Programming Languages and Techniques (CIS120)

Lecture 16

Feb 26, 2014

Linked Queues: Iteration

Announcements

- View midterm exams with Ms. Laura Fox (Levine 308, 9-4)
  - If you want a *copy* of your exam, let her know by Thursday 4PM
  - Solution posted on course website
- Read Ch. 16 & 17 of lecture notes
- Homework 5 due Friday at Midnight

Mutable Queues

singly linked data structures

(Mutable) Queue Interface

```
module type QUEUE =
sig
  (* type of the data structure *)
  type 'a queue

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

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


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

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

end
```

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.

Implementing Linked Queues

LinkedQ.ml

create and is_empty

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


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

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

enq

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

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

Is this function correct? (This is the code from the end of class last time, it passed our tests.)

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

1. Yes
2. No
3. I can’t tell

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 "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

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)

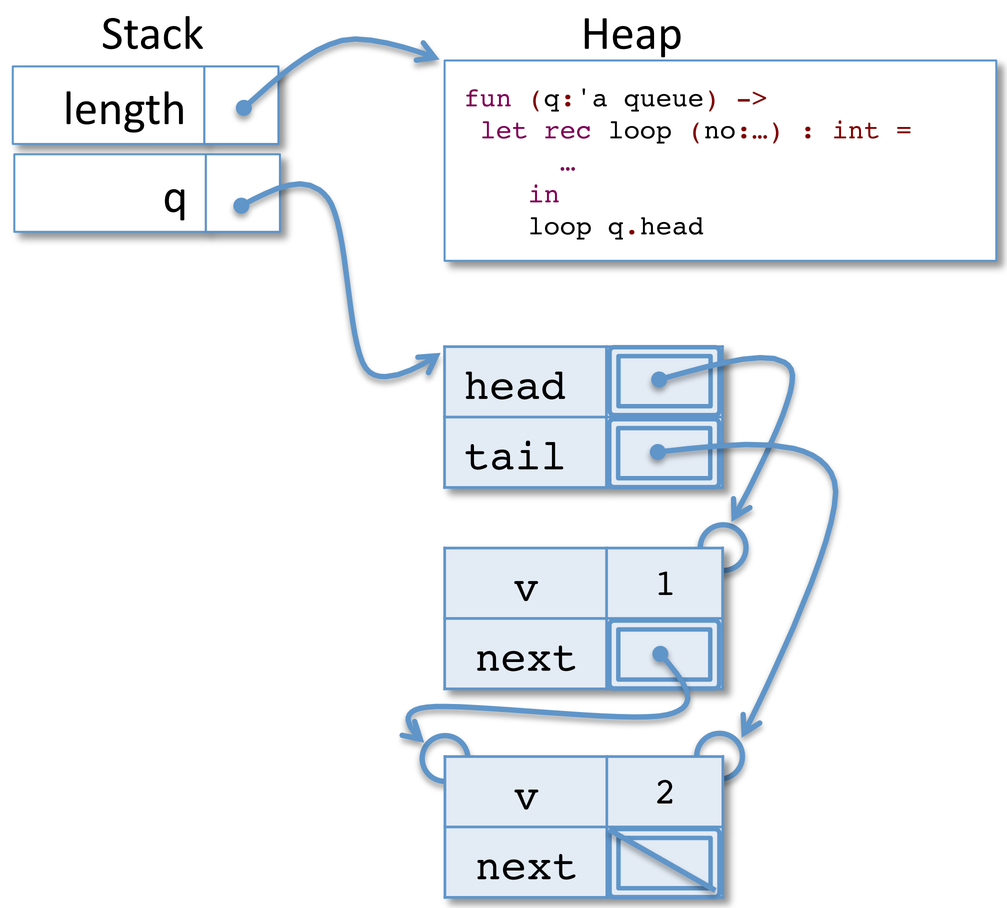
```
(* Calculate the length of the queue recursively *)
let length (q:'a queue) : int =
  let rec loop (no: 'a qnode option) : int =
    begin match no with
      | None -> 0
      | Some n -> 1 + (loop n.next)
    end
  in
  loop q.head
```

- 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

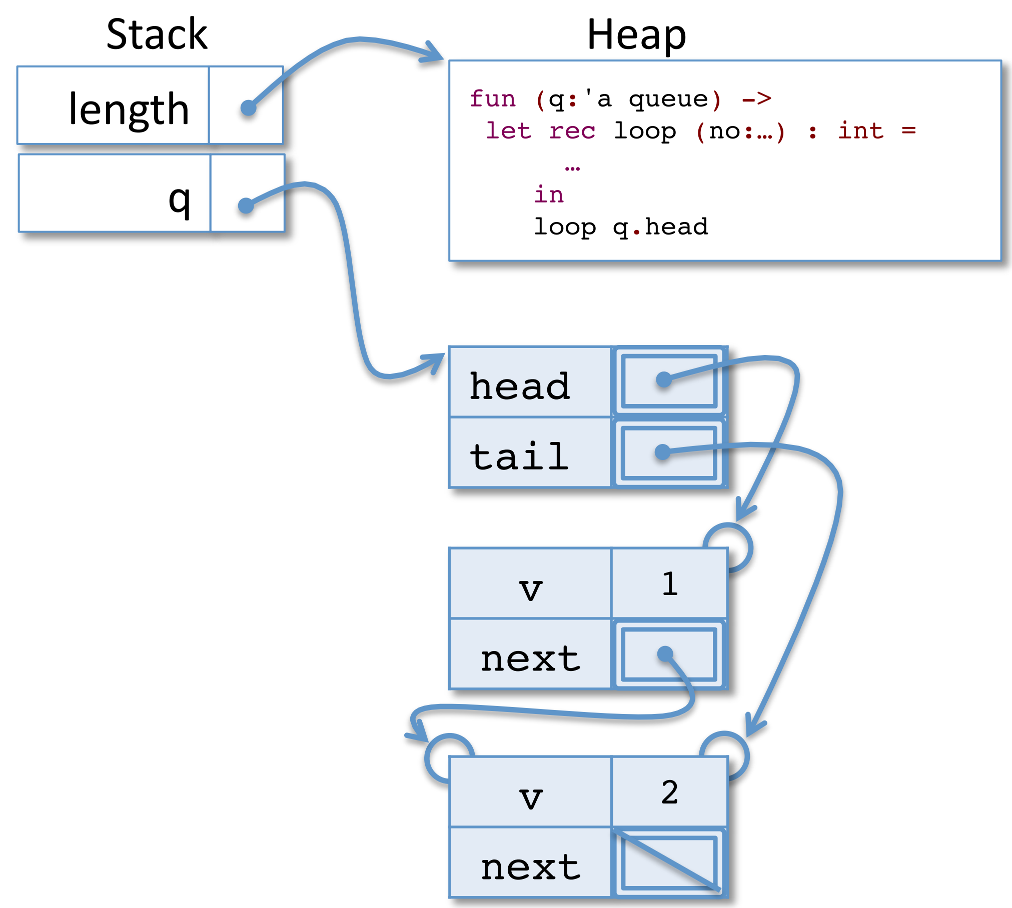
```
length q
```



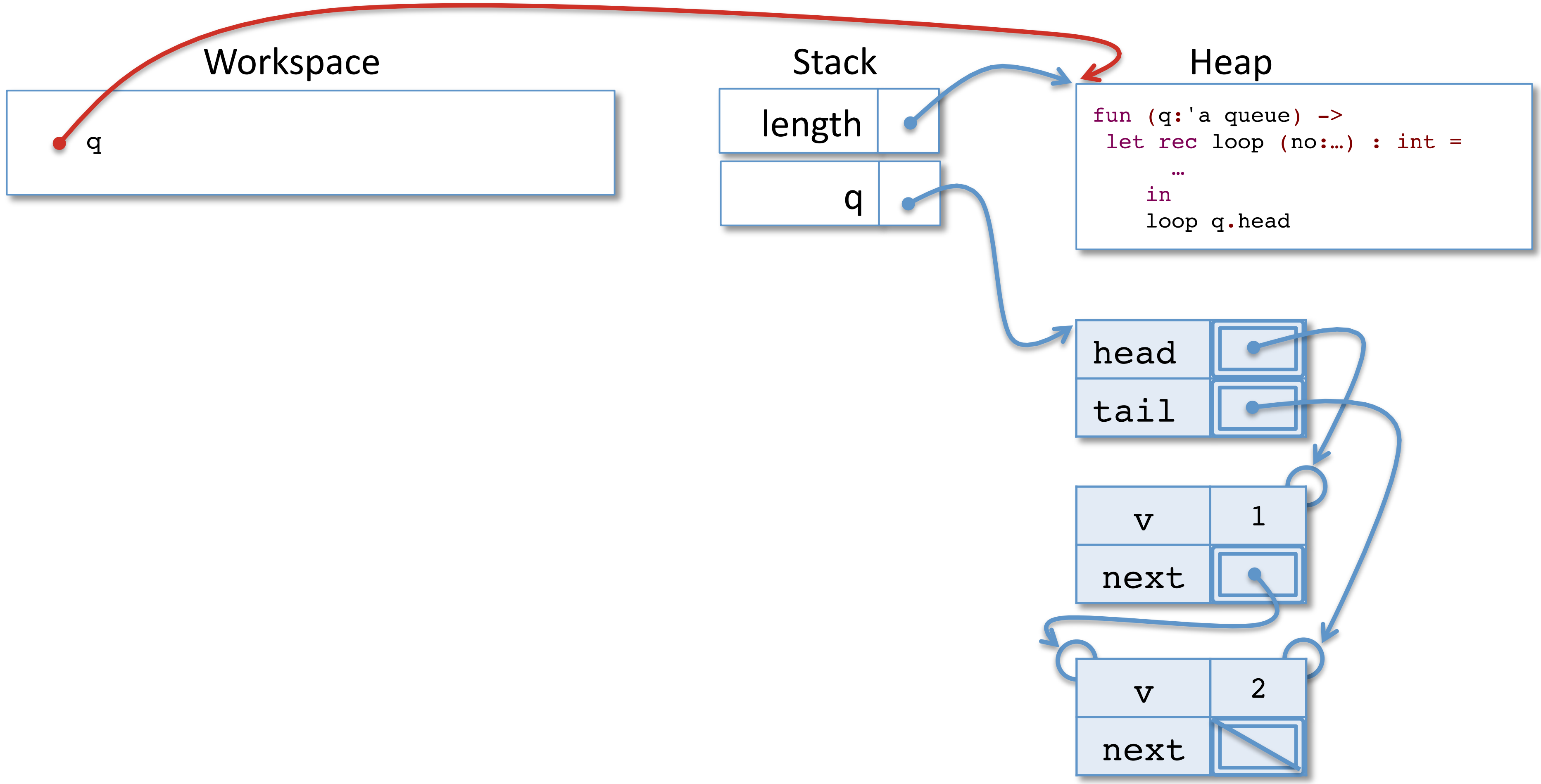
Evaluating length

Workspace

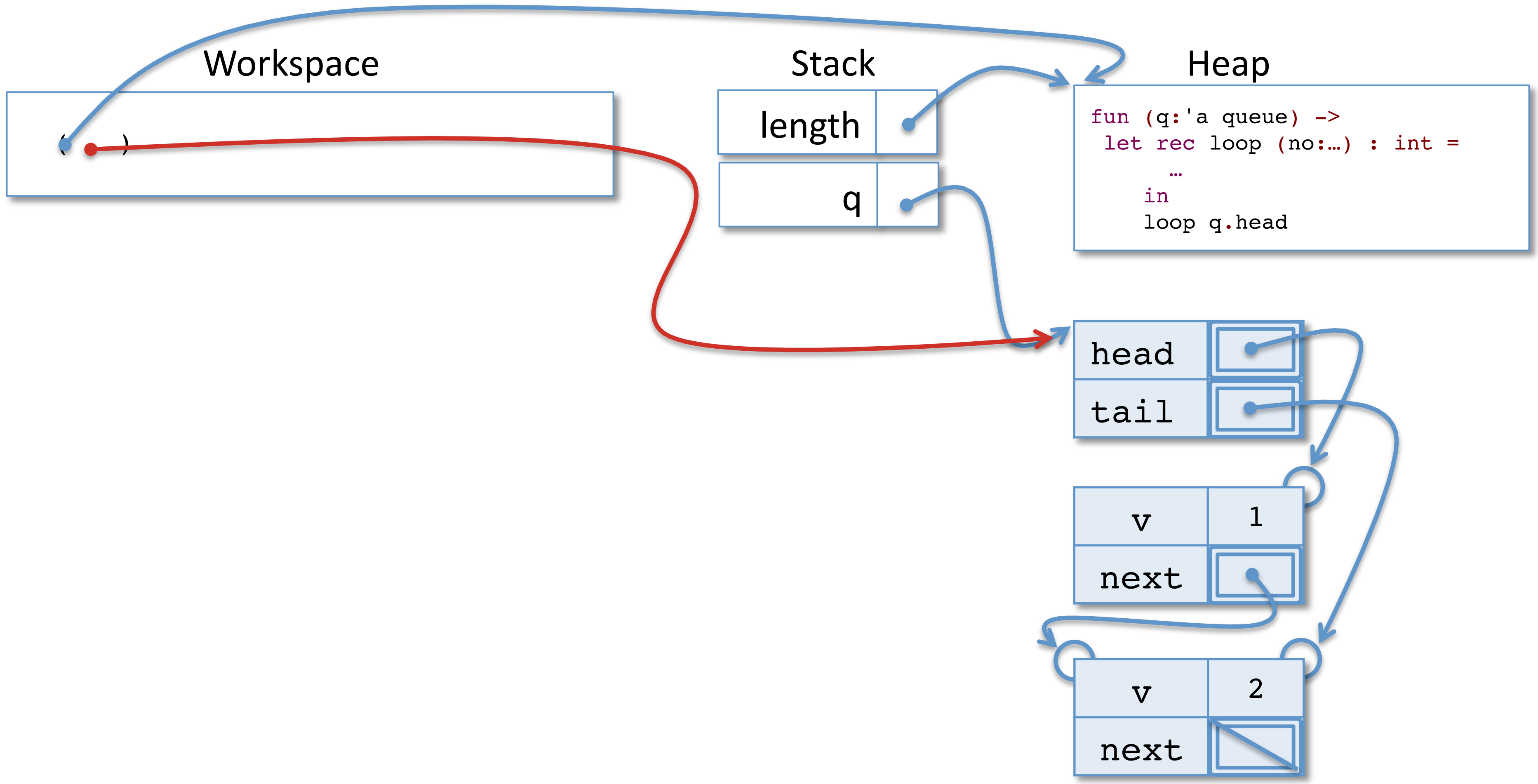
```
length q
```



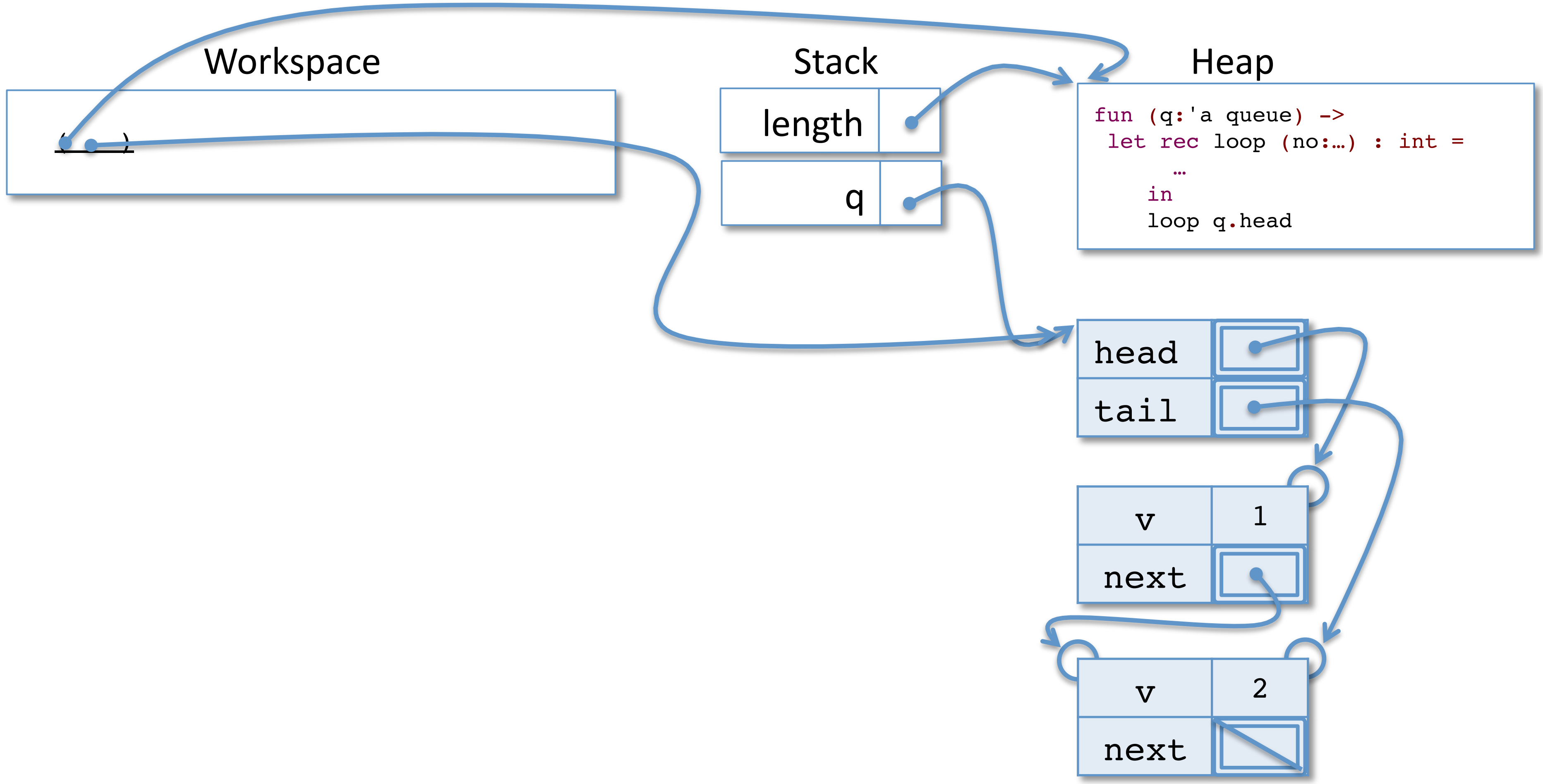
Evaluating length



Evaluating length



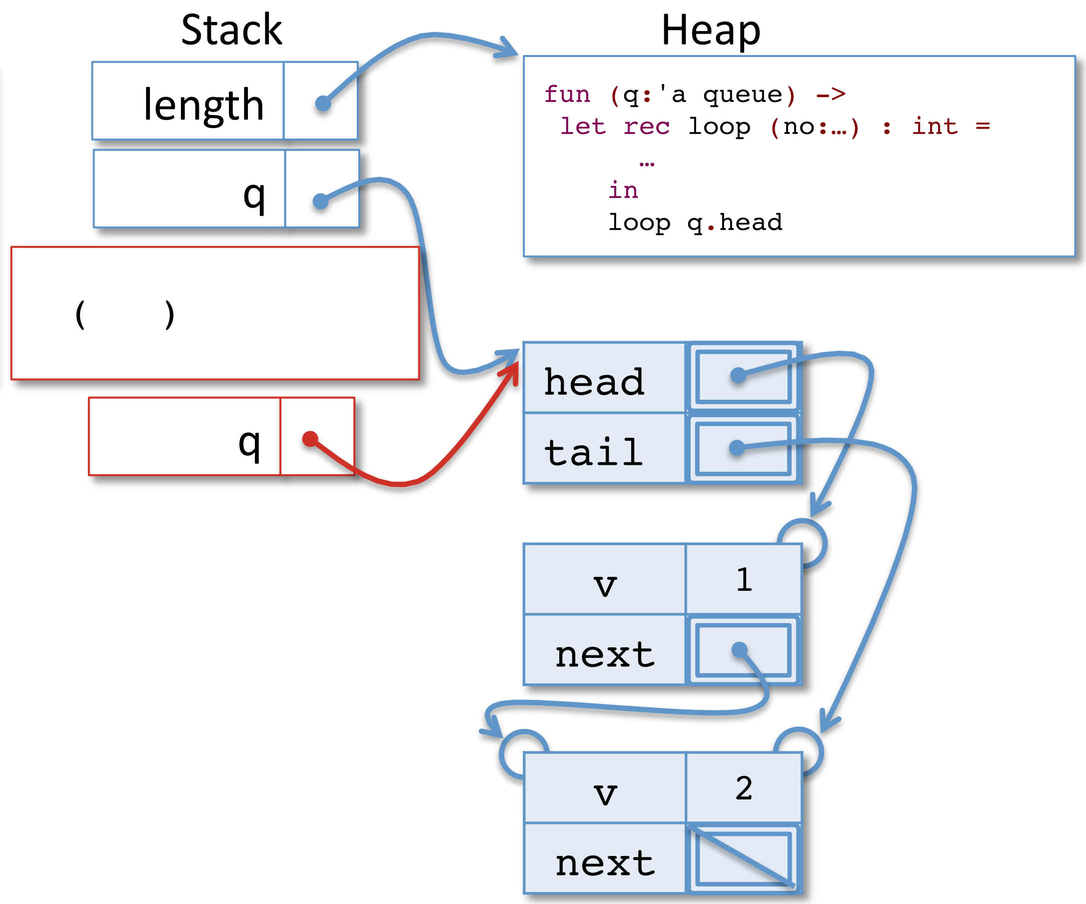
Evaluating length



Evaluating length

Workspace

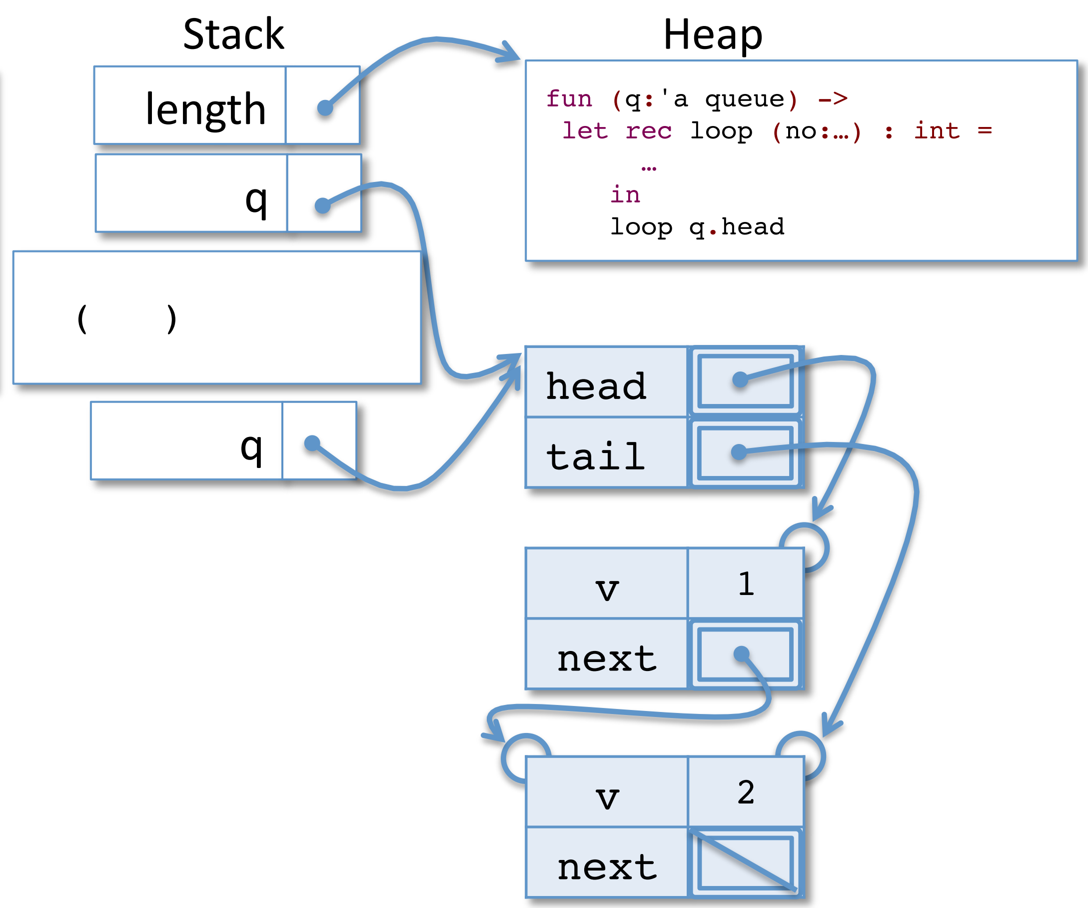
```
let rec loop (no: …) : int =
      begin match no with
        | None -> 0
        | Some n -> 1 + (loop n.next)
      end
    in
    loop q.head
```



Evaluating length

Workspace

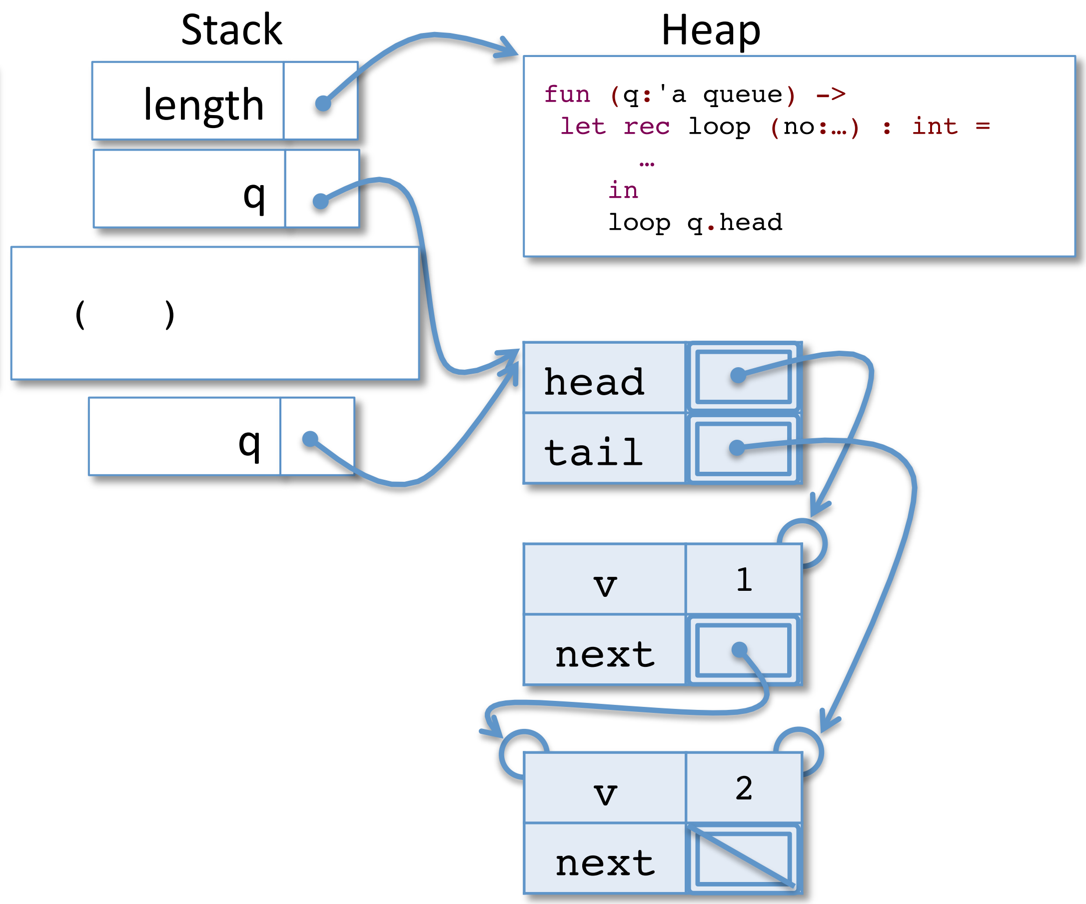
```
let loop = fun (no: …) ->
      begin match no with
        | None -> 0
        | Some n -> 1 + (loop n.next)
      end
    in
    loop q.head
```



