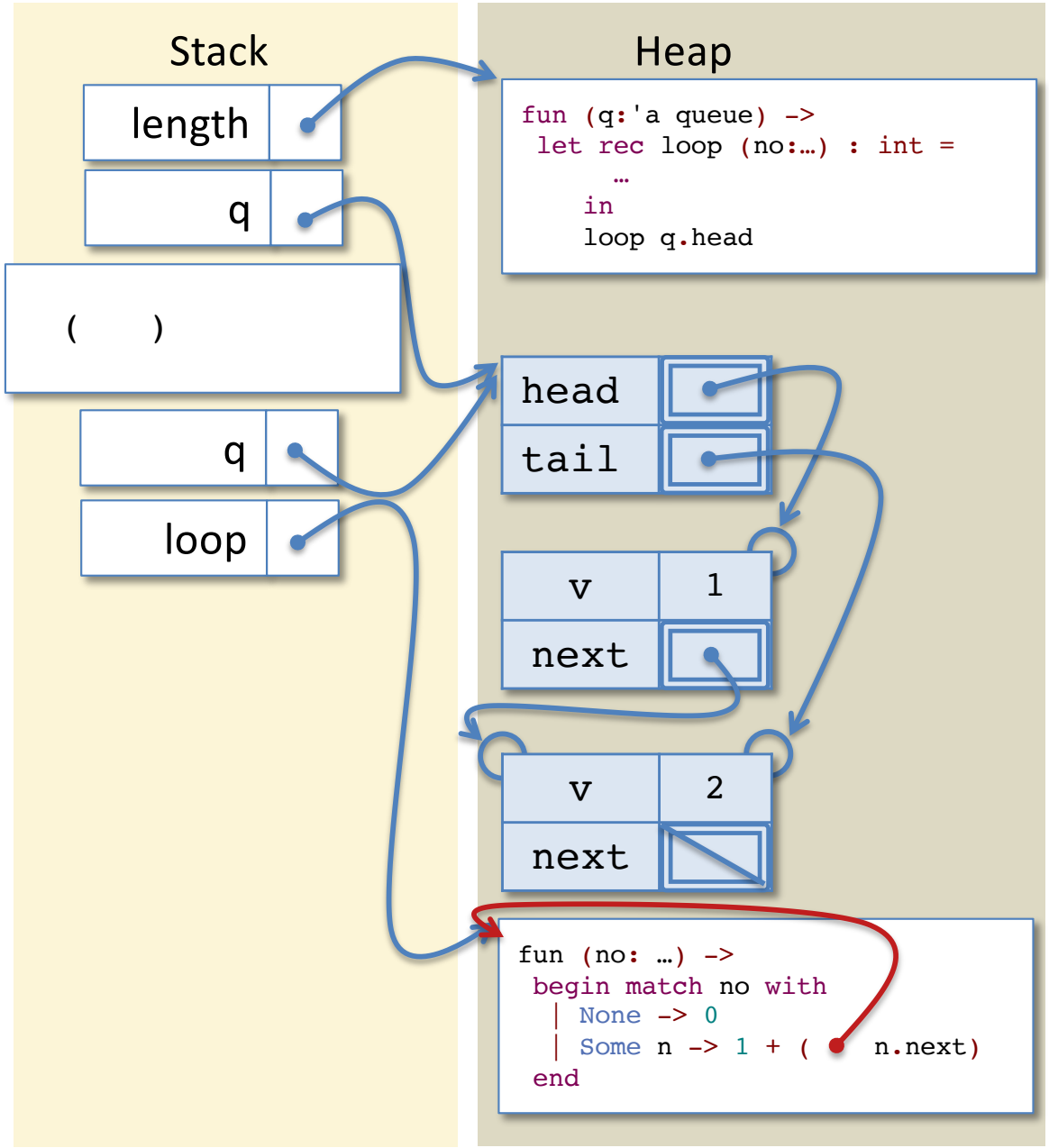
```
let loop = fun (no: …) ->
      begin match no with
          | None -> 0
          | Some n -> 1 + (loop n.next)
      end
    in
    loop q.head
```



Evaluating length

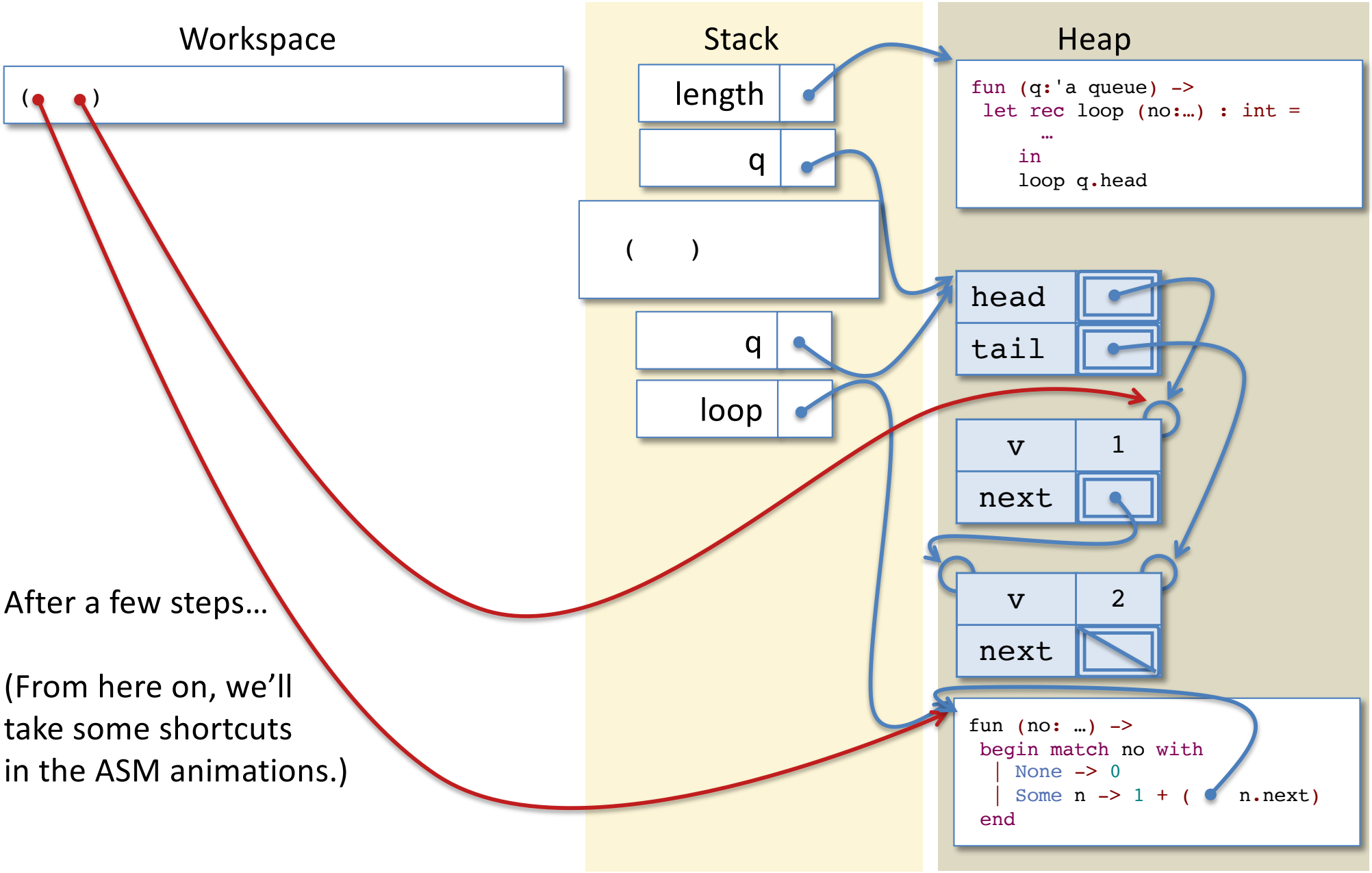
Workspace

```
loop q.head
```

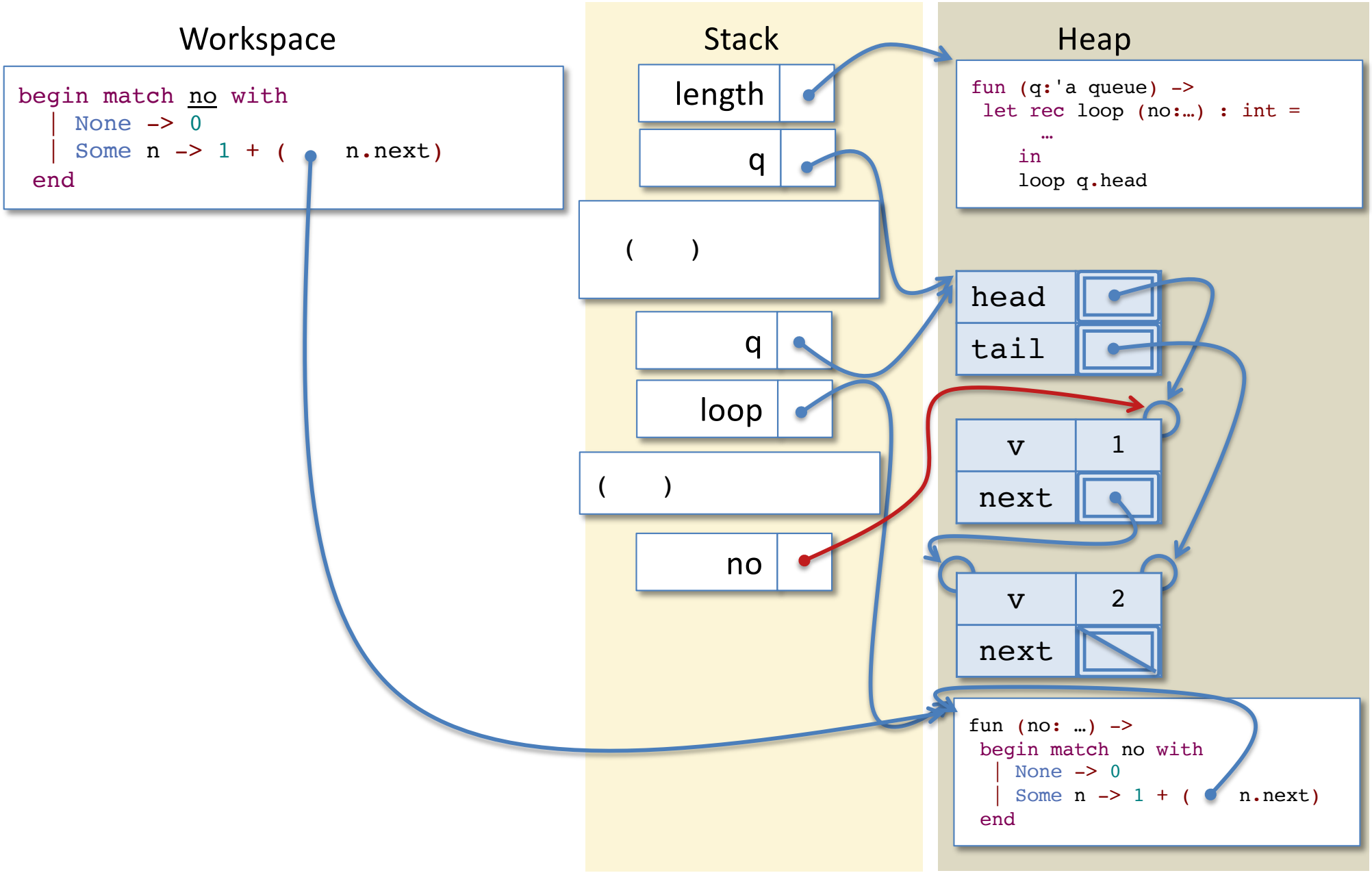


Recall that the loop reference in its own definitions is backpatched….

