Programming Languages and Techniques (CIS120)

Lecture 16

October 7th , 2015

Iteration for Linked Queues

Lecture notes: Chapter 16

Announcements

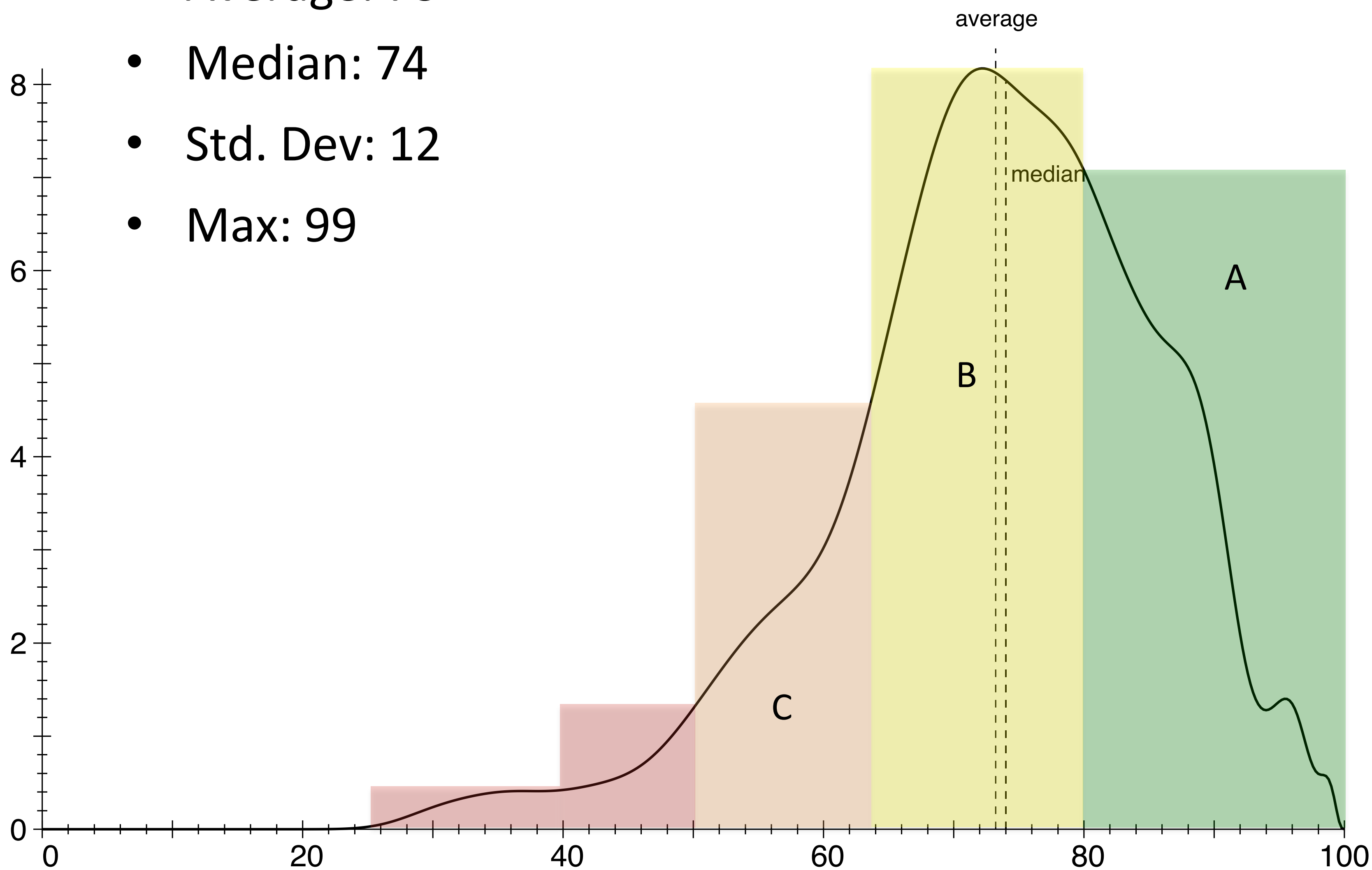
- No Labs today or tomorrow (Fall Break)

- HW 4: Queues
  - Due Tuesday, October 13th at 11:59pm

- Midterm Exam
  - Graded: Available for you to examine, copy, etc. in Levine 308
  - Solutions (and blank) on the web site

Midterm 1 Results

- Average: 73
- Median: 74
- Std. Dev: 12
- Max: 99



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

```
((fun (x:int) (y:bool) -> 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 $f : T_1 \text{ -> } T_2$
   - and an argument $e : T_1$ of the input type
   - $(f\ e) : T_2$

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

                                 bool

      int -> bool -> ??

                ??
```

Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:
   - Given a function $f : T_1 \text{->} T_2$
   - and an argument $e : T_1$ of the input type
   - $(f\ e) : T_2$

```
((fun (x:int) (y:bool) -> y)  3)   : ??
                                bool
     int -> bool -> 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 -> T_2
   - and an argument e : T_1 of the input type
   - (f e) : T_2

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

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 -> T_2
   - and an argument e : T_1 of the input type
   - (f e) : T_2

$$\underbrace{(\underbrace{(\texttt{fun (x:int) (y:bool) } \underbrace{\texttt{-> y}}_{\texttt{bool}})}_{\texttt{int -> bool -> bool}}\ \ \underbrace{\texttt{3)}}_{\texttt{int}}}_{\texttt{bool -> bool}} \quad \texttt{: ??}$$

Here:
T_1 = `int`
T_2 = `bool -> bool`

Typechecking III

- For generic expressions:
  - *Unify* the types based on use:
  - Given a function f : T_1 -> T_2
  - and an argument e : U_1 of the input type
    - *"match up"* T1 and U1 to obtain information about type parameters in T_1 and U_1 based on the 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)
```

$\underbrace{\texttt{fun (x:'v) -> entries (add 3 x empty)}}_{\texttt{'v -> ??}}$

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)
```

```
‘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 -> ??
```

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

‘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)
                             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)
```

int ‘v (‘k,‘v) map

T_2

Application:
T_1 = ‘k
T_2 = ‘v -> (‘k,’v) map -> (‘k,’v) map

Instantiate: ‘k = 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)
```

int ‘v (int,‘v) map

T_2

T'_2

Another Application:

T'_1 = ‘v

T'_2 = (int,’v) map -> (int,’v) map

Instantiate: ‘v = ‘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)
```

`int` `‘v` `(int,‘v) map`

T_2

T'_2

T''_2= (int,’v) map

A third Application:
T''_1 = (int,’v) map
T''_2 = (int,’v) 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
```

$$\underbrace{\texttt{fun (x:'v) -> } \underbrace{\texttt{entries}}_{U_1 \to U_2}\ \underbrace{\texttt{(}\underbrace{\underbrace{\underbrace{\texttt{add}}\ \underbrace{\texttt{3}}_{\texttt{int}}}_{T_2}\ \underbrace{\texttt{x}}_{\texttt{'v}}}_{T'_2}\ \underbrace{\texttt{empty}}_{\texttt{(int,'v) map}}\texttt{)}}_{T''_2 = (int,'v)\ map}}_{\texttt{'v -> ??}}$$

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 (int,‘v) map

T_2

T'_2

U_1 -> U_2

T''_2= (int,’v) map

??

```
‘v -> ??
```

Another Application:
U_1 = (‘k,’v) map
U_2 = (‘k * ‘v) list

Unify U_1 with T''_2
(‘k,’v) map ~~ (int, ‘v) map

Instantiate ‘k = int

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   (int,'v) map

T_2

T'_2

U_1 -> U_2

T''_2= (int,'v) map

U_2= (int * 'v) list

'v -> ??

Another Application:
U_1 = (int,'v) map
U_2 = (int * '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
```

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

int ‘v (int,‘v) map

T_2

T'_2

U_1 -> U_2

T''_2= (int,’v) map

U_2= (int * ‘v) list

```
‘v -> (int * ‘v) list
```

Another Application:

U_1 = (int,’v) map

U_2 = (int * ‘v) 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`

Mutable Queues: Recap

singly linked data structures

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:

- the “internal nodes” of the queue with links from one to the next
- the head and tail references themselves

All of the references are options so that the queue can be empty (and so that the links can terminate).

Linked Queue Invariants

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

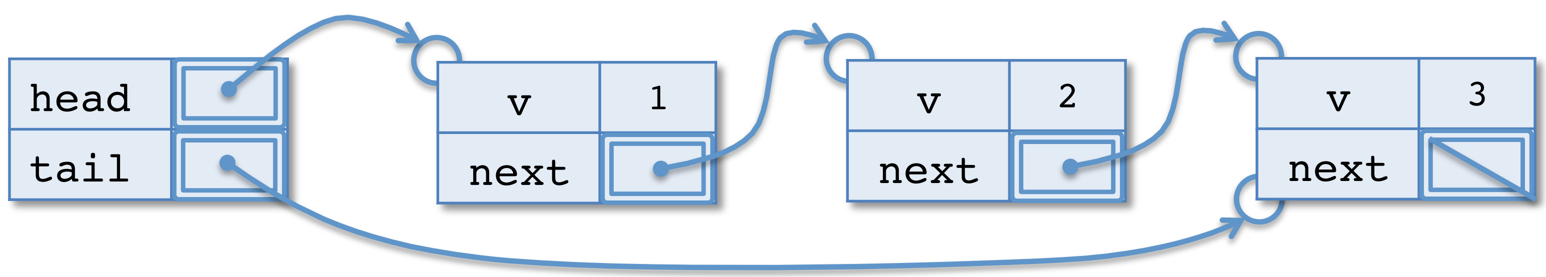
> Either:
> (1) `head` and `tail` are both `None` (i.e. the queue is empty)
> or
> (2) `head` is `Some n1`, `tail` is `Some n2` and
> - `n2` is reachable from `n1` by following 'next' pointers
> - `n2.next` is `None`

- We can check that these properties rule out all of the "bogus" examples.
- A queue operation may assume that these invariants hold of its inputs, and must ensure that the invariants hold when it's done.

Is this a valid queue?

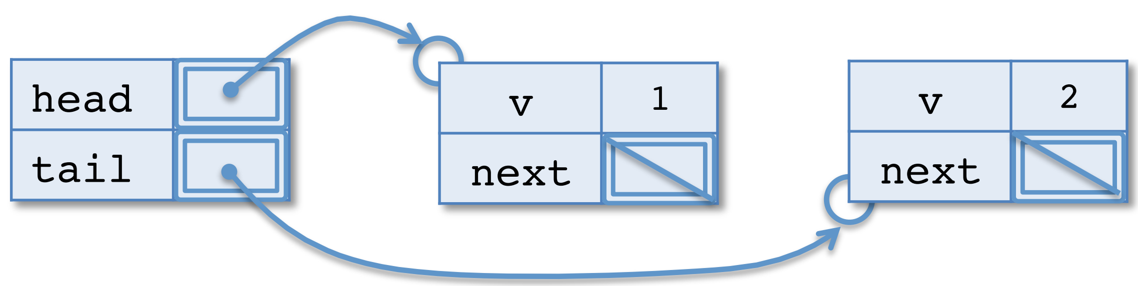
1. Yes

2. No



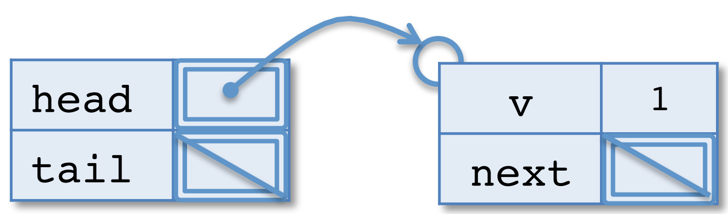
Is this a valid queue?

1. Yes
2. No



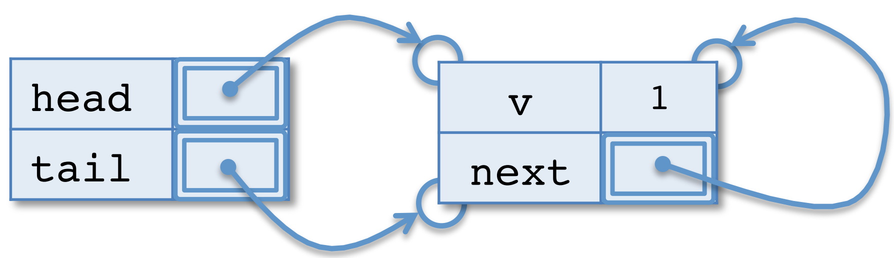
Is this a valid queue?

1. Yes
2. No



Is this a valid queue?

1. Yes
2. No



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.

