Programming Languages and Techniques (CIS1200)

Lecture 16

Mutable Queues

Iteration & Tail Recursion

Chapter 16

Announcements

- Midterm 1 Grades and Solutions available
  - Submit Regrade requests via Gradescope by Friday, 3/7
- HW04 due Tuesday (at 11.59pm)
- Final Exam
  - Wednesday, May 7^{th}, 9-11am

(Mutable) Queue Interface

```
module type QUEUE =
sig
  (* abstract type *)
  type 'a queue

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

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

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

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

end
```

Implementing Linked Queues

Representing links

Data Structure for Linked 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:

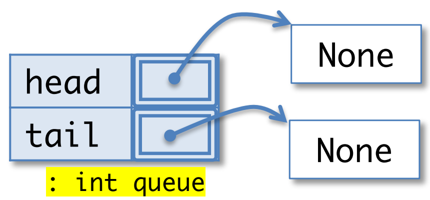
1. the "internal nodes" of the queue, with links from one to the next
2. a record with links to the head and tail nodes

All of these links are *options* so that the queue can be empty

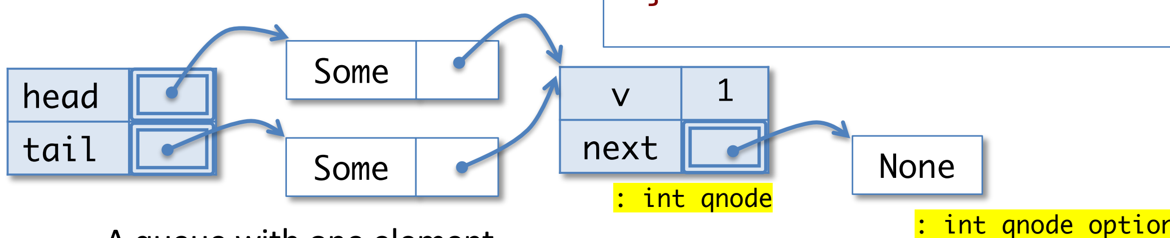
Queues in the Heap

Type Information

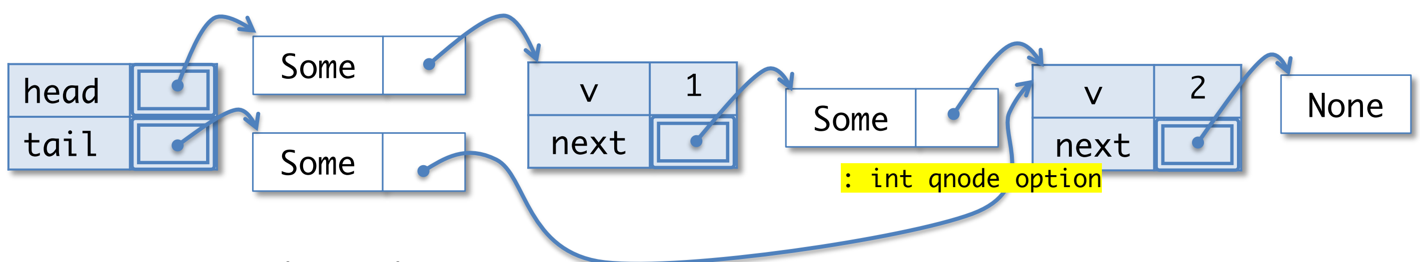
```
type 'a qnode = {
     v: 'a;
     mutable next : 'a qnode option
}
type 'a queue = { mutable head : 'a qnode option;
                  mutable tail : 'a qnode option
}
```



An empty queue

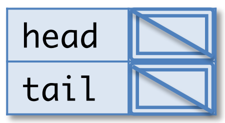


A queue with one element

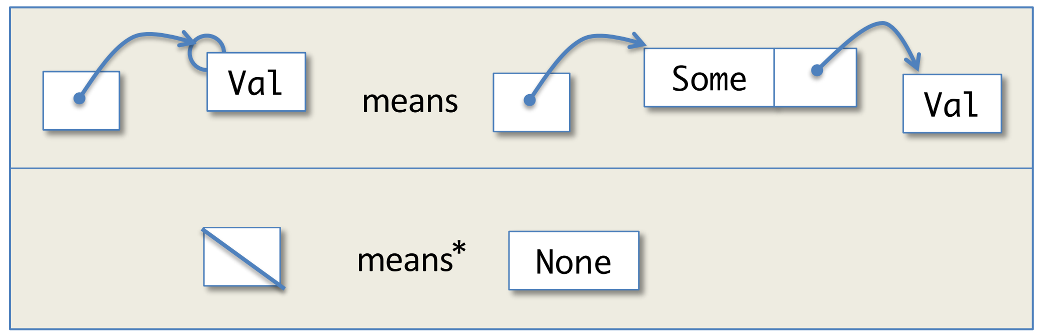


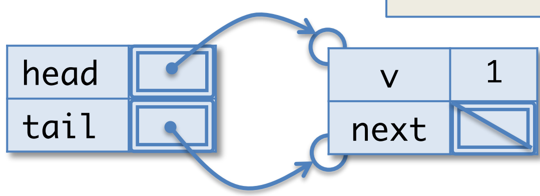
A queue with two elements

Visual Shorthand: Abbreviating Options



An empty queue





A queue with one element

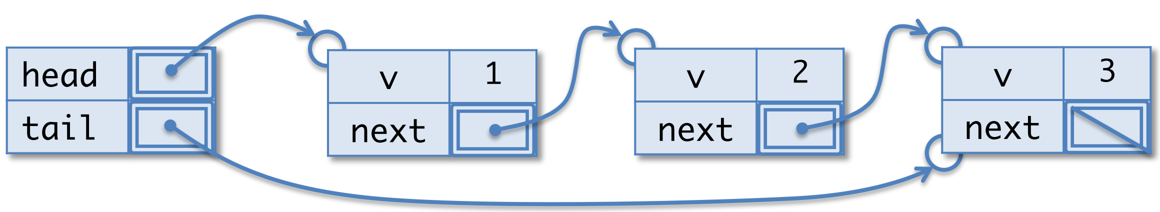
*Note: OCaml optimizes "nullary" constructors like Nil, None, Empty so that they aren't allocated in the heap. This is why

`None == None`

even though

`not ((Some x) == (Some x)).`

Be careful with reference equality and options!



A queue with three elements

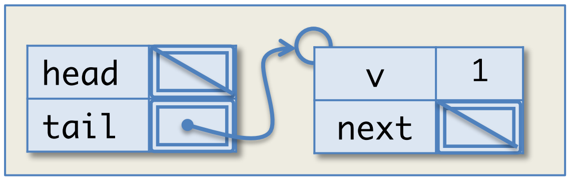
Linked Queue Invariants

Just as we imposed some restrictions on which trees count as legitimate Binary Search Trees, we require that Linked Queues satisfy the following representation *invariants*:

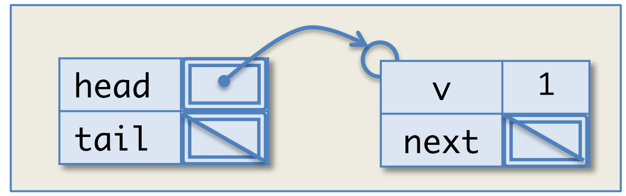
Either:
(1) `head` and `tail` are both None (i.e., the queue is empty)
or
(2) `head` is `Some n1`, `tail` is `Some n2` and
- n2 is reachable from n1 by following 'next' pointers
- `n2.next` is None

- We can prove that these properties suffice to rule out all of the "bogus" examples.
- Each queue operation may assume that these invariants hold on its inputs and must ensure that the invariants hold when it's done.

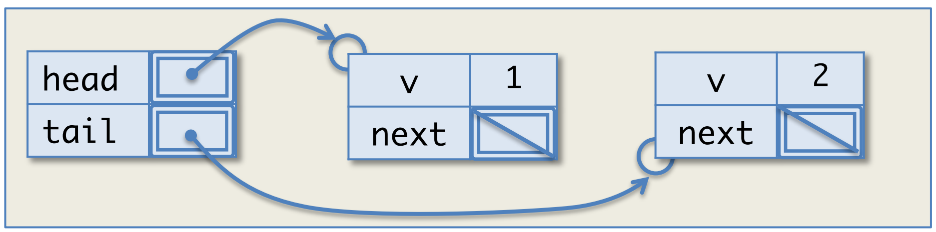
“Bogus” values of type `int queue`



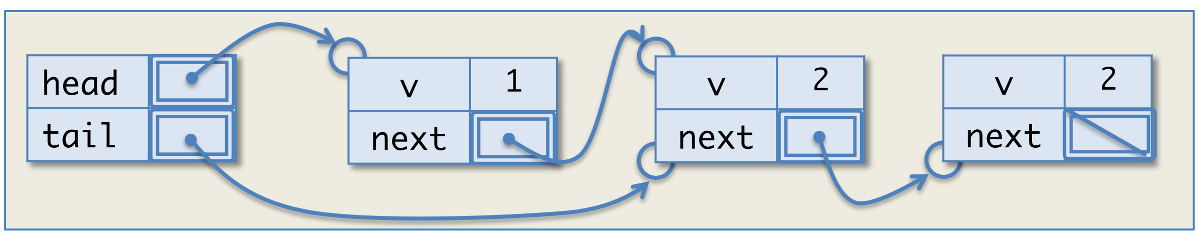
head is None, tail is Some



head is Some, tail is None

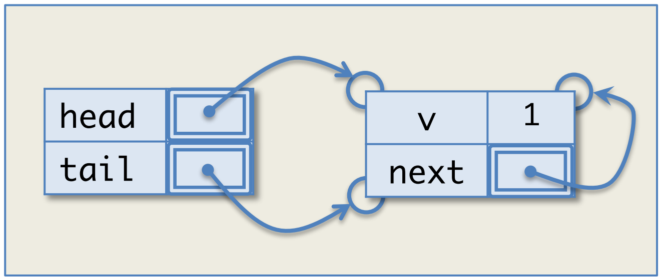


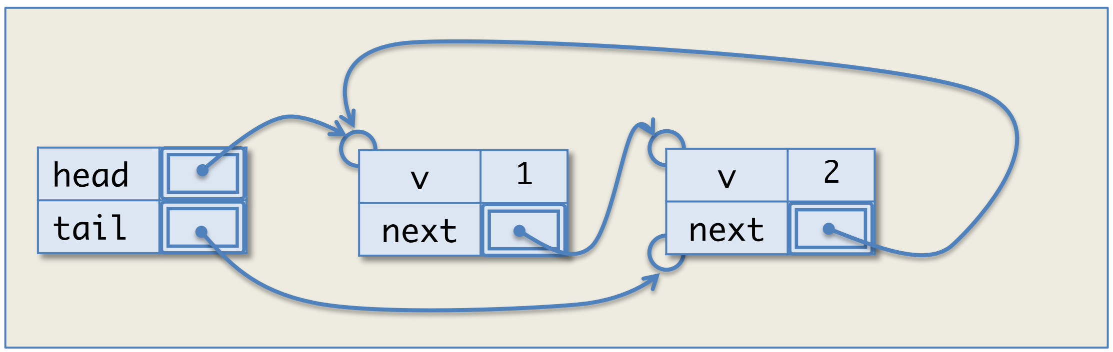
tail is not reachable from head



tail doesn’t point to the last element of the queue

"Bogus" cyclic `int` queue values





(And many, many others...)

Implementing Linked Queues

q.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
```

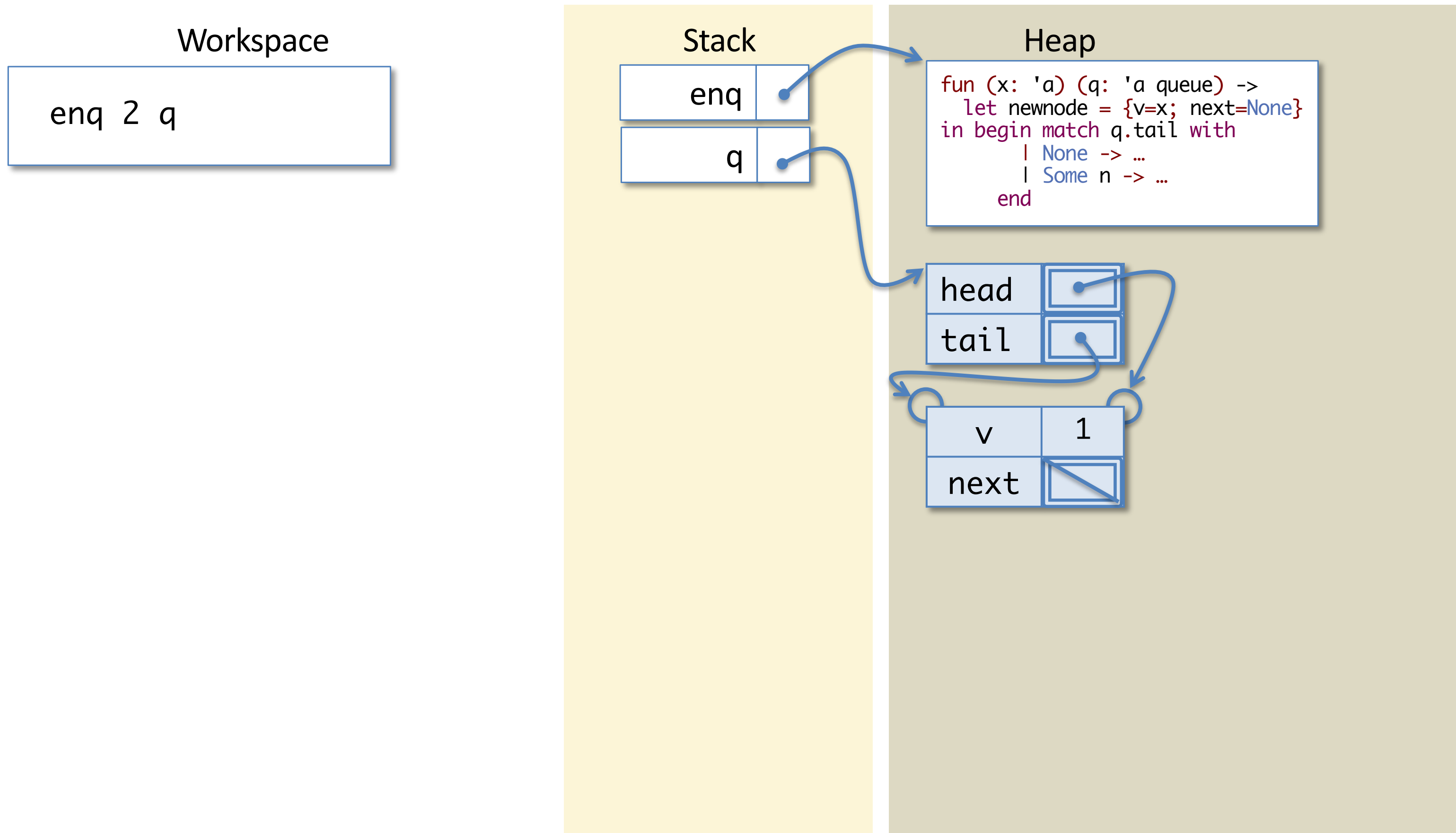
- `create` *establishes* the queue invariants
  - both head and `tail` are None
- `is_empty` *assumes* the queue invariants
  - it doesn't have to check that `q.tail` is None

enq

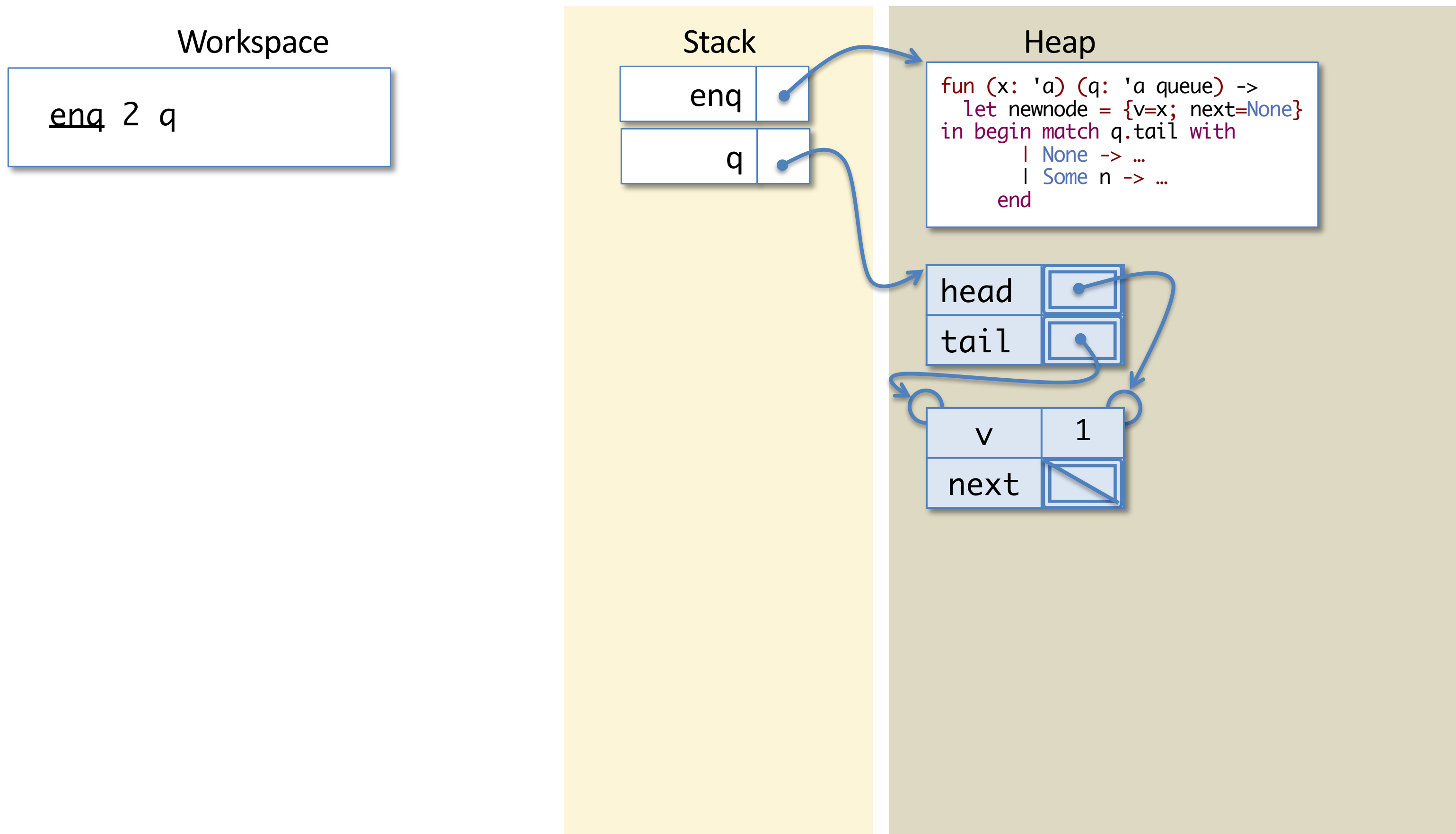
```
(* add an element to the tail of a queue *)
let enq (x: 'a) (q: 'a queue) : unit =
   let newnode = {v=x; next=None} in
   begin match q.tail with
     | None ->
        q.head <- Some newnode;
        q.tail <- Some newnode
     | Some n ->
        n.next <- Some newnode;
        q.tail <- Some 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 must “patch up” the “next” link of the old tail node to maintain the queue invariant.

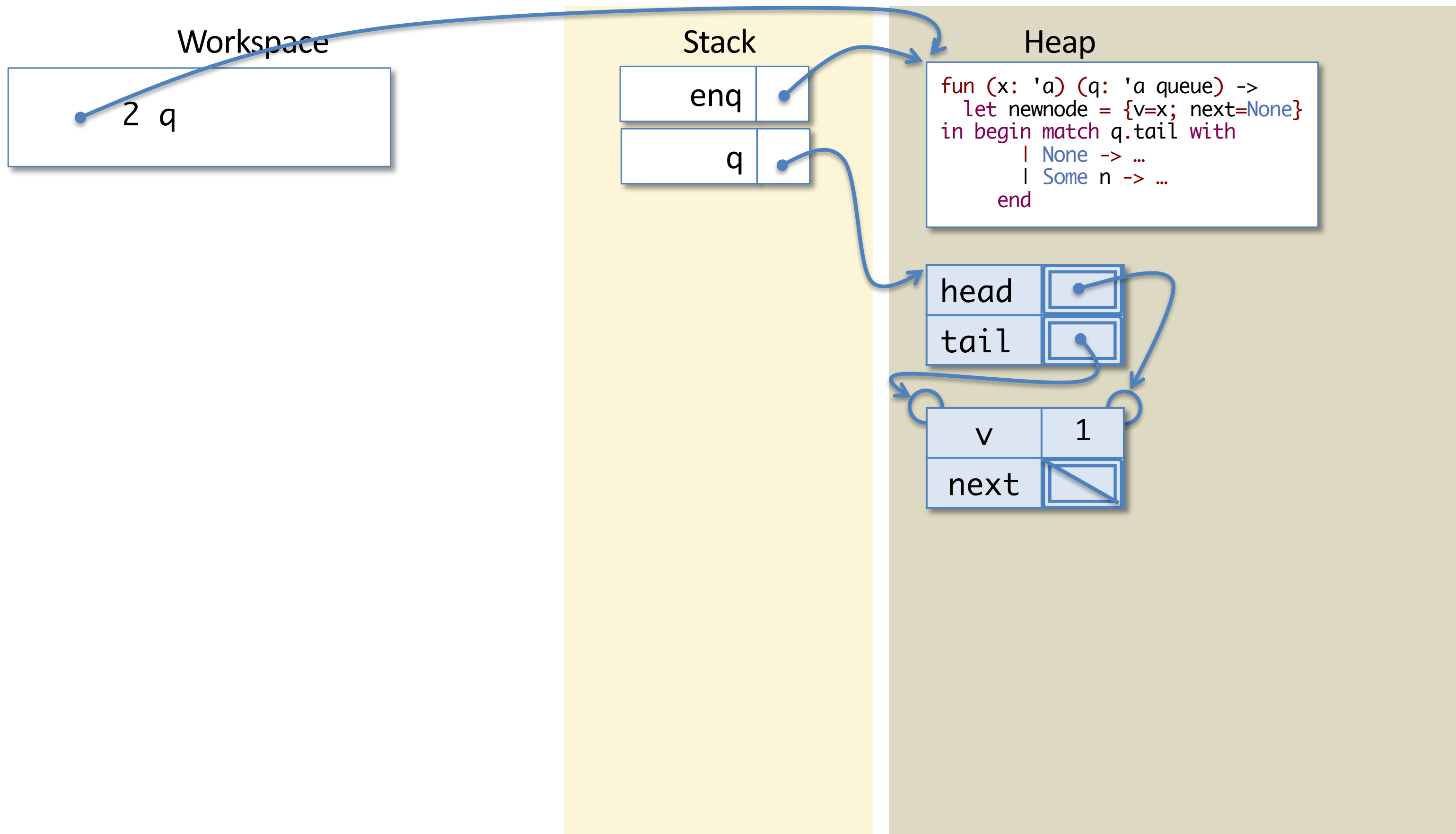
Calling Enq on a non-empty queue



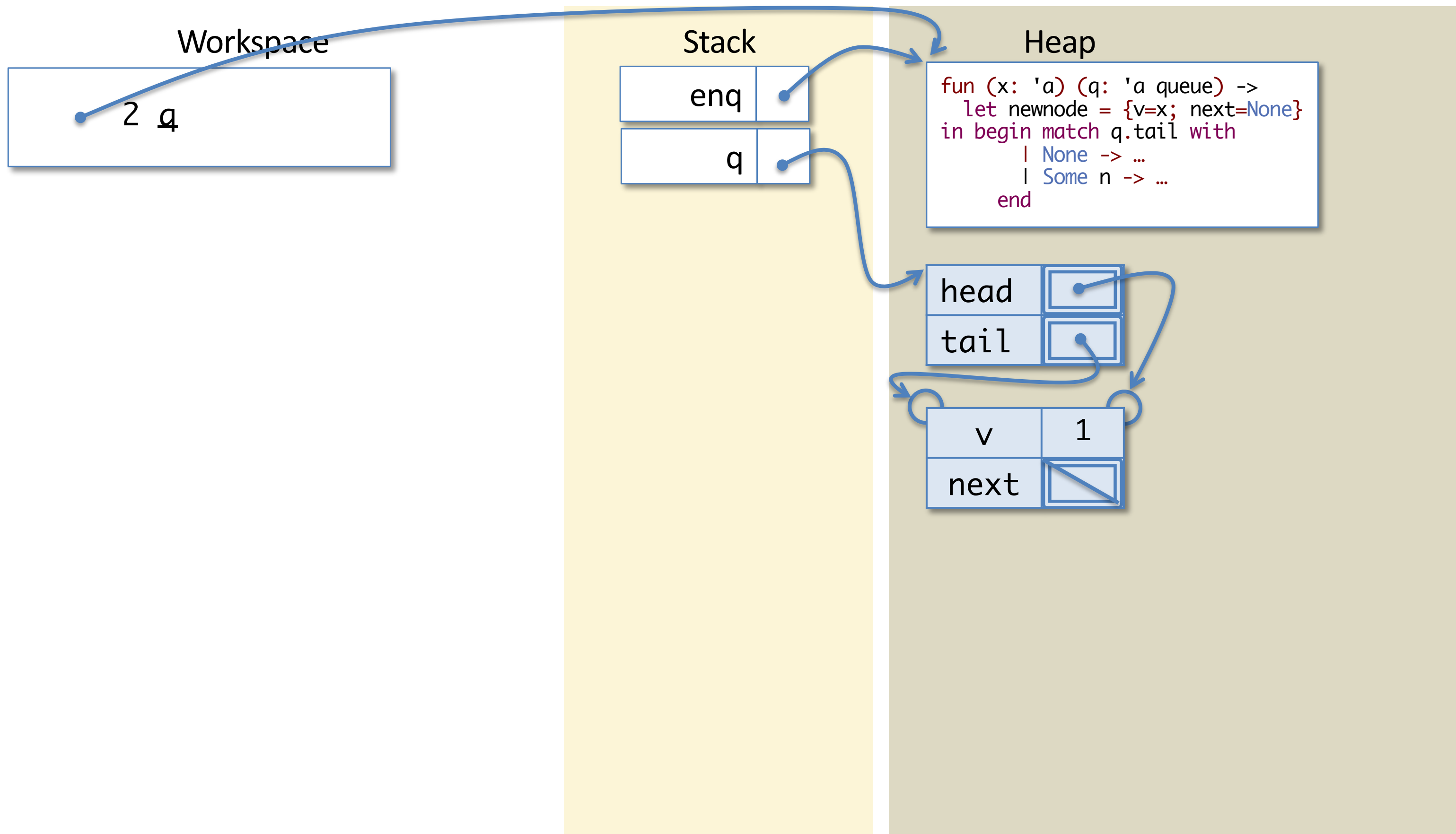
Calling Enq on a non-empty queue



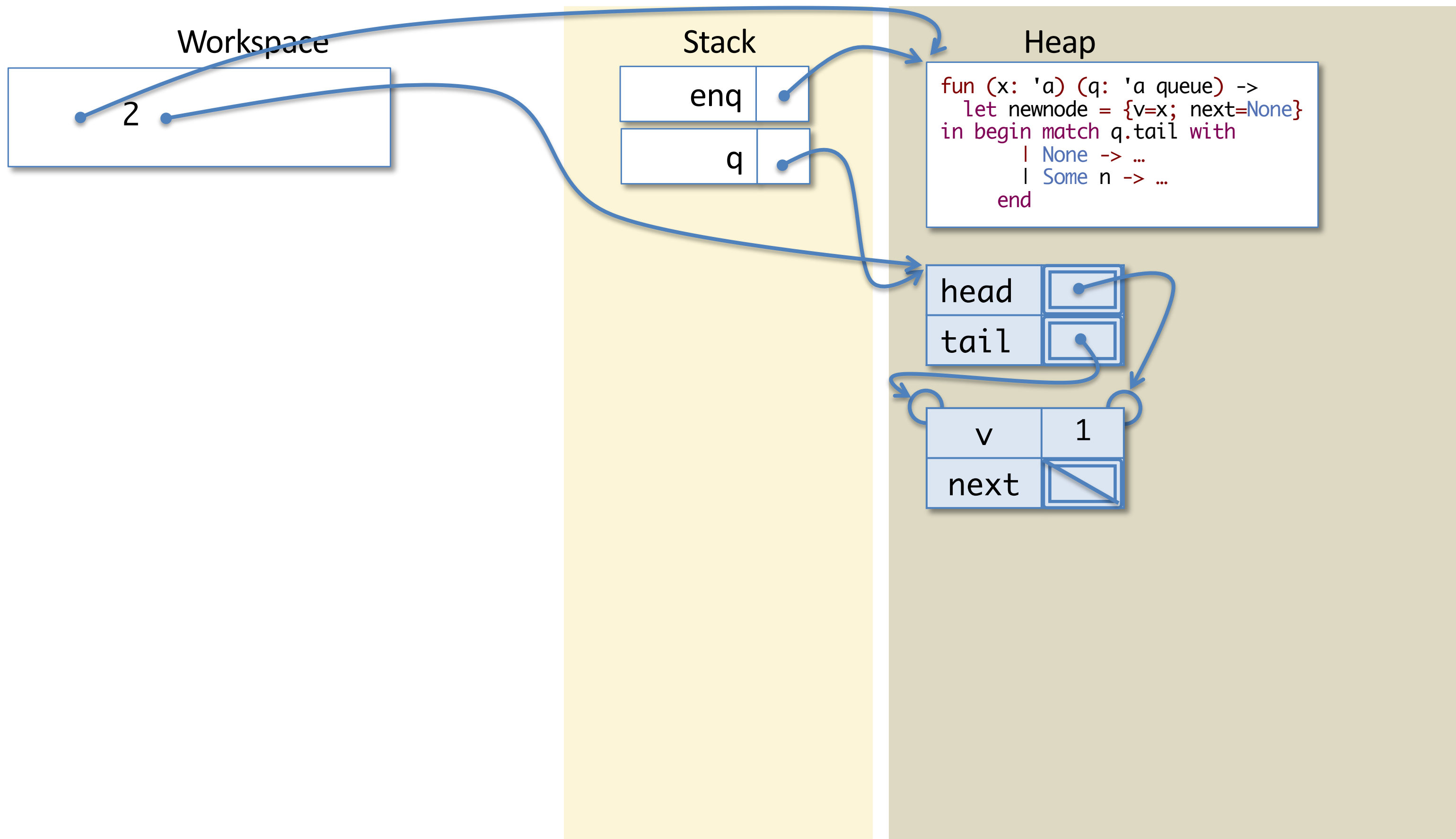
Calling Enq on a non-empty queue



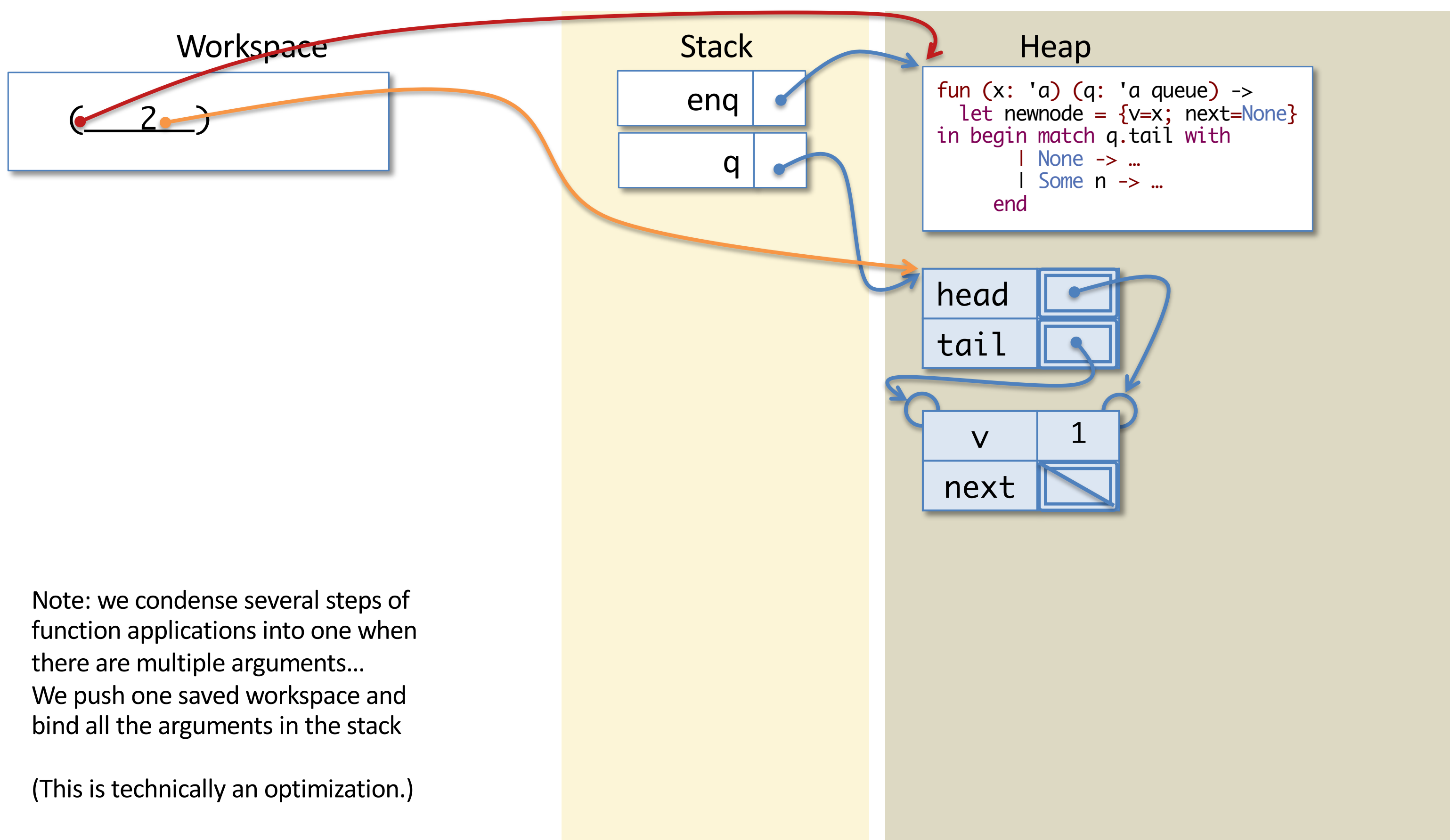
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue



Calling Enq on a non-empty queue



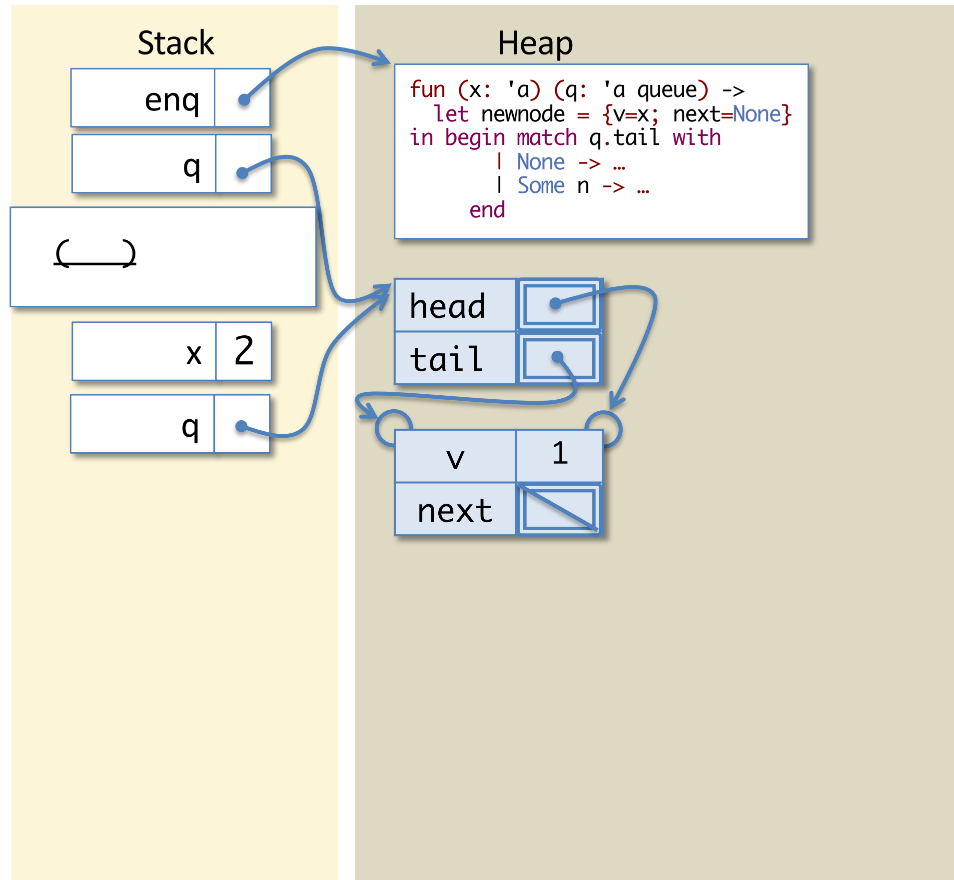
Note: we condense several steps of function applications into one when there are multiple arguments…
We push one saved workspace and bind all the arguments in the stack

(This is technically an optimization.)