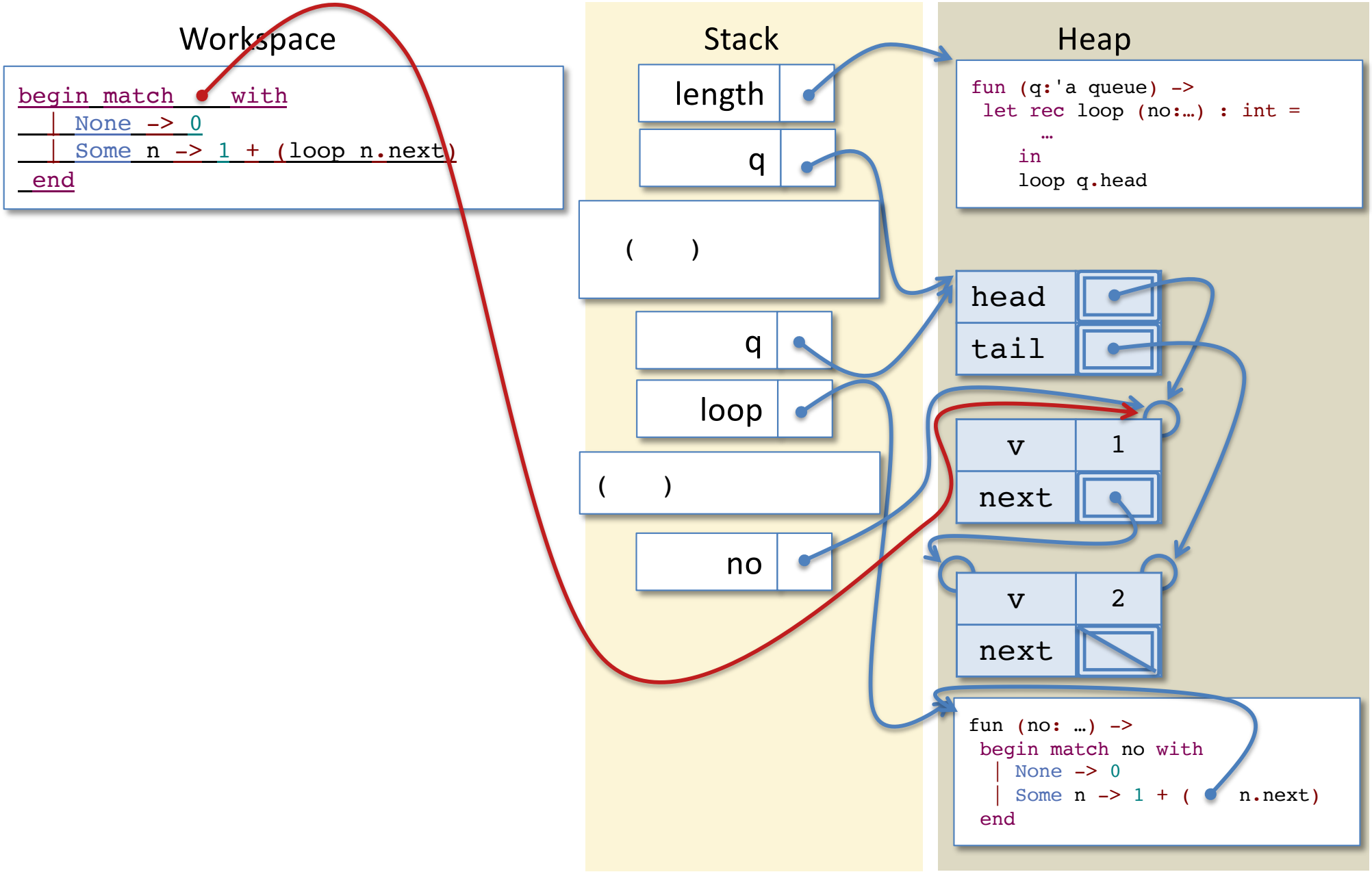
Evaluating length



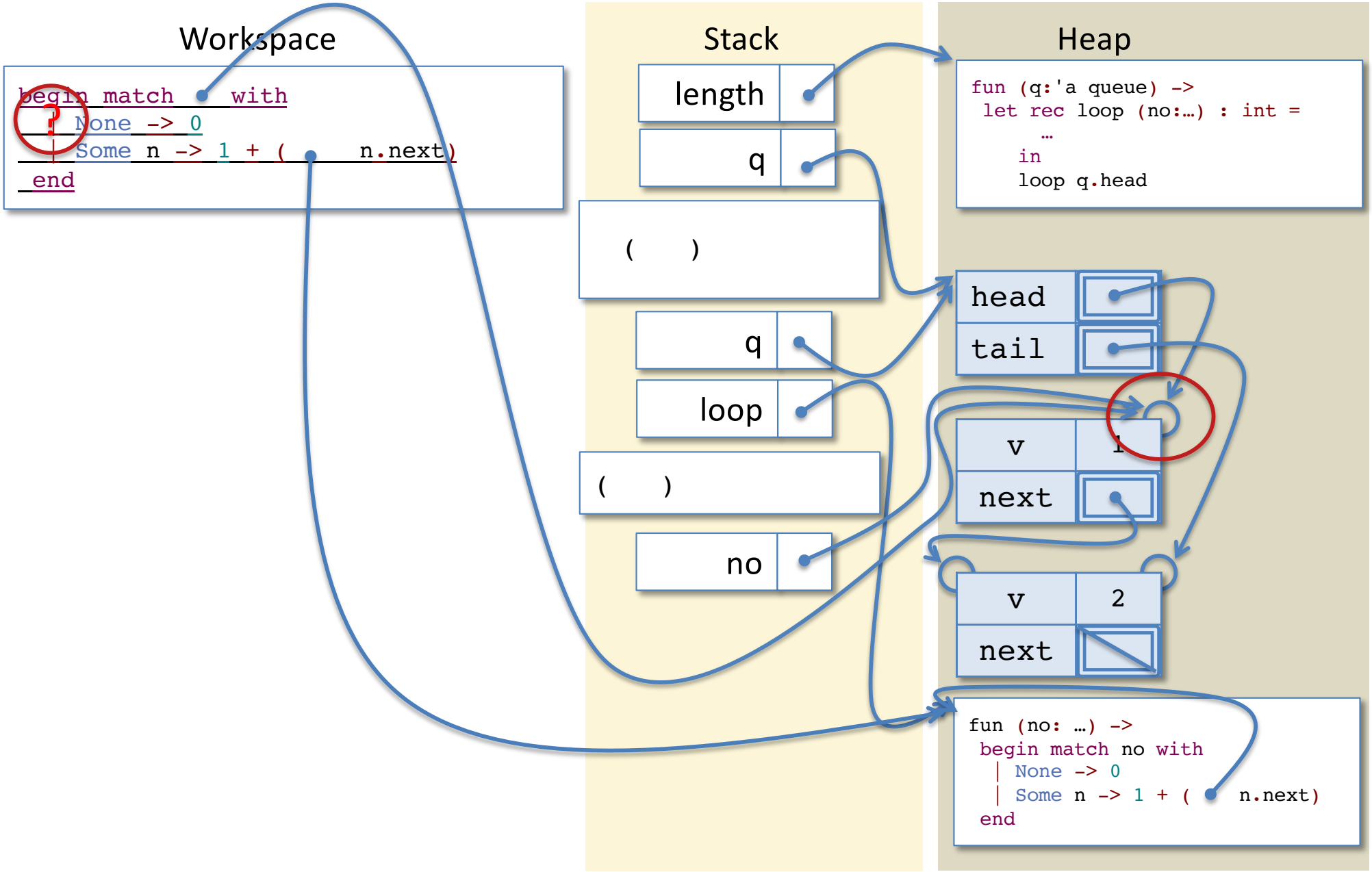
Evaluating length



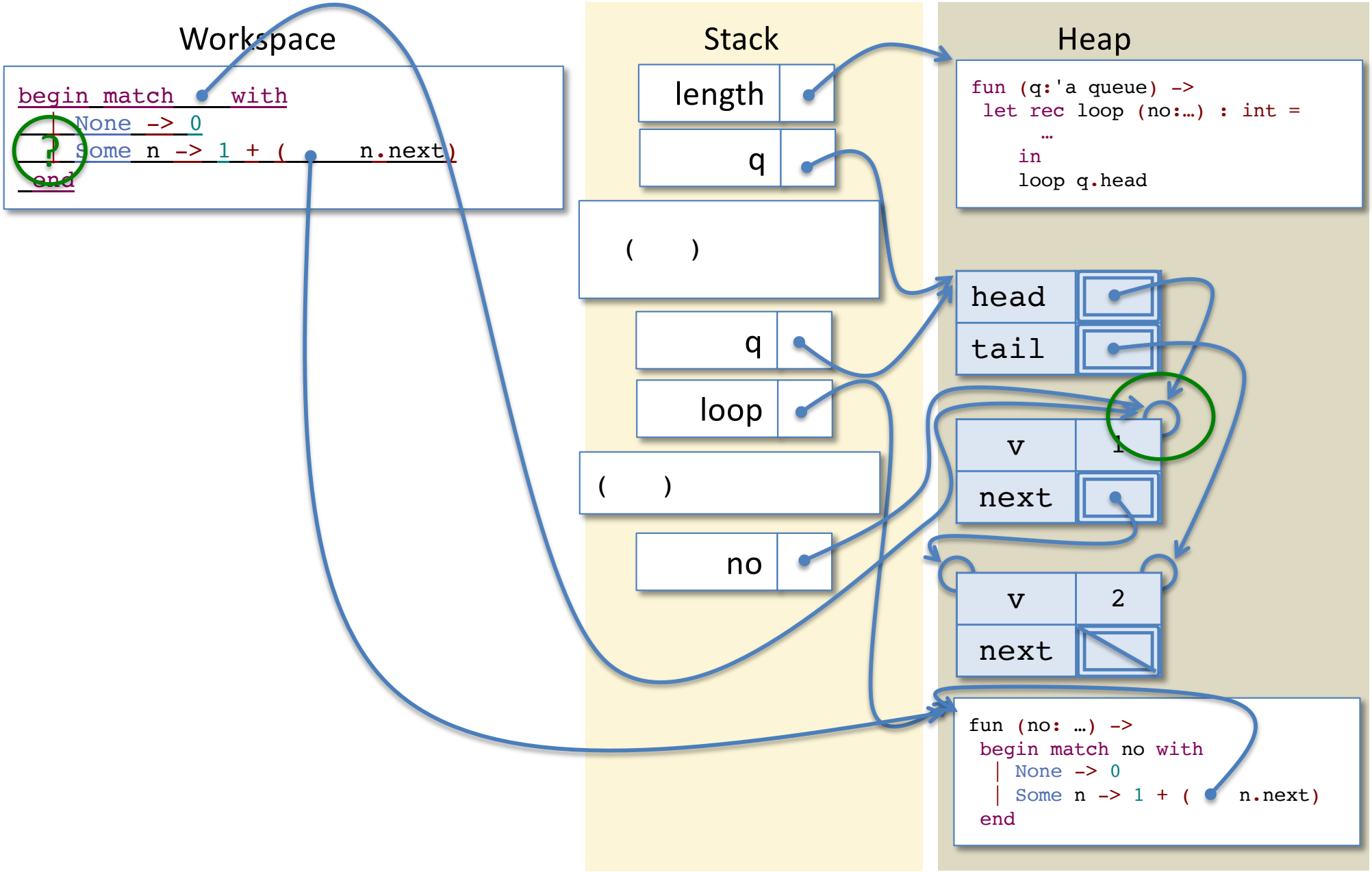
Evaluating length



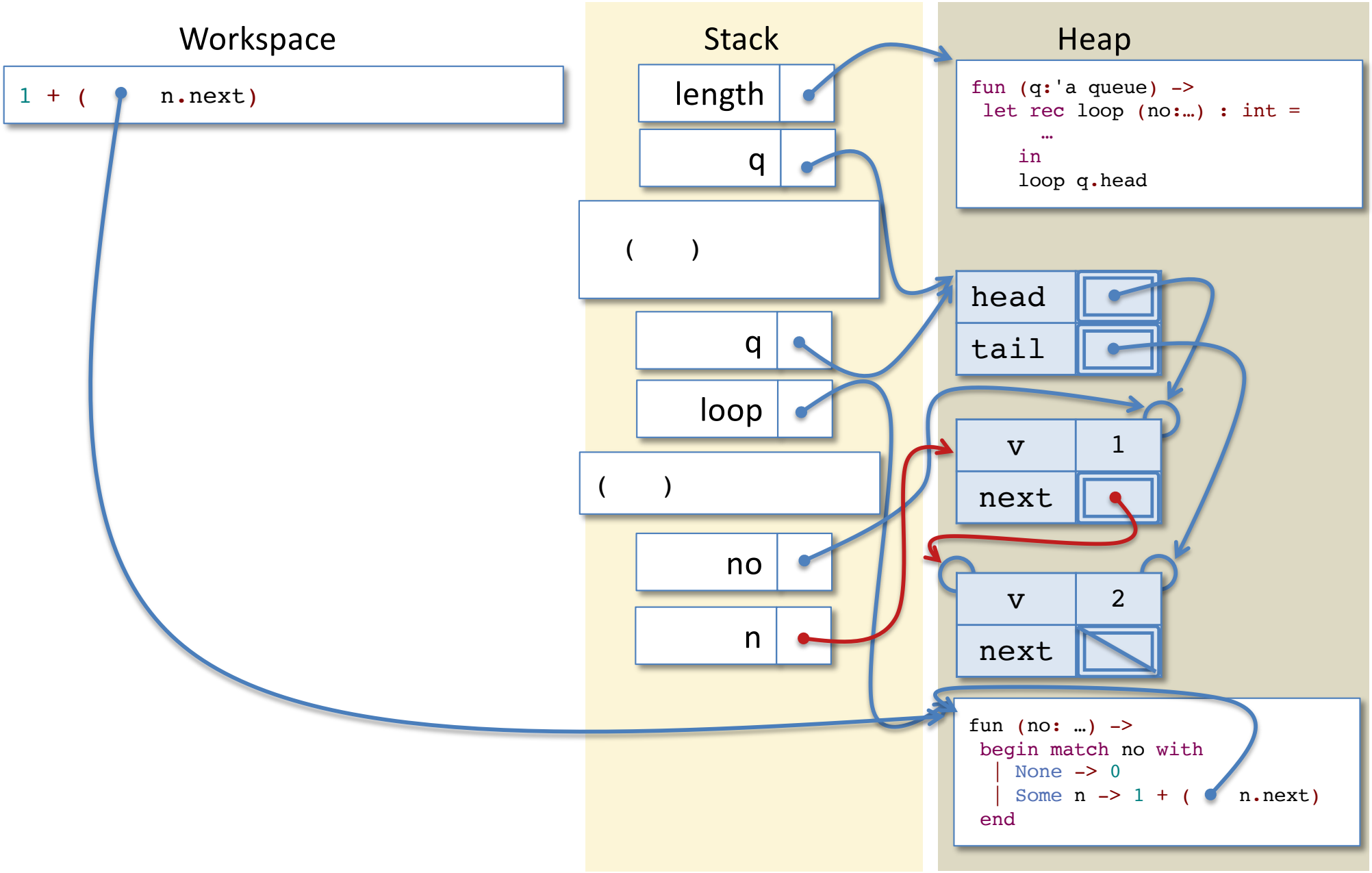
Evaluating length



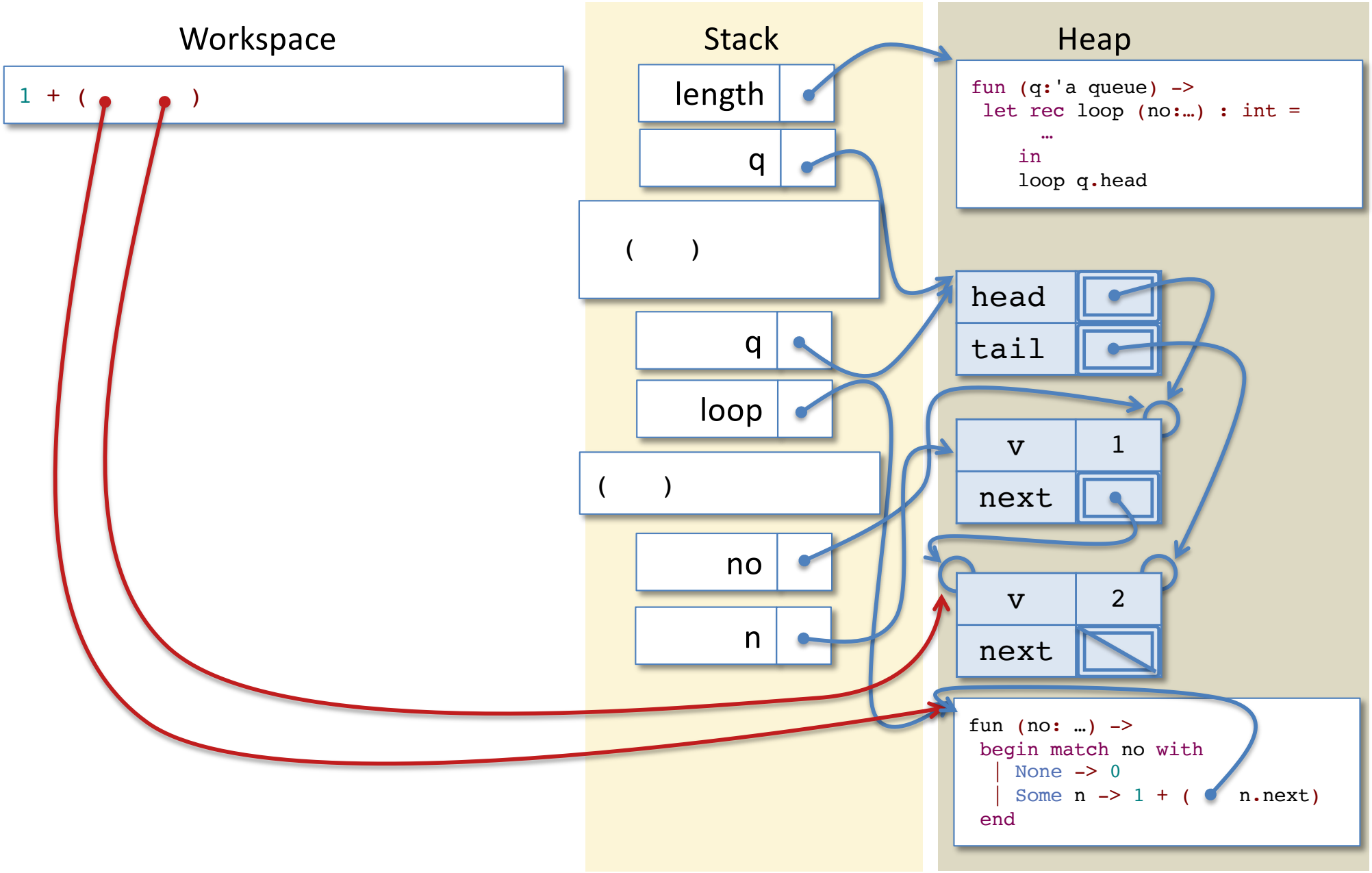
Evaluating length



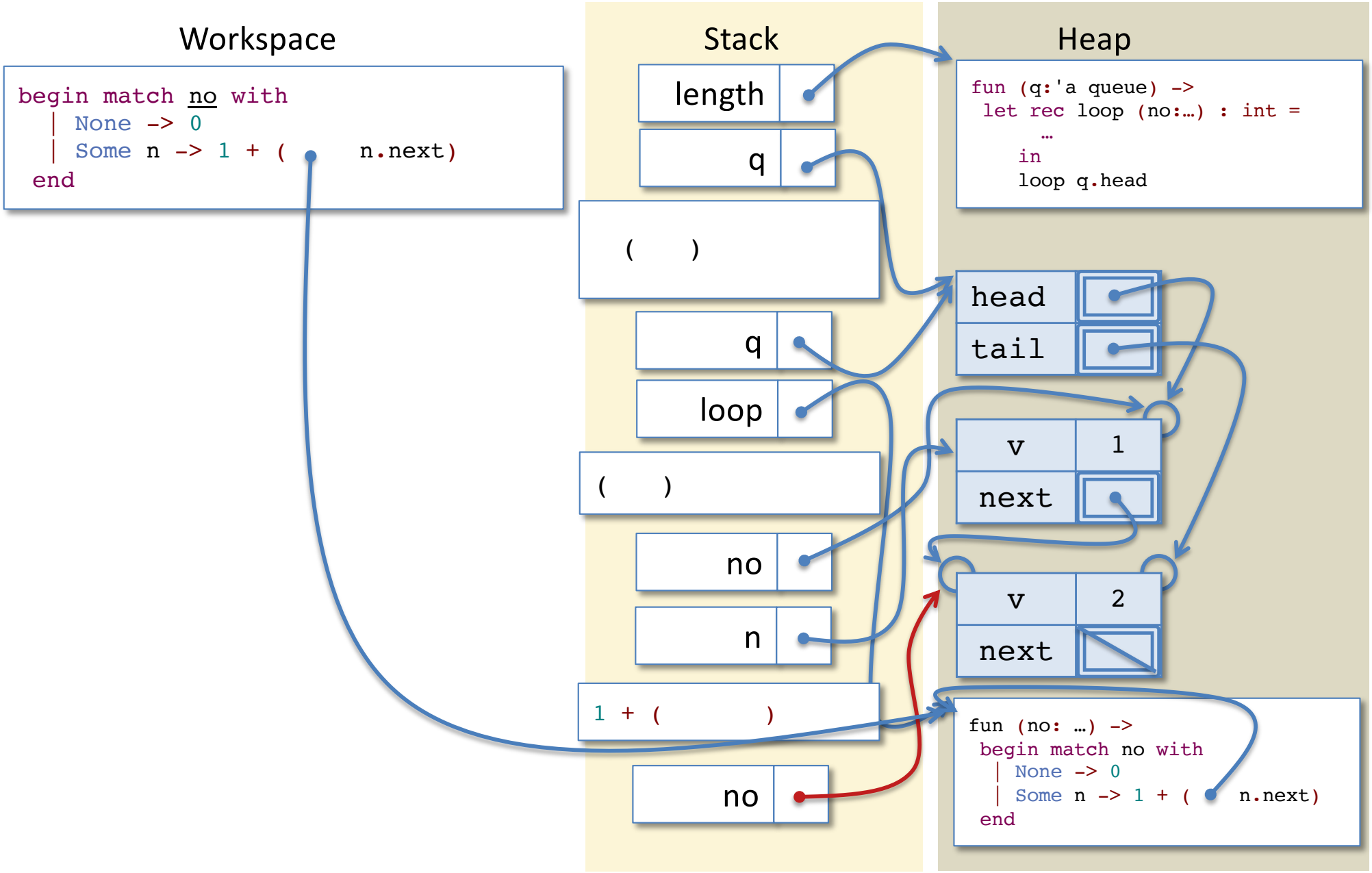
Evaluating length



Evaluating length

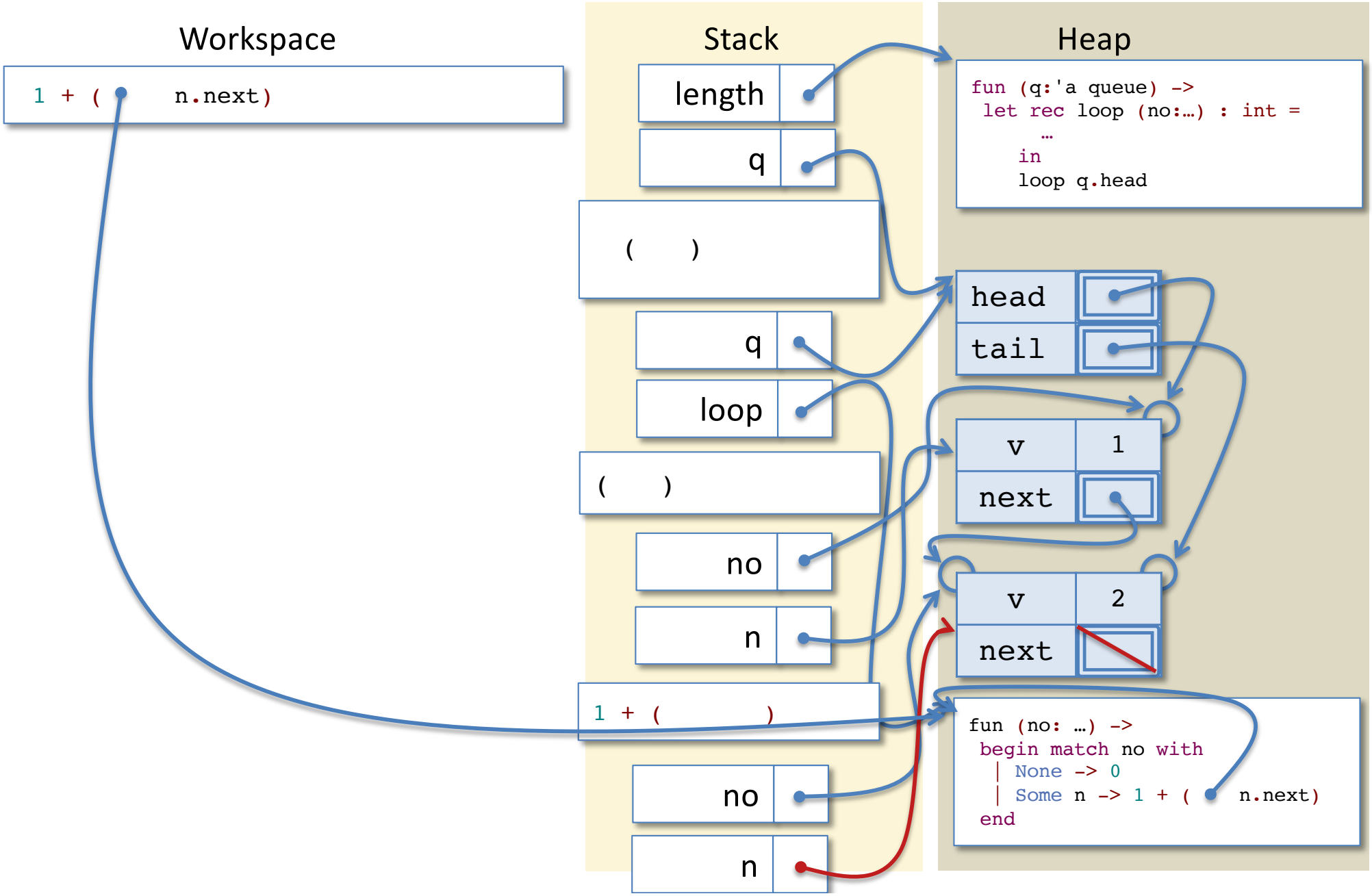


Evaluating length



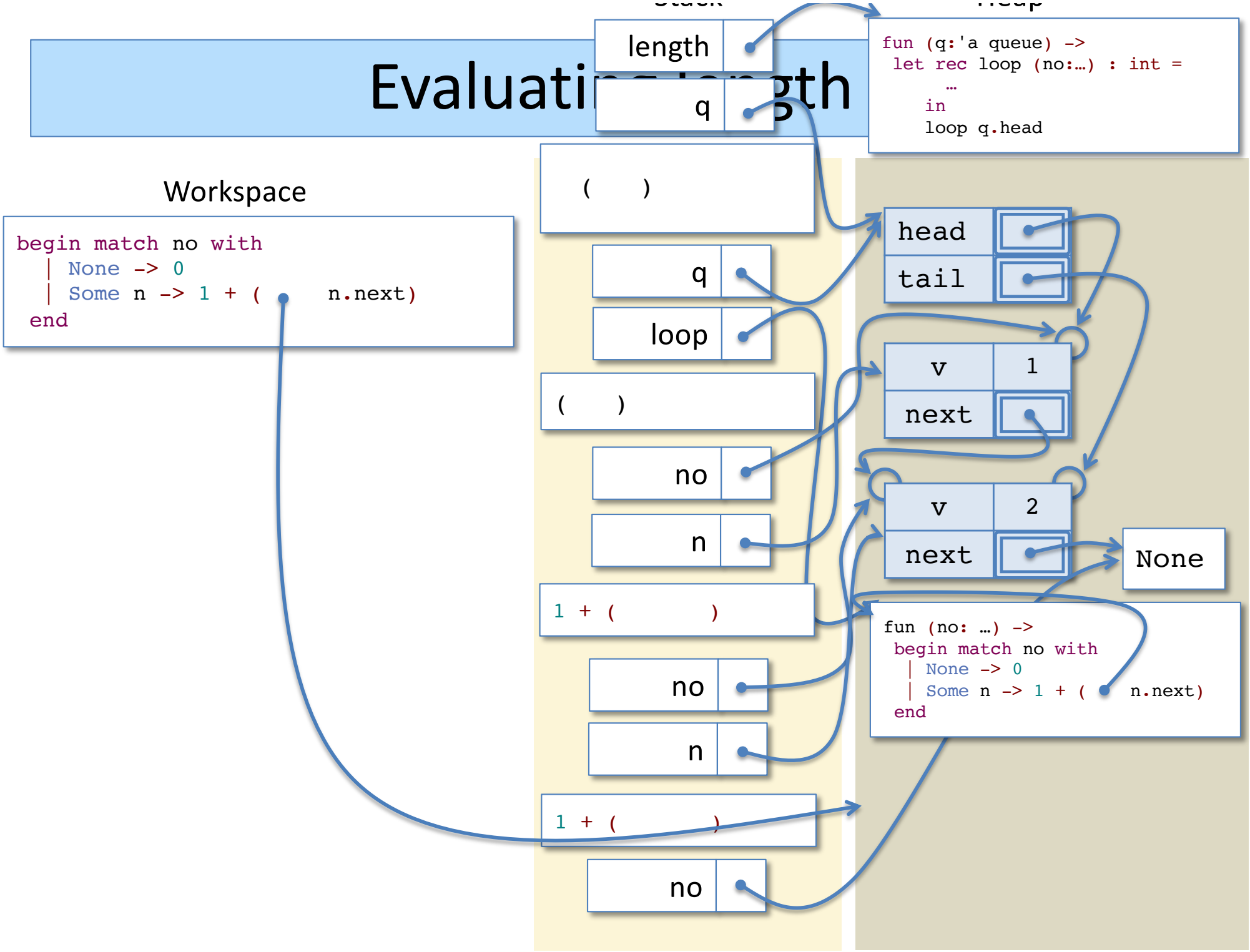
…after a few steps…

Evaluating length

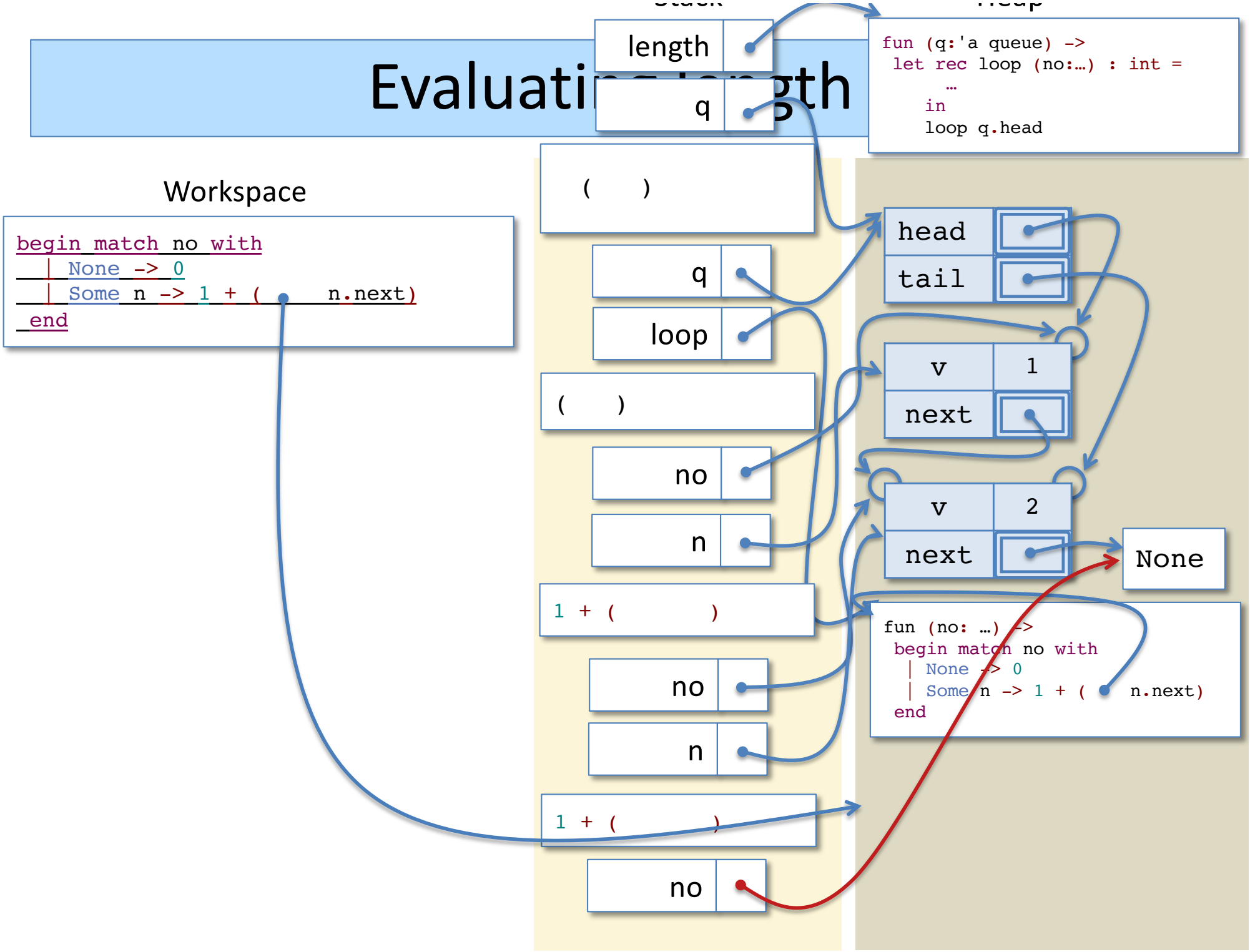


…after a few more steps…

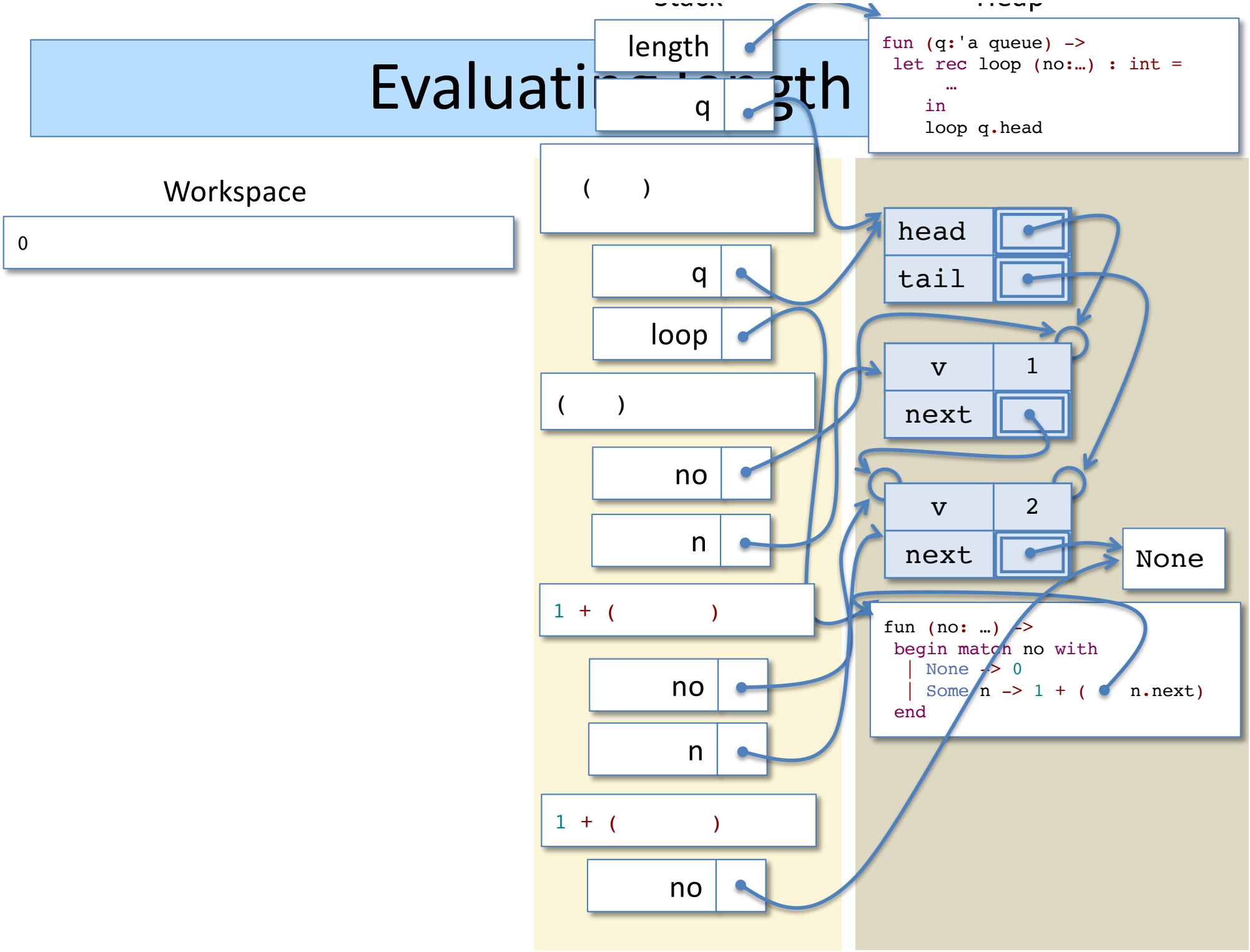