deq

```
(* remove an element from the head of the queue *)
 let deq (q: 'a queue) : 'a =
   begin match q.head with
     | None ->
         failwith "deq called on 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 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)

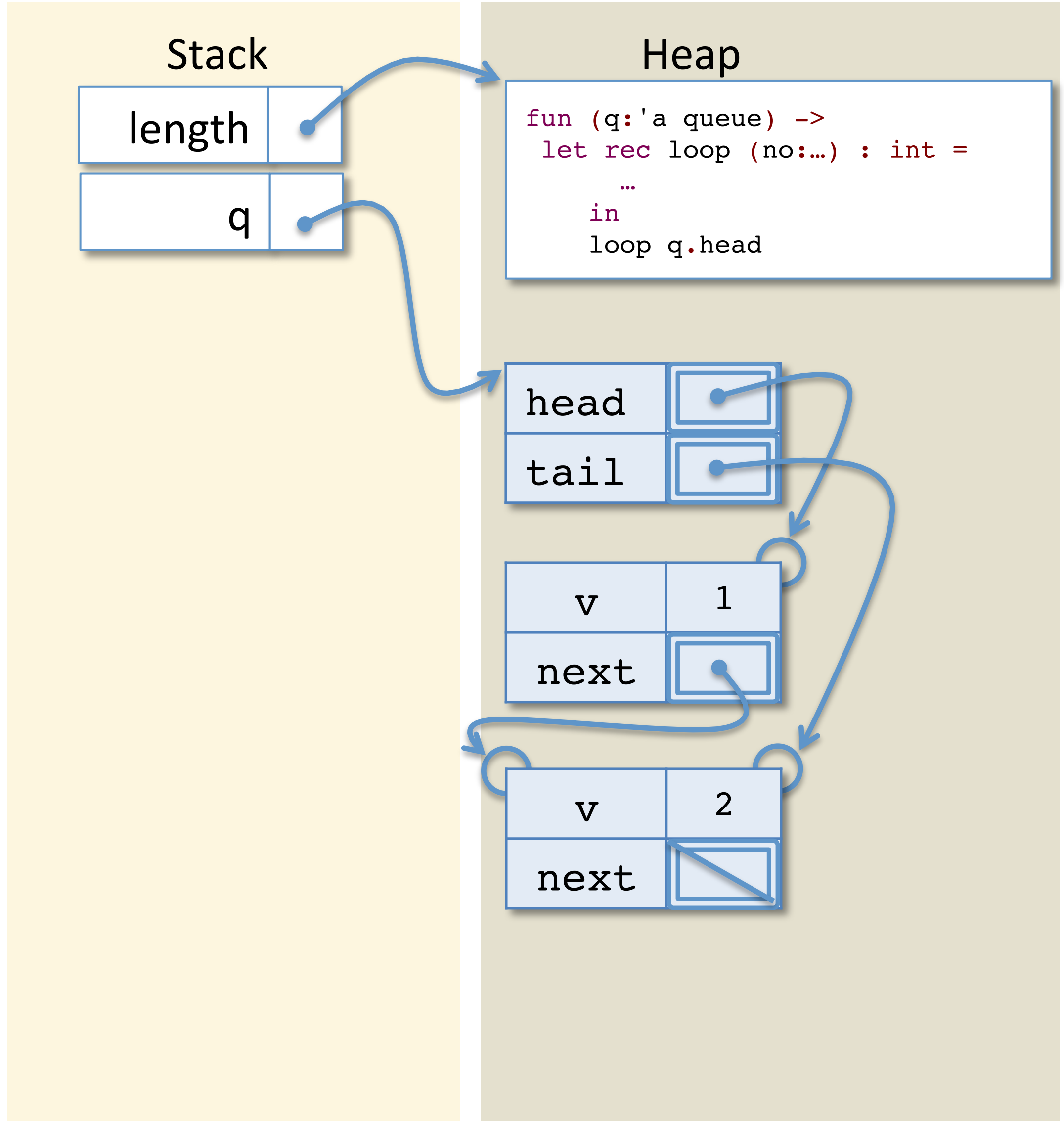
```
(* Calculate the length of the queue recursively *)
let length (q:'a queue) : int =
  let rec loop (no: 'a qnode option) : int =
    begin match no with
      | None -> 0
      | Some n -> 1 + (loop n.next)
    end
  in
  loop q.head
```

- 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

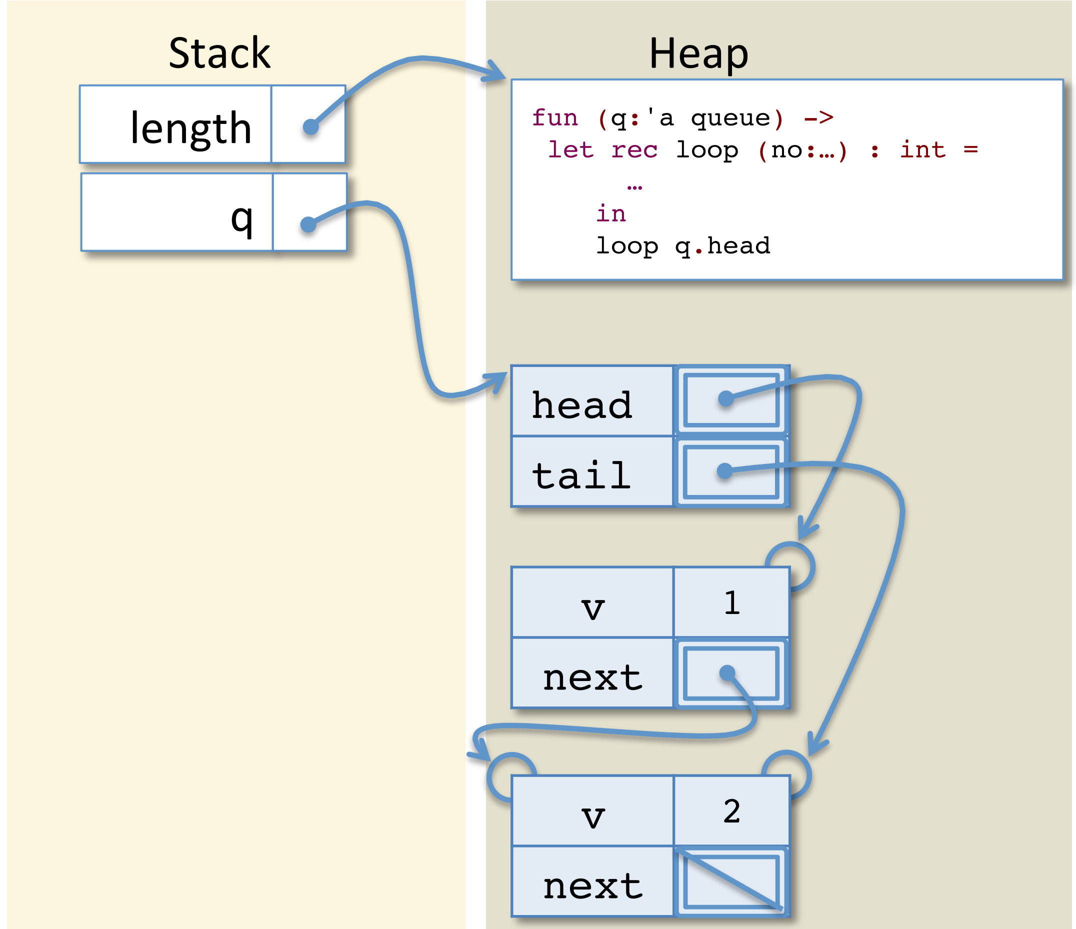
```
length q
```



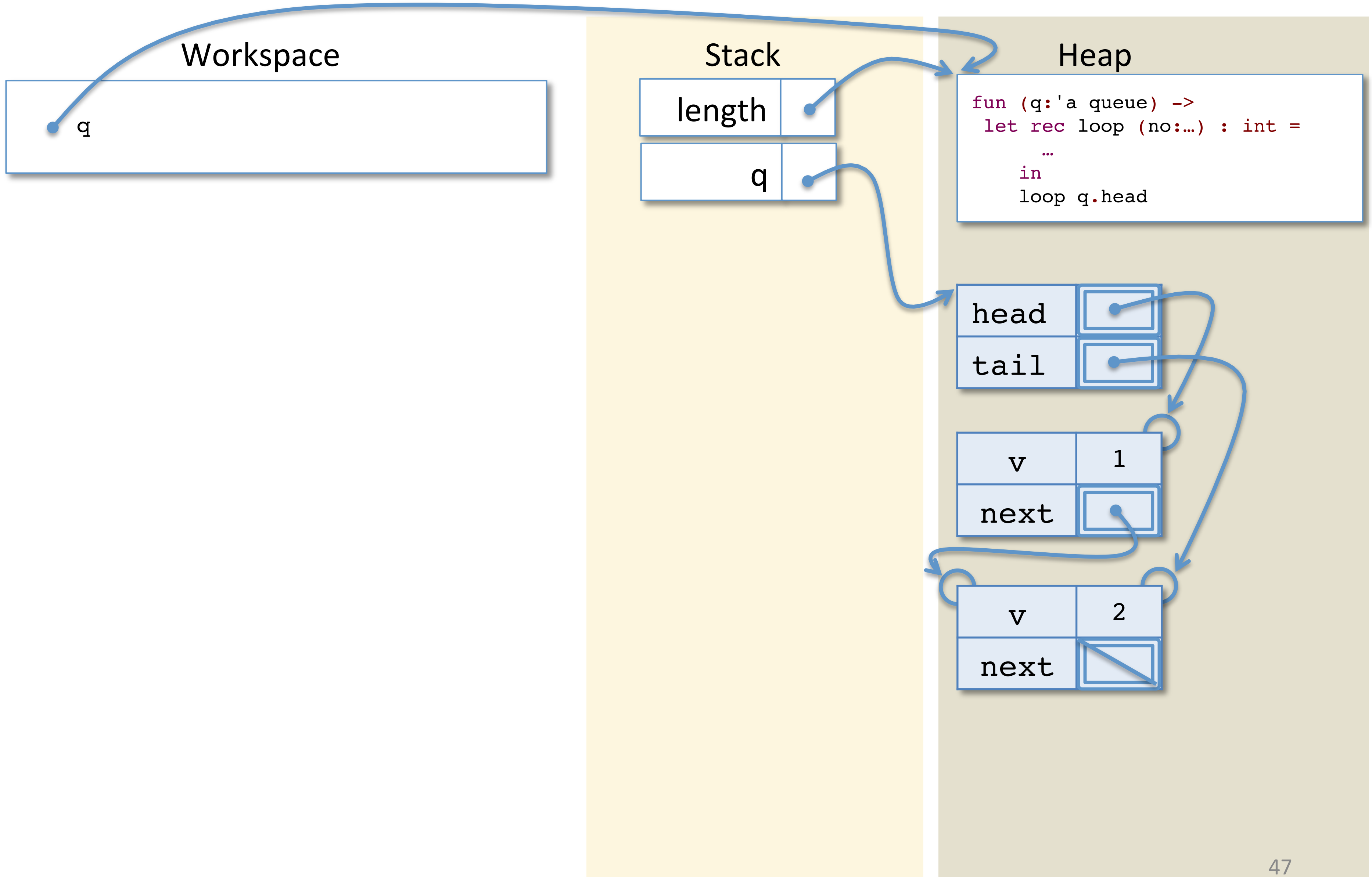
Evaluating length

Workspace

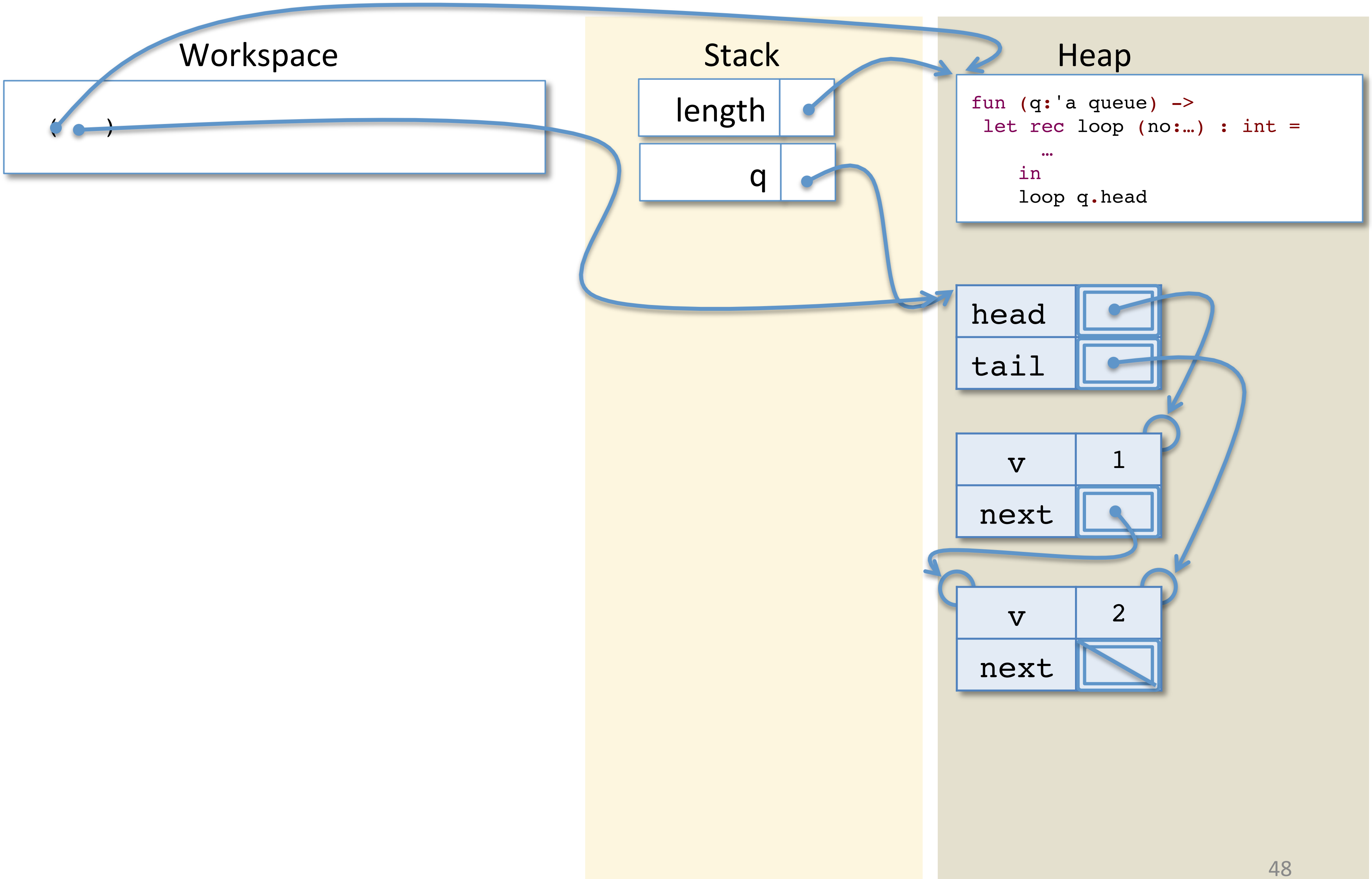
<u>length</u> q



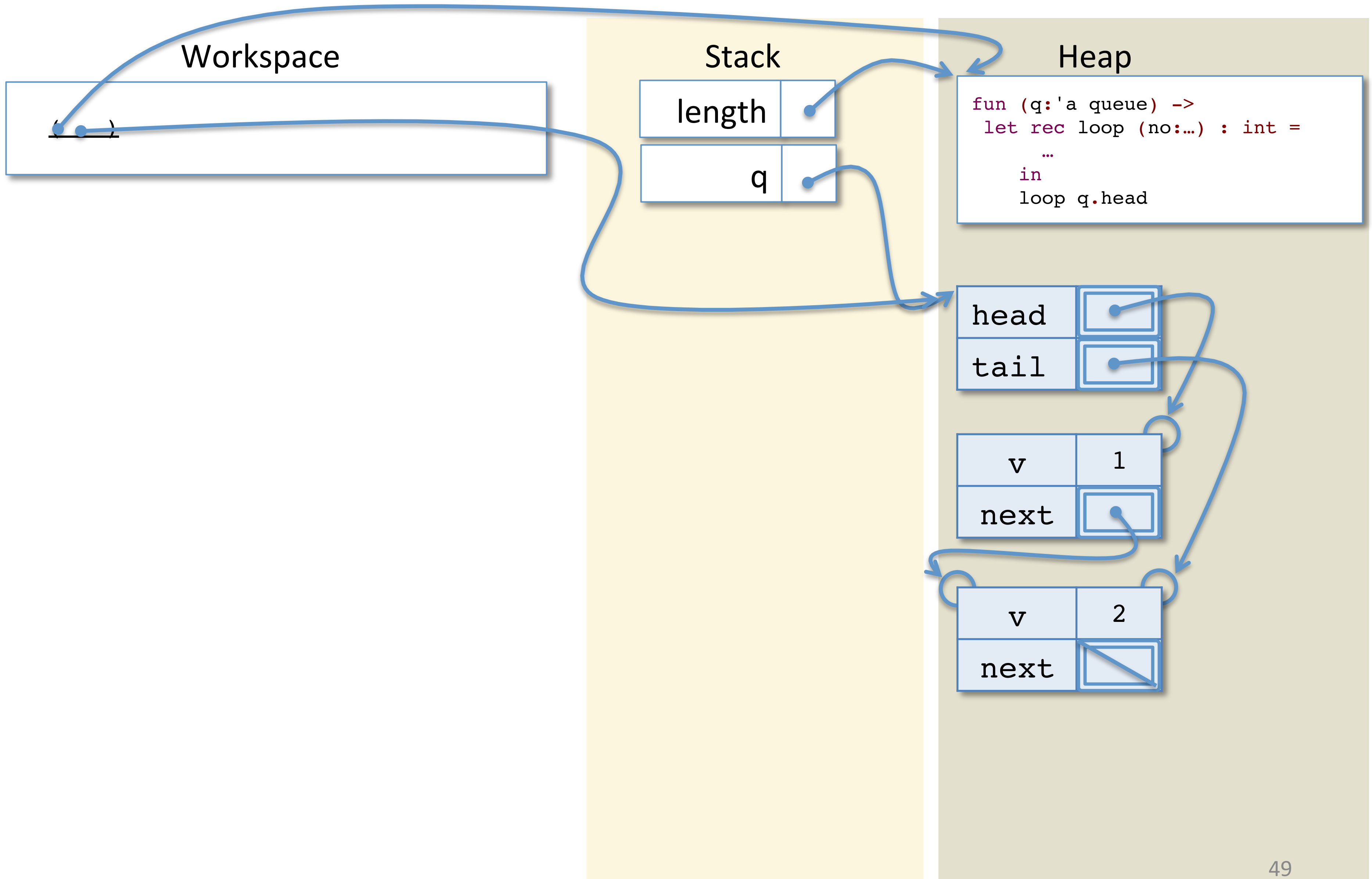
Evaluating length



Evaluating length



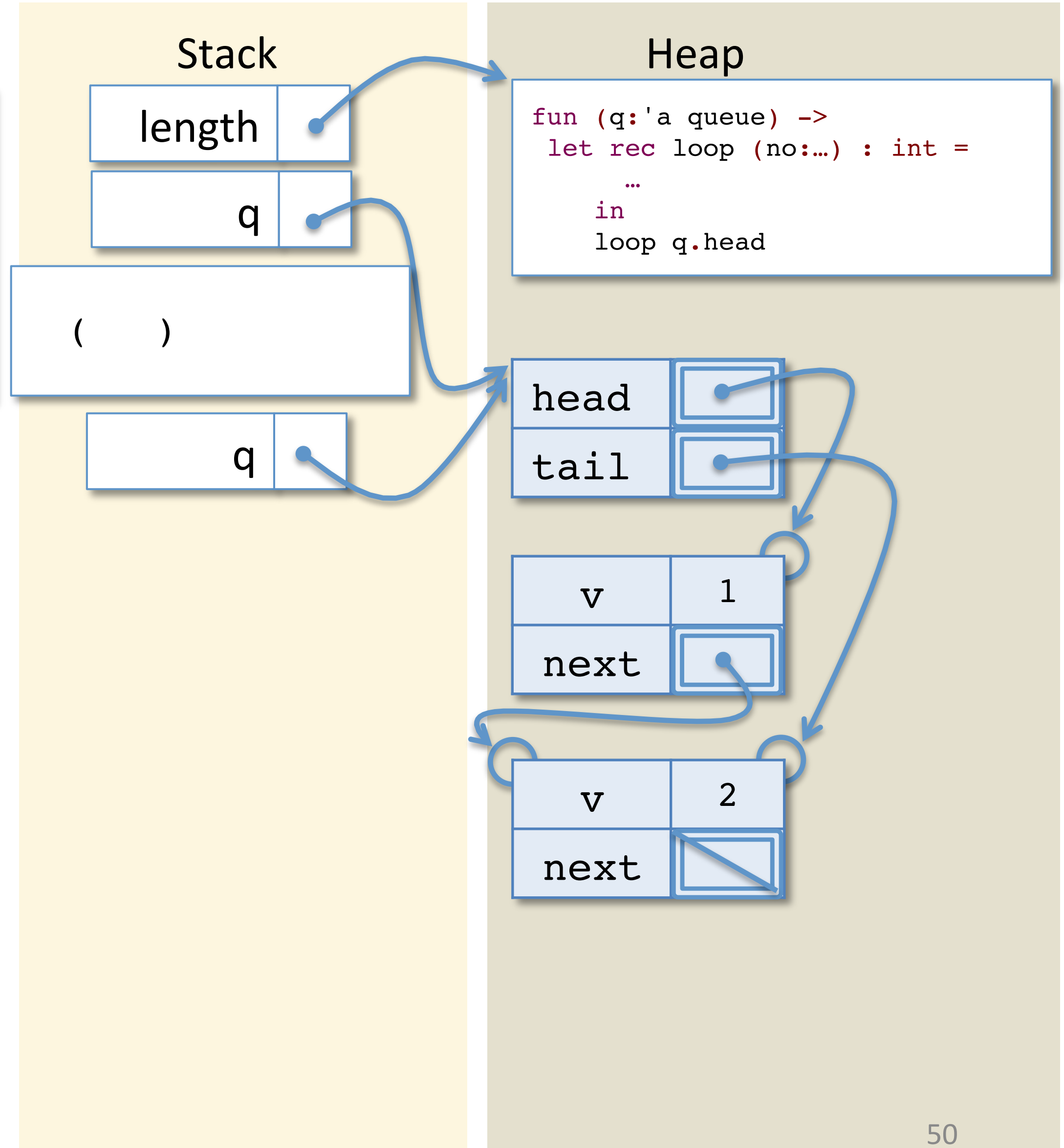
Evaluating length



Evaluating length

Workspace

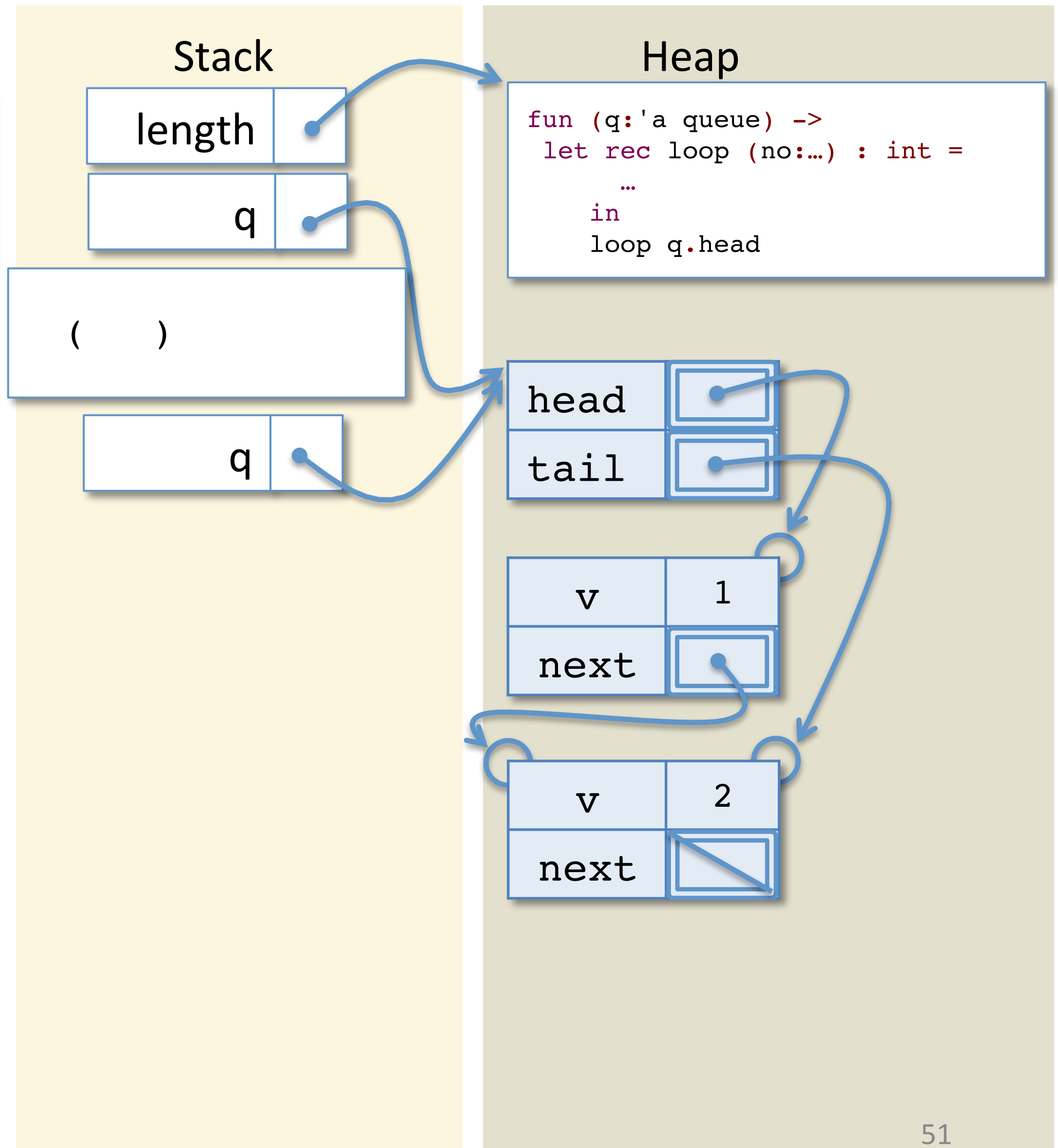
```
let rec loop (no: …) : int =
      begin match no with
        | None -> 0
        | Some n -> 1 + (loop n.next)
      end
    in
    loop q.head
```



Evaluating length

Workspace

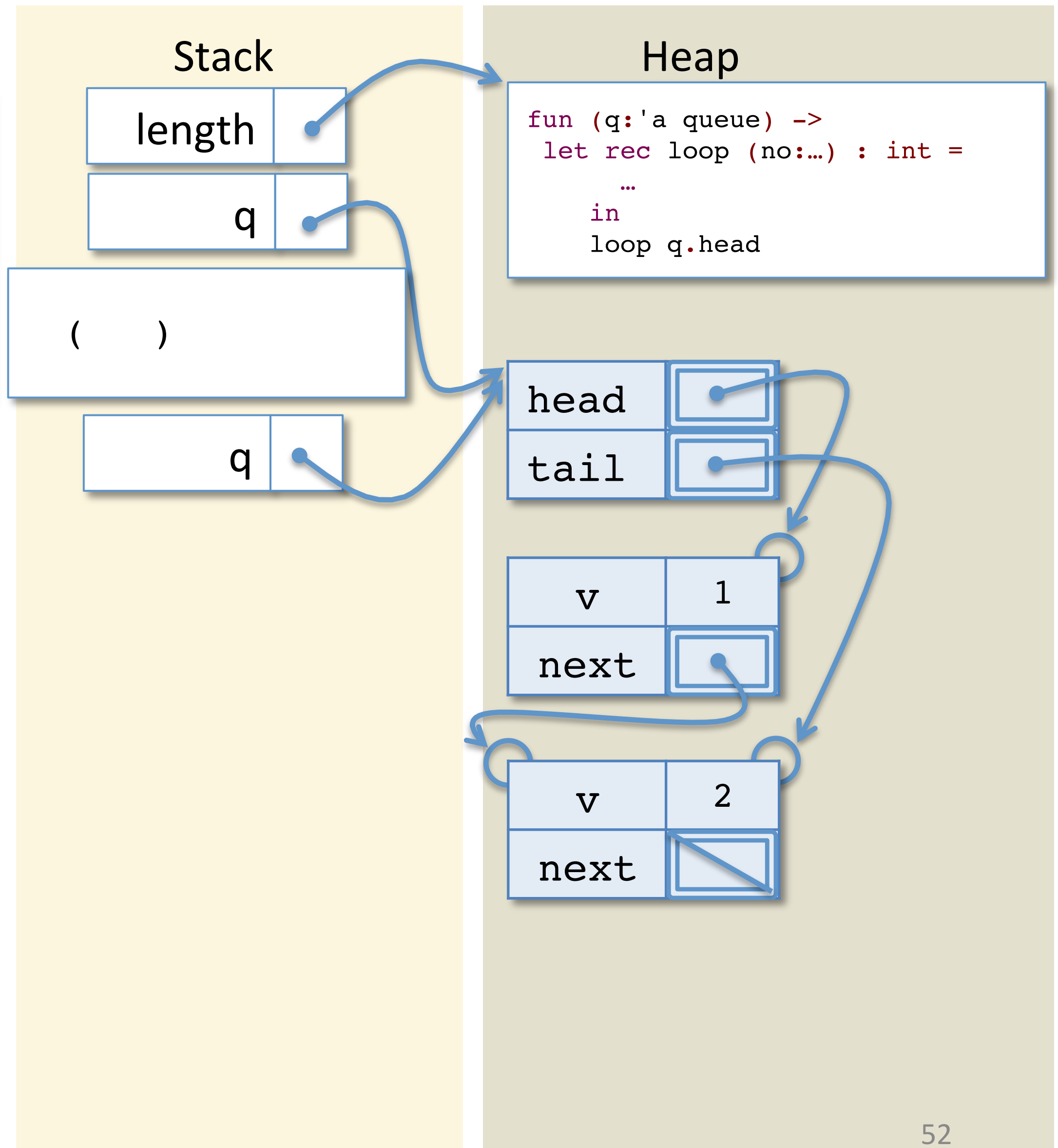
```
let loop = fun (no: …) ->
      begin match no with
        | None -> 0
        | Some n -> 1 + (loop n.next)
      end
    in
    loop q.head
```



Evaluating length

Workspace

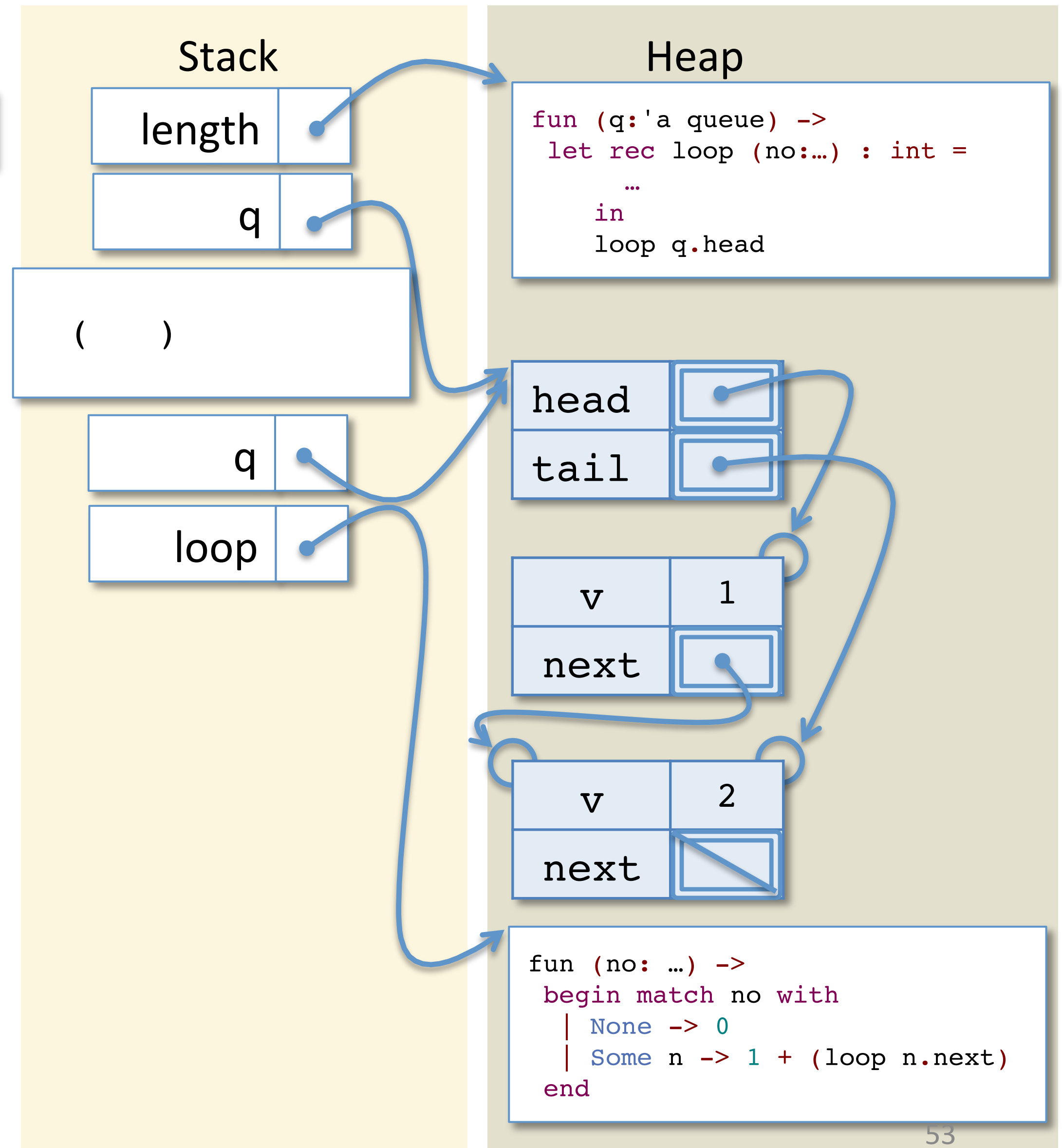
```
let loop = fun (no: …) ->
      begin match no with
        | None -> 0
        | Some n -> 1 + (loop n.next)
      end
    in
    loop q.head
```



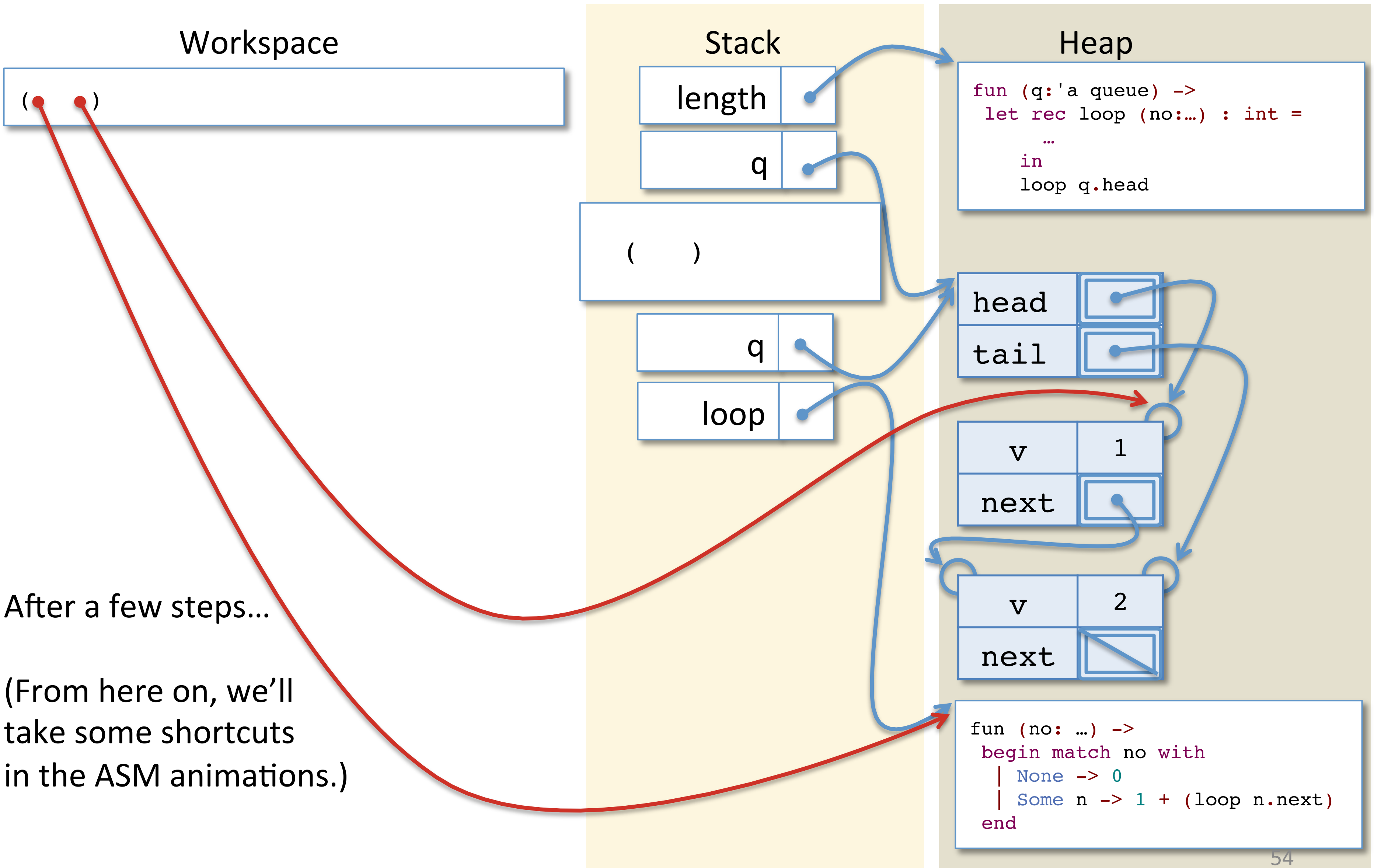
Evaluating length

Workspace