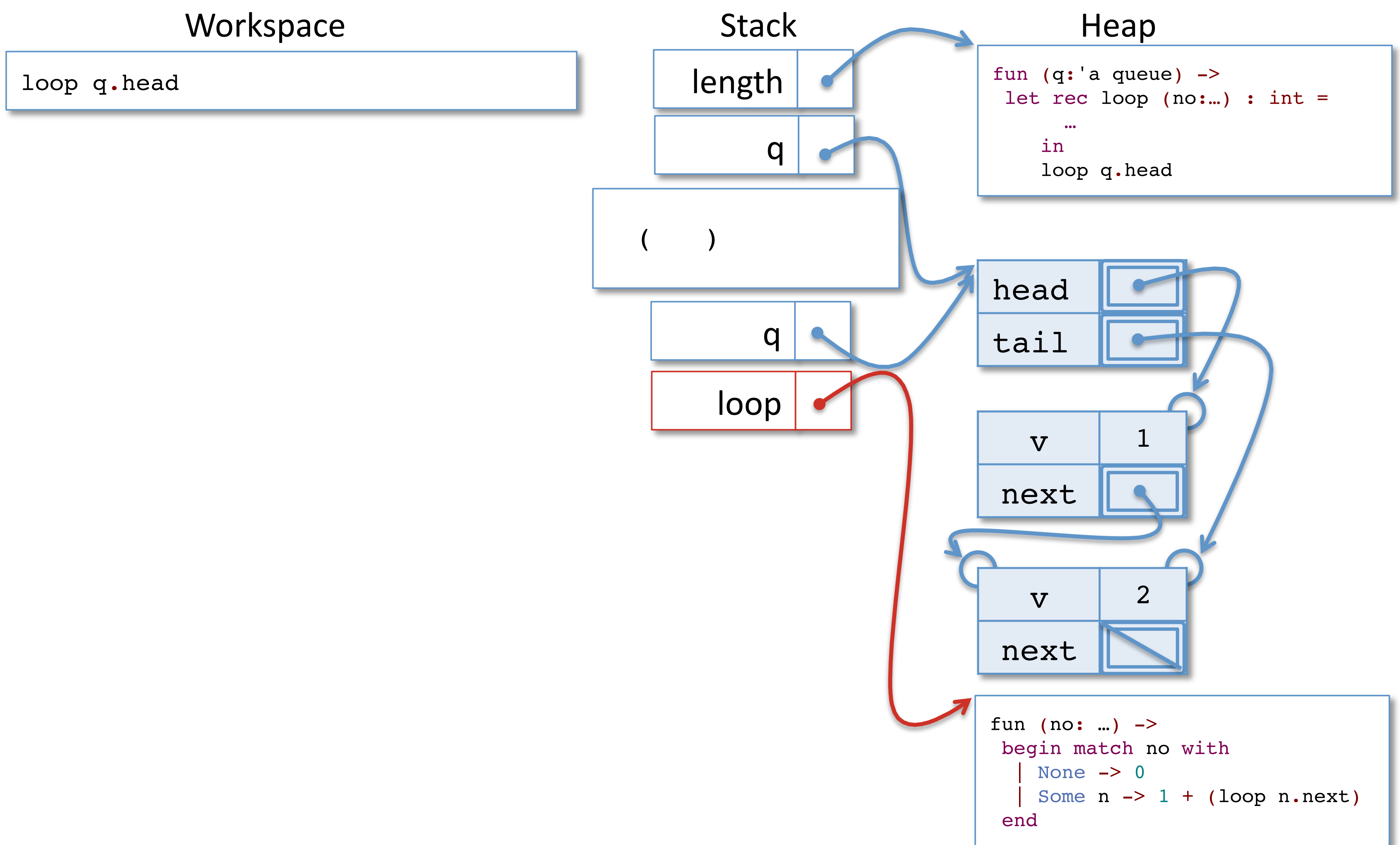
Evaluating length

Workspace

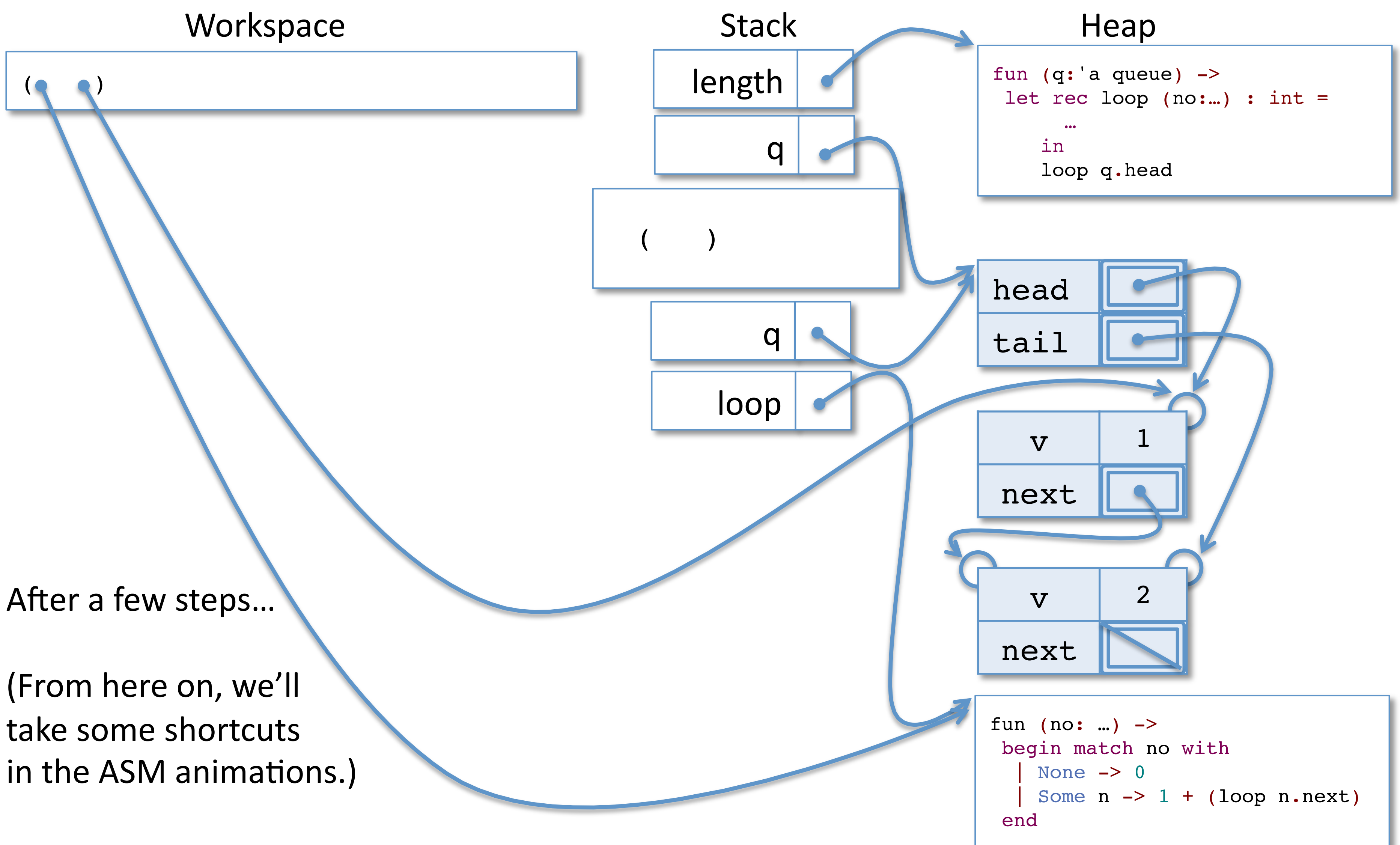
```
let loop = fun (no: …) ->
      begin match no with
        | None -> 0
        | Some n -> 1 + (loop n.next)
      end
    in
    loop q.head
```



Evaluating length



Evaluating length



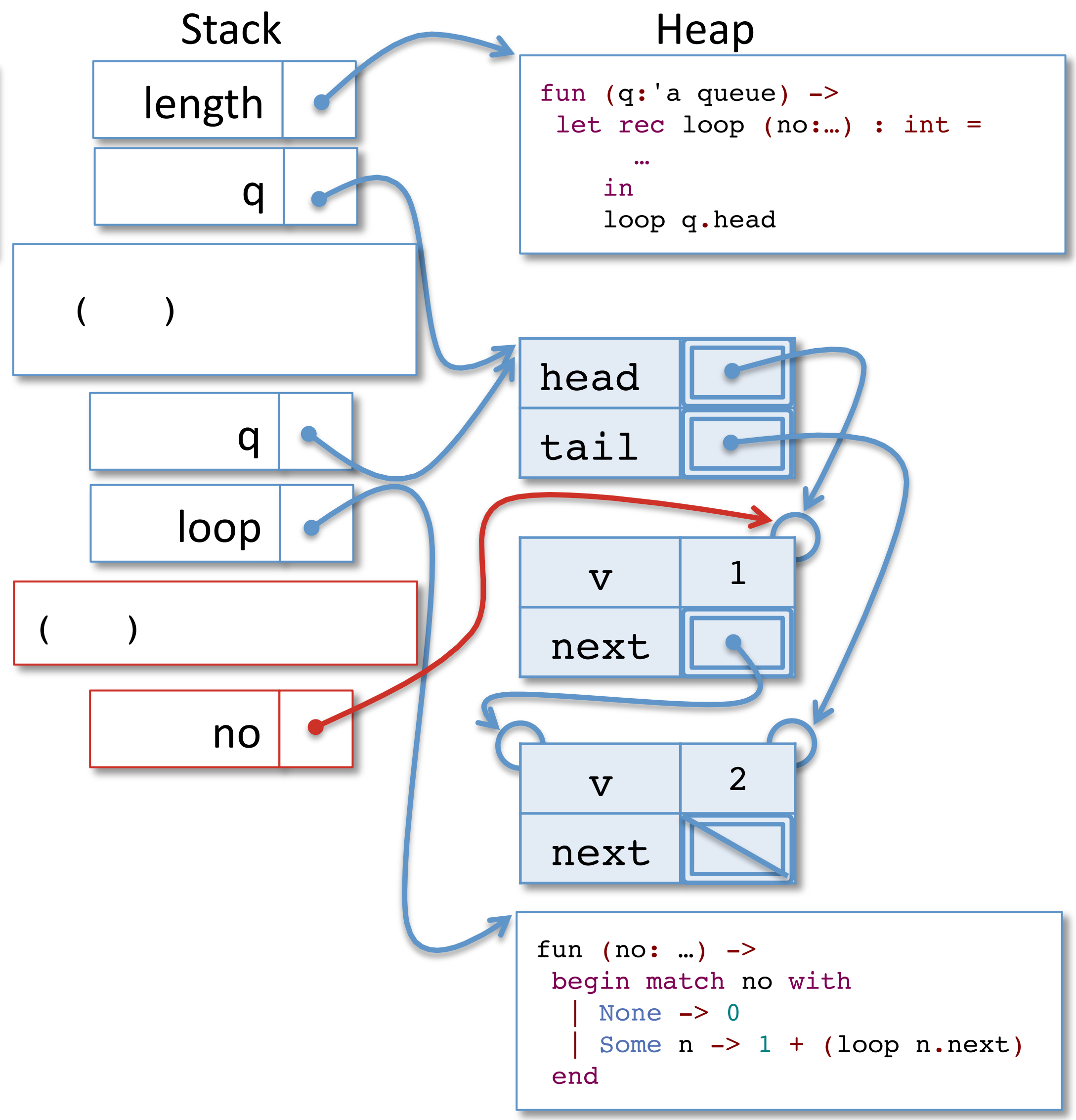
After a few steps…

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

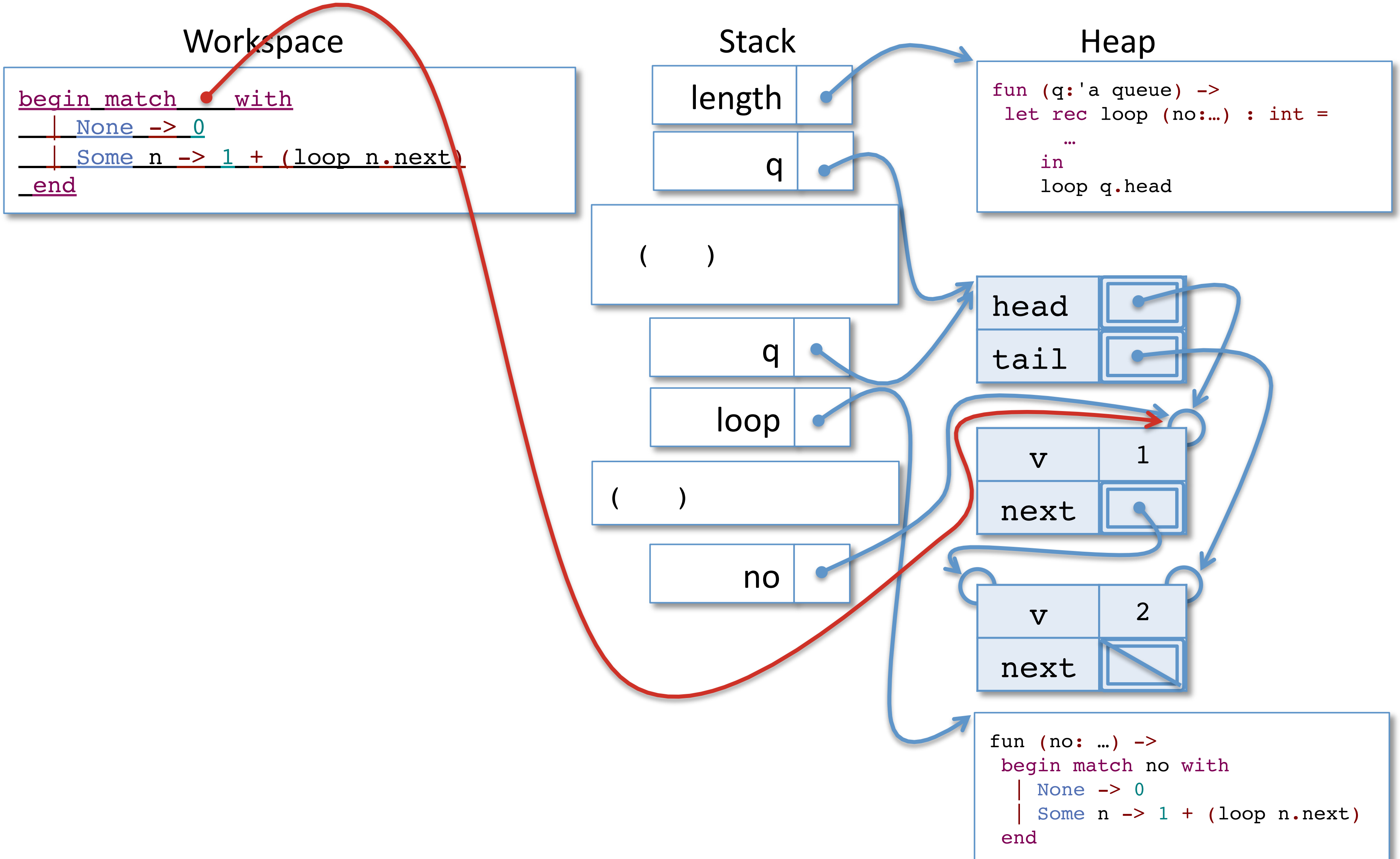
Evaluating length

Workspace

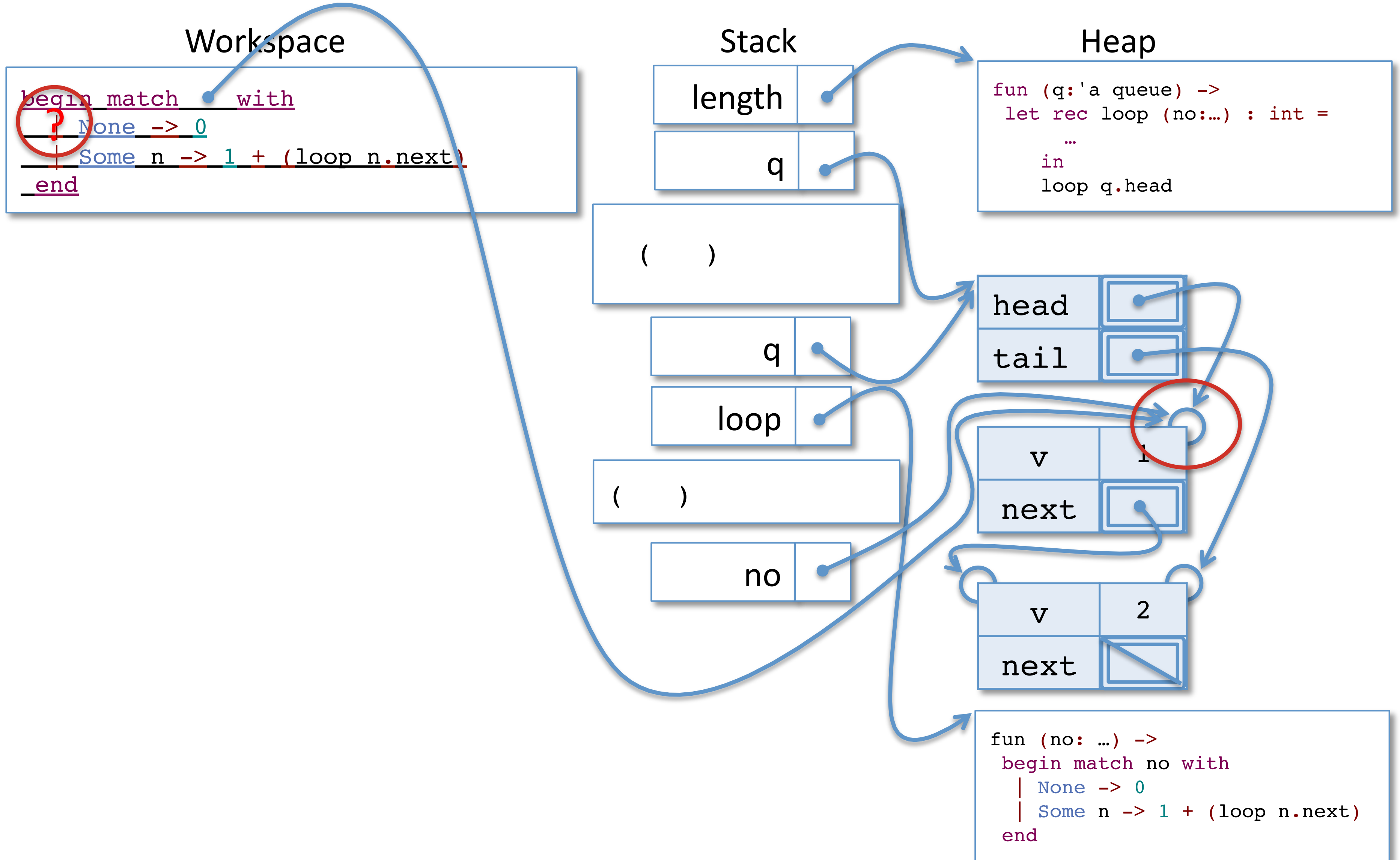
```
begin match no with
  | None -> 0
  | Some n -> 1 + (loop n.next)
end
```