Evaluating length



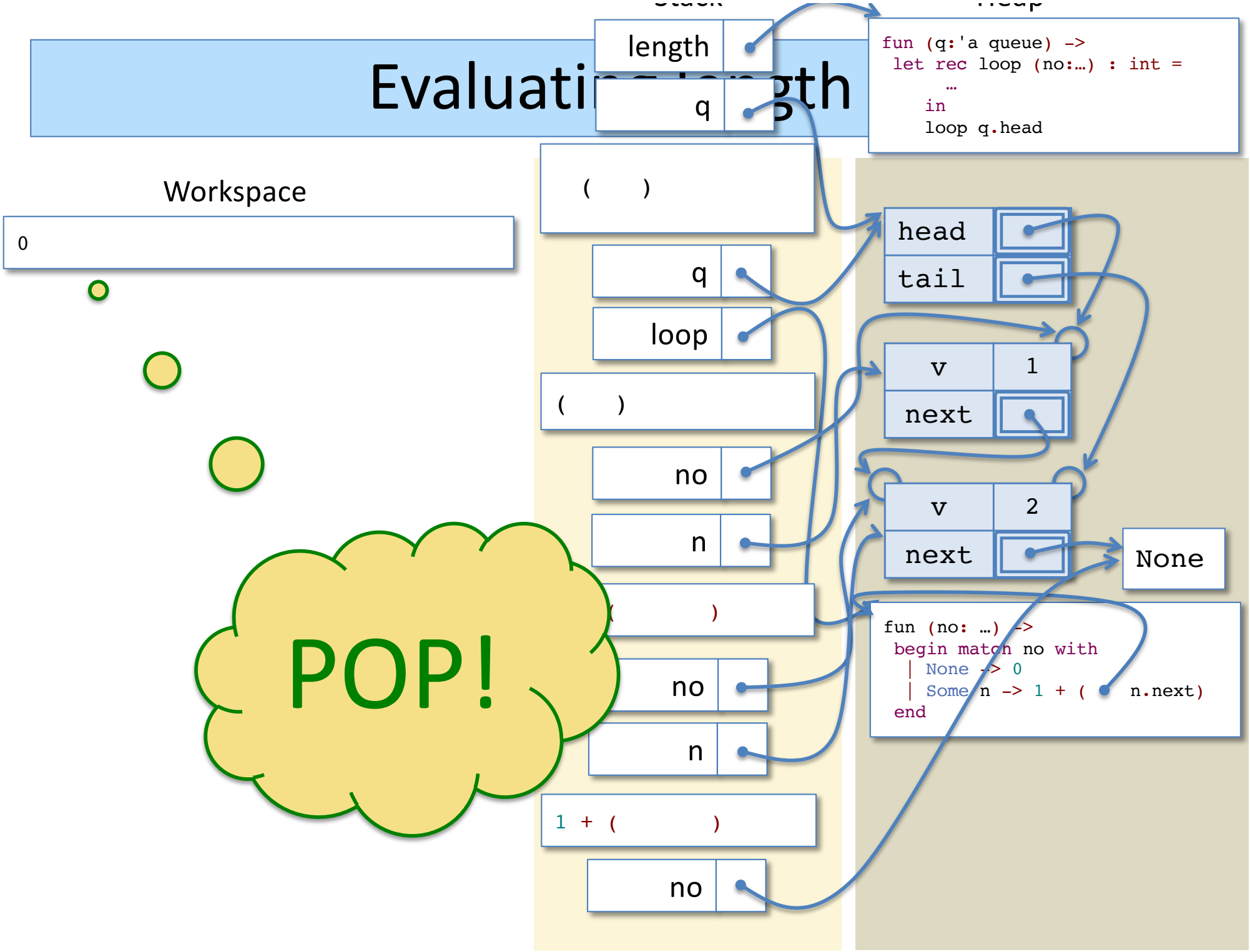
Evaluating length



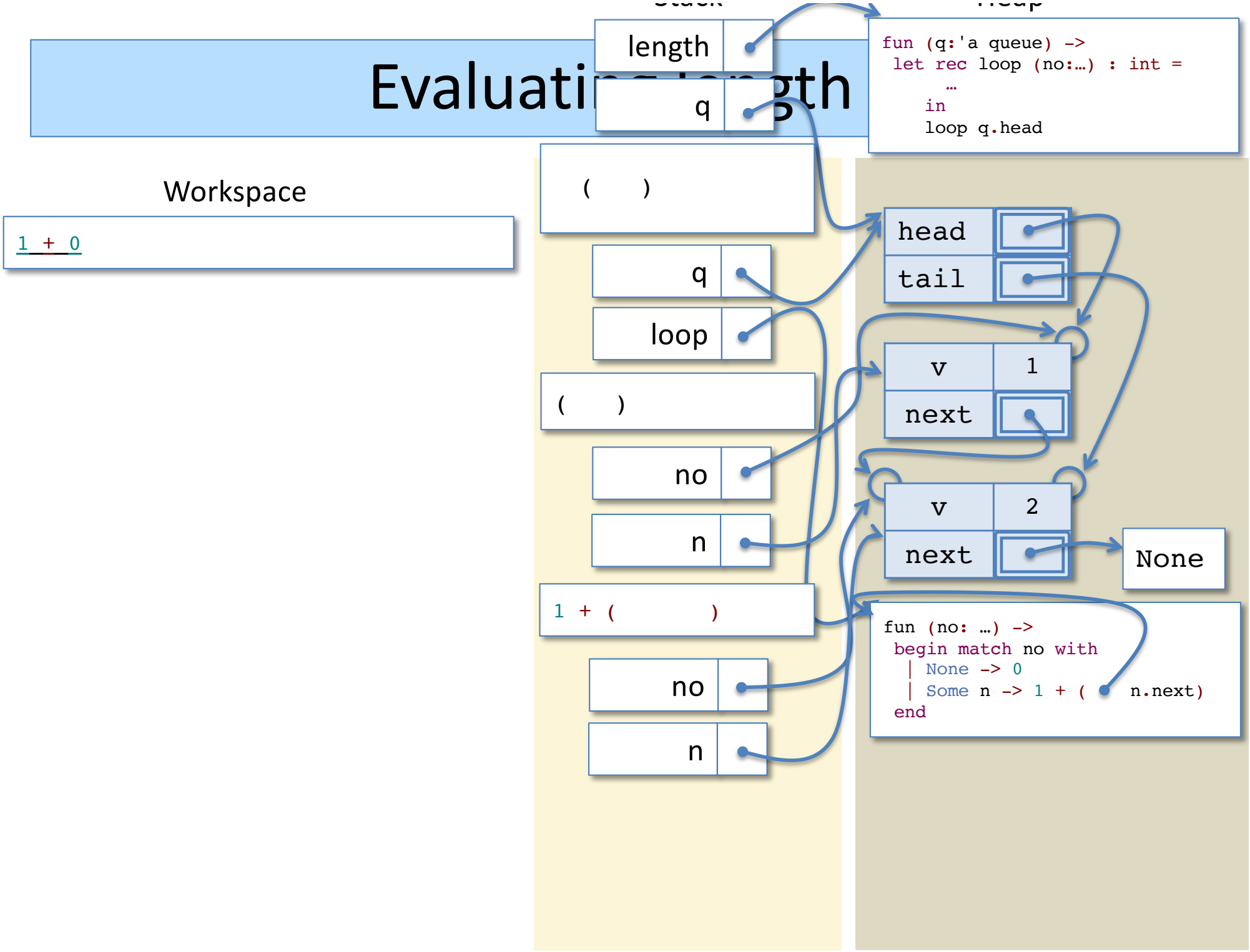
Evaluating length



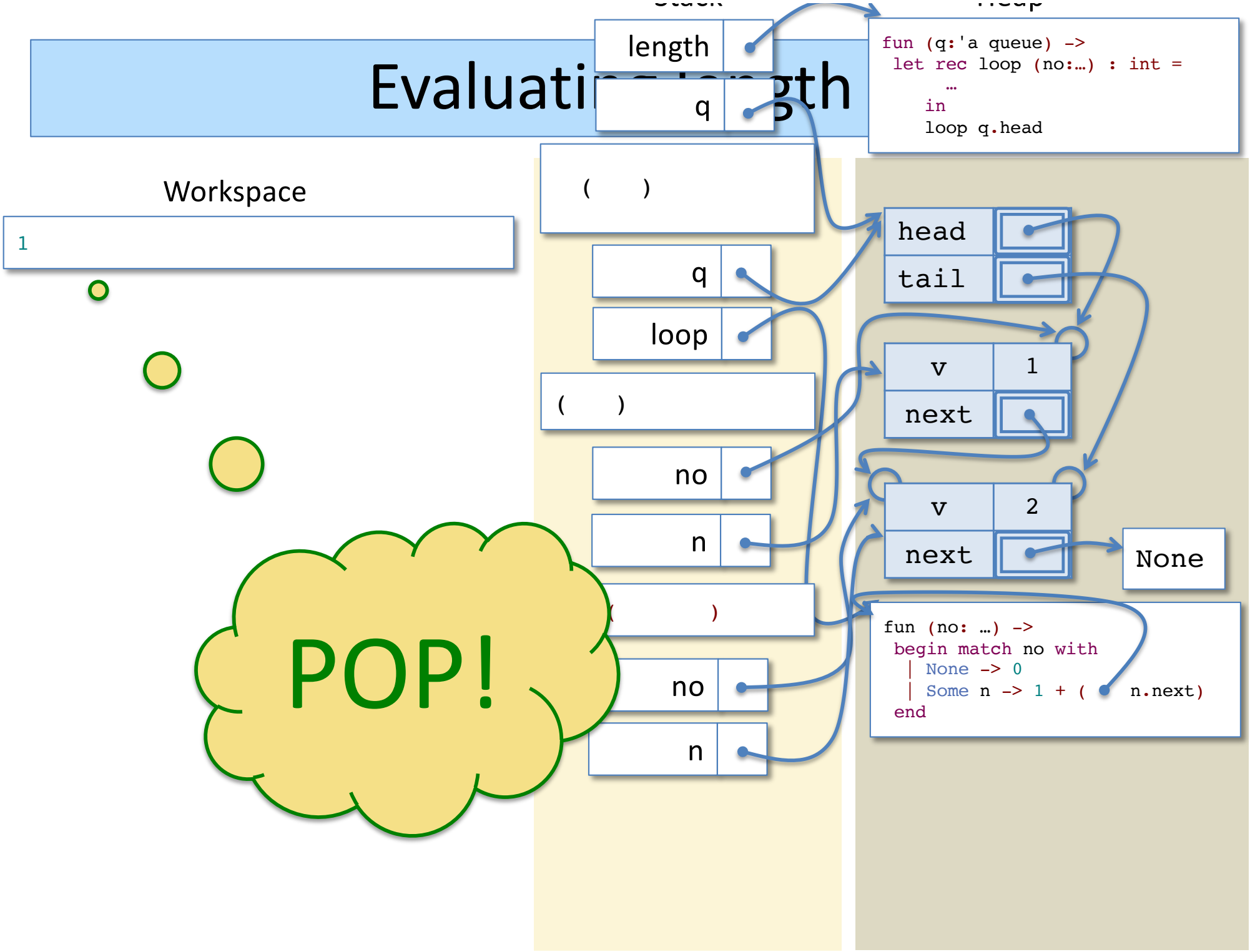
Evaluating length



Evaluating length



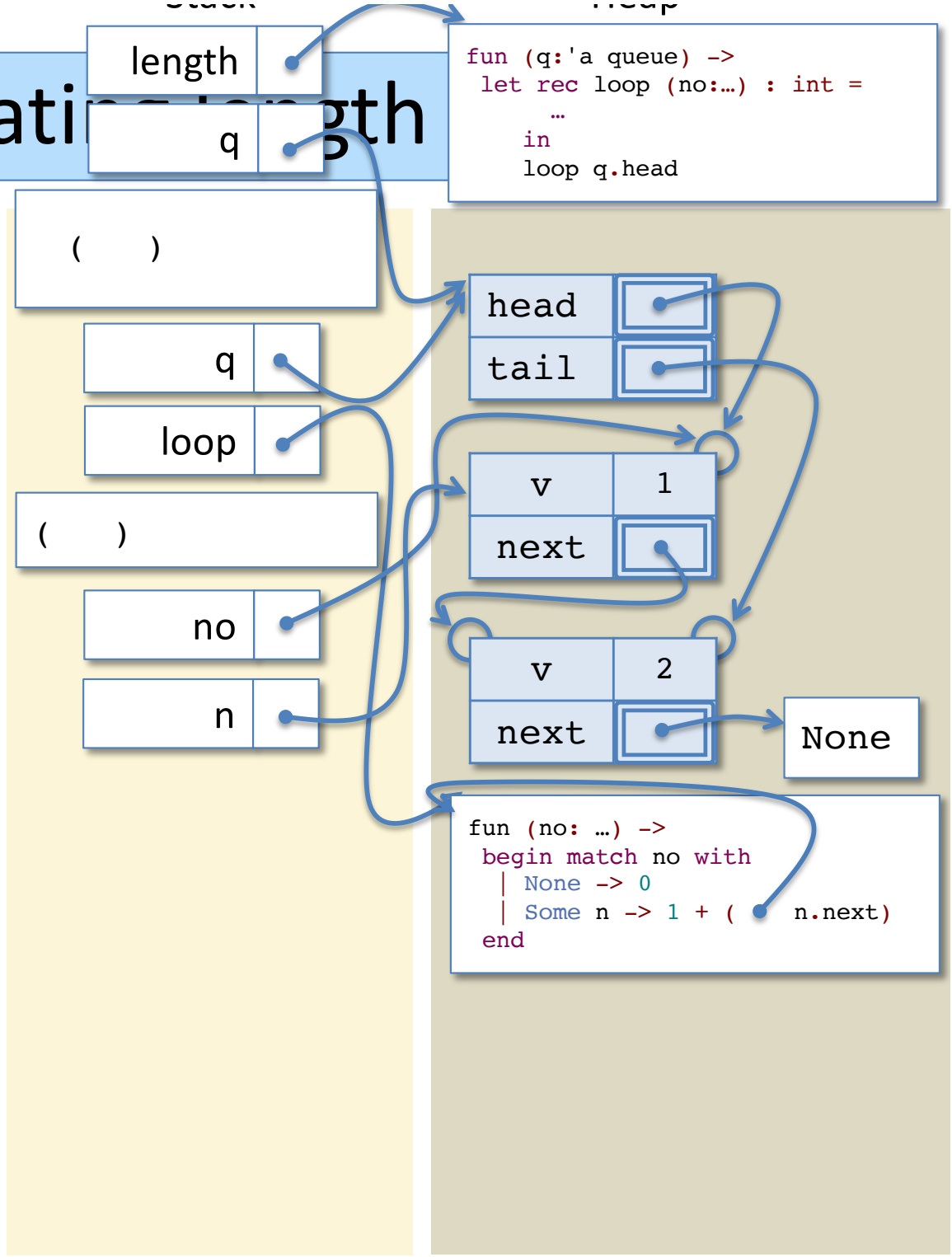
Evaluating length



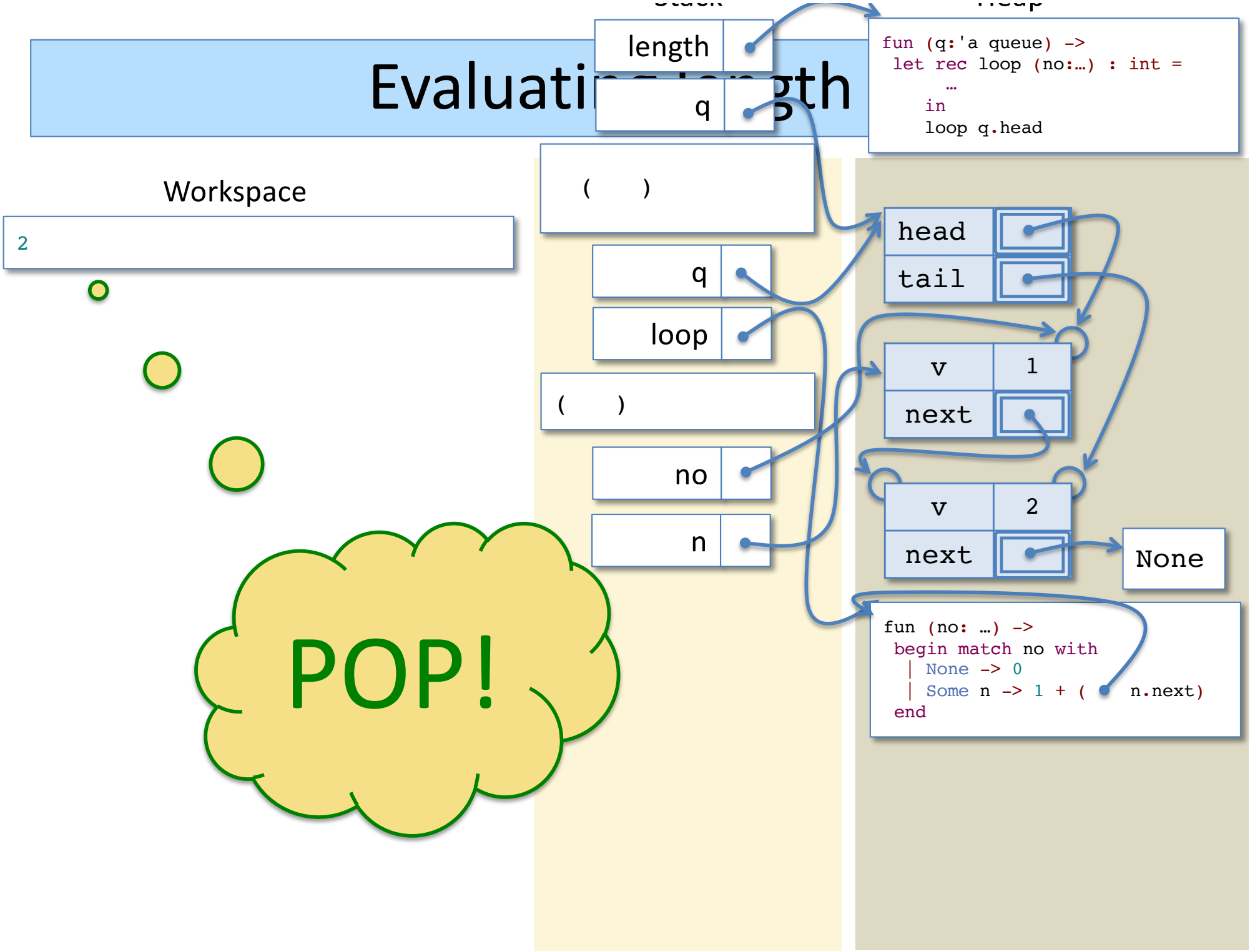
Evaluating length

Workspace

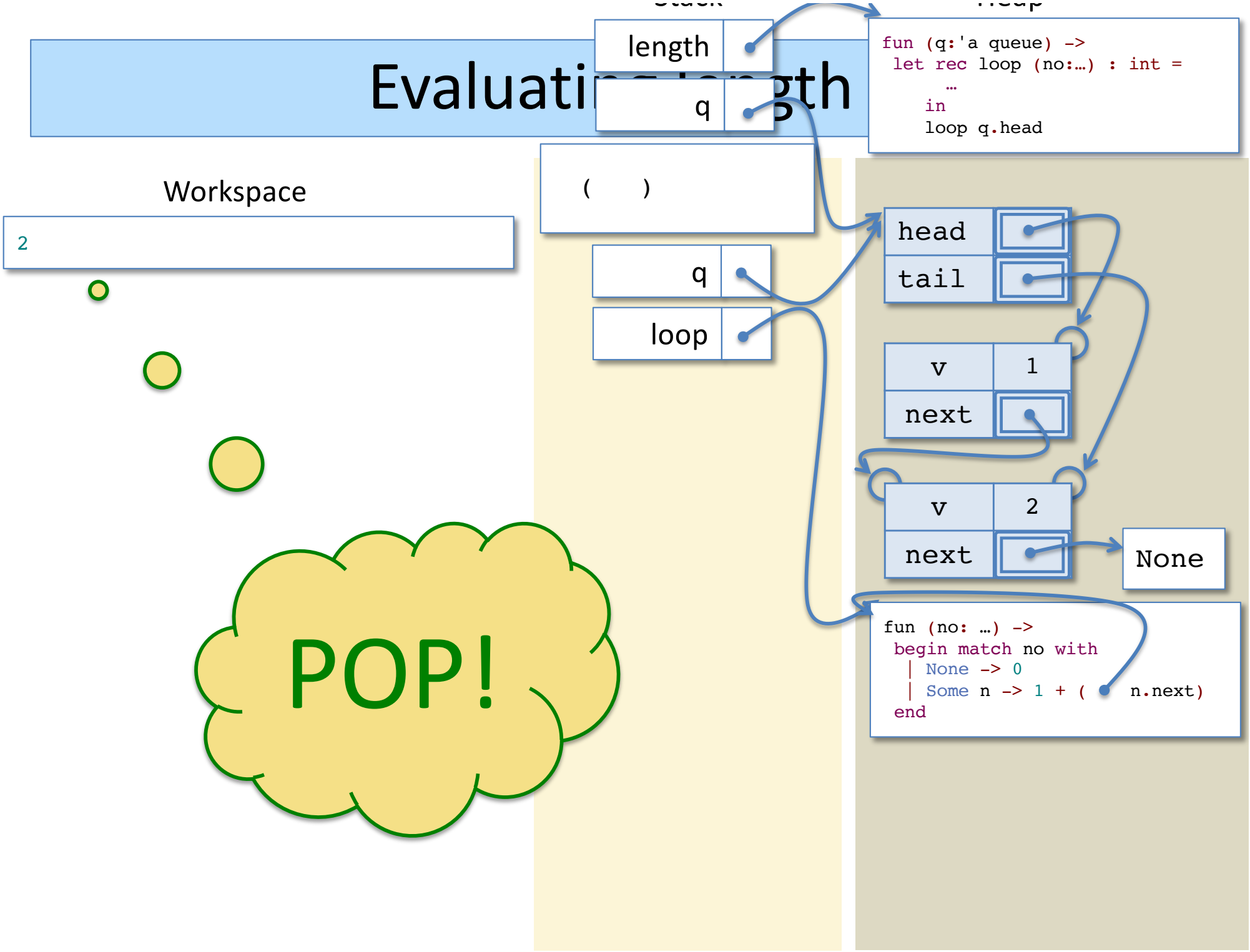
```
1 + 1
```



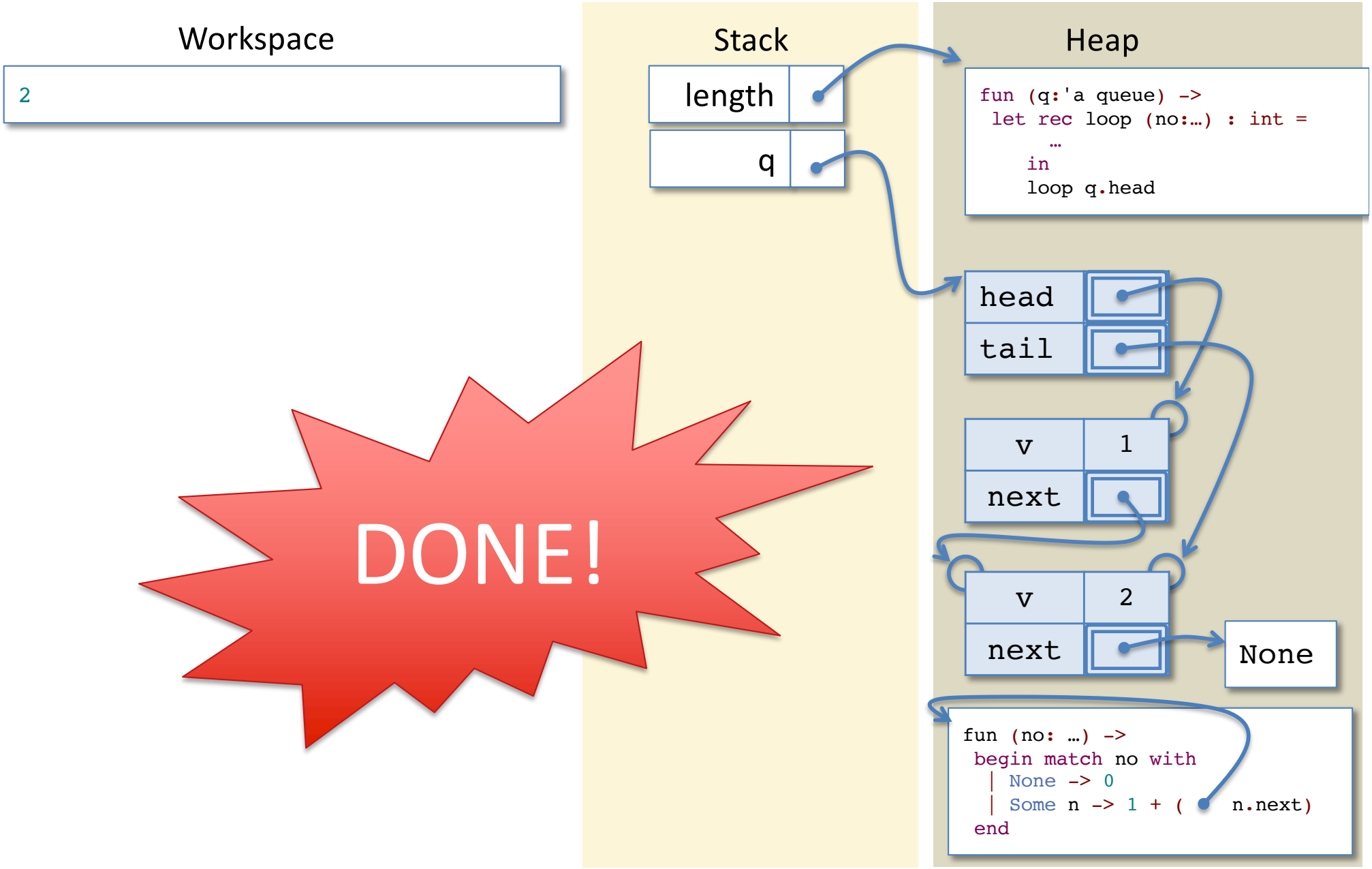
Evaluating length



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 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

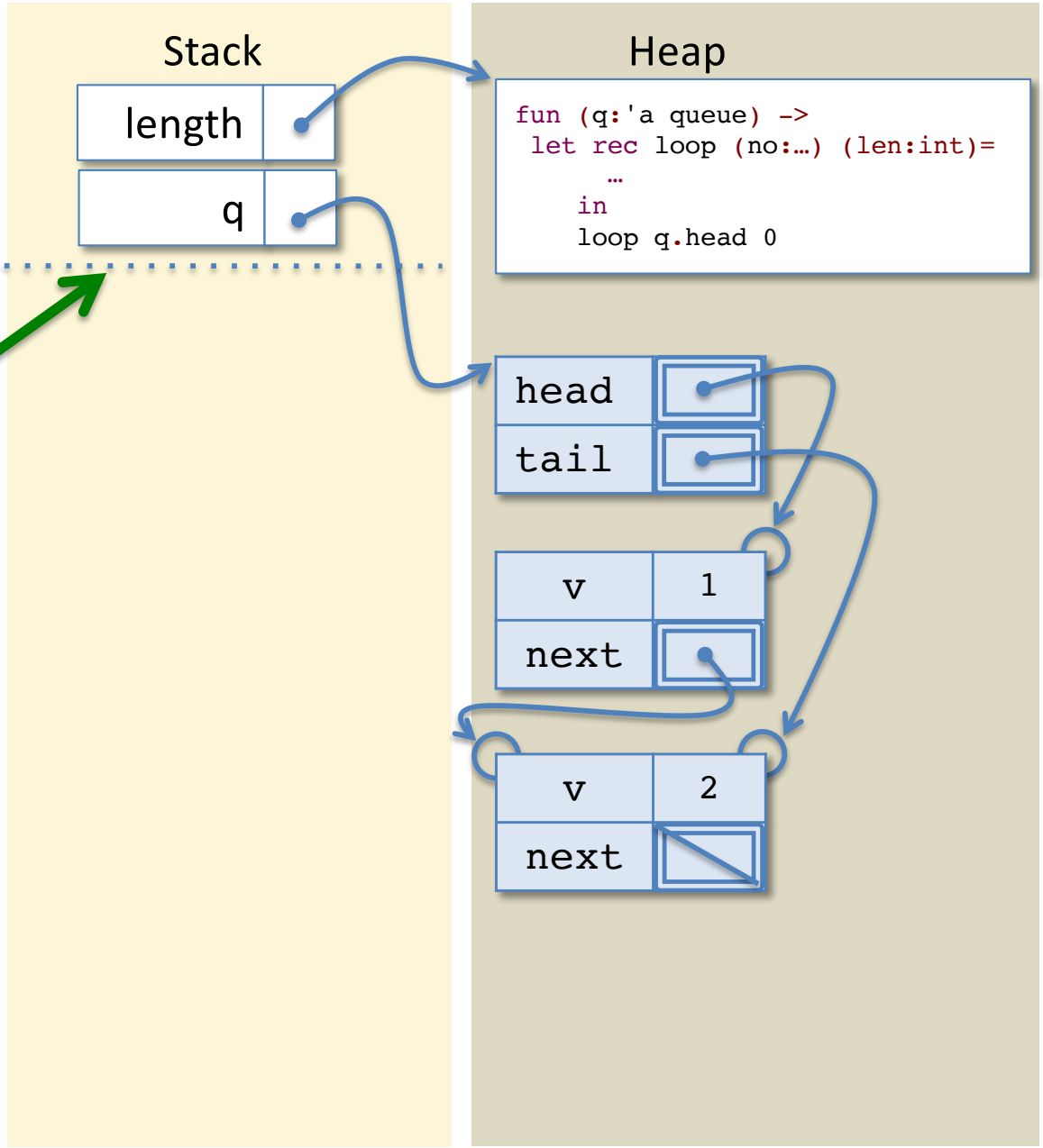