```
loop q.head
```



Evaluating length



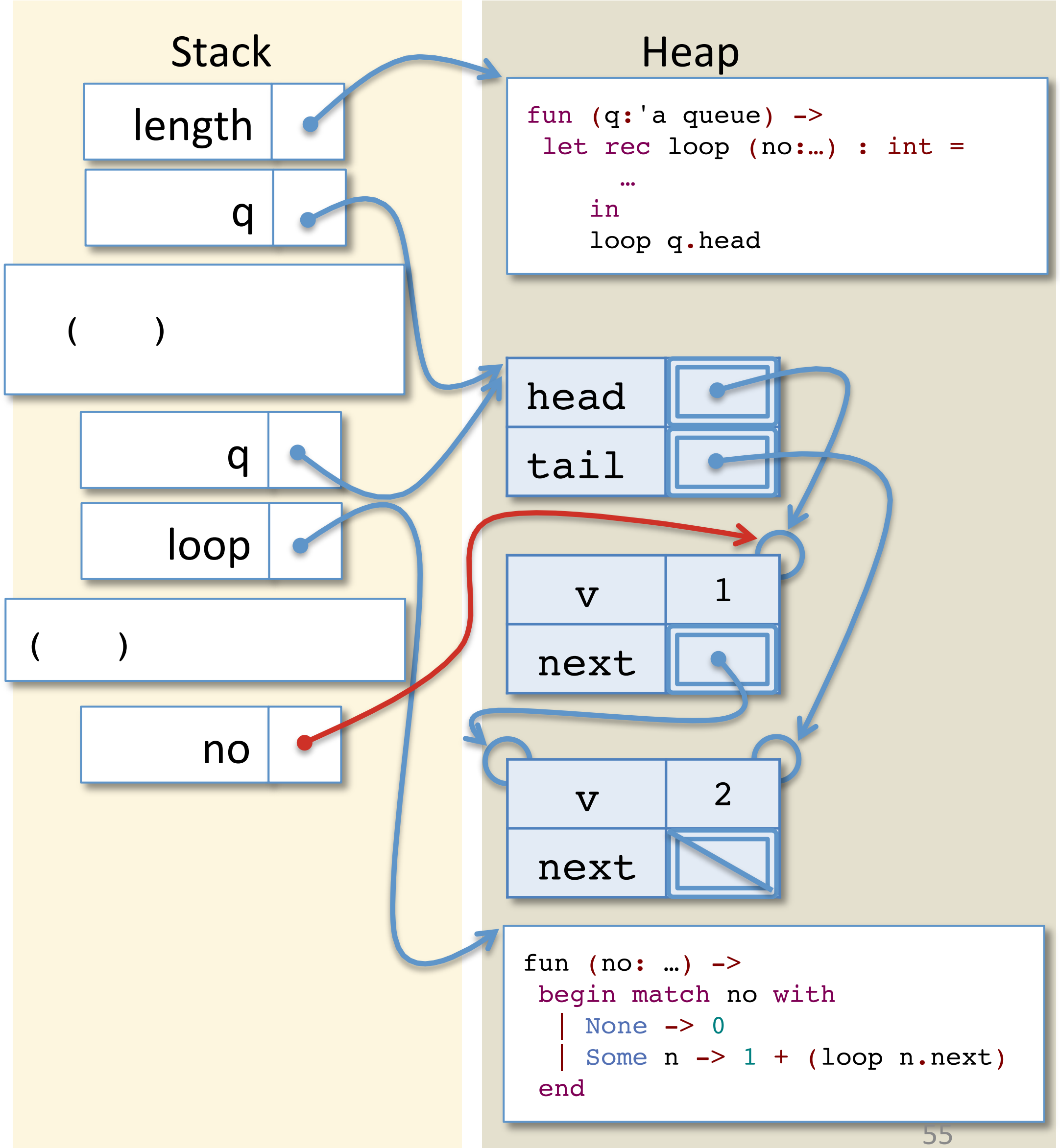
After a few steps…

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

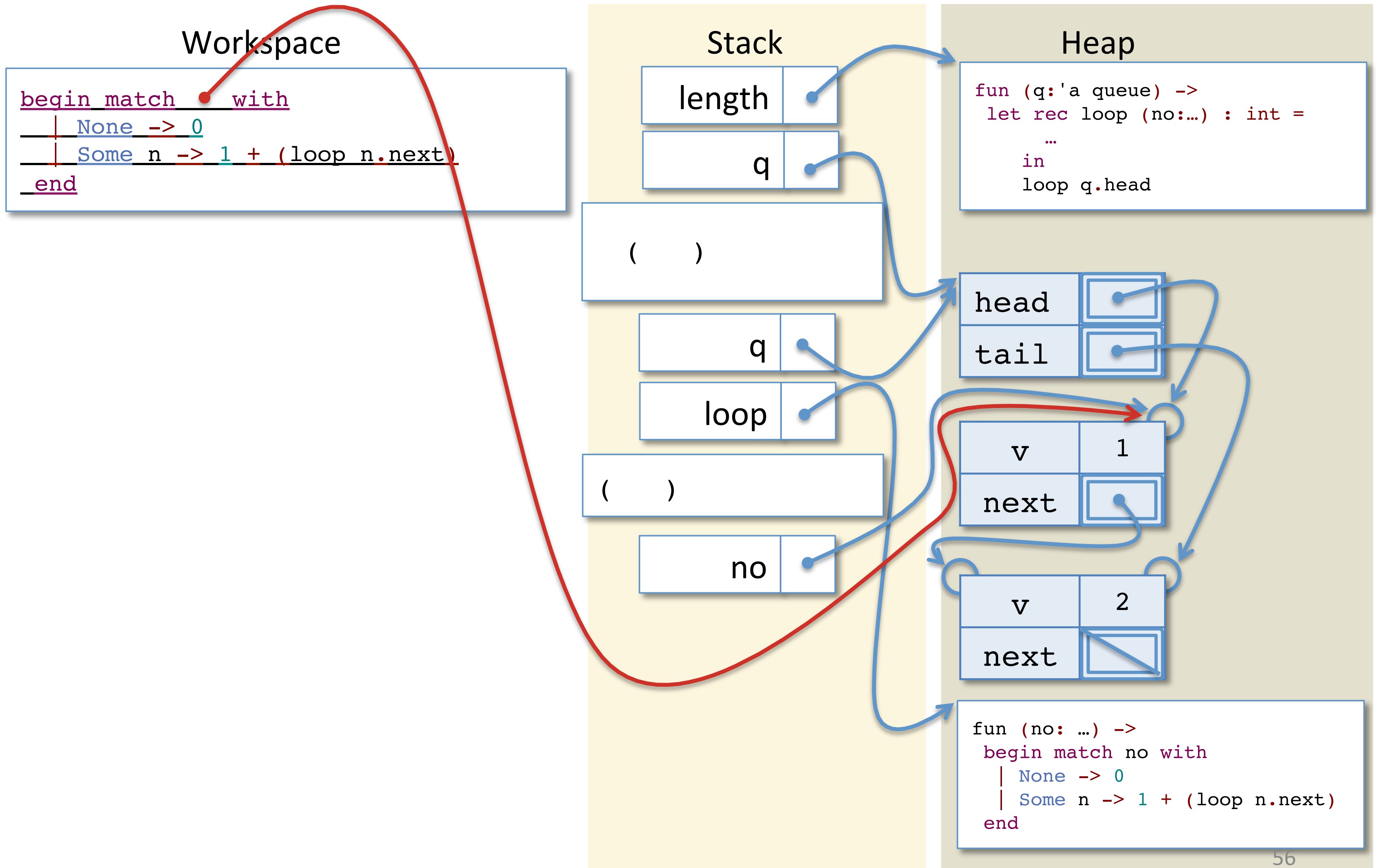
Evaluating length

Workspace

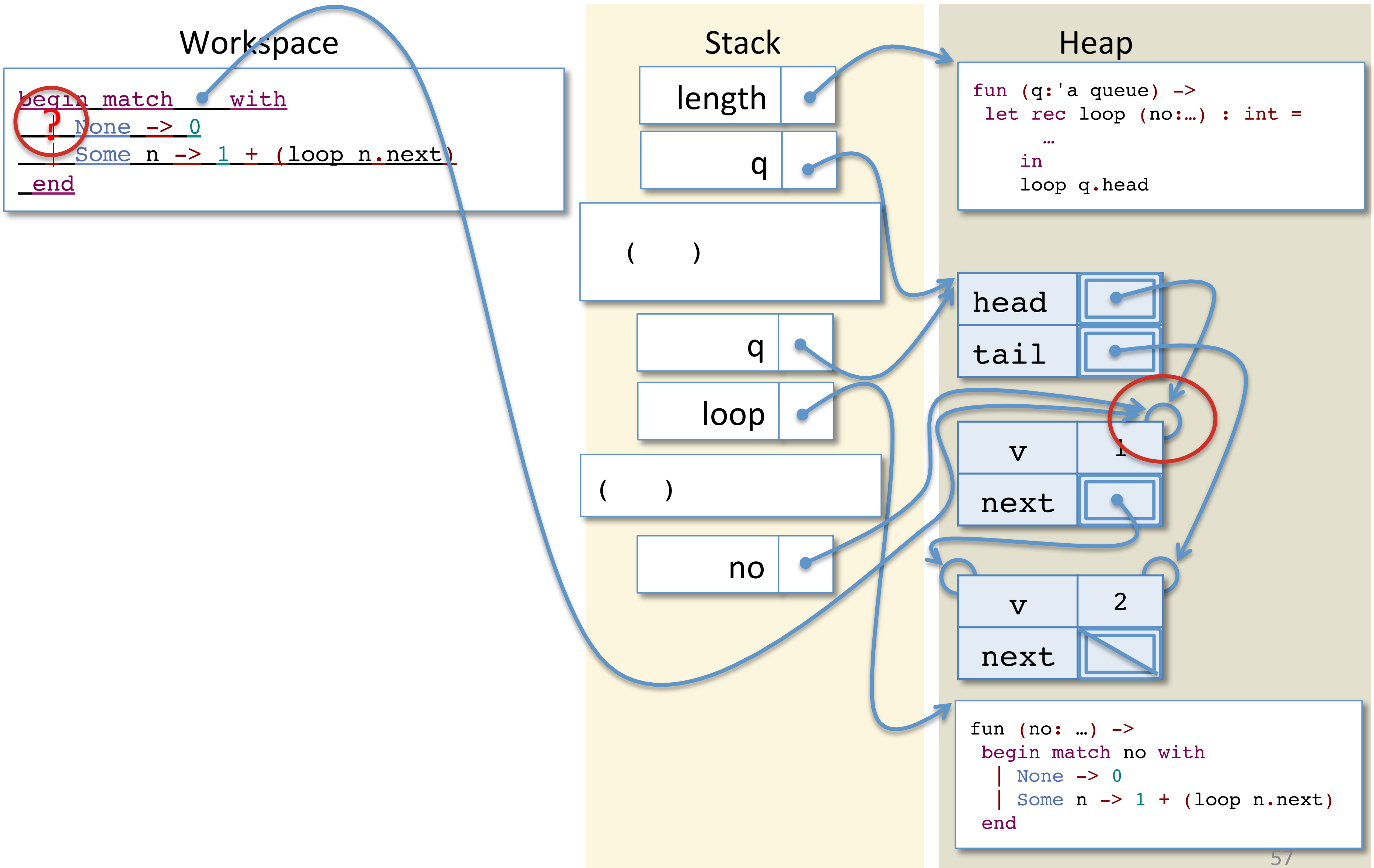
```
begin match no with
  | None -> 0
  | Some n -> 1 + (loop n.next)
end
```



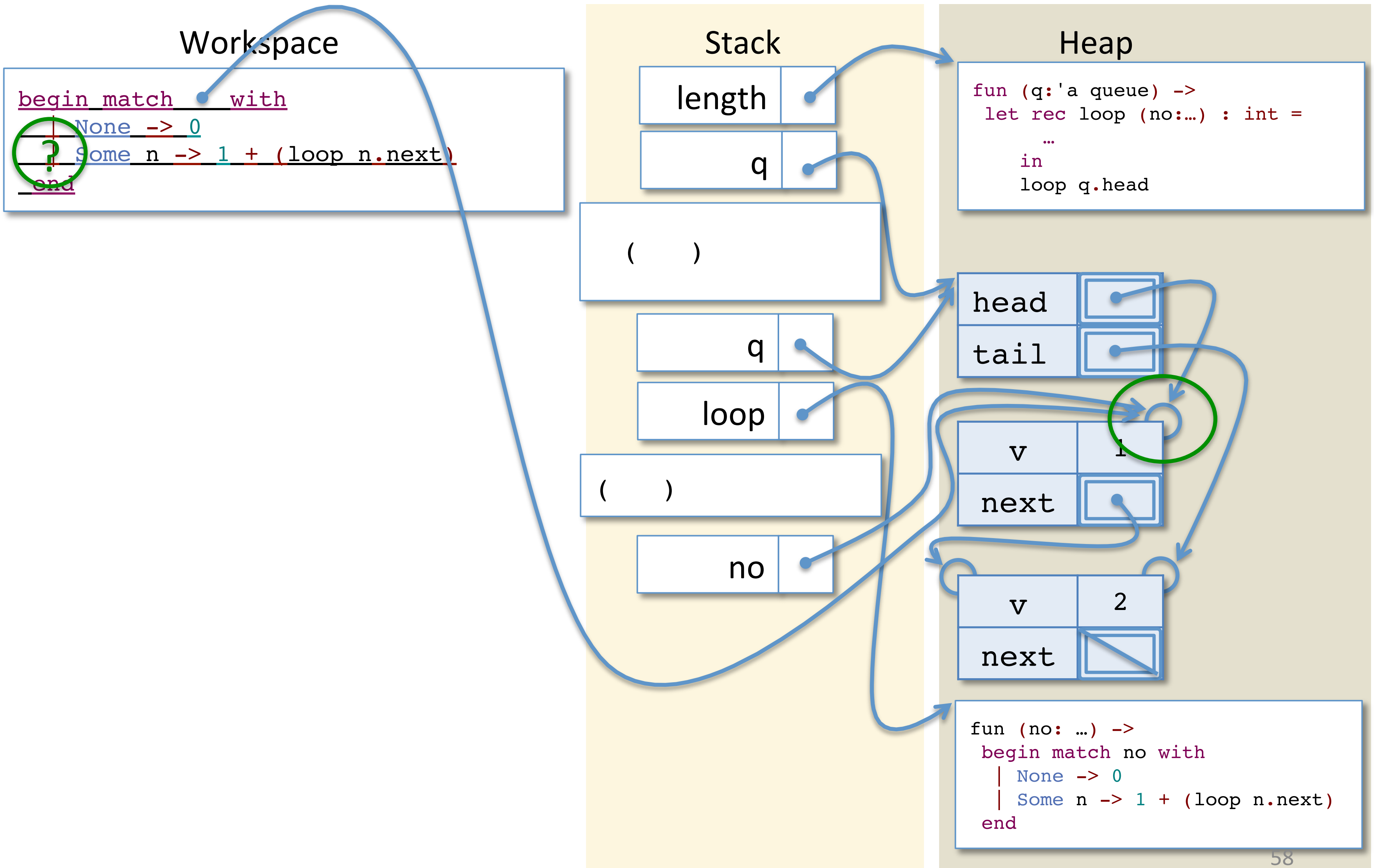
Evaluating length



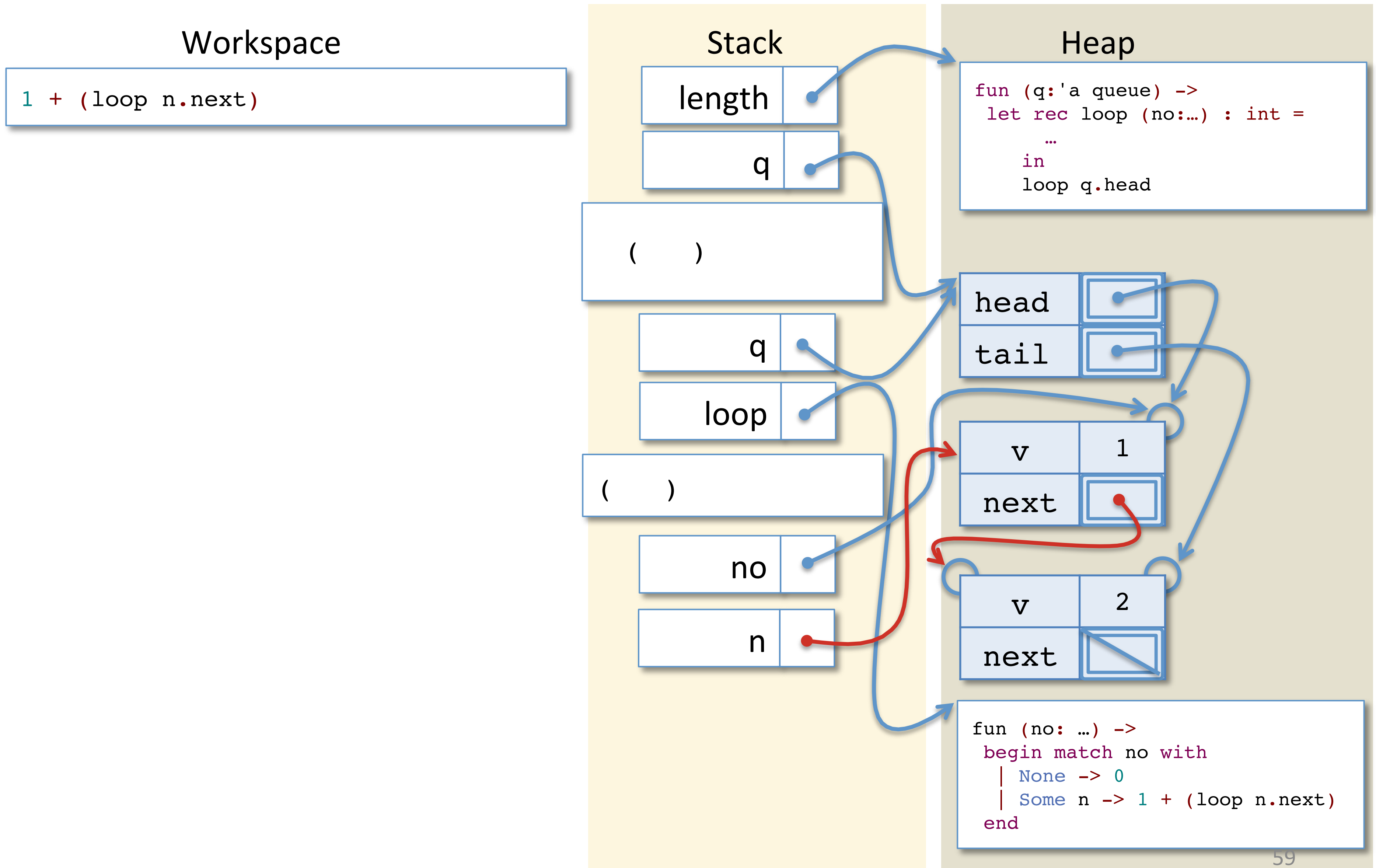
Evaluating length



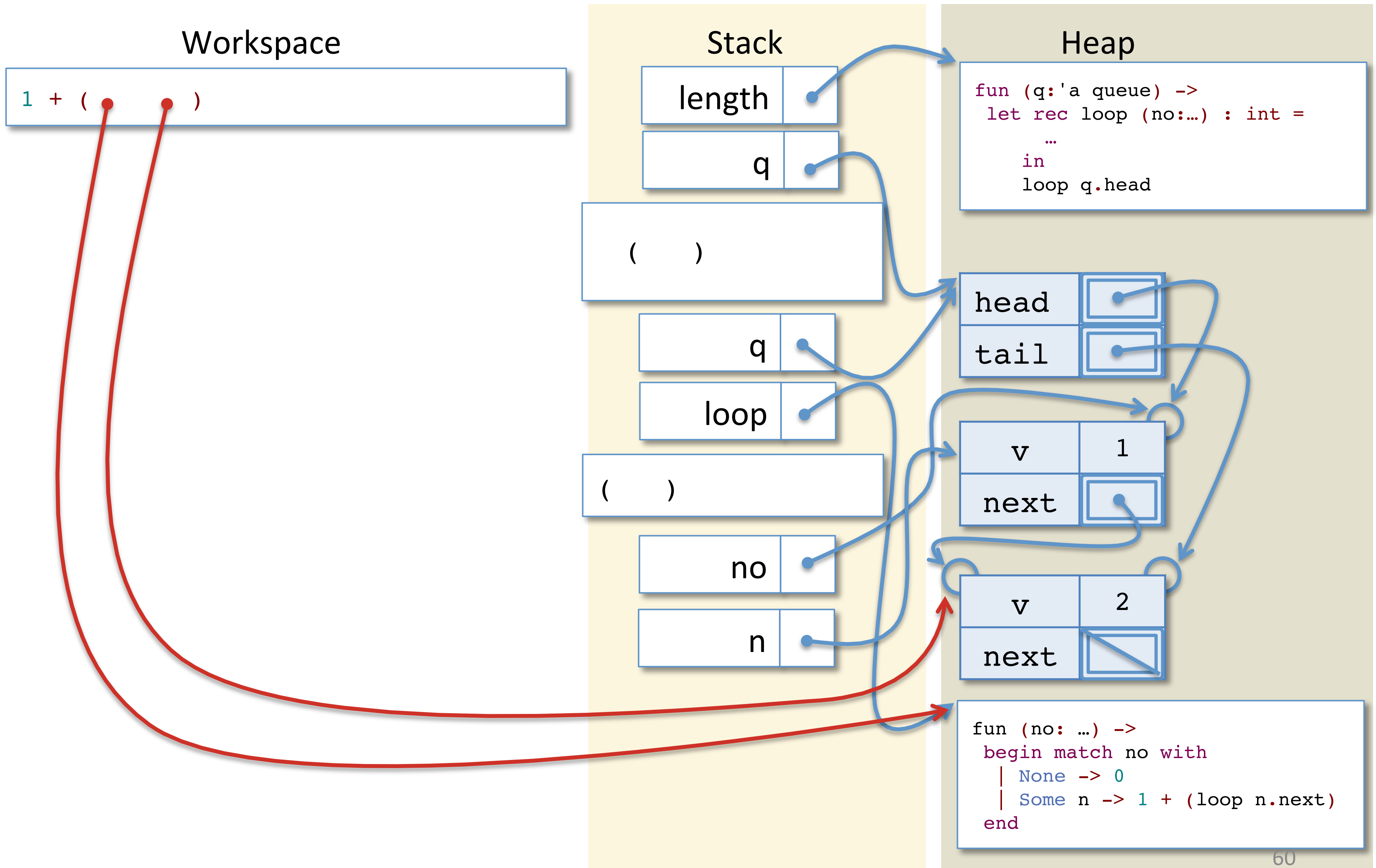
Evaluating length



Evaluating length



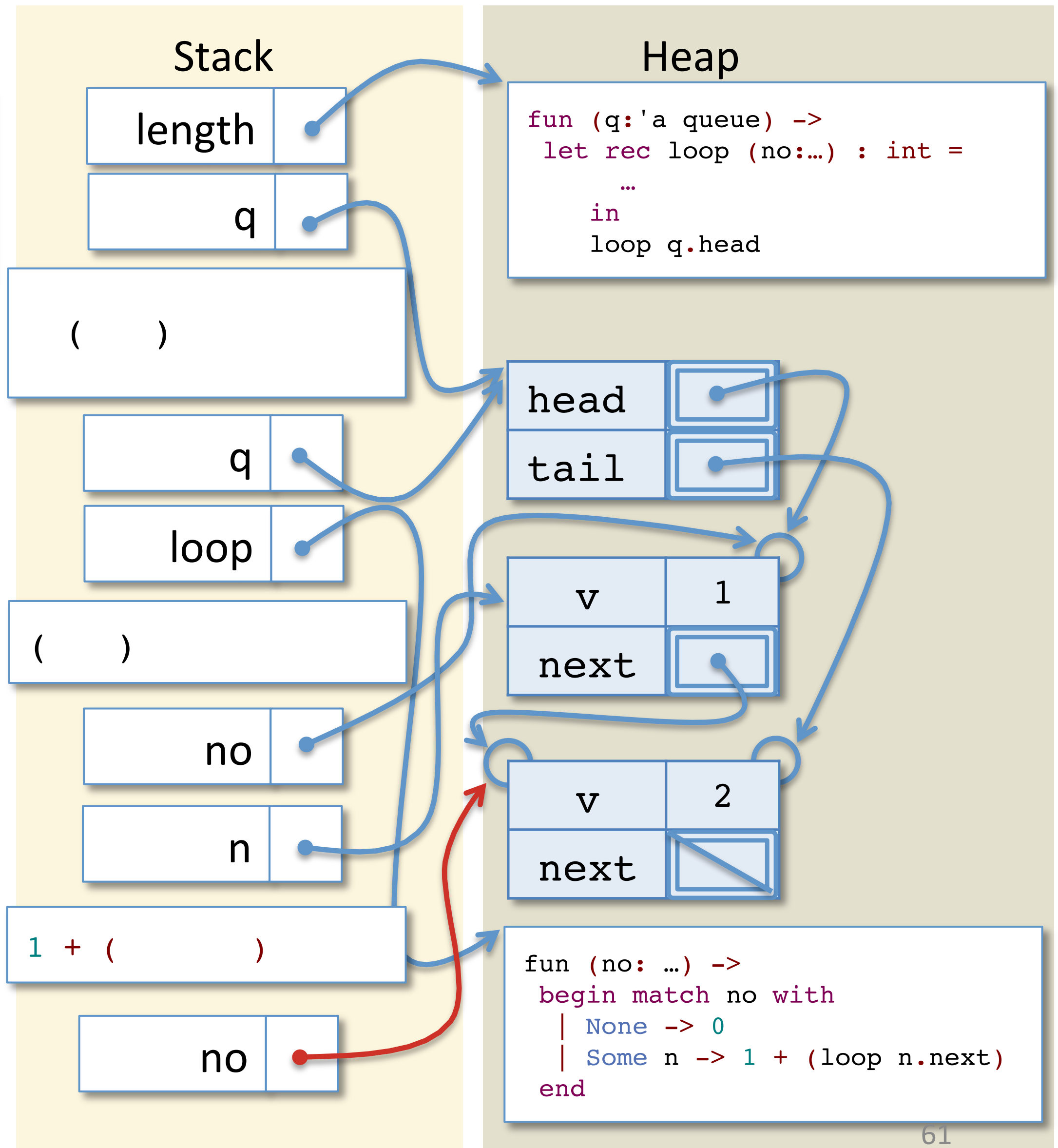
Evaluating length



Evaluating length

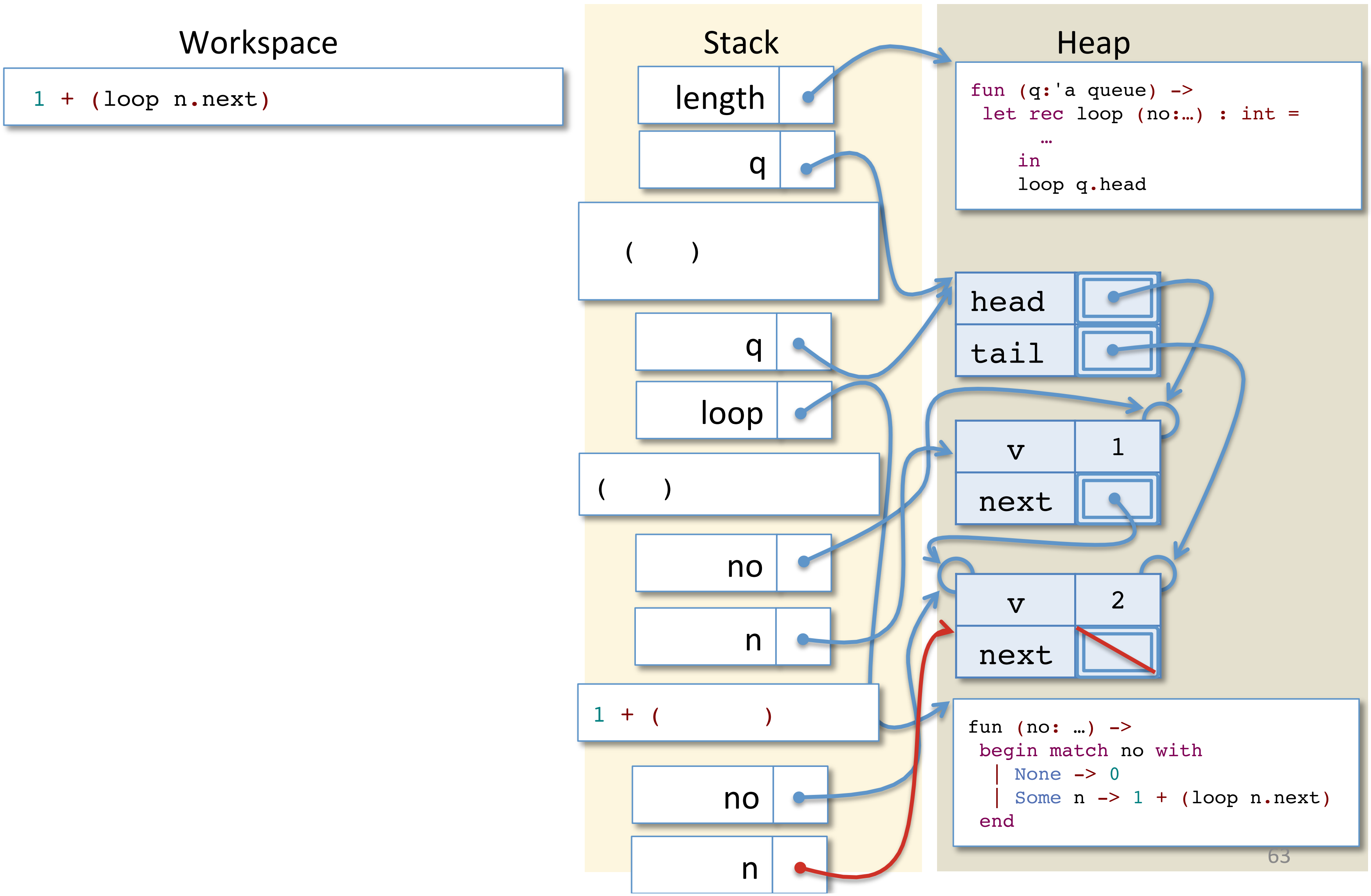
Workspace

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



…after a few steps…

Evaluating length

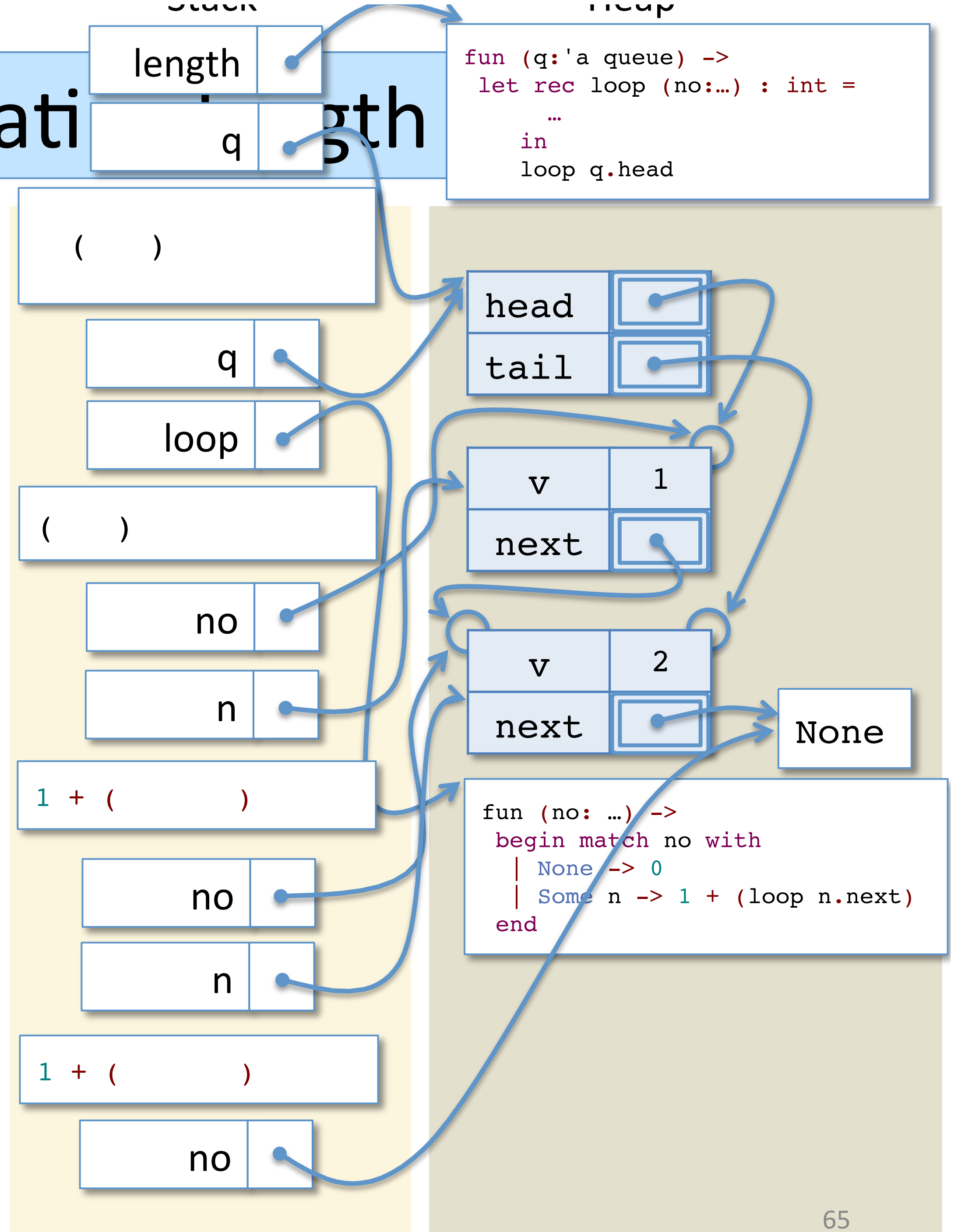


…after a few more steps…

Evaluati gth

Workspace

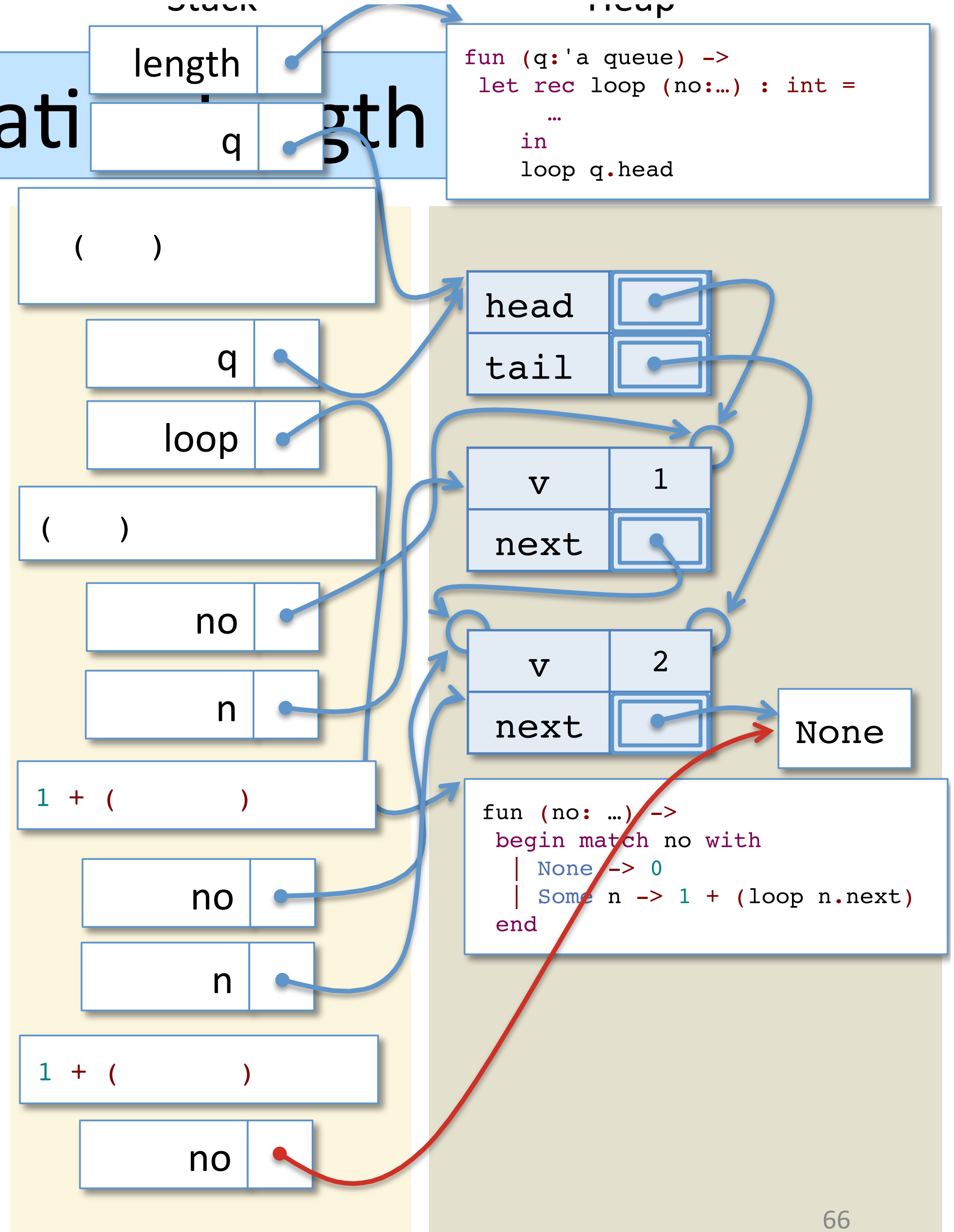
```
begin match no with
  | None -> 0
  | Some n -> 1 + (loop n.next)
end
```



Evaluati gth

Workspace

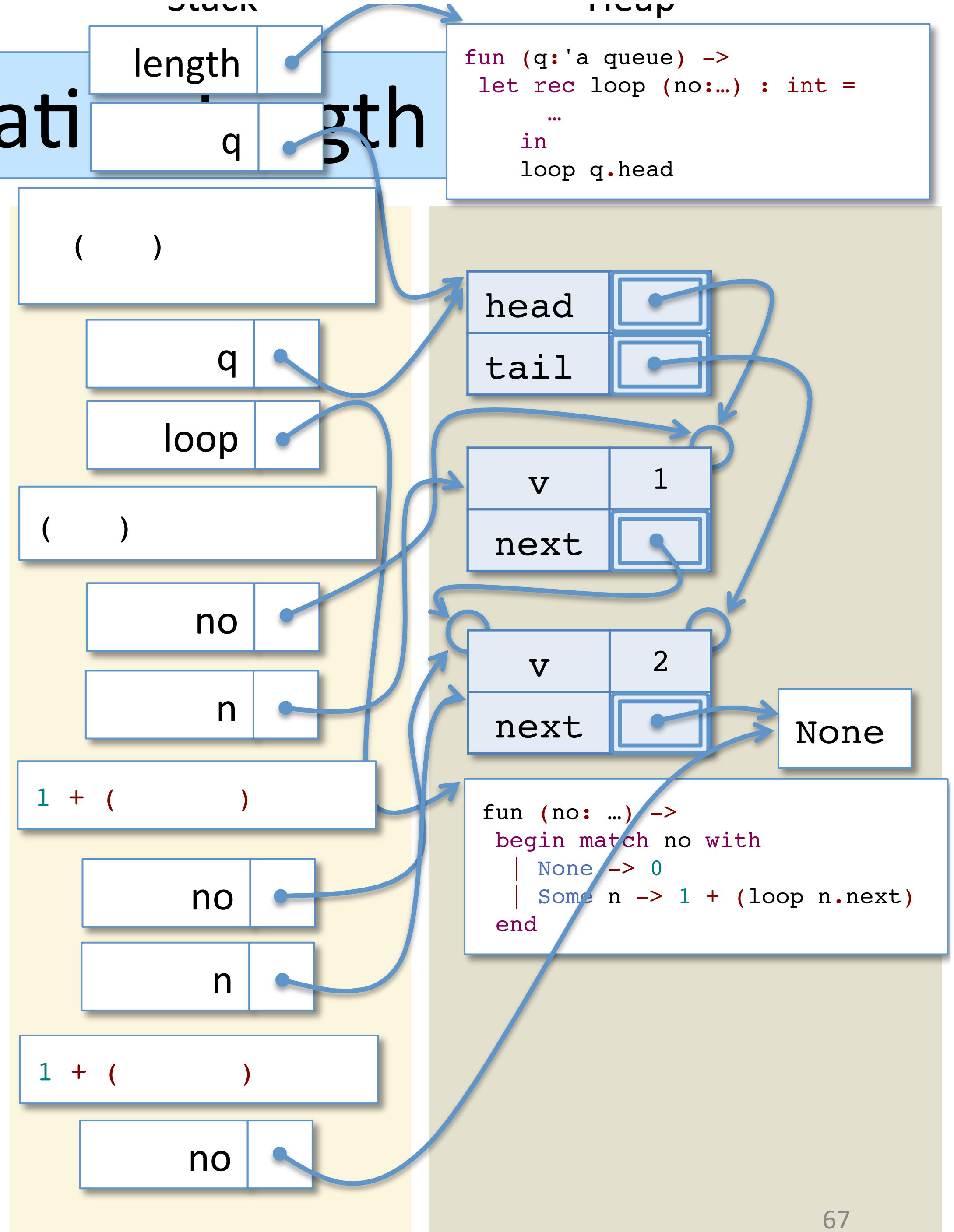
```
begin match no with
  | None -> 0
  | Some n -> 1 + (loop n.next)
end
```

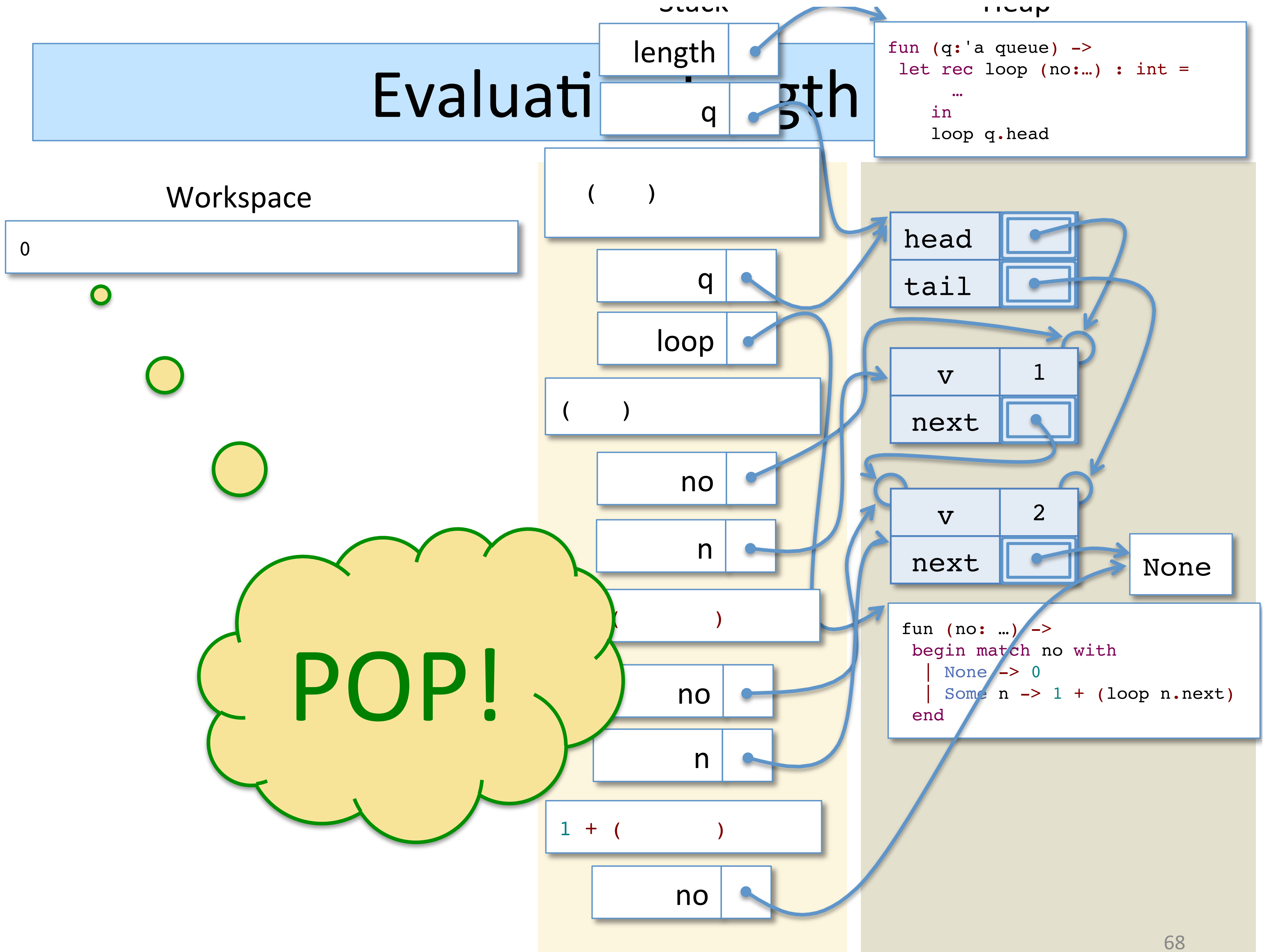


Evaluati gth

Workspace

0



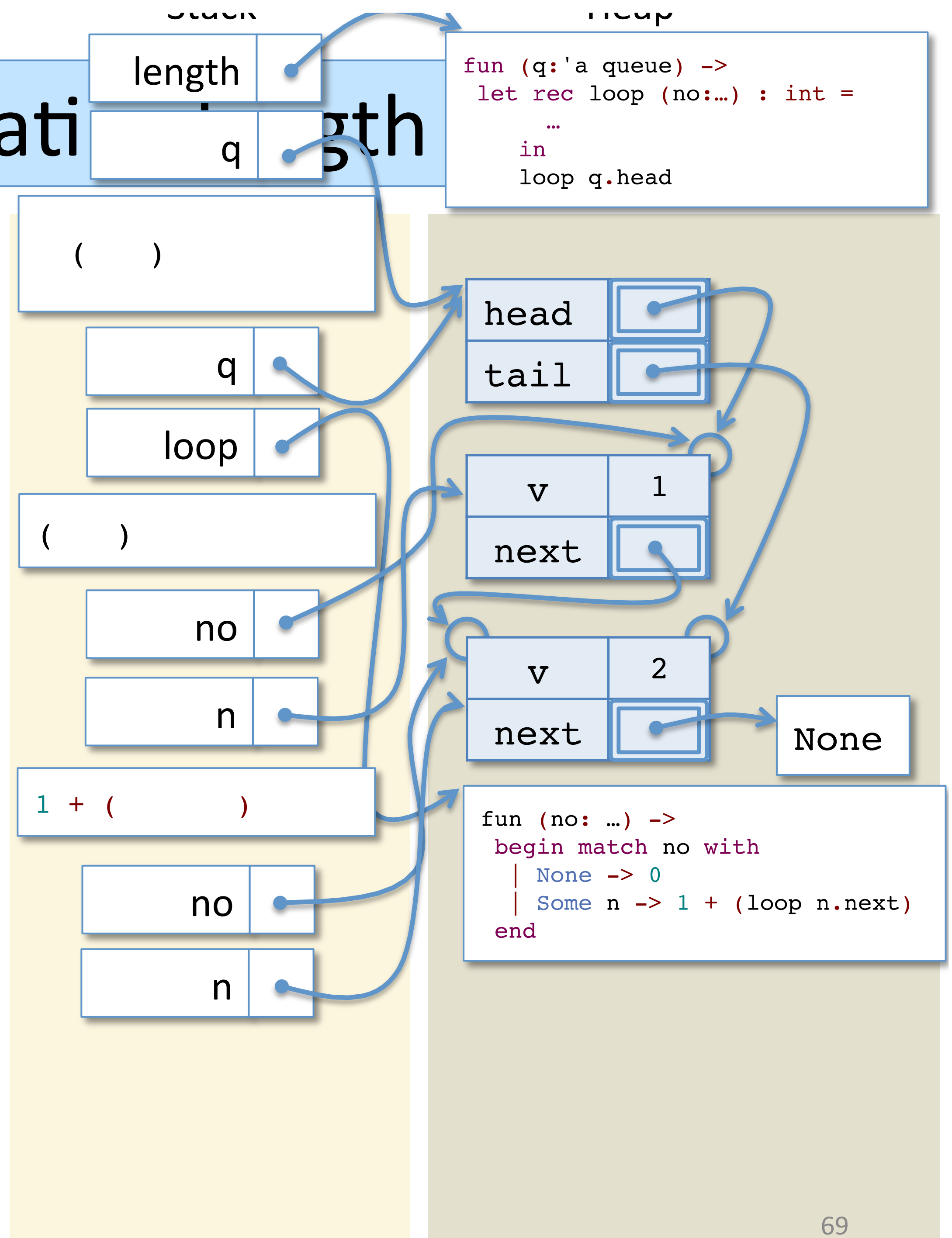


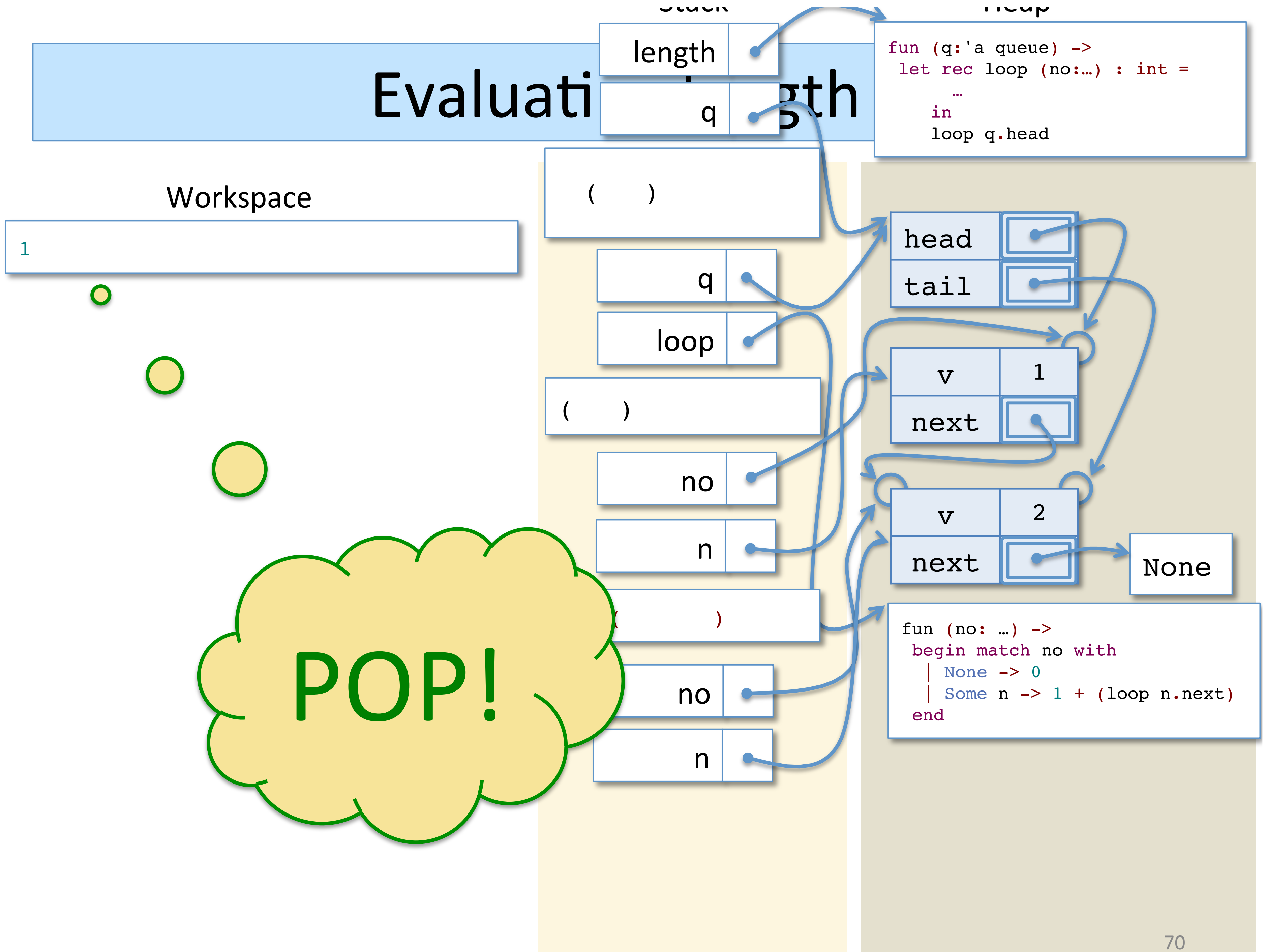
Evaluating length
Stack
Heap
Workspace
0
length
q
( )
q
loop
( )
no
n
( )
no
n
1 + ( )
no
fun (q:'a queue) ->
let rec loop (no:…) : int =
…
in
loop q.head
head
tail
v
1
next
v
2
next
None
fun (no: …) ->
begin match no with
| None -> 0
| Some n -> 1 + (loop n.next)
end
POP!

Evaluati gth

Workspace

1 + 0

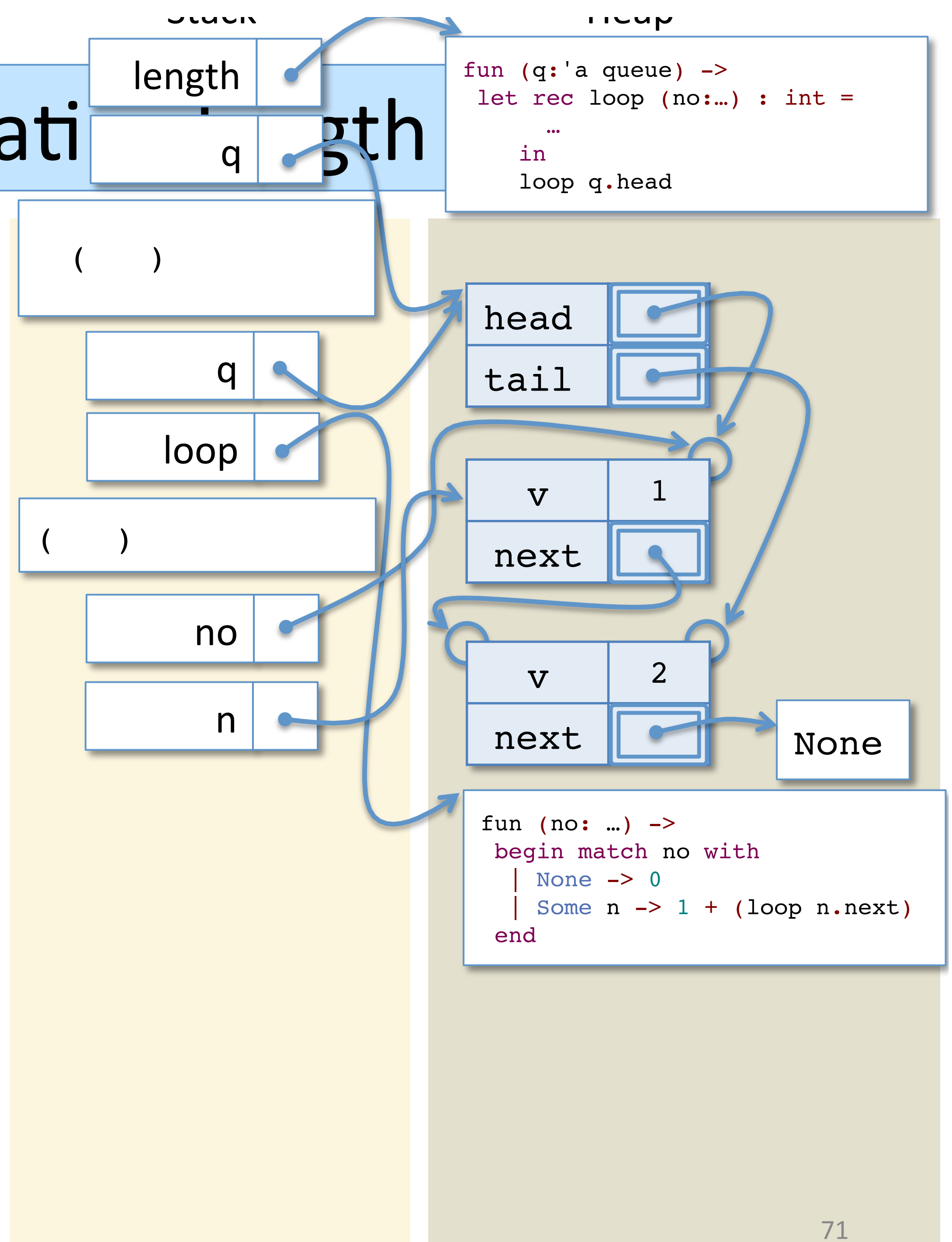



Evaluati
gth
Stack
Heap
Workspace
1
POP!
length
q
( )
q
loop
( )
no
n
( )
no
n
fun (q:'a queue) ->
let rec loop (no:…) : int =
…
in
loop q.head
head
tail
v
1
next
v
2
next
None
fun (no: …) ->
begin match no with
| None -> 0
| Some n -> 1 + (loop n.next)
end

Evaluati gth

Workspace

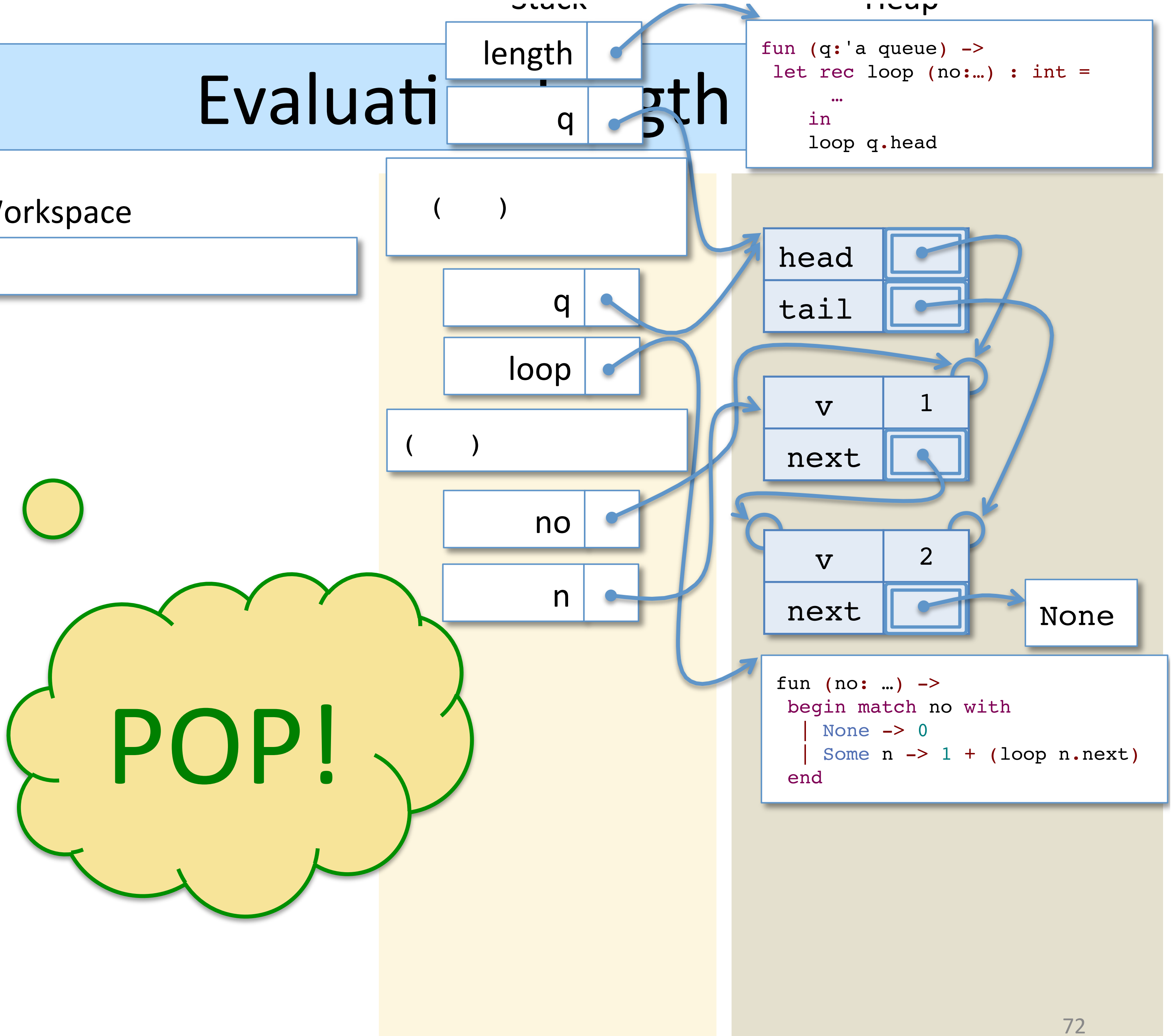
1 + 1

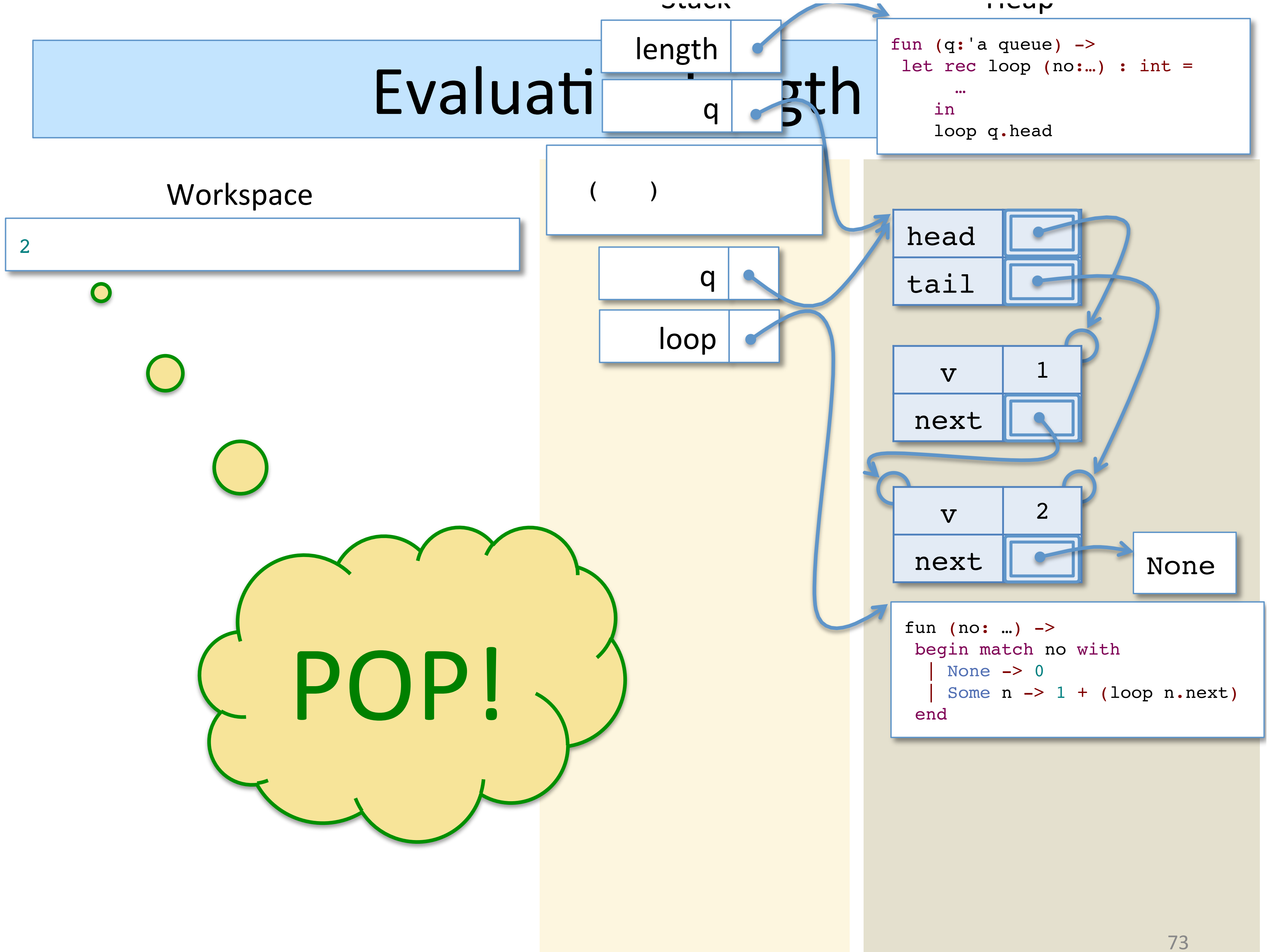


Evaluati gth

Workspace

2

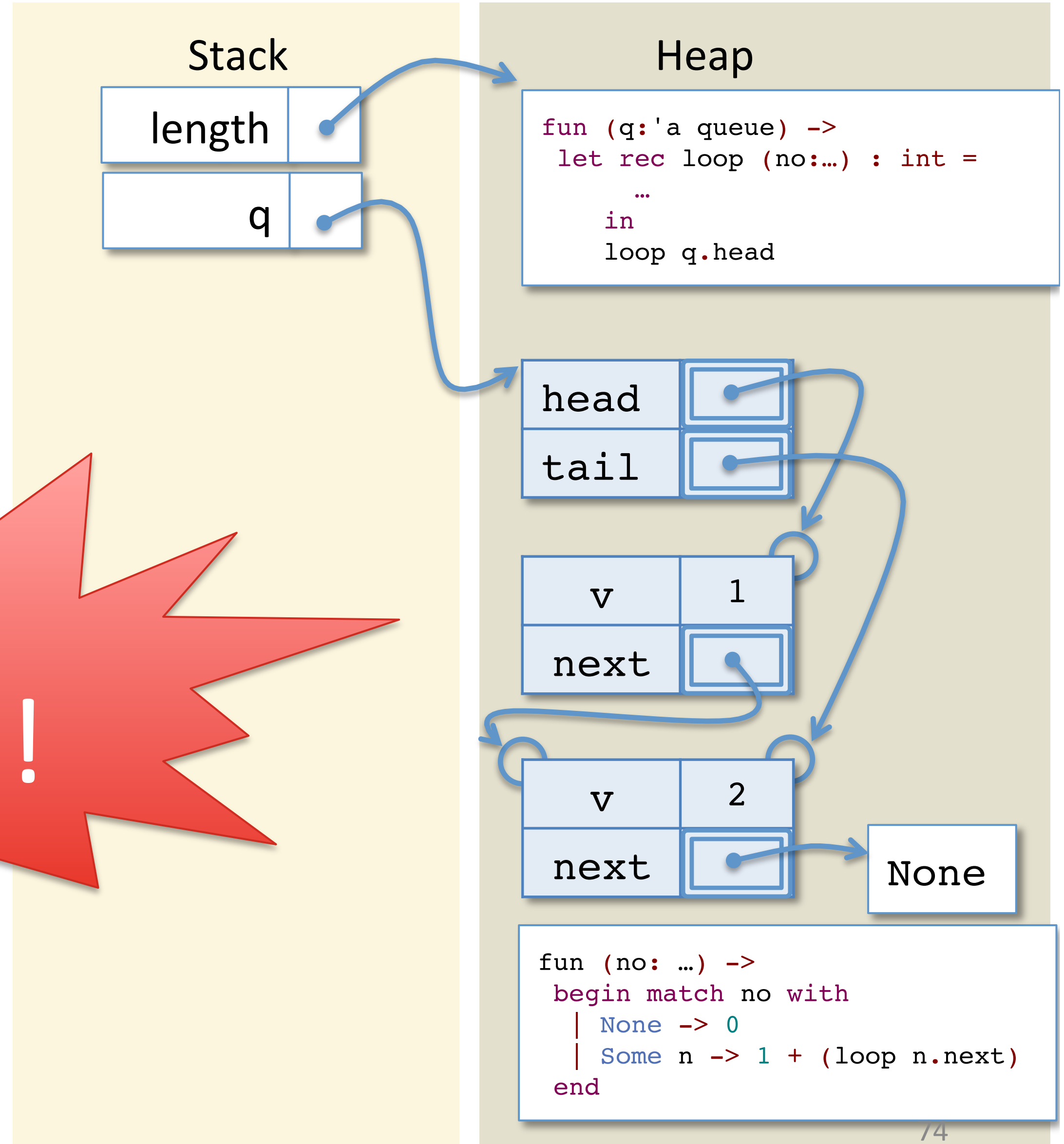



Evaluati gth
Stack
Heap
length
q
( )
q
loop
Workspace
2
fun (q:'a queue) ->
let rec loop (no:…) : int =
…
in
loop q.head
head
tail
v 1
next
v 2
next
None
fun (no: …) ->
begin match no with
| None -> 0
| Some n -> 1 + (loop n.next)
end
POP!

Evaluating length

Workspace

2



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

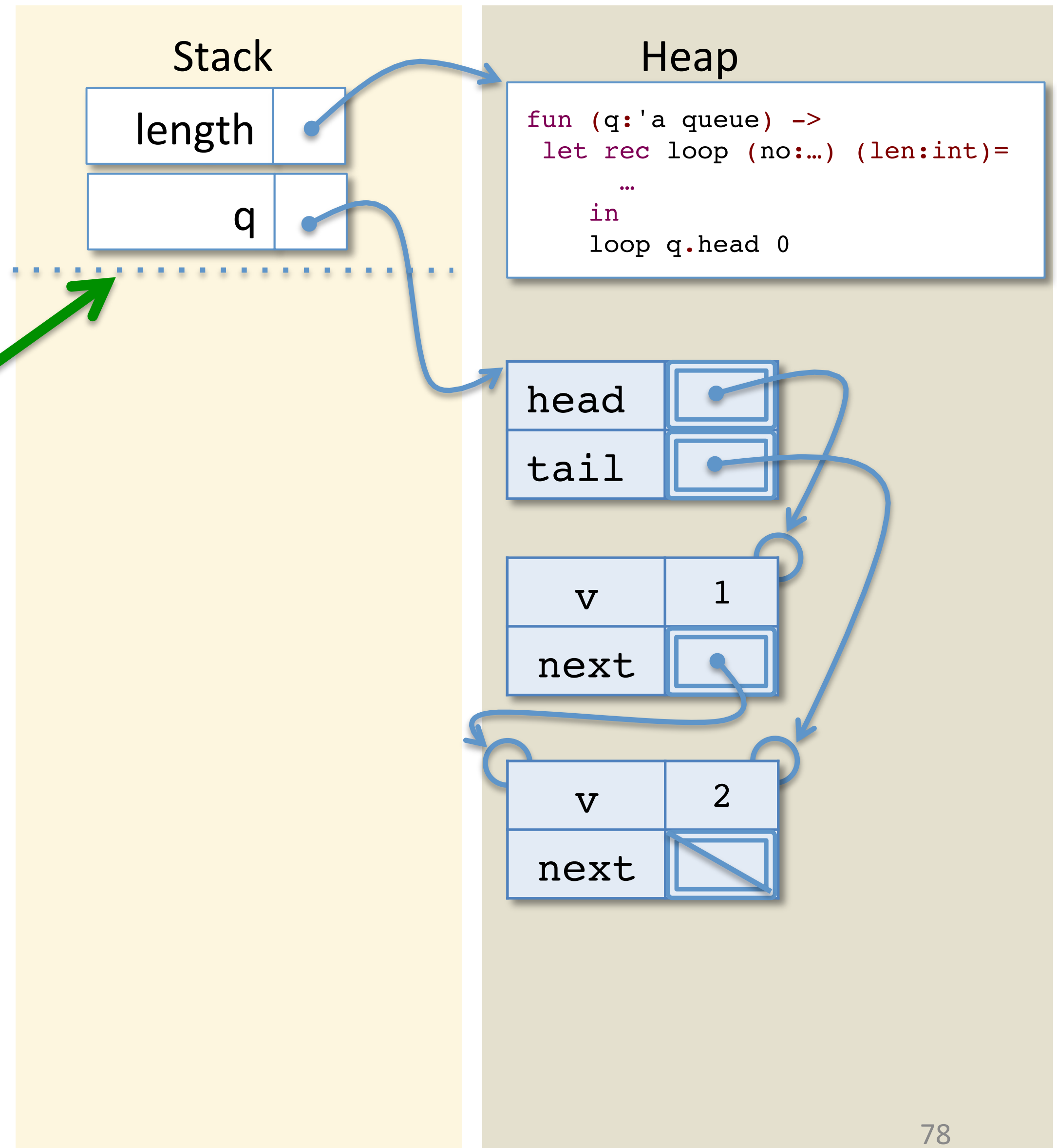
- Why does it matter that ‘loop’ is only called in an empty workspace?
- We can *optimize* the abstract stack machine:
  - The workspace pushed onto the stack tells us “what to do” when the function call returns.
  - If the pushed workspace is empty, we will always ‘pop’ immediately after the function 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

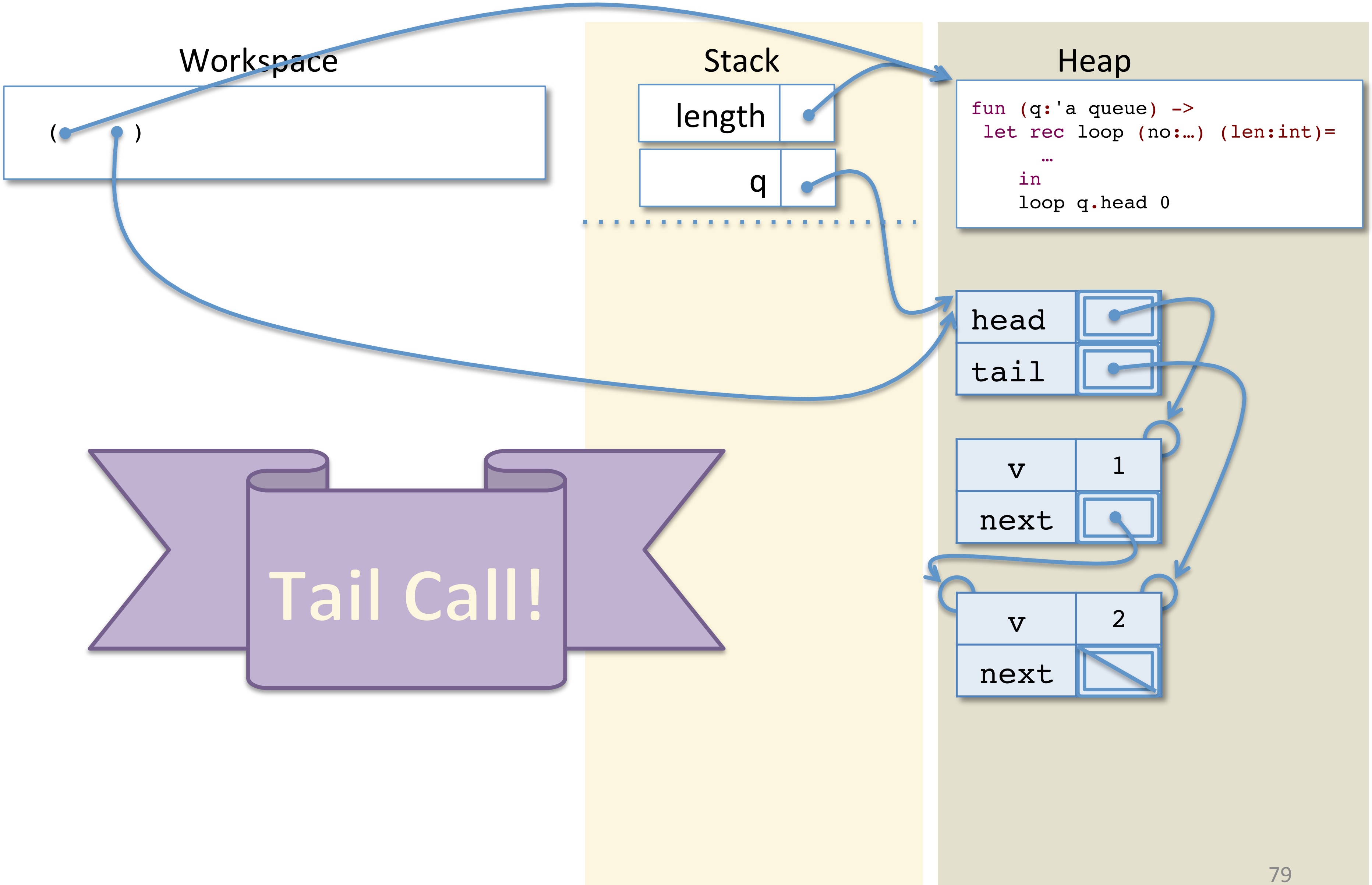
Workspace