Calling Enq on a non-empty queue

Workspace

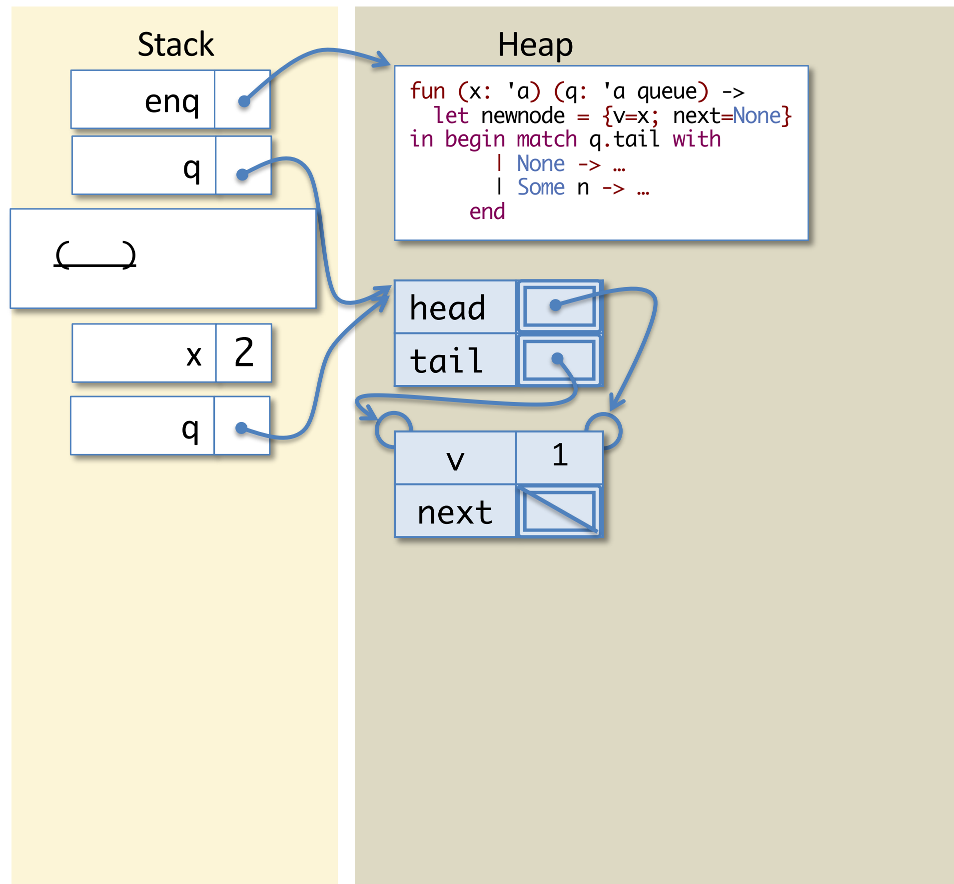
```
let newnode = {v=x; next=None} in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```



Calling Enq on a non-empty queue

Workspace

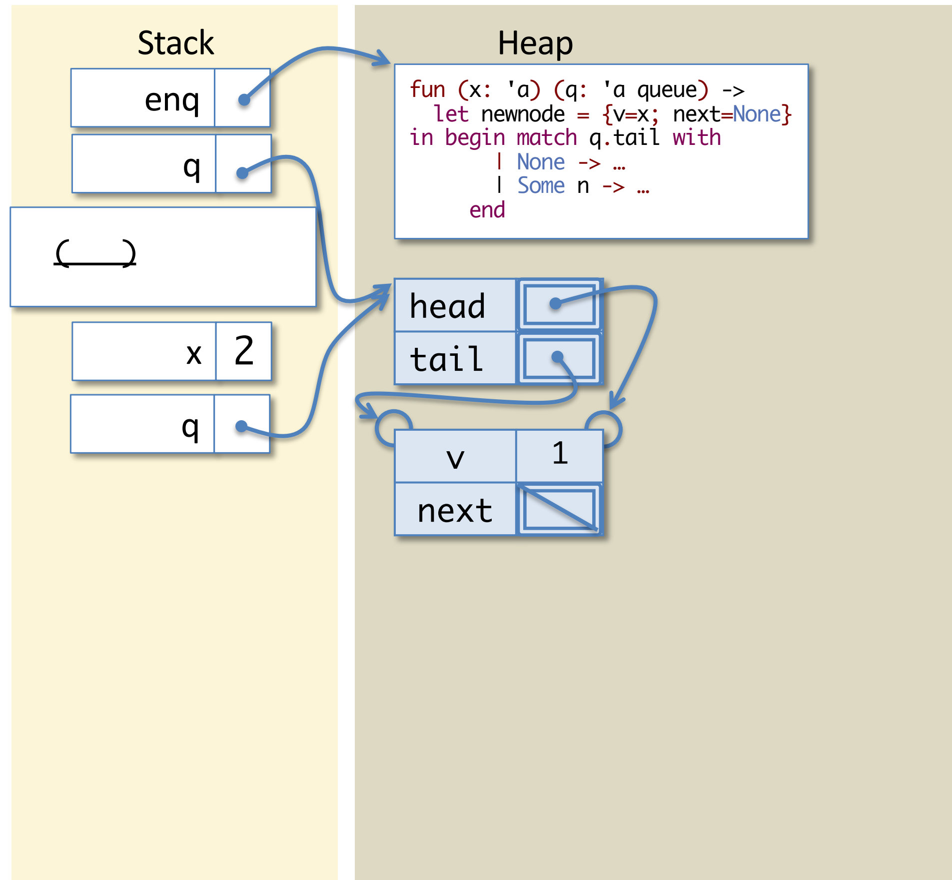
```
let newnode = {v=x; next=None} in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```



Calling Enq on a non-empty queue

Workspace

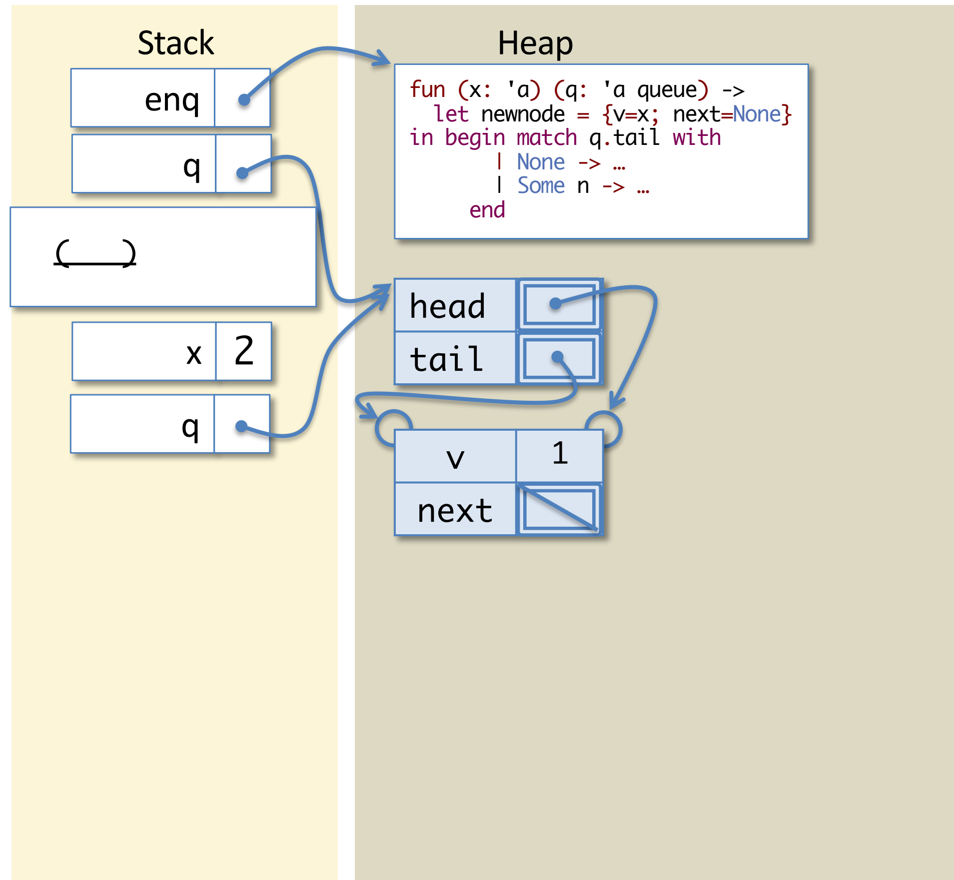
```
let newnode = {v=2; next=None} in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```



Calling Enq on a non-empty queue

Workspace

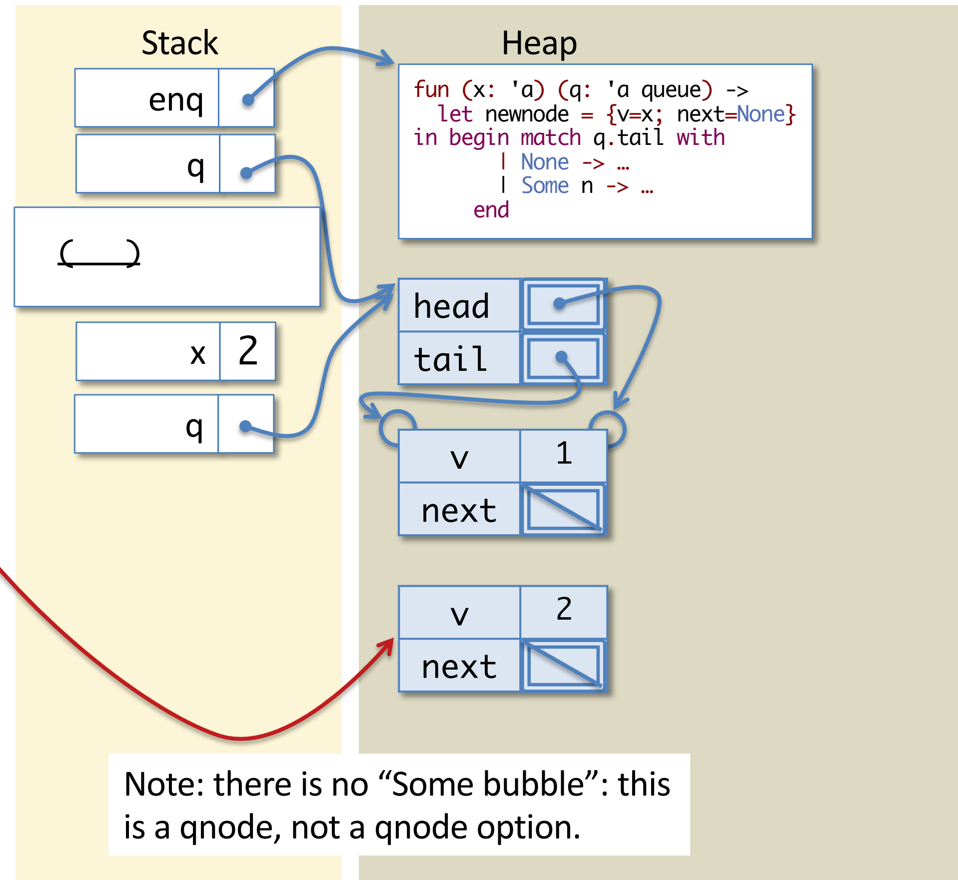
```
let newnode = {v=2; next=None} in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```



Calling Enq on a non-empty queue

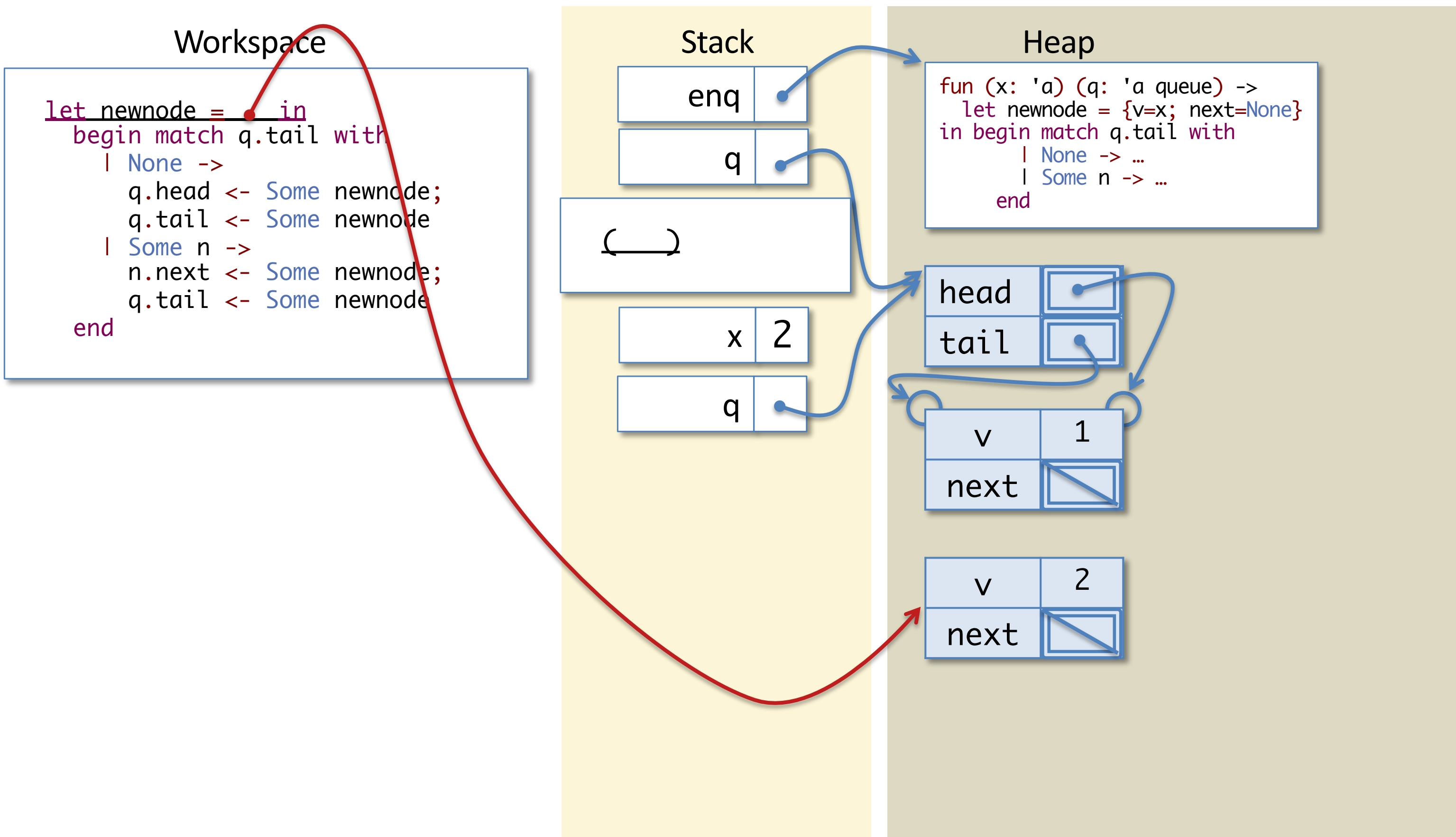
Workspace

```
let newnode = • 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
```



Note: there is no "Some bubble": this is a qnode, not a qnode option.

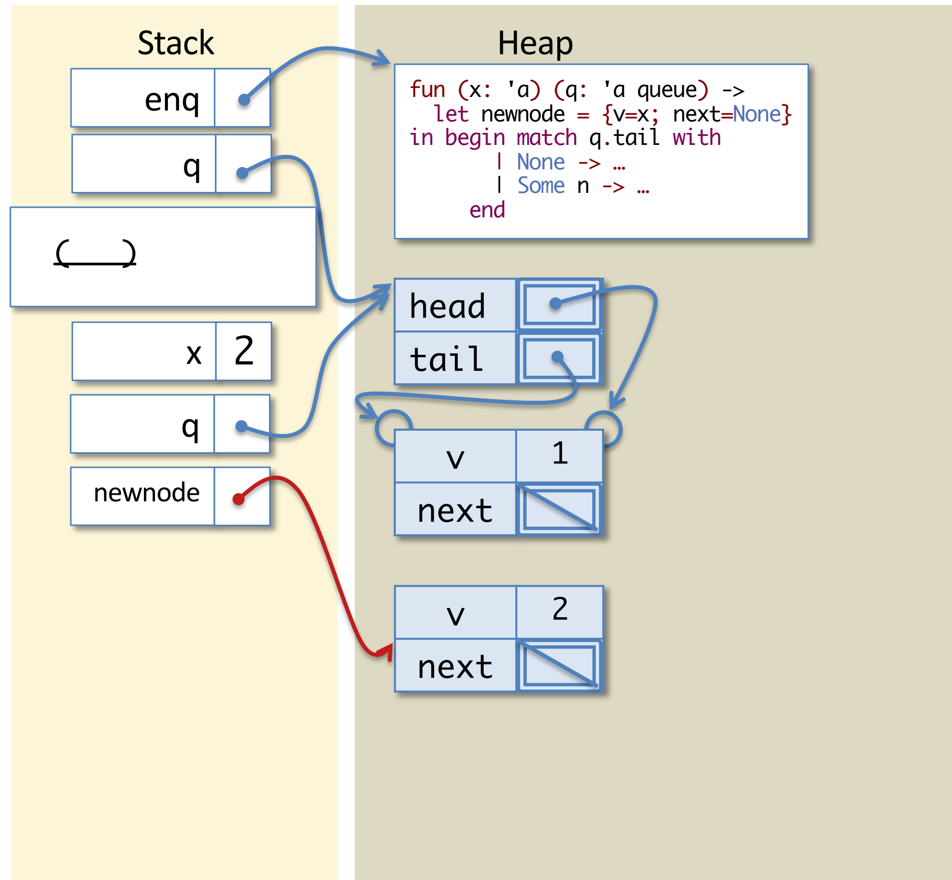
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue

Workspace

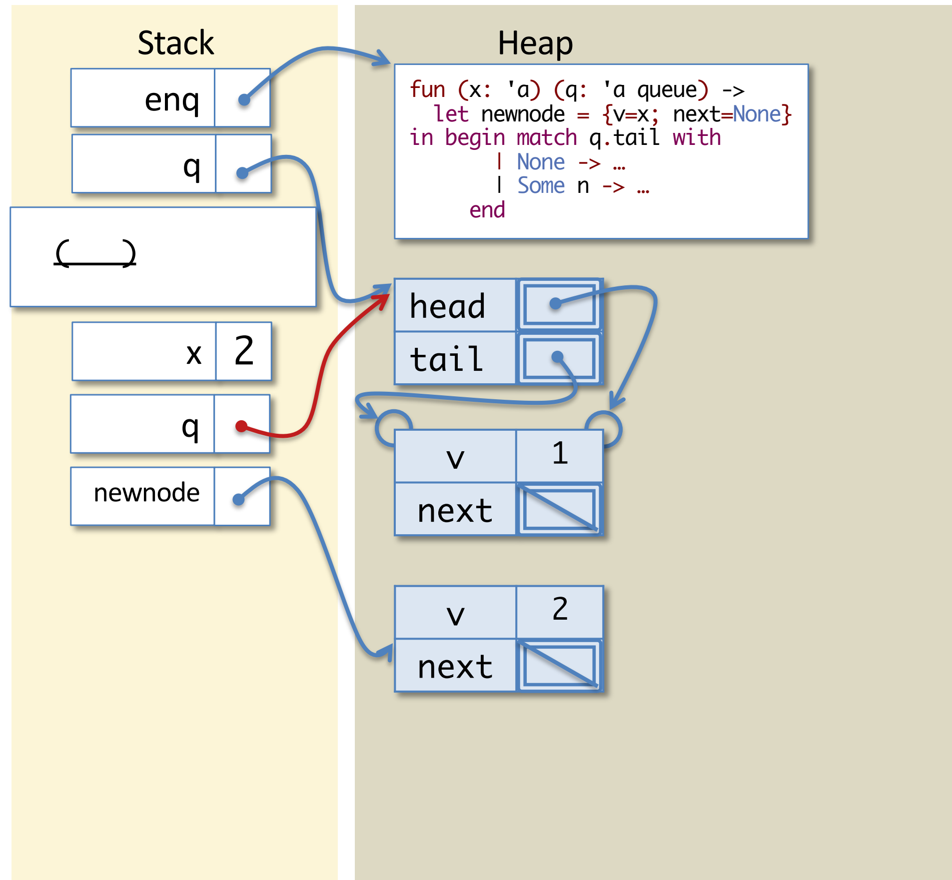
```
begin match q.tail with
  | None ->
    q.head <- Some newnode;
    q.tail <- Some newnode
  | Some n ->
    n.next <- Some newnode;
    q.tail <- Some newnode
end
```



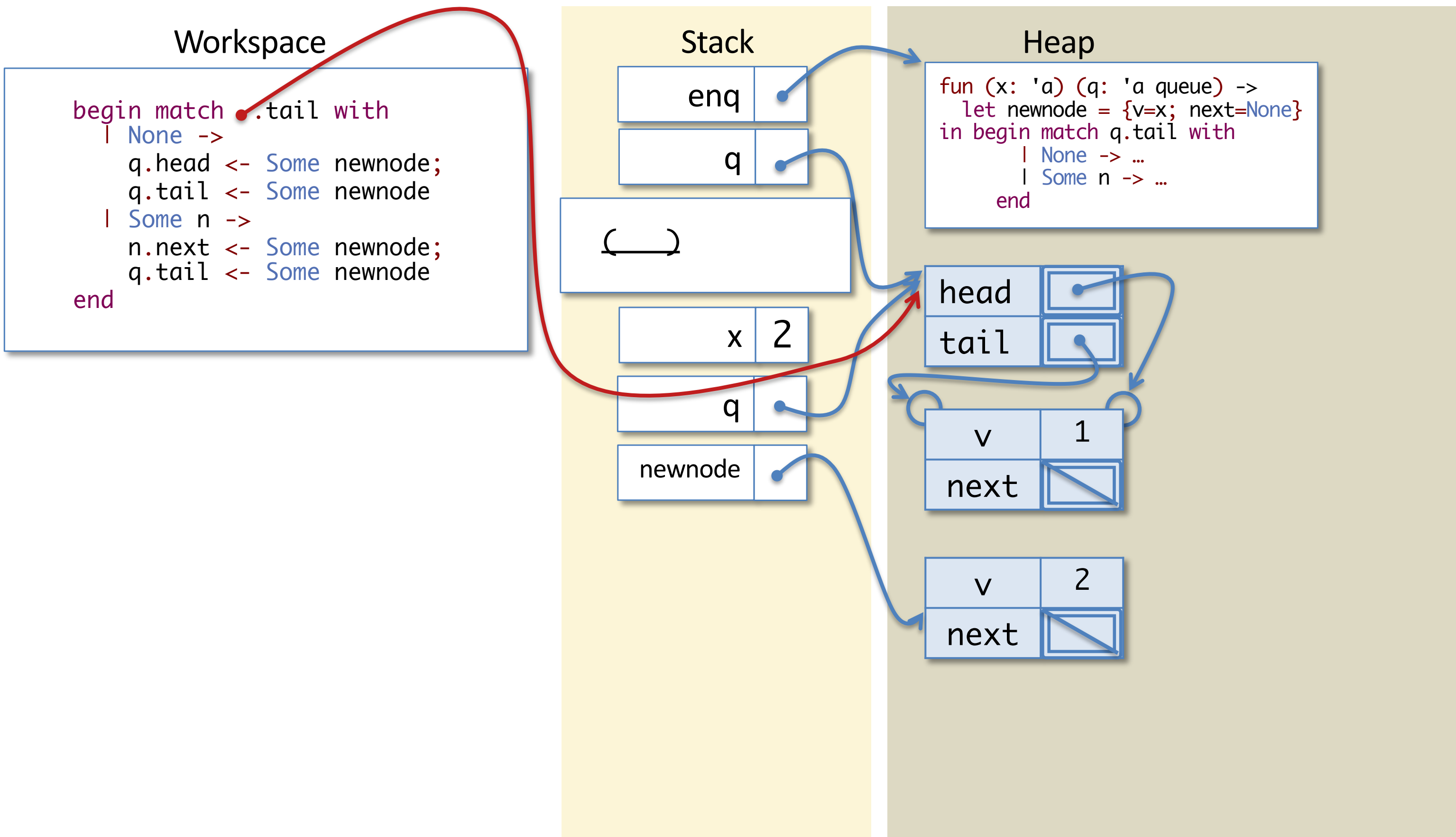
Calling Enq on a non-empty queue

Workspace

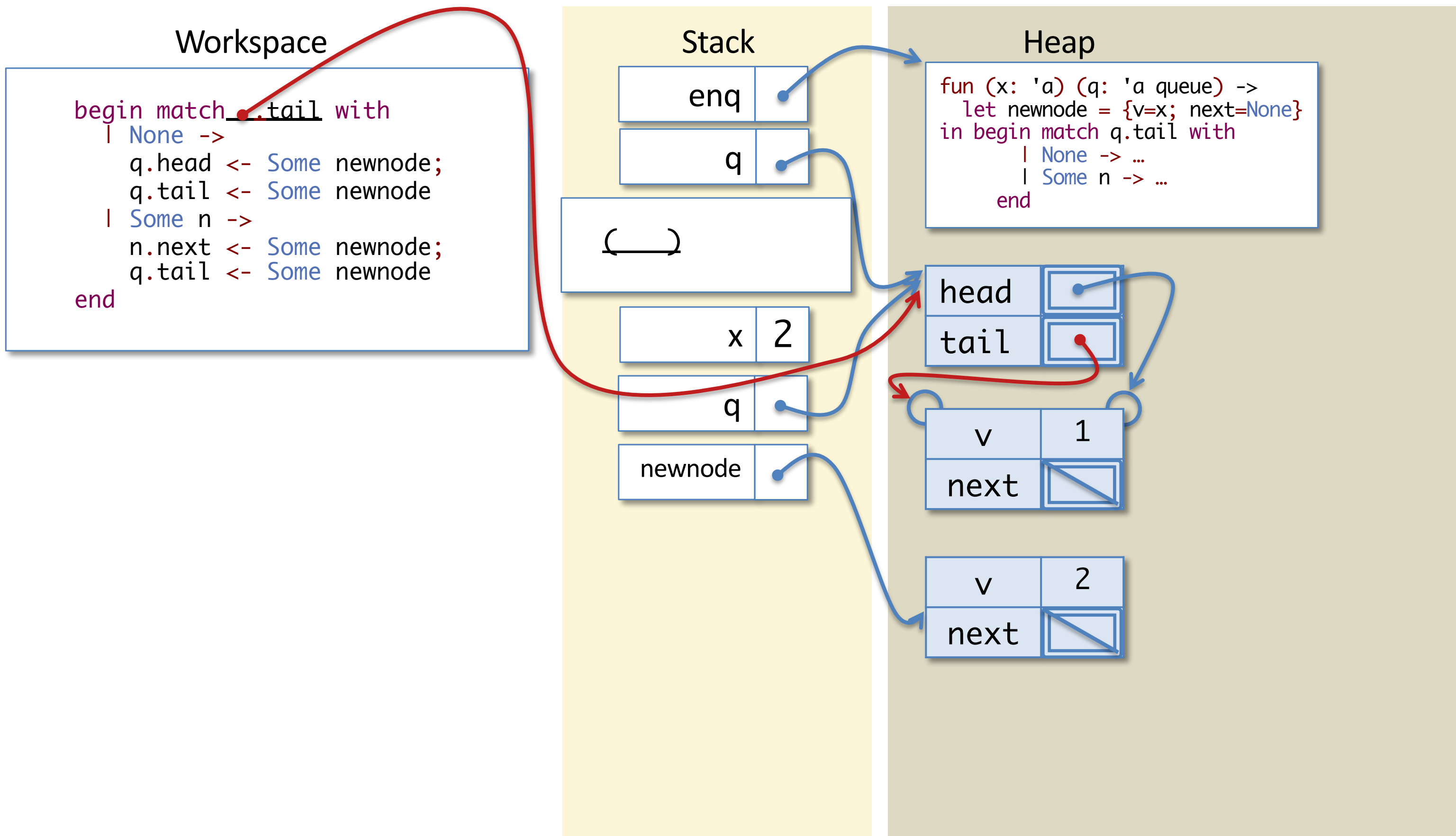
```
begin match q.tail with
  | None ->
    q.head <- Some newnode;
    q.tail <- Some newnode
  | Some n ->
    n.next <- Some newnode;
    q.tail <- Some newnode
end
```