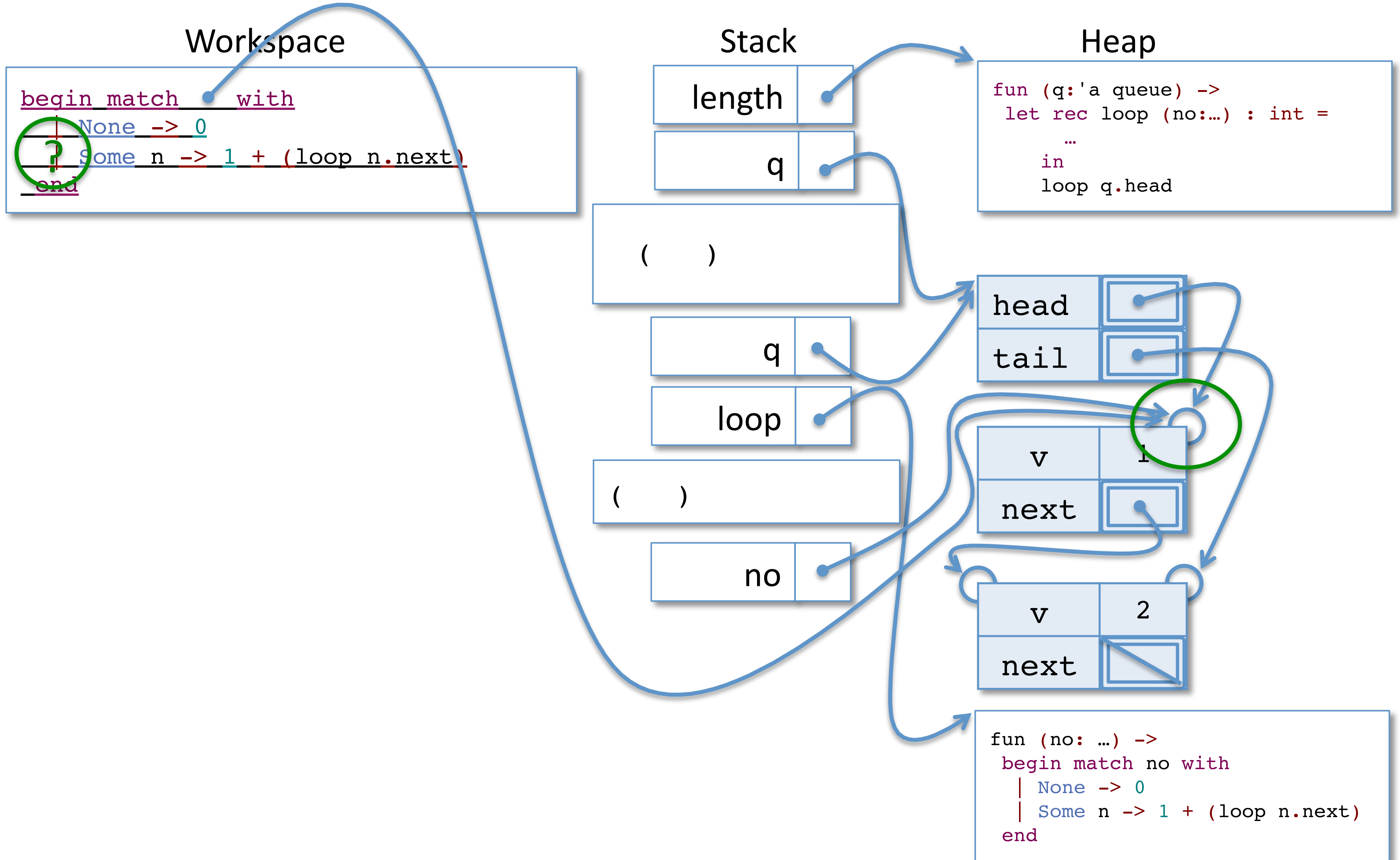
Evaluating length



Evaluating length



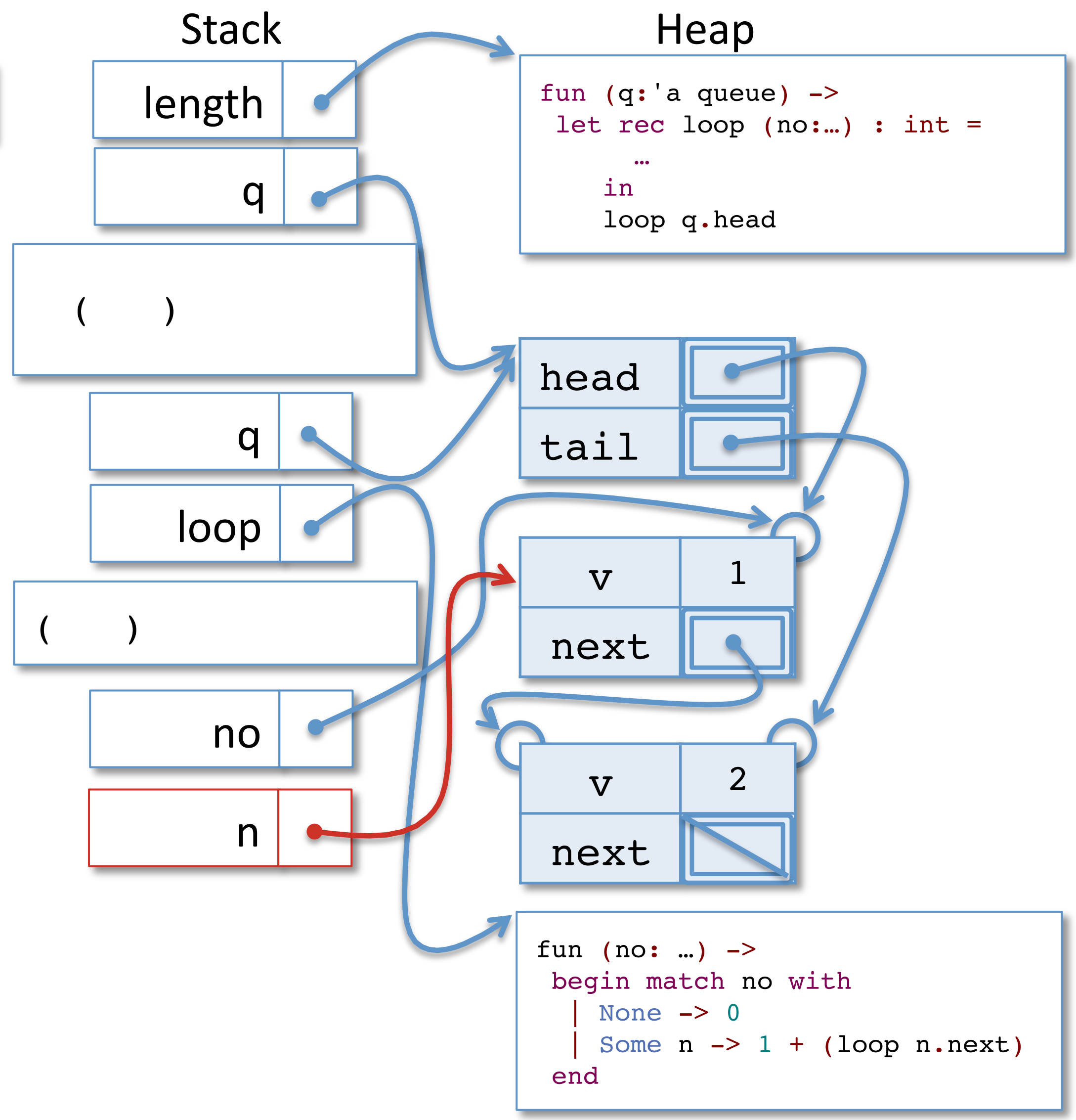
Evaluating length



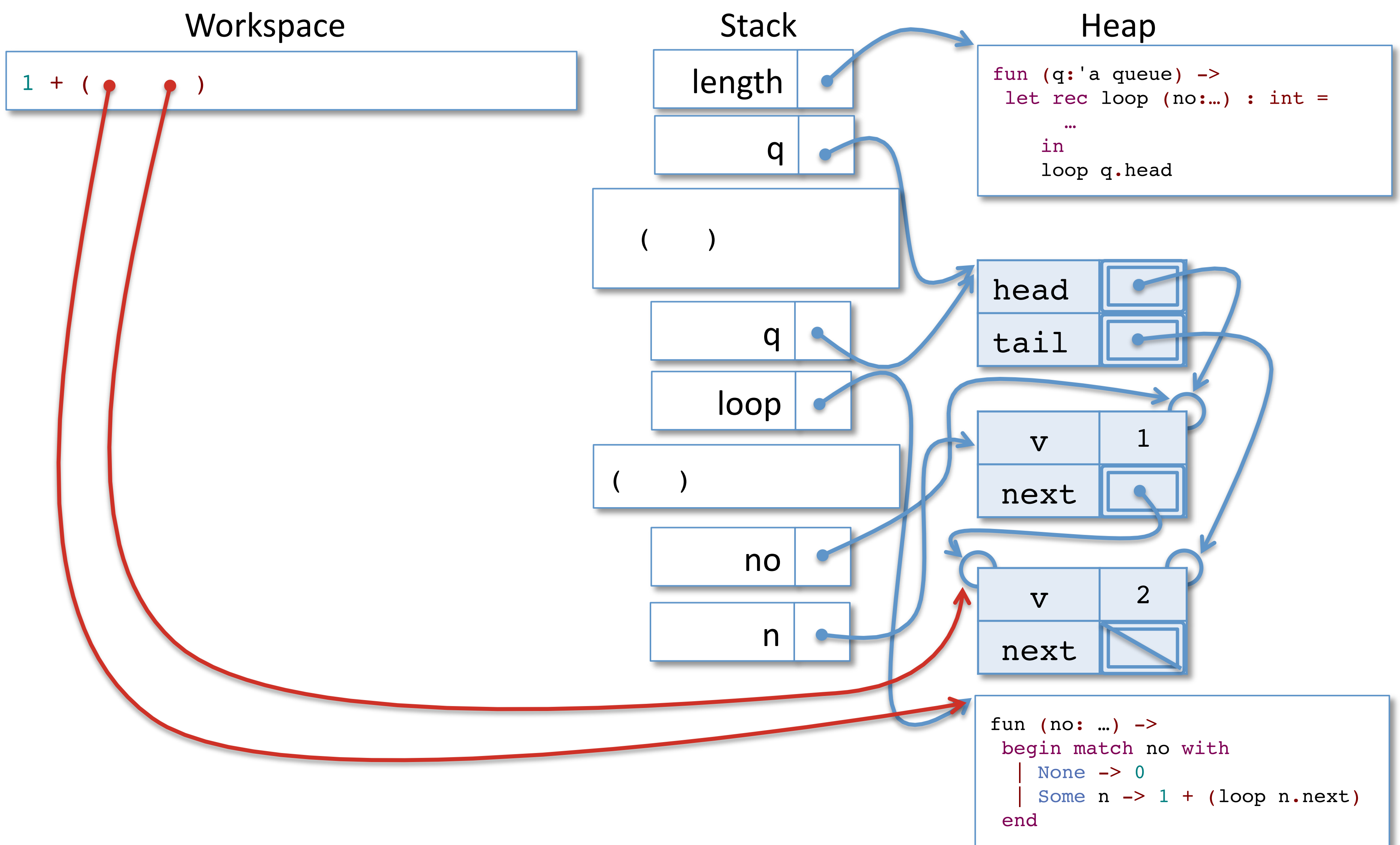
Evaluating length

Workspace

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



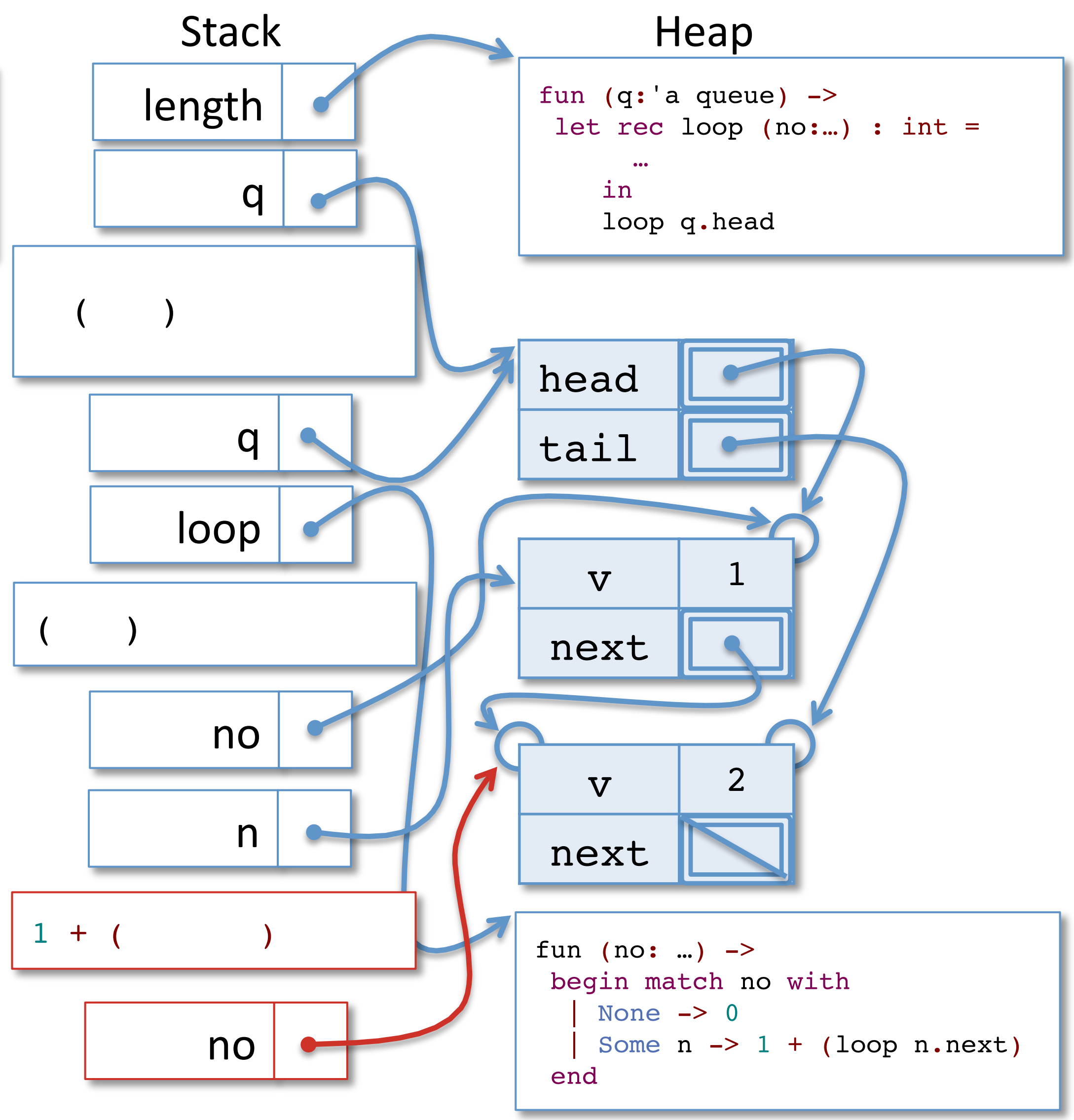
Evaluating length



Evaluating length

Workspace

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

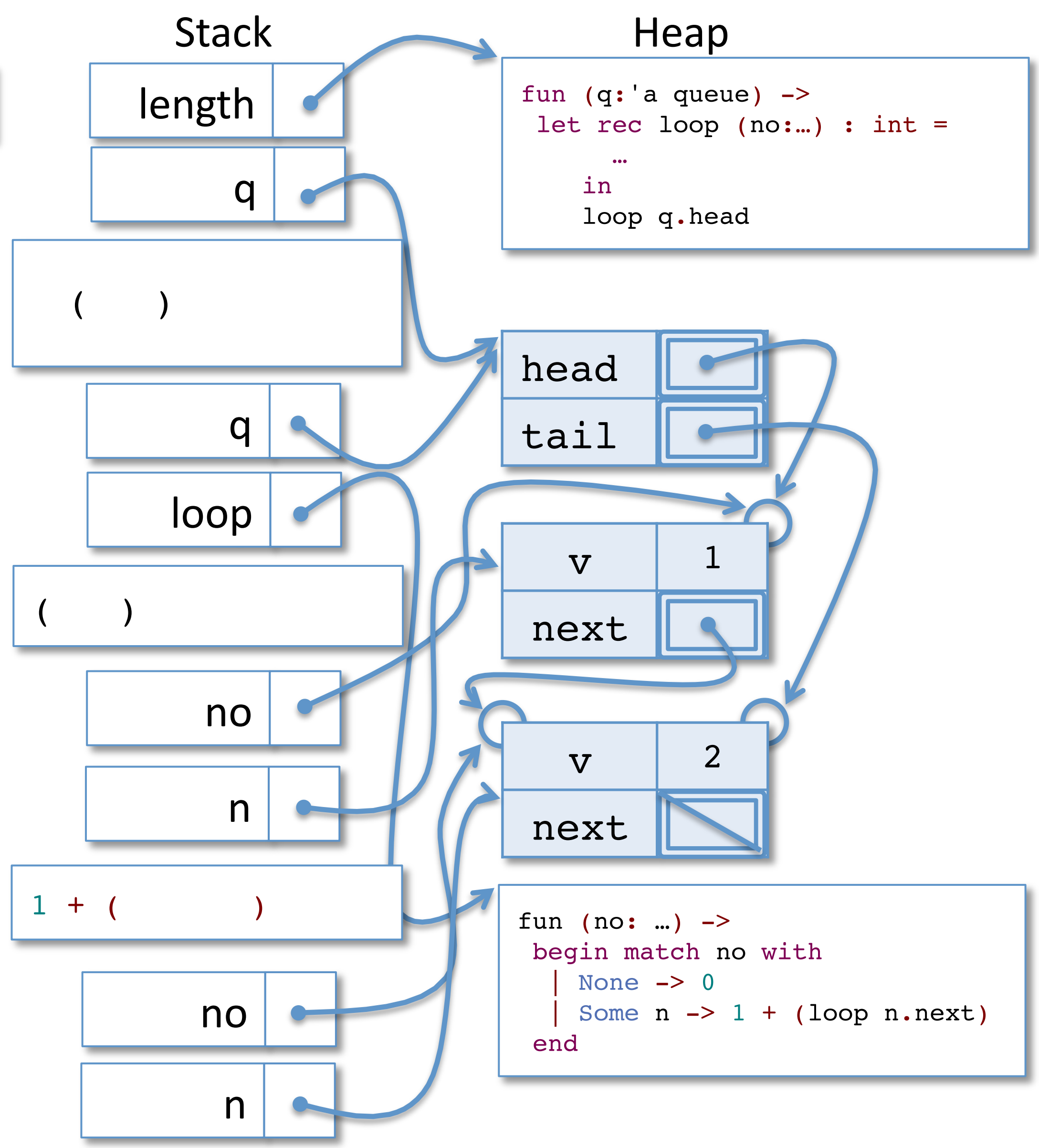


…after a few steps…

Evaluating length

Workspace

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

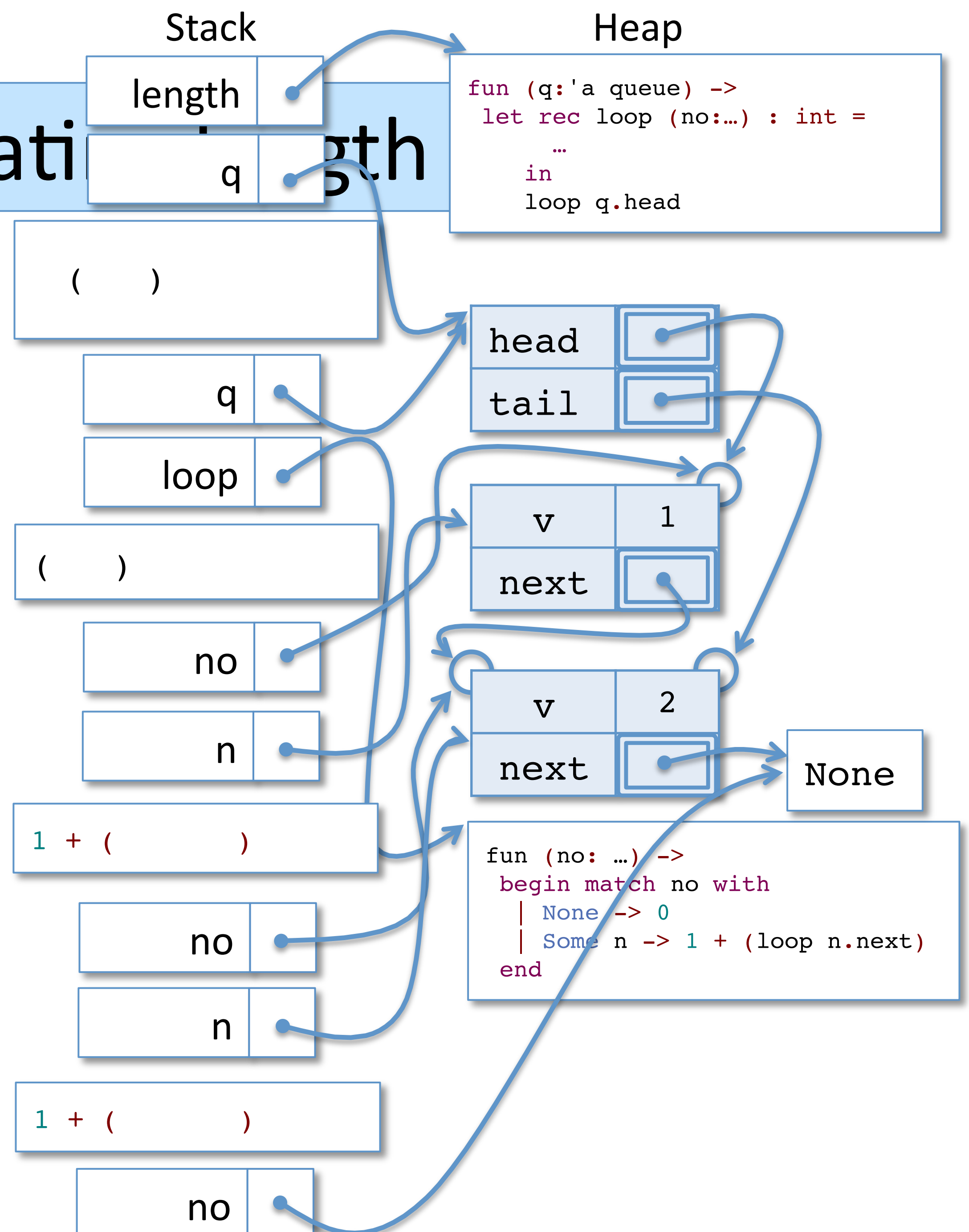


…after a few more steps…

Evaluati… …gth

Workspace

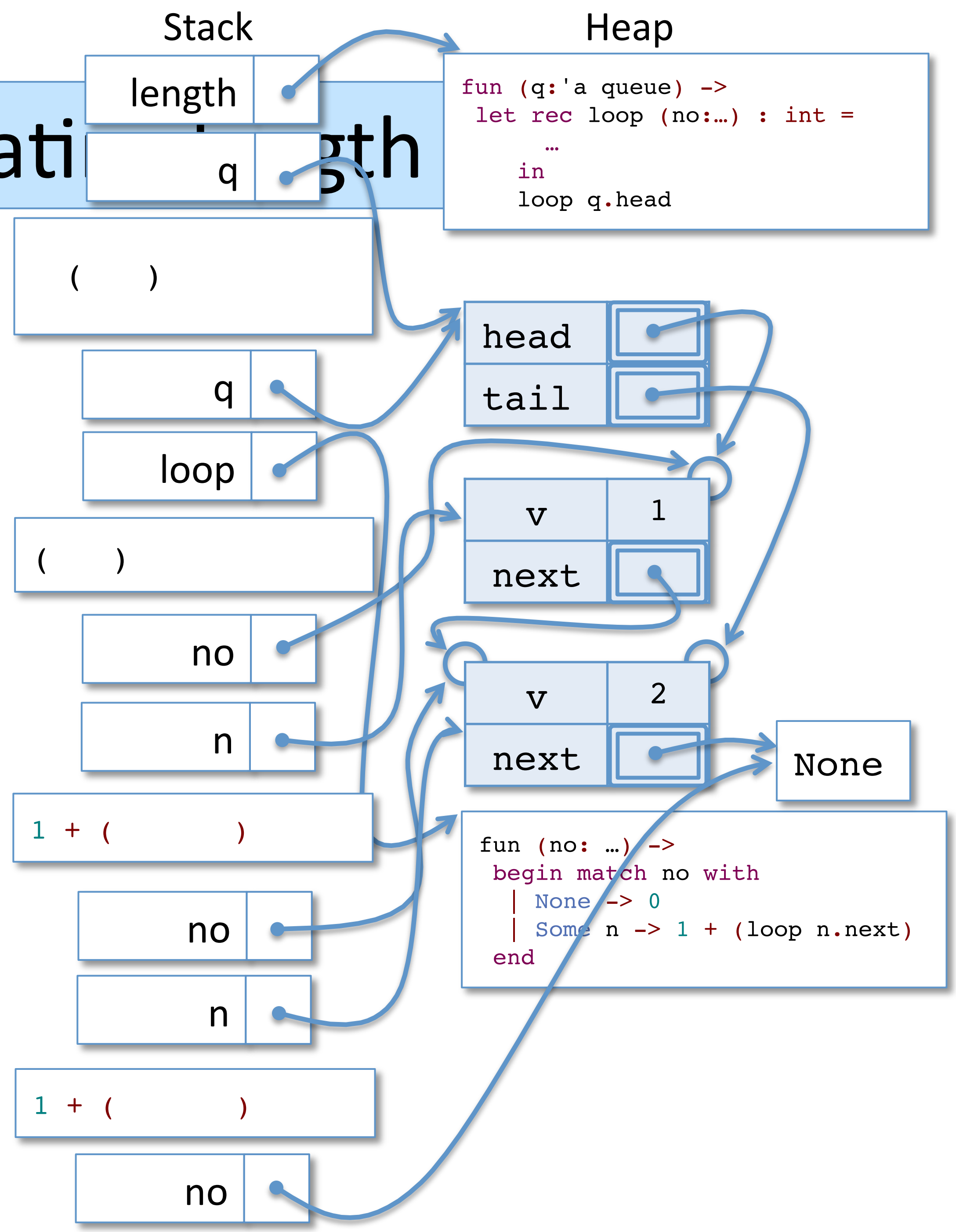
```
begin match no with
  | None -> 0
  | Some n -> 1 + (loop n.next)
 end
```