```
length q
```

Bindings above this line
are top-level declarations.



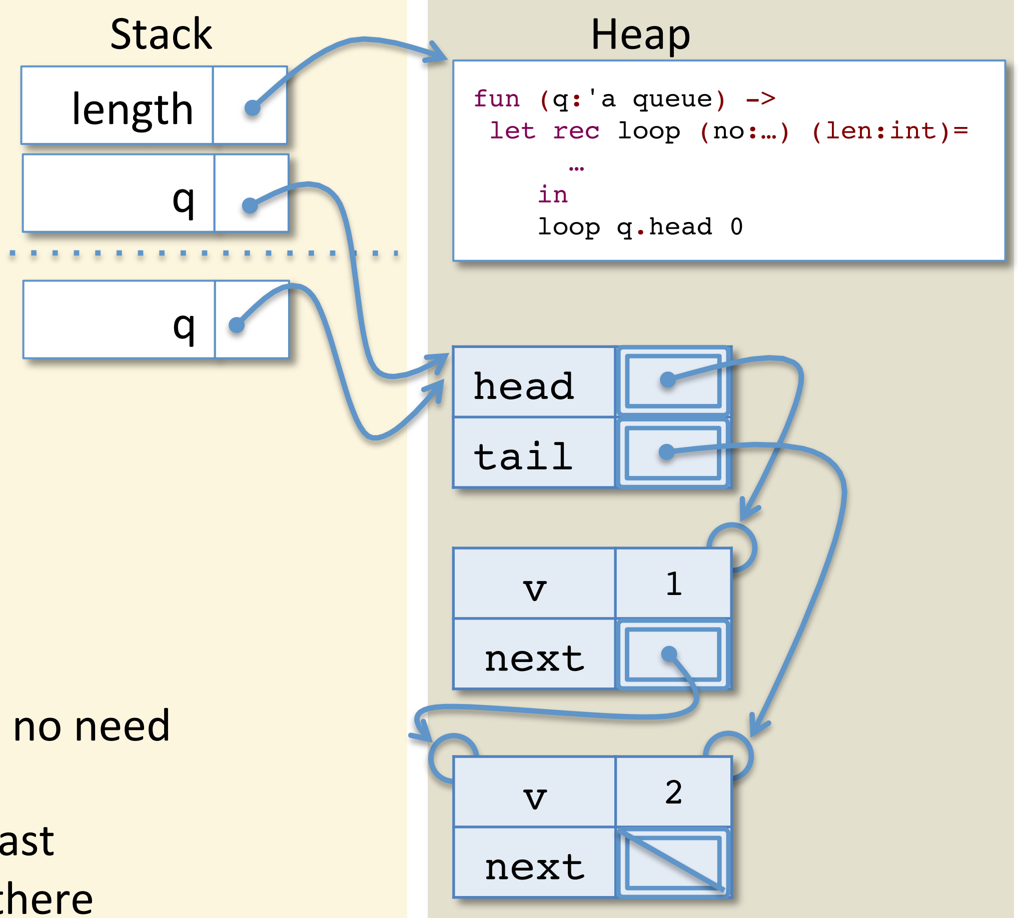
Tail Calls and Iterative length



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:

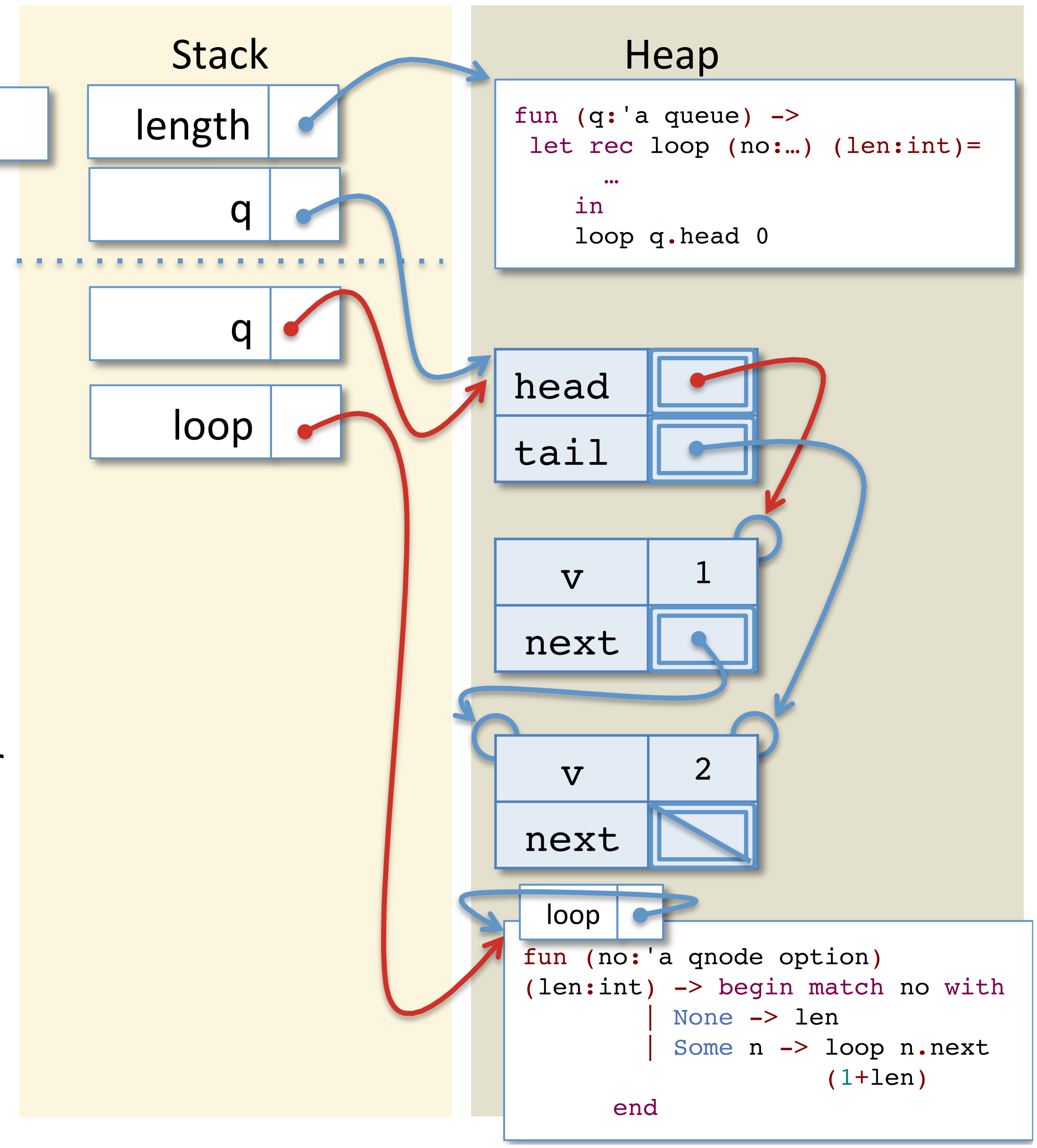
(1) No workspace is saved – there is no need do to that for tail calls

(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)
```