- Why does it matter that ‘loop’ is only called in an empty workspace?
- Answer: We can *optimize* the abstract stack machine!
  - The workspace pushed onto the stack tells us “what to do” when the function call returns. If empty, "what to do" is pop immediately.
  - If there is nothing to do after a function call, we are done with the current set of local variables – so pop them early to save space.
  - Plus, no need to save the empty workspace either.
- 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

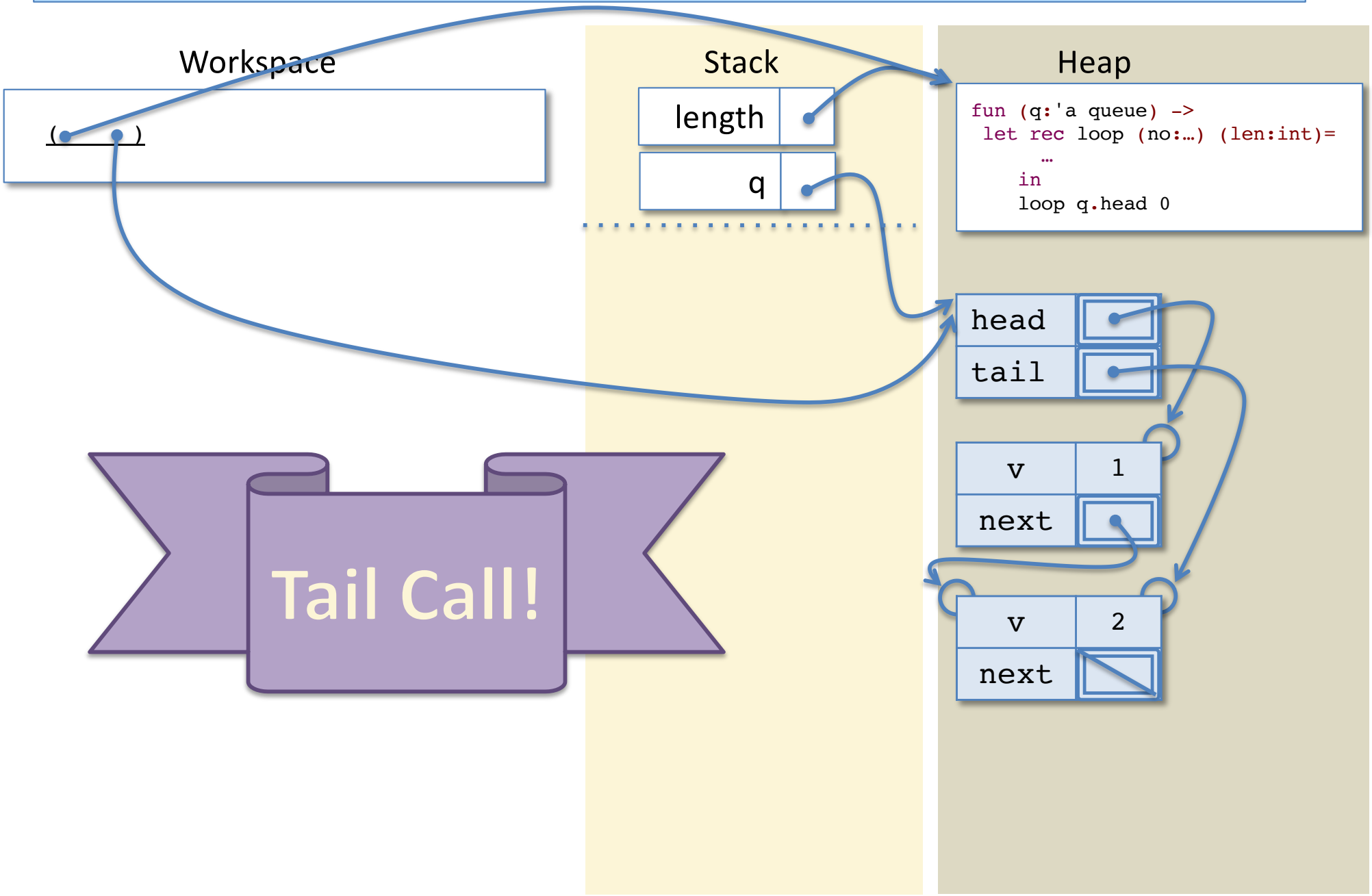
Workspace

```
length q
```

Bindings above this line
are top-level declarations.