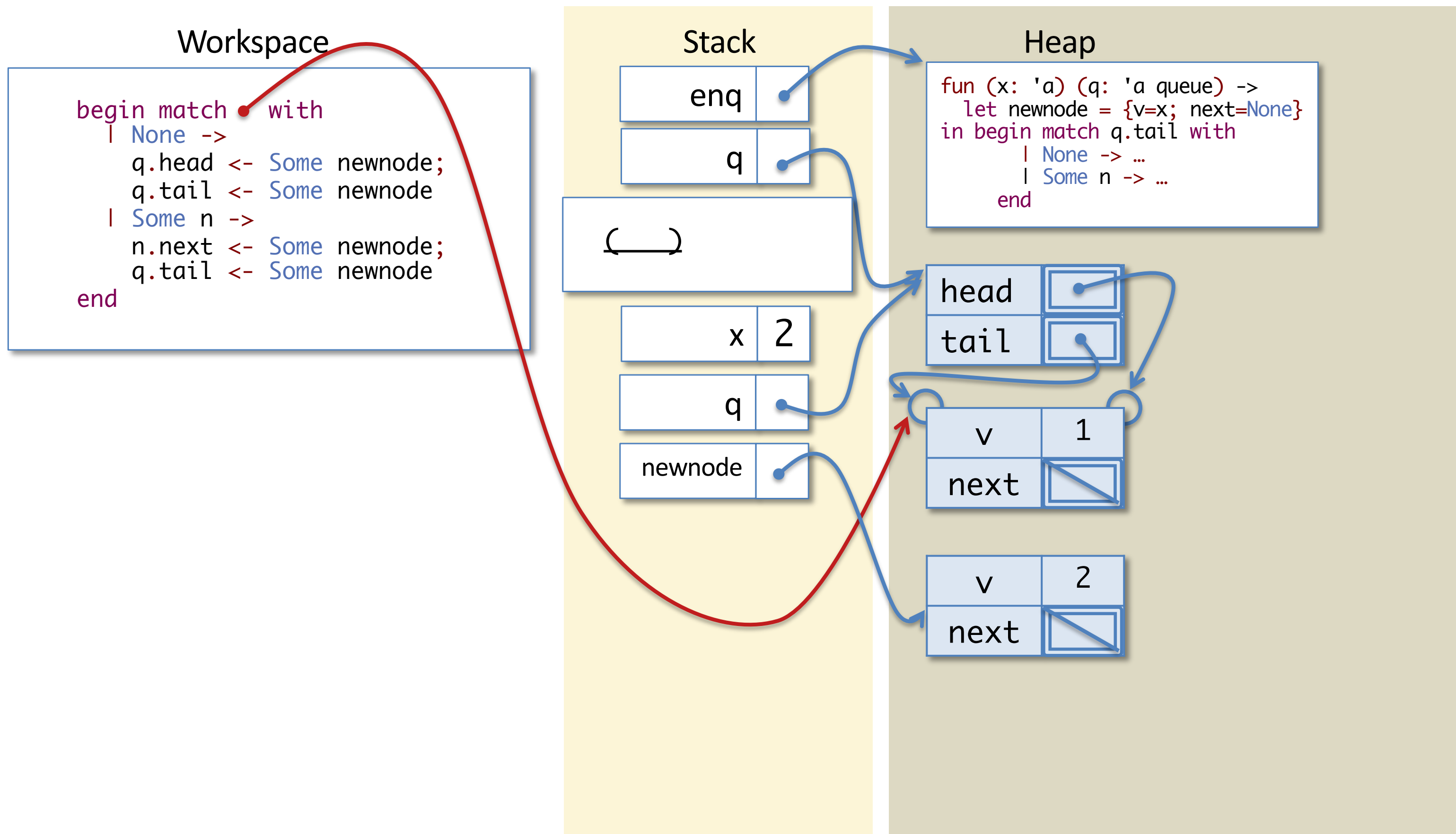
Calling Enq on a non-empty queue



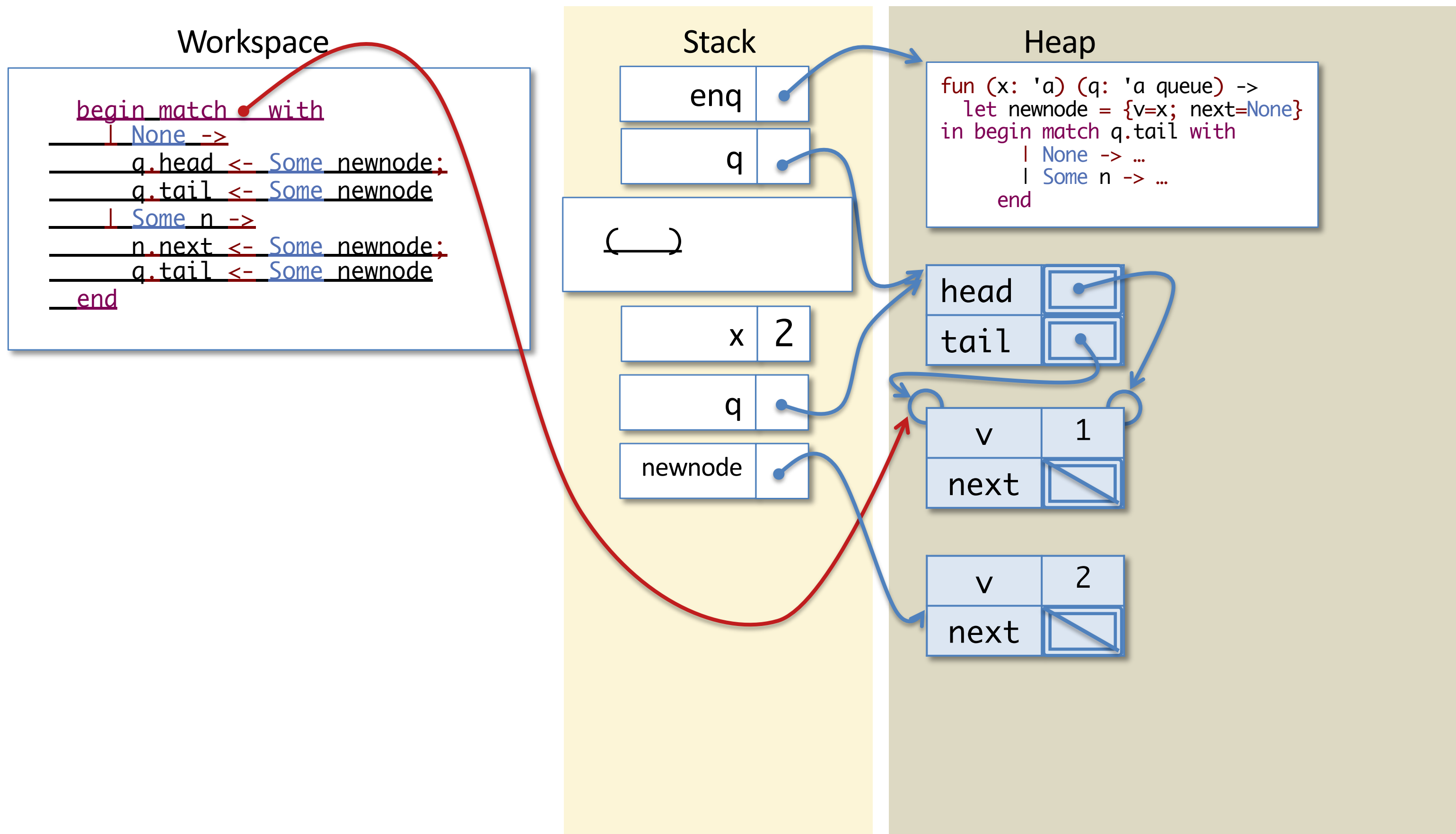
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue



Calling Enq on a non-empty queue



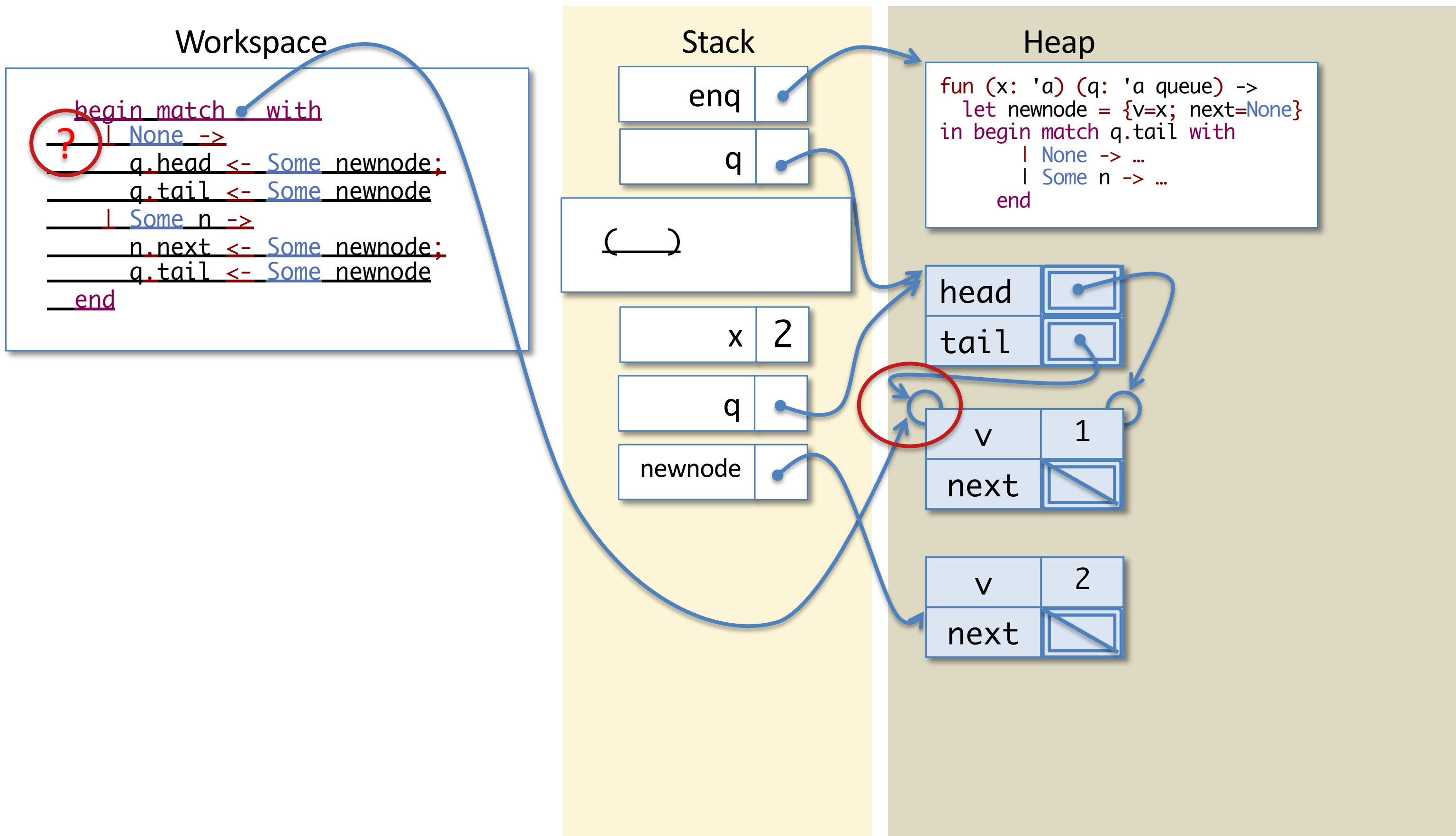
Simplifying Match

- A match expression

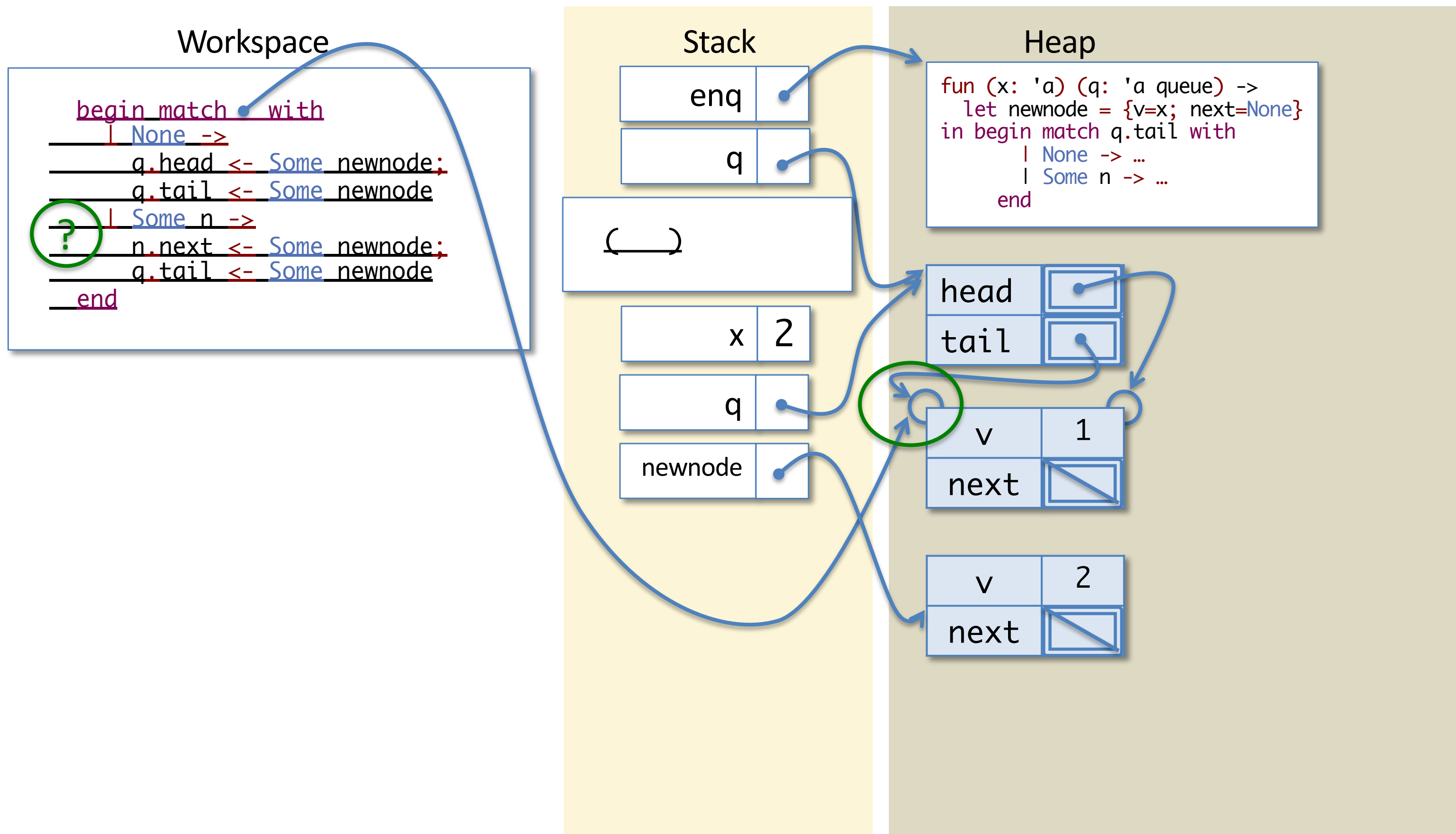
```
begin match e with
 | pat₁ -> branch₁
 | …
 | patₙ -> branchₙ
end
```

  is ready if e is a value
  - Note that e will always be a pointer to a constructor cell in the heap
  - This expression is simplified by finding the first pattern $\texttt{pat}_i$ that matches the cell and adding new bindings for the pattern variables (to the parts of `e` that line up) to the end of the stack
  - replacing the whole match expression in the workspace with the corresponding $\texttt{branch}_i$

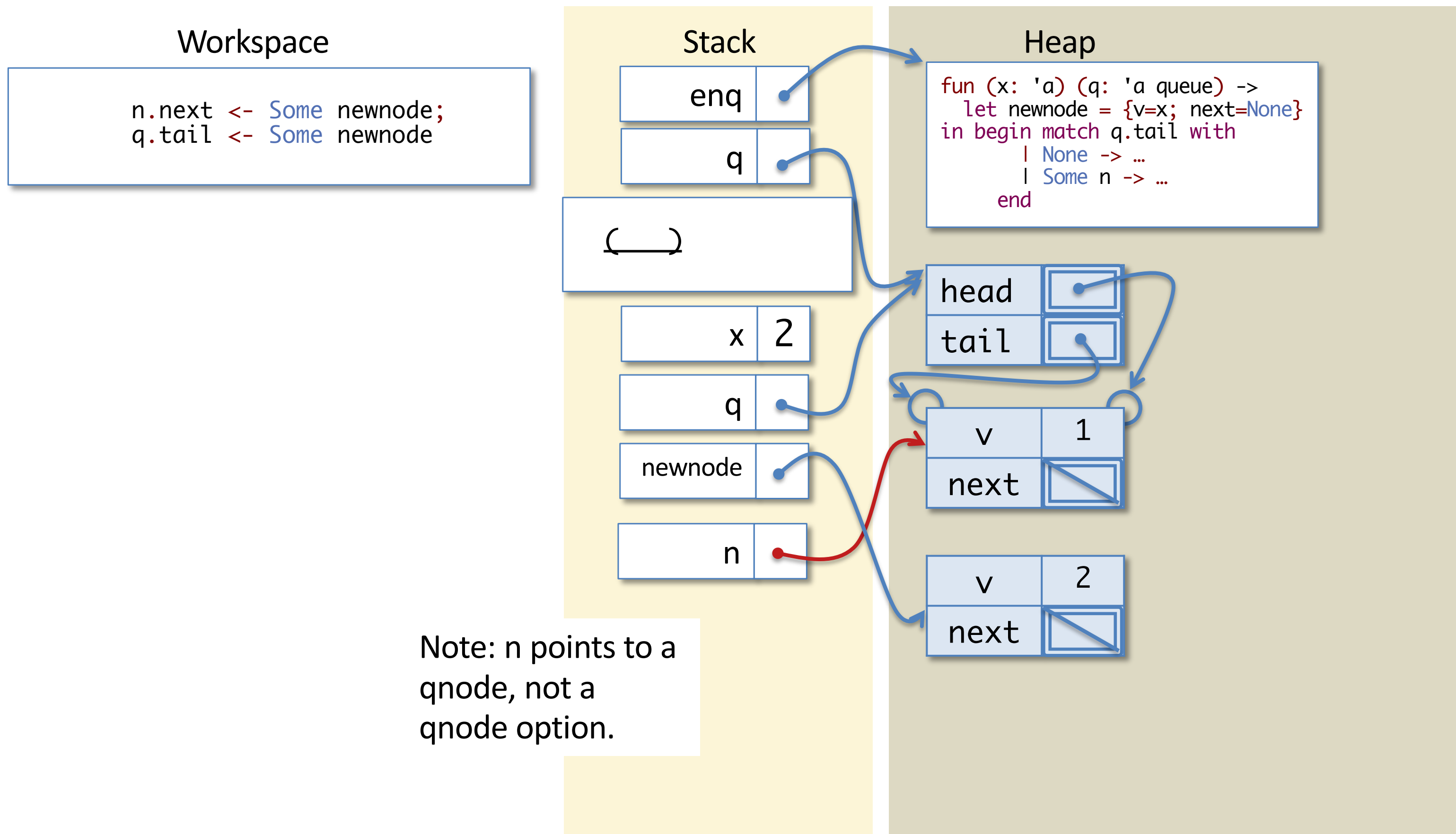
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue



Calling Enq on a non-empty queue

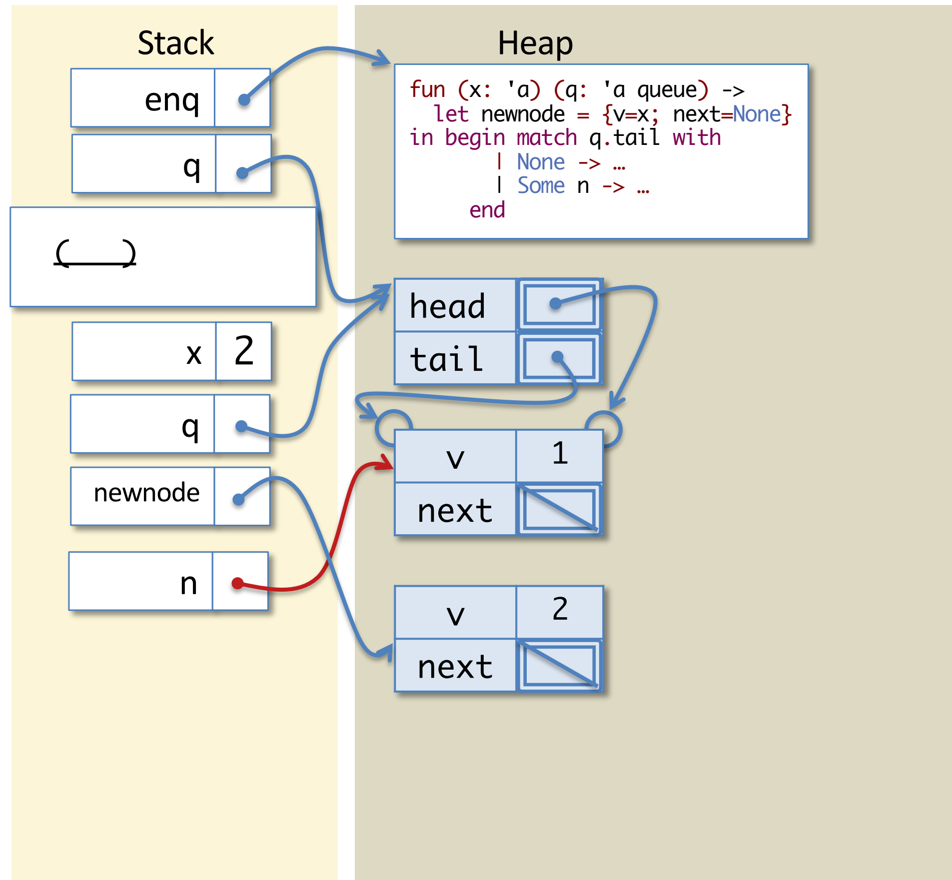


Note: n points to a qnode, not a qnode option.

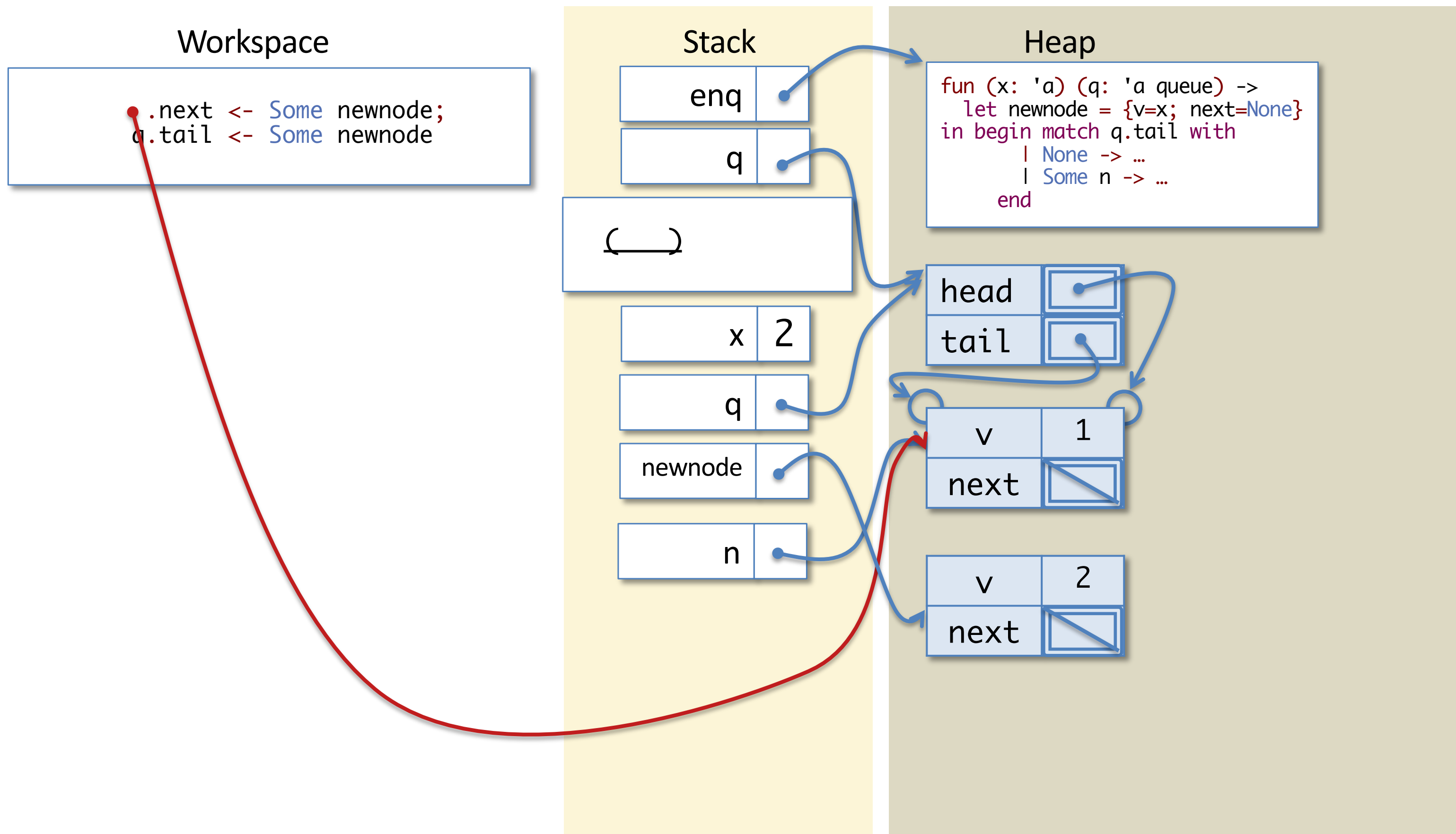
Calling Enq on a non-empty queue

Workspace

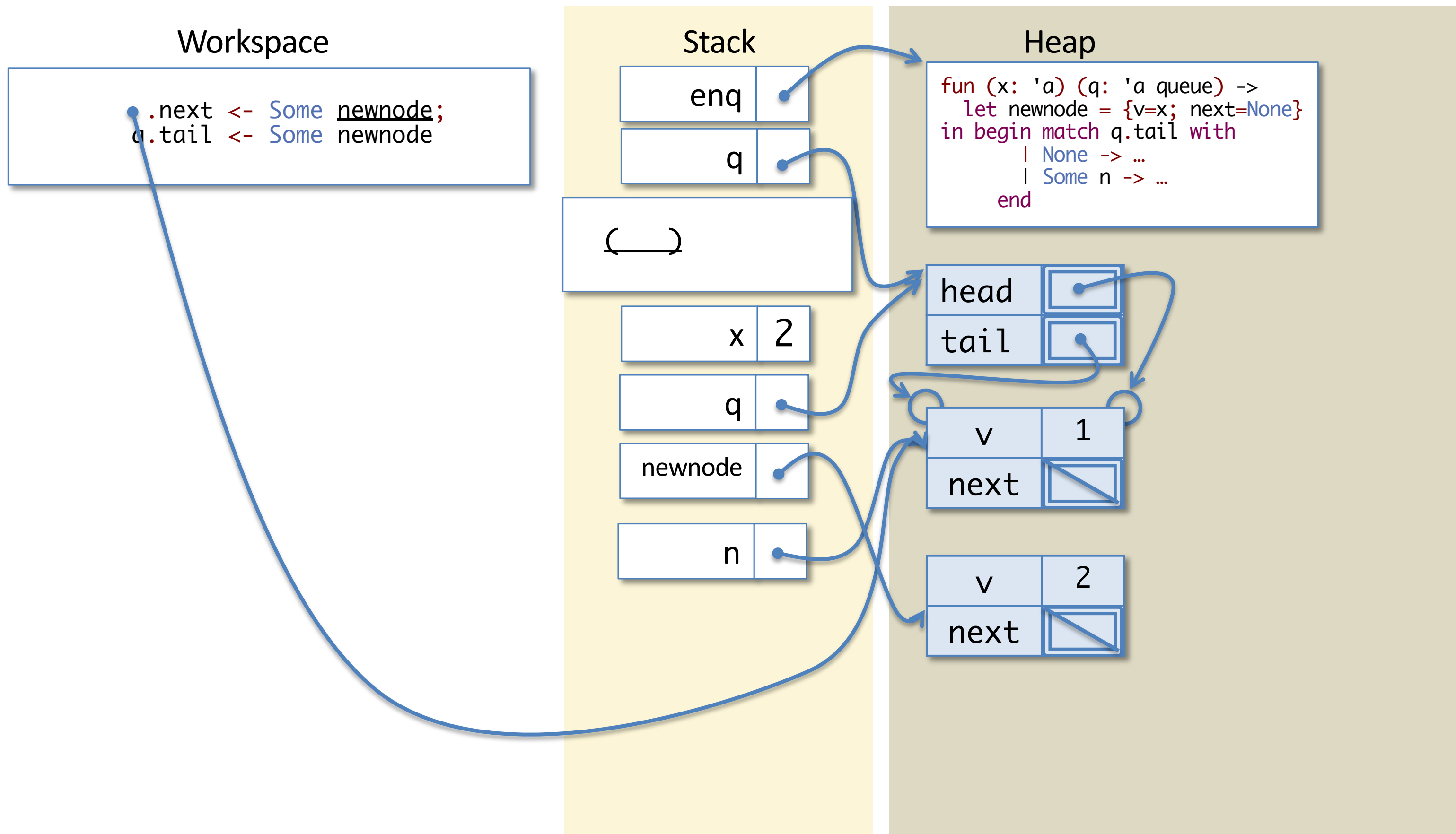
```
n.next <- Some newnode;
q.tail <- Some newnode
```