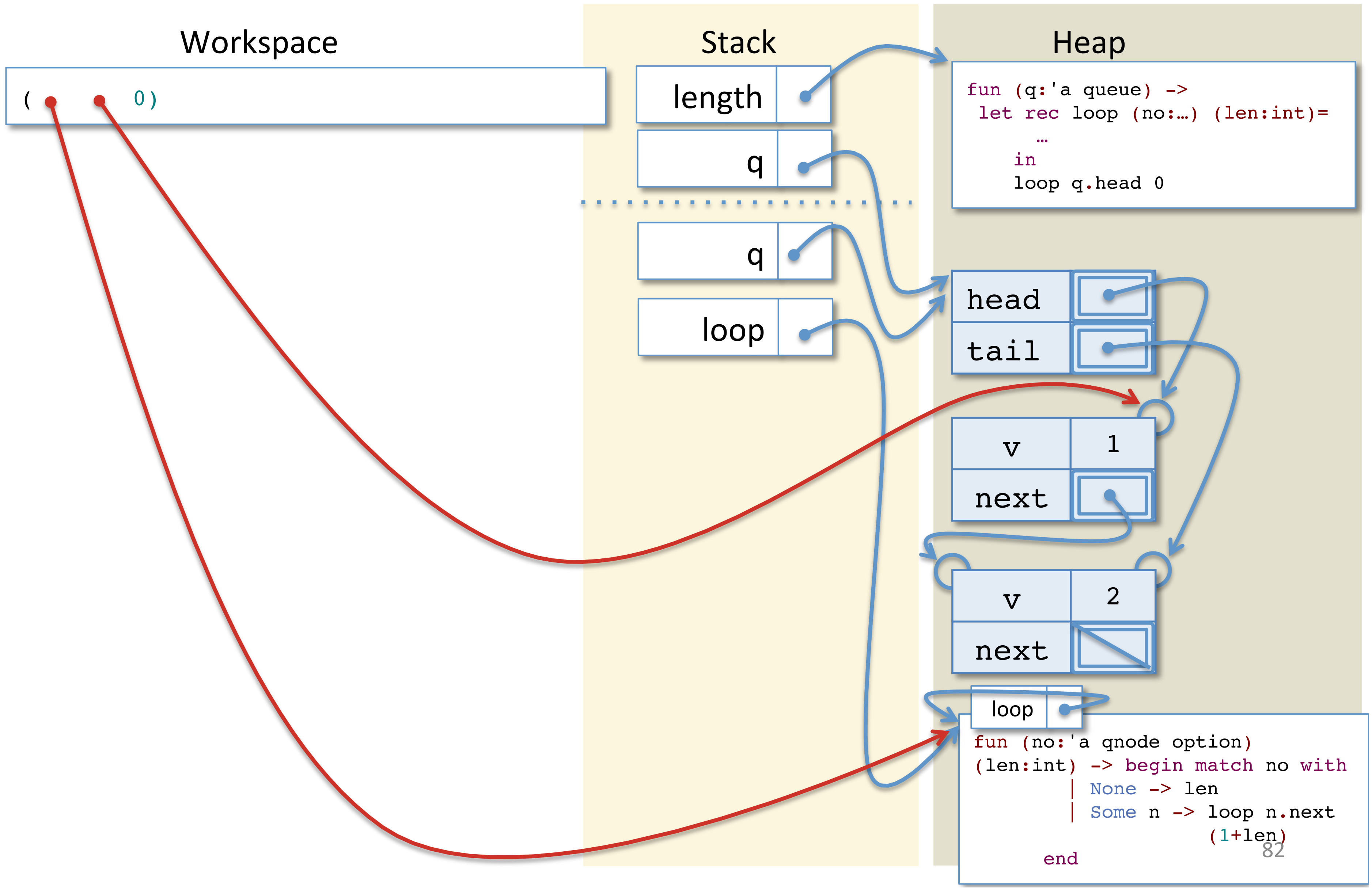


A detail we've been sweeping under the rug until now:

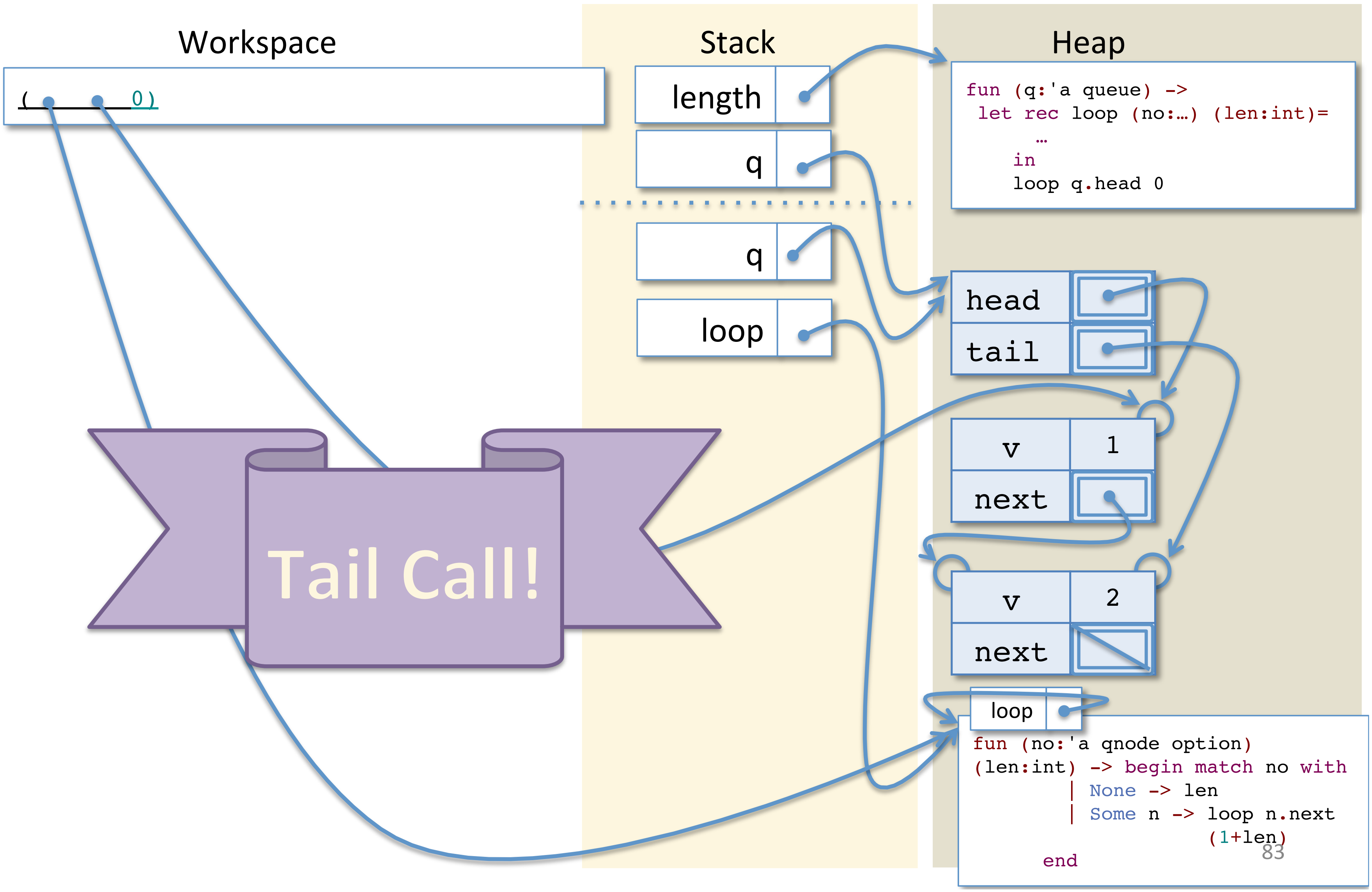
The *closure* of the local recursive function `loop` includes a binding for the loop function itself!

Why? The loop body mentions the loop identifier.