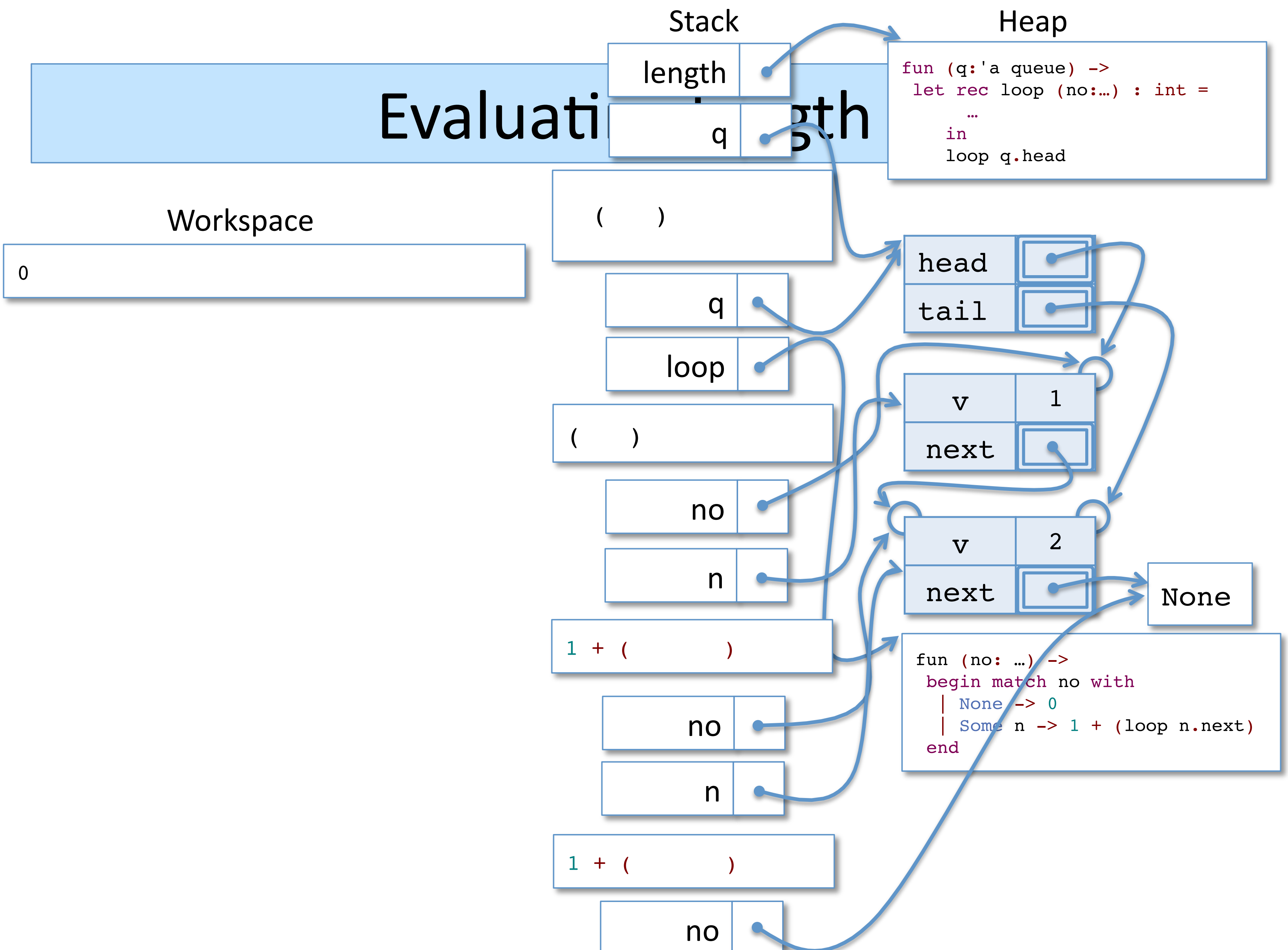
Evaluati gth

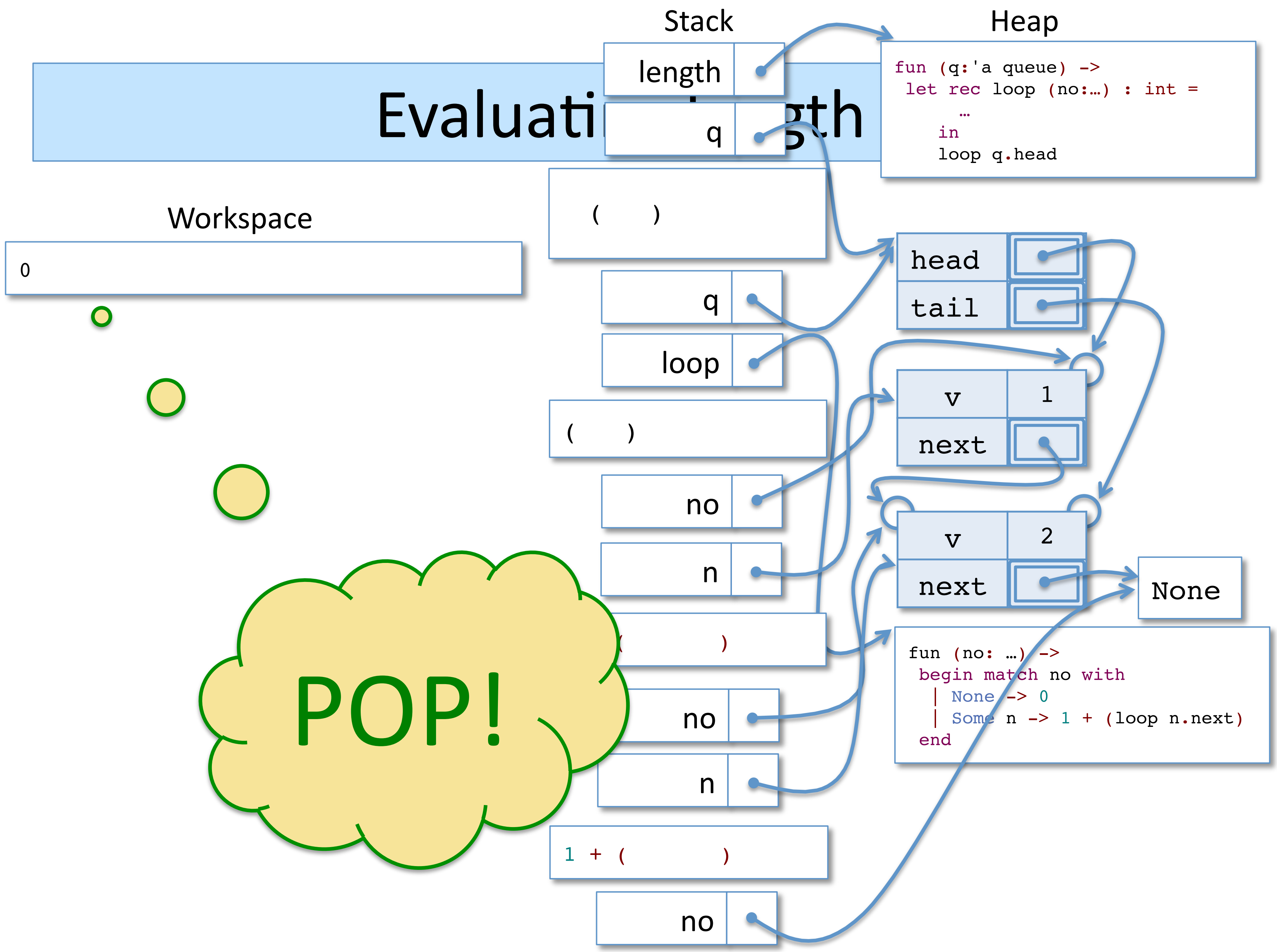
Workspace

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



Evaluati… …gth

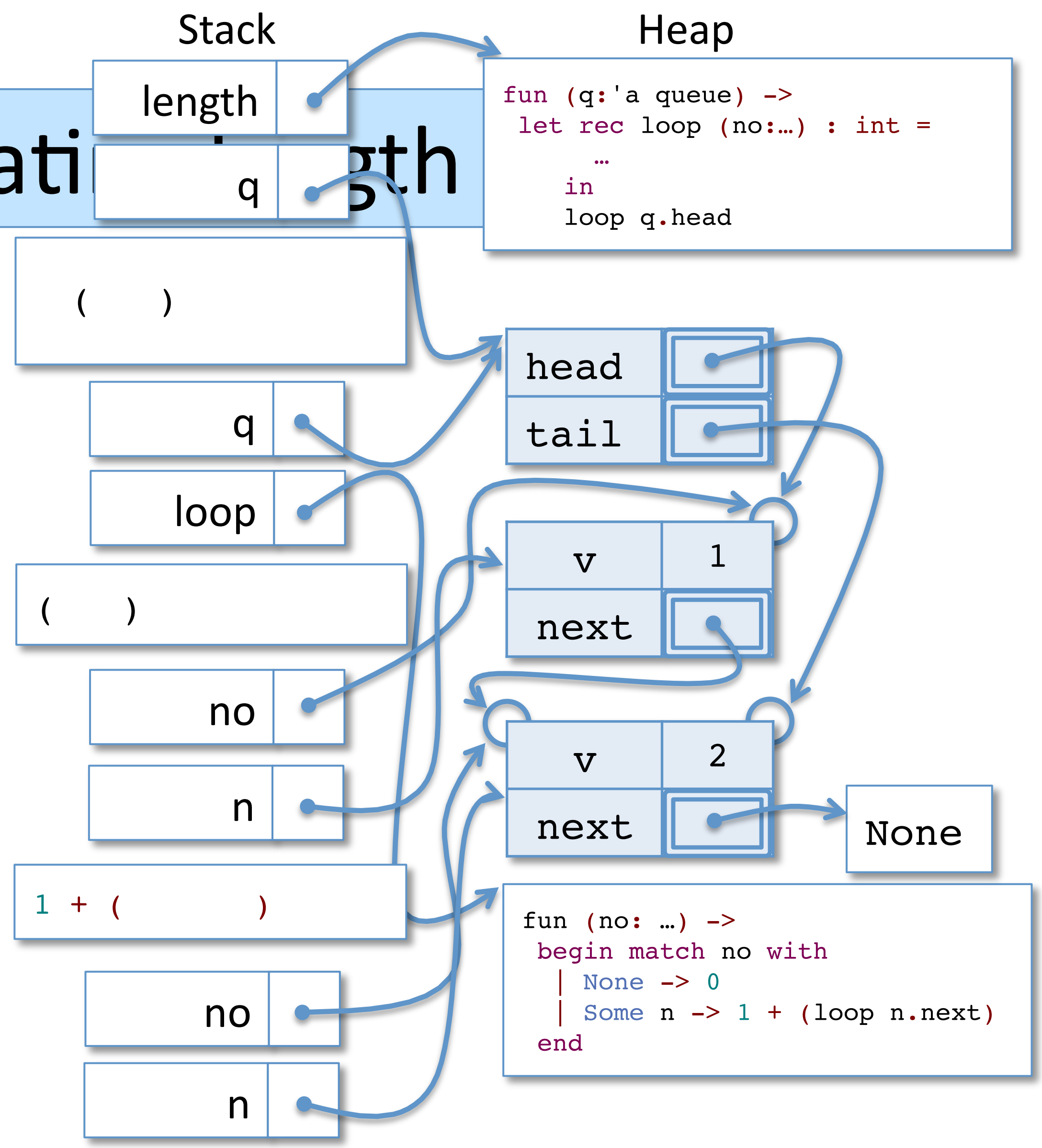


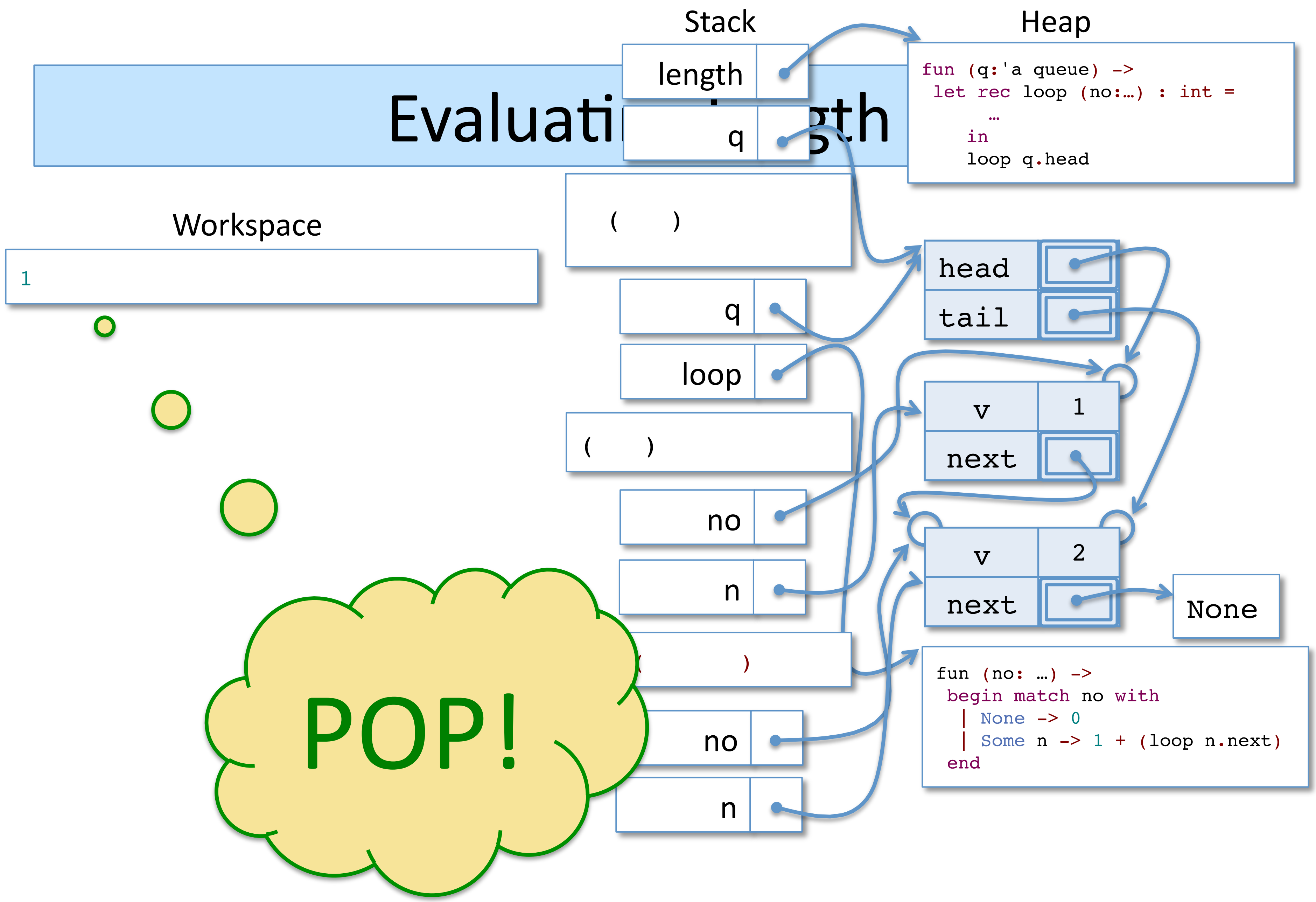

Stack
Heap
Evaluati
gth
length
q
( )
Workspace
0
q
loop
( )
no
n
( )
no
n
1 + ( )
no
POP!
fun (q:'a queue) ->
let rec loop (no:…) : int =
…
in
loop q.head
head
tail
v
1
next
v
2
next
None
fun (no: …) ->
begin match no with
| None -> 0
| Some n -> 1 + (loop n.next)
end

Evaluati… gth

Workspace

1 + 0



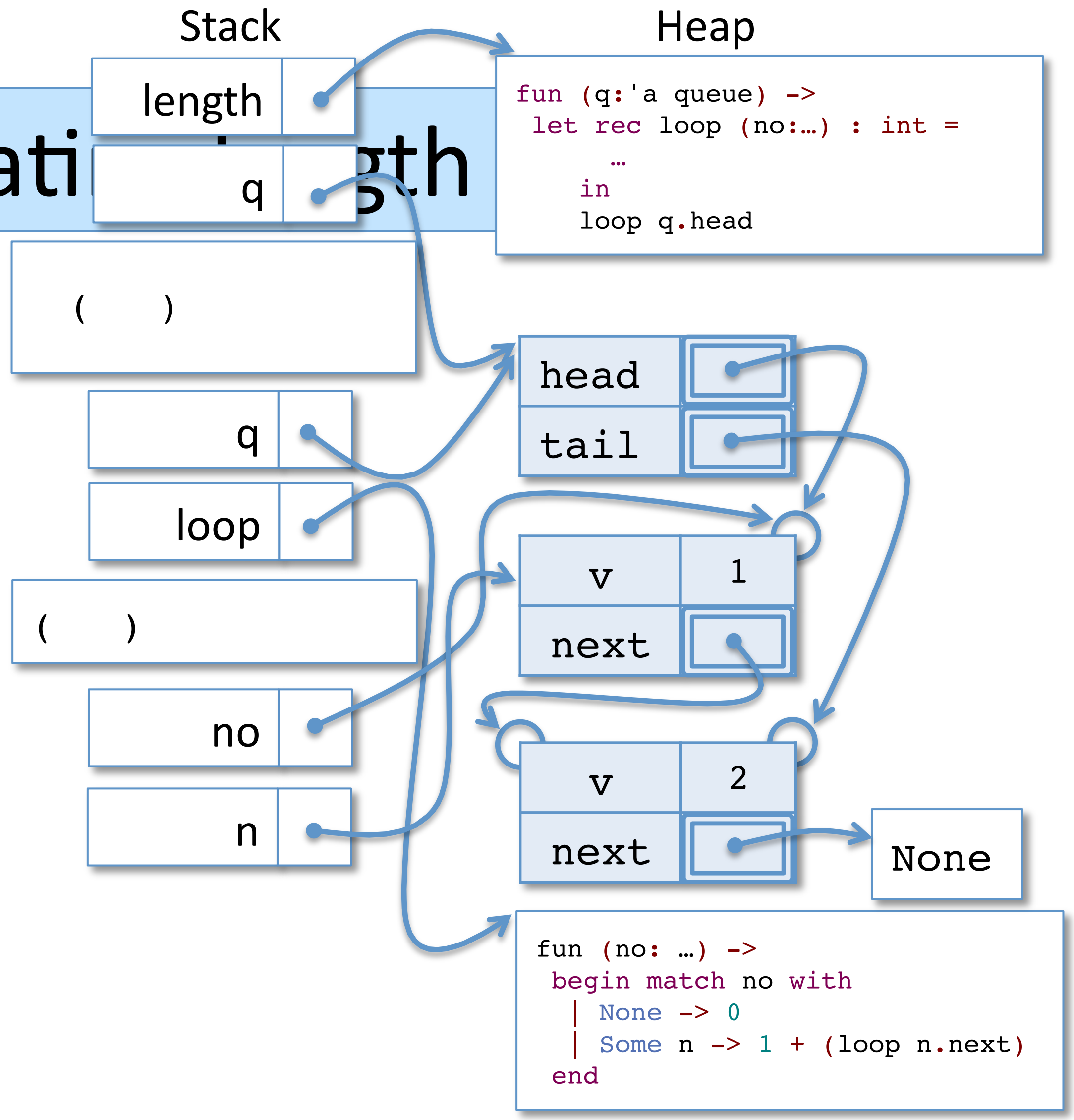


Evaluati
gth
Stack
Heap
Workspace
1
length
q
( )
q
loop
( )
no
n
( )
no
n
fun (q:'a queue) ->
let rec loop (no:…) : int =
…
in
loop q.head
head
tail
v
1
next
v
2
next
None
fun (no: …) ->
begin match no with
| None -> 0
| Some n -> 1 + (loop n.next)
end
POP!

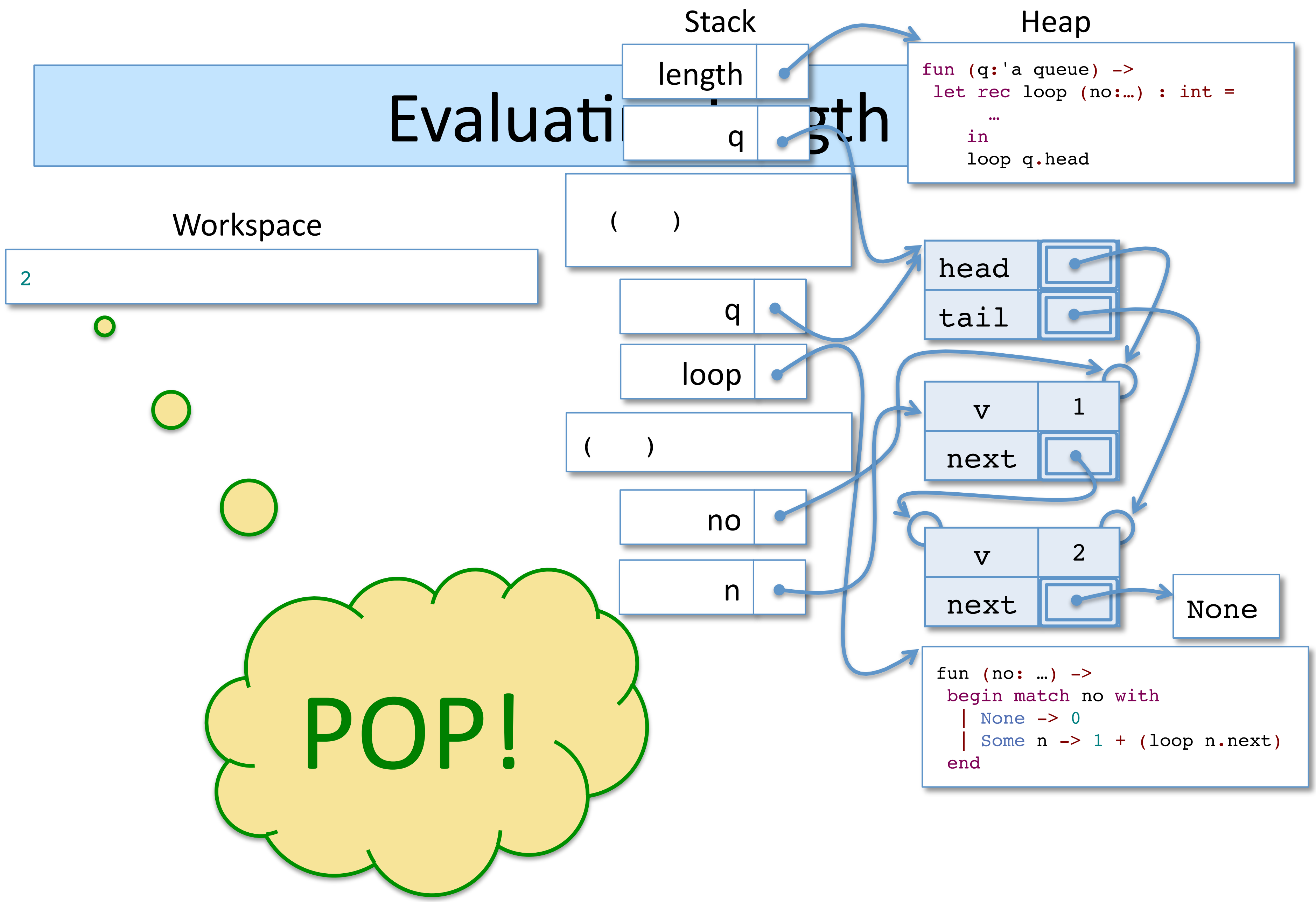
Evaluati gth

Workspace

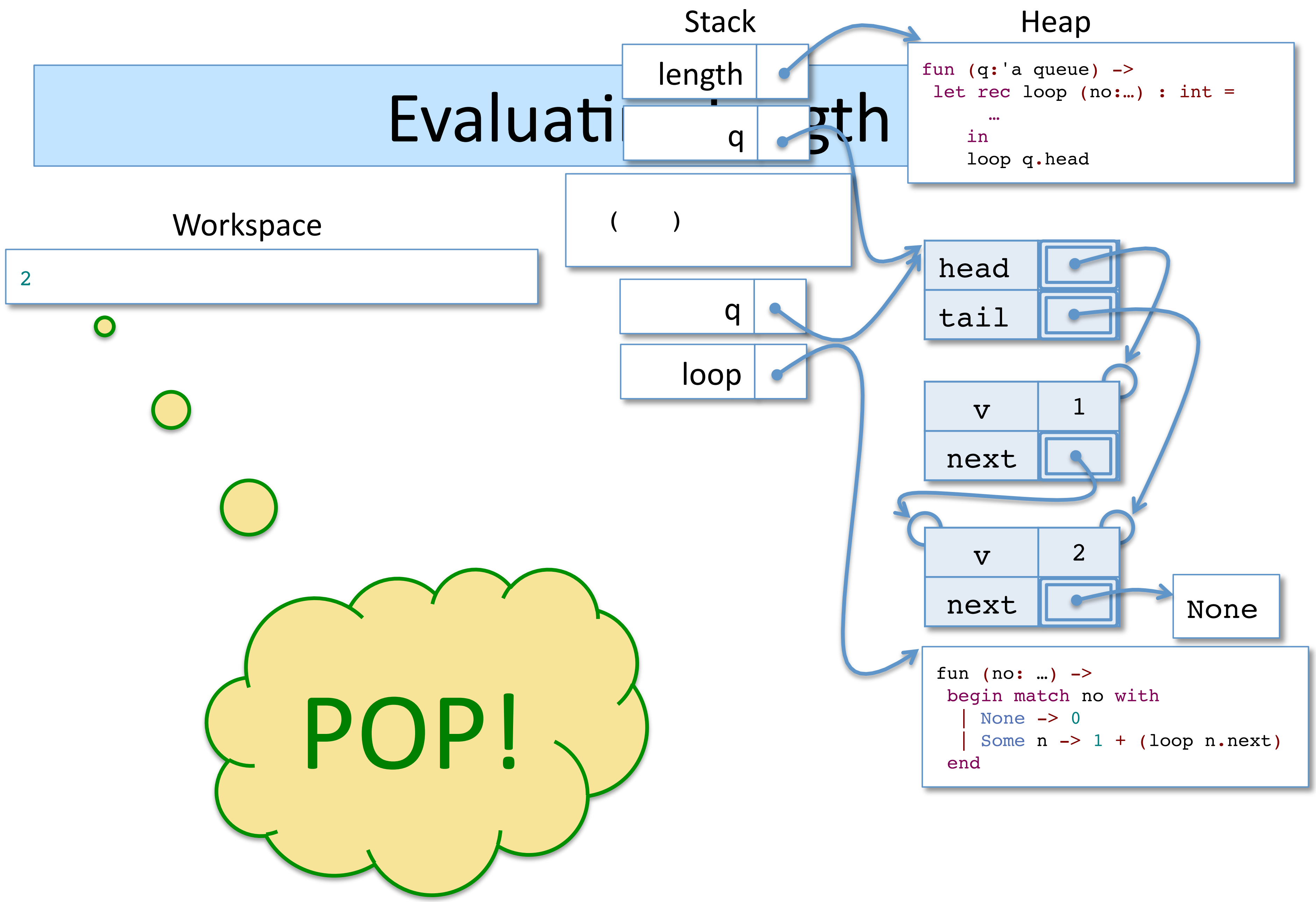
1 + 1



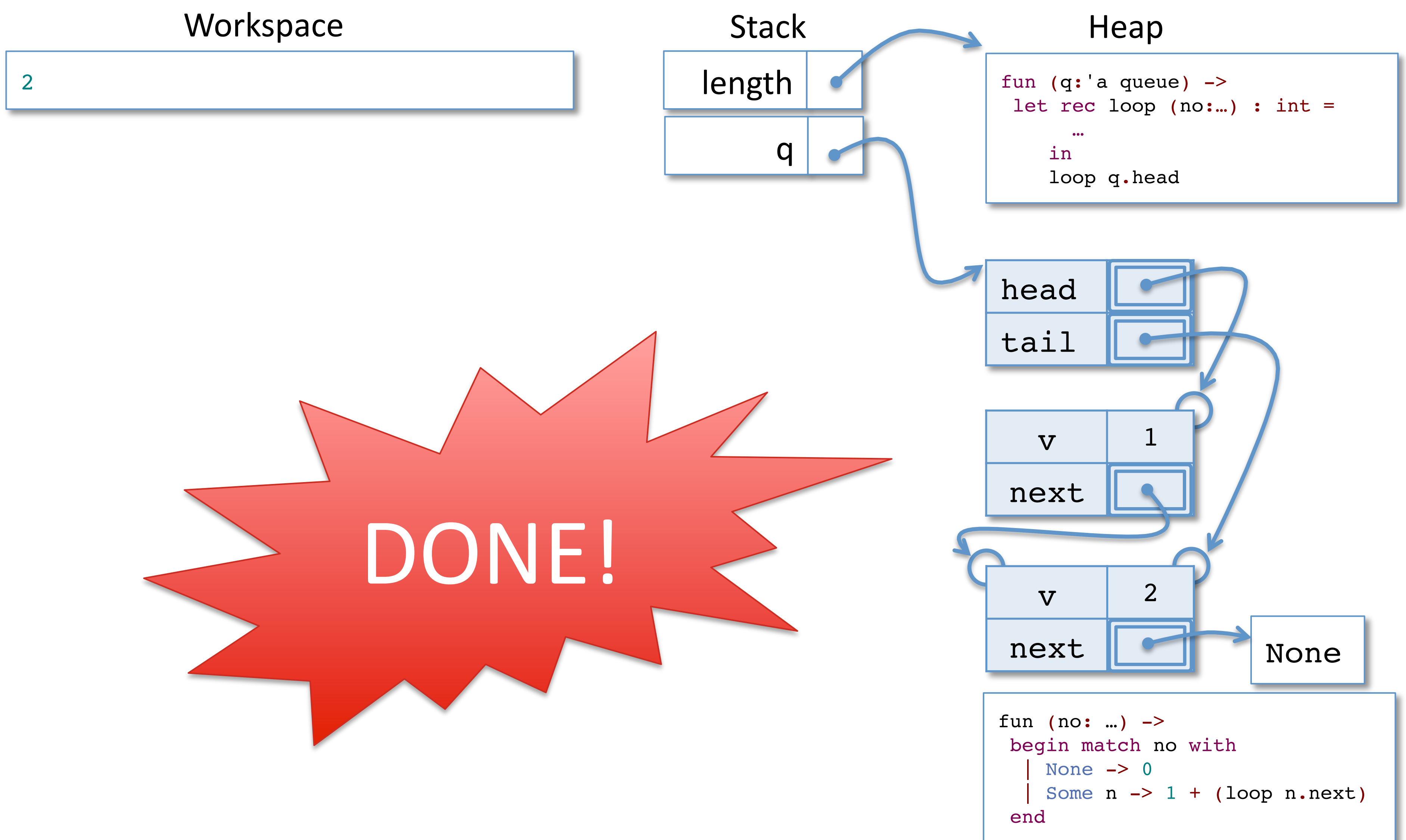
Evaluati… gth



Evaluati gth



Evaluating length



Iteration

loops

`length` (using iteration)

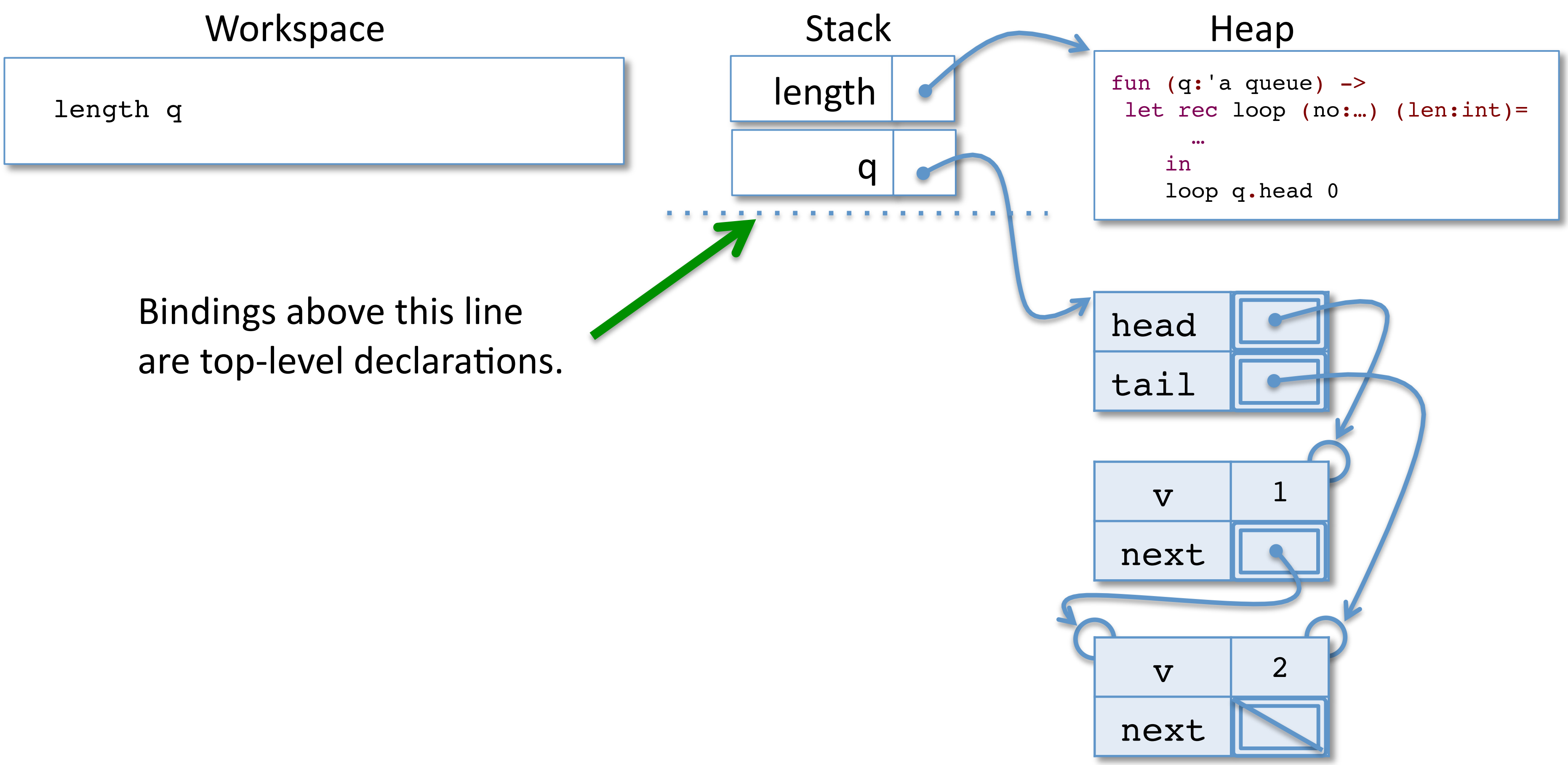
```
(* Calculate the length of the list using iteration *)
let length (q:'a queue) : int =
  let rec loop (no:'a qnode option) (len:int) : int =
      begin match no with
        | None -> len
        | Some n -> loop n.next (1+len)
      end
  in
    loop q.head 0
```

- 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.

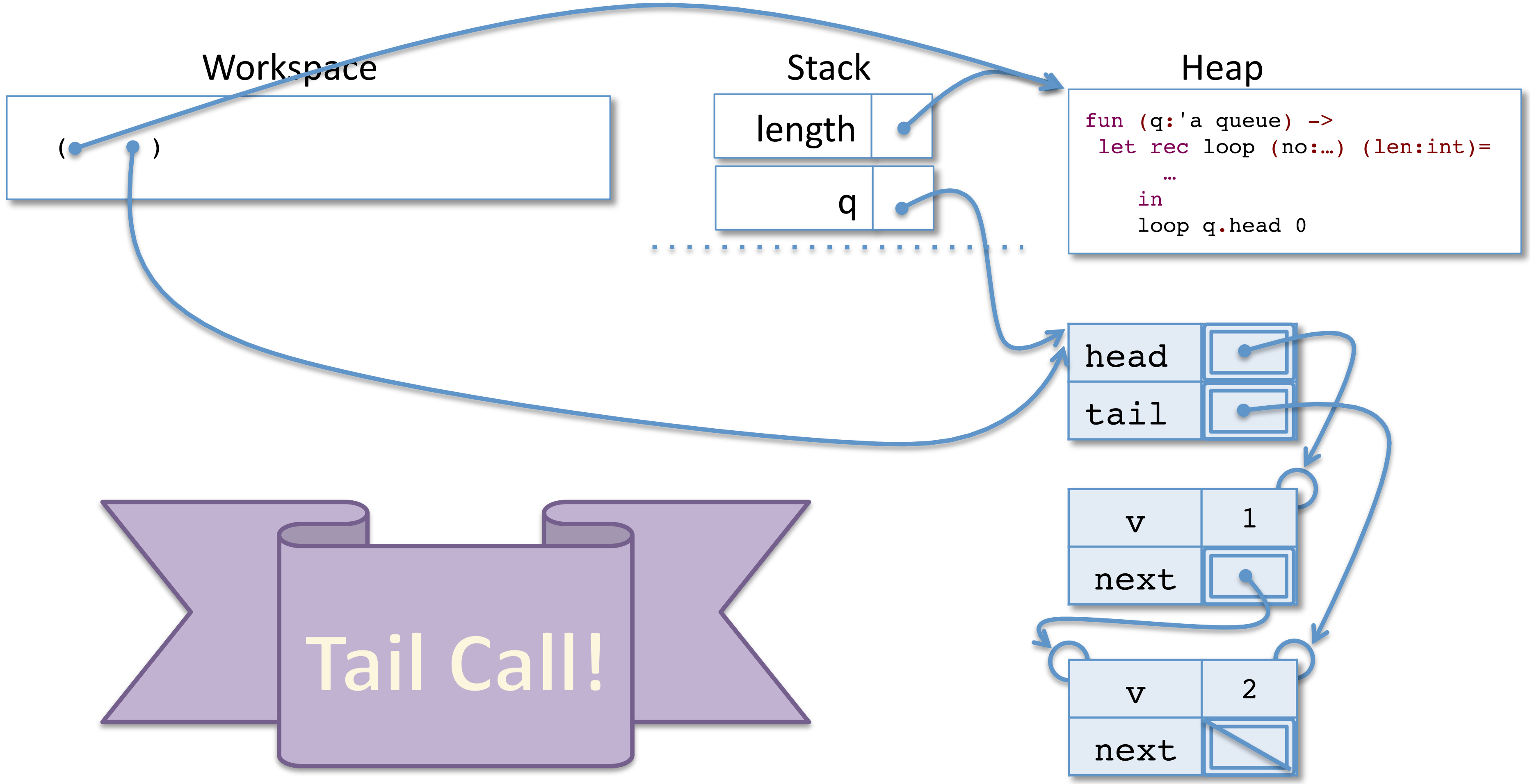
Tail Call Optimization

- 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 ‘while’ or ‘for’ loop in Java or C, using a constant amount of stack space.