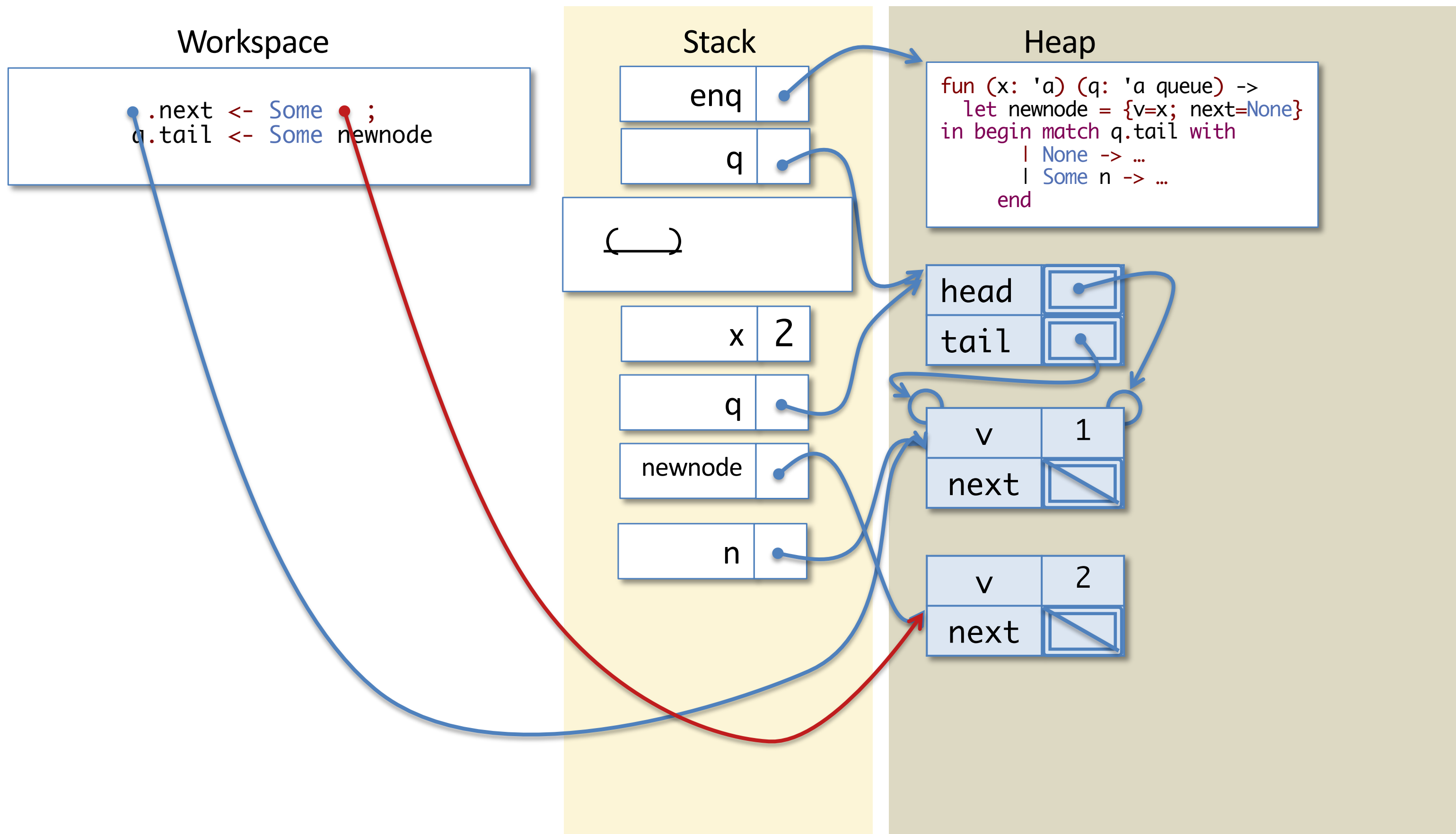
Calling Enq on a non-empty queue



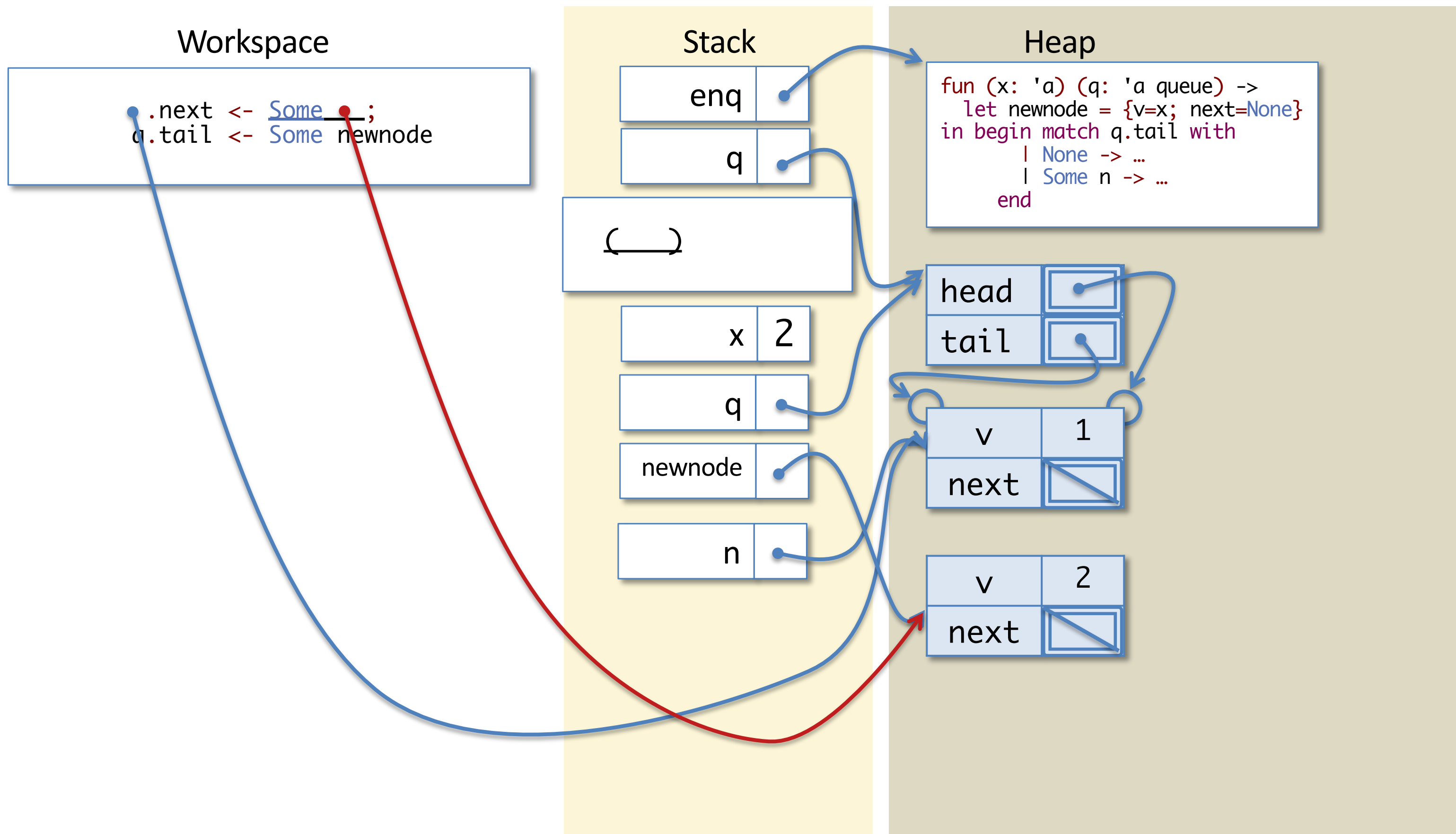
Calling Enq on a non-empty queue



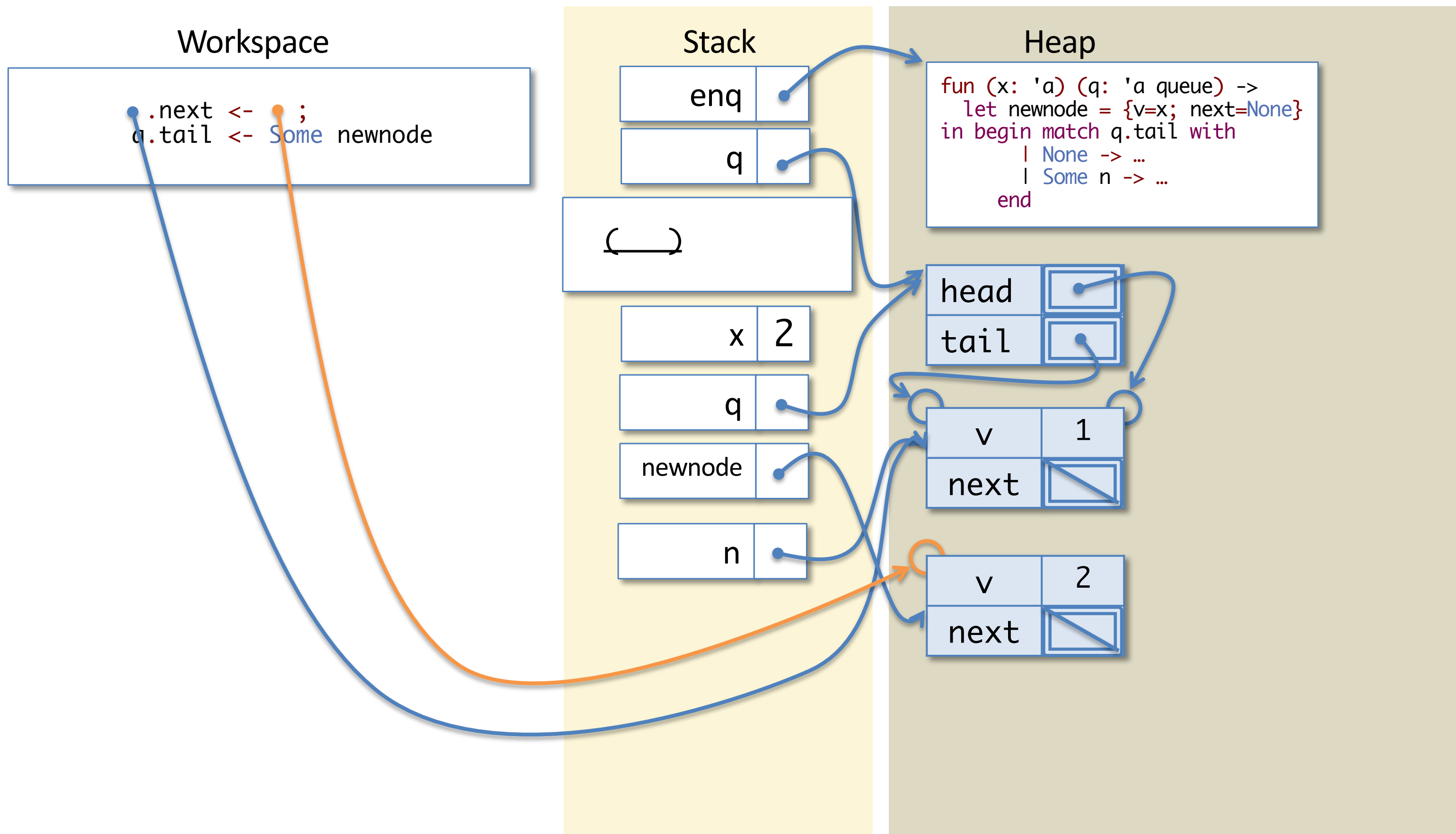
Calling Enq on a non-empty queue



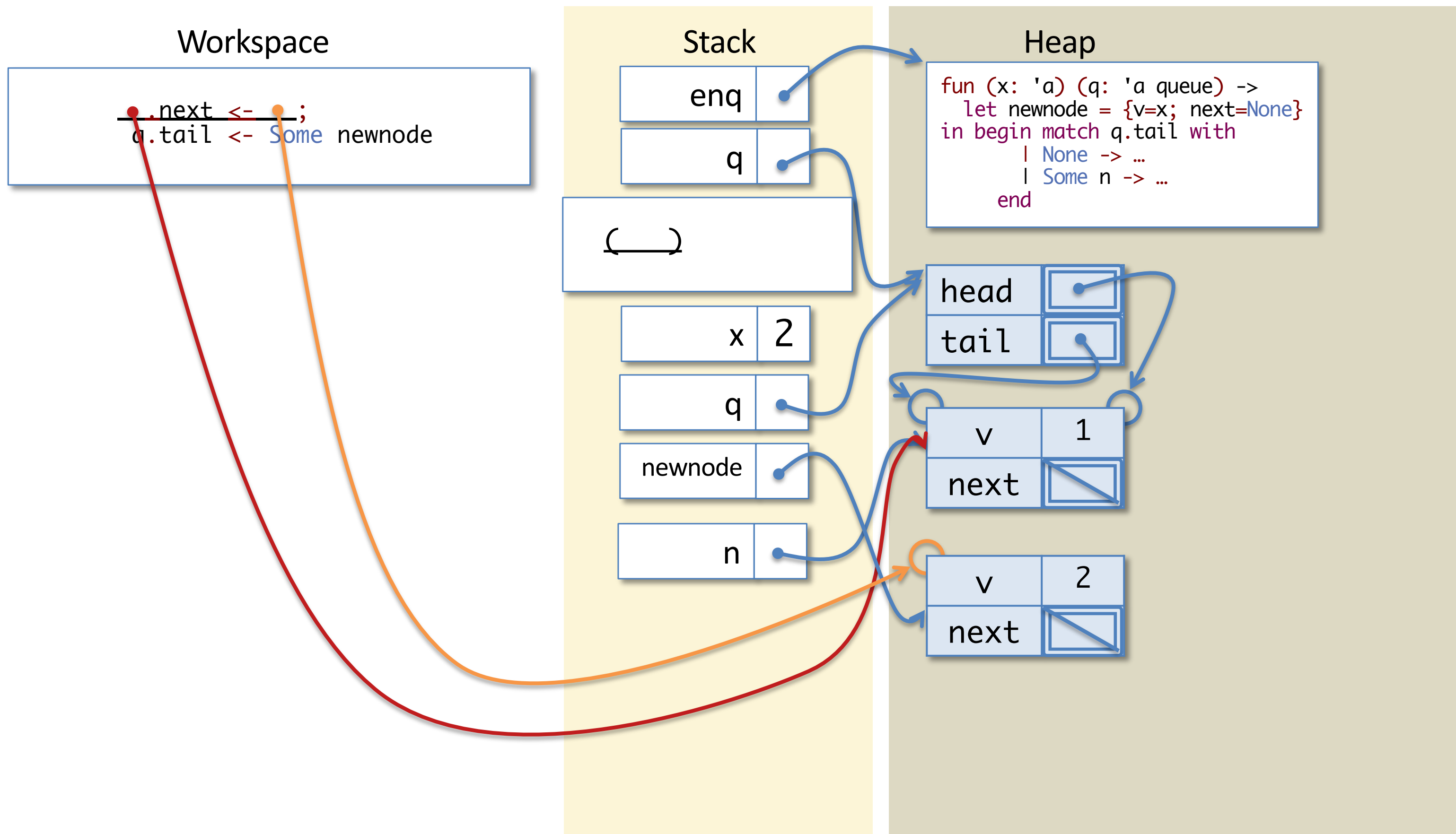
Calling Enq on a non-empty queue



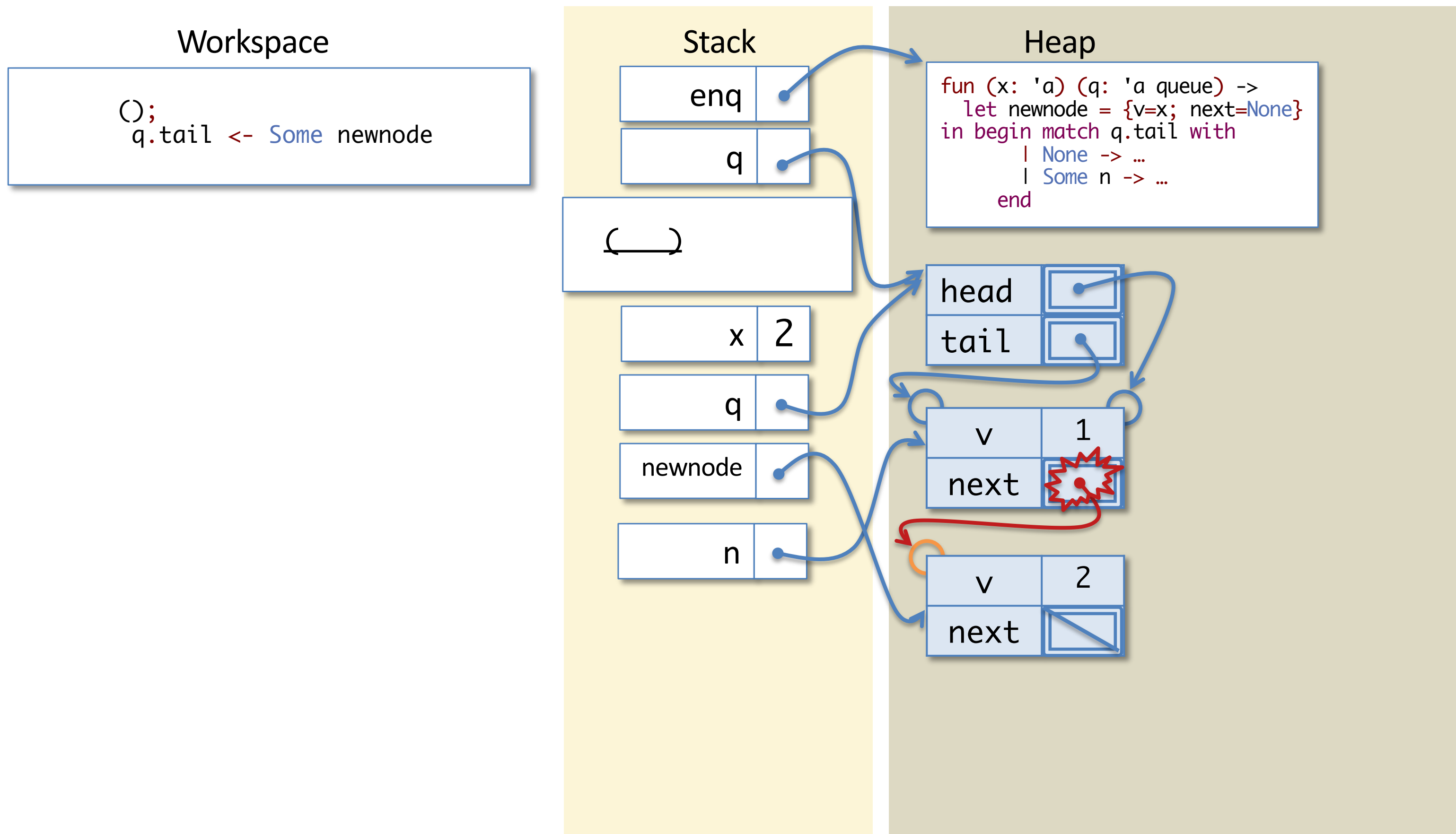
Calling Enq on a non-empty queue



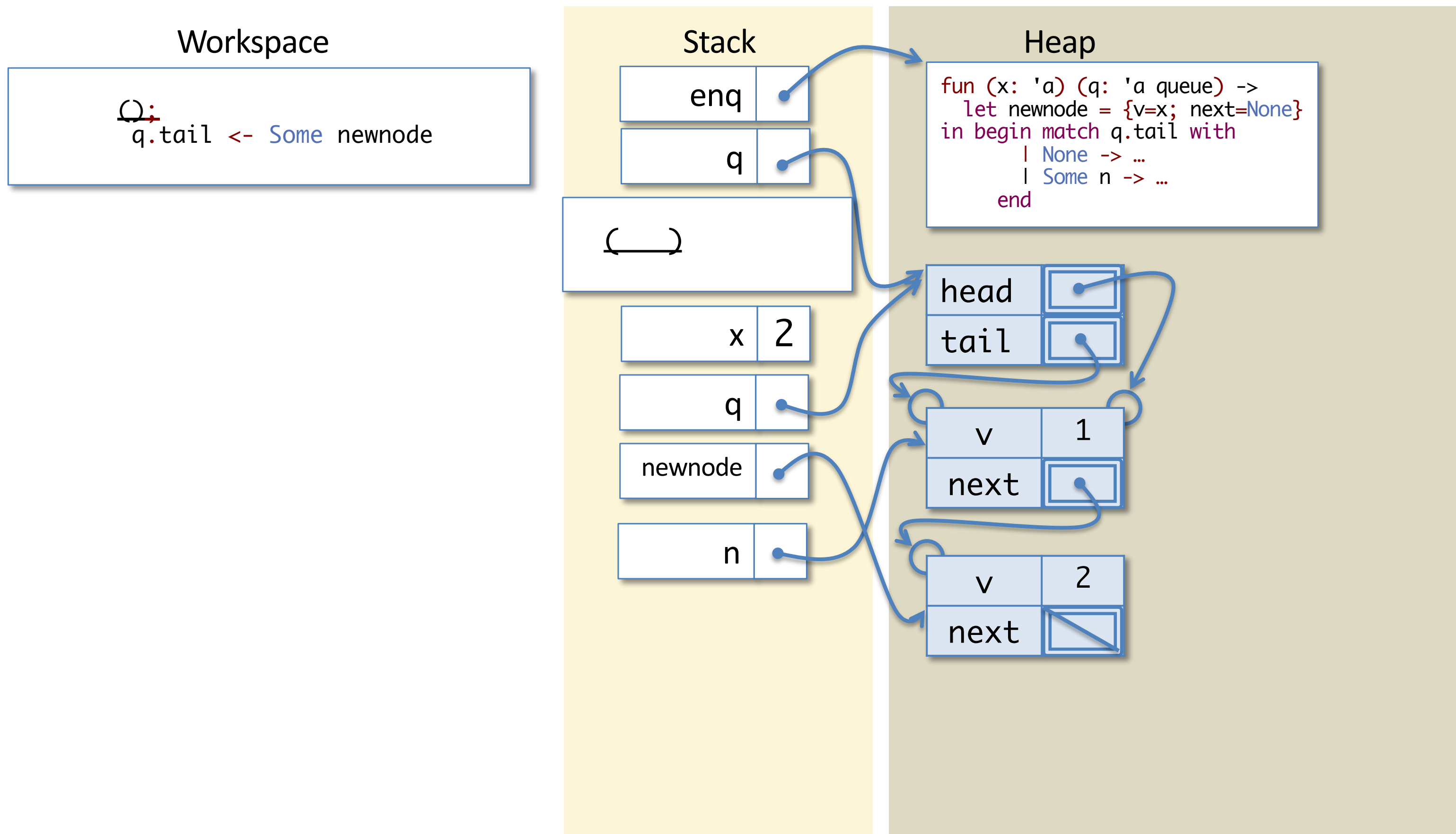
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue



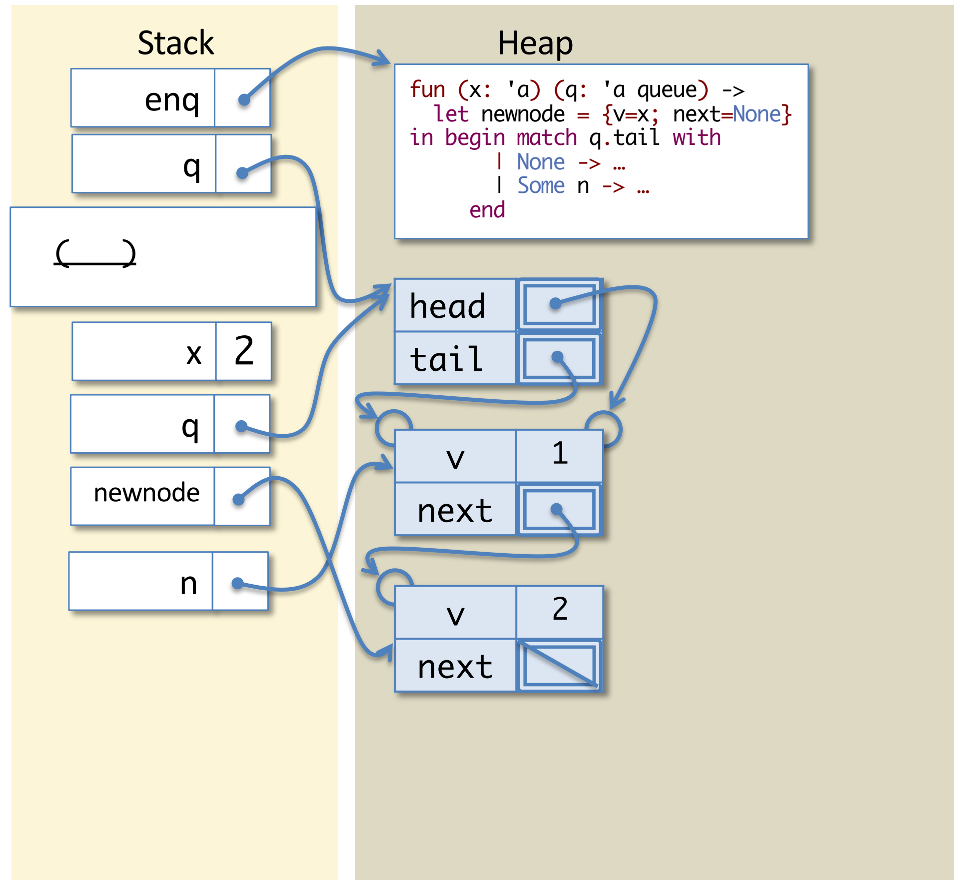
Calling Enq on a non-empty queue



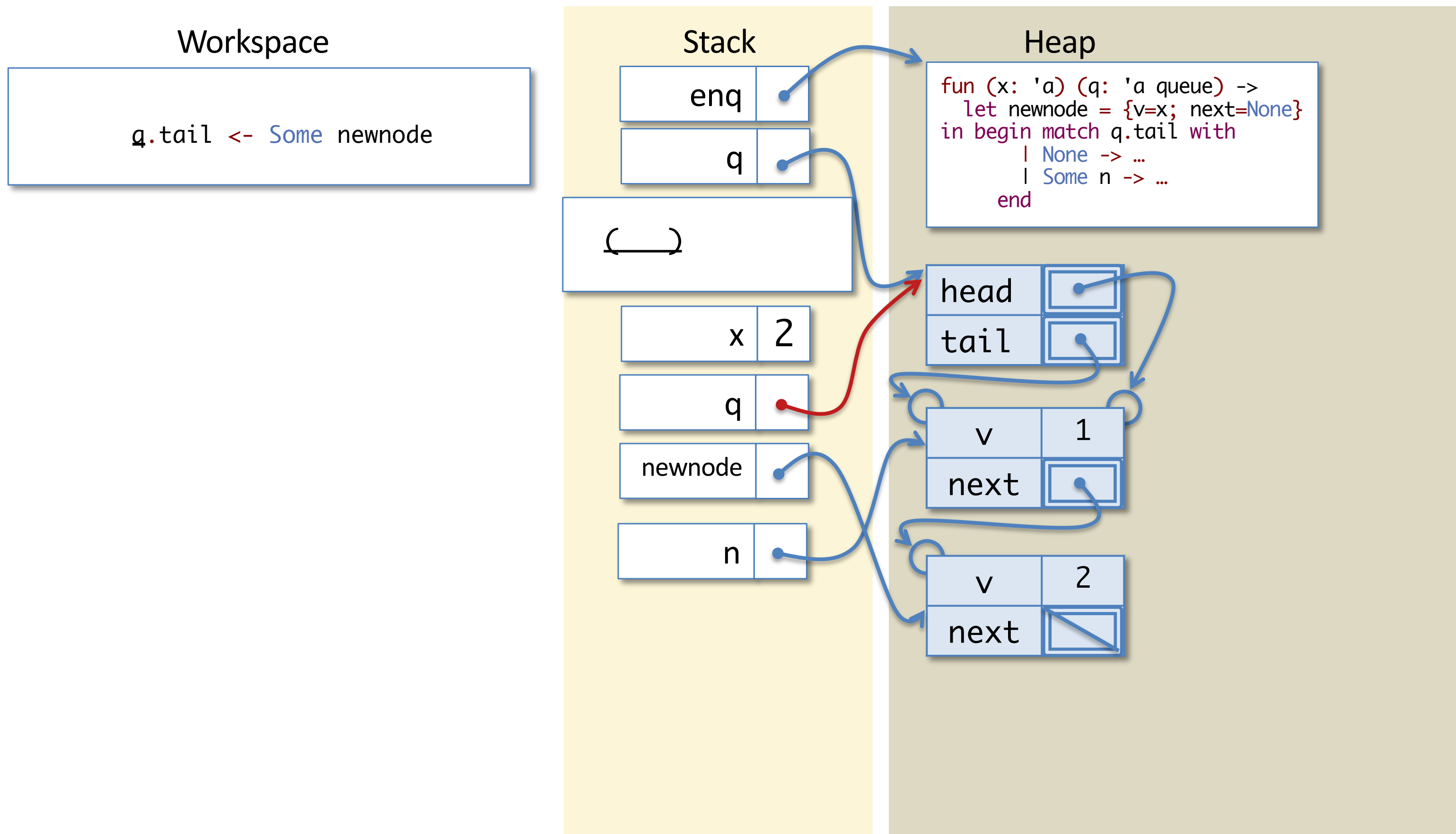
Calling Enq on a non-empty queue

Workspace

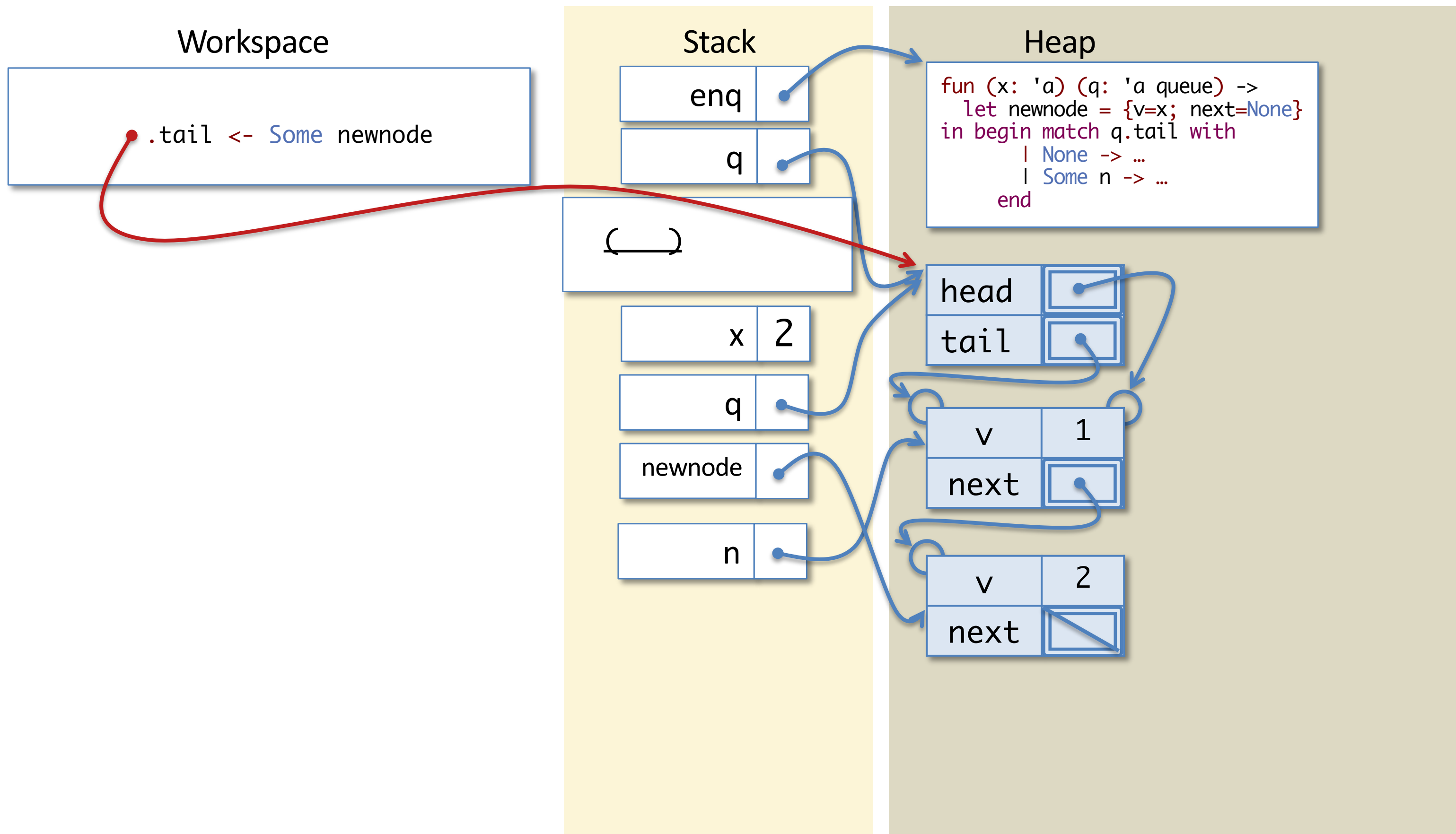
```
q.tail <- Some newnode
```



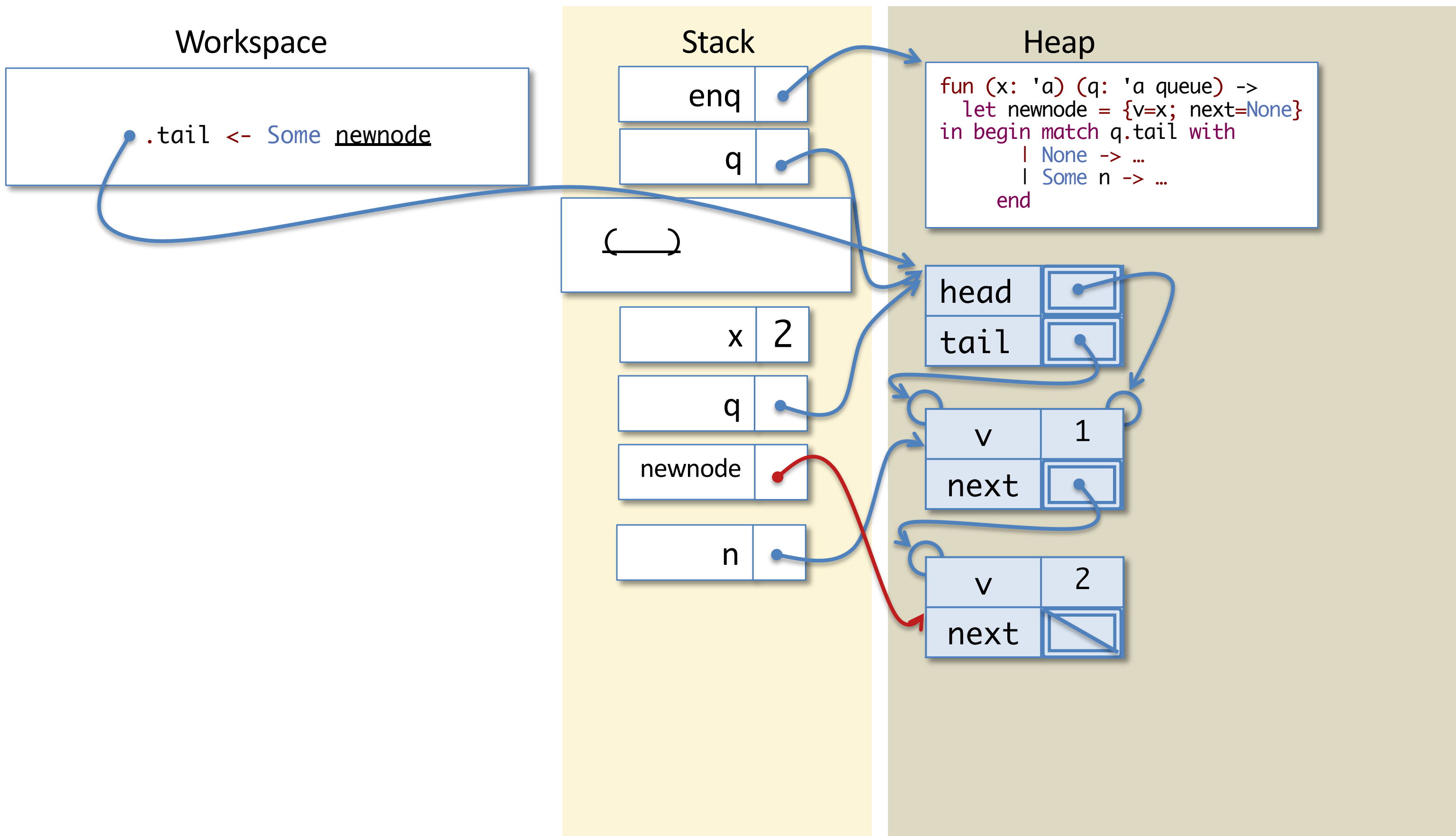
Calling Enq on a non-empty queue



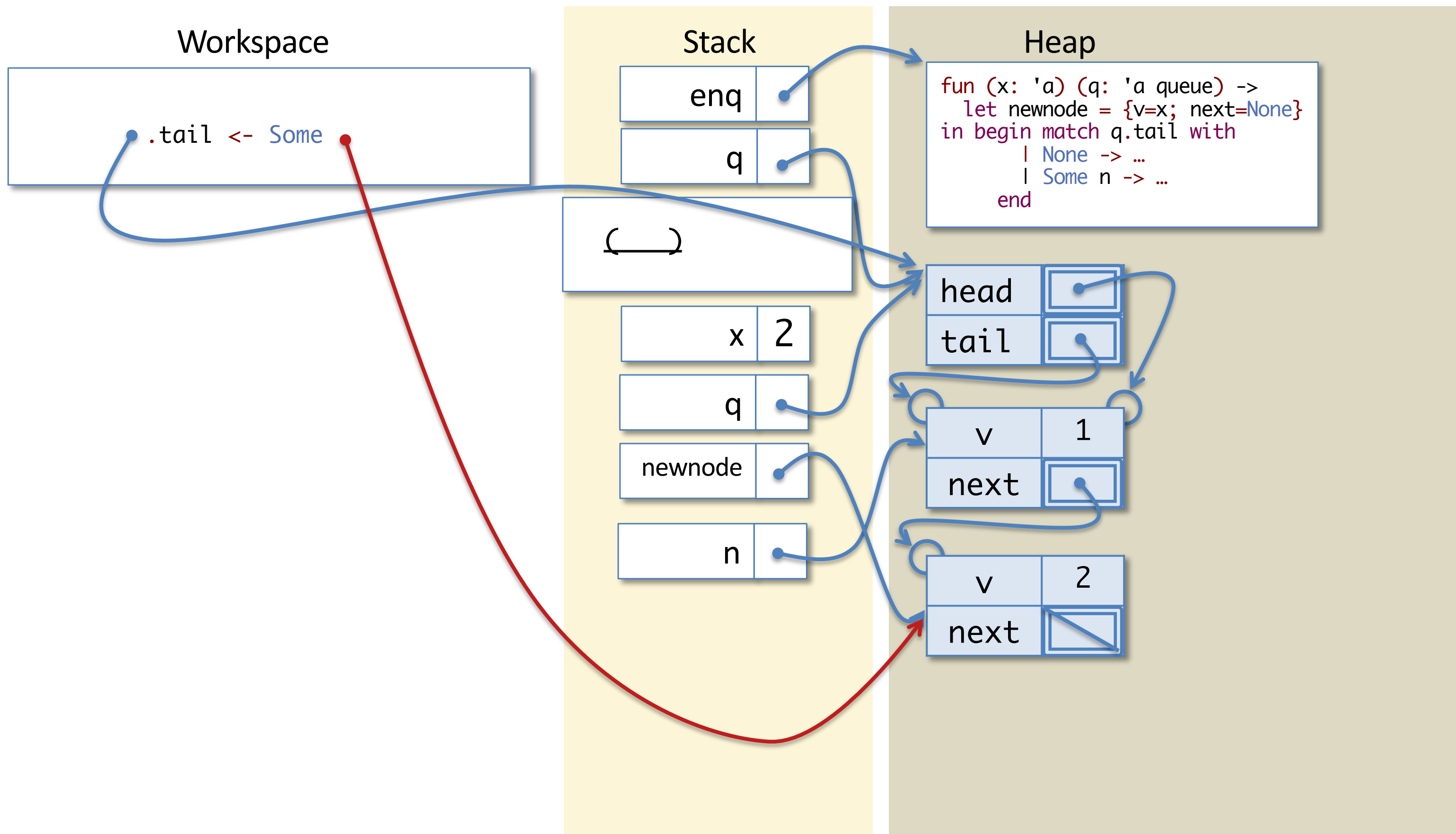
Calling Enq on a non-empty queue



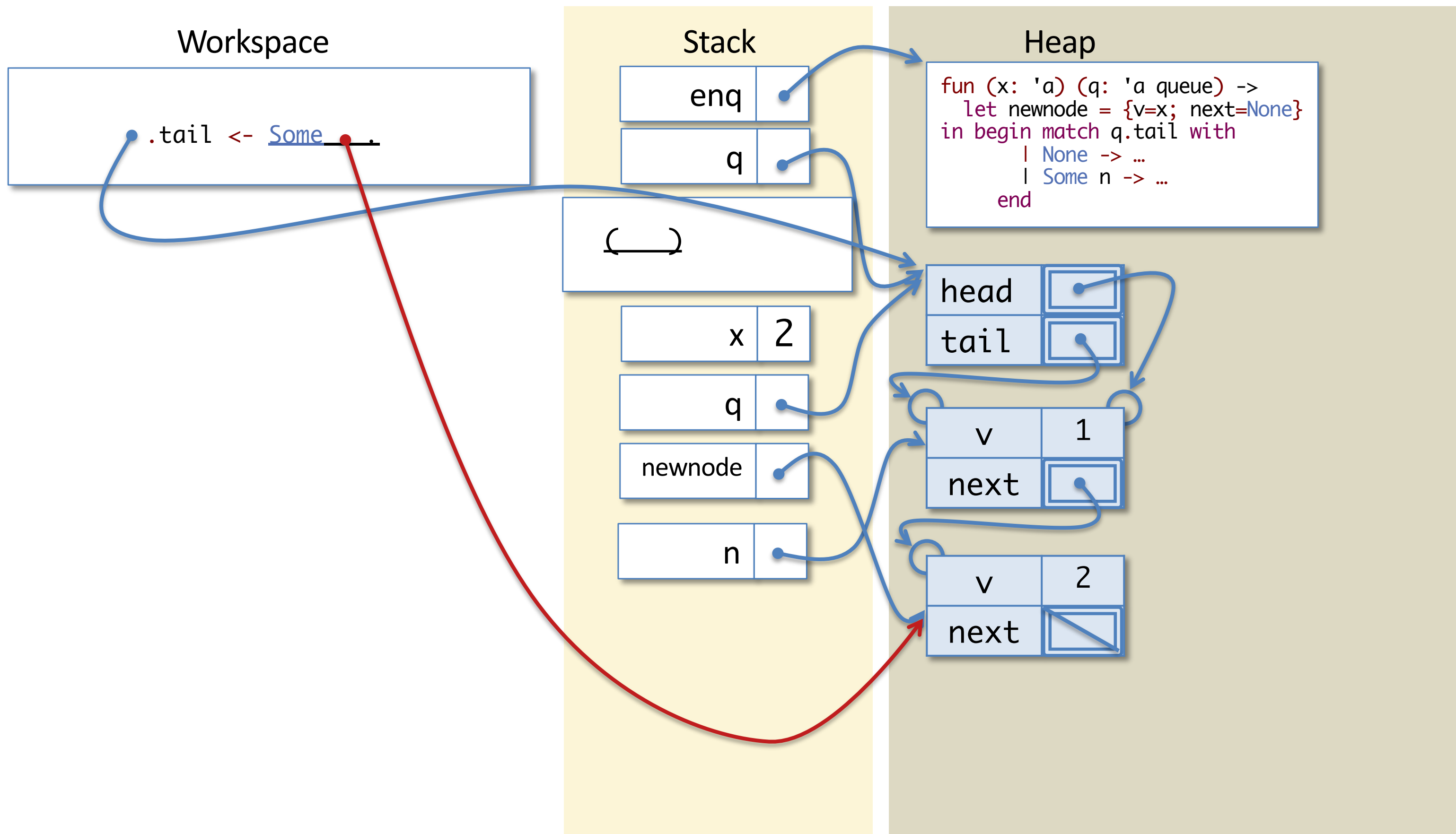
Calling Enq on a non-empty queue



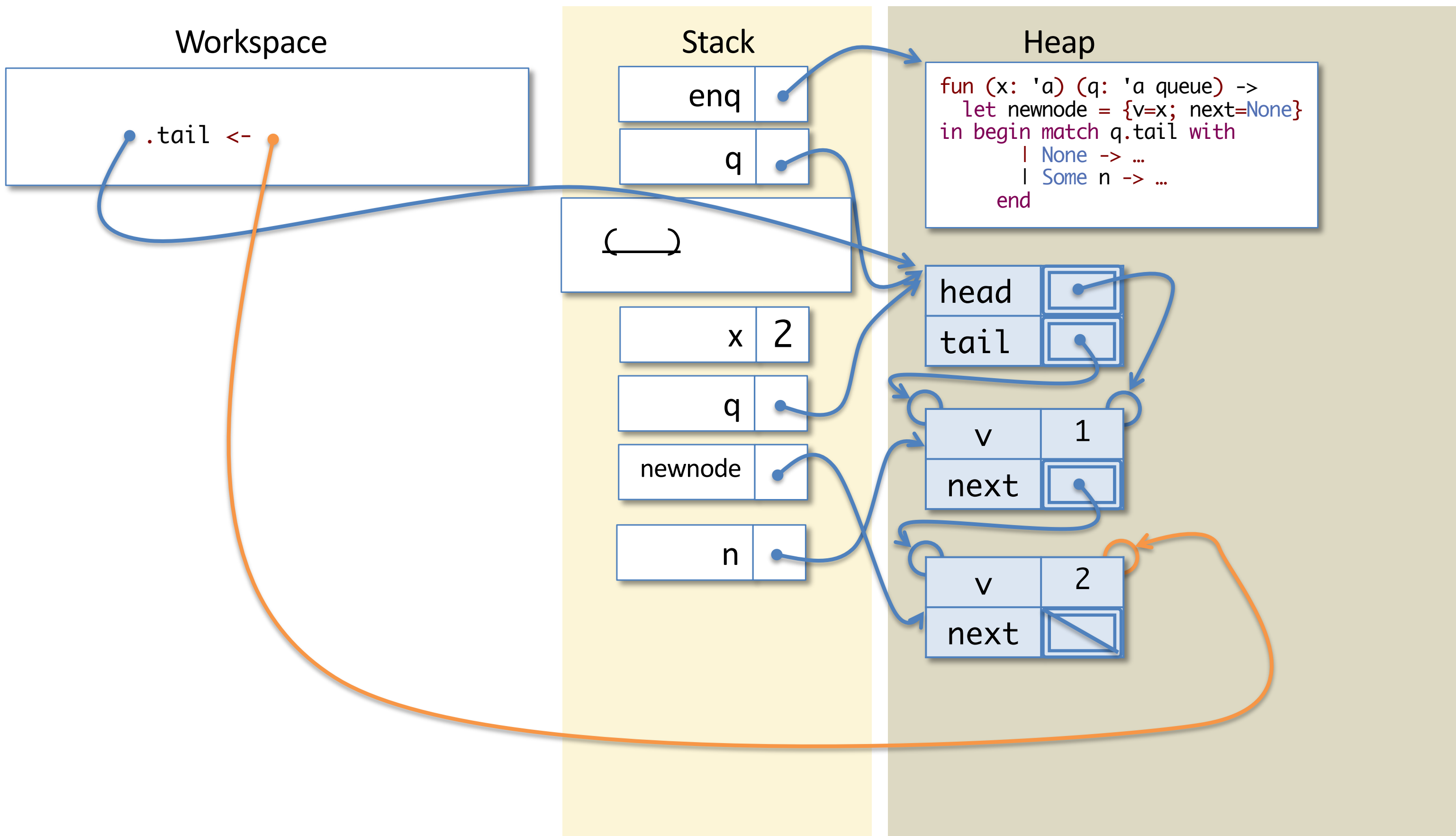
Calling Enq on a non-empty queue



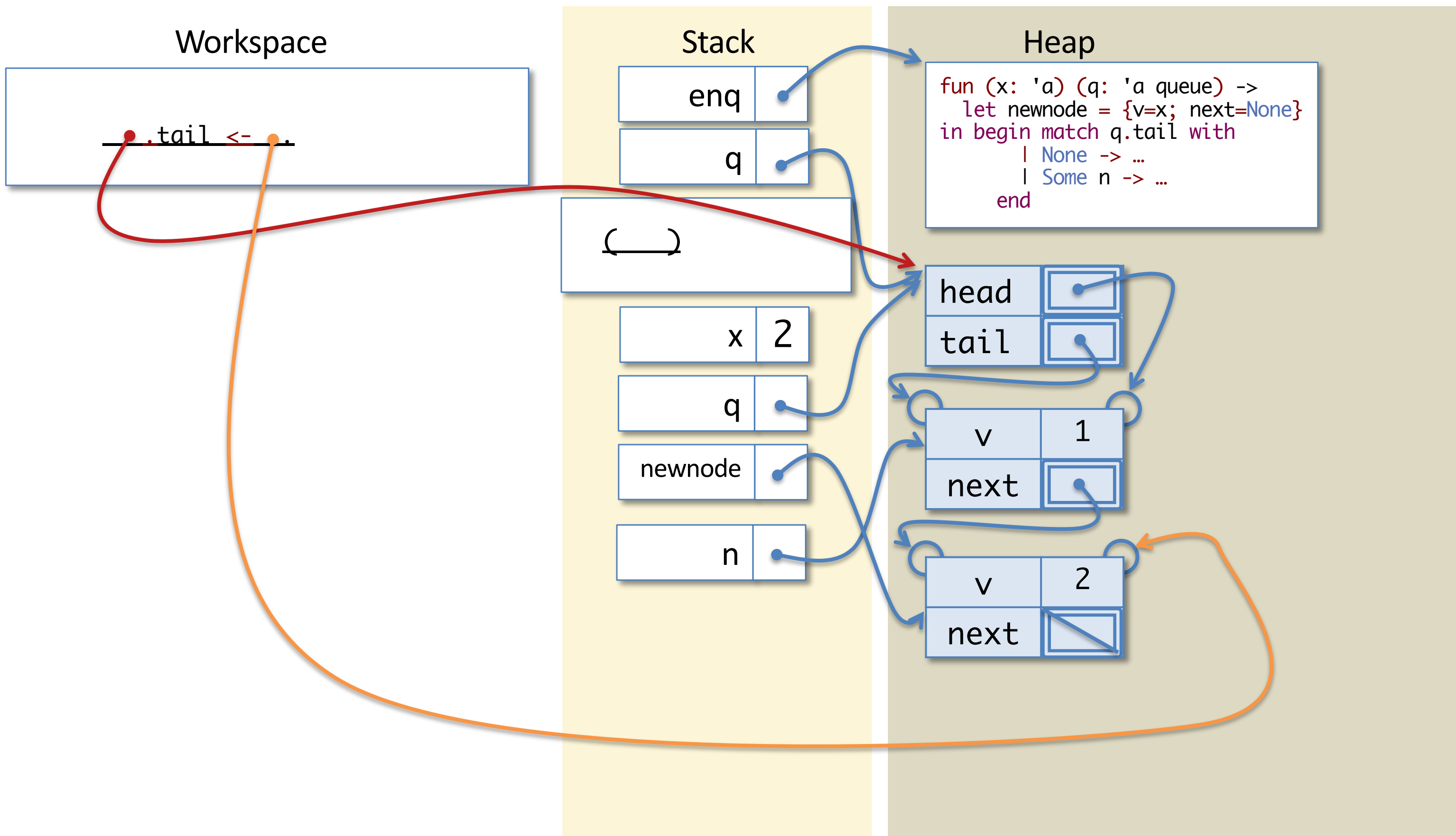
Calling Enq on a non-empty queue



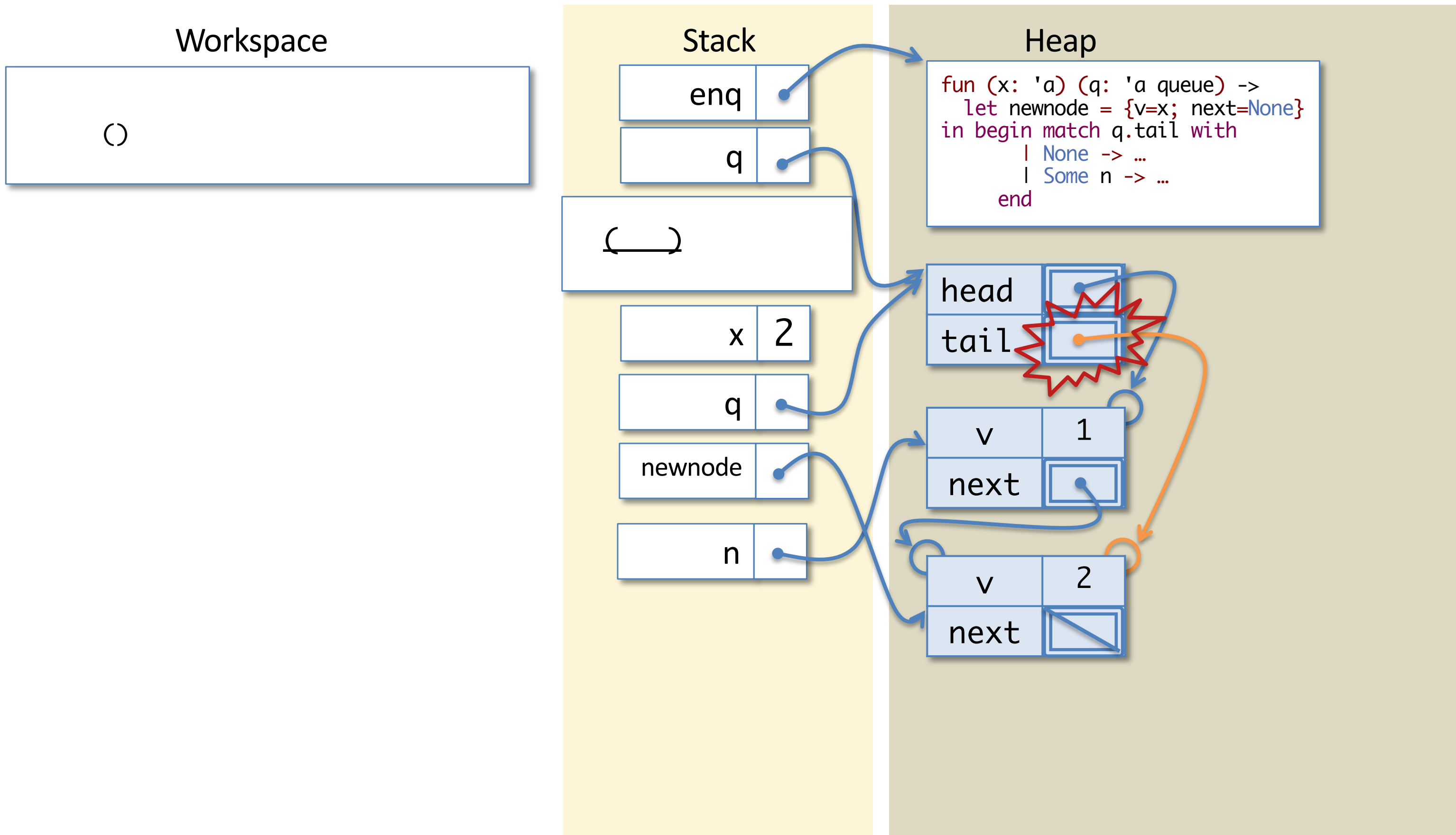
Calling Enq on a non-empty queue



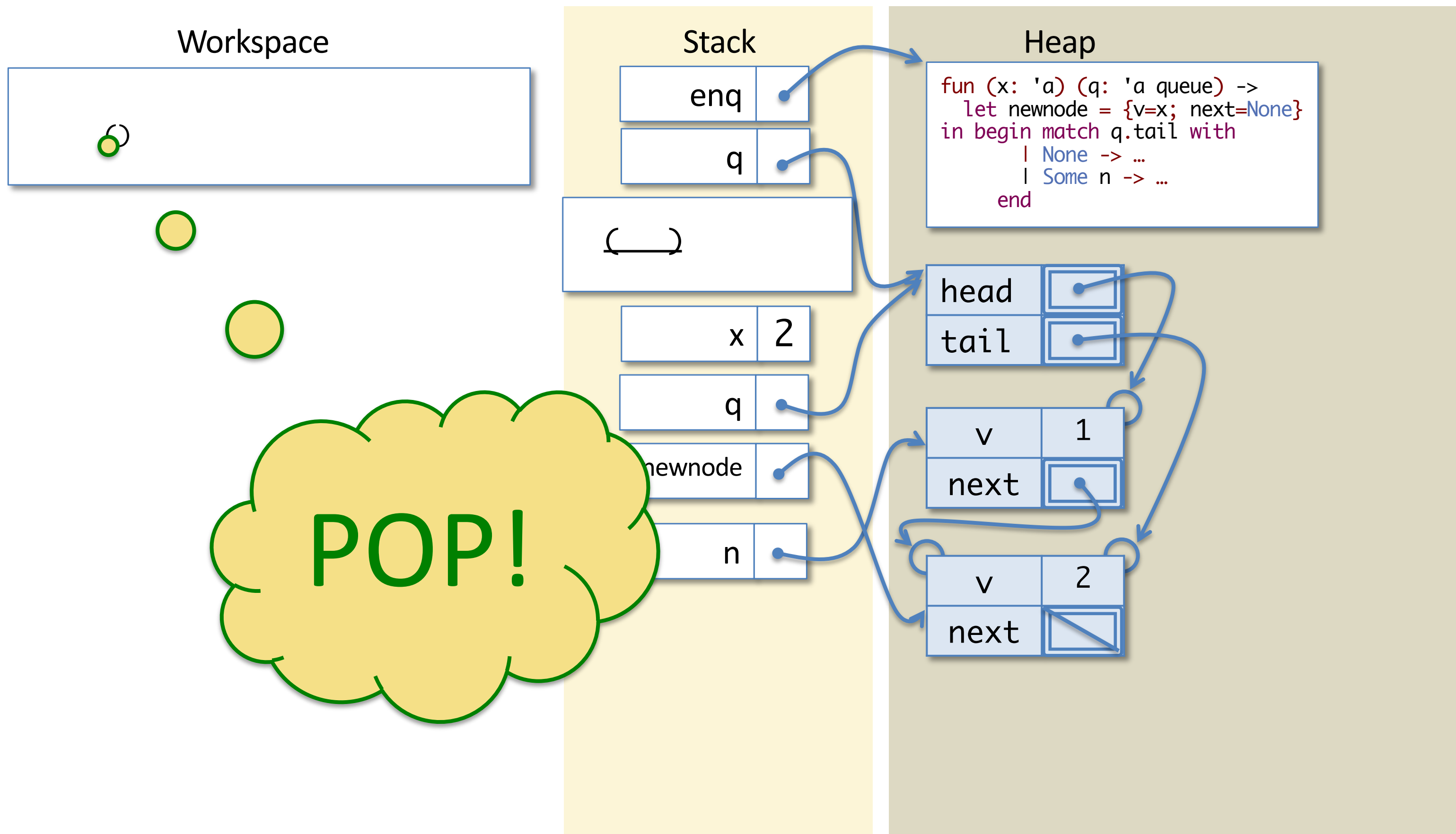
Calling Enq on a non-empty queue



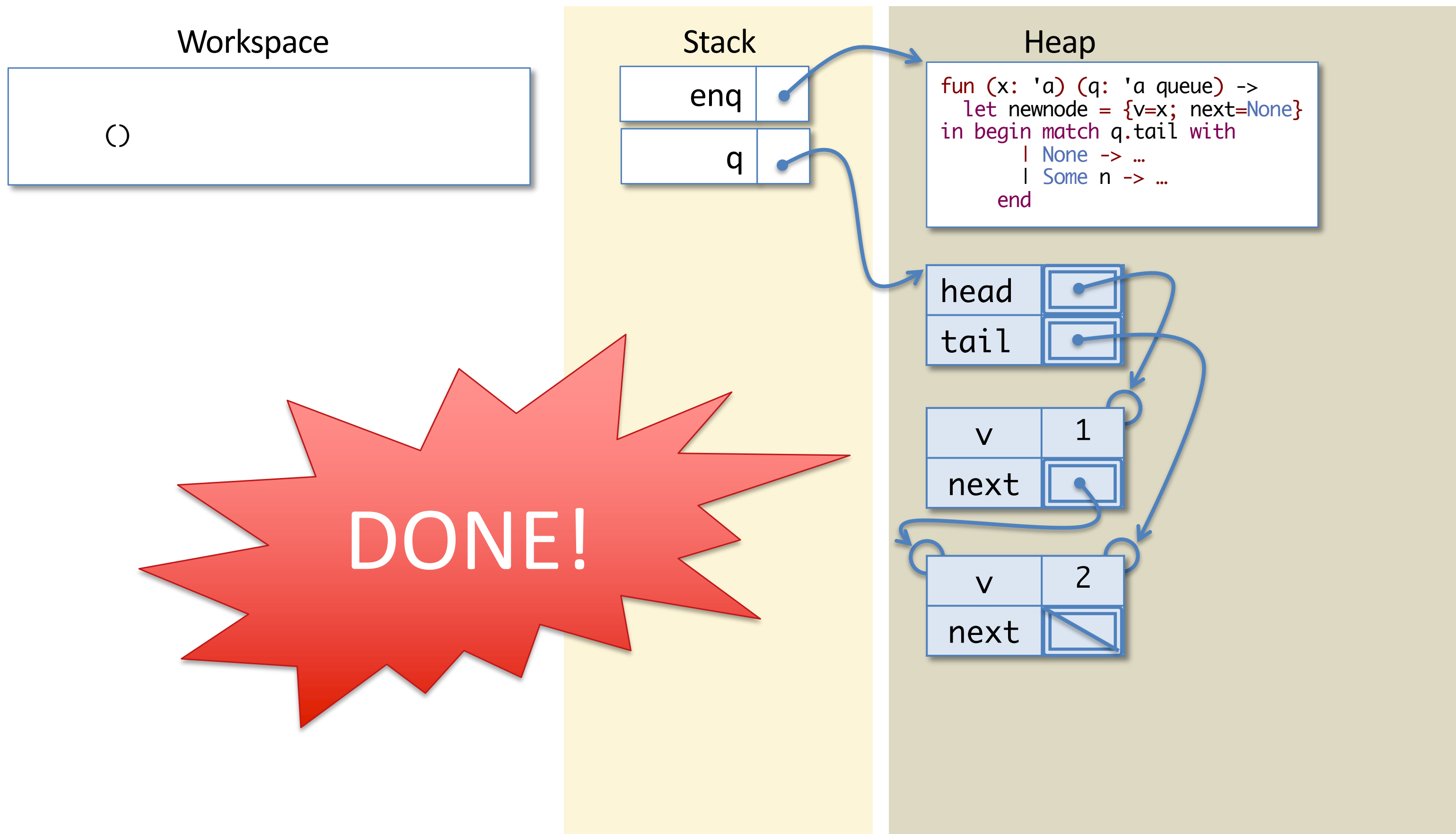
Calling Enq on a non-empty queue



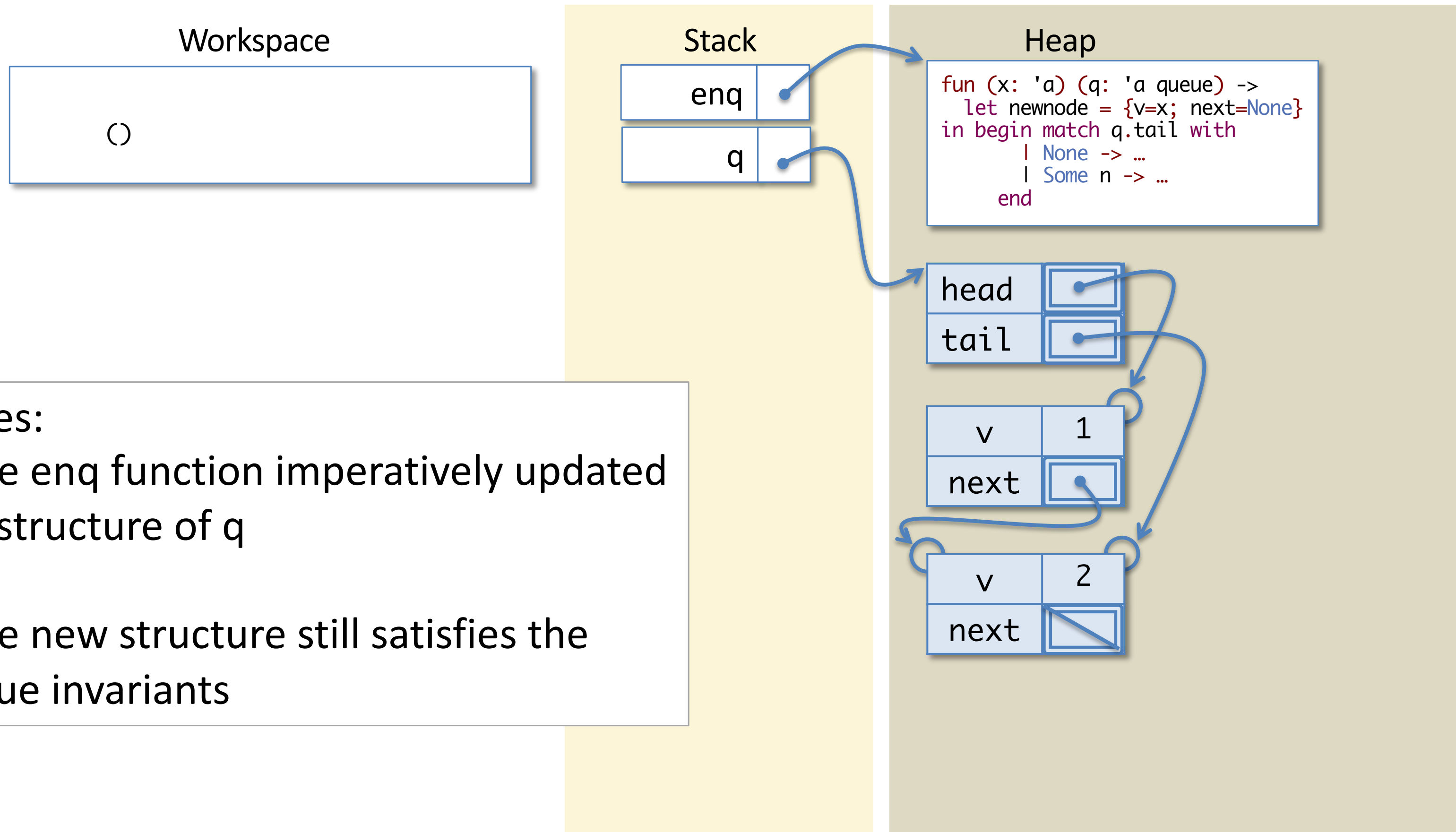
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue



Calling Enq on a non-empty queue



Notes:

- the enq function imperatively updated the structure of q

- the new structure still satisfies the queue invariants

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 }
```

```
(* remove element at the head of queue and return it  *)
 let deq (q: 'a queue) : ‘a option =
   begin match q.head with
     | None -> None
     | Some n ->
         q.head <- n.next;
         Some n.v
   end
```

Which test case shows the bug?

1.
```
let q = create () in
  enq 1 q;
  begin match deq q with
    | None -> false
    | Some hd -> hd = 1
  end
```

2.
```
let q = create () in
  enq 1 q;
  enq 2 q;
  ignore (deq q);
  begin match deq q with
    | None -> false
    | Some hd -> hd = 2
  end
```

3.
```
let q = create () in
  enq 1 q;
  ignore (deq q);
  enq 2 q;
  begin match deq q with
    | None -> false
    | Some hd -> hd = 2
  end
```

ANSWER: 3

4.

deq

```
(* remove an element from the head of the queue *)
 let deq (q: 'a queue) : ‘a option =
   begin match q.head with
     | None -> None
     | Some n ->
         q.head <- n.next;
         if n.next = None then q.tail <- None;
         Some n.v
   end
```

- The code for deq must also “patch pointers” to maintain the queue invariant:
  - The head pointer is always updated to the next element in the queue.
  - If the removed node was the last one in the queue, the tail pointer must be updated to None

Mutable Queues: Queue Length

working with singly linked data structures

Queue Length

Suppose we want to extend the interface with a length function:

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

  (* Get the length of the queue *)
  val length : 'a queue -> int
end
```

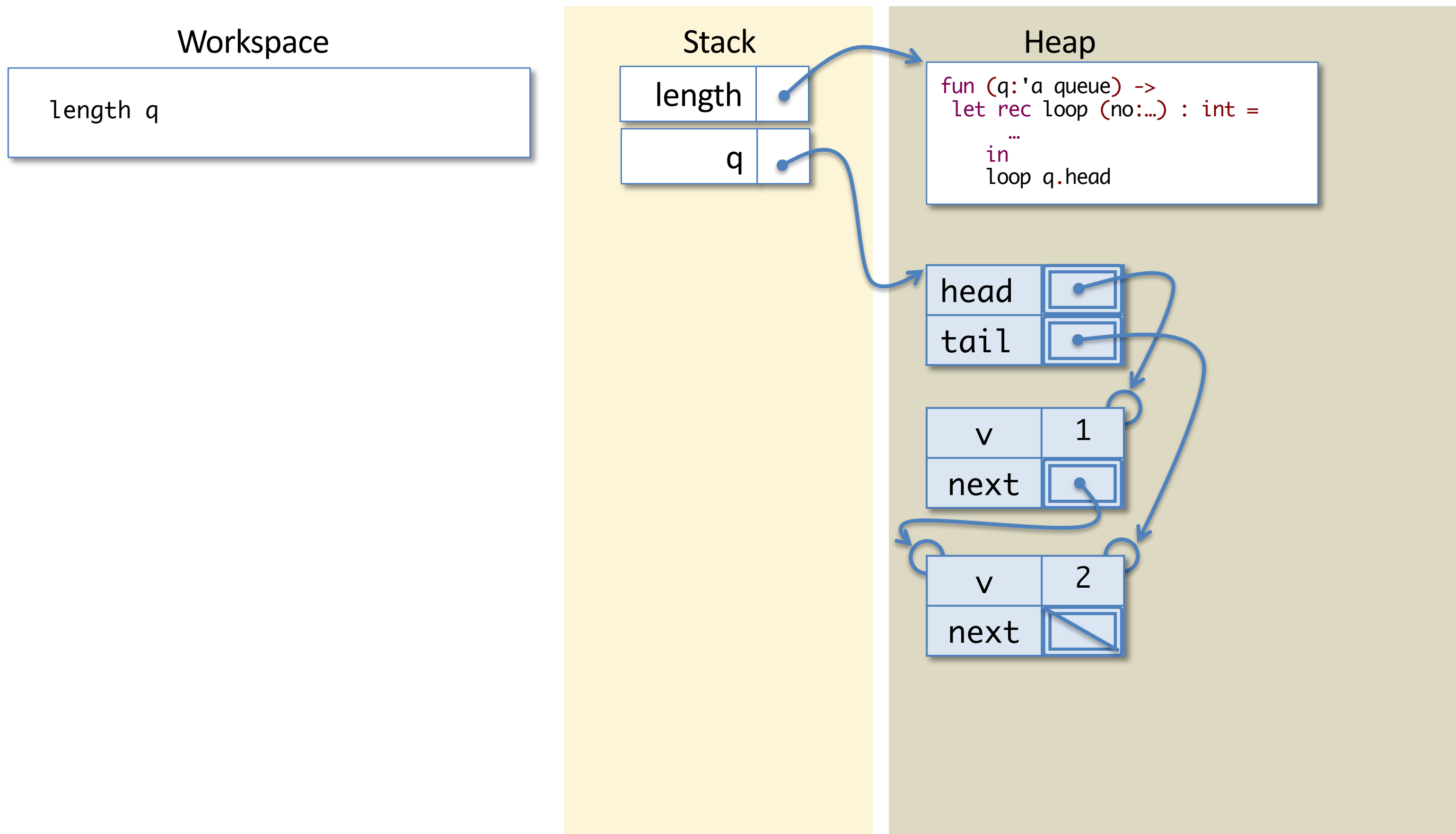
How can we implement it?