Tail Calls and Iterative length



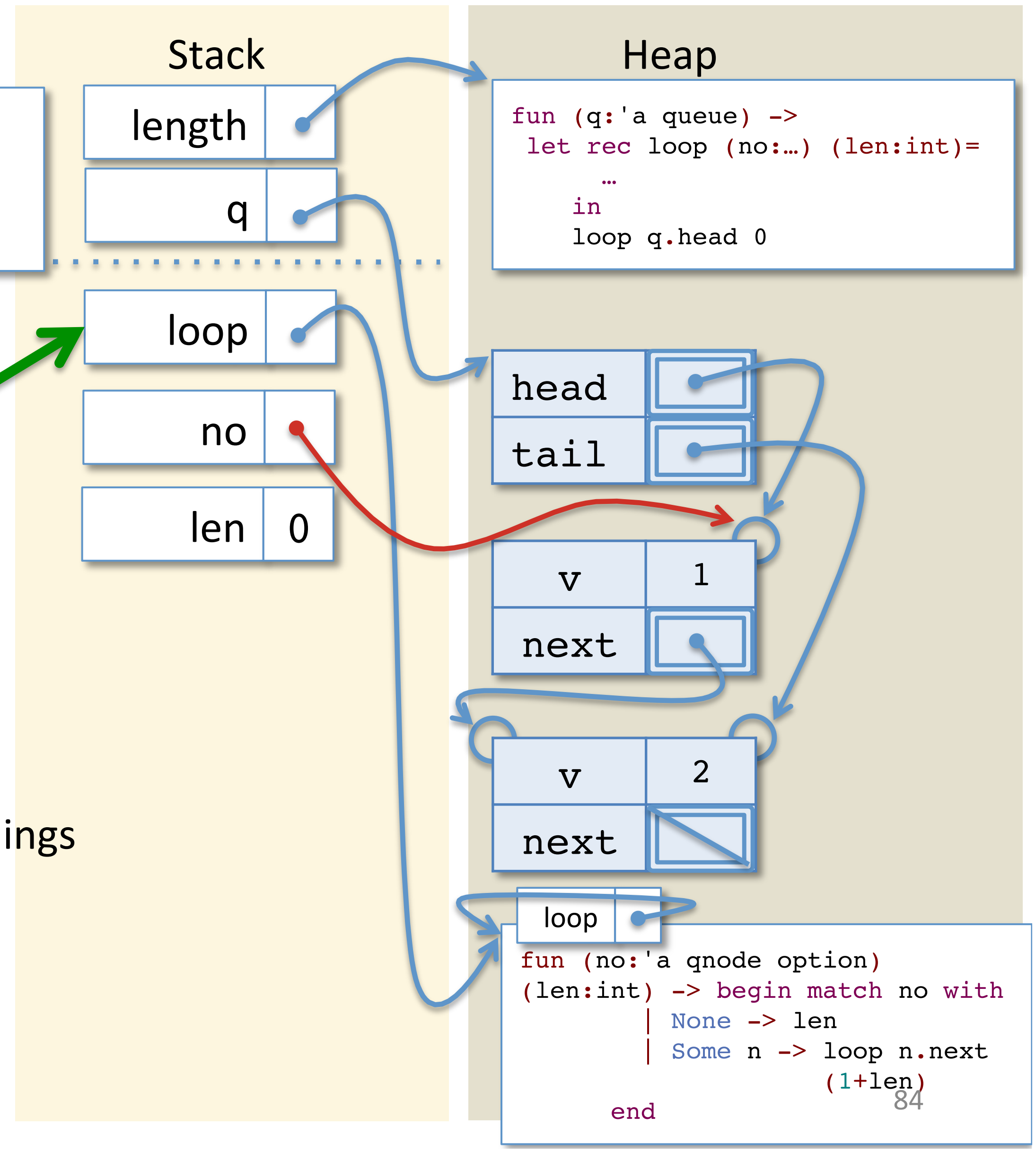
Tail Calls and Iterative length



Tail Calls and Iterative length

Workspace

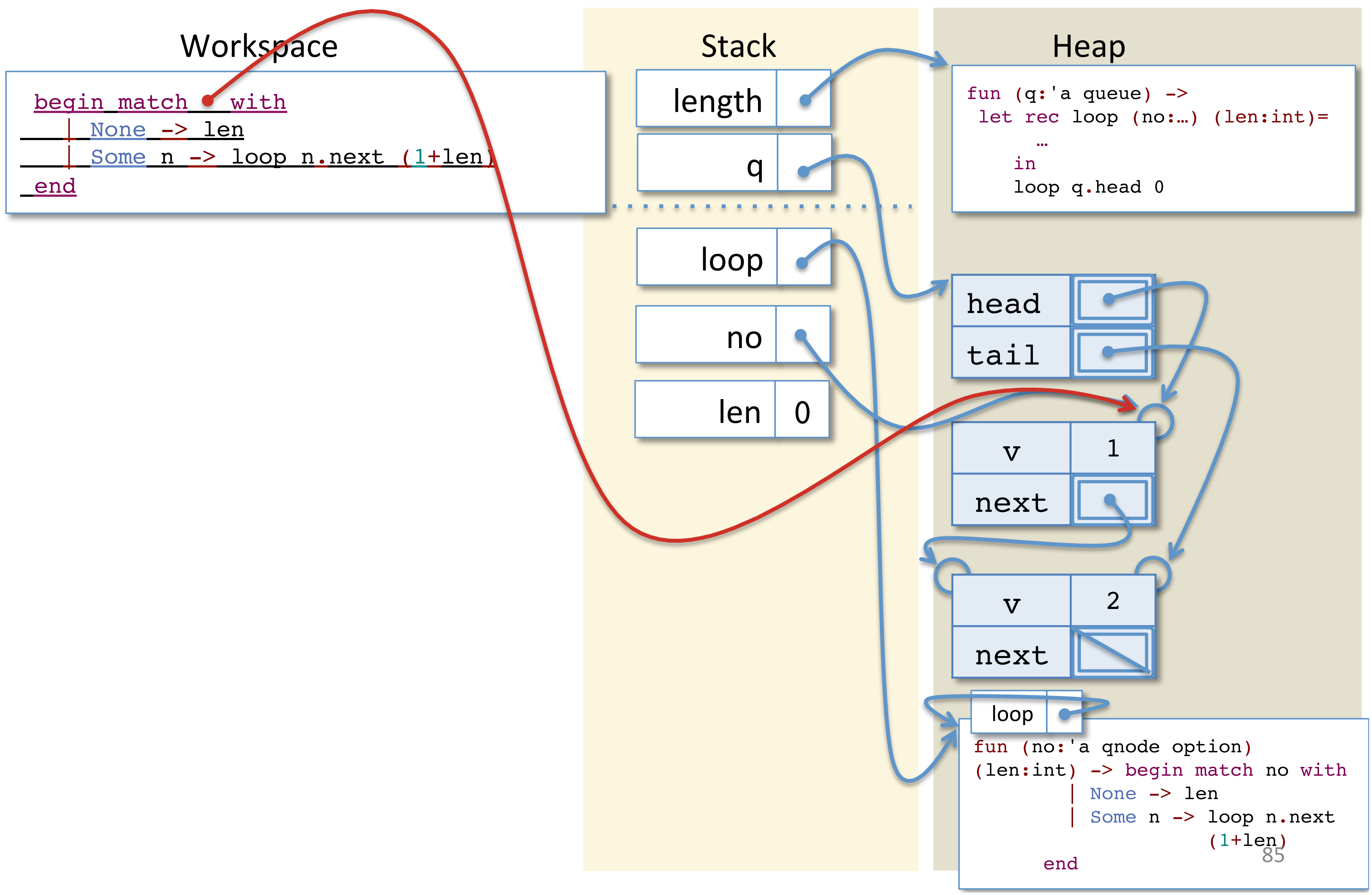
```
begin match no with
   | None -> len
   | Some n -> loop n.next (1+len)
end
```



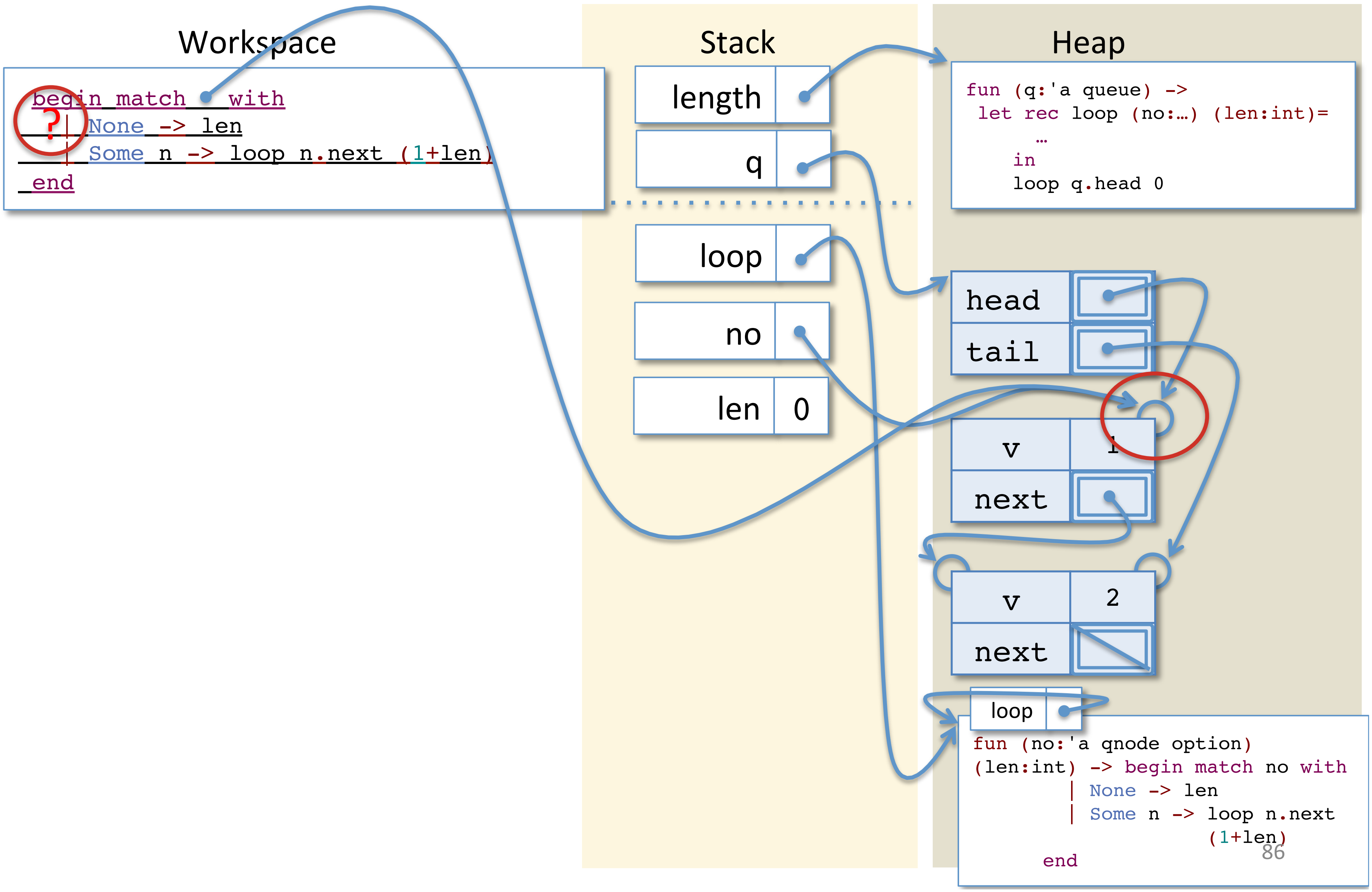
This binding comes from the closure.