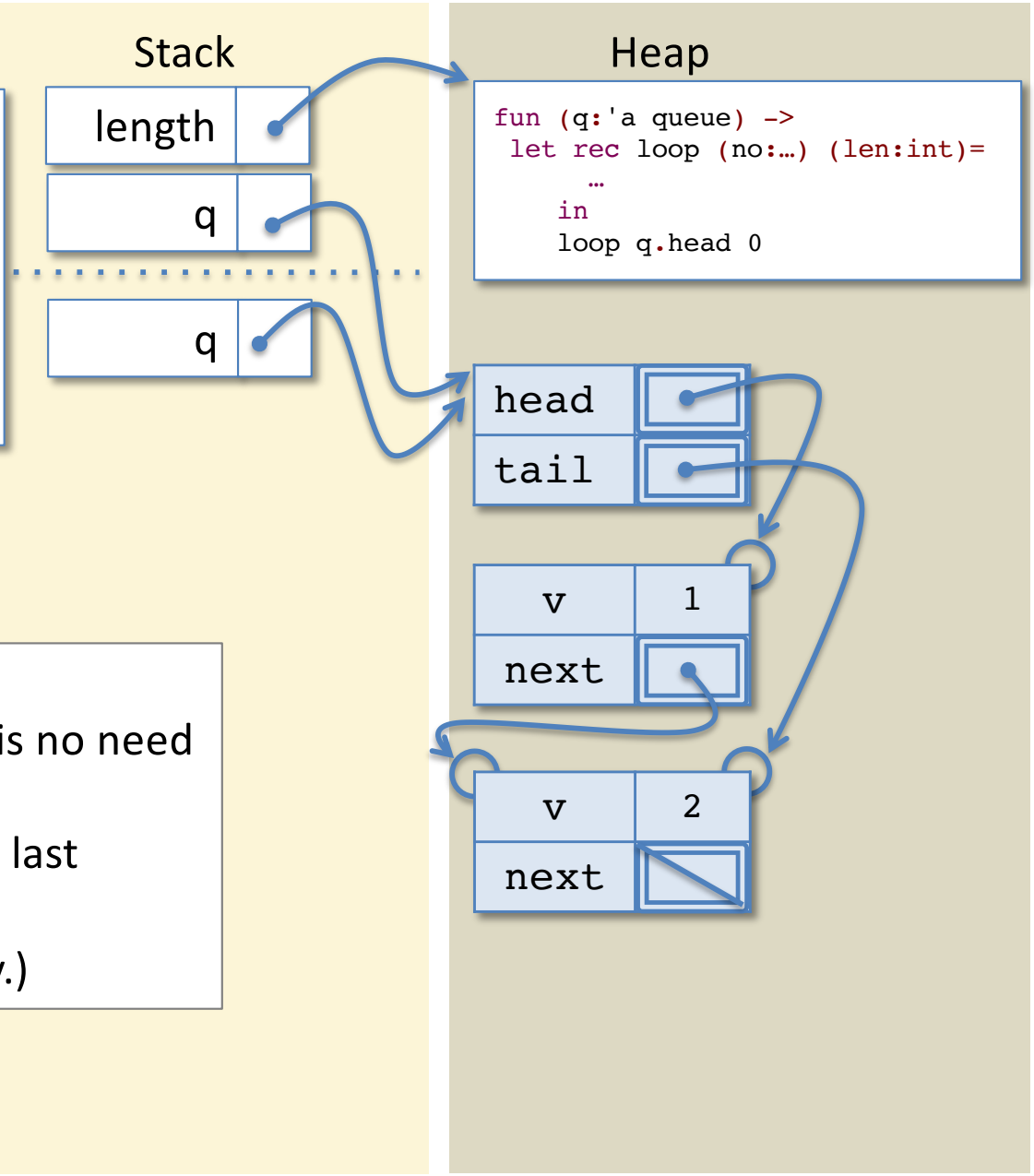
Tail Calls and Iterative length



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
```

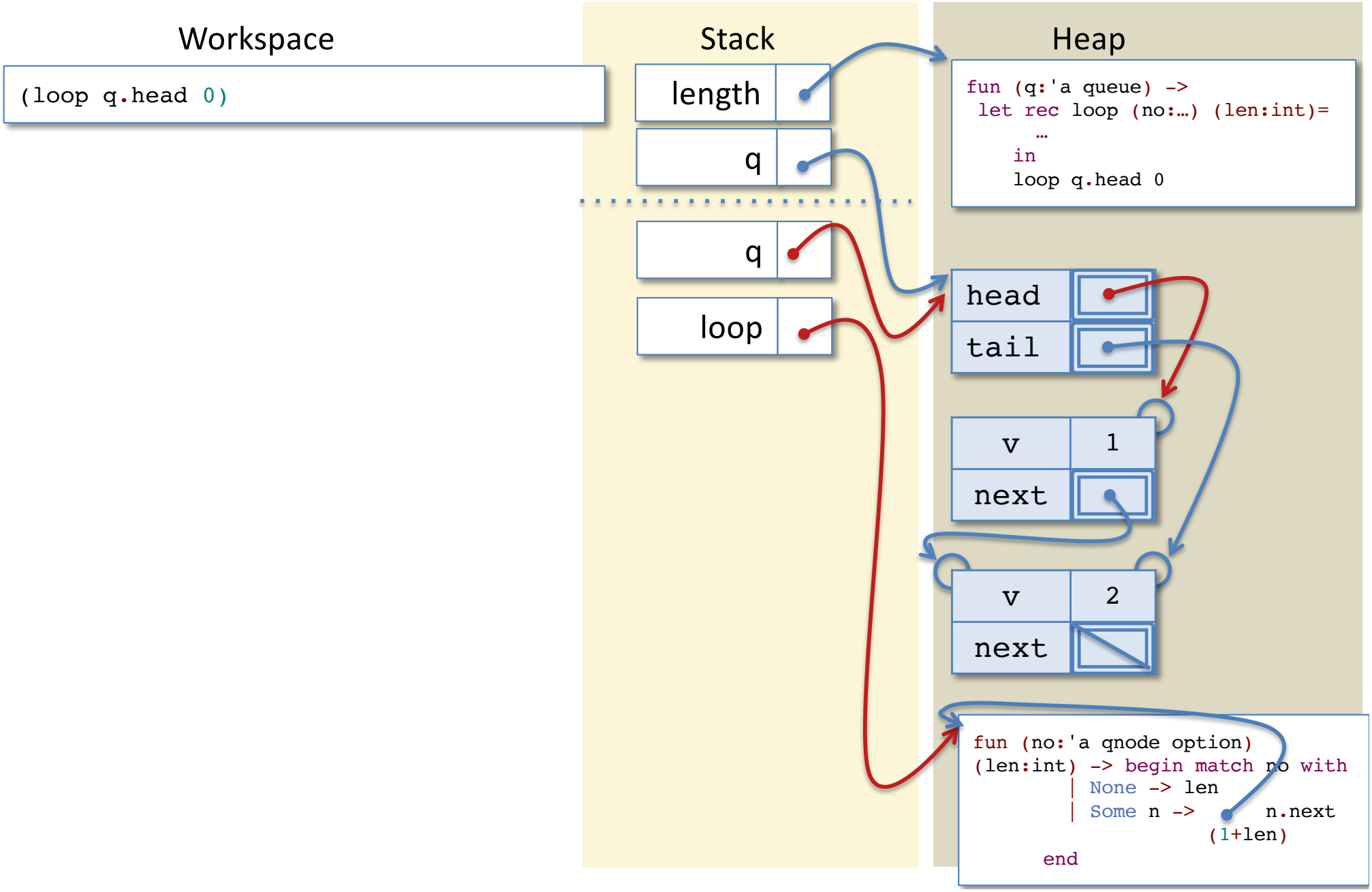


Note:

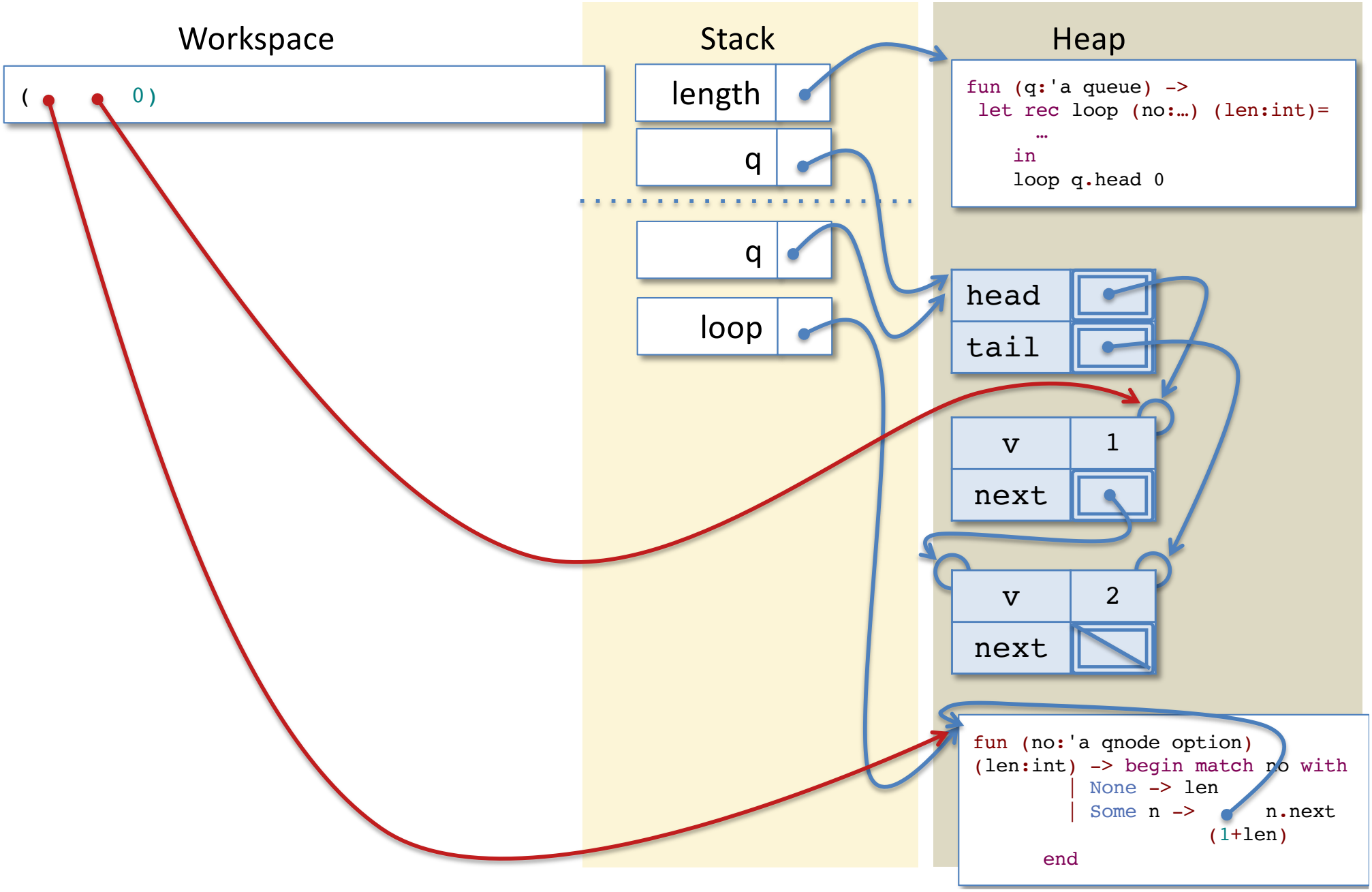
(1) No workspace is saved – there is no need do to that for tail calls

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

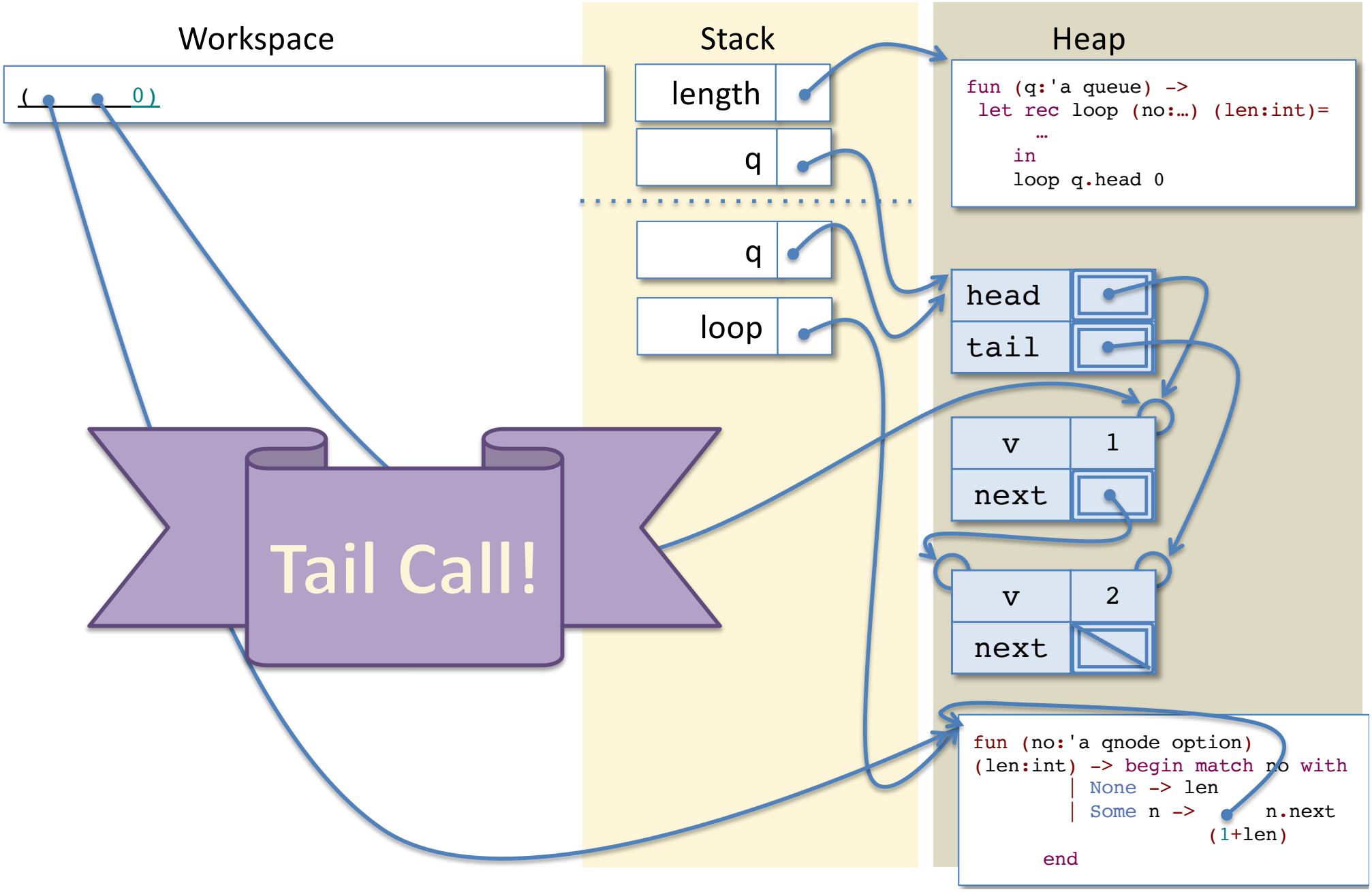
Tail Calls and Iterative length



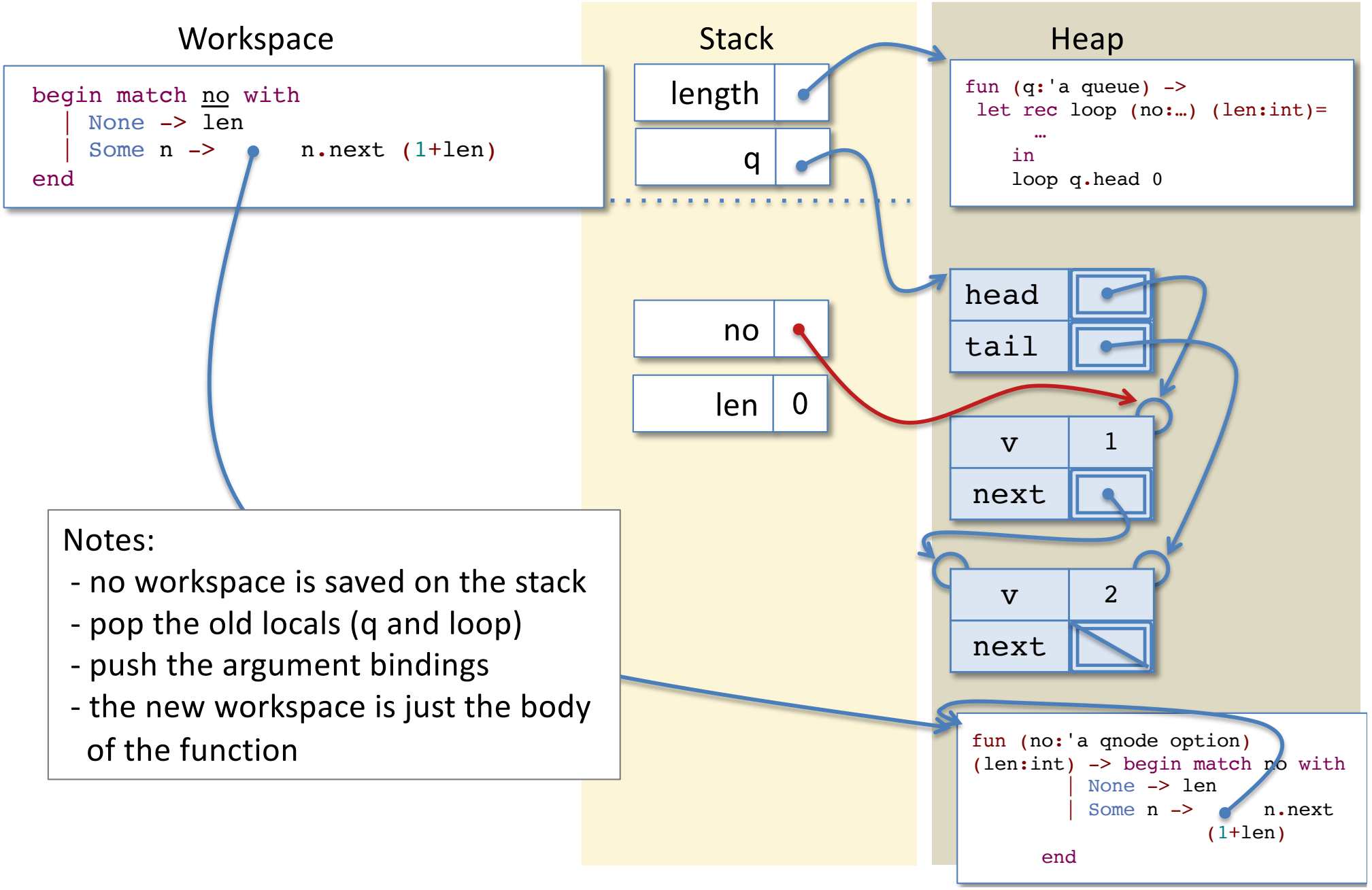
Tail Calls and Iterative length



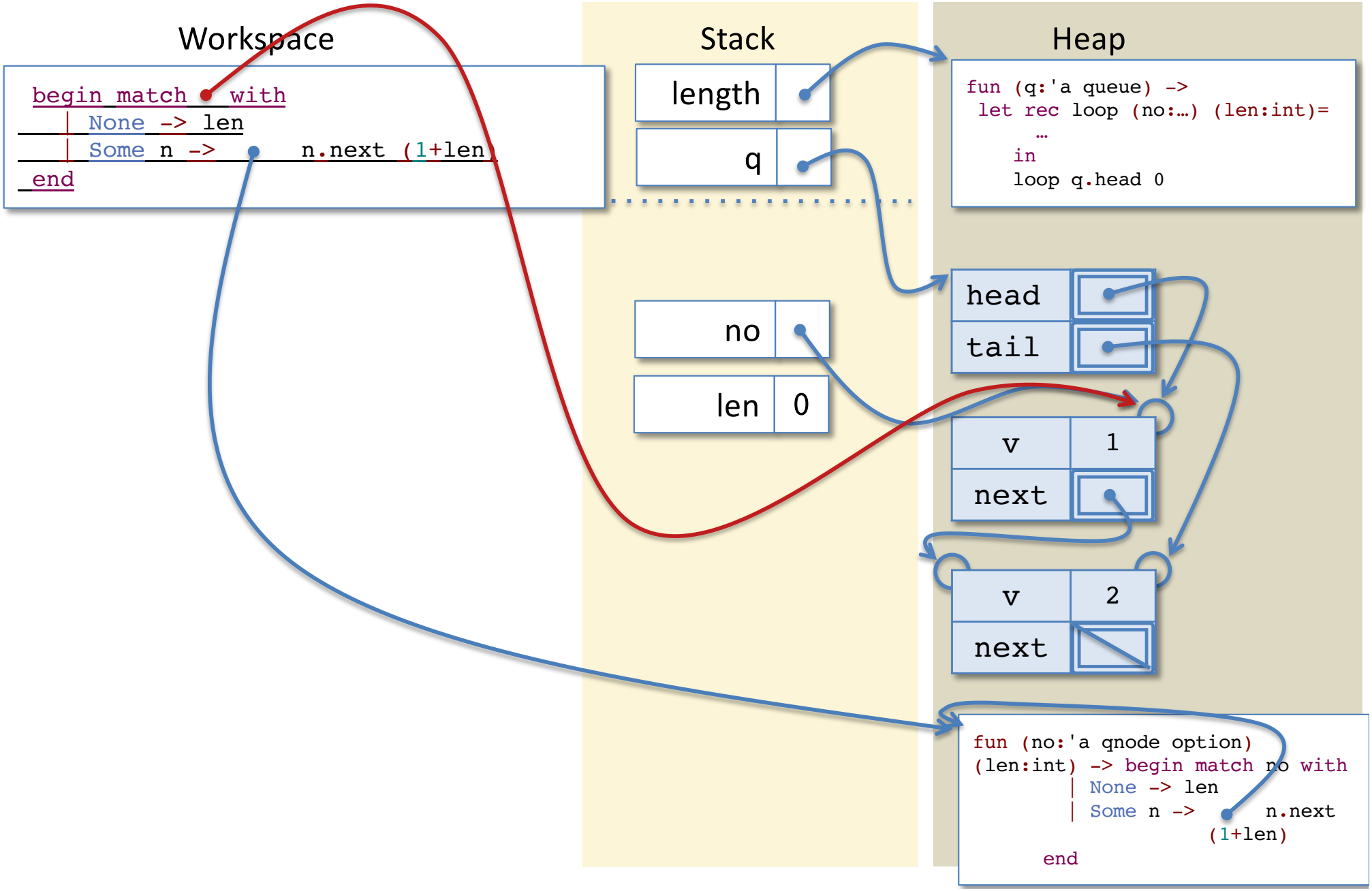
Tail Calls and Iterative length



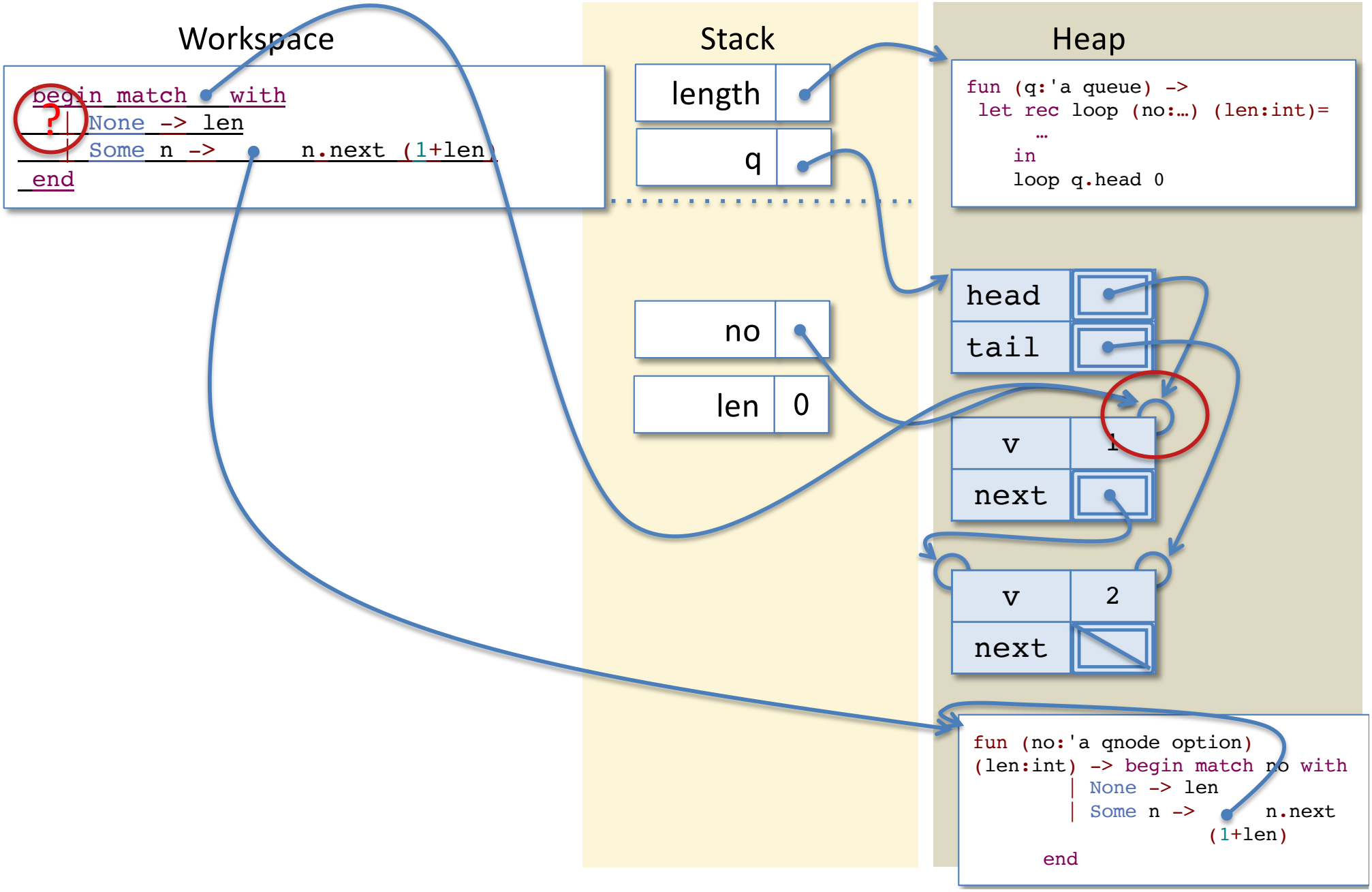