Tail Calls and Iterative length



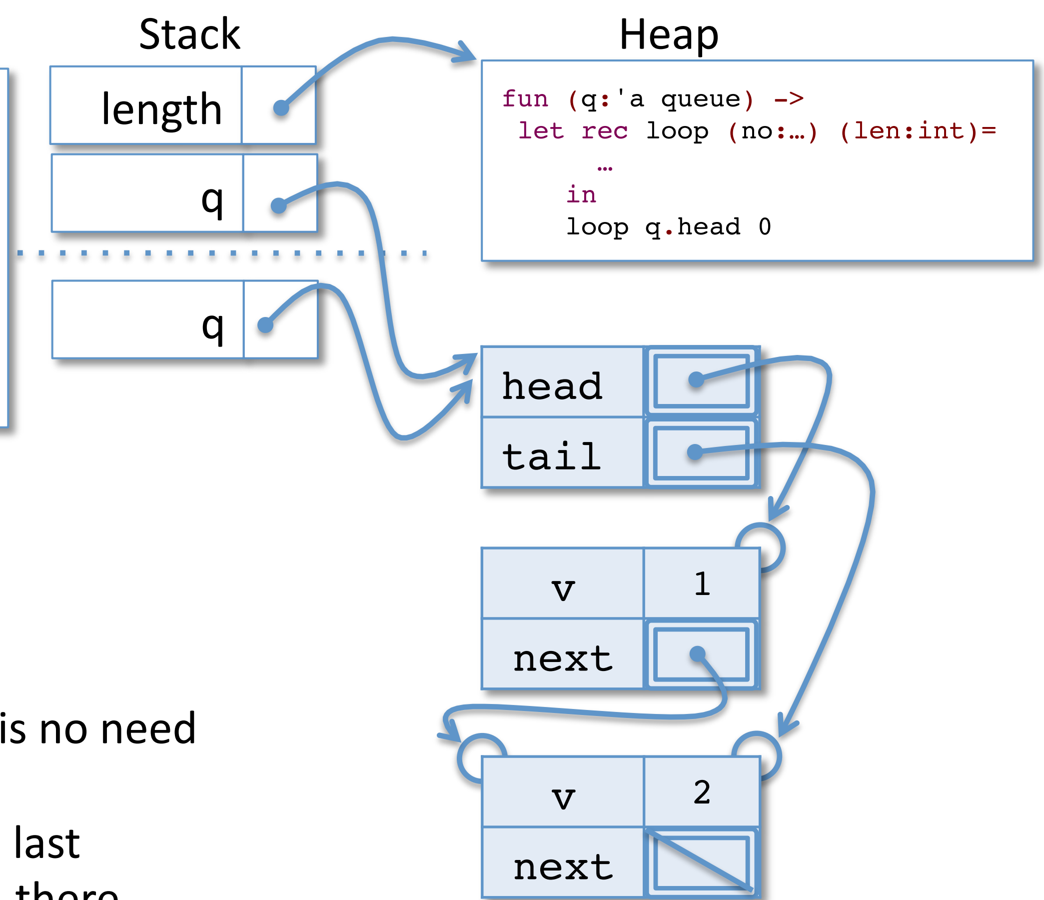
Tail Calls and Iterative length



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
```



Note:

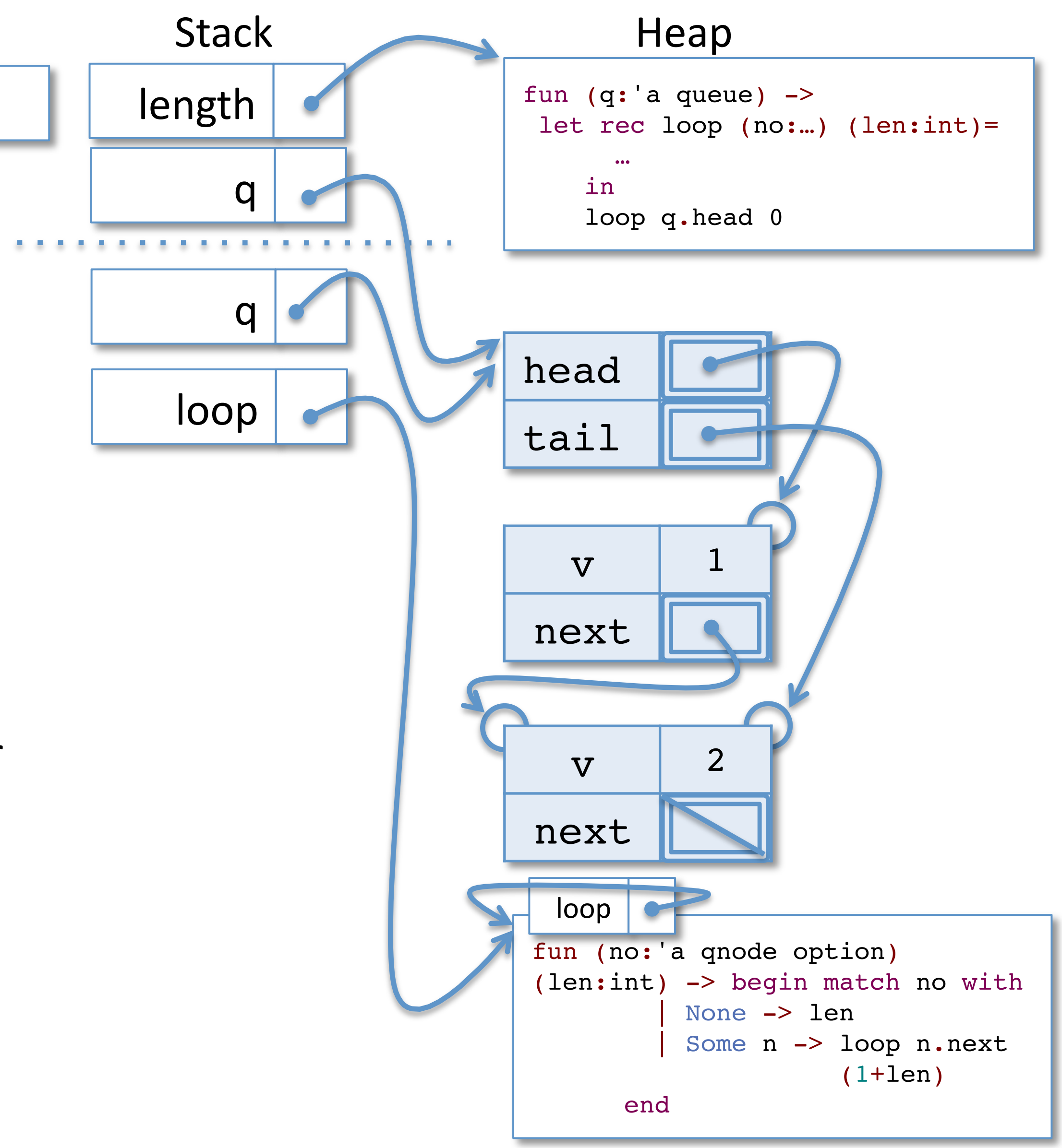
(1) No workspace is saved – there is no need do to that for tail calls

(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)
```

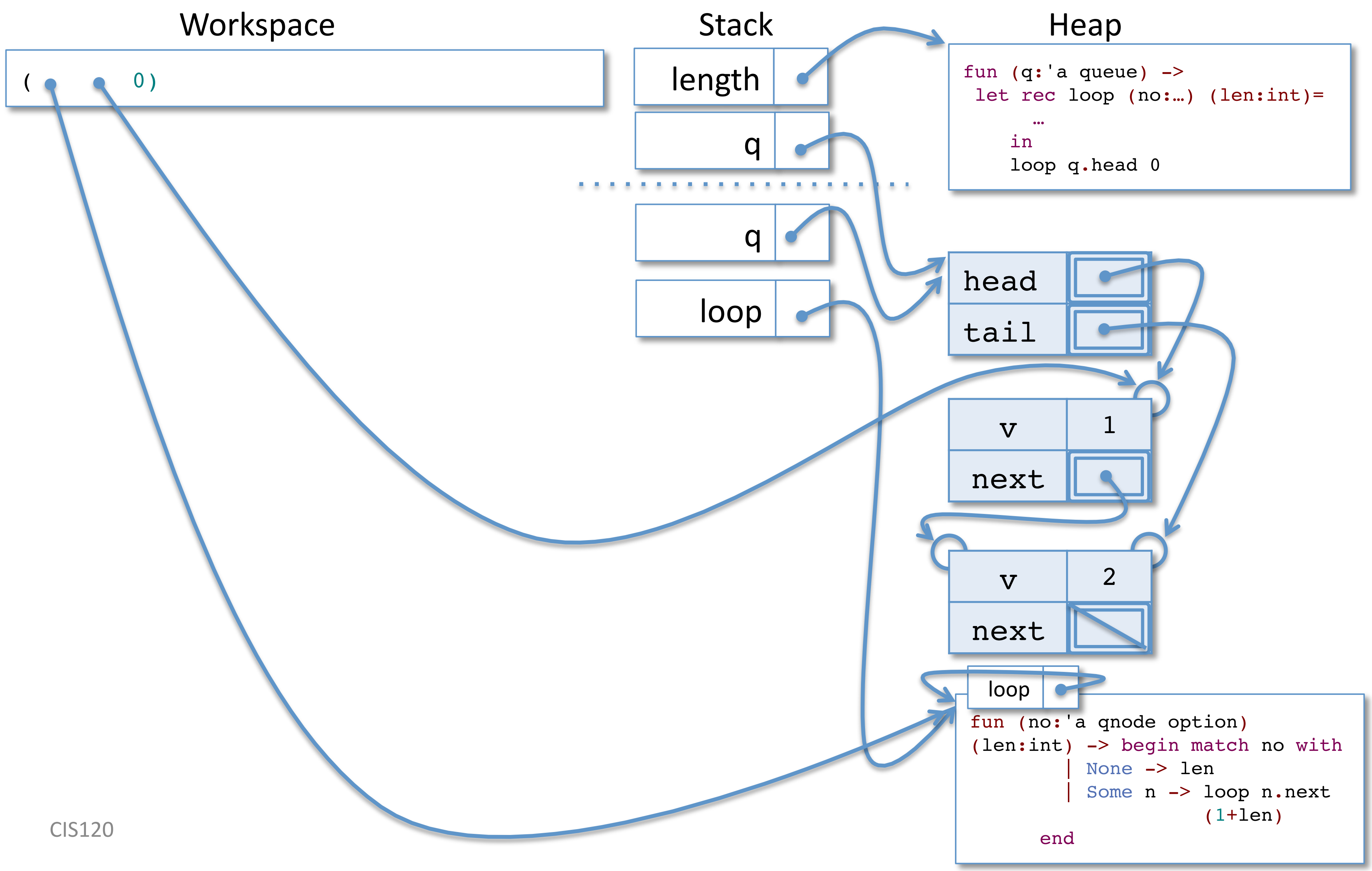


A detail we've been sweeping under the rug until now:

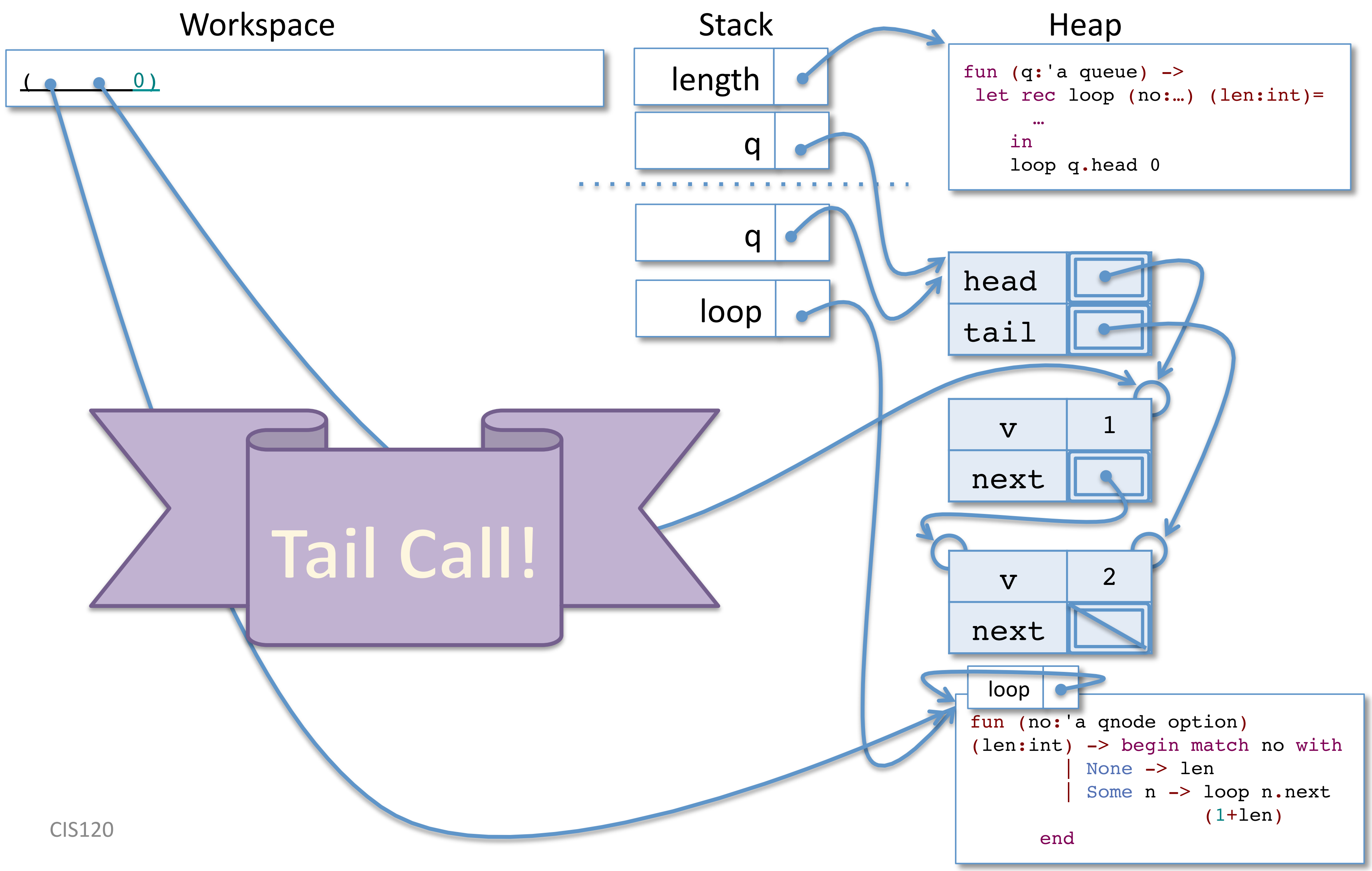
The *closure* of the local recursive function `loop` includes a binding for the loop function itself!

Why? The loop body mentions the loop identifier.

Tail Calls and Iterative length



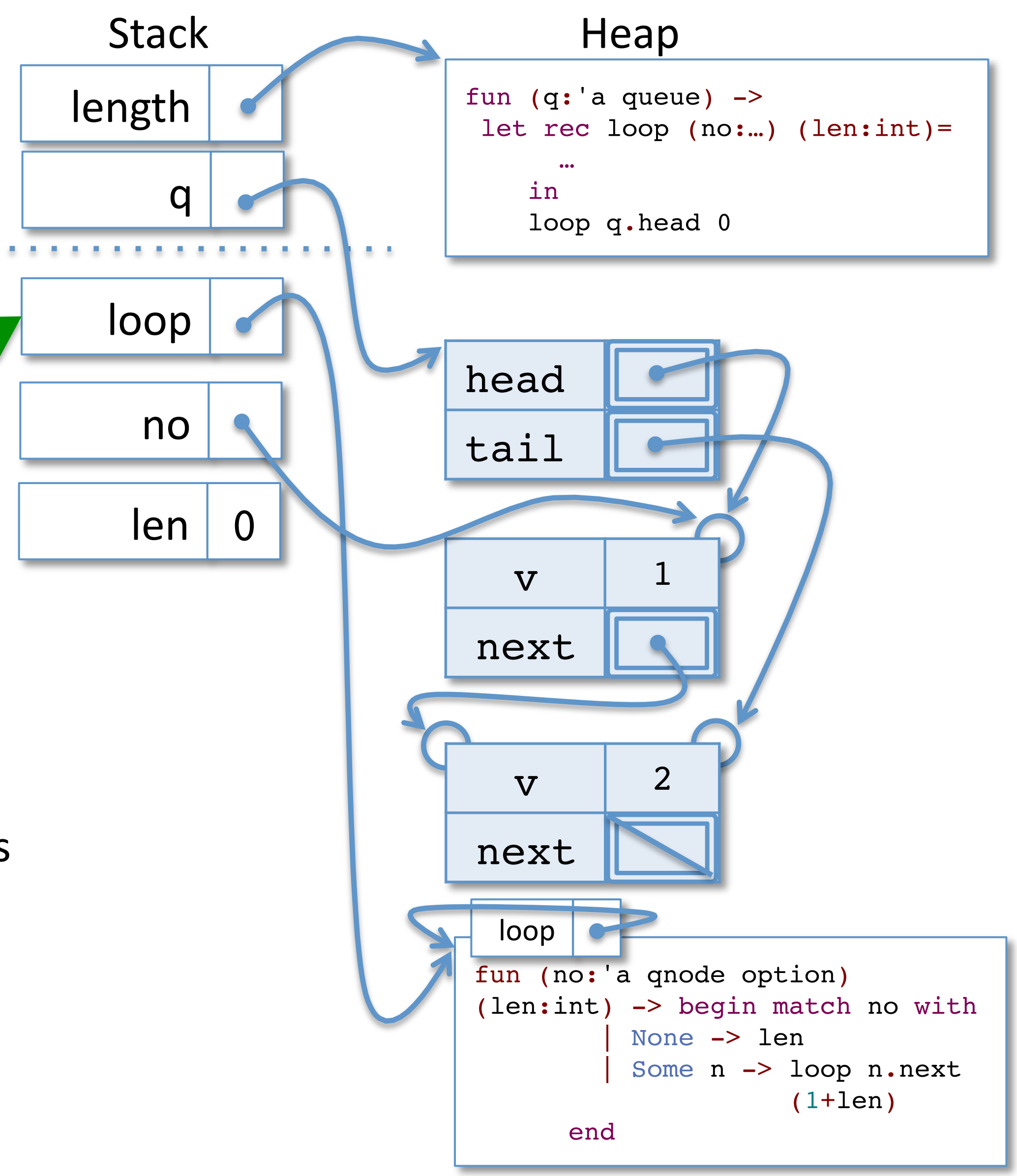
Tail Calls and Iterative length



Tail Calls and Iterative length

Workspace

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

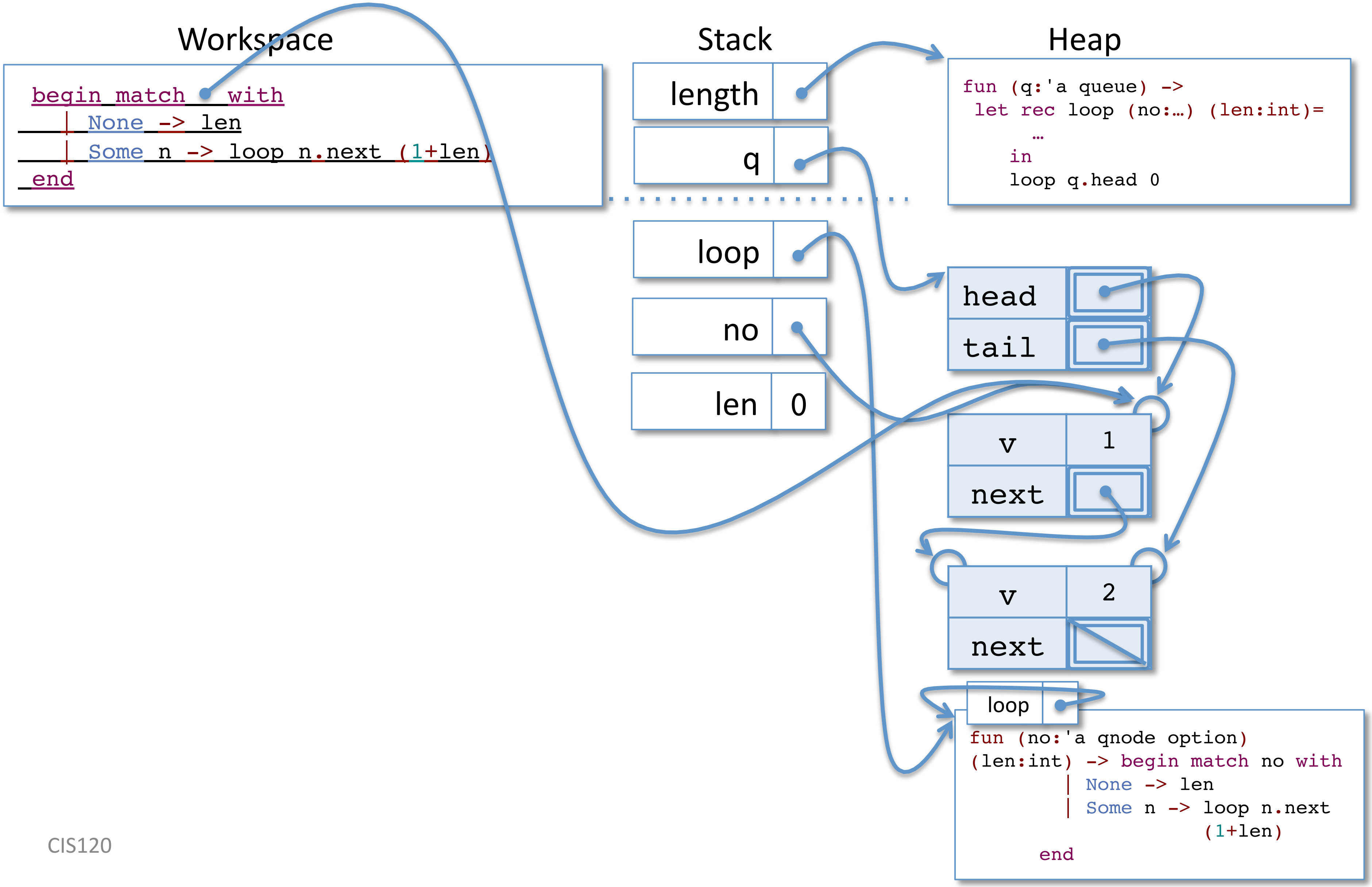


This binding comes from the closure.