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

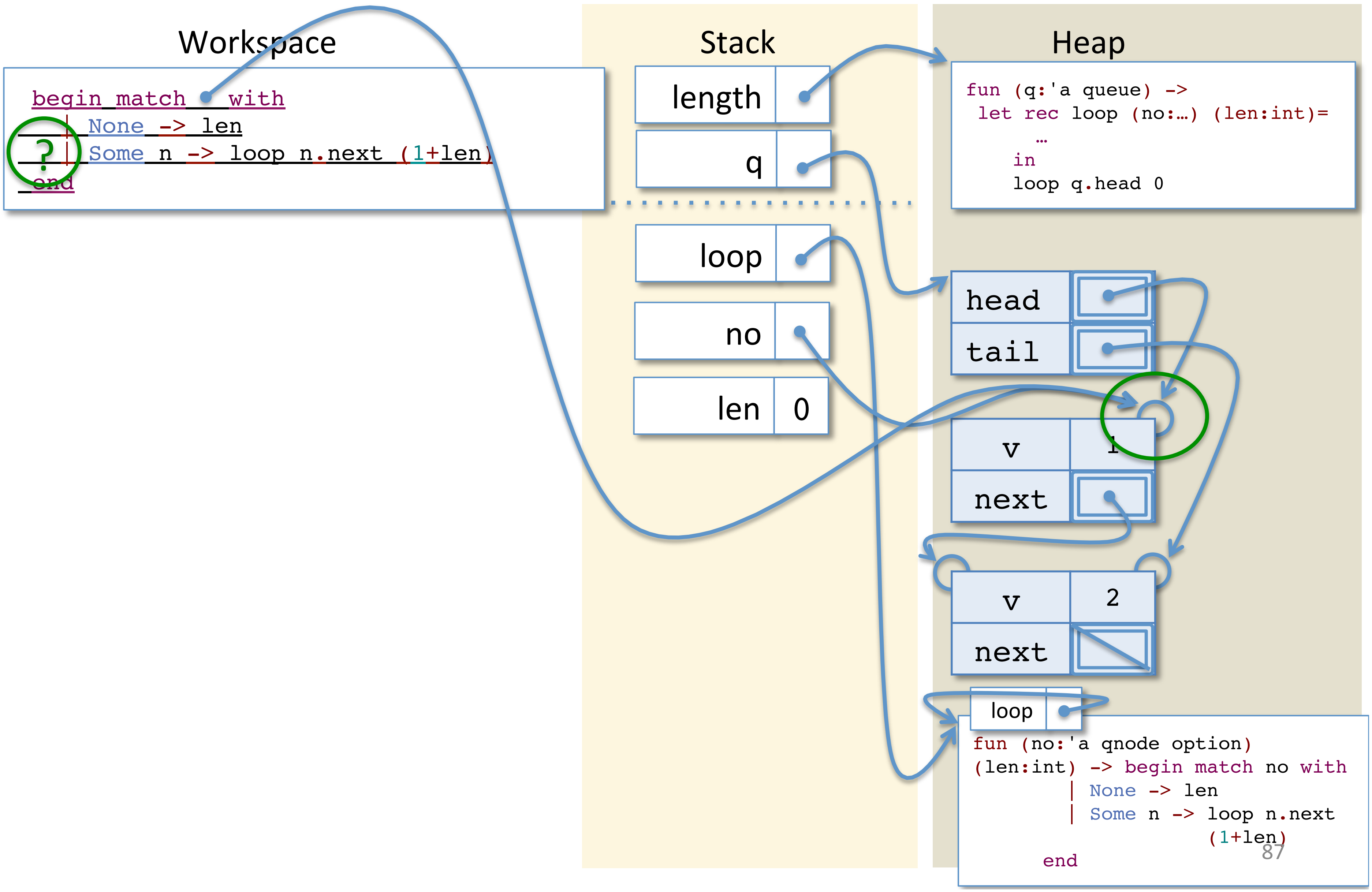
Tail Calls and Iterative length



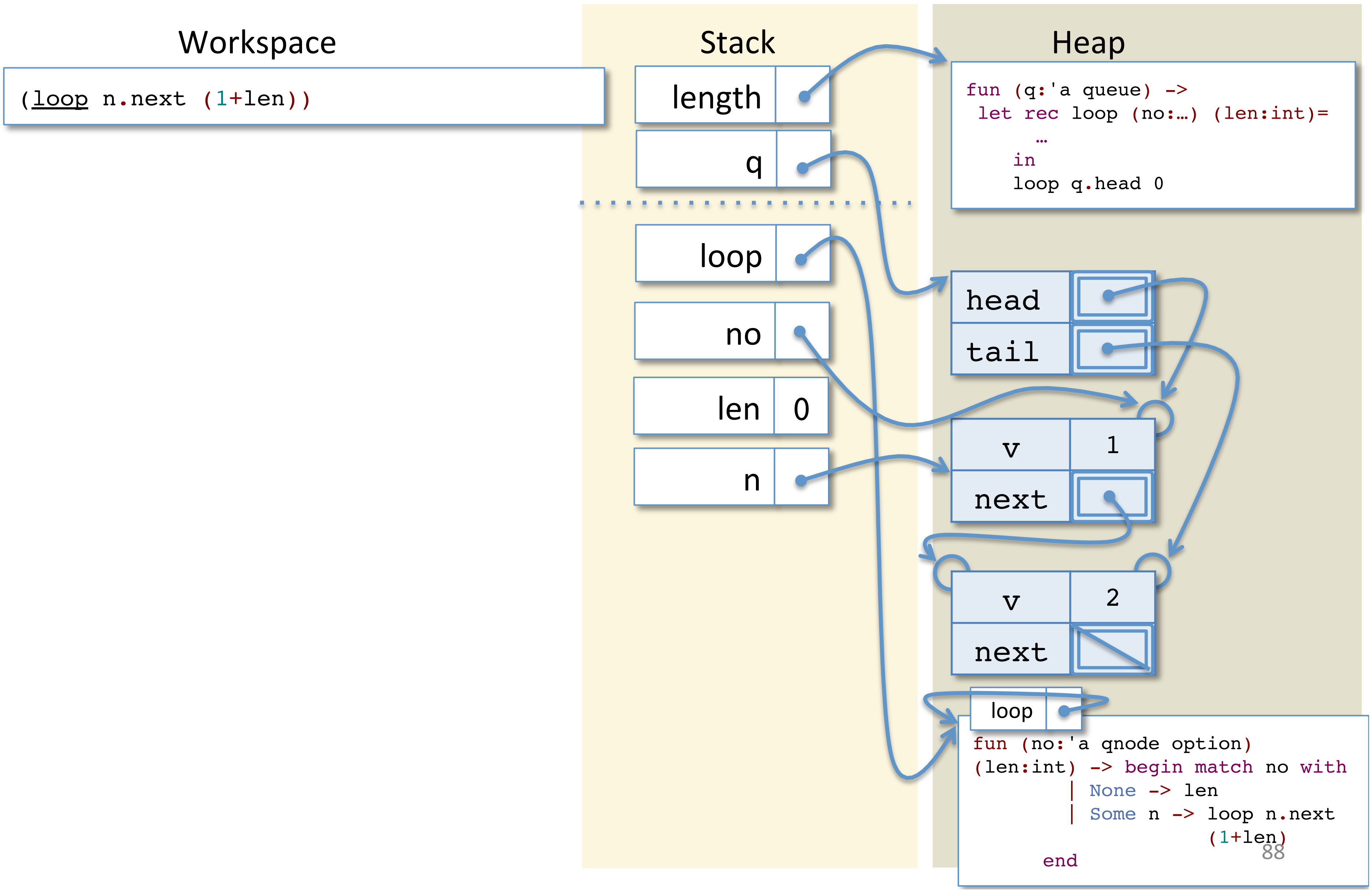
Tail Calls and Iterative length



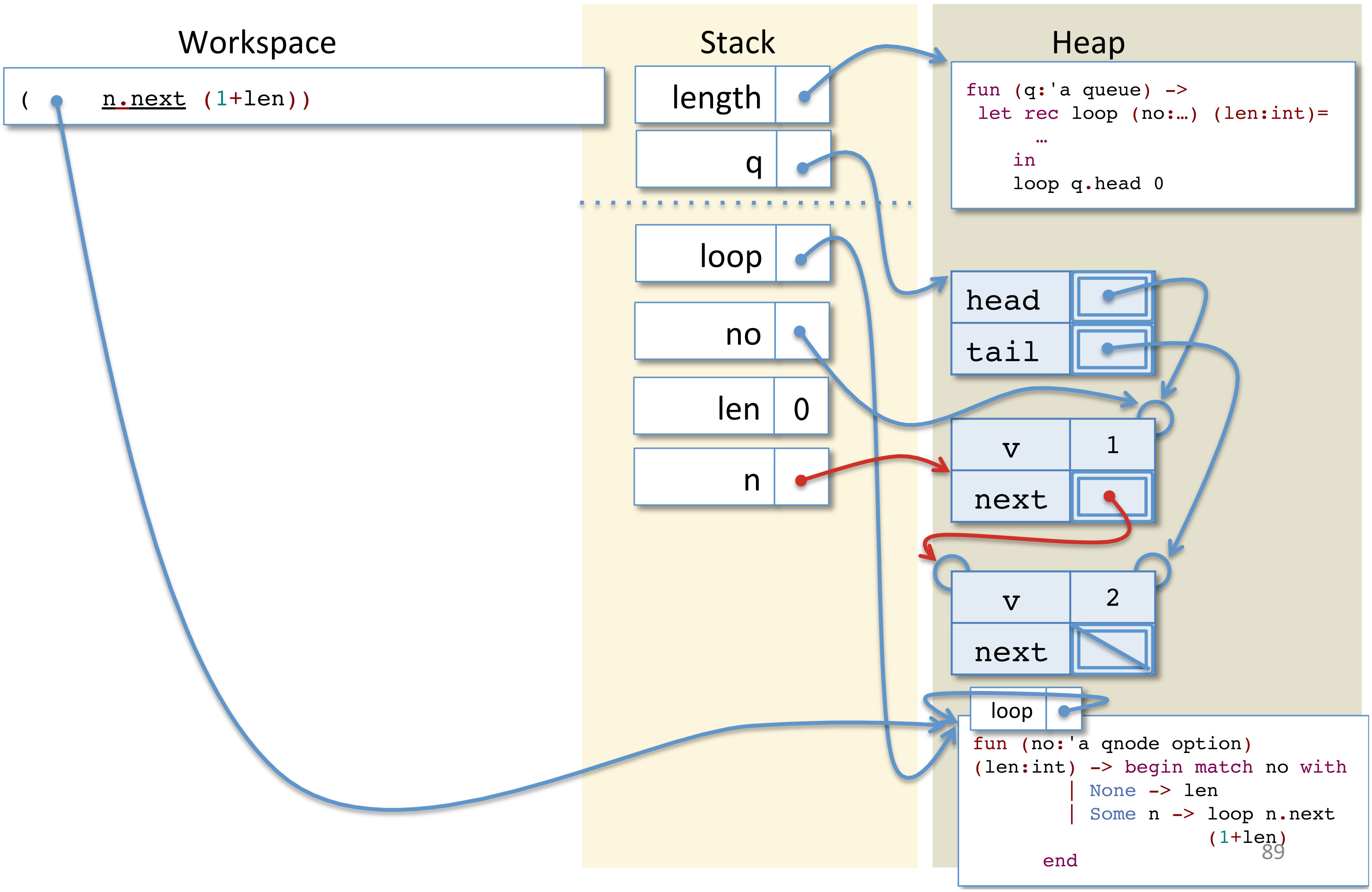
Tail Calls and Iterative length



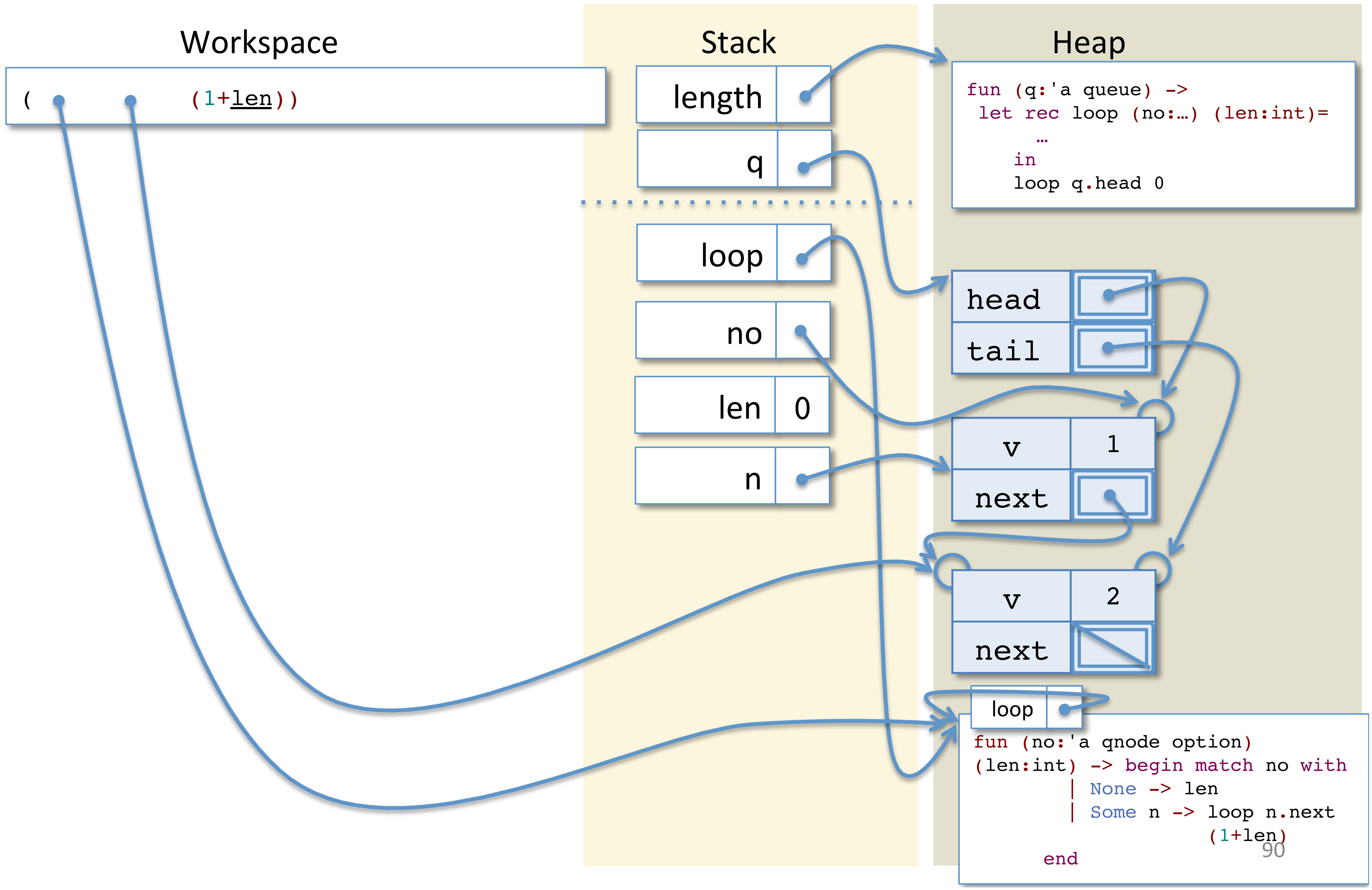
Tail Calls and Iterative length



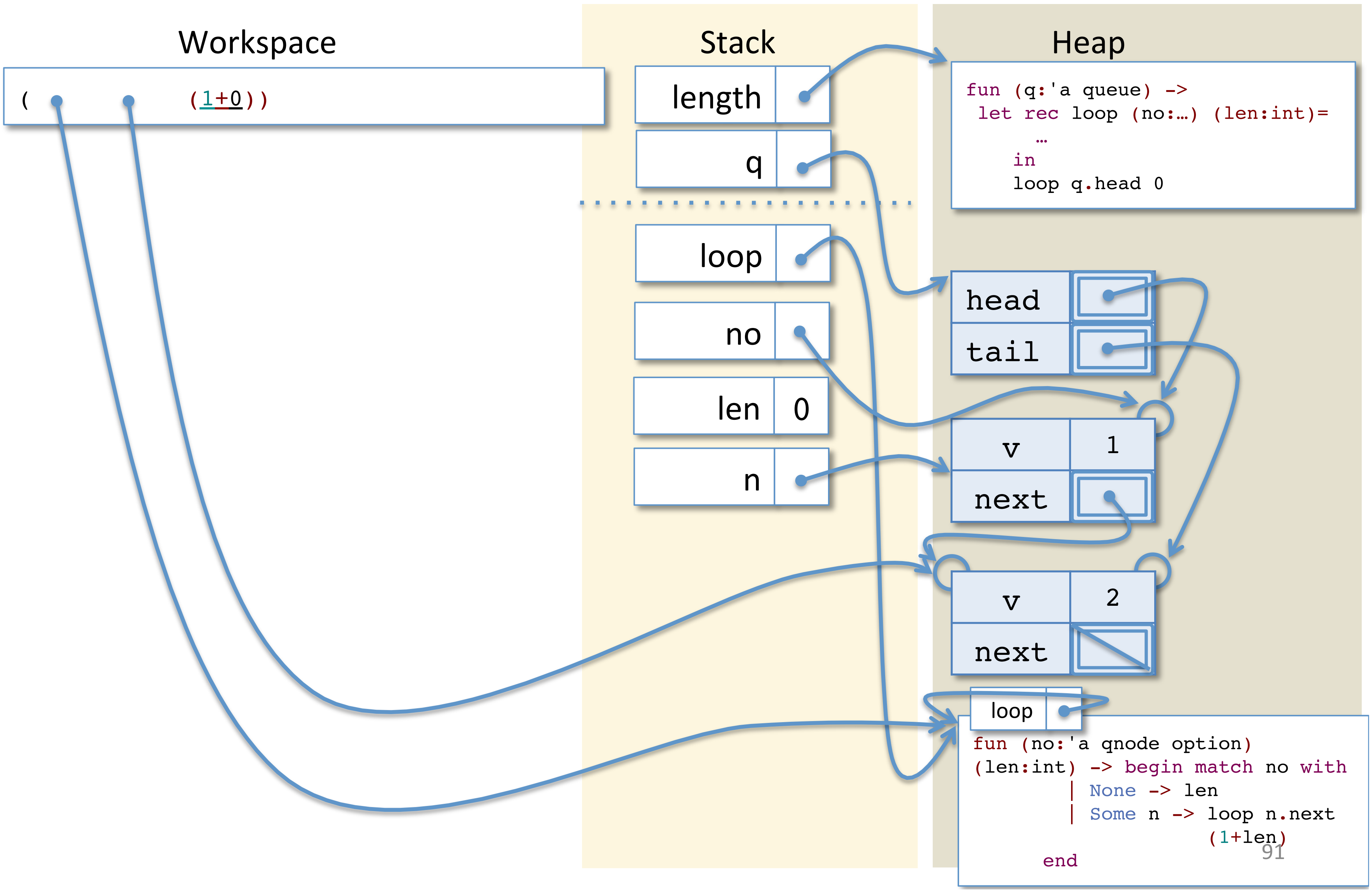
Tail Calls and Iterative length



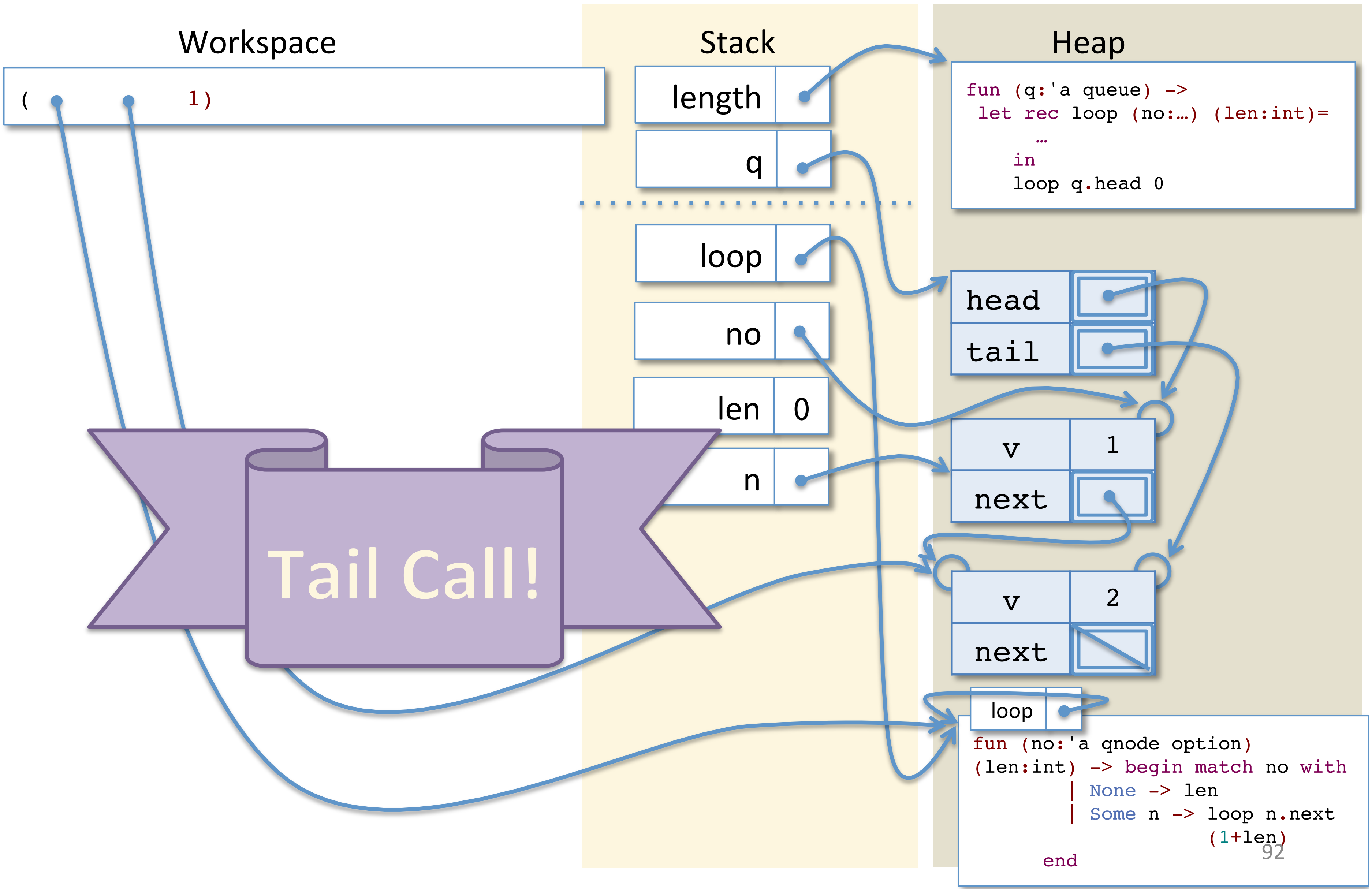
Tail Calls and Iterative length



Tail Calls and Iterative length



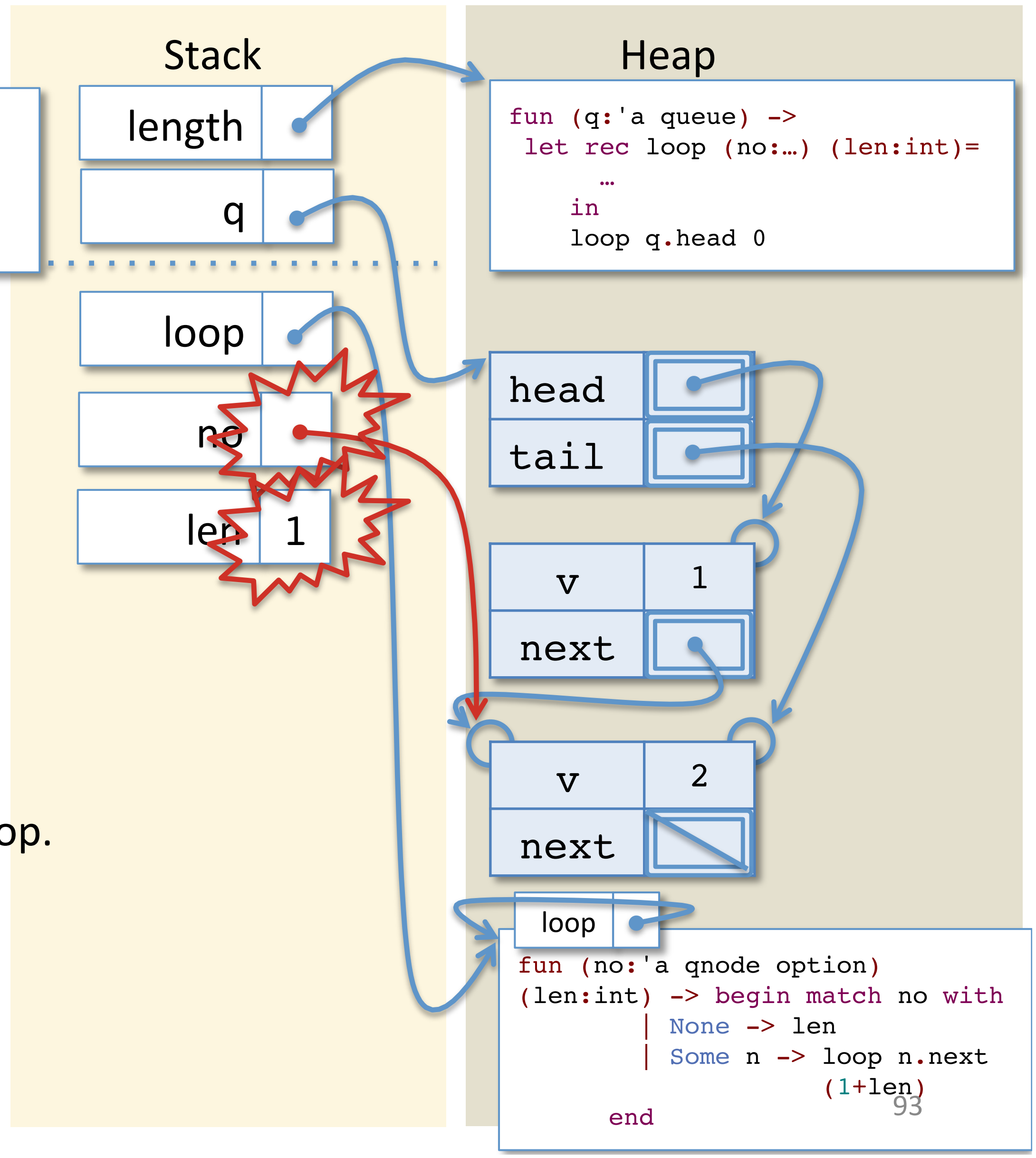
Tail Calls and Iterative length



Tail Calls and Iterative length

Workspace

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

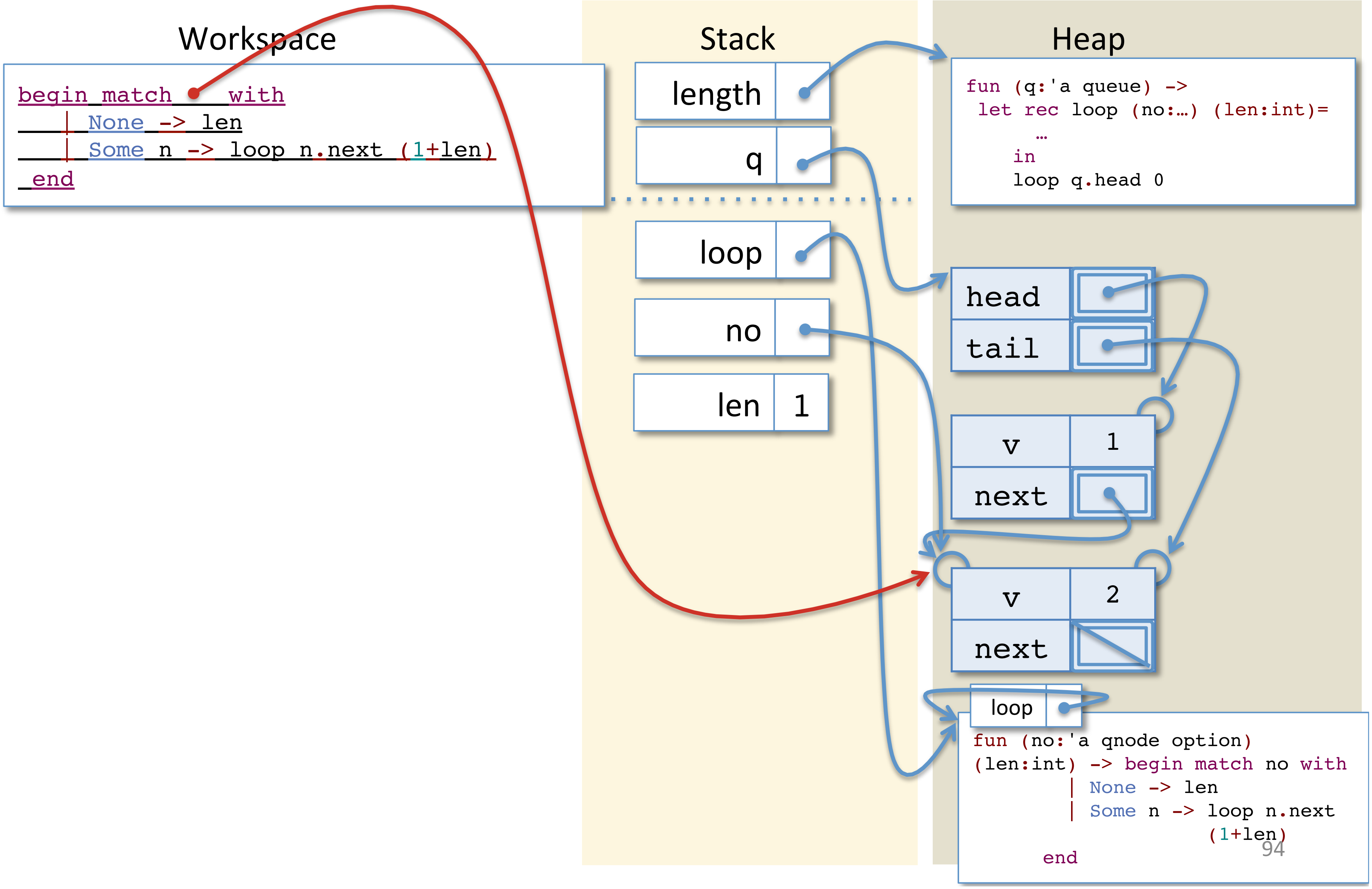


Note: we popped the old values of loop, no, len, and n when we did the tail call. Then we pushed the new values of loop, no, and len.

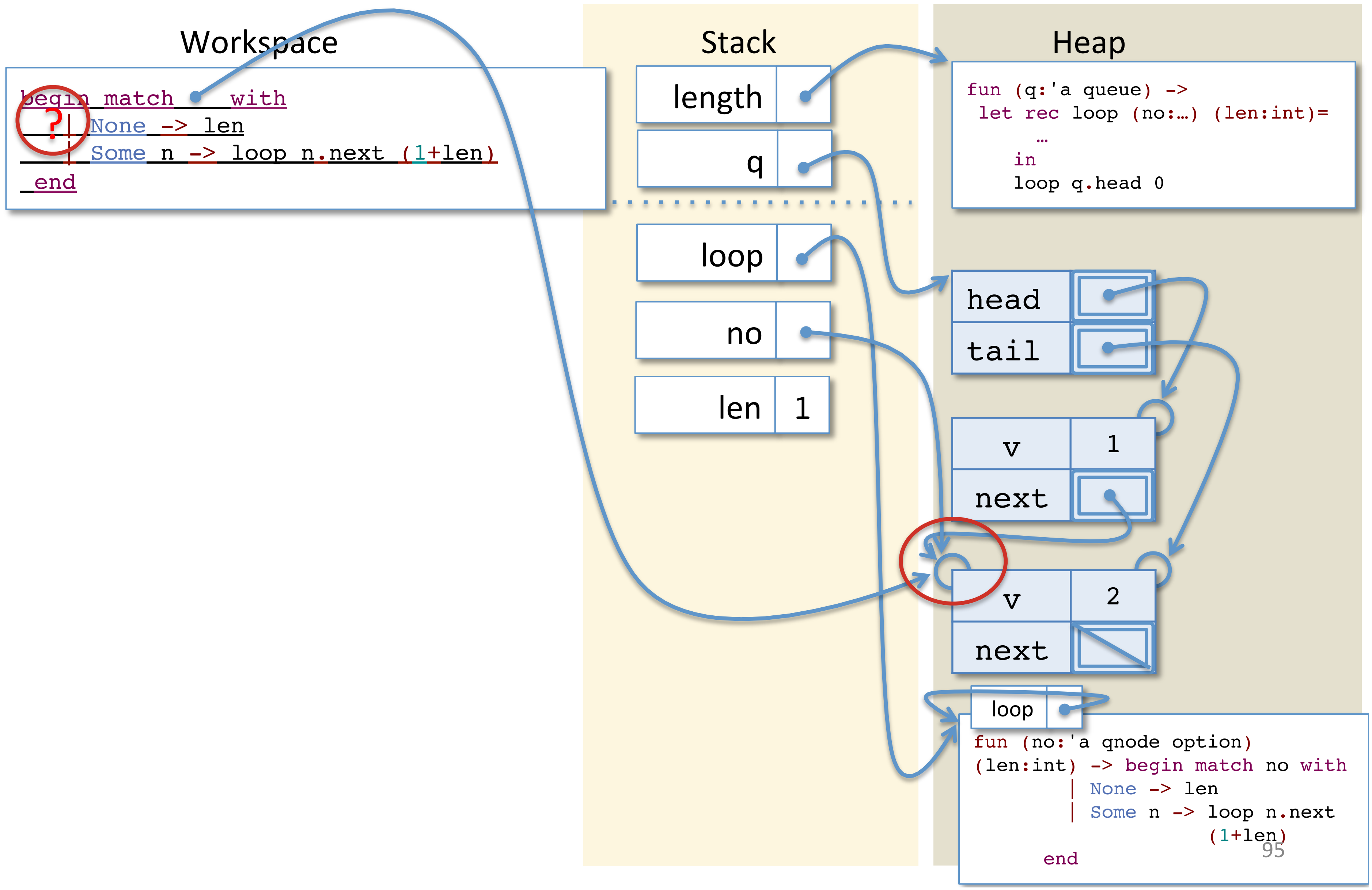
This leaves the stack in almost the same shape as when we first called loop.