Tail Calls and Iterative length



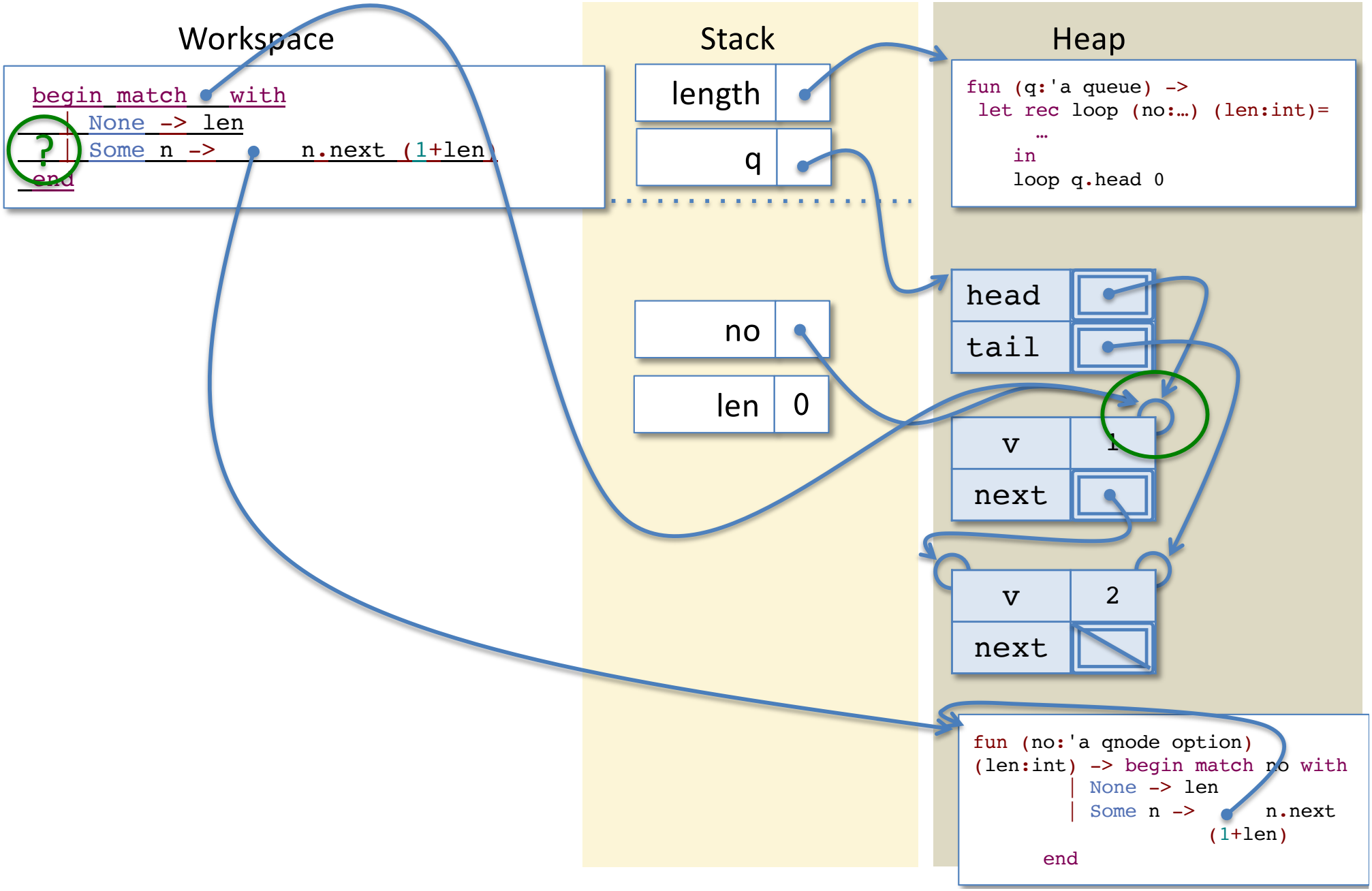
Tail Calls and Iterative length



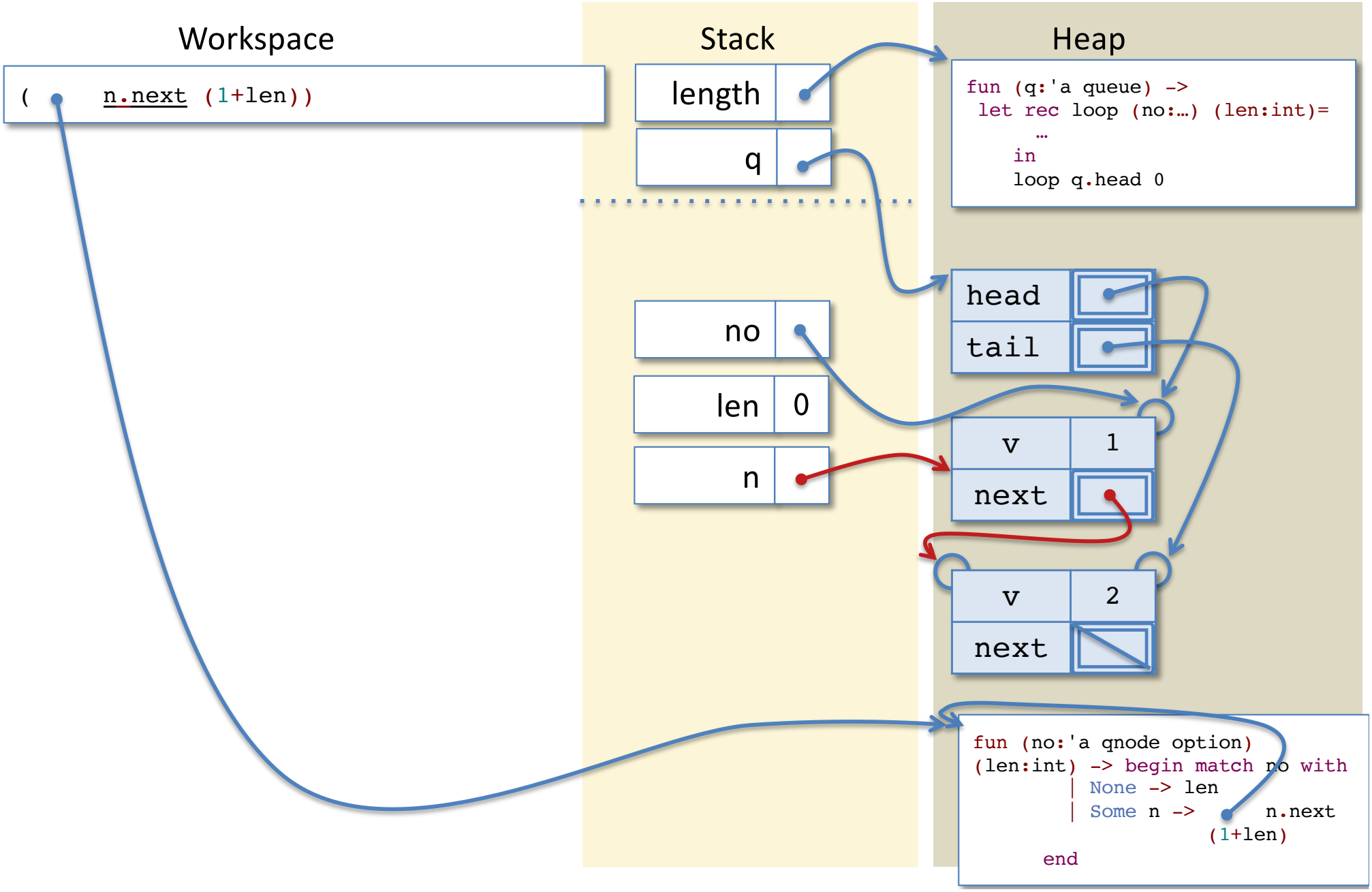
Tail Calls and Iterative length



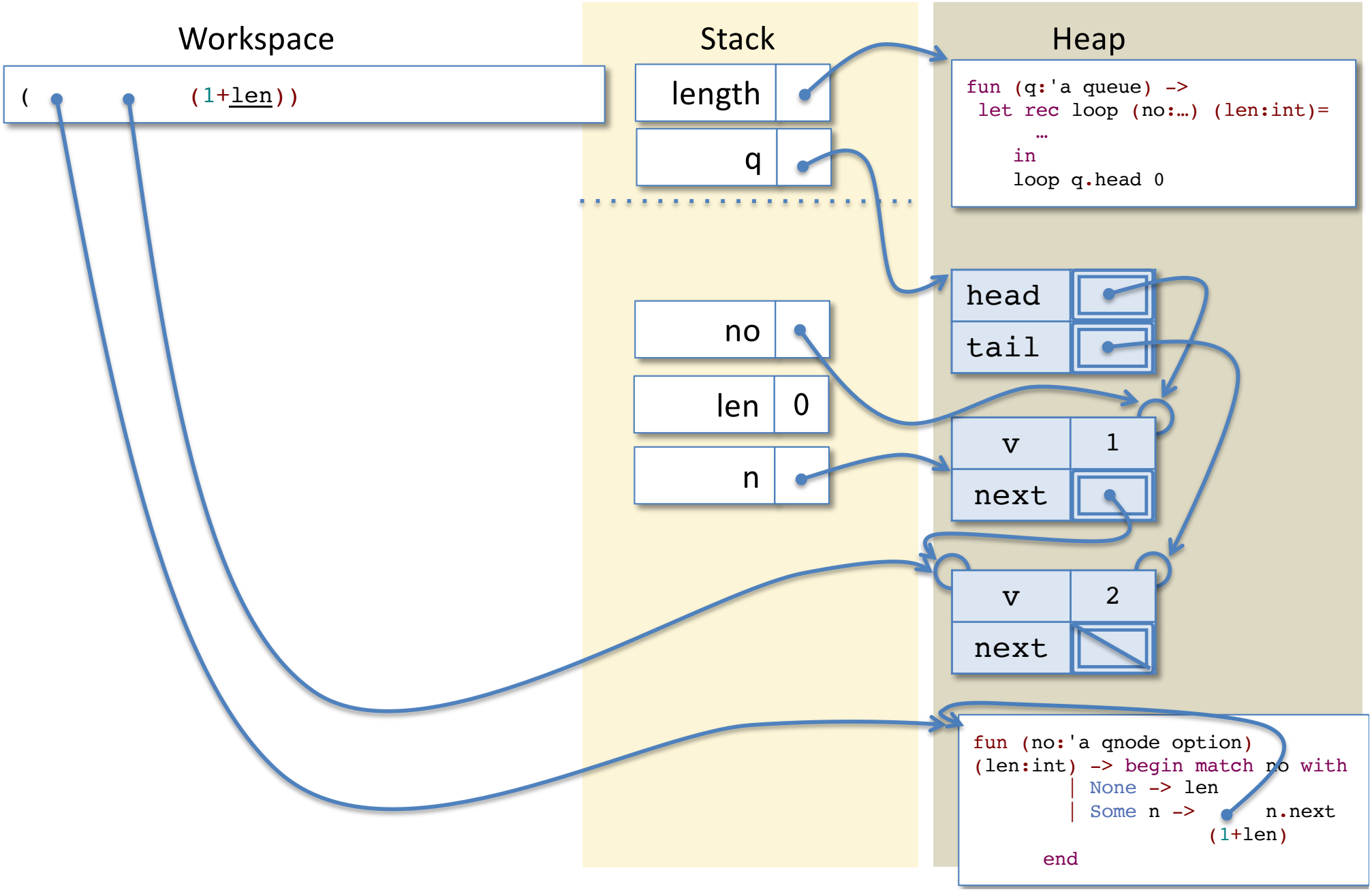
Tail Calls and Iterative length



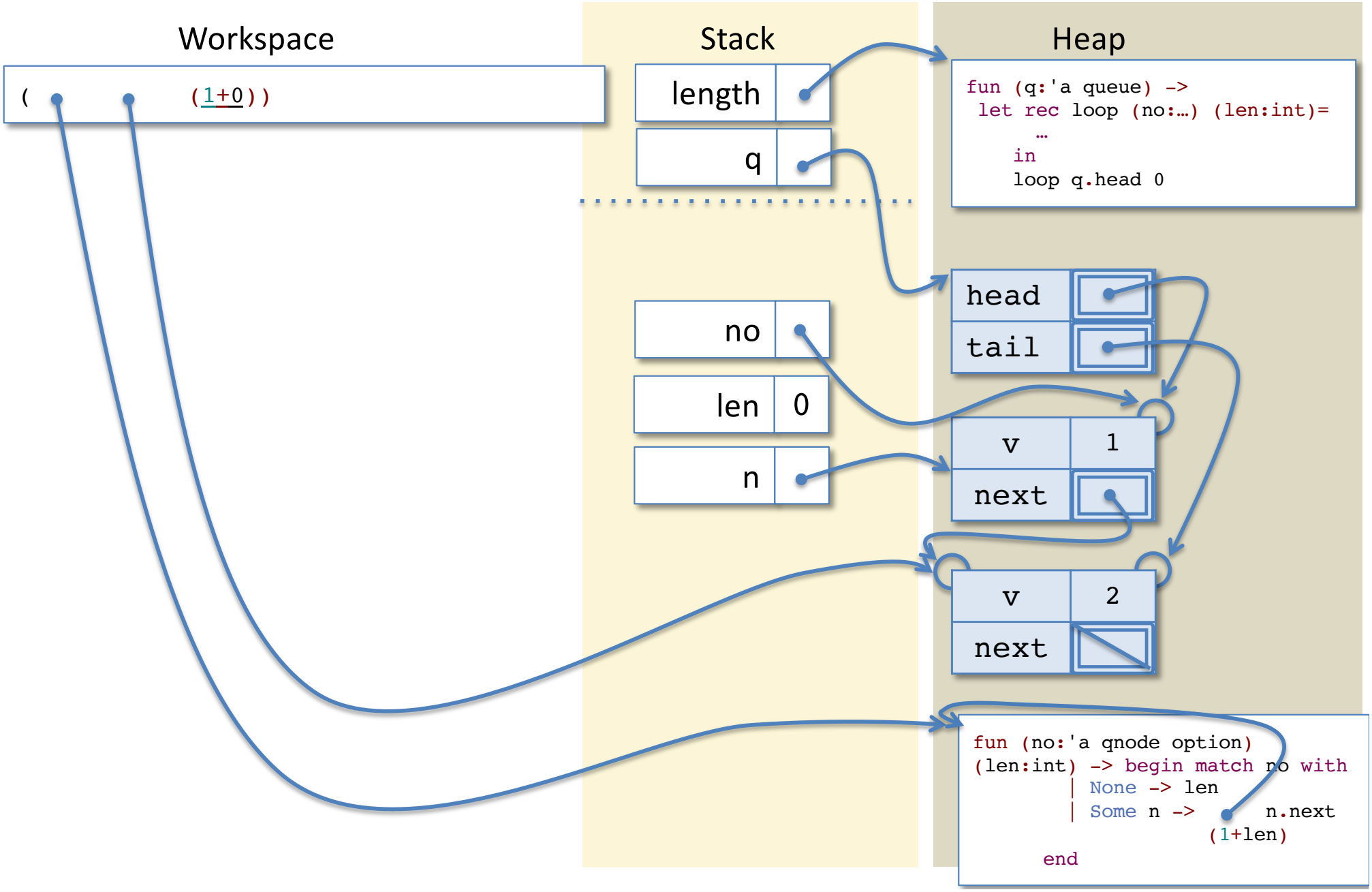
Tail Calls and Iterative length



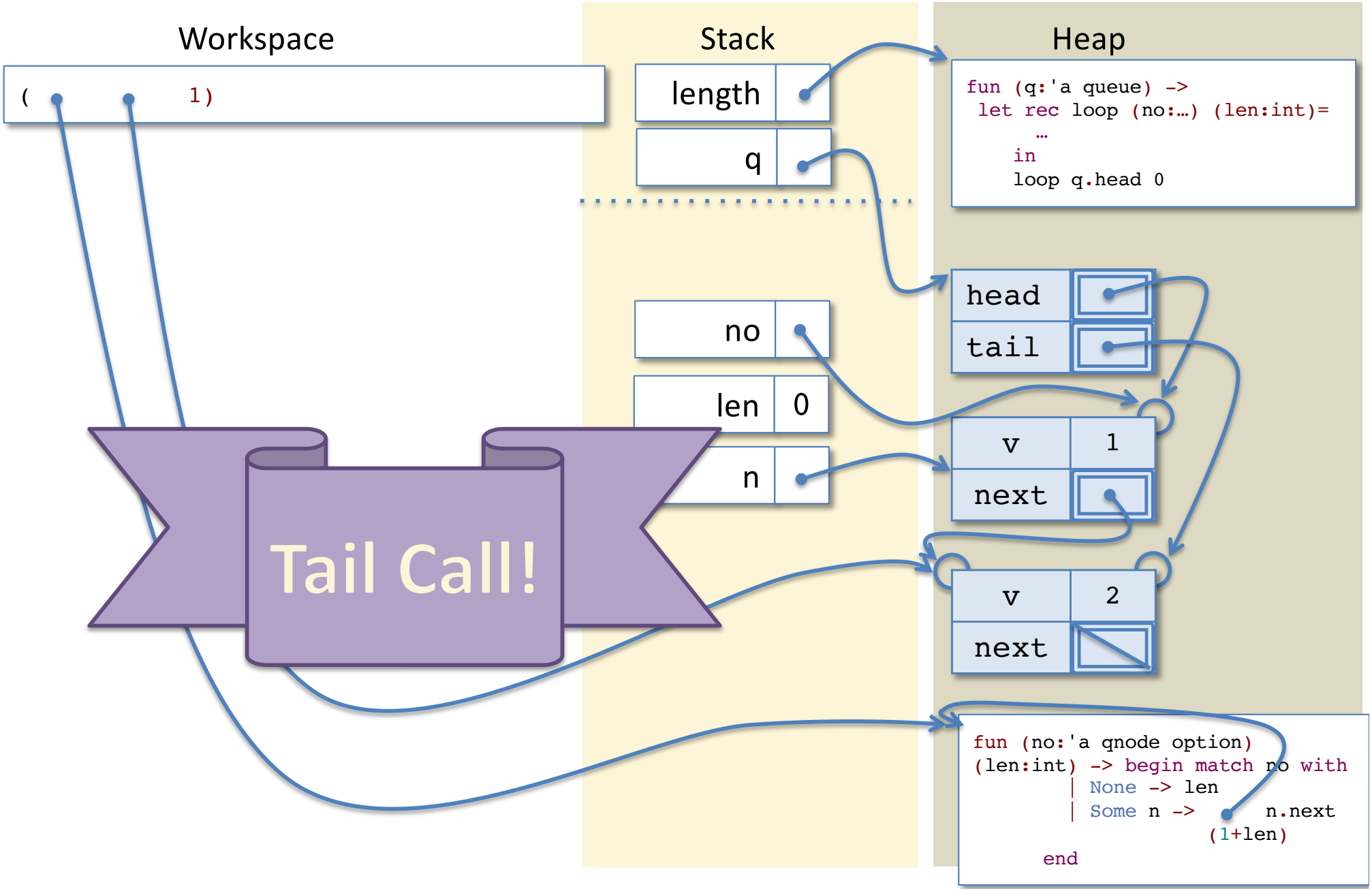
Tail Calls and Iterative length



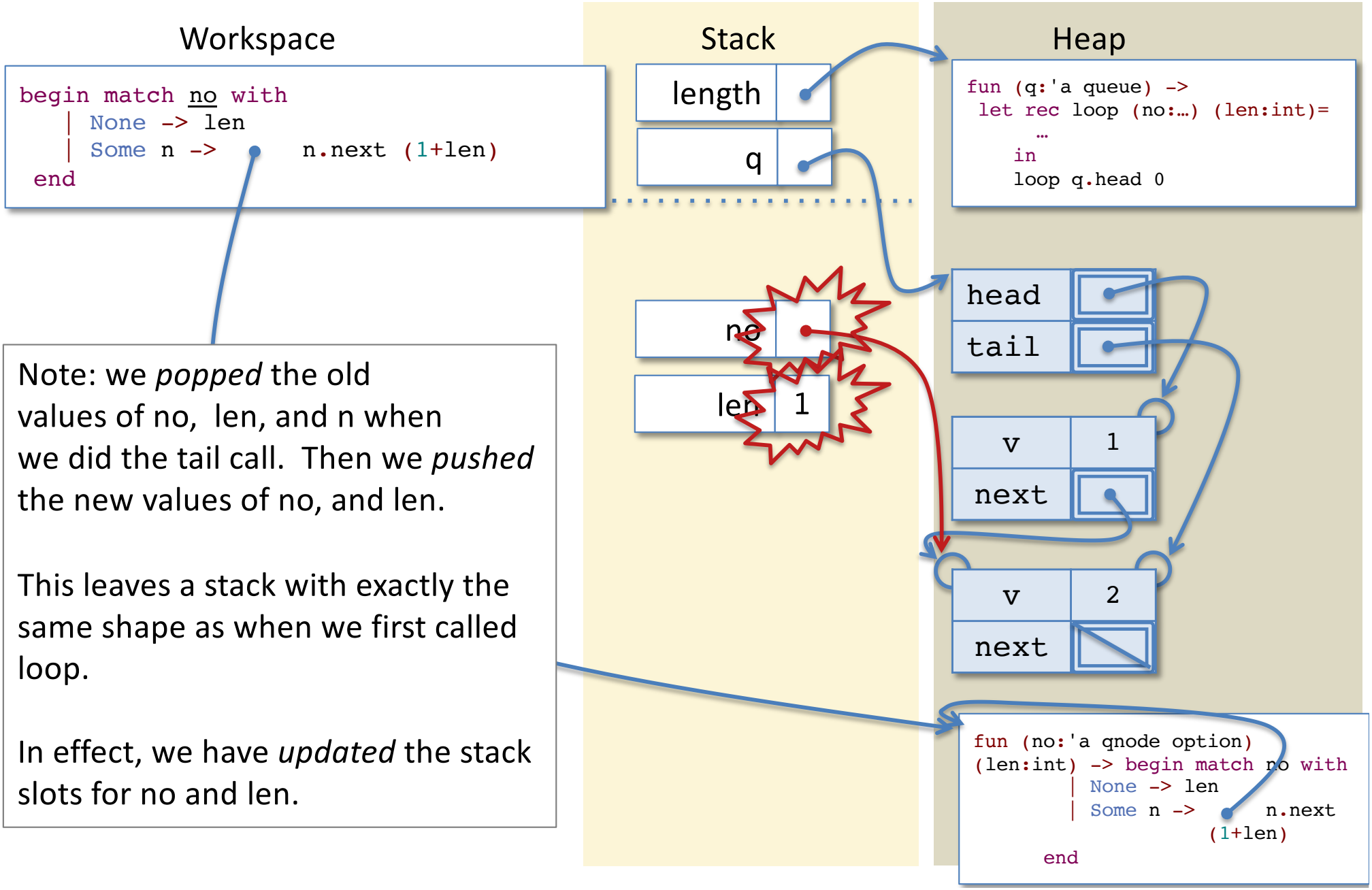
Tail Calls and Iterative length



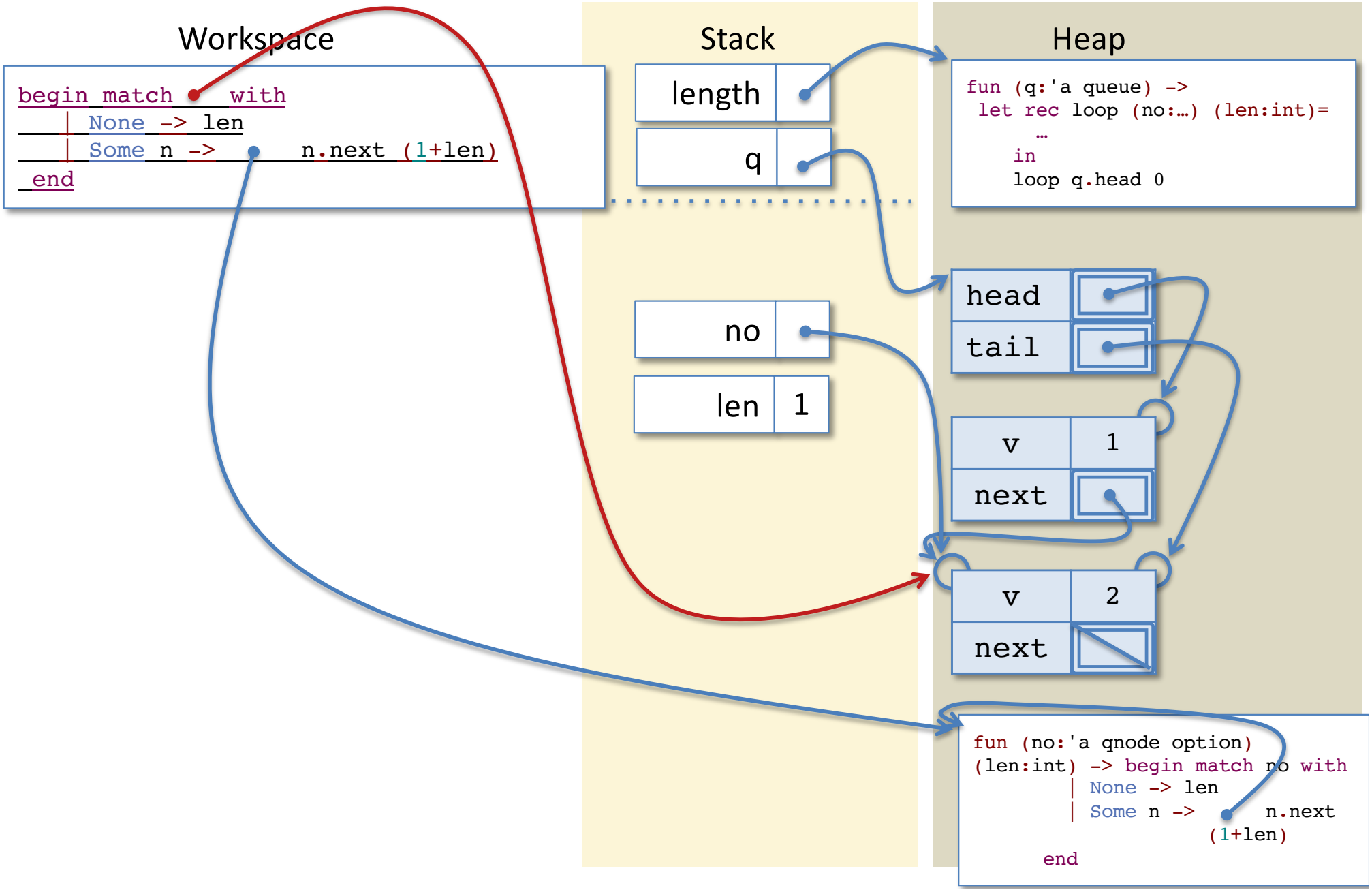
Tail Calls and Iterative length