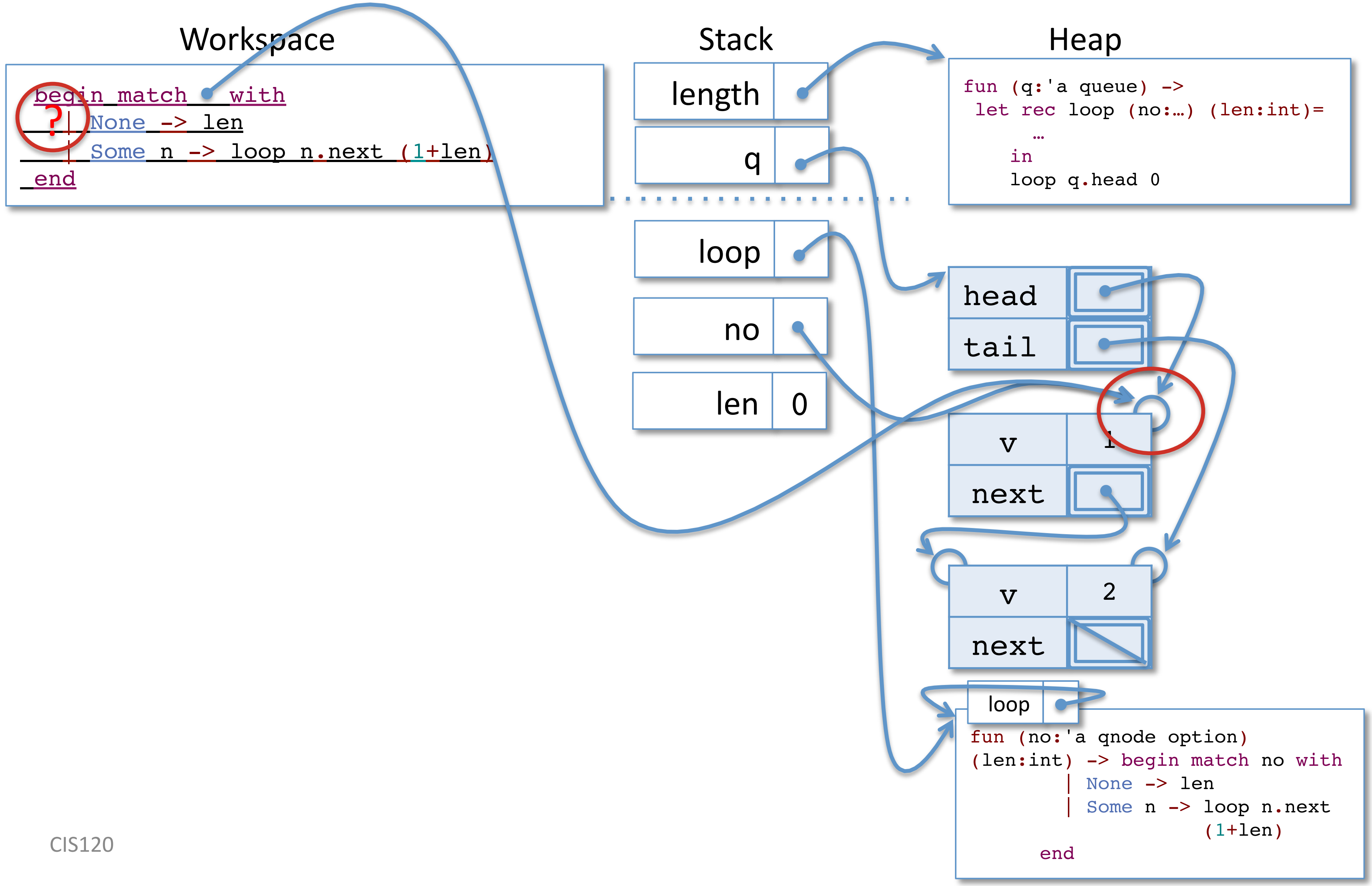
Notes:

- 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

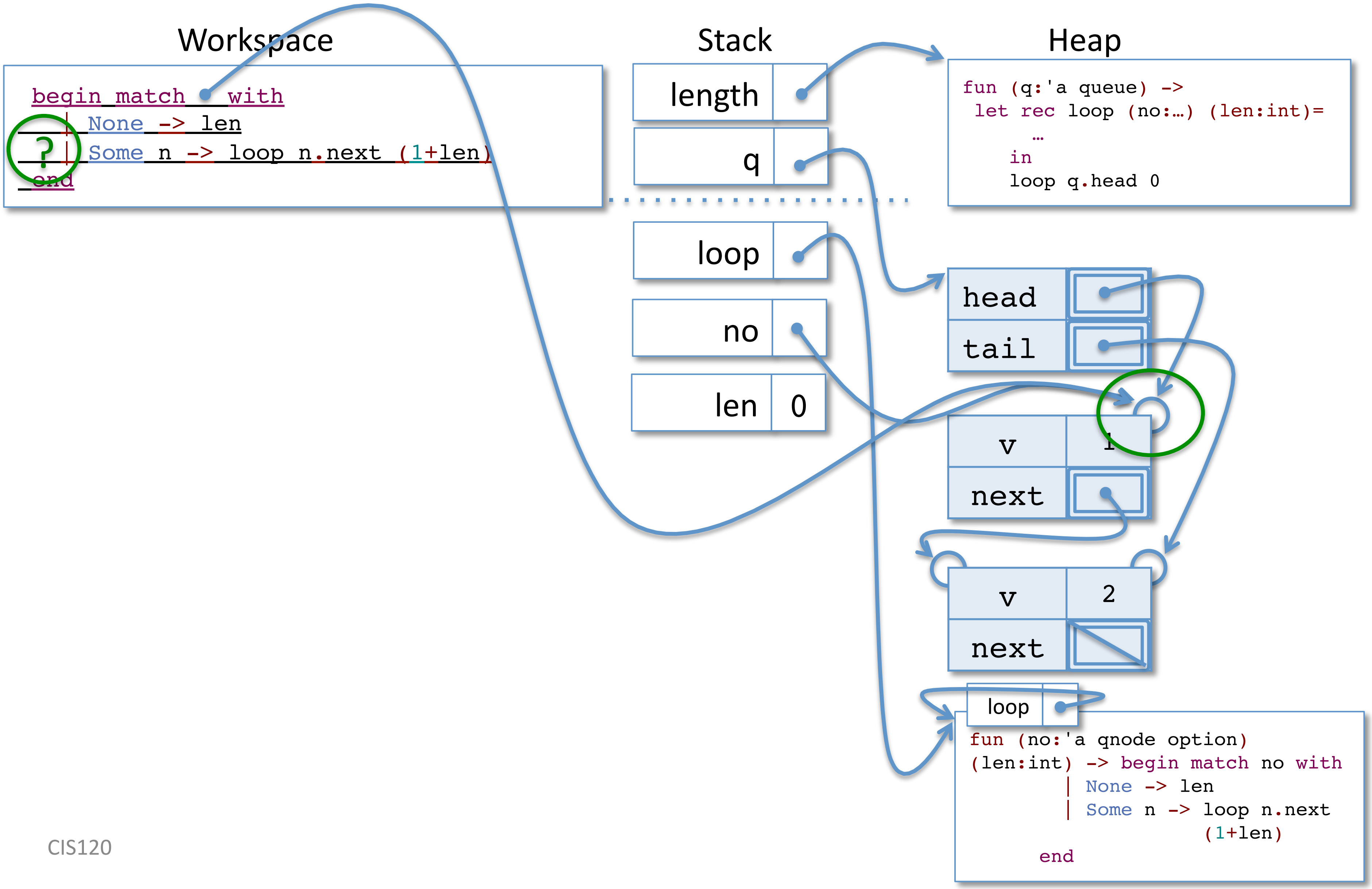
Tail Calls and Iterative length



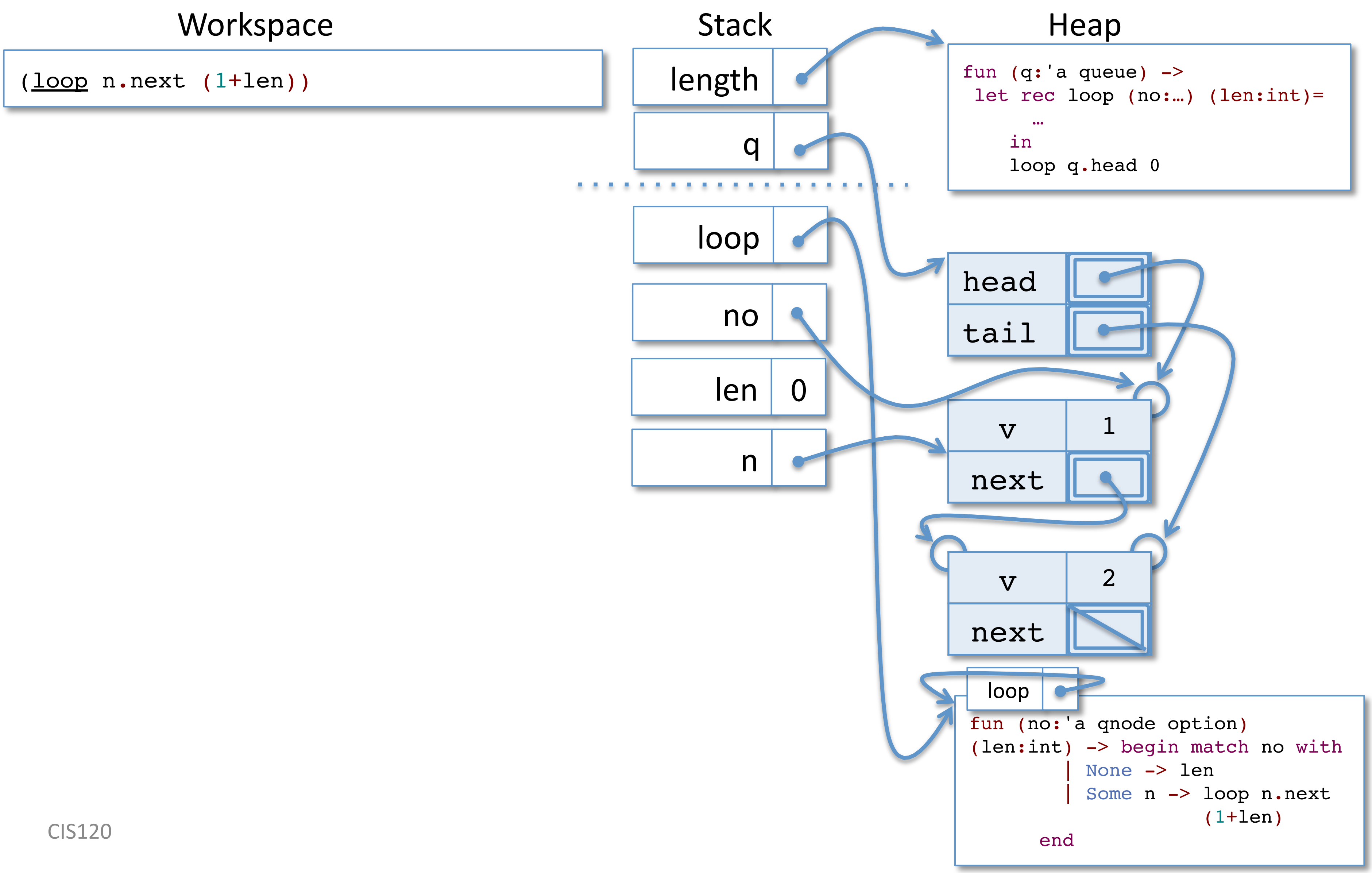
Tail Calls and Iterative length



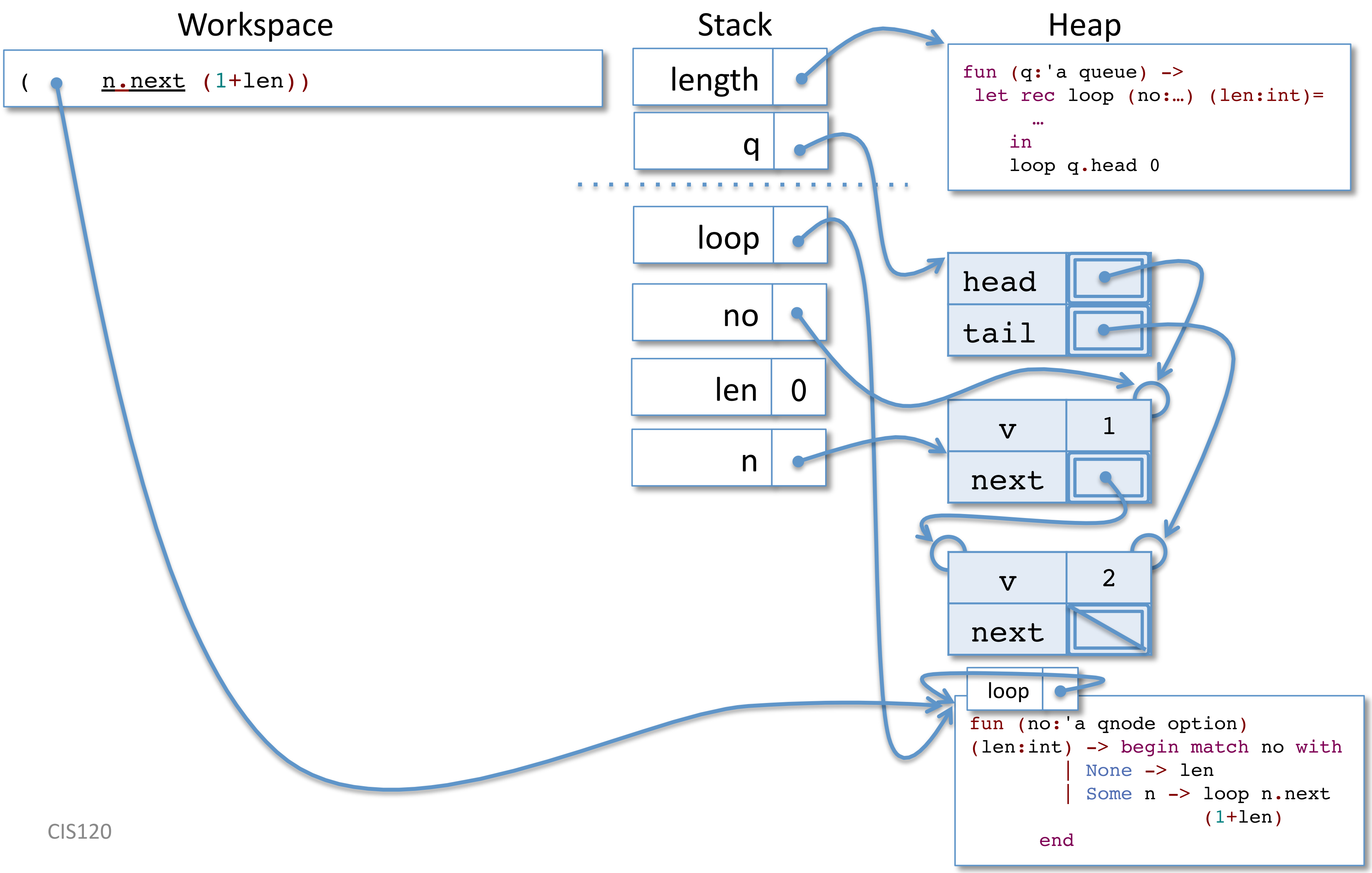
Tail Calls and Iterative length



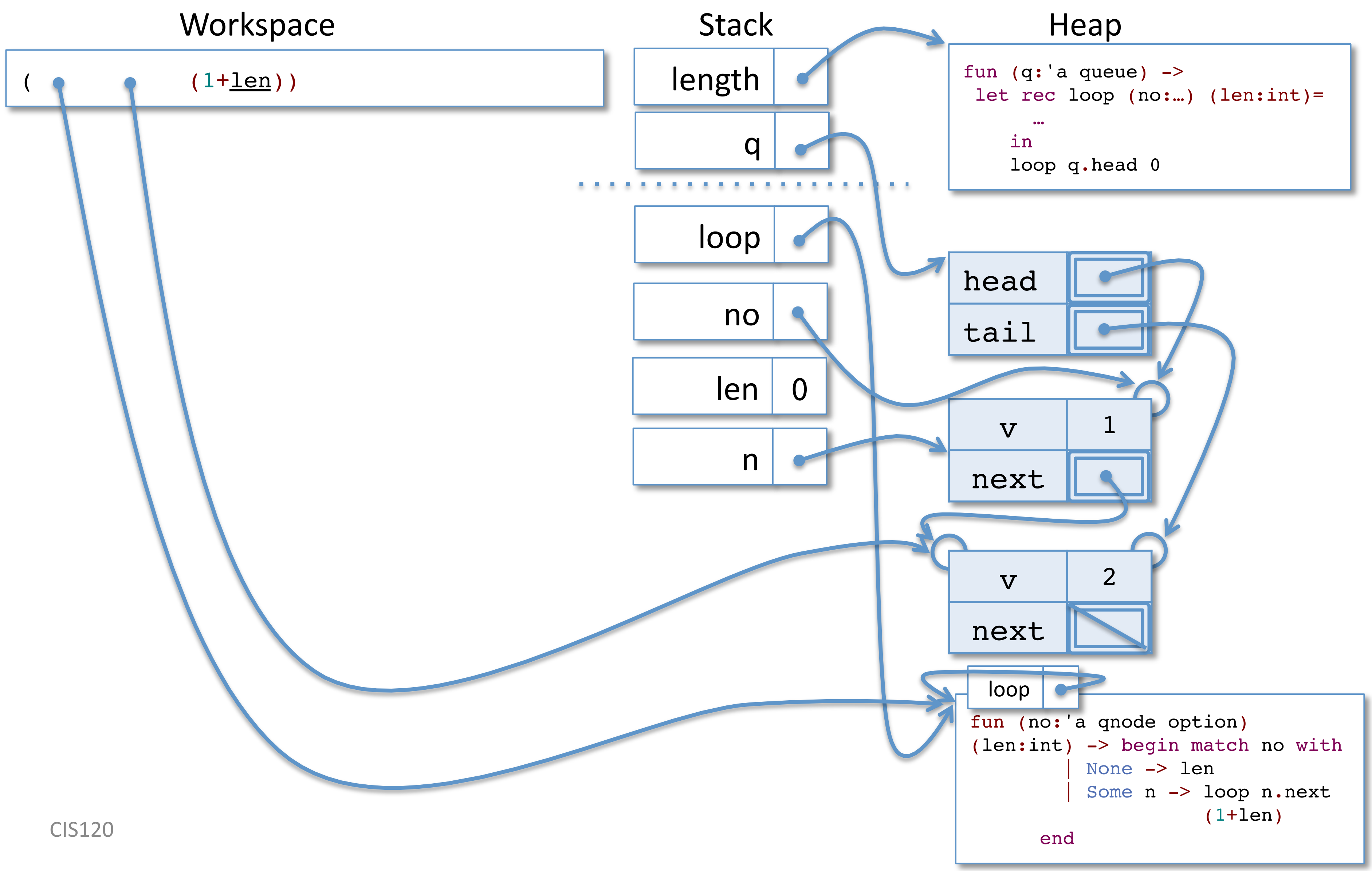
Tail Calls and Iterative length



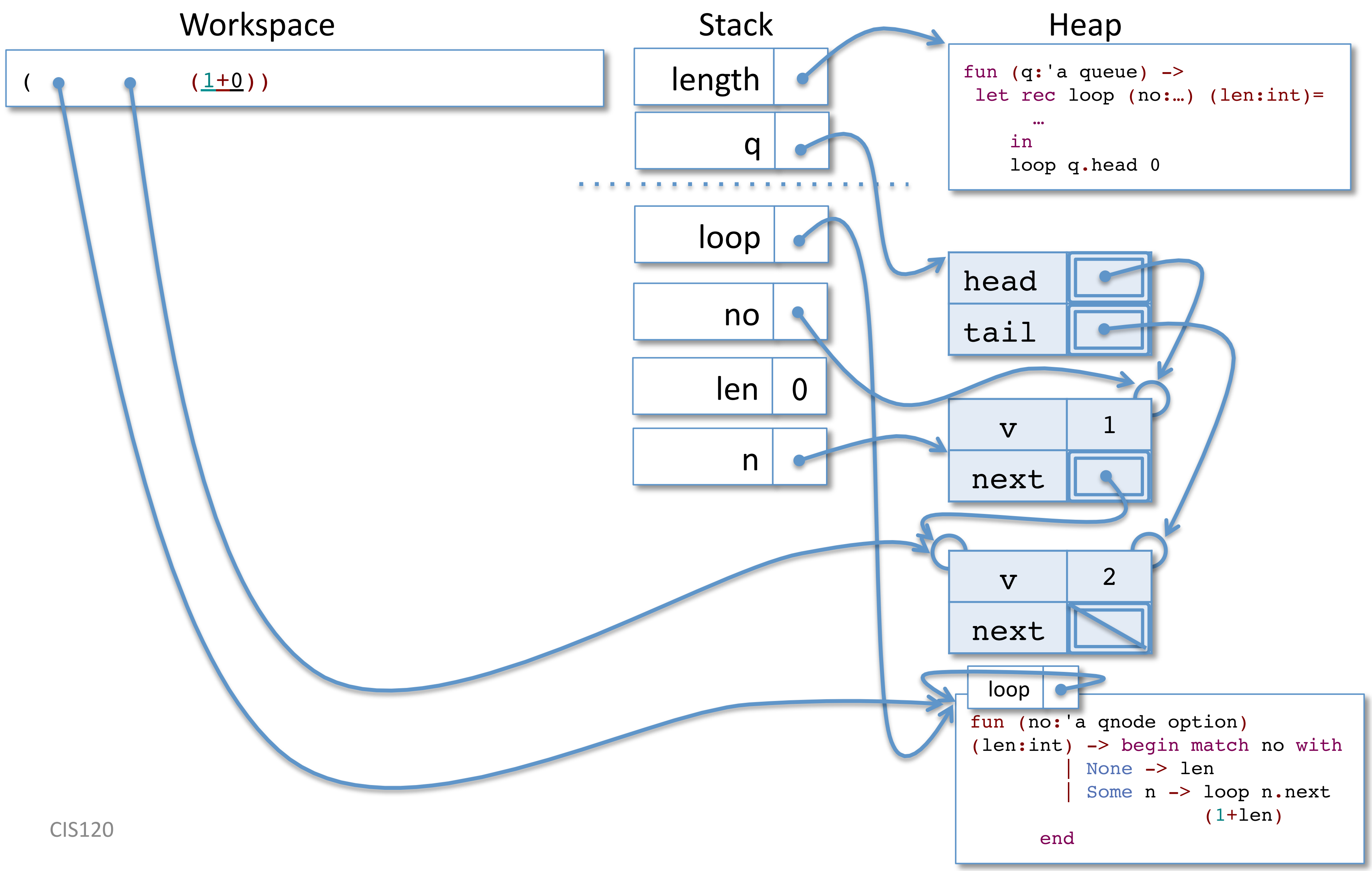
Tail Calls and Iterative length



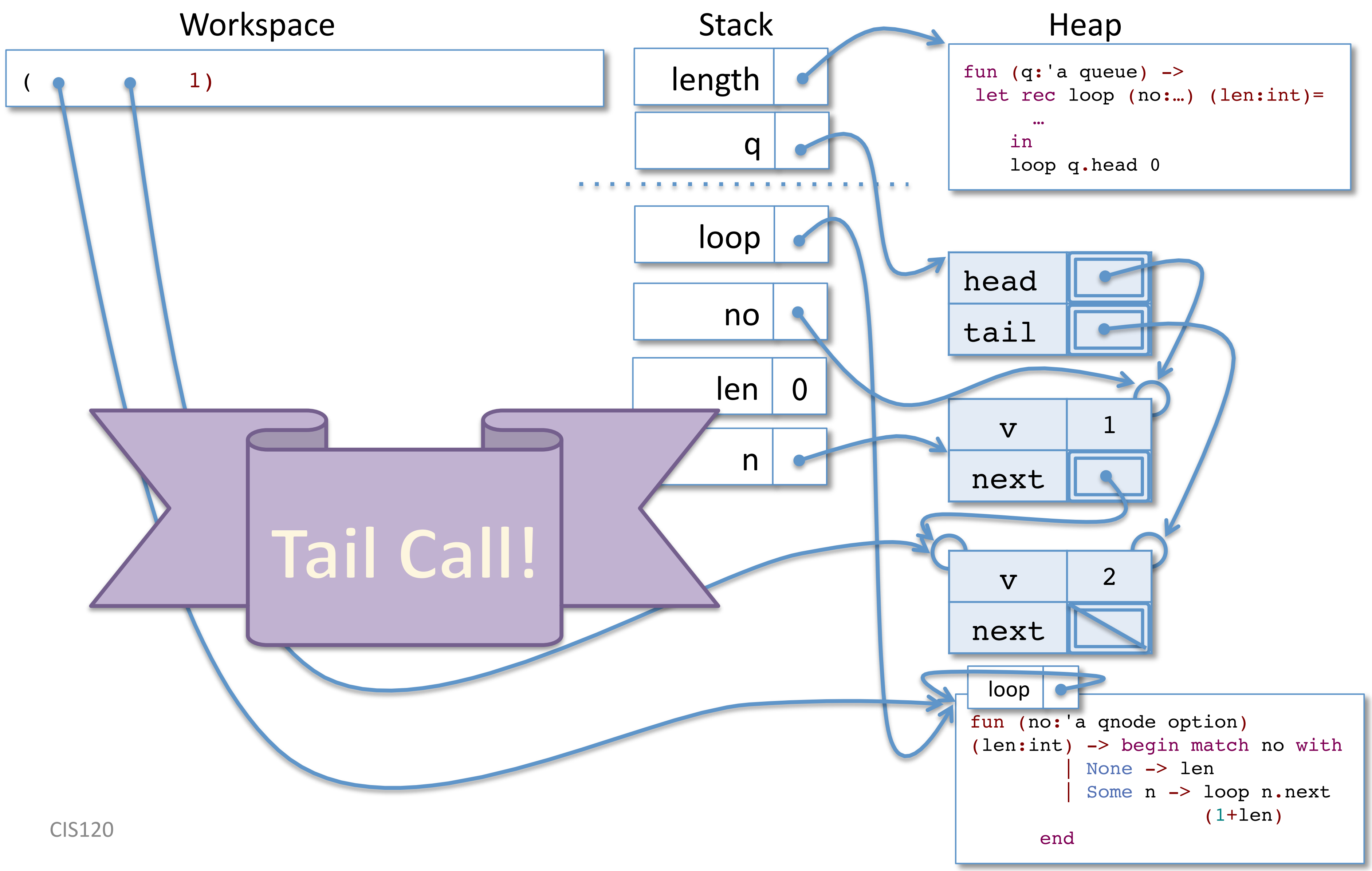
Tail Calls and Iterative length



Tail Calls and Iterative length



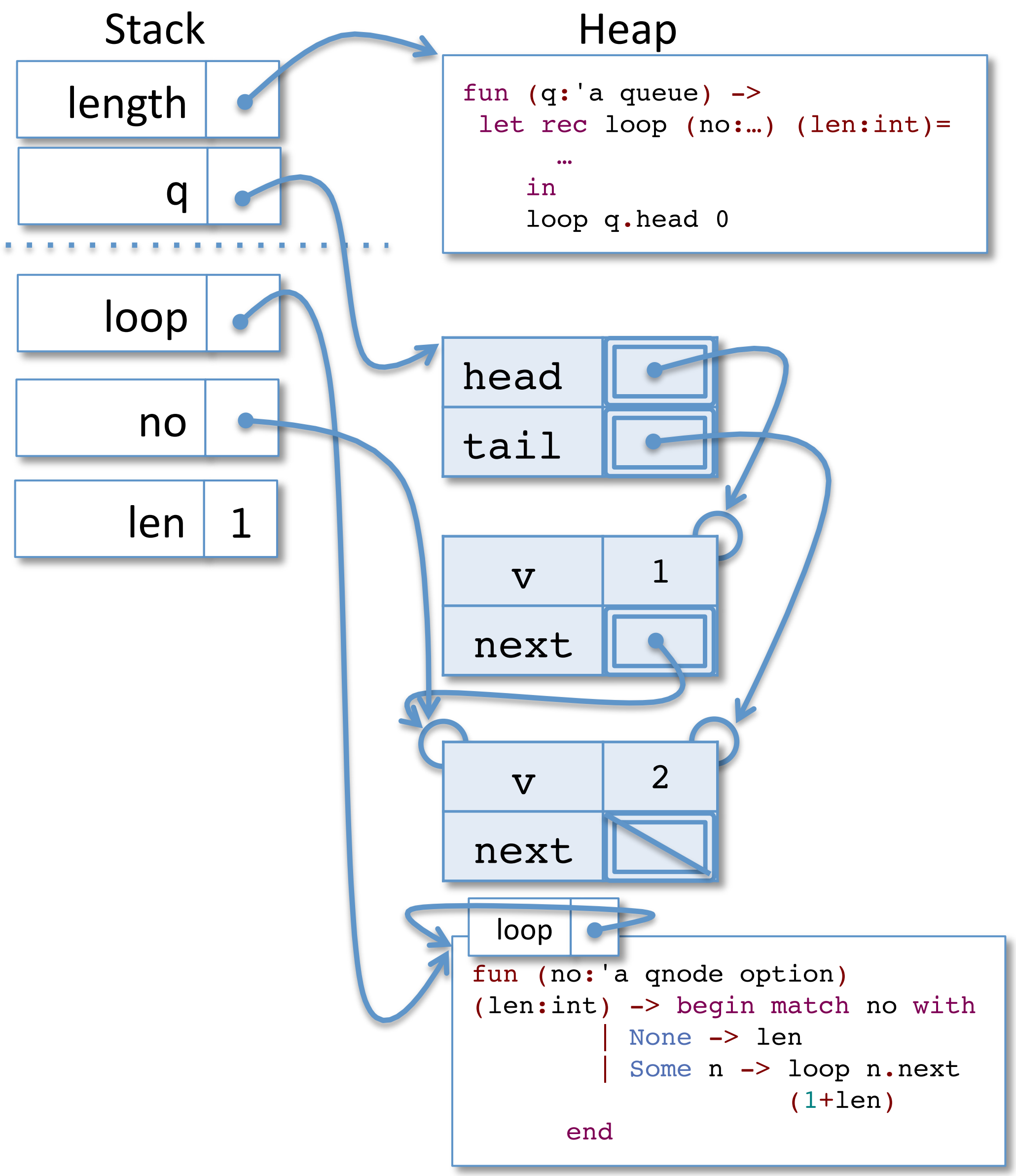
Tail Calls and Iterative length



Tail Calls and Iterative length

Workspace

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

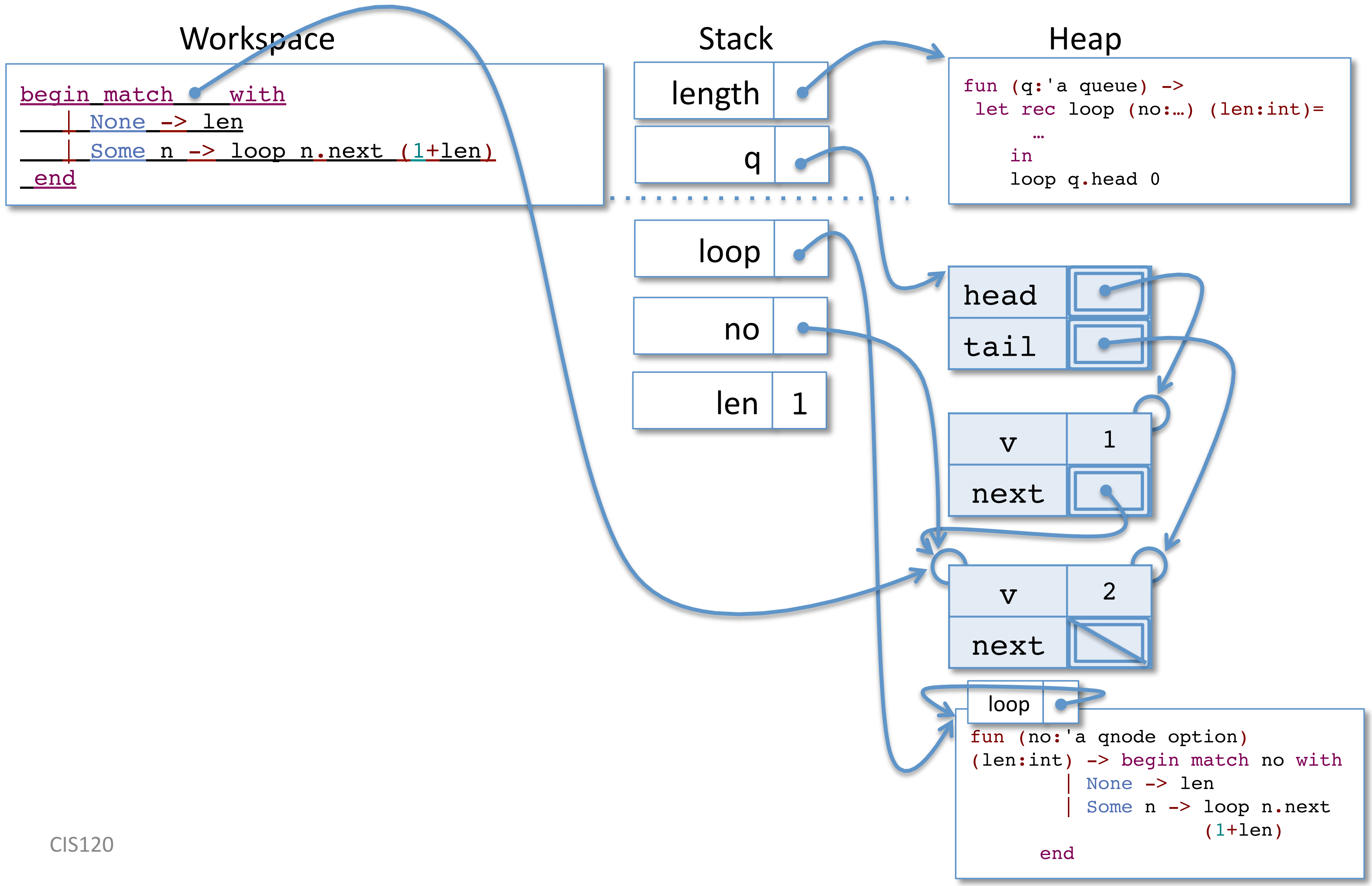


Note: we popped the old values of loop, no, len, and n when we did the tail call. Then we pushed the new values of loop, no, and len.