`length` (recursively)

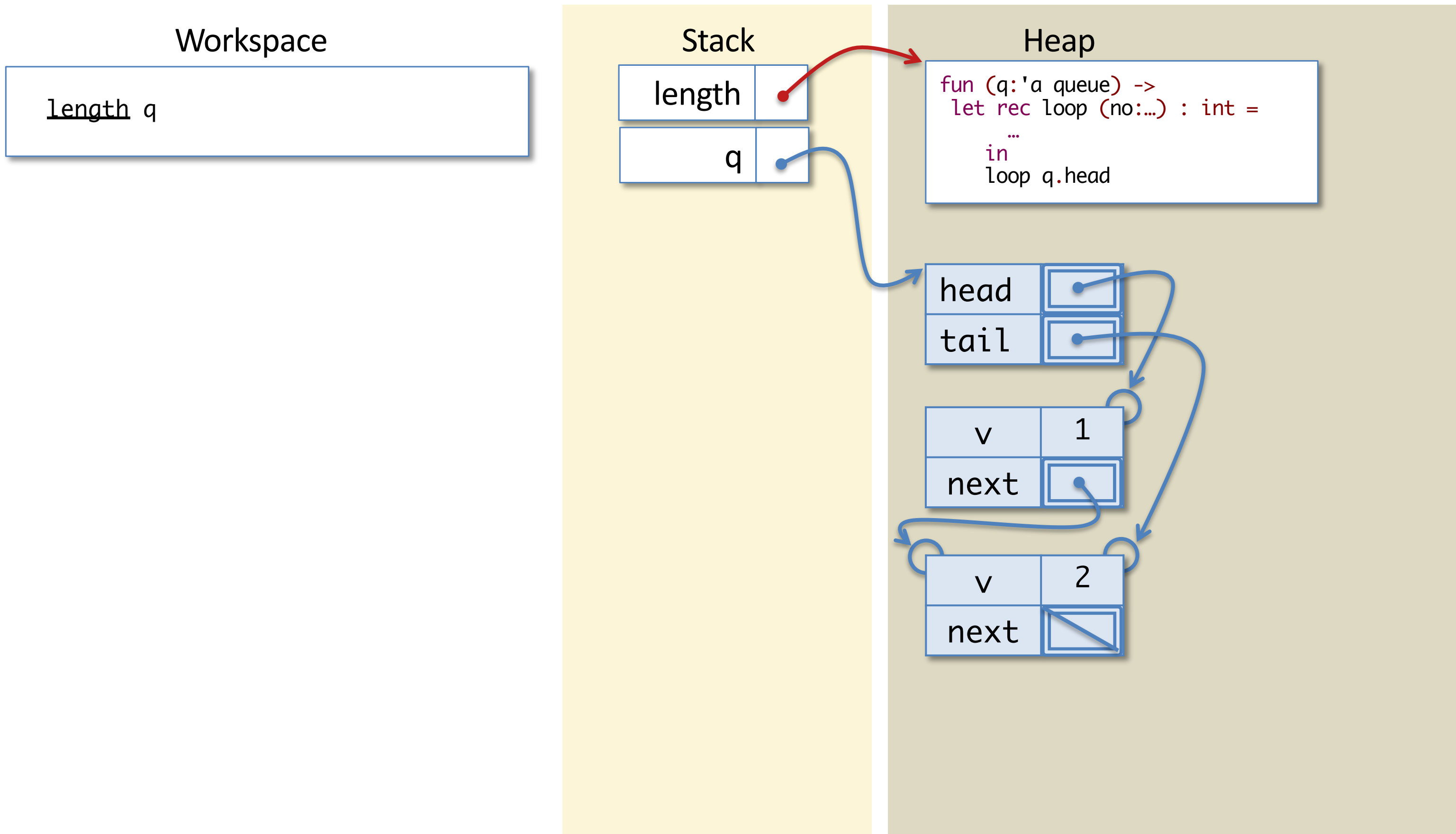
```
(* Calculate the length of the queue recursively *)
let length (q:'a queue) : int =
  let rec loop (no: 'a qnode option) : int =
    begin match no with
      | None -> 0
      | Some n -> 1 + (loop n.next)
    end
  in
  loop q.head
```

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

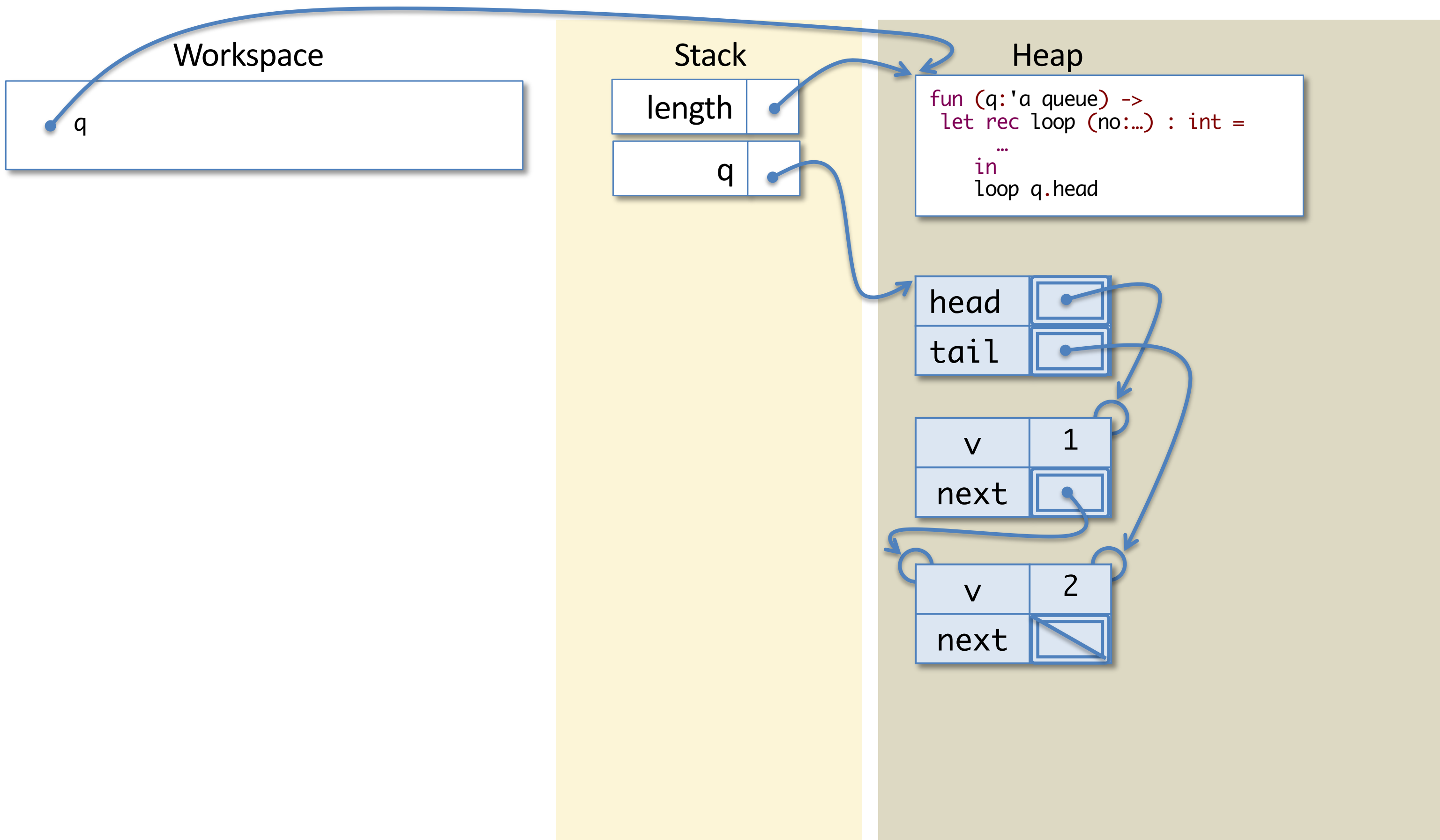
Evaluating length



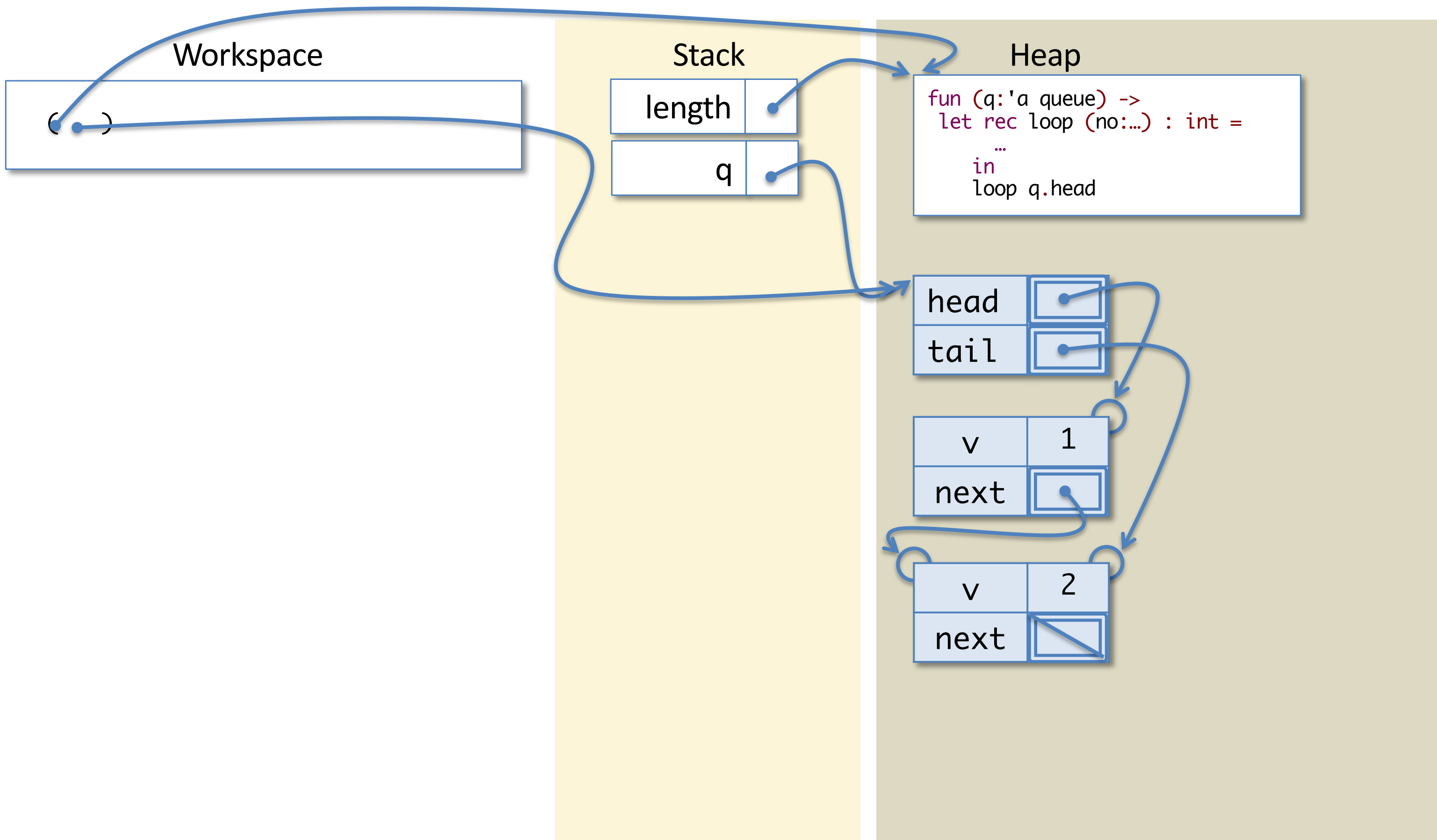
Evaluating length



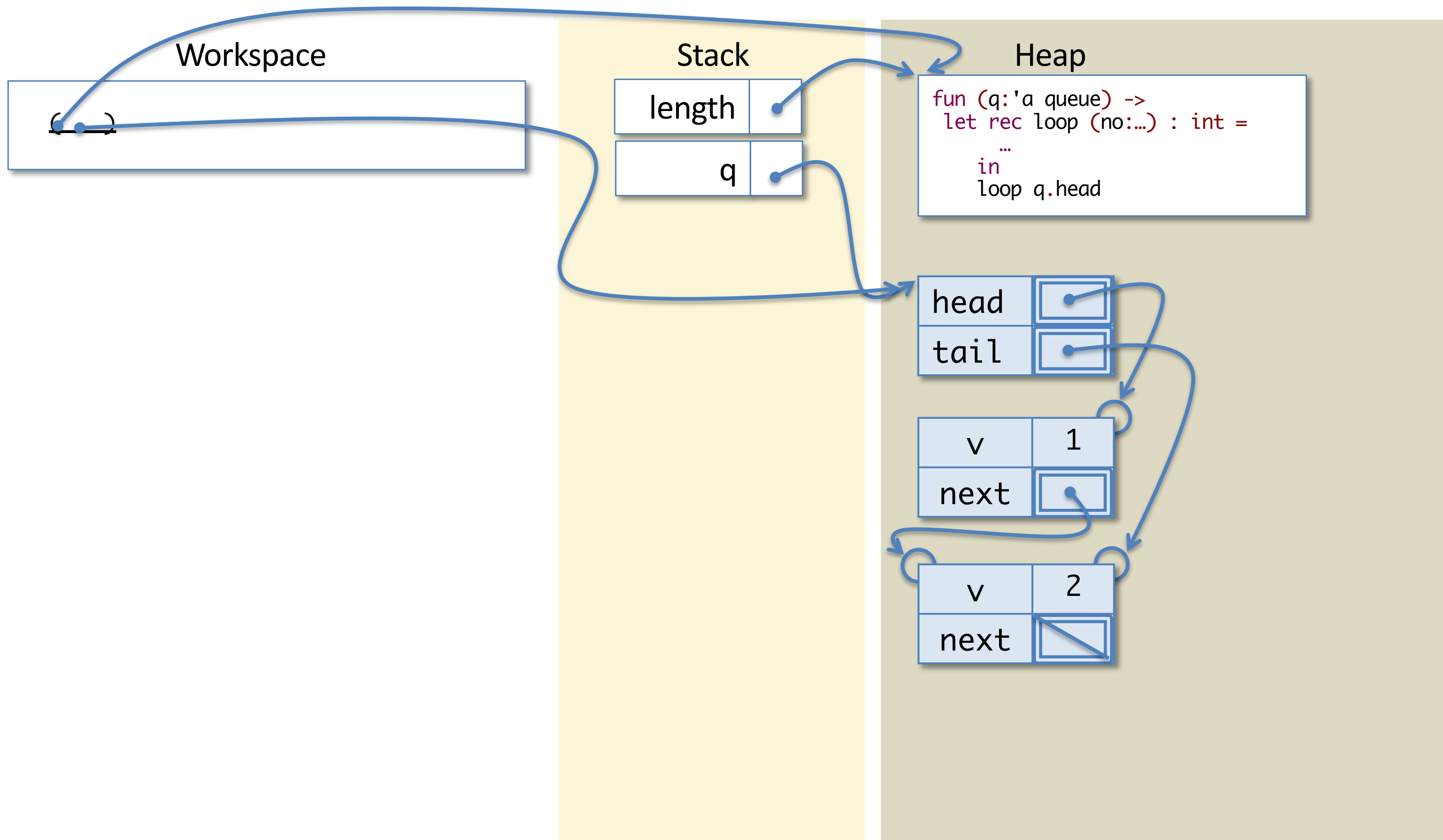
Evaluating length



Evaluating length



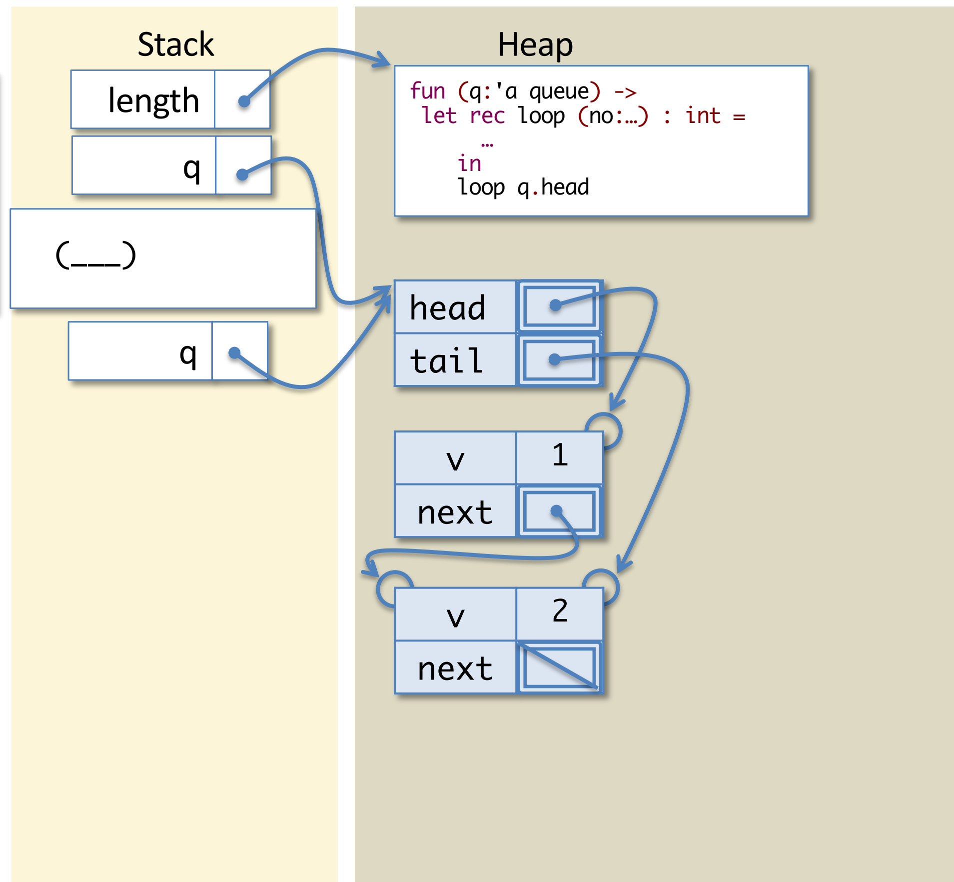
Evaluating length



Evaluating length

Workspace

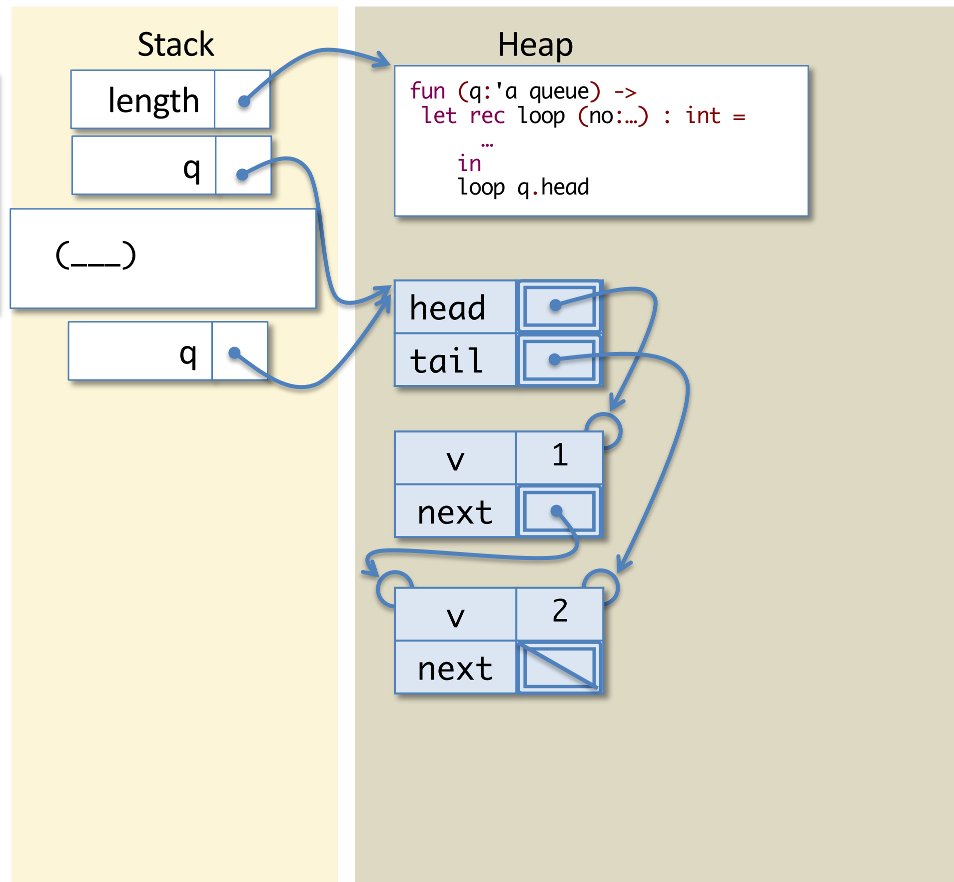
```
let rec loop (no: …) : int =
      begin match no with
        | None -> 0
        | Some n -> 1 + (loop n.next)
      end
    in
    loop q.head
```



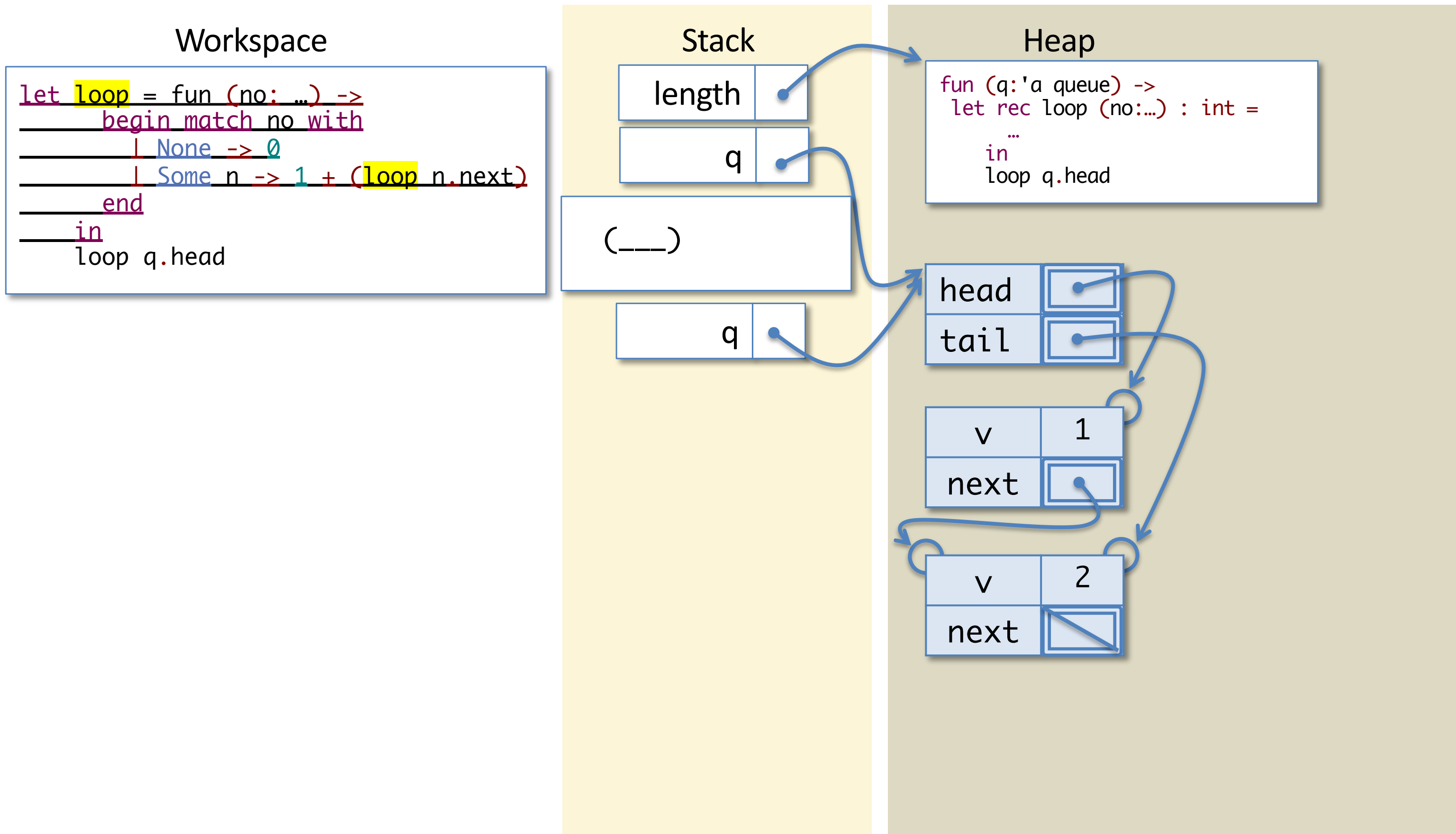
Evaluating length

Workspace

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



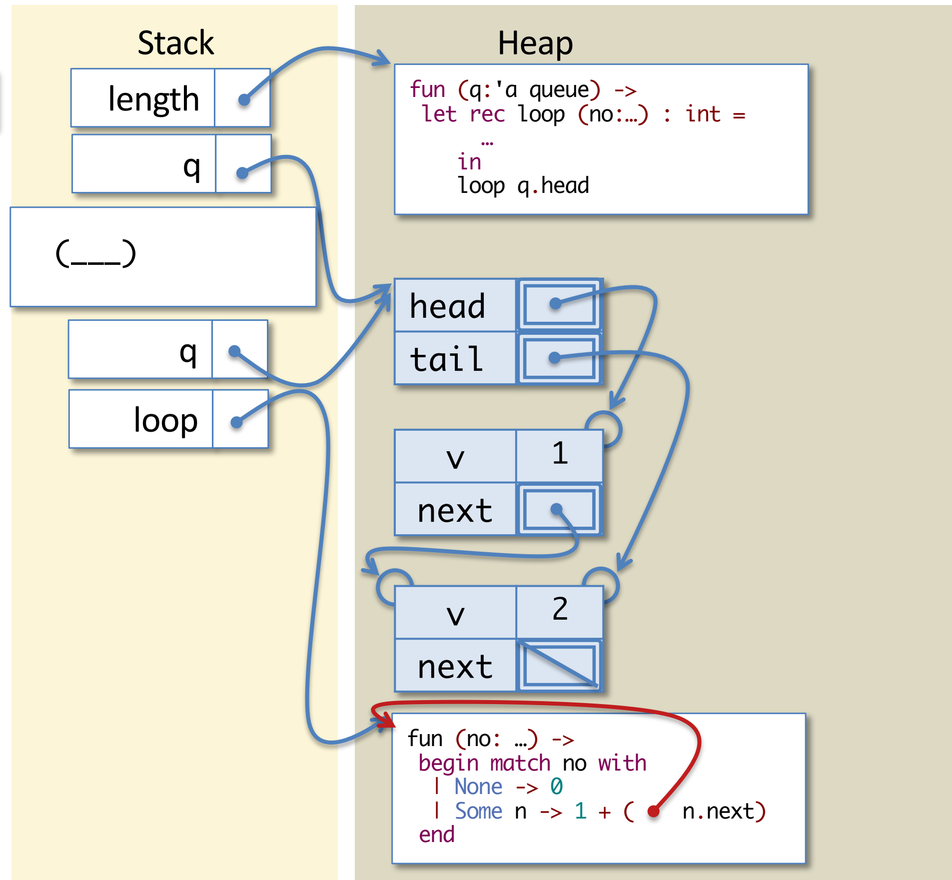
Evaluating length



Evaluating length

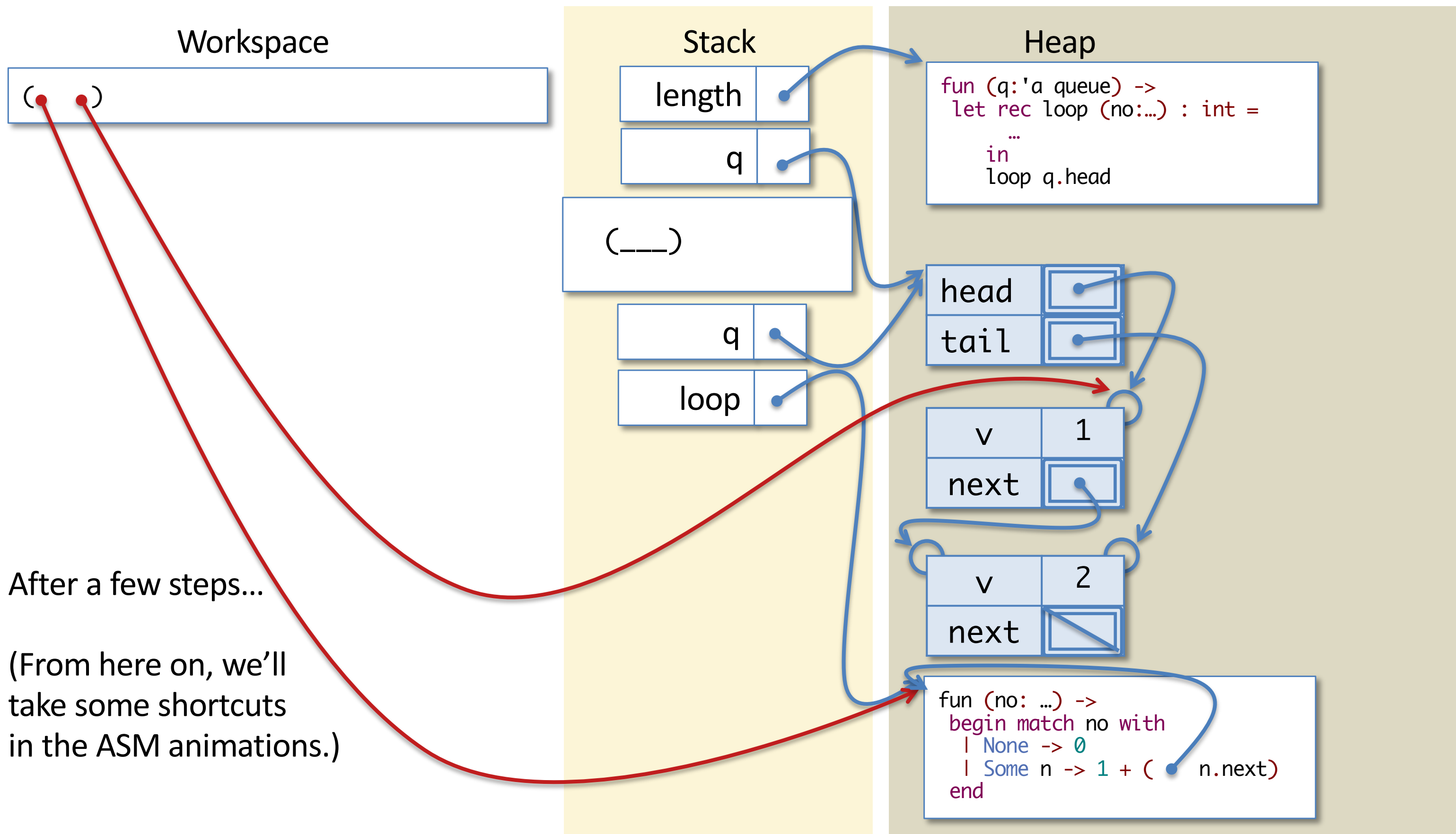
Workspace

```
loop q.head
```



The reference to loop in its own body is *backpatched* when it moves into the heap…