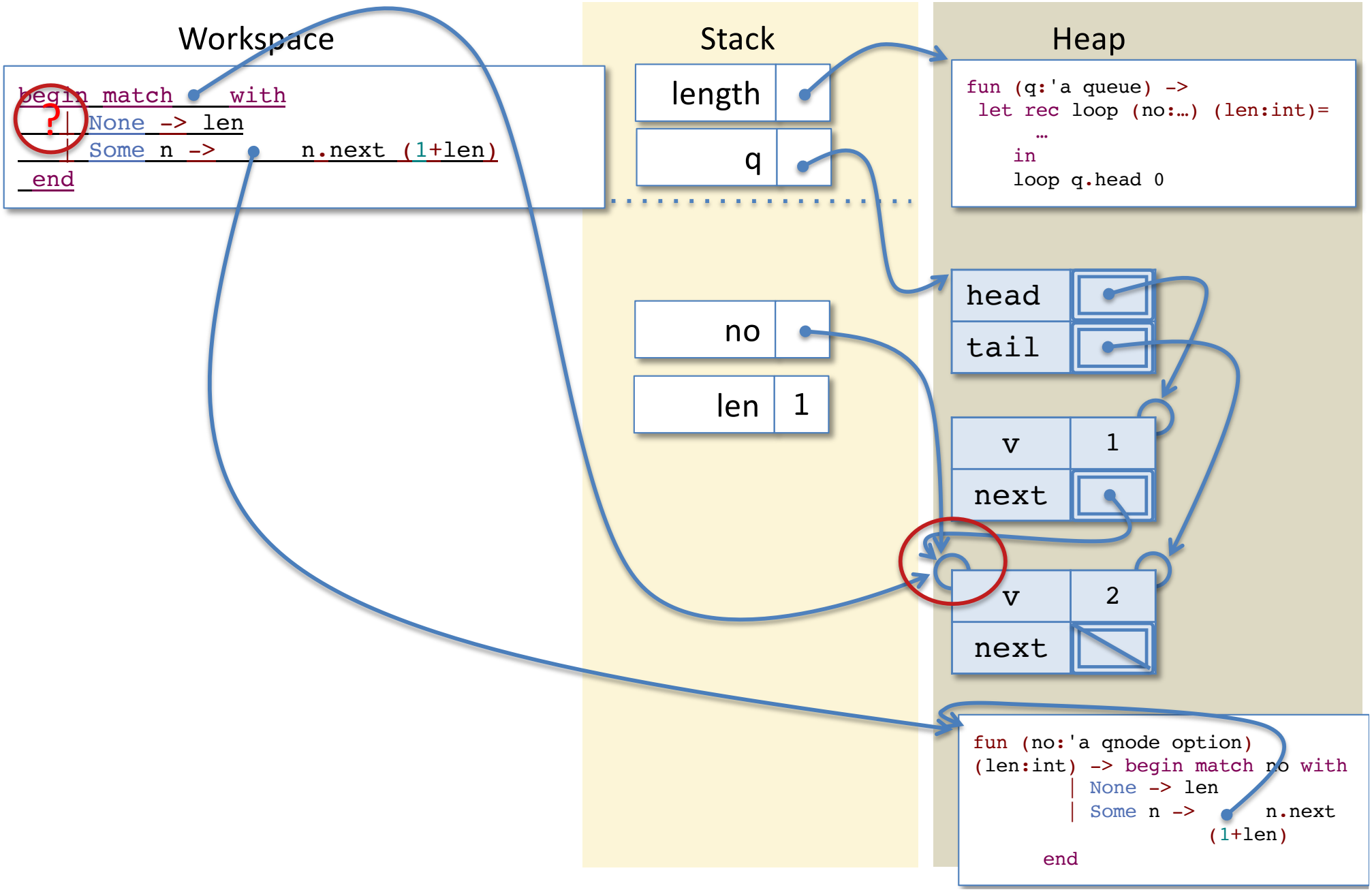
Tail Calls and Iterative length



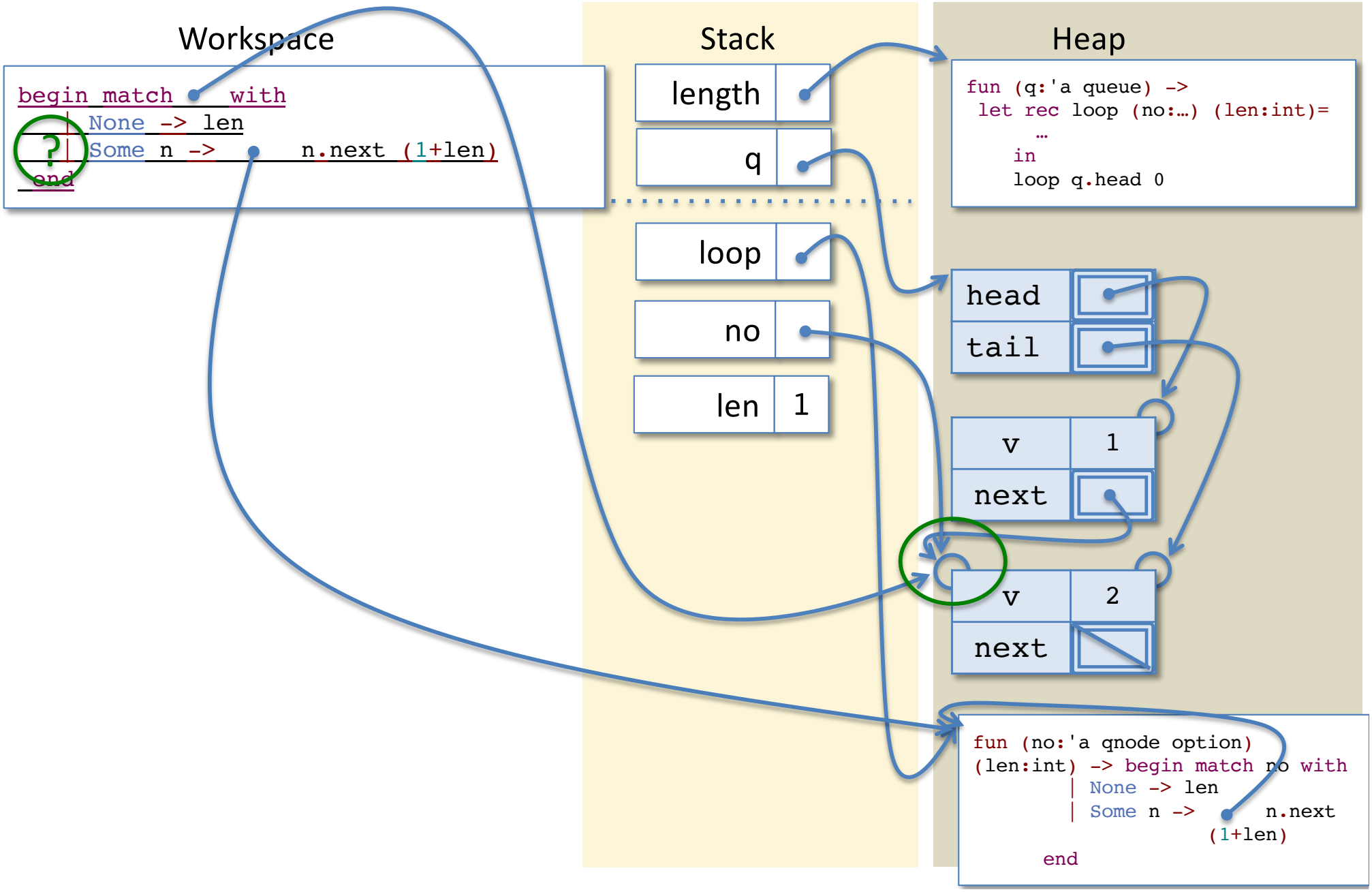
Tail Calls and Iterative length



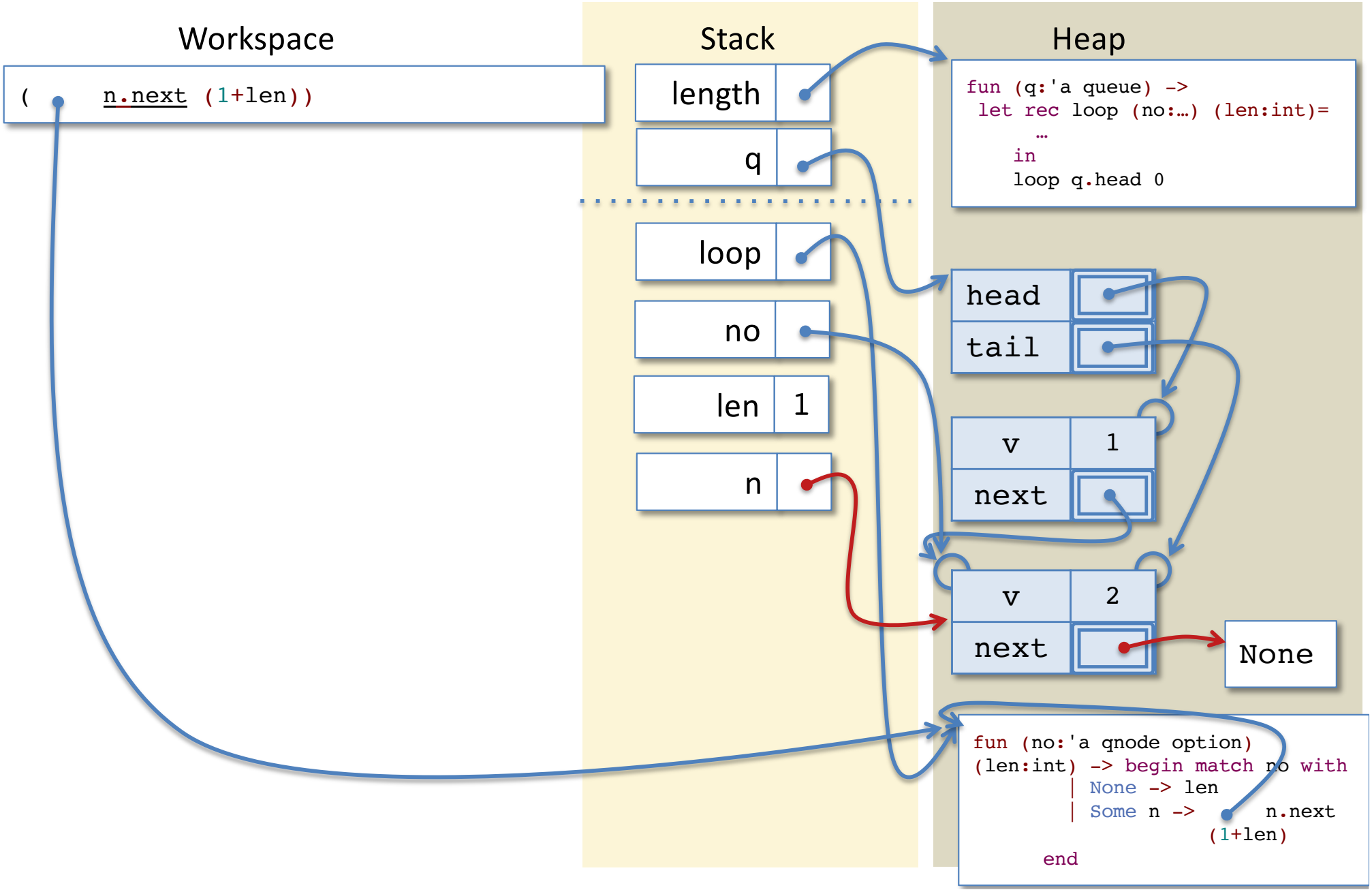
Tail Calls and Iterative length



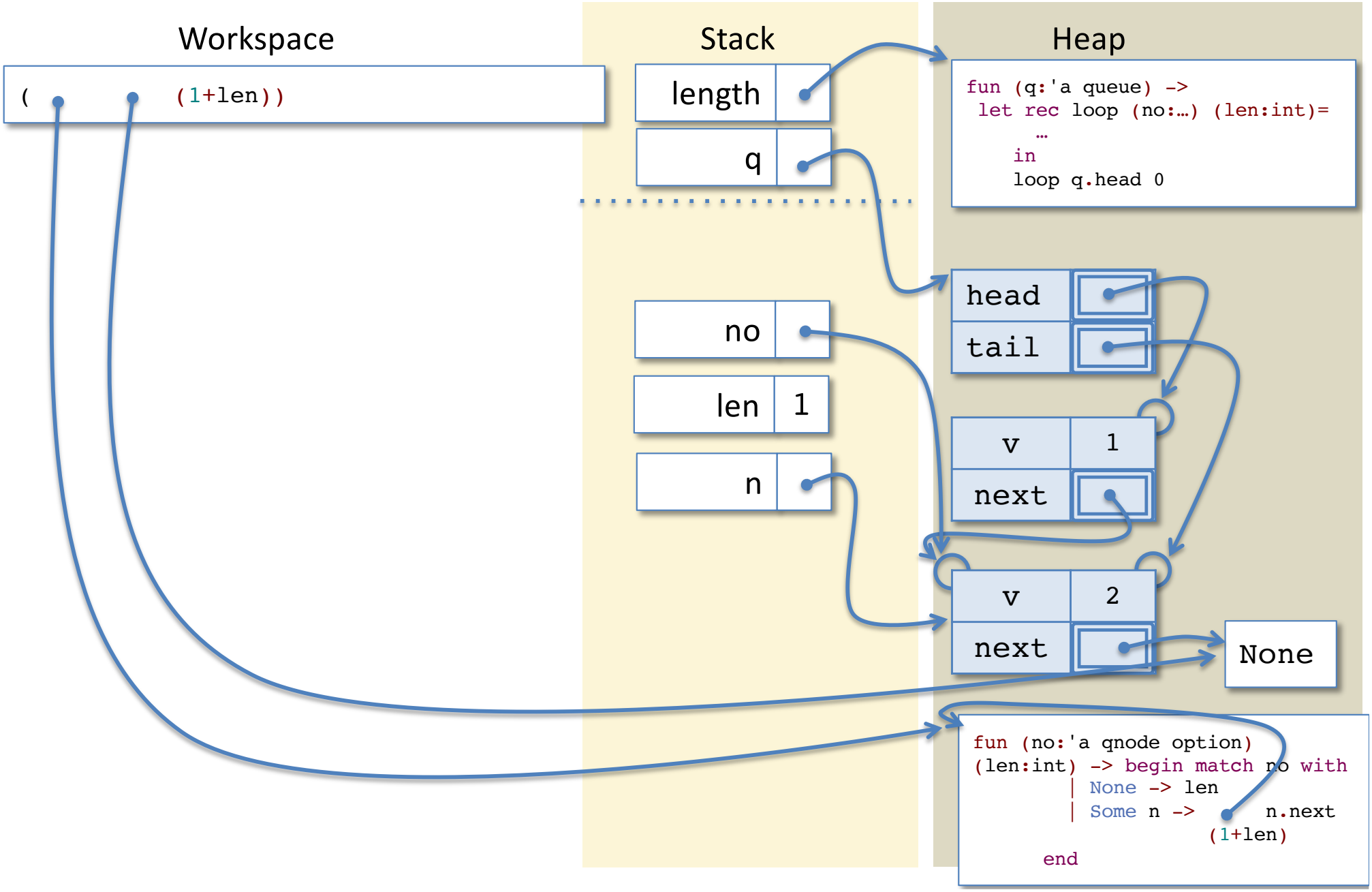
Tail Calls and Iterative length



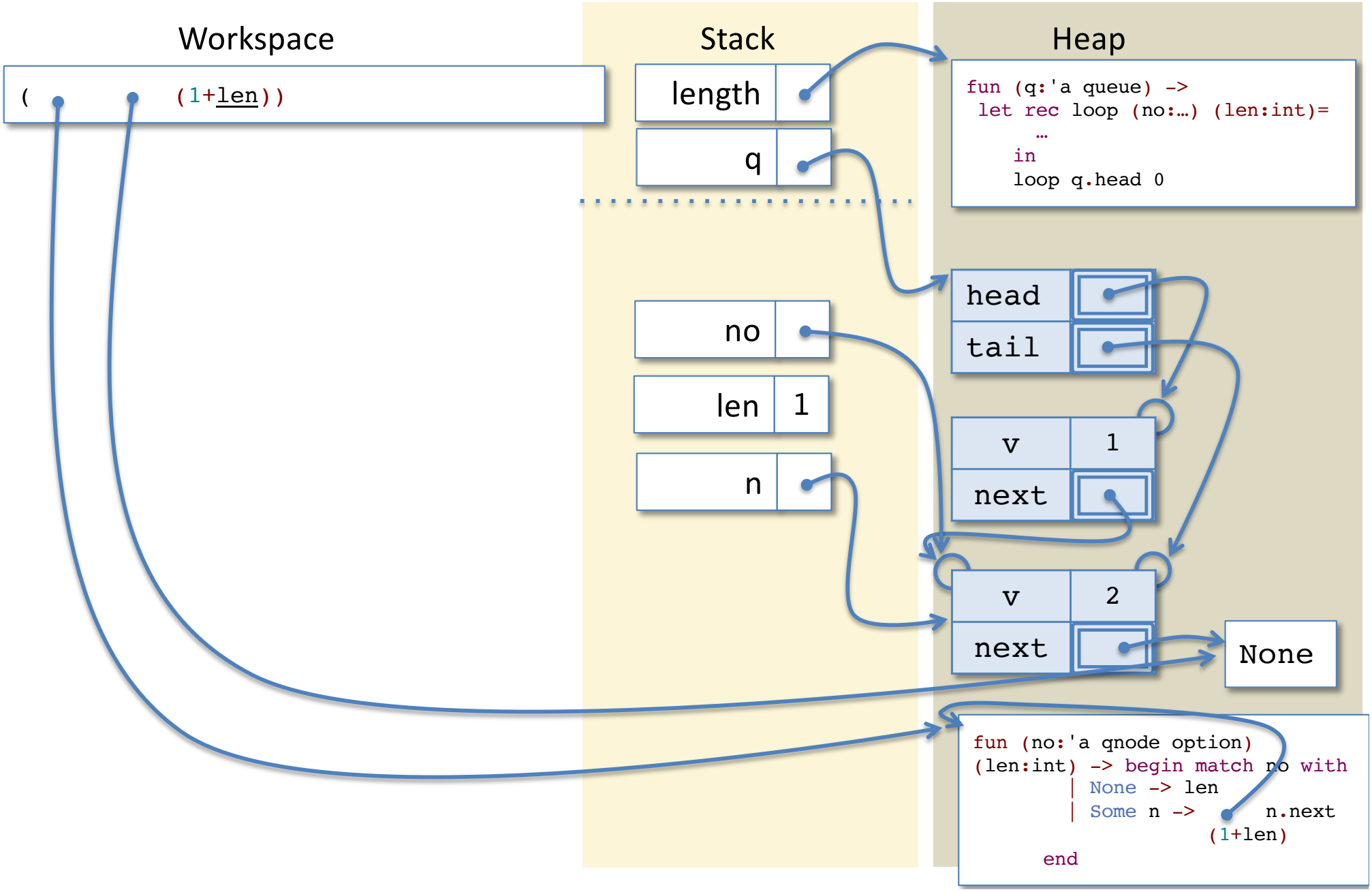
Tail Calls and Iterative length



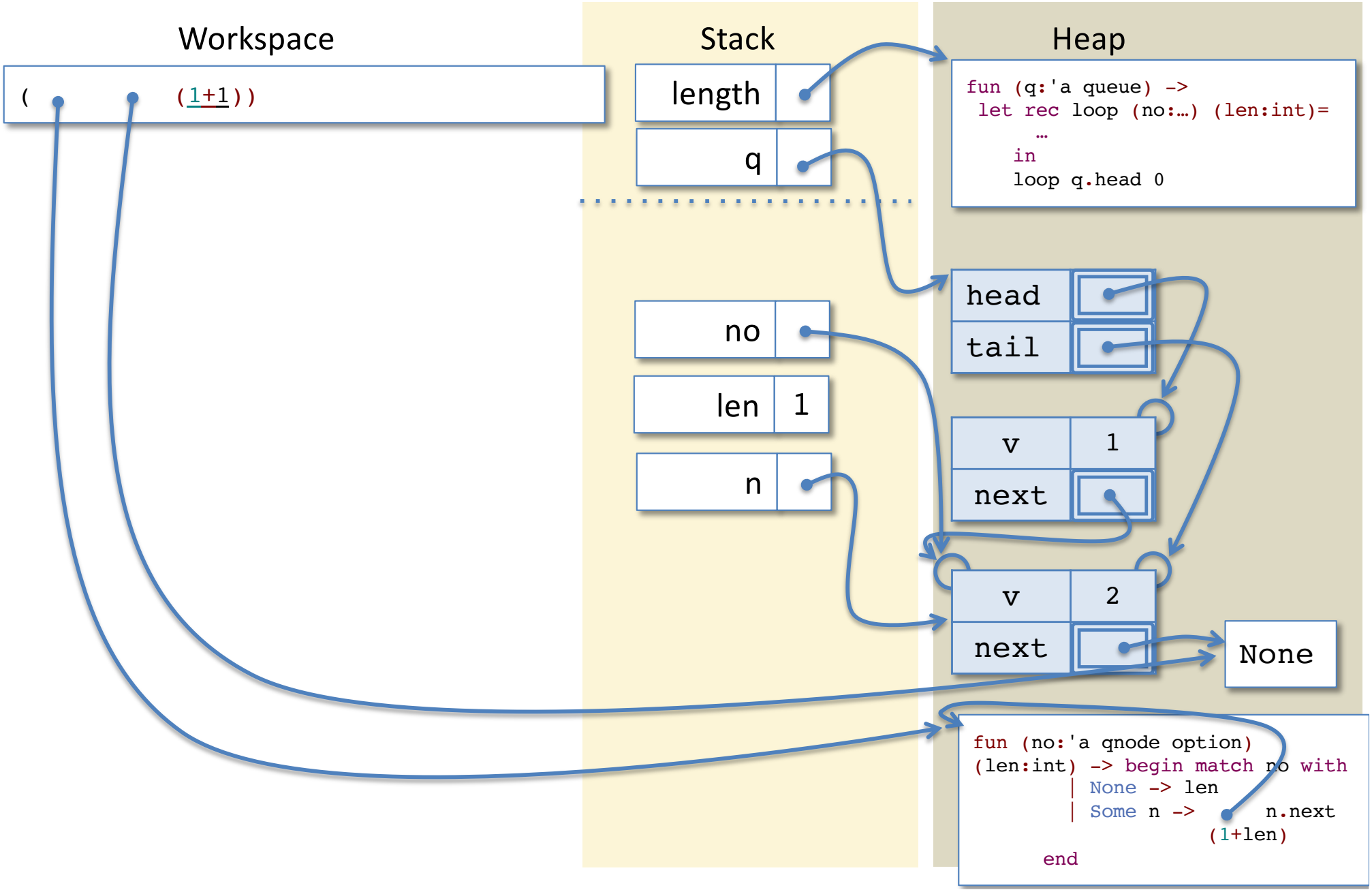
Tail Calls and Iterative length



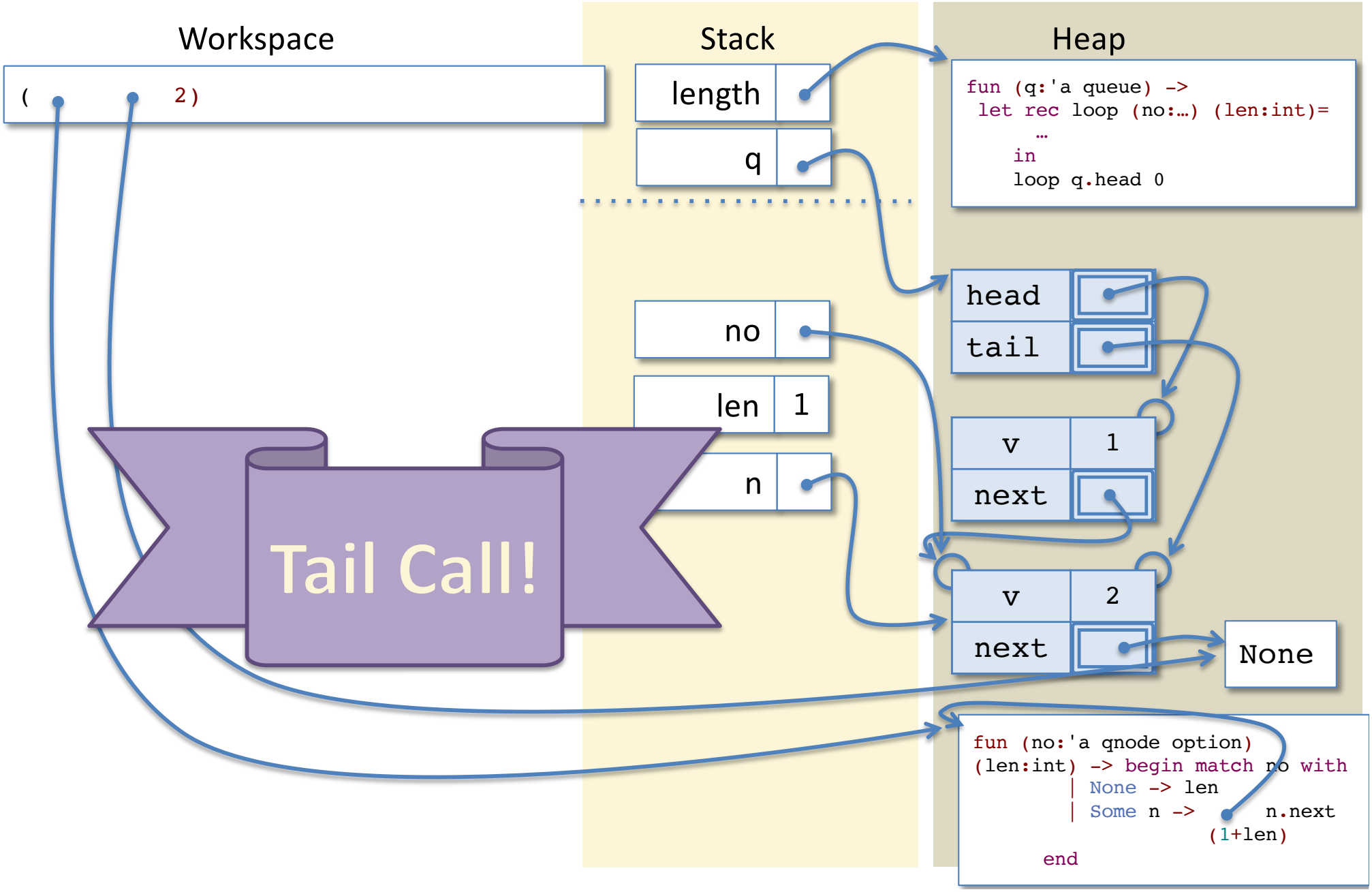
Tail Calls and Iterative length



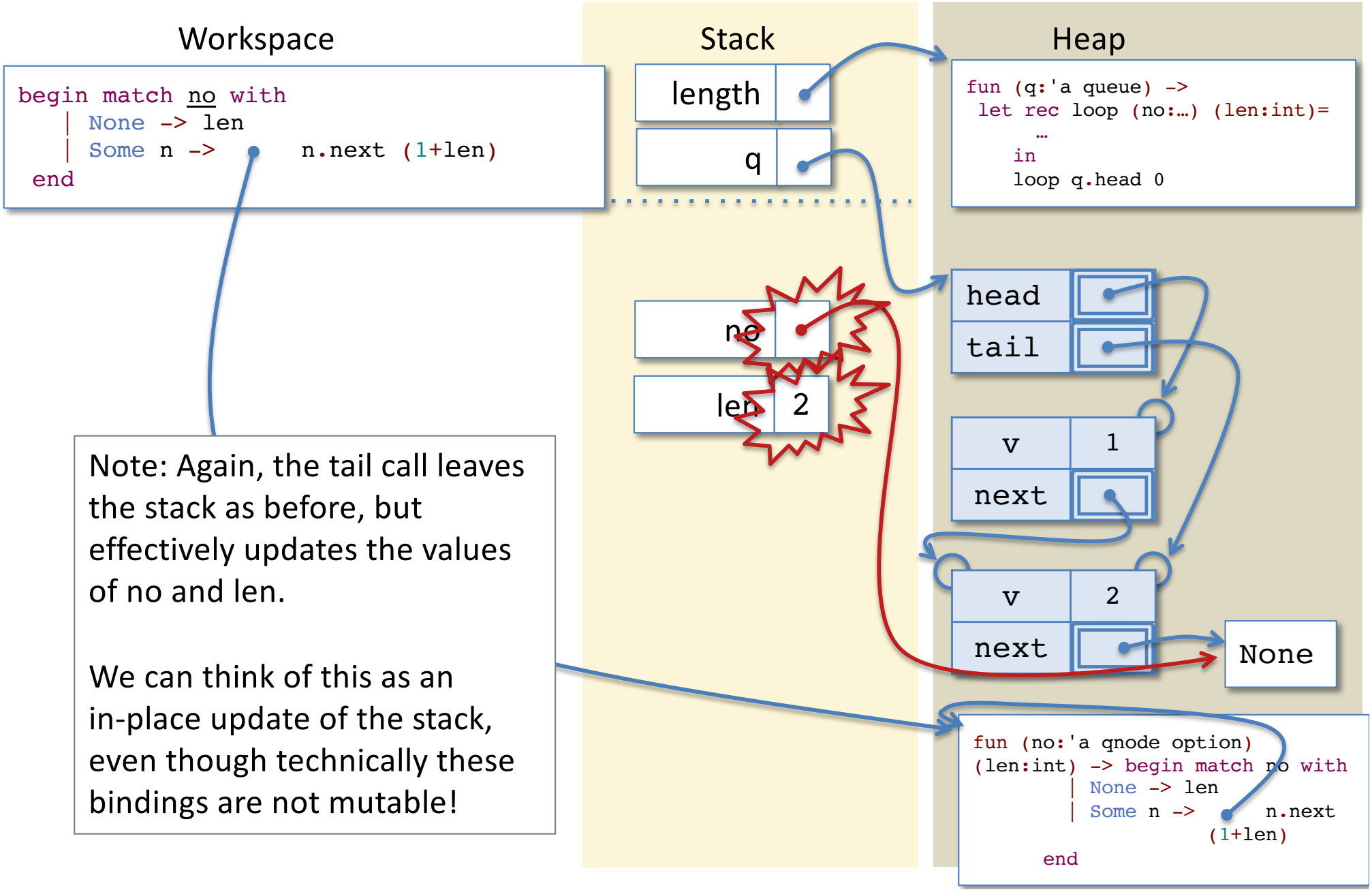
Tail Calls and Iterative length