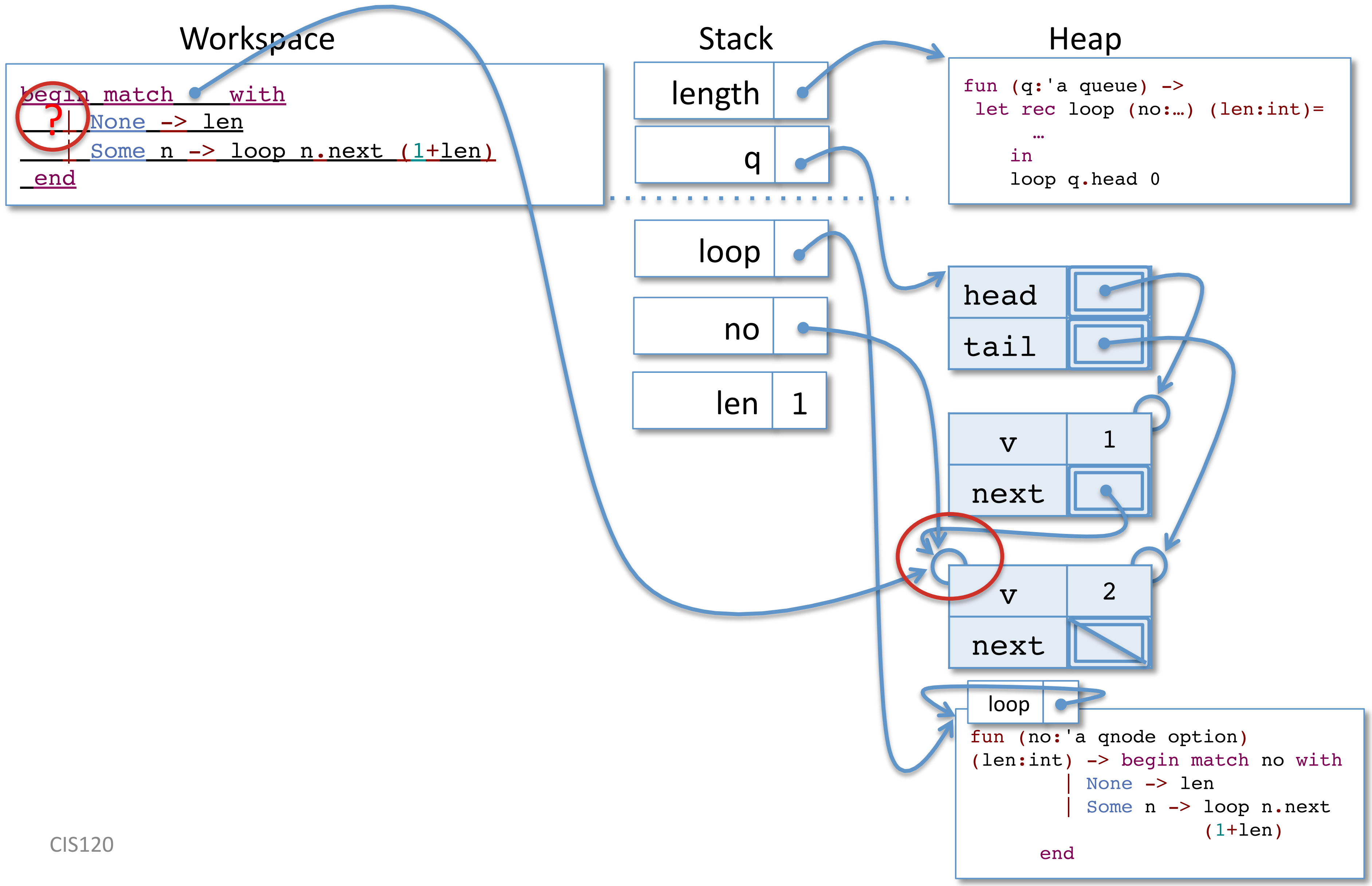
This leaves the stack in almost the same shape as when we first called loop.