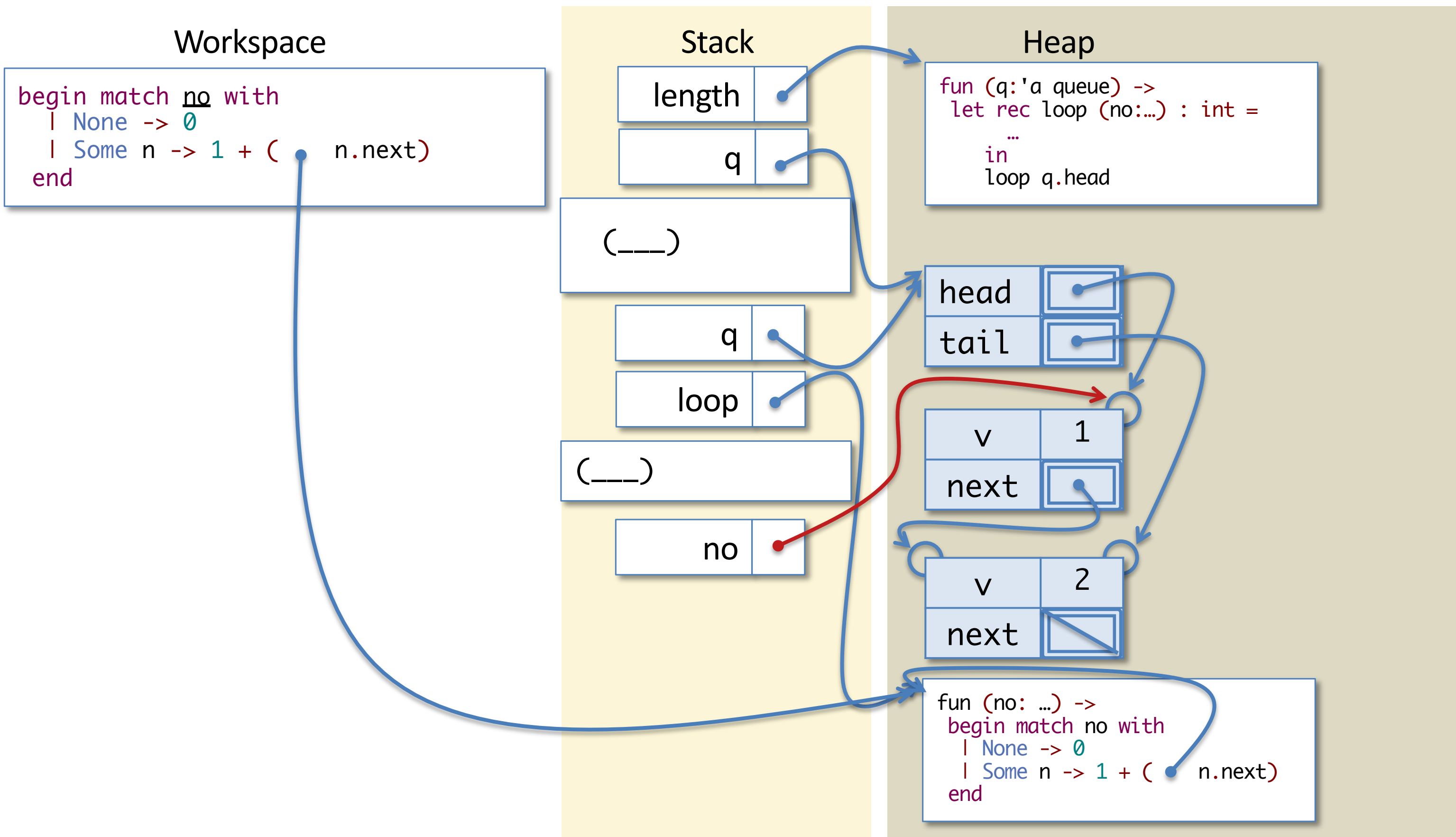
Evaluating length



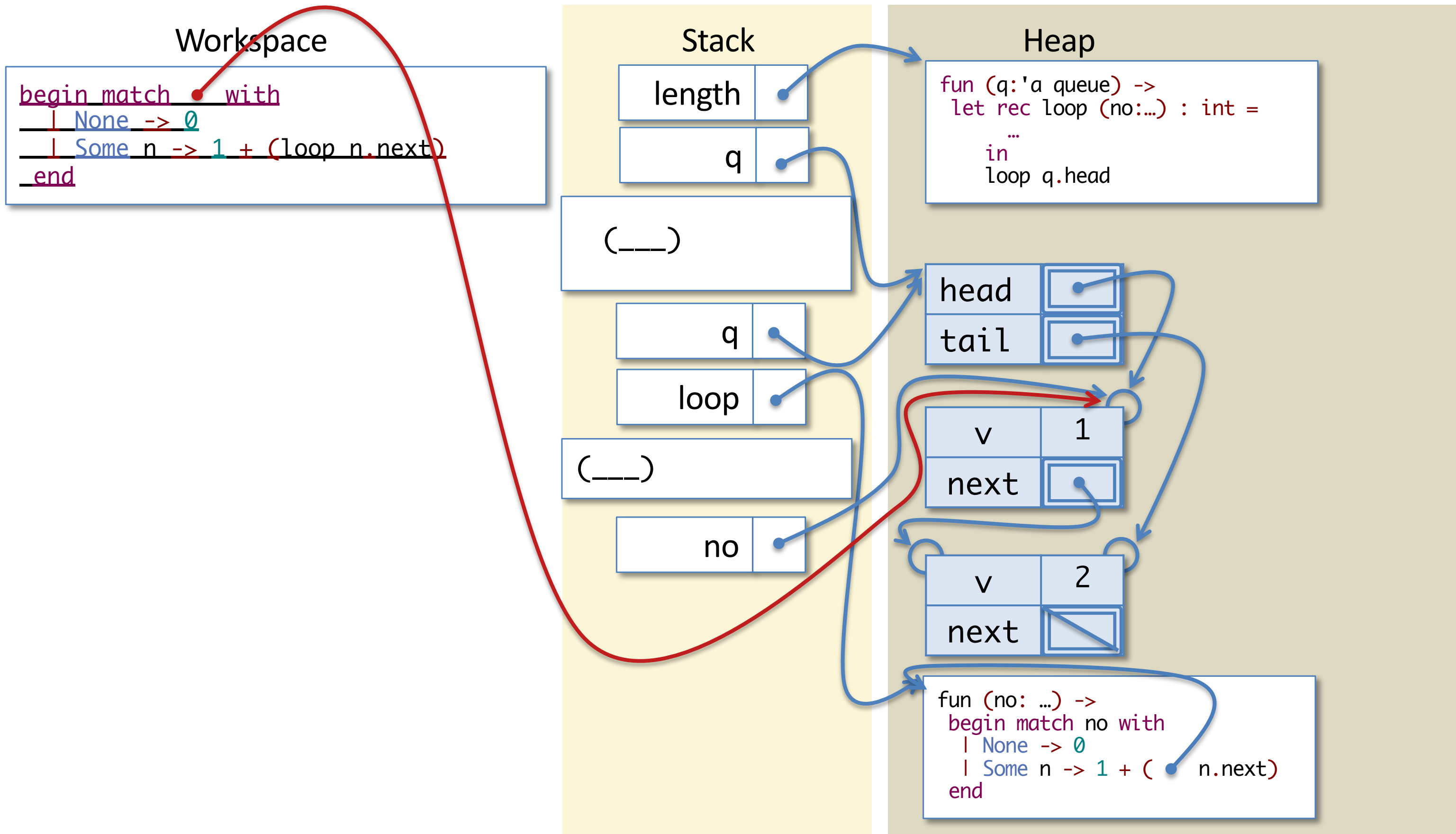
After a few steps…

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

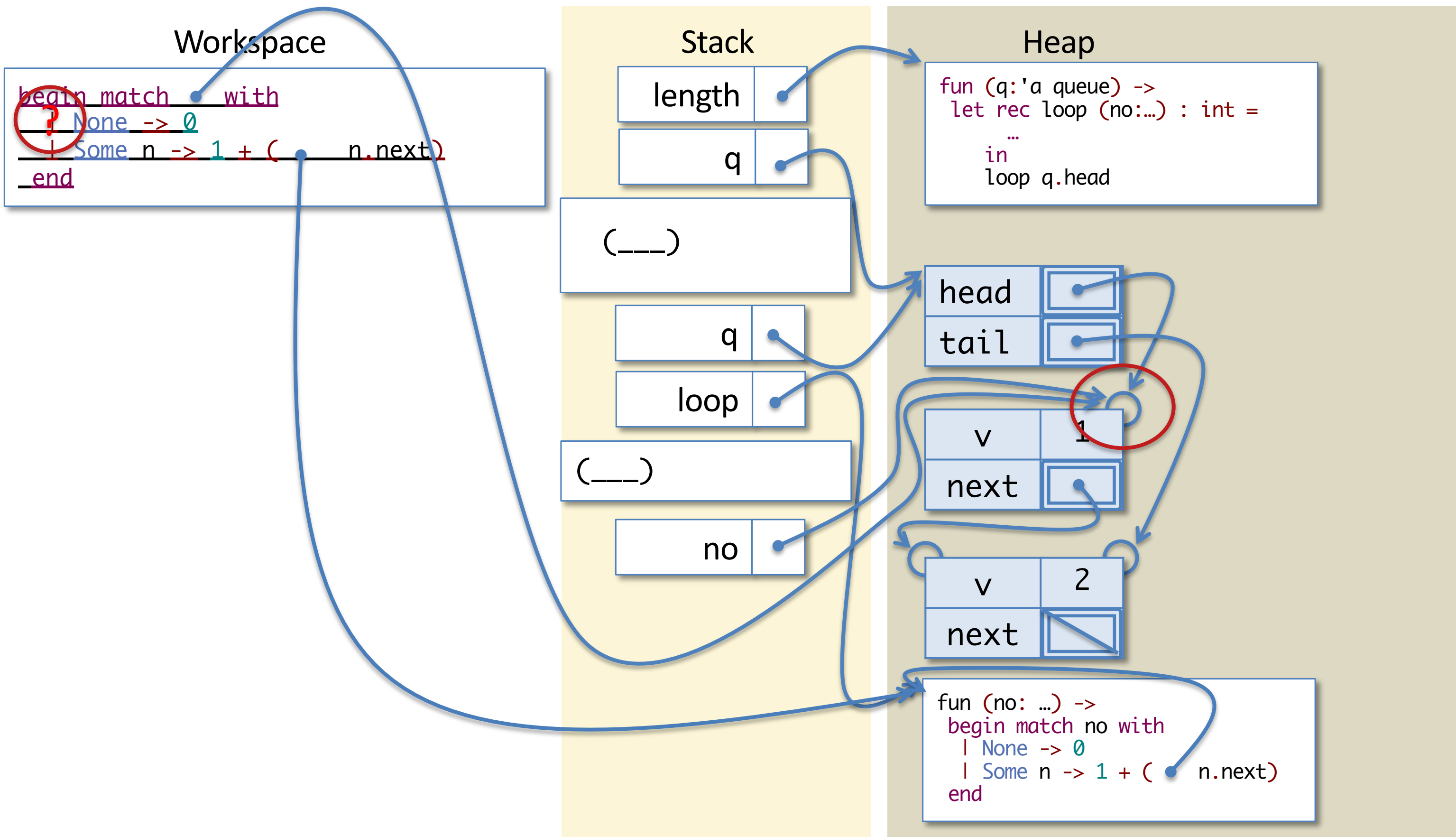
Evaluating length



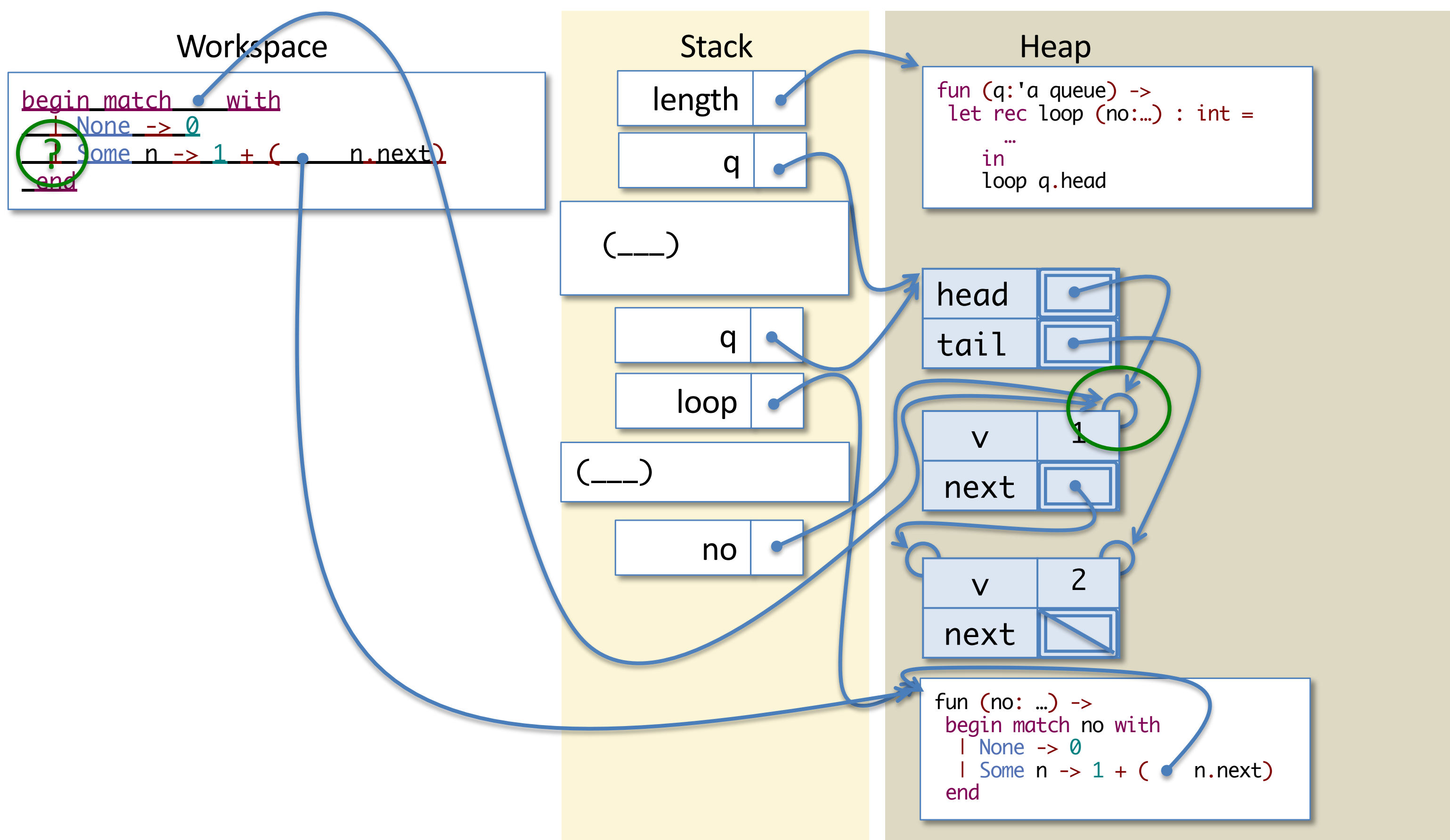
Evaluating length



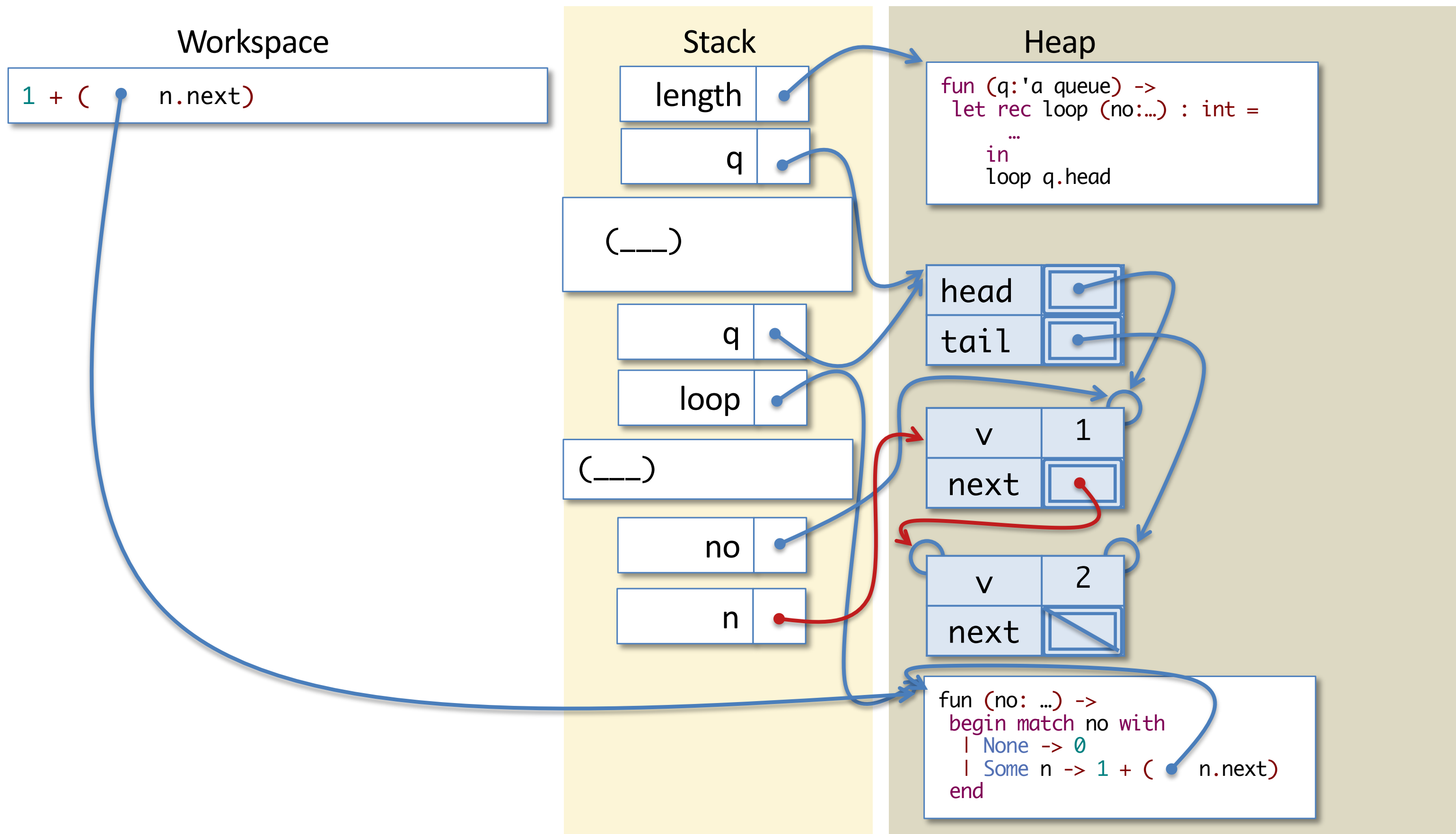
Evaluating length



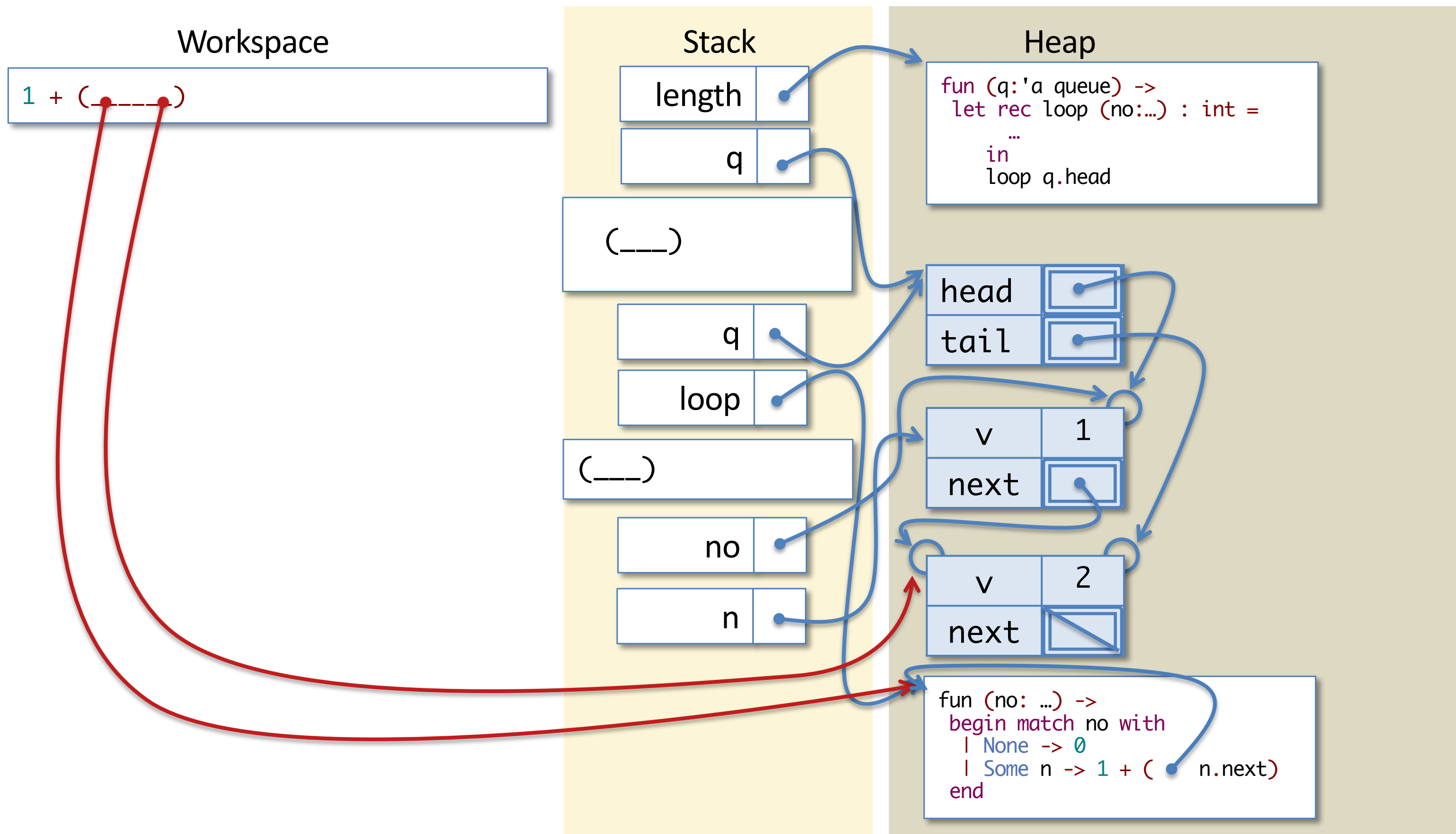
Evaluating length



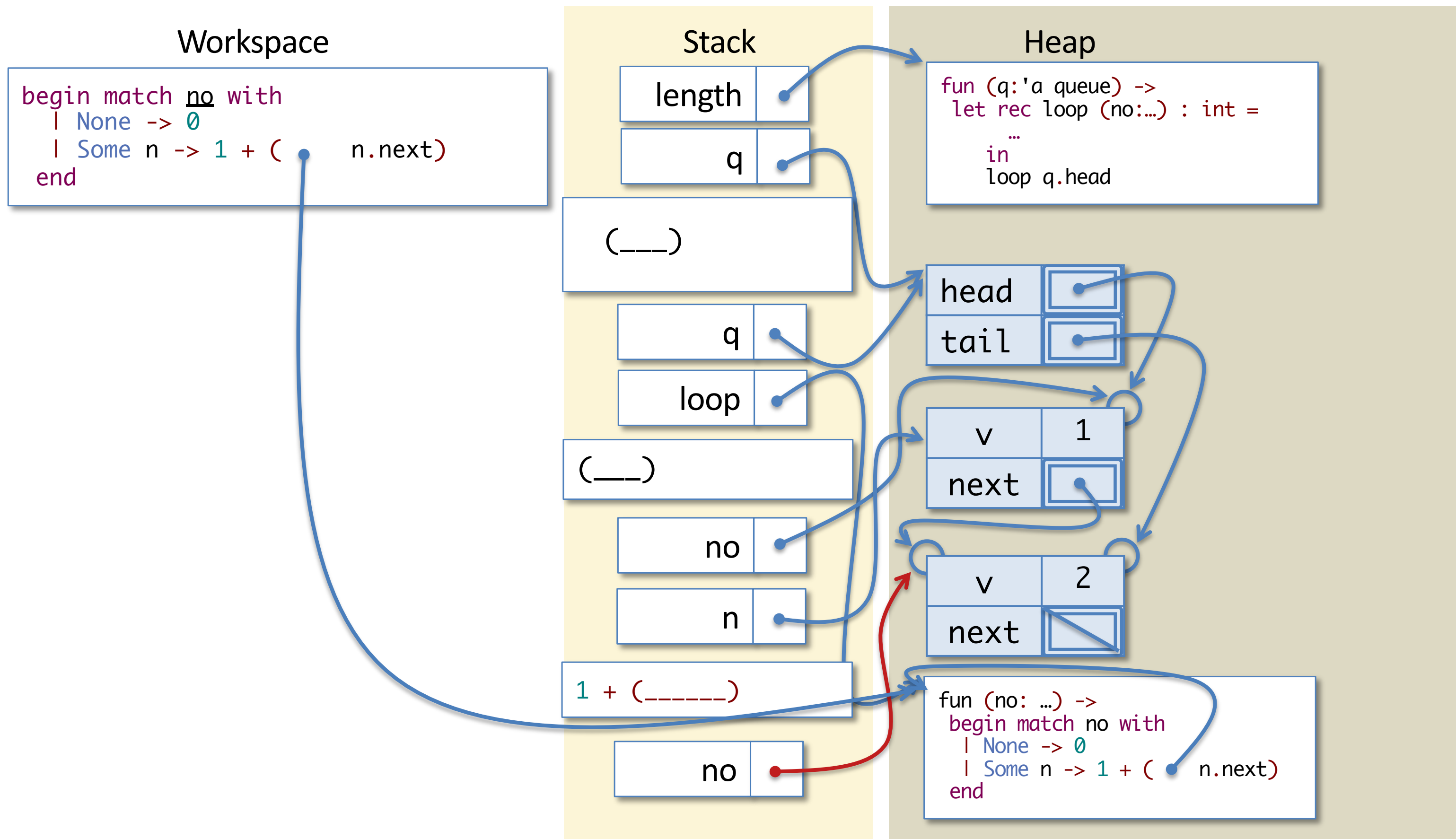
Evaluating length



Evaluating length

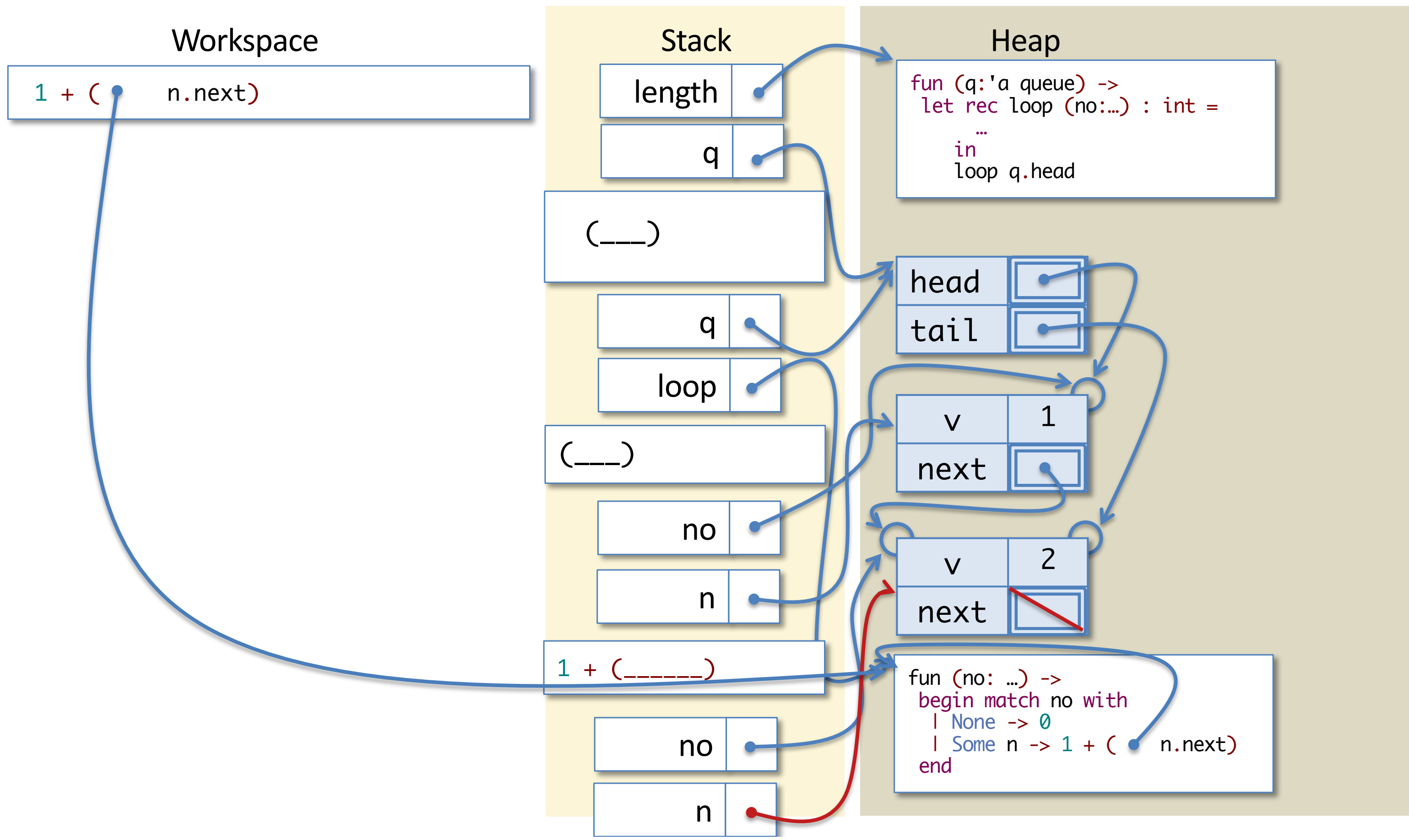


Evaluating length



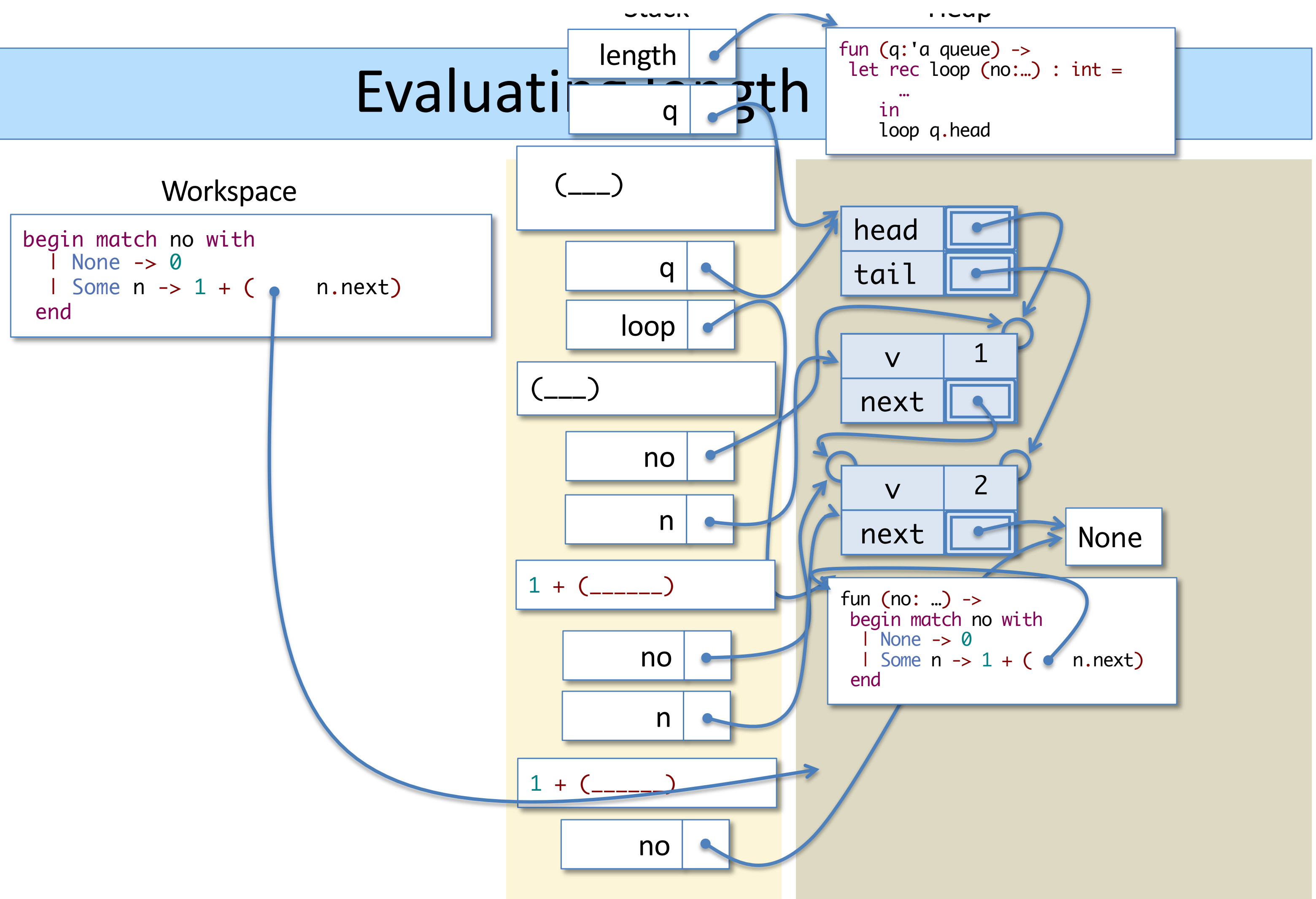
…after a few more steps…

Evaluating length



…after a few more steps…

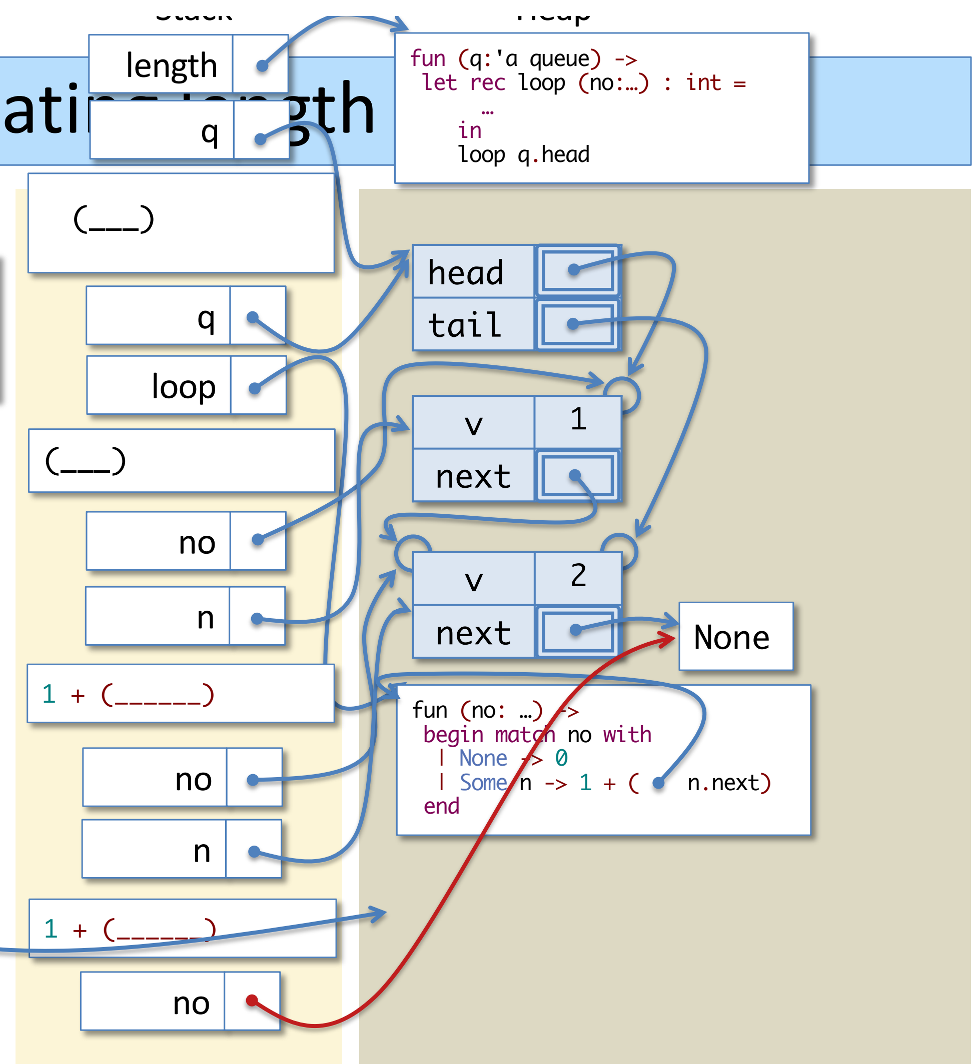
Evaluating length



Evaluating length

Workspace

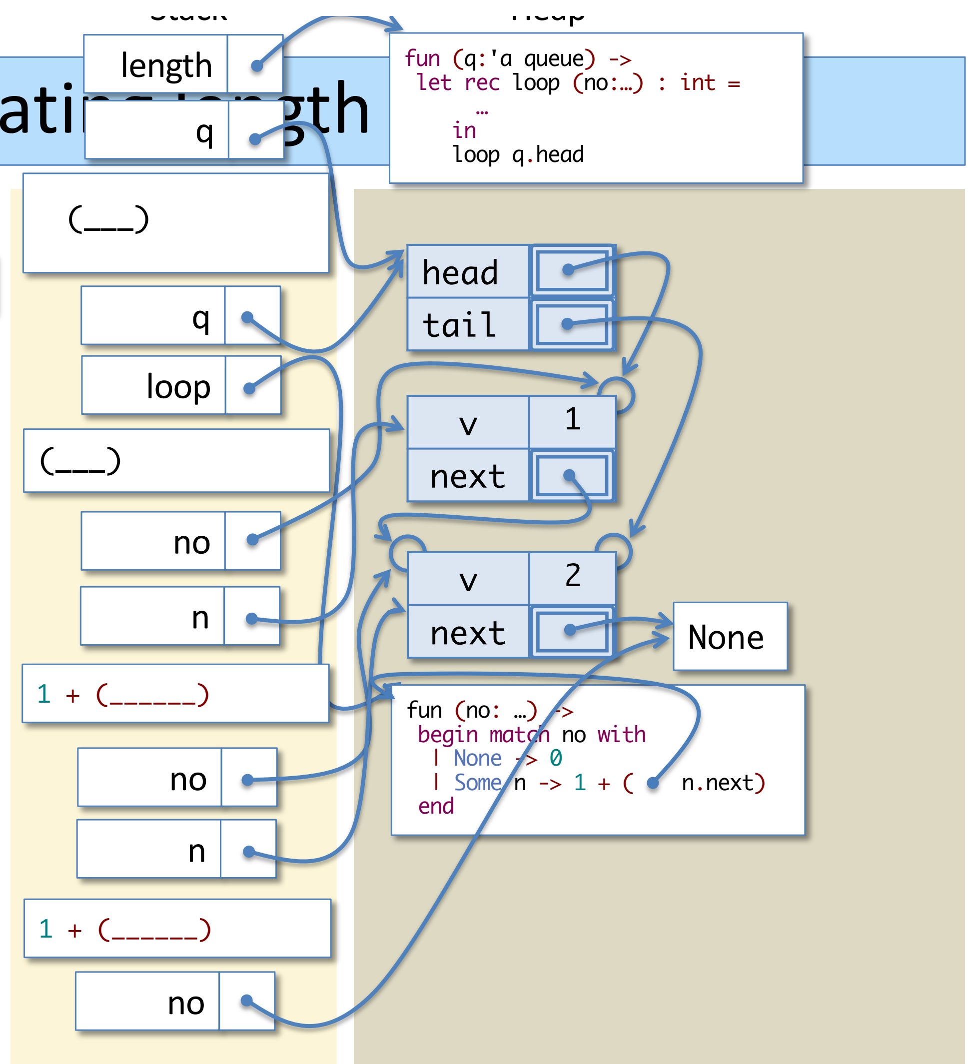
```
begin match no with
  | None -> 0
  | Some n -> 1 + (       n.next)
end
```