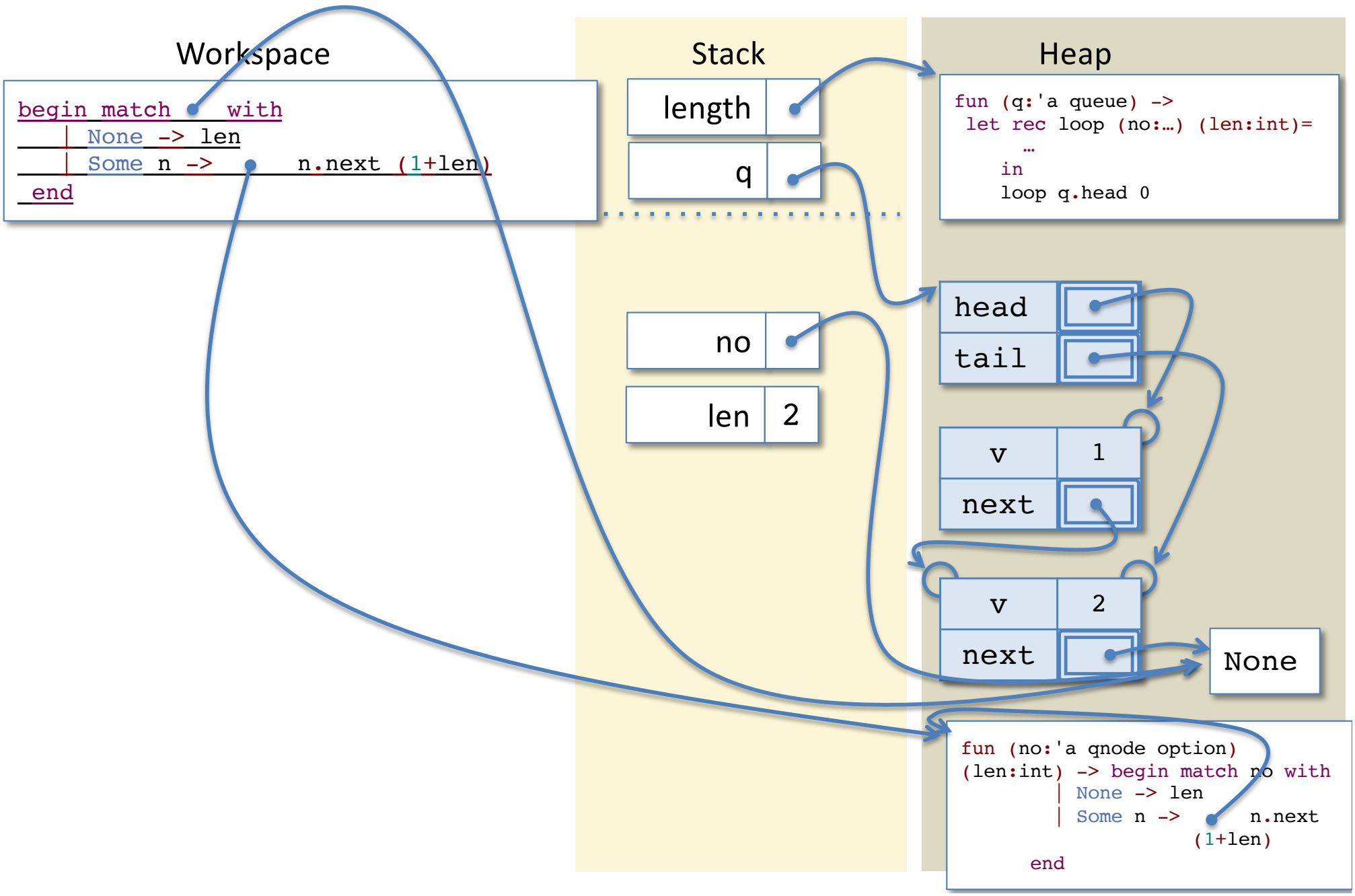
Tail Calls and Iterative length



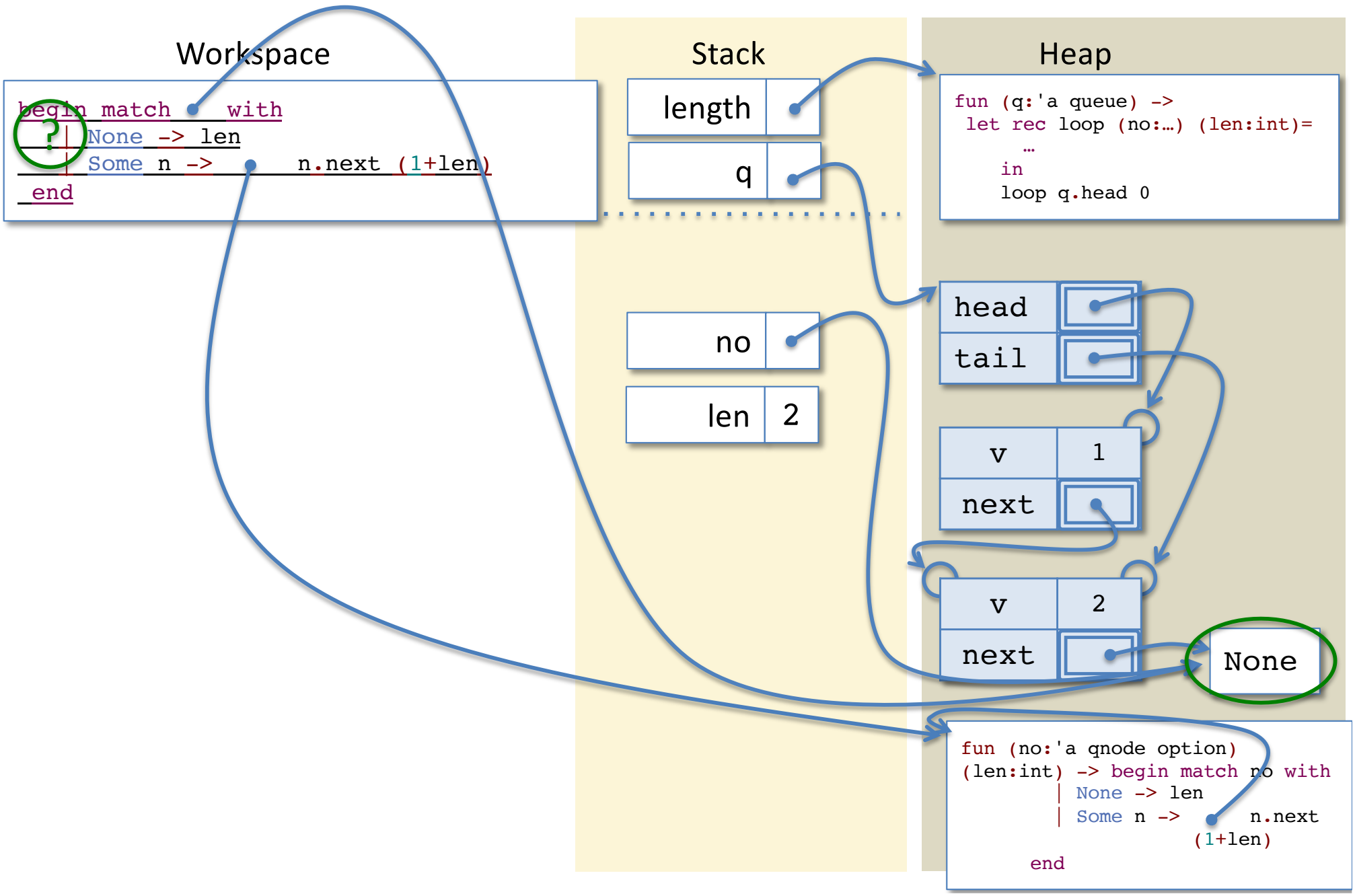
Tail Calls and Iterative length



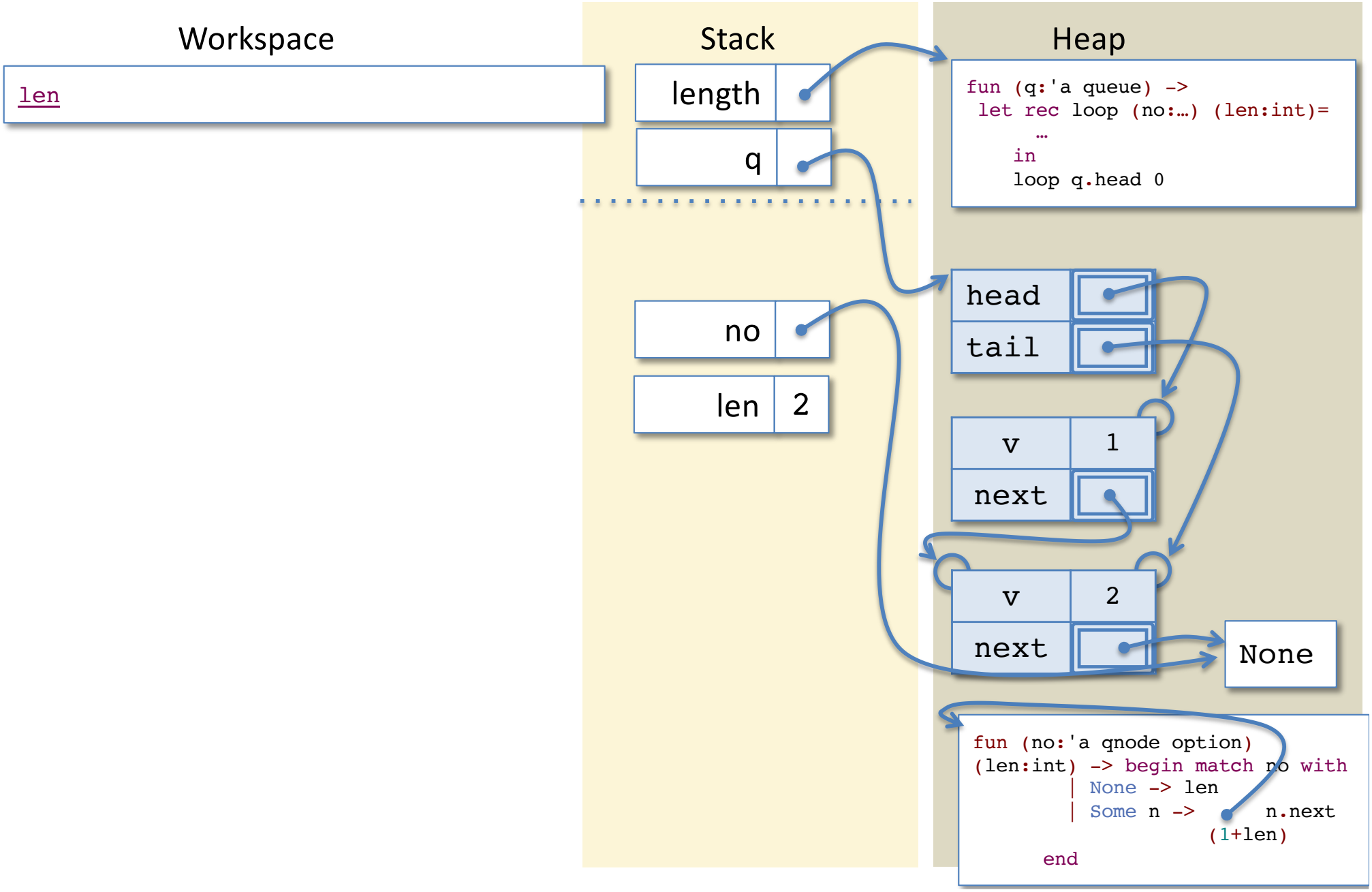
Tail Calls and Iterative length



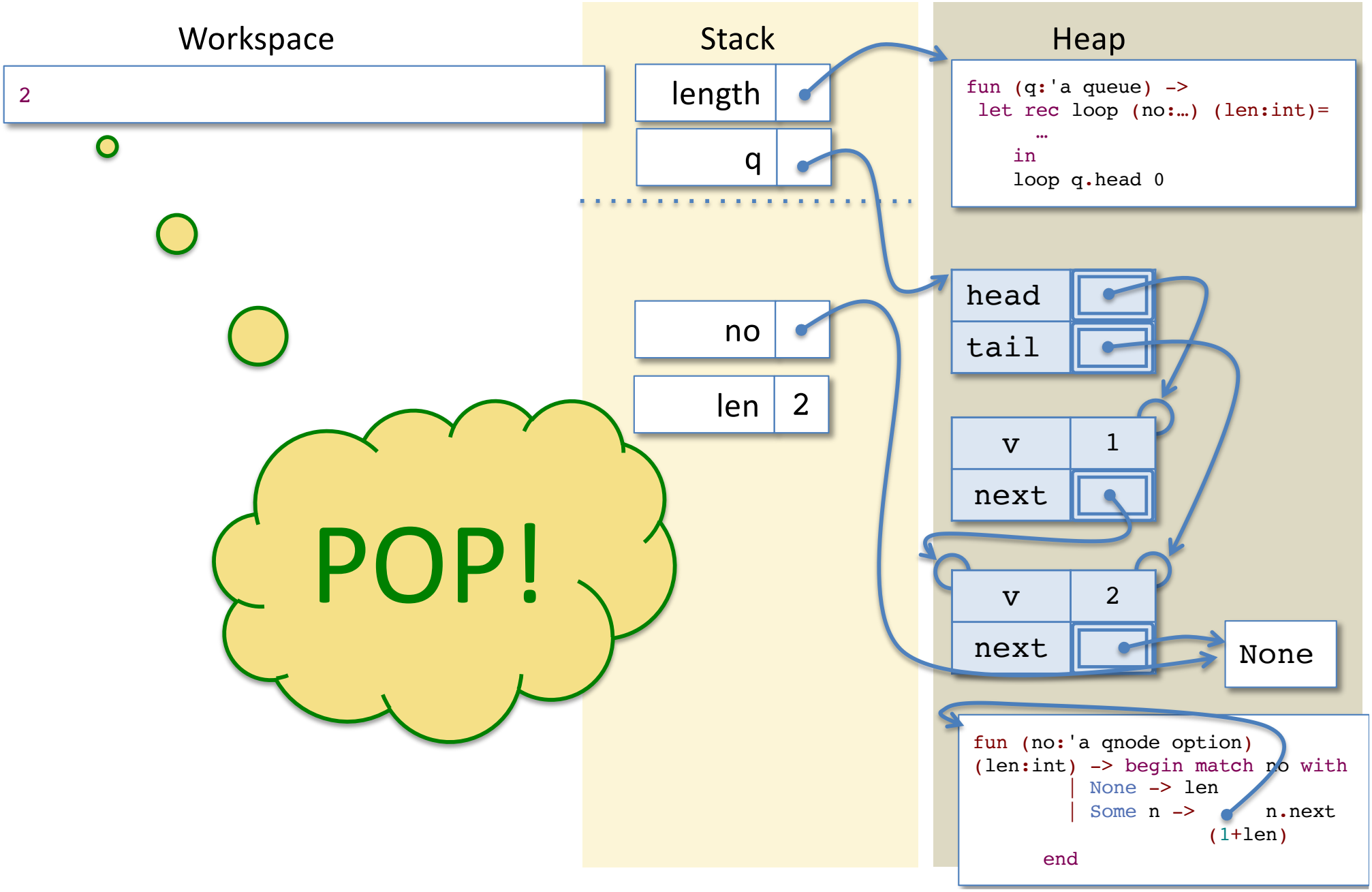
Tail Calls and Iterative length



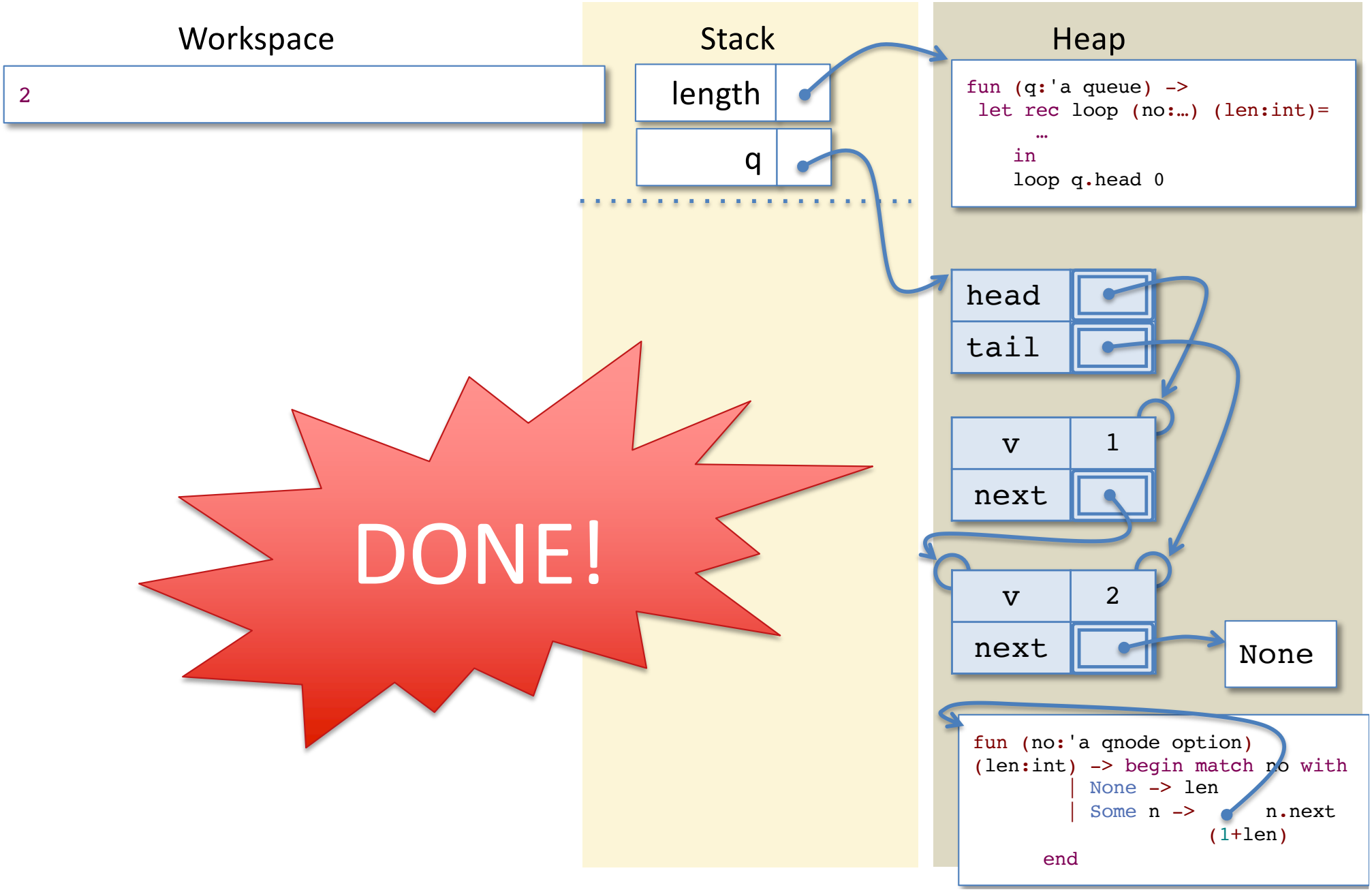
Tail Calls and Iterative length



Tail Calls and Iterative length



Tail Calls and Iterative length



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
```

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 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)