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

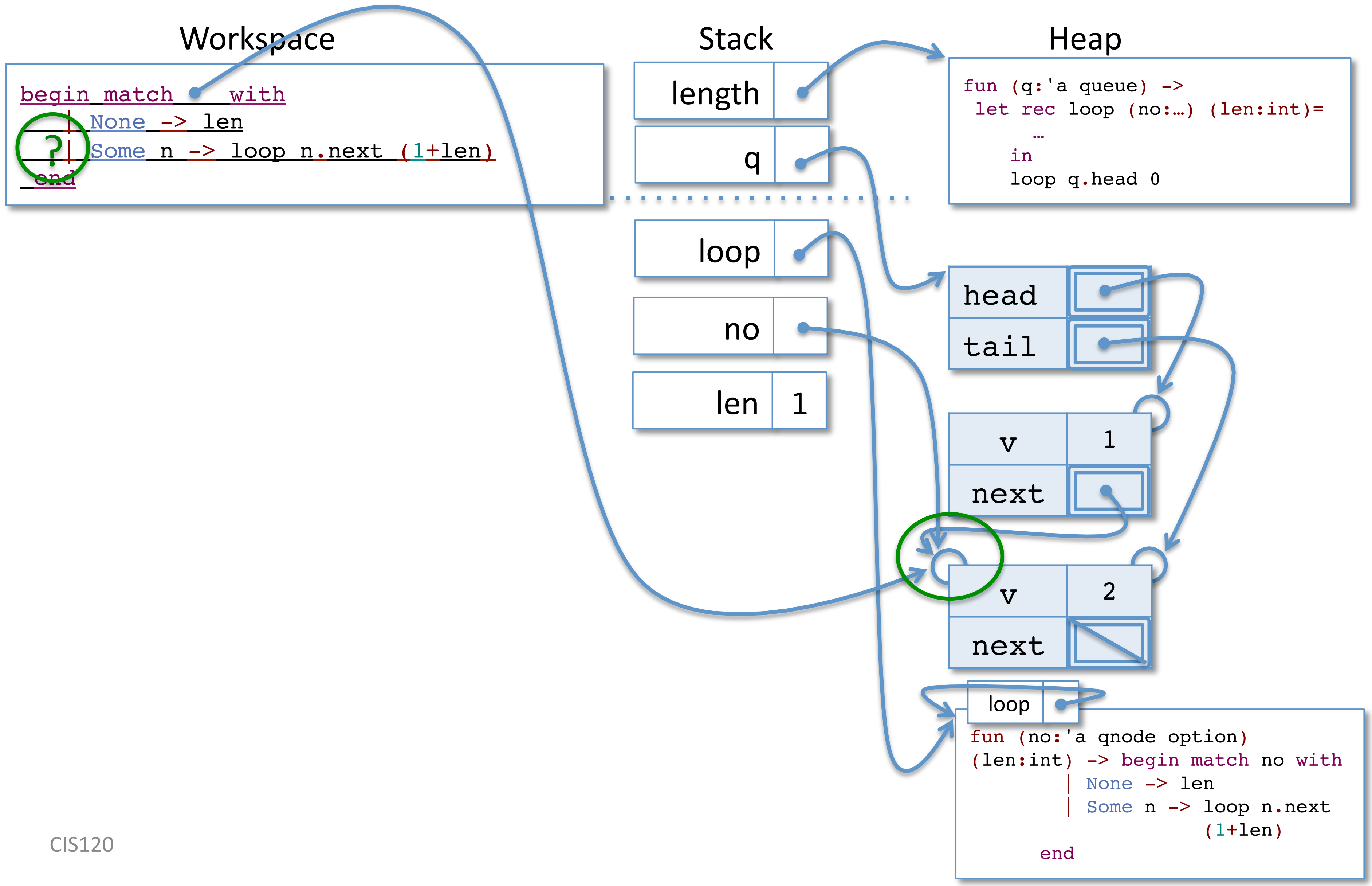
Tail Calls and Iterative length



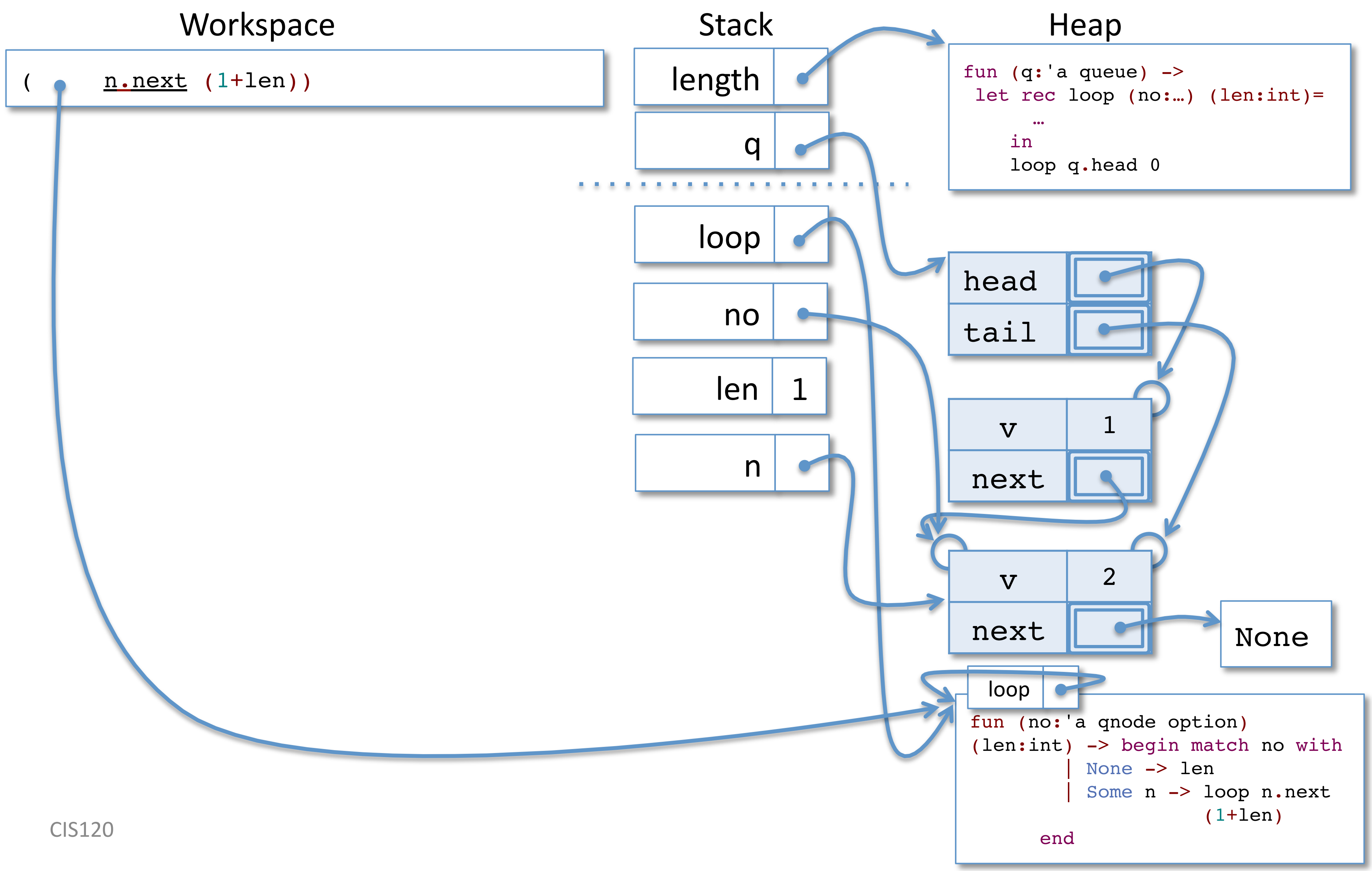
Tail Calls and Iterative length



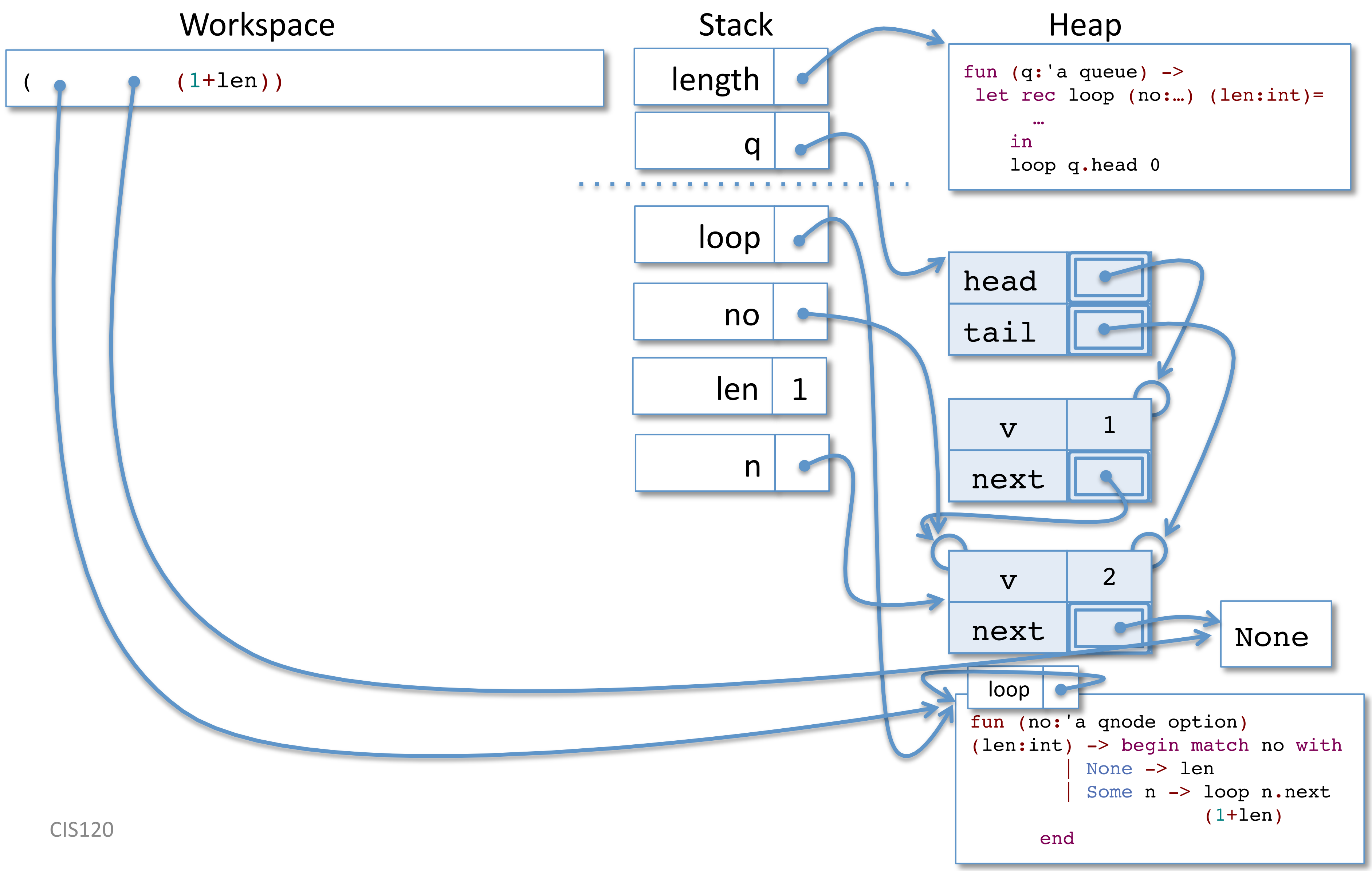
Tail Calls and Iterative length



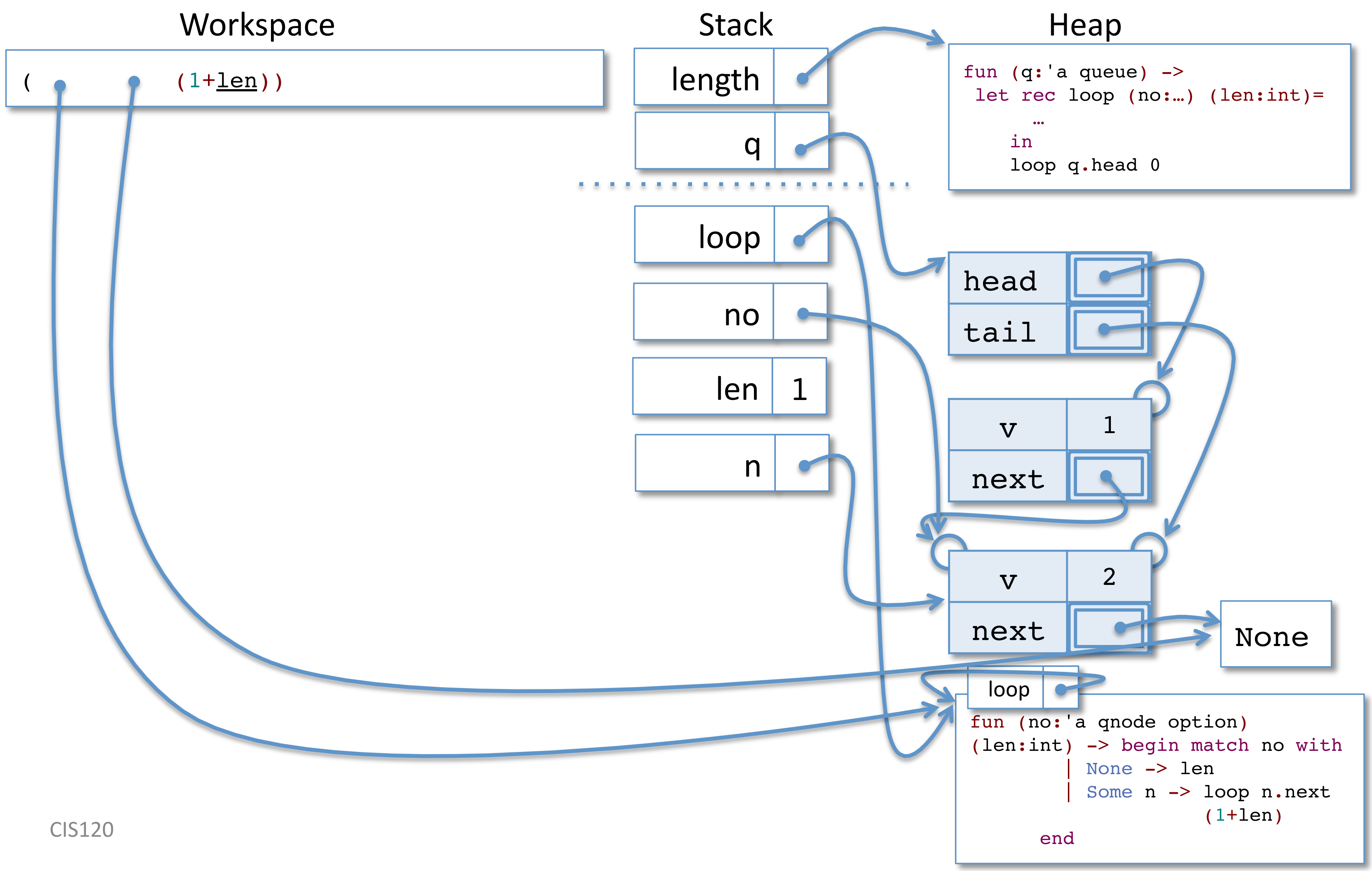
Tail Calls and Iterative length



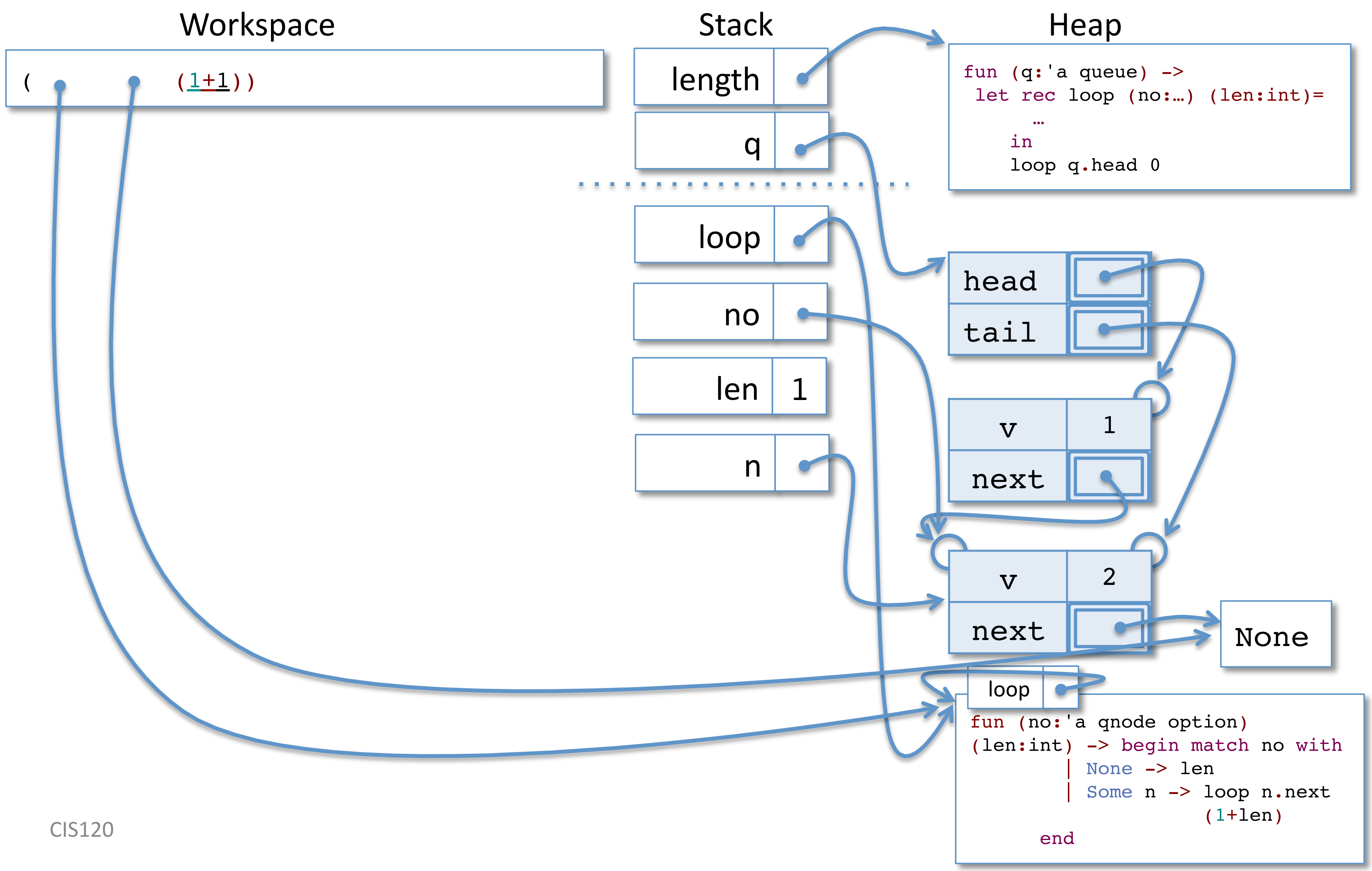
Tail Calls and Iterative length



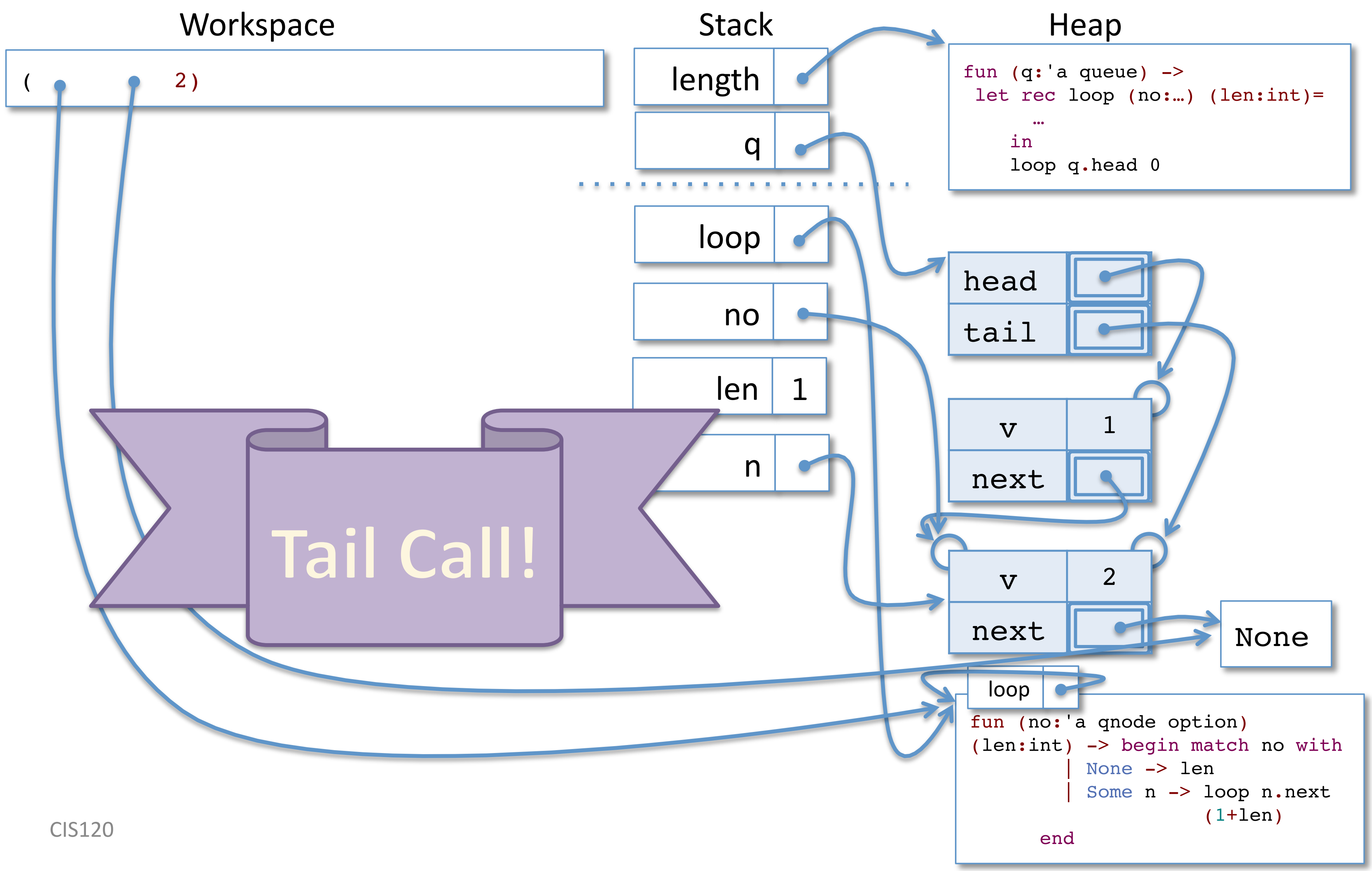
Tail Calls and Iterative length



Tail Calls and Iterative length



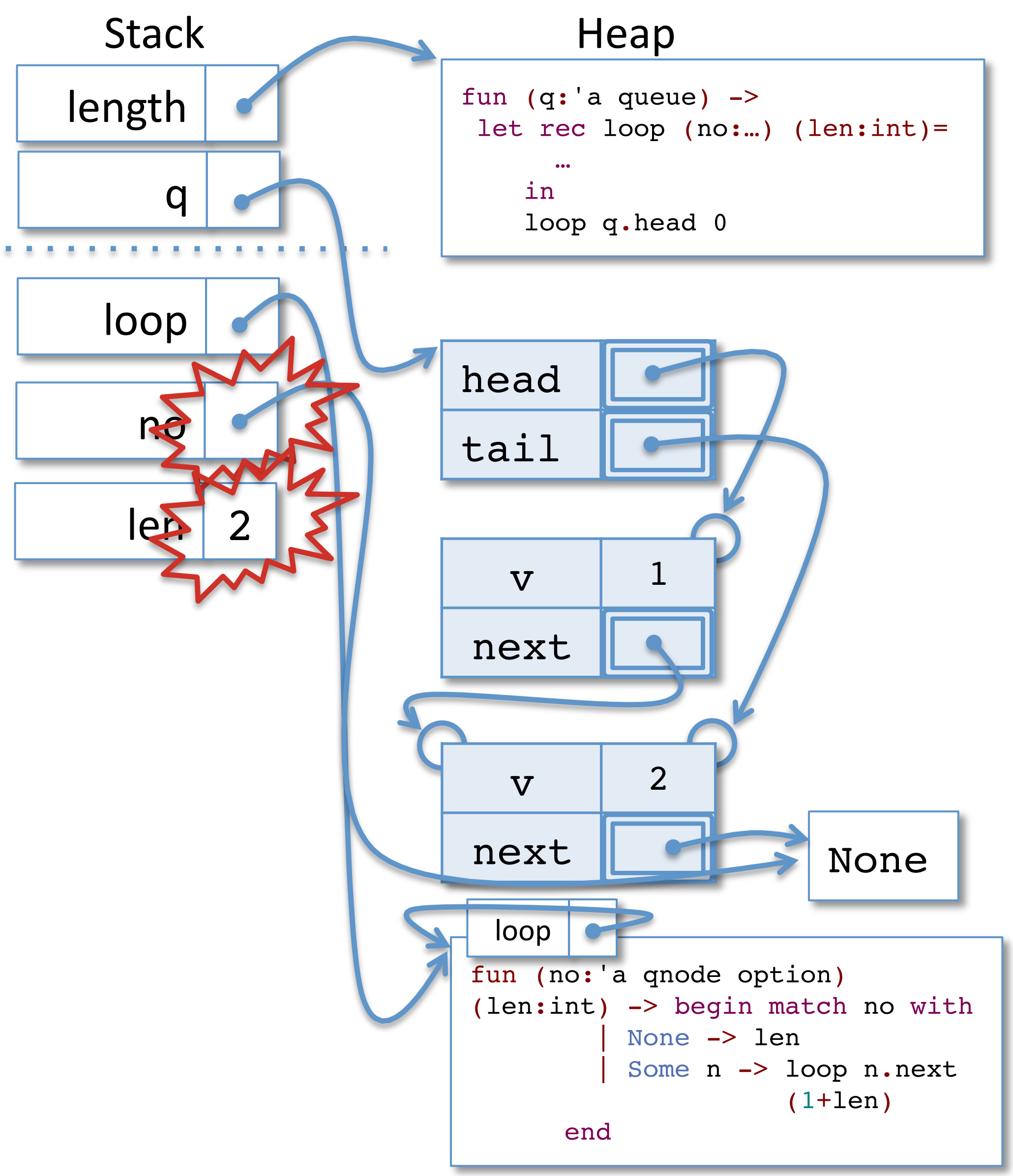
Tail Calls and Iterative length



Tail Calls and Iterative length

Workspace

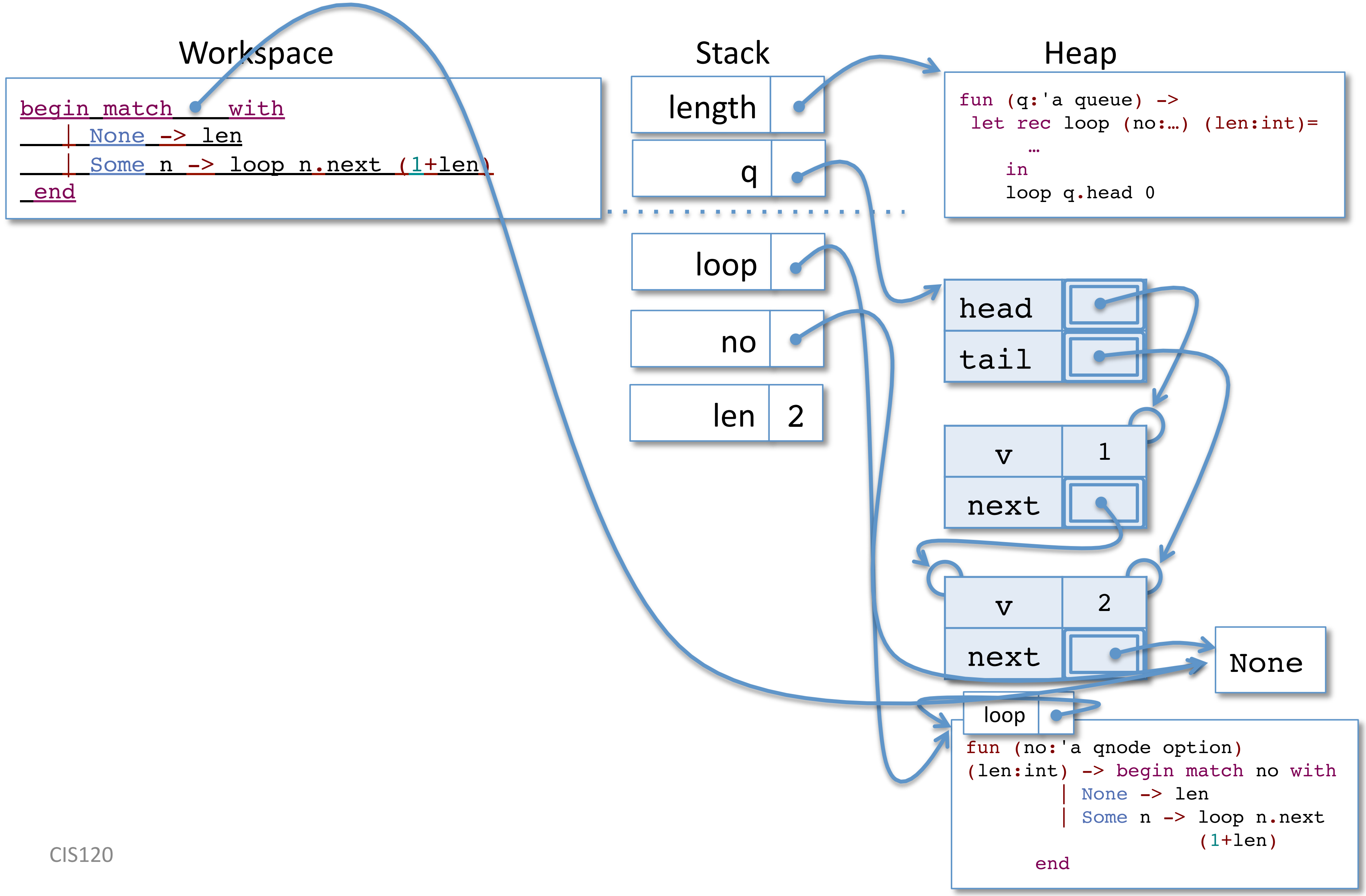
```
begin match no with
    | None -> len
    | Some n -> loop n.next (1+len)
 end
```



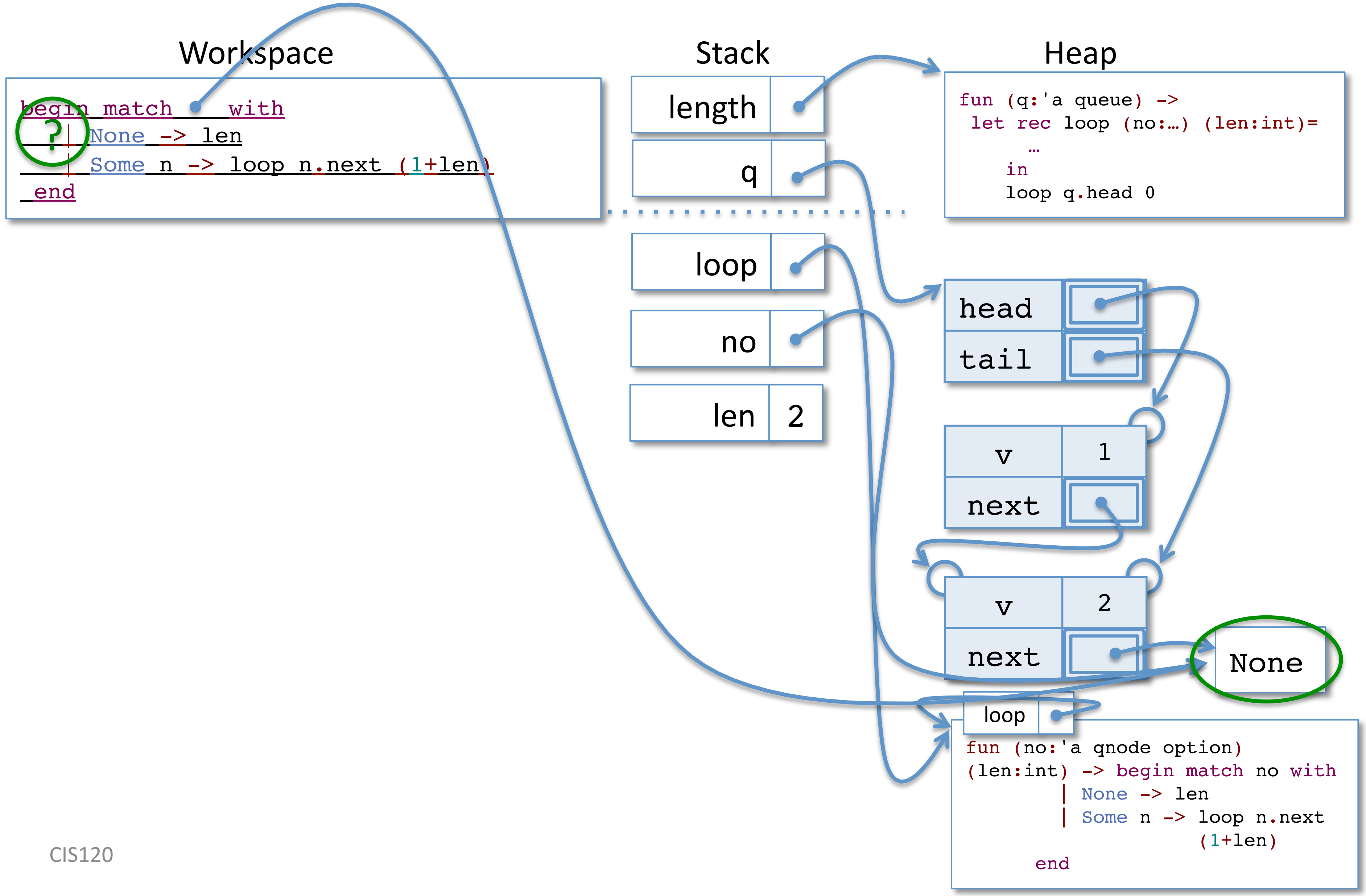
Note: Again, the tail call leaves the stack as before, but effectively updates the values of no and len.

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

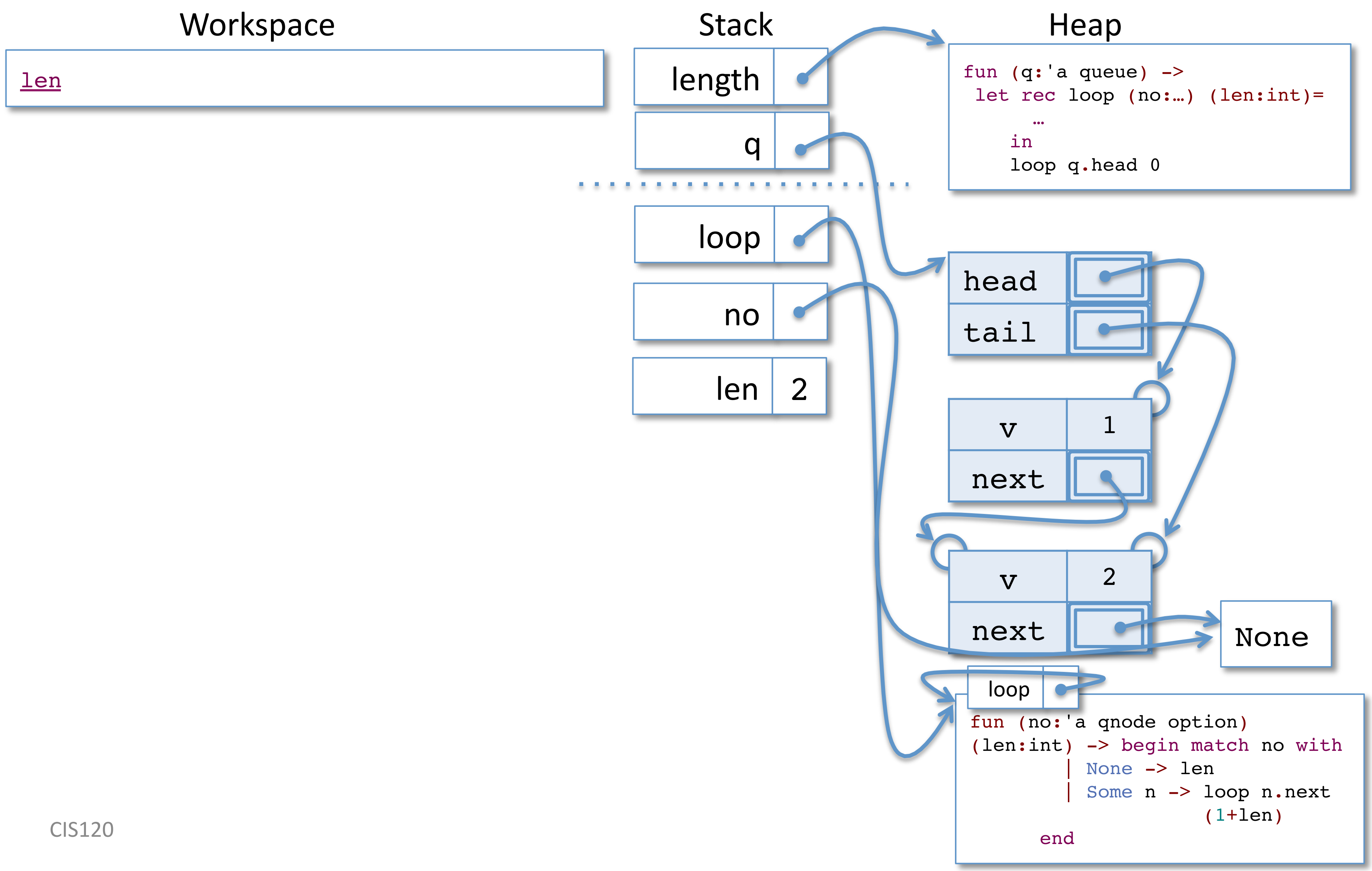
Tail Calls and Iterative length



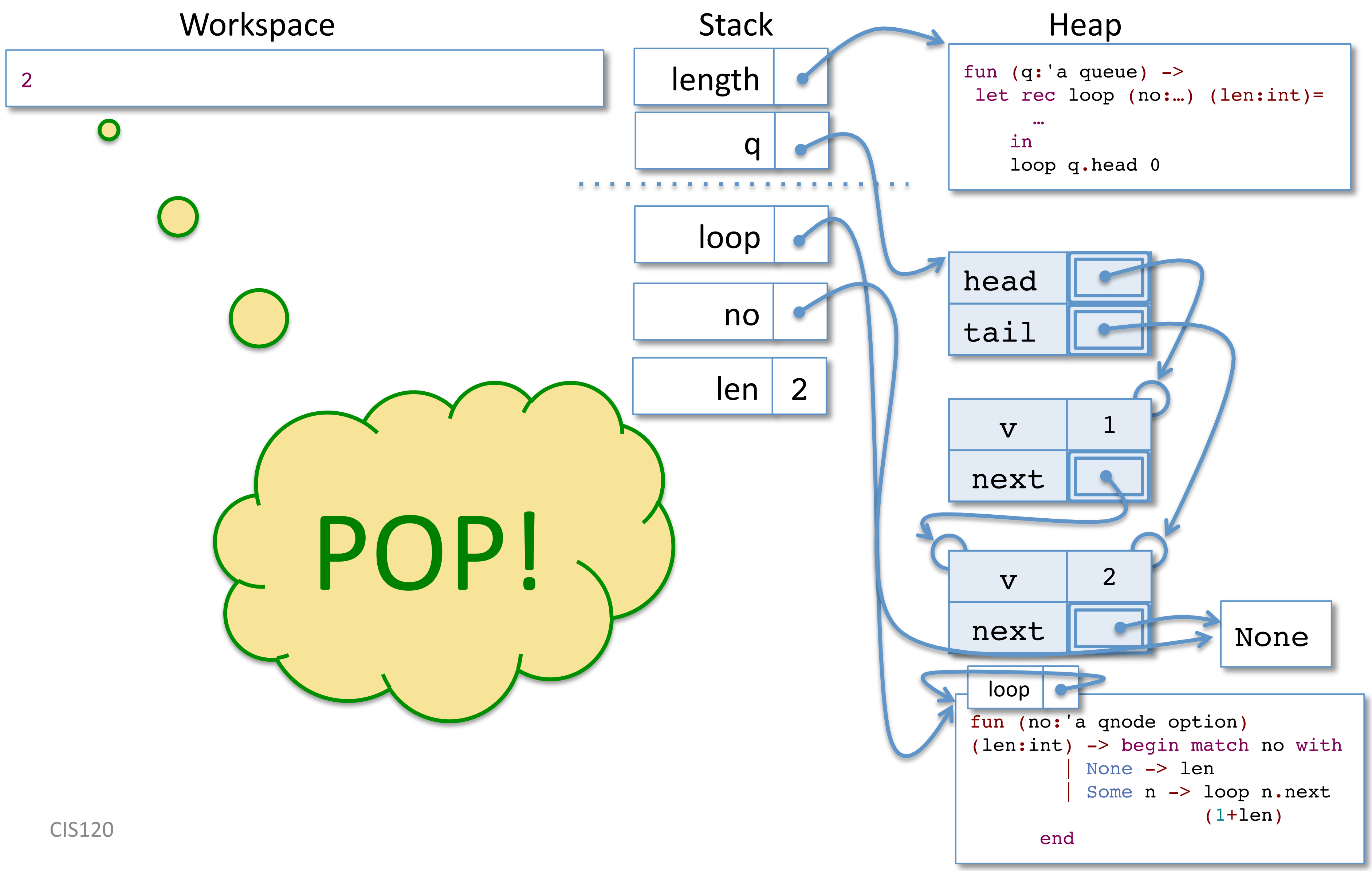
Tail Calls and Iterative length



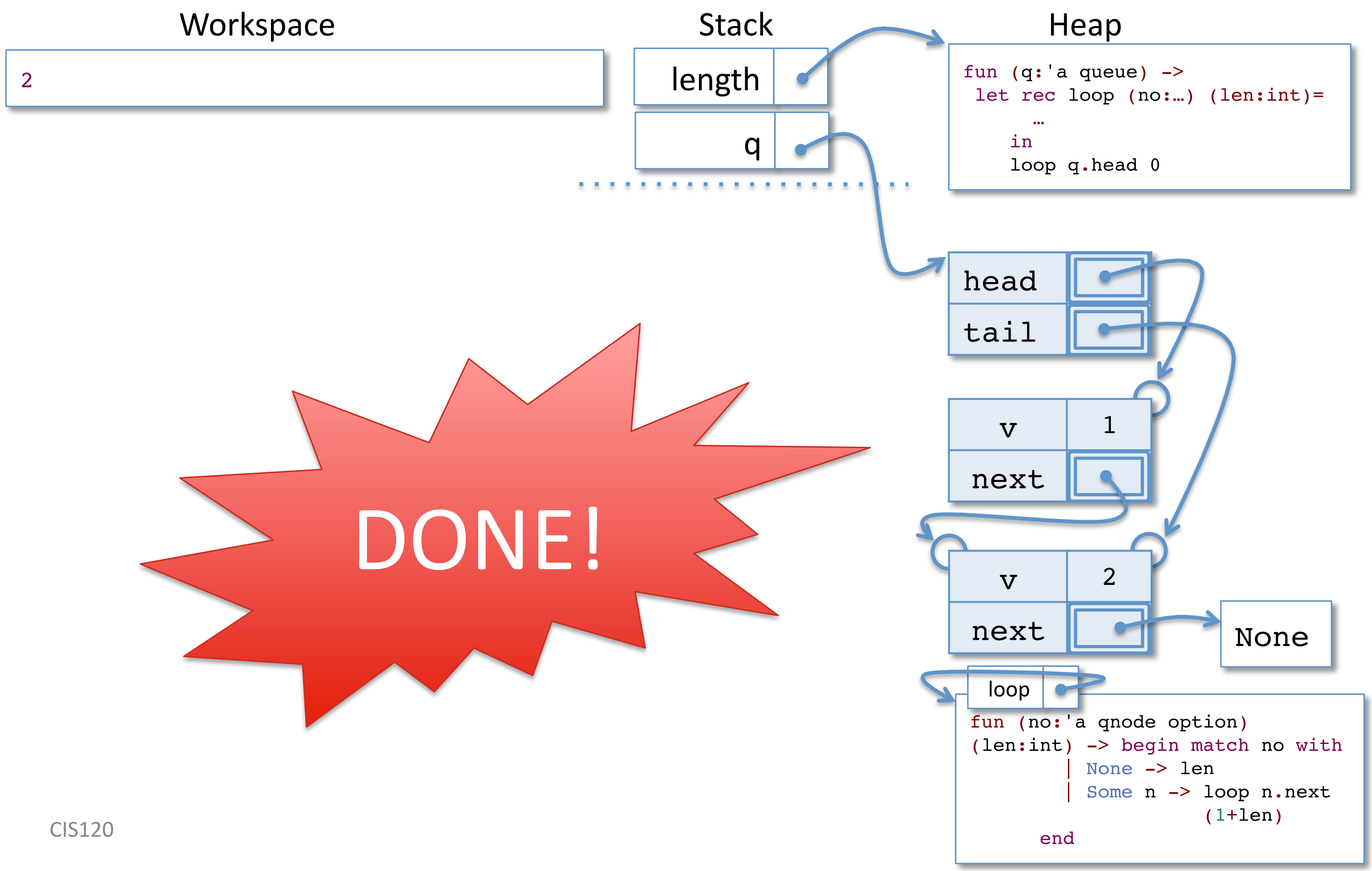
Tail Calls and Iterative length



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

Is this program iterative (i.e. does it use *tail* recursion)?

```
(* 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) : 'a list =
      begin match no with
      | None -> []
      | Some n -> n.v :: loop n.next
      end
  in loop q.head
```

1. Yes
2. No
3. I can’t tell

Is this program tail recursive?

```
(* 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 []
```

1. Yes
2. No
3. I can't tell

Is this program tail recursive?

```
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 -----"
```

1. Yes
2. No
3. I can't tell

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…

Infinite Loops

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

- This program will go into an infinite loop.
- Every recursive call allocates new memory in the heap. Will produce an Out_of_memory error.

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”.

Checking Queue validity

Detecting Loops

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`

- A queue operation *may assume* that these invariants hold of its inputs, and *must ensure* that the invariants hold when it's done.

valid

```
(* Determine whether the q is valid *)
let valid (q: 'a queue) : bool =
      begin match (q.head,q.tail) with
      | (None,None) ->  true
      | (Some(qh),Some(qt)) ->
          begin match get_tail qh with
            | Some n -> qt == n   (* tail is the last node *)
            | None -> false
          end
      | (_,_) ->  false
      end
```

get_tail (using iteration)

```
(* get the tail (if any) from a queue *)
let rec get_tail (q: 'a queue) : 'a qnode option =
    let rec loop (qn: 'a qnode) (seen: 'a qnode list)
        : 'a qnode option =
      begin match qn.next with
        | None -> Some qn
        | Some n ->
            if contains_alias n seen then None
            else loop n (qn::seen)
      end
    in loop q.head []
```

- This function does *not* assume that q has no cycles.
  - It returns Some n if n is a “tail” reachable from q.head
  - It returns None if there is a cycle in the queue
- The state is an index pointer and a list of all the nodes seen.
  - contains_alias is a helper function that checks to see whether n has an alias in the list

“Objects” and Hidden State

Objects in Java

class declaration

```
public class Counter {

   private int count;

   public Counter () {
      count = 0;
   }

   public int incr () {
      count = count + 1;
      return count;
   }

   public int decr () {
      count = count - 1;
      return count;
   }
}
```

class name

instance variable

constructor

methods

object creation and use

```
public class Main {

  public static void
      main (String[] args) {

        Counter c = new Counter();

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

  }
}
```

constructor invocation

method call

What is an Object?

- Object = Instance variables (fields) + Methods
  - Field = Mutable record
  - Methods = (Immutable) record of first-class functions that update the fields

- Objects *encapsulate* state when the methods are the ***only*** way to mutate the fields.

- Objects are first-class.

- Can we get similar behavior in OCaml?

An “incr” function

- This function increments a counter and return its new value each time it is called:

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

let ctr = { count = 0 }

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

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

Using Hidden State

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

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

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

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