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

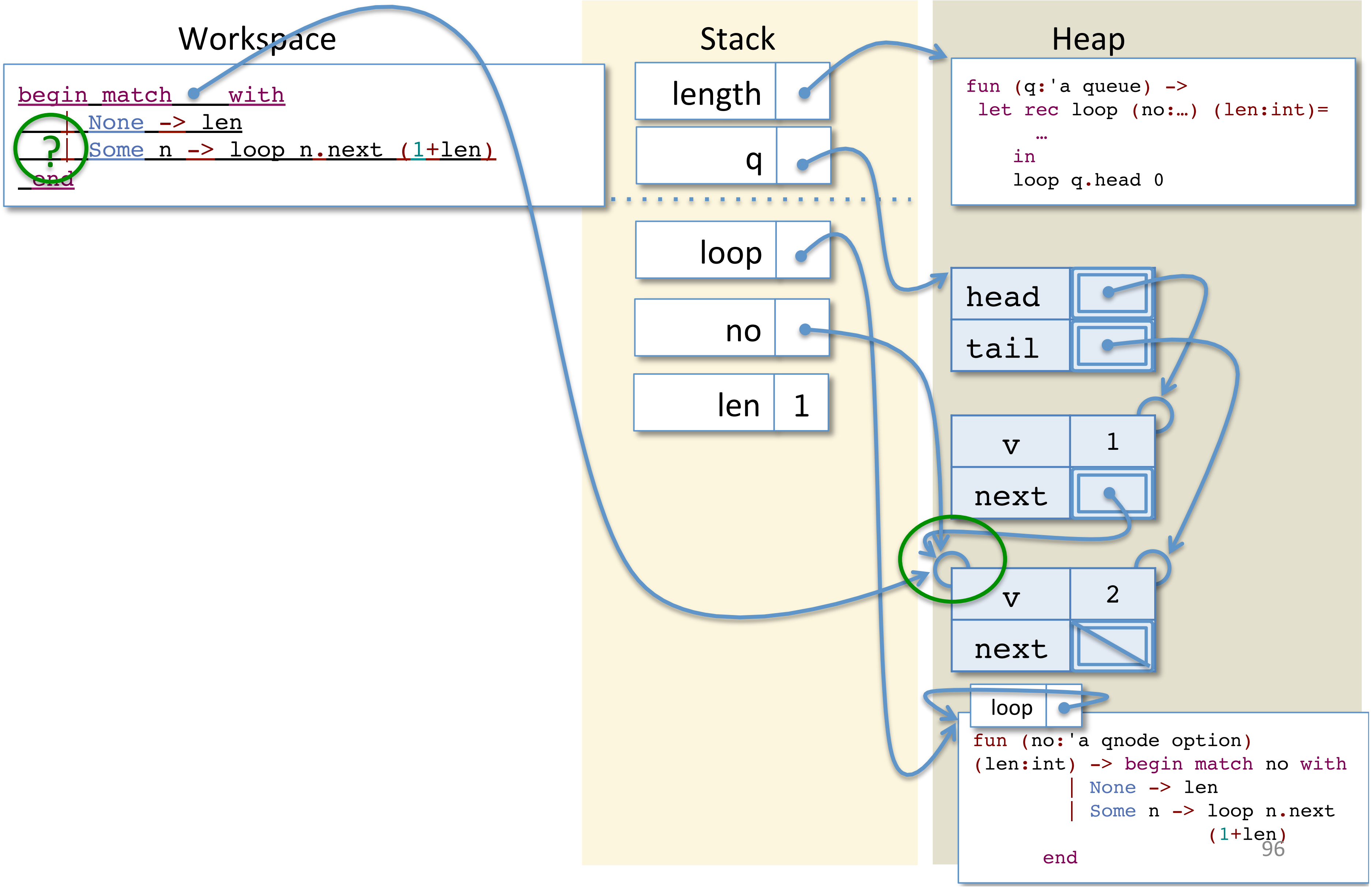
Tail Calls and Iterative length



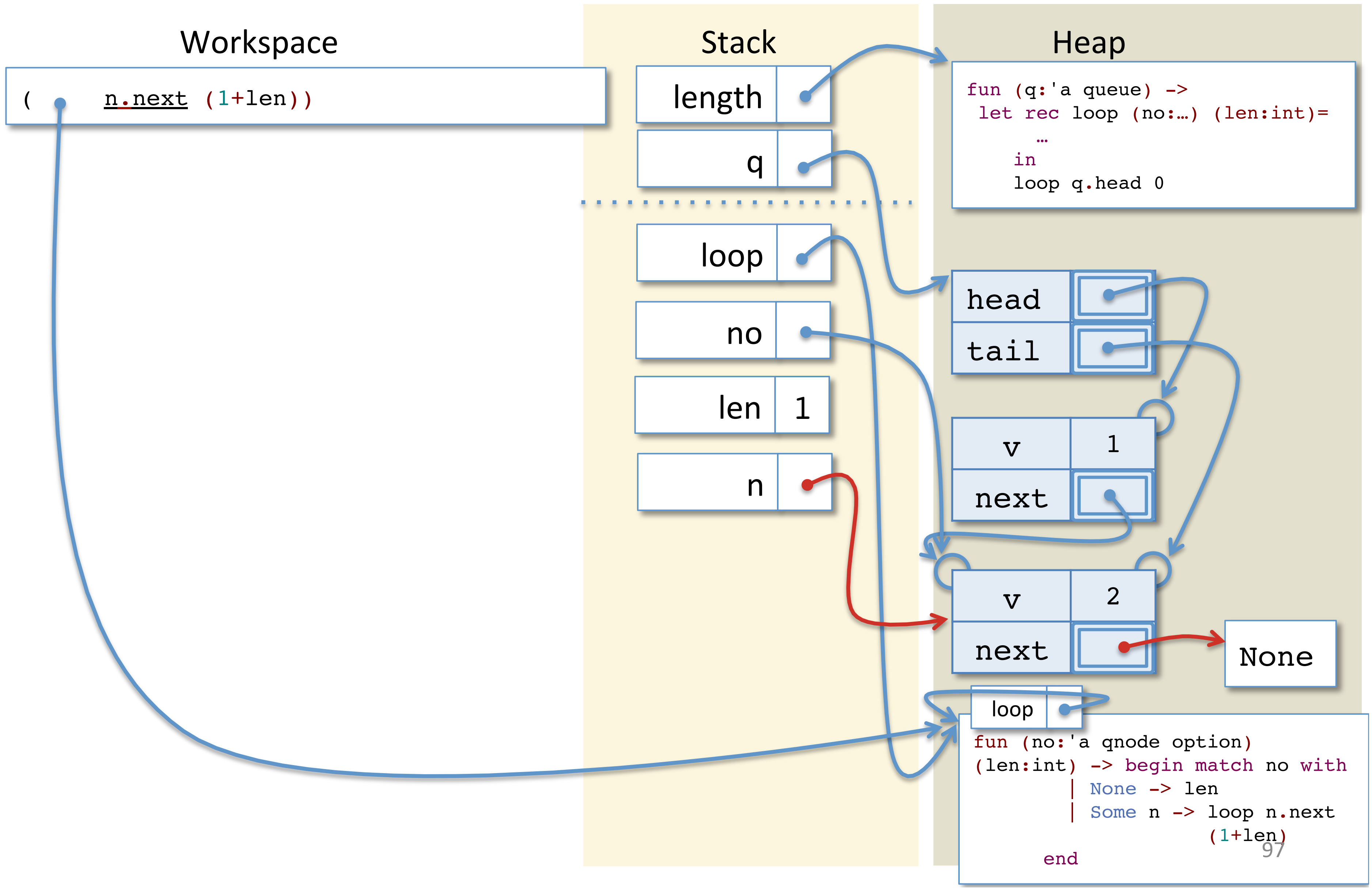
Tail Calls and Iterative length



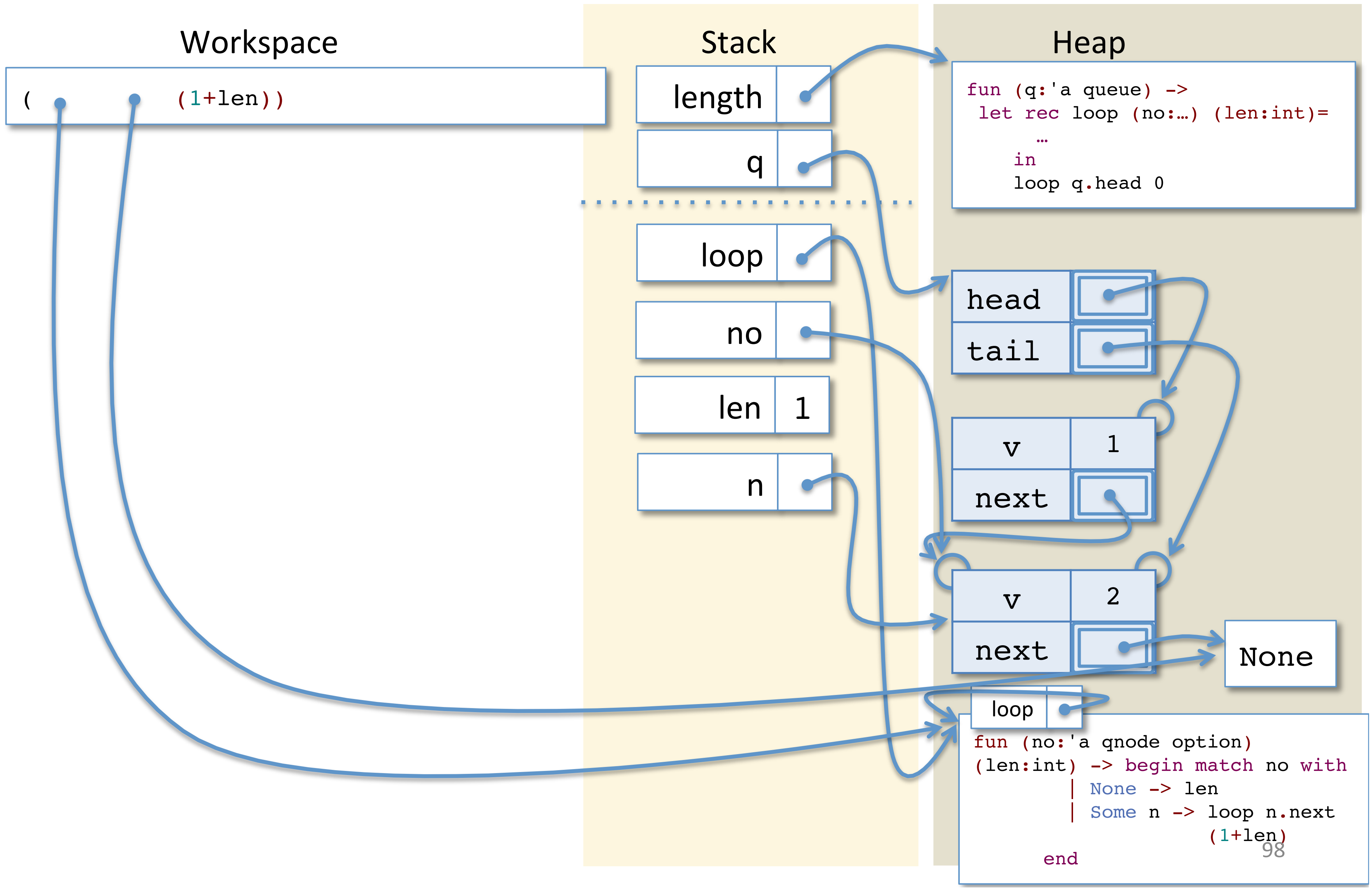
Tail Calls and Iterative length



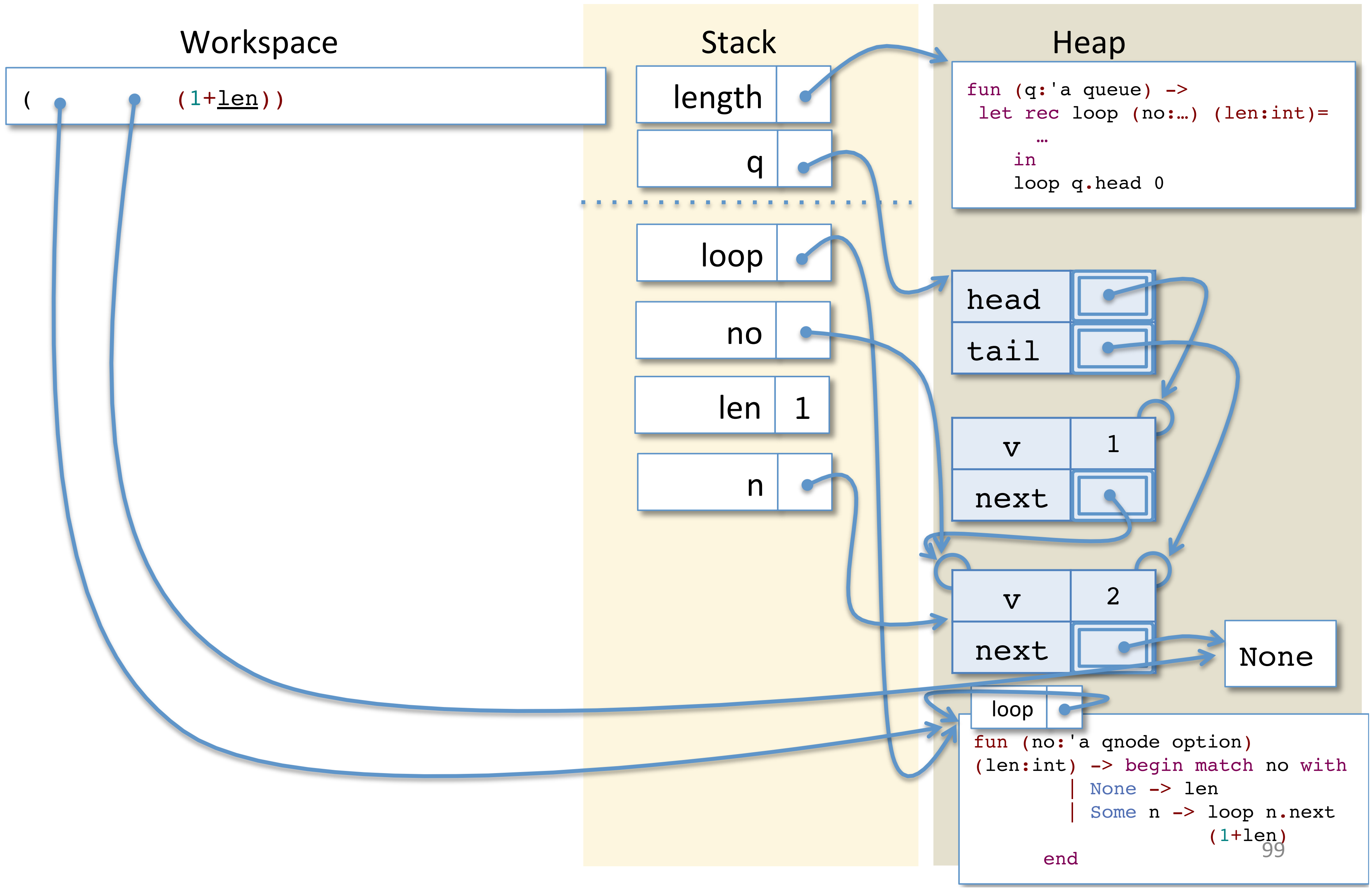
Tail Calls and Iterative length



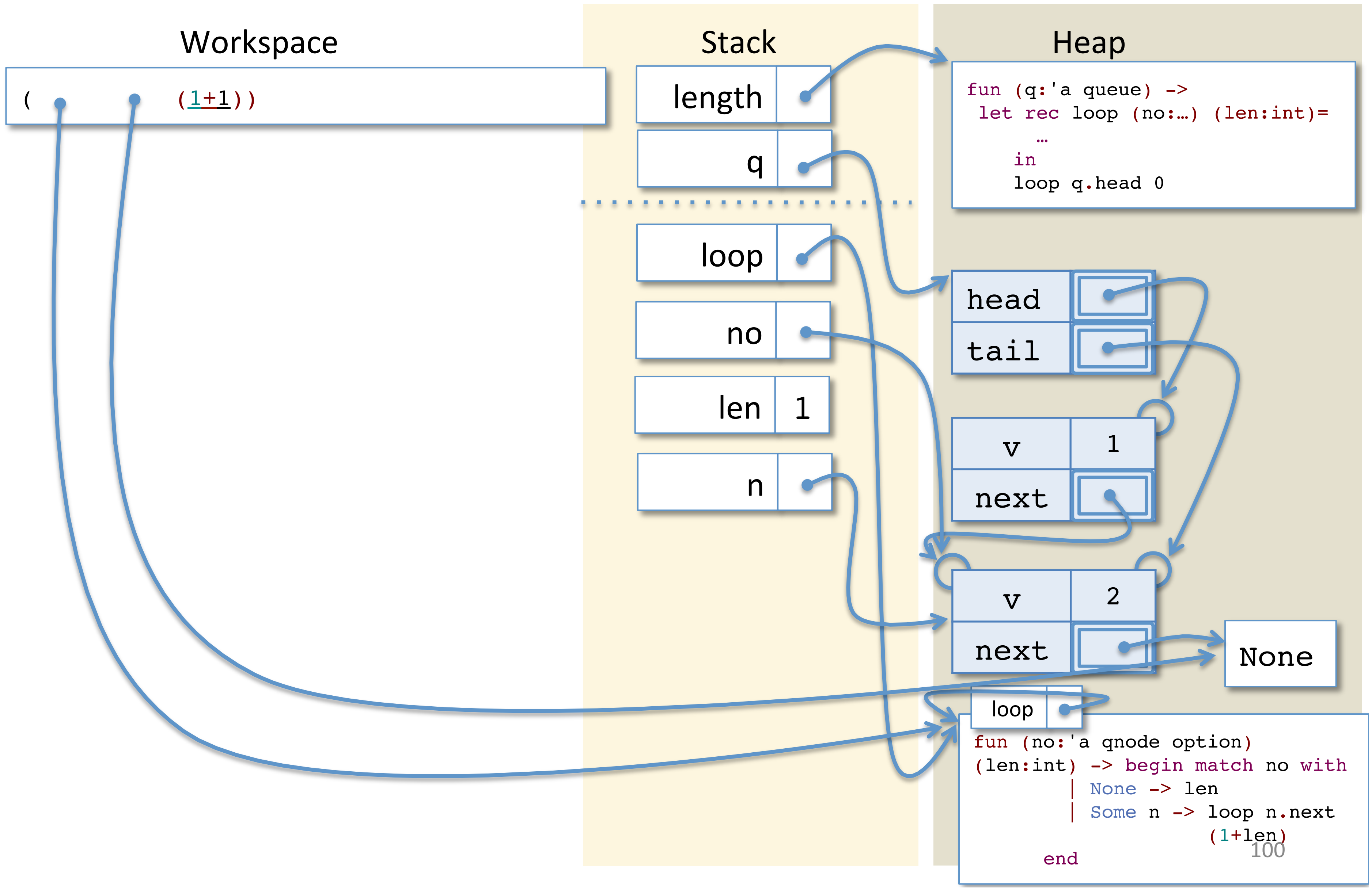
Tail Calls and Iterative length



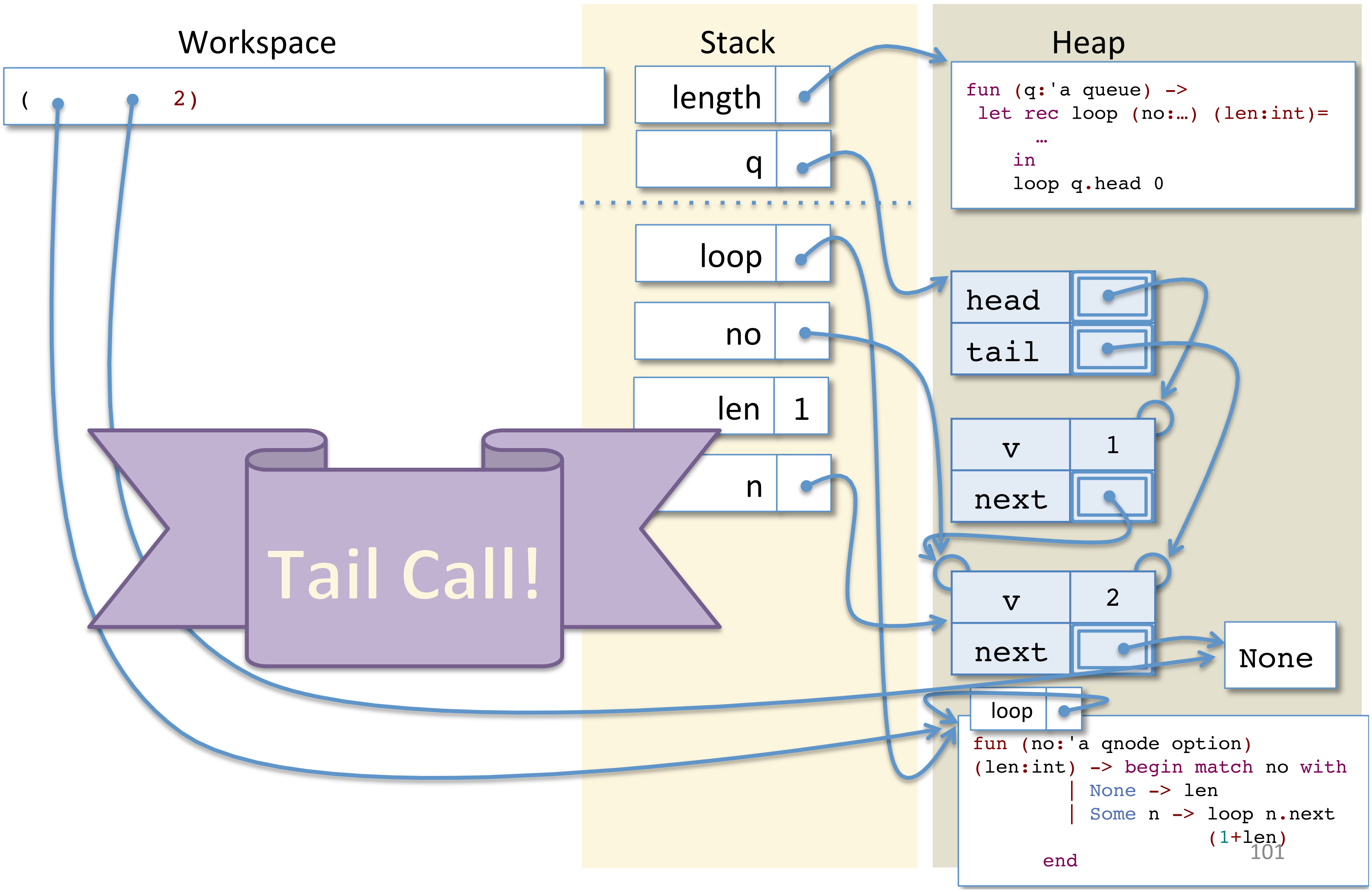
Tail Calls and Iterative length



Tail Calls and Iterative length



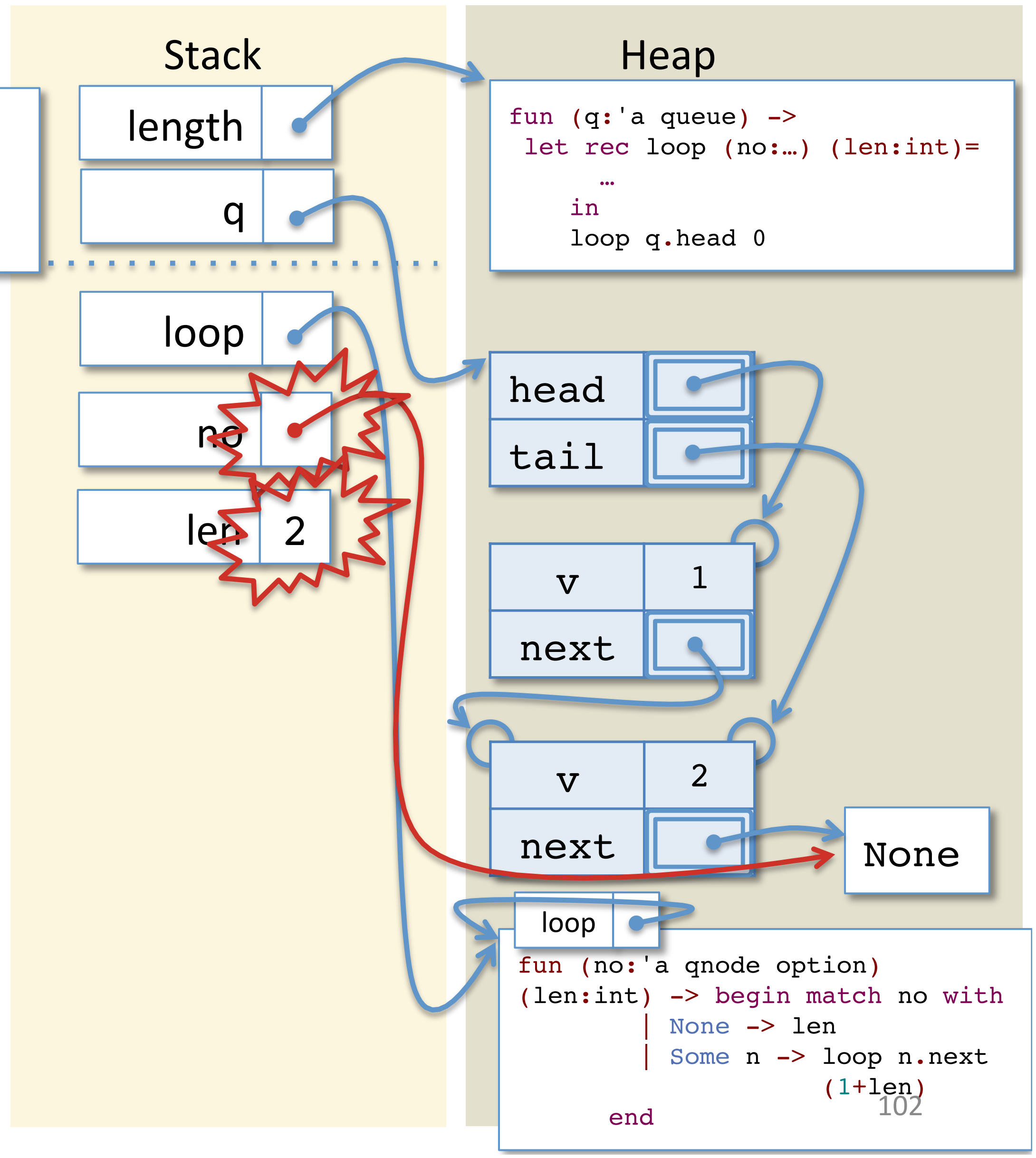
Tail Calls and Iterative length



Tail Calls and Iterative length

Workspace

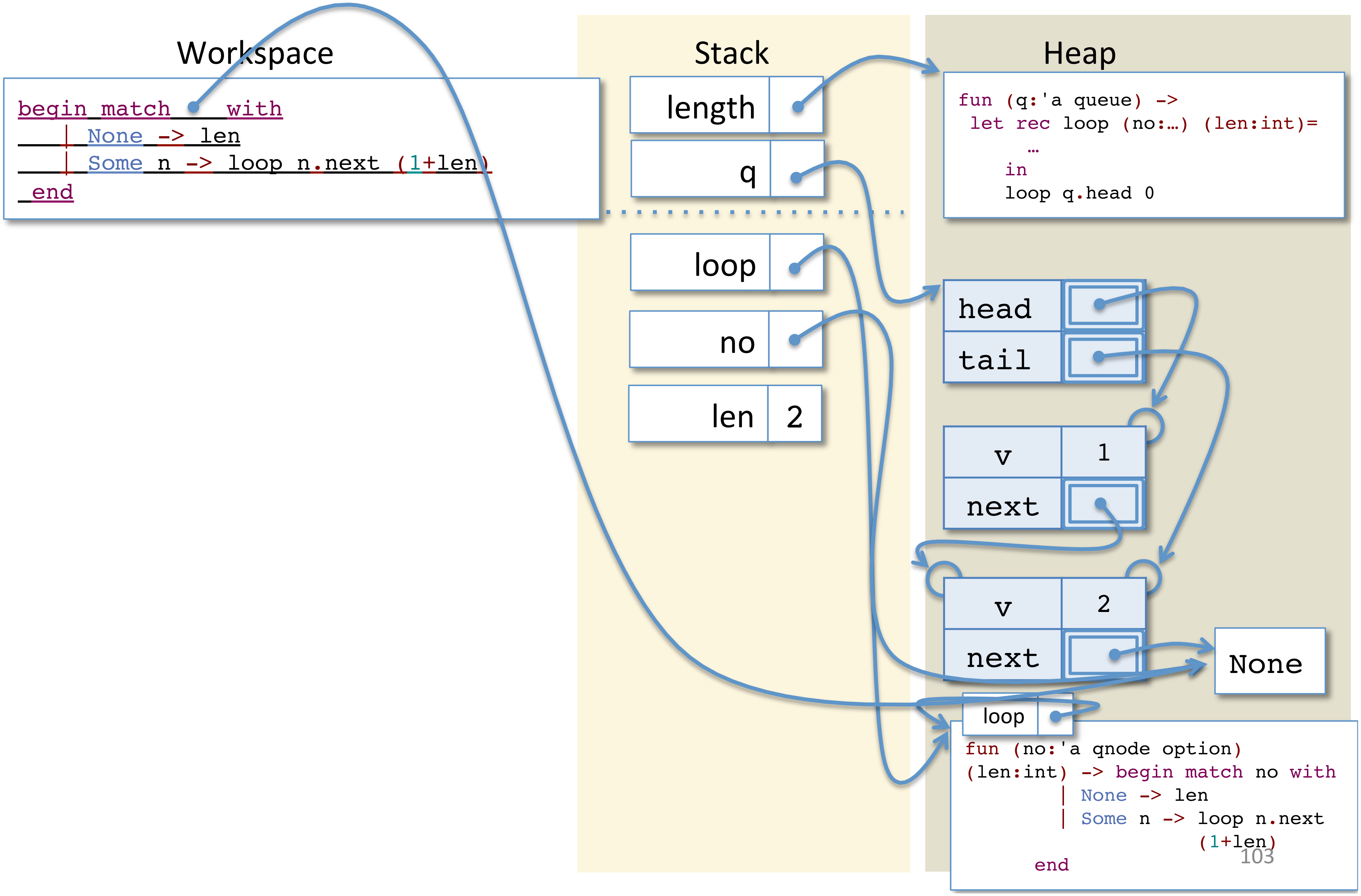
```
begin match no with
    | None -> len
    | Some n -> loop n.next (1+len)
 end
```



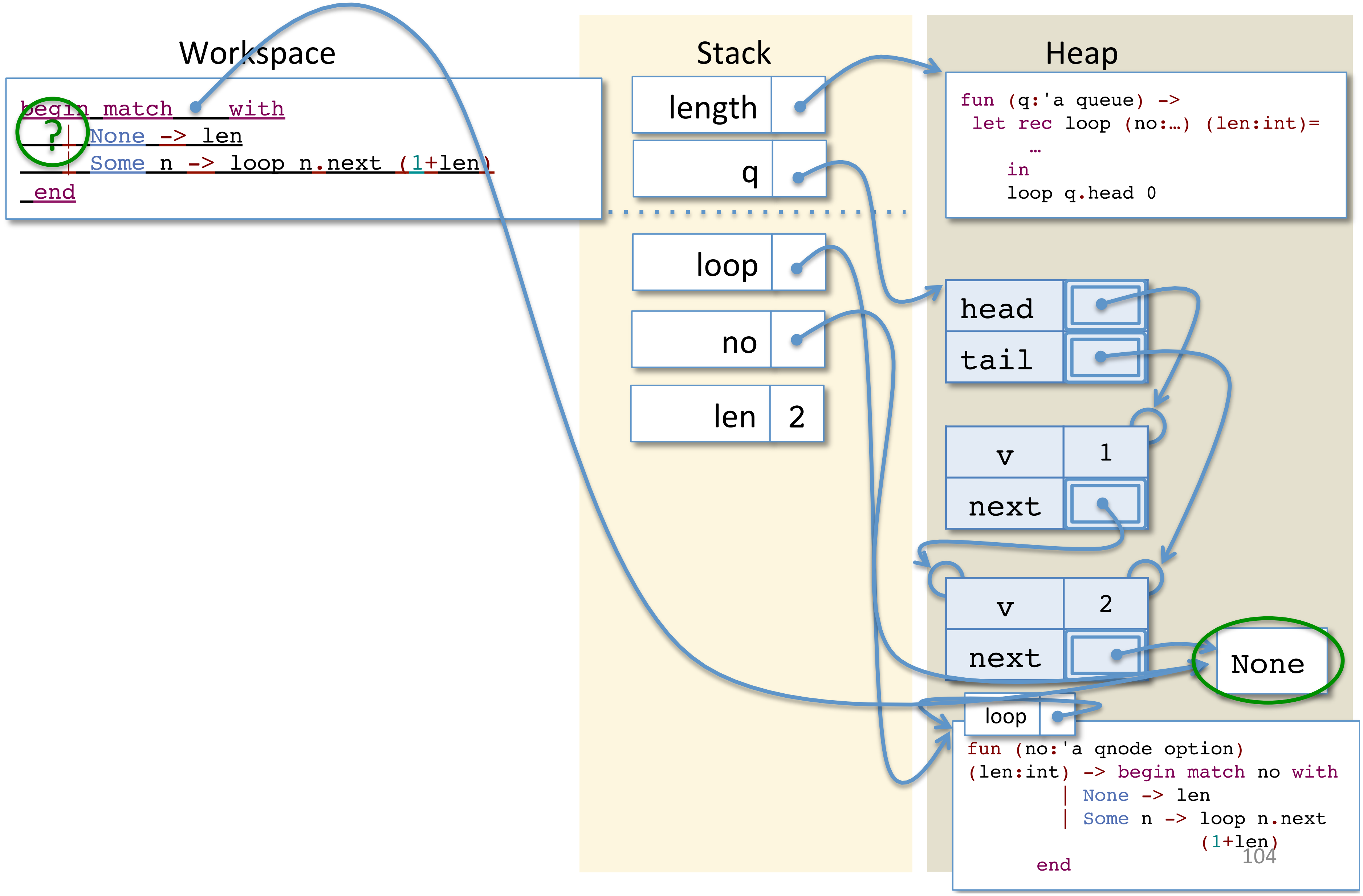
Note: Again, the tail call leaves the stack as before, but effectively updates the values of no and len.

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

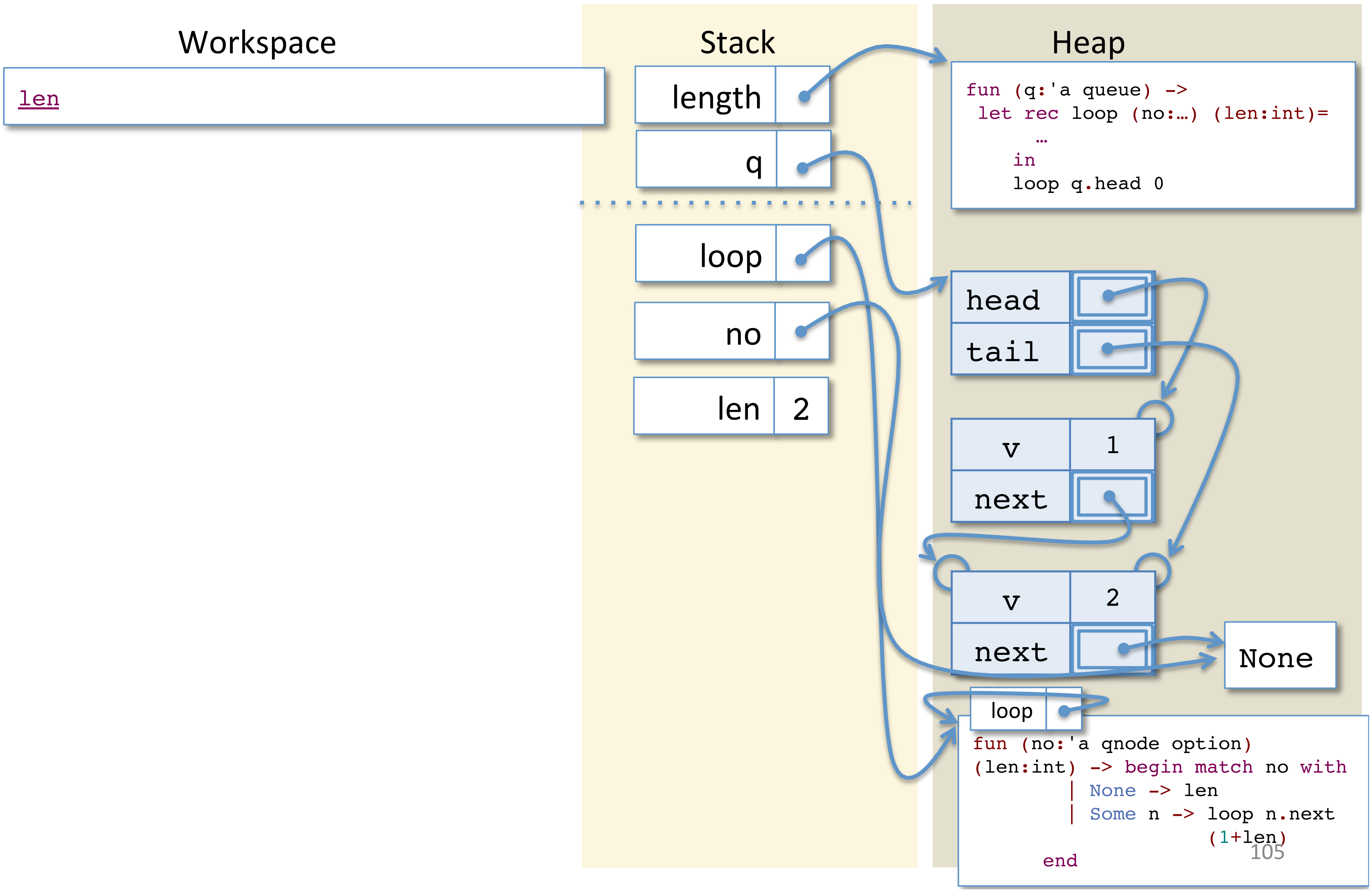
Tail Calls and Iterative length



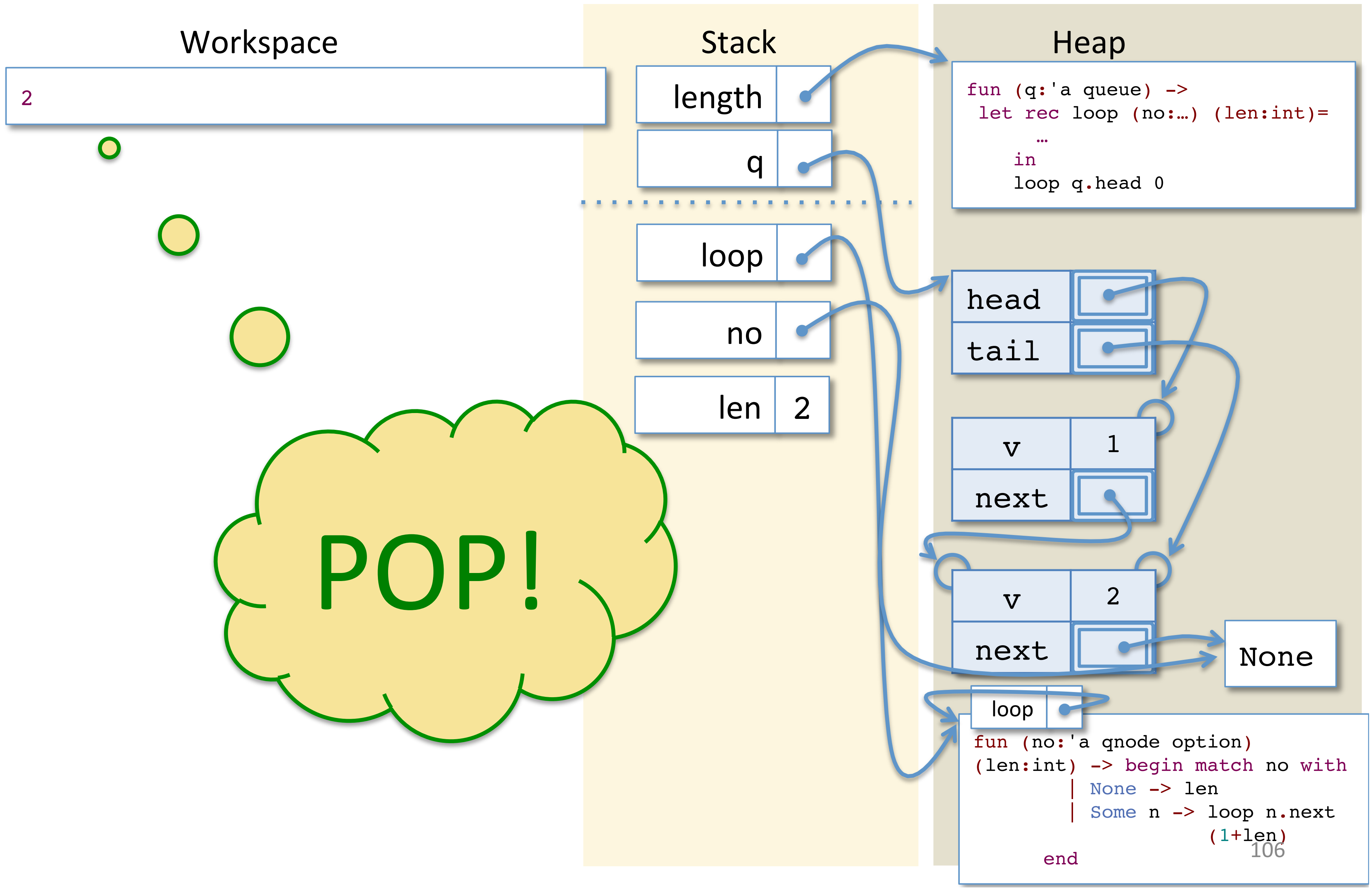
Tail Calls and Iterative length



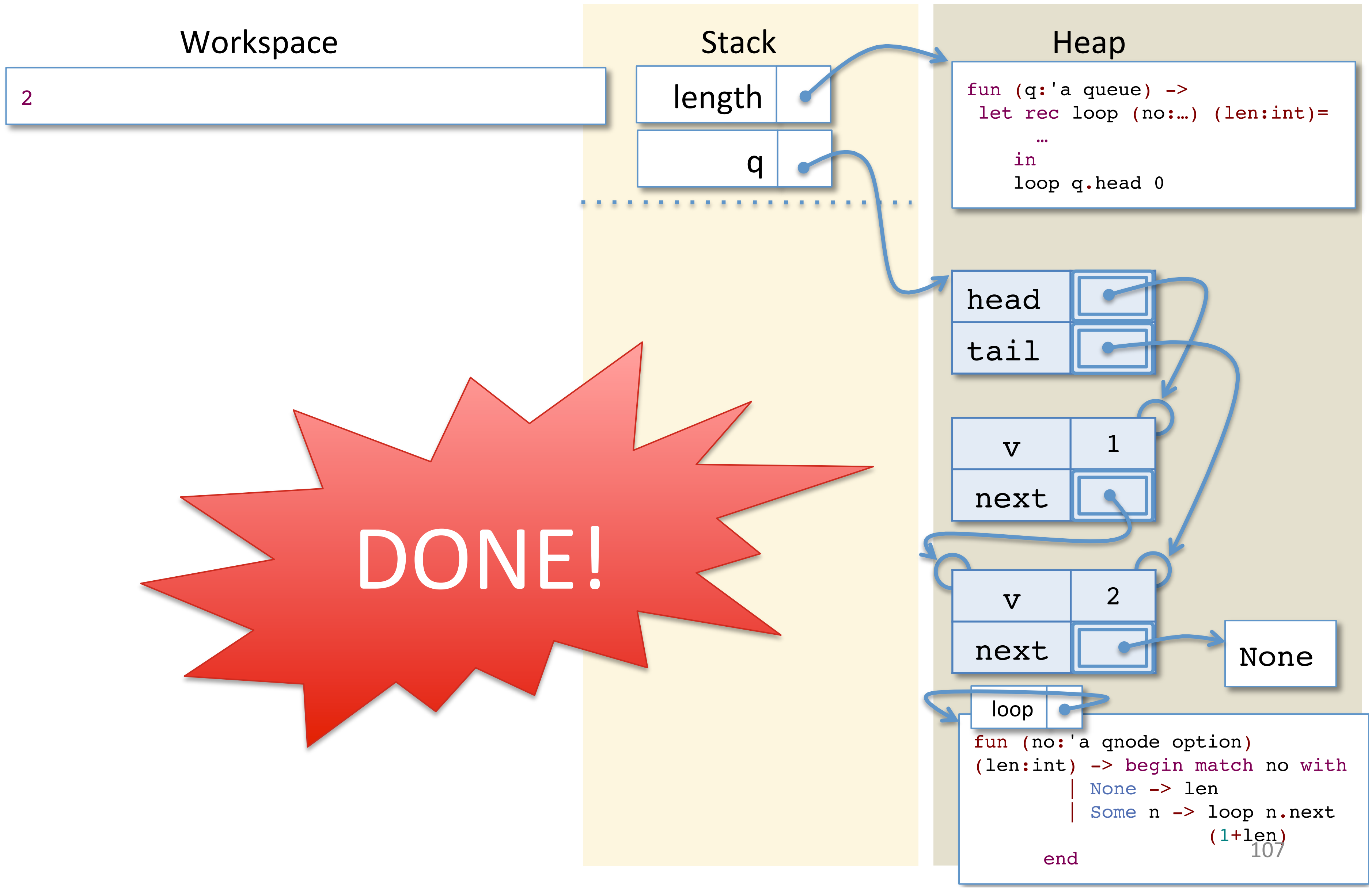
Tail Calls and Iterative length



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…