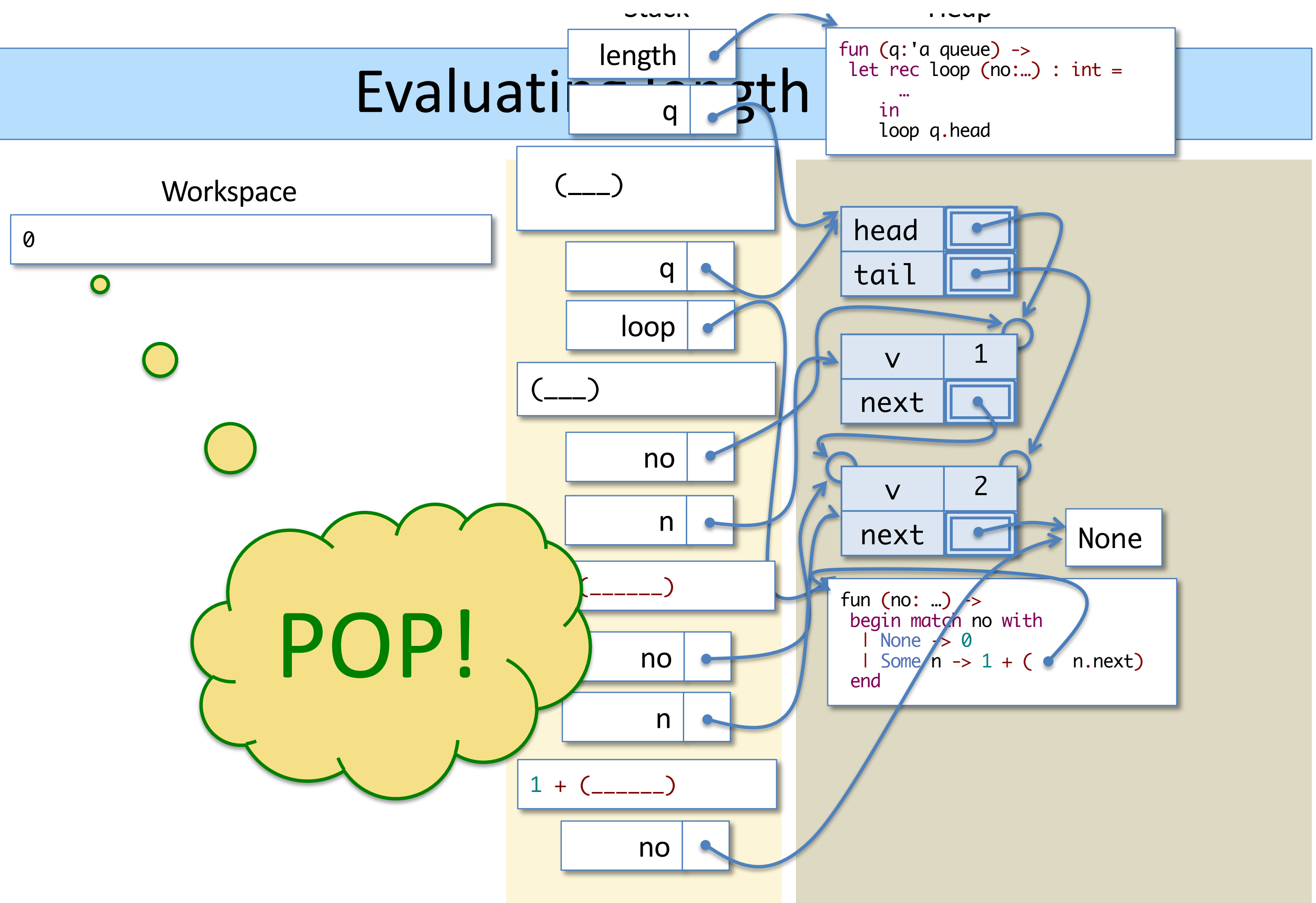
Evaluating length

Workspace

```
0
```



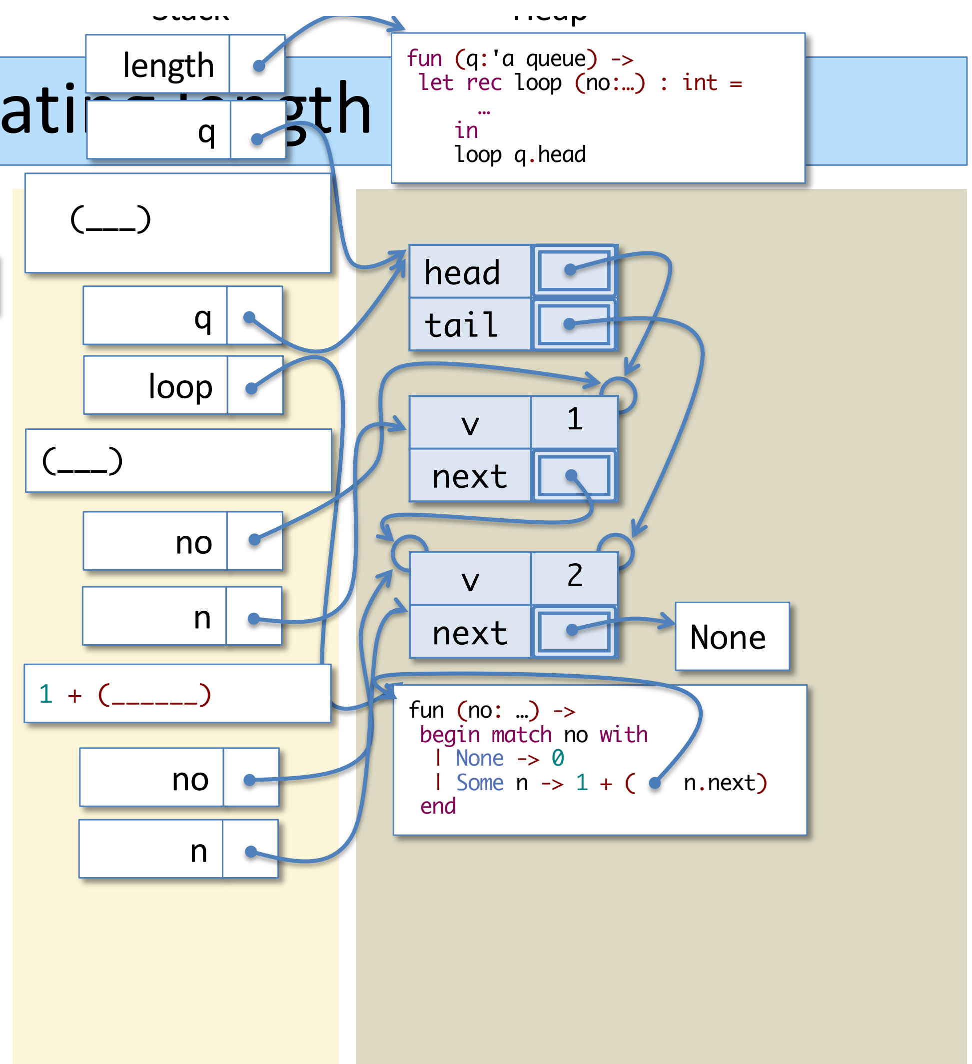
Evaluati[illegible]gth



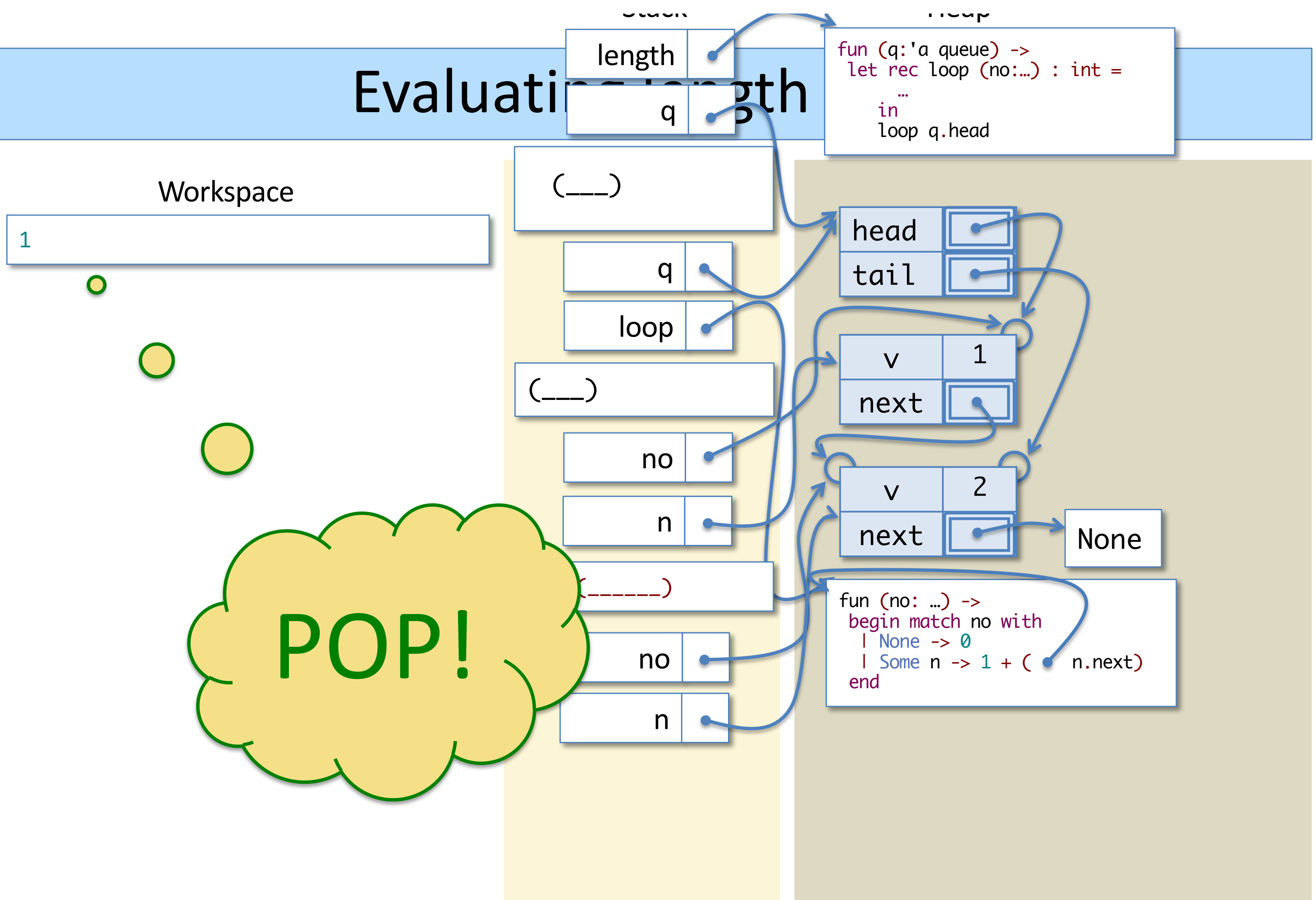
Evaluating length

Workspace

`1 + 0`



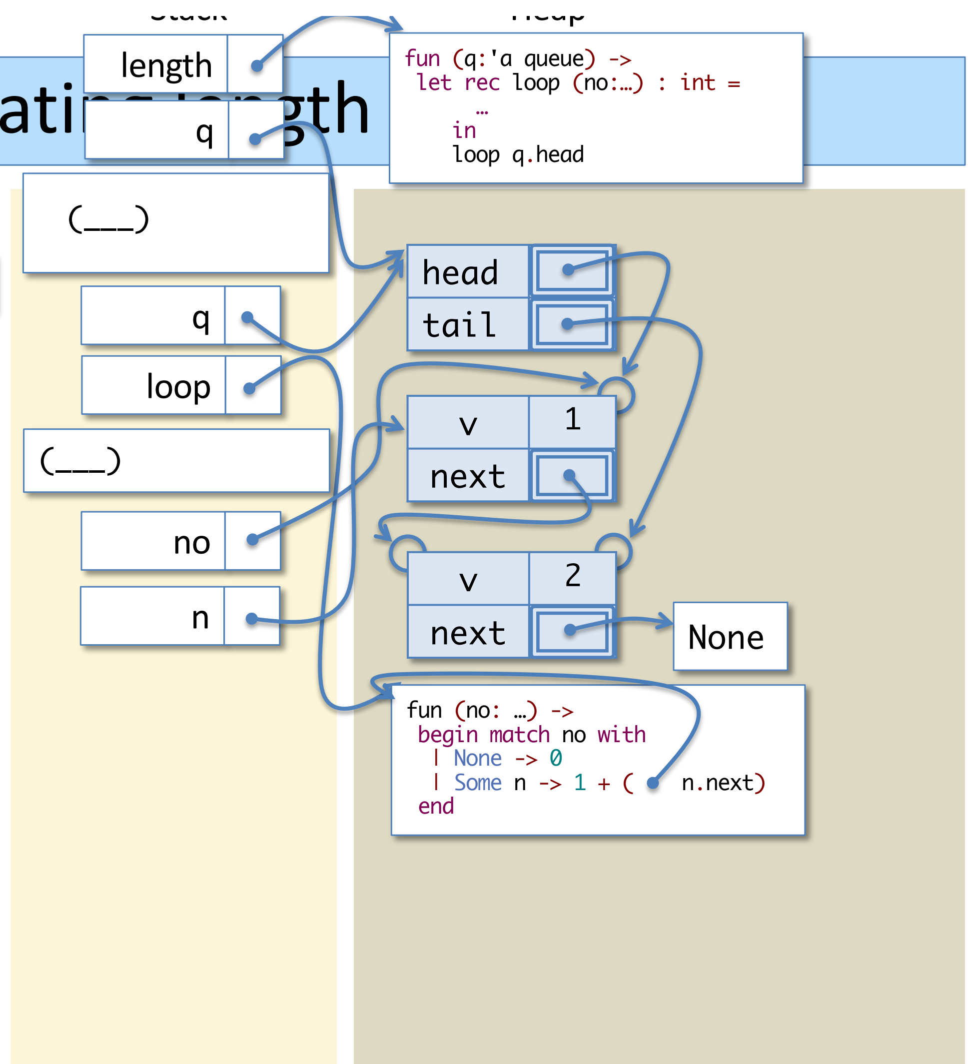
Evaluating length



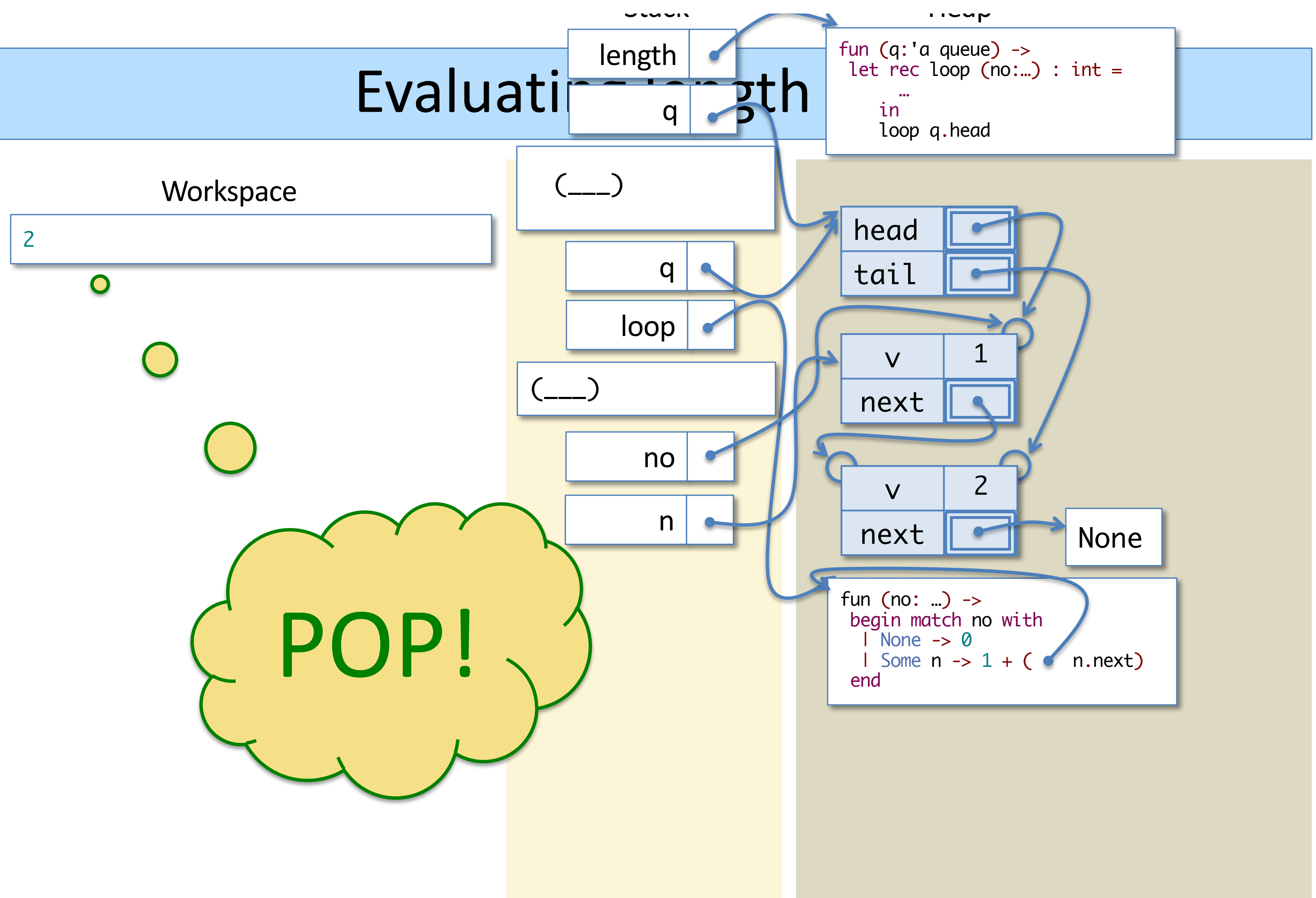
Evaluating length

Workspace

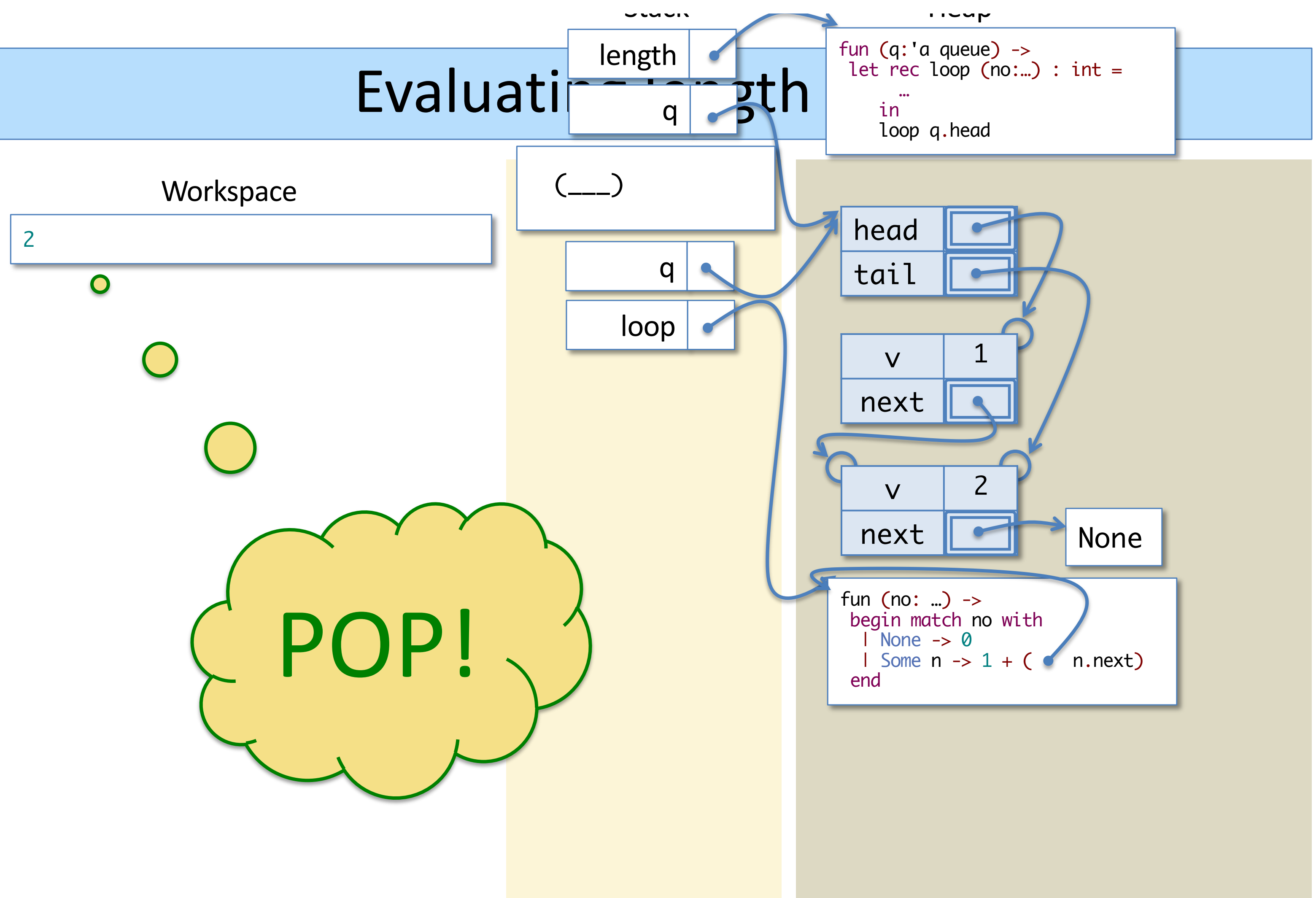
1 + 1



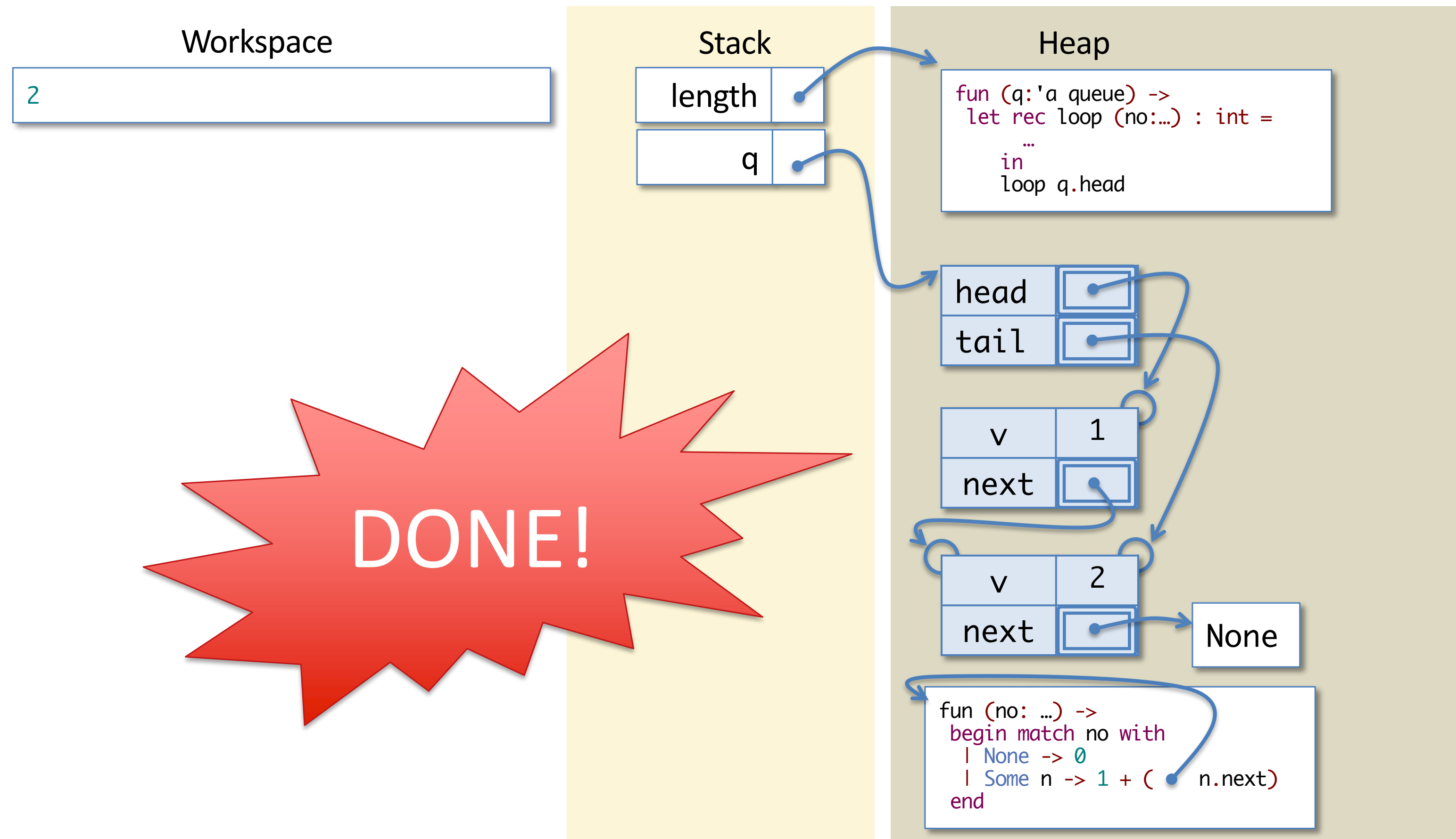
Evaluating length



Evaluating length



Evaluating length



Iteration

Using tail calls for loops

`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
```

As we've just seen, this implementation uses a lot of stack space if q is large.

Can we do better?

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
```

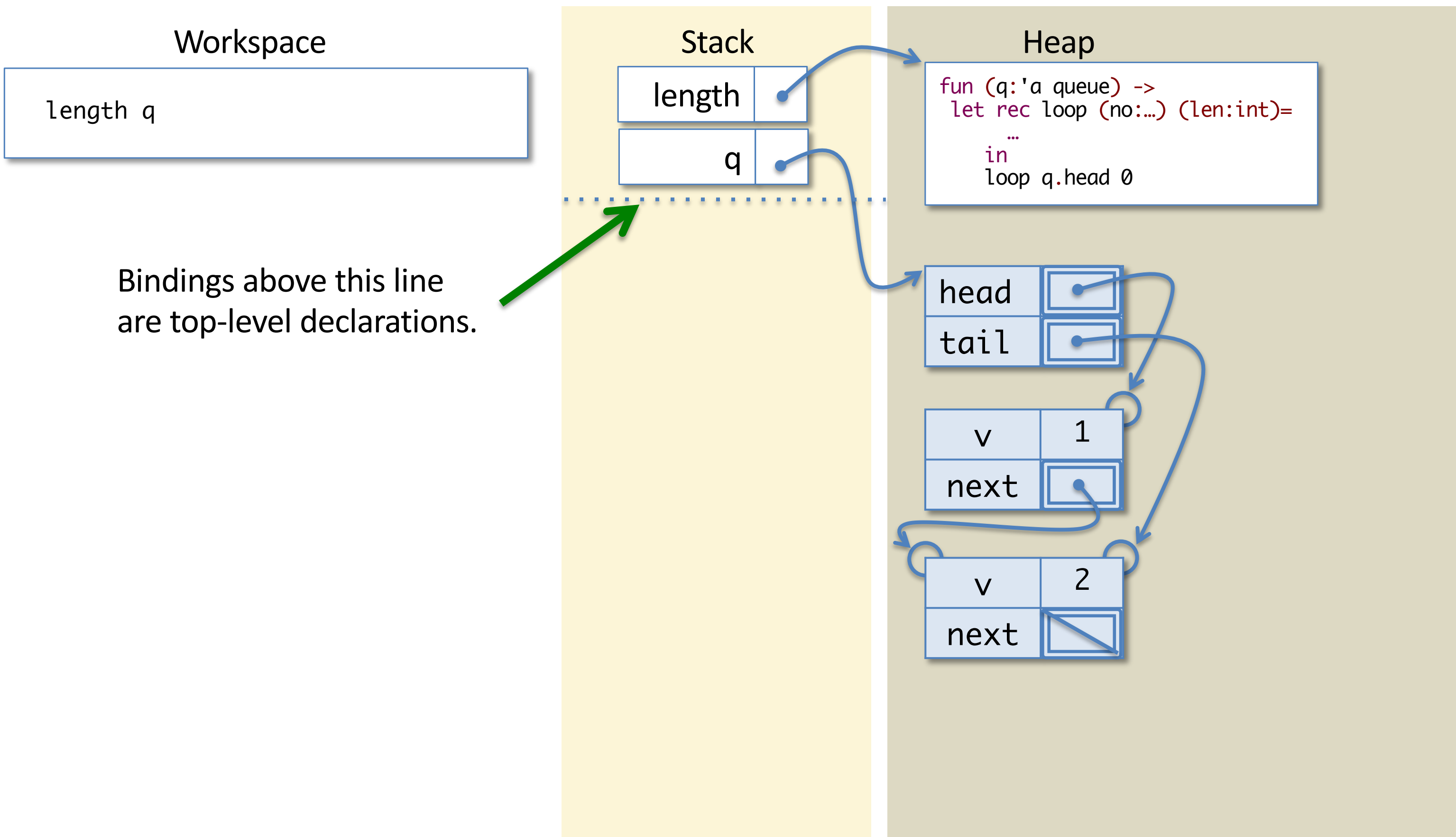
This implementation of length also uses a helper function, loop:

- 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.

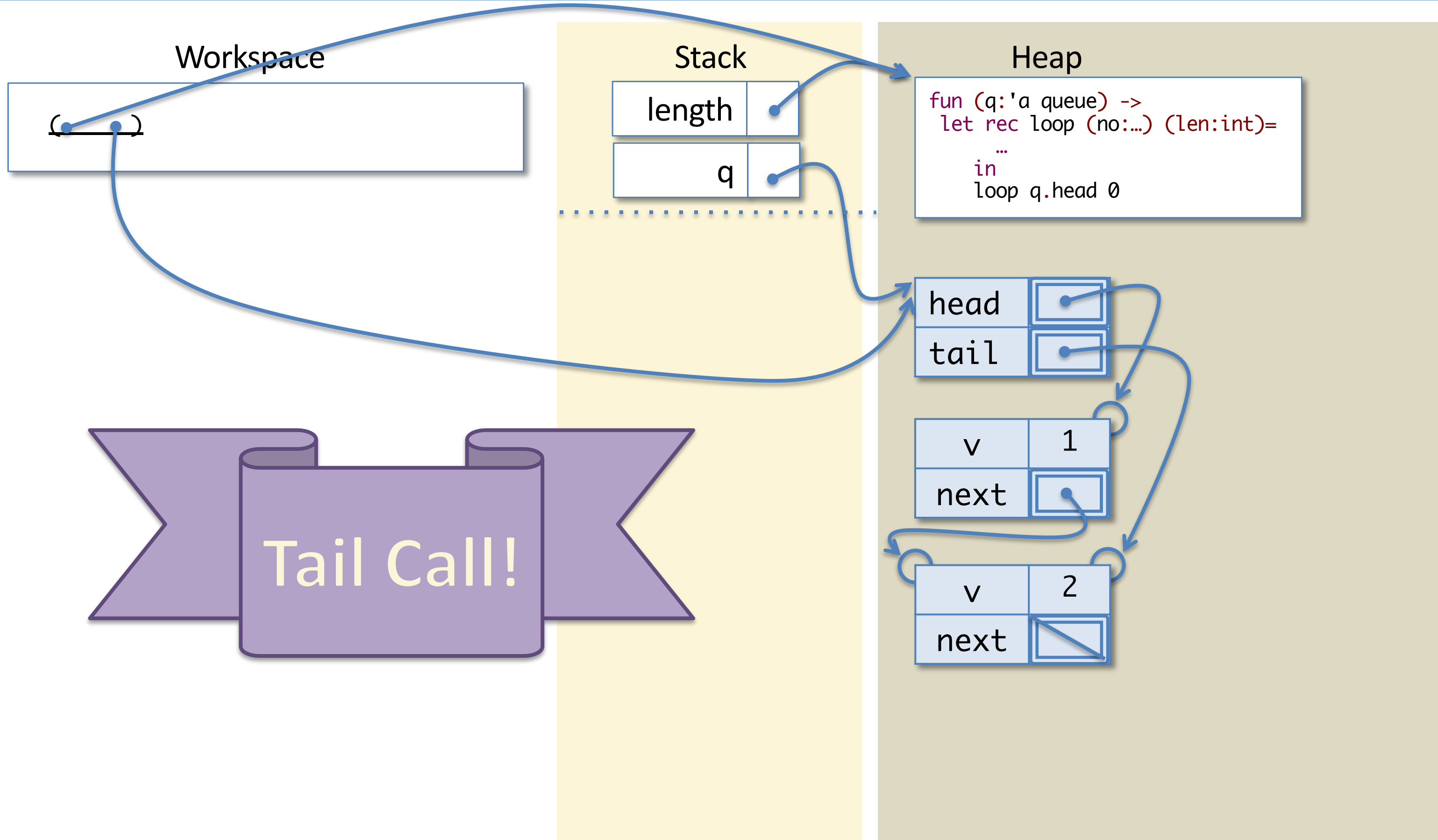
“Tail Call Optimization”

- Question: Why does it matter that ‘loop’ is only called in an empty workspace?
- Answer: We can *optimize* the behavior of the abstract stack machine in this case!
  - 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 using a *constant* amount of stack space, just like a ‘for’ loop in Java or C.

Tail Calls and Iterative length



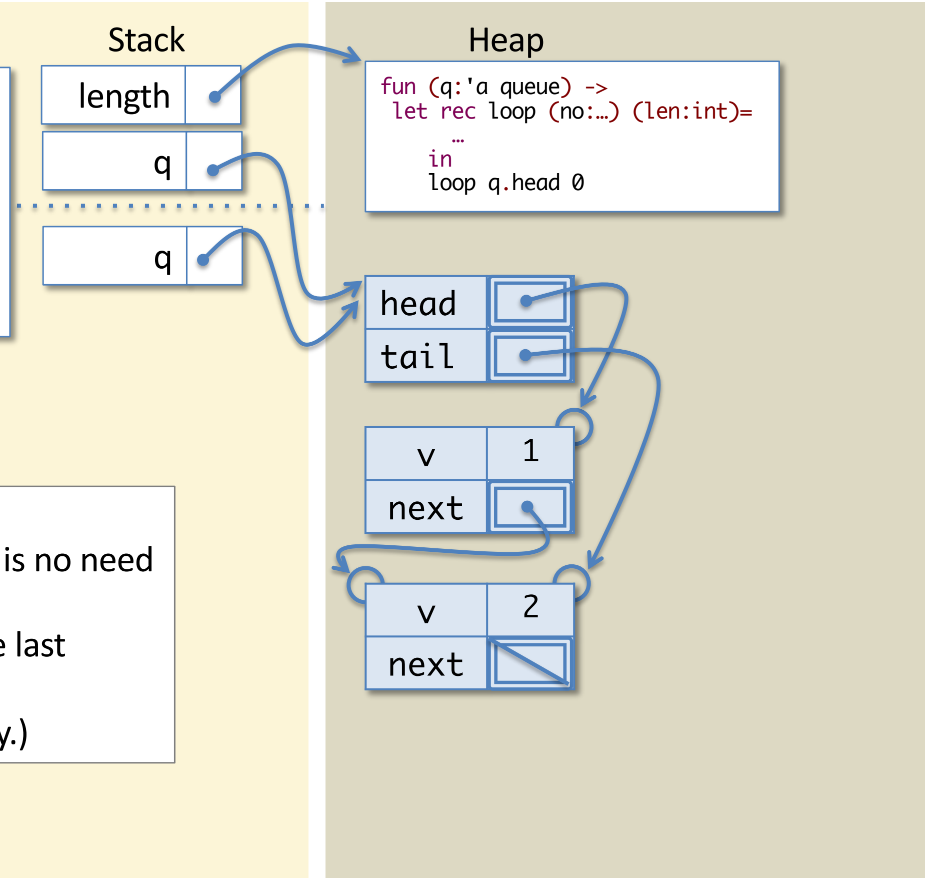
Tail Calls and Iterative length



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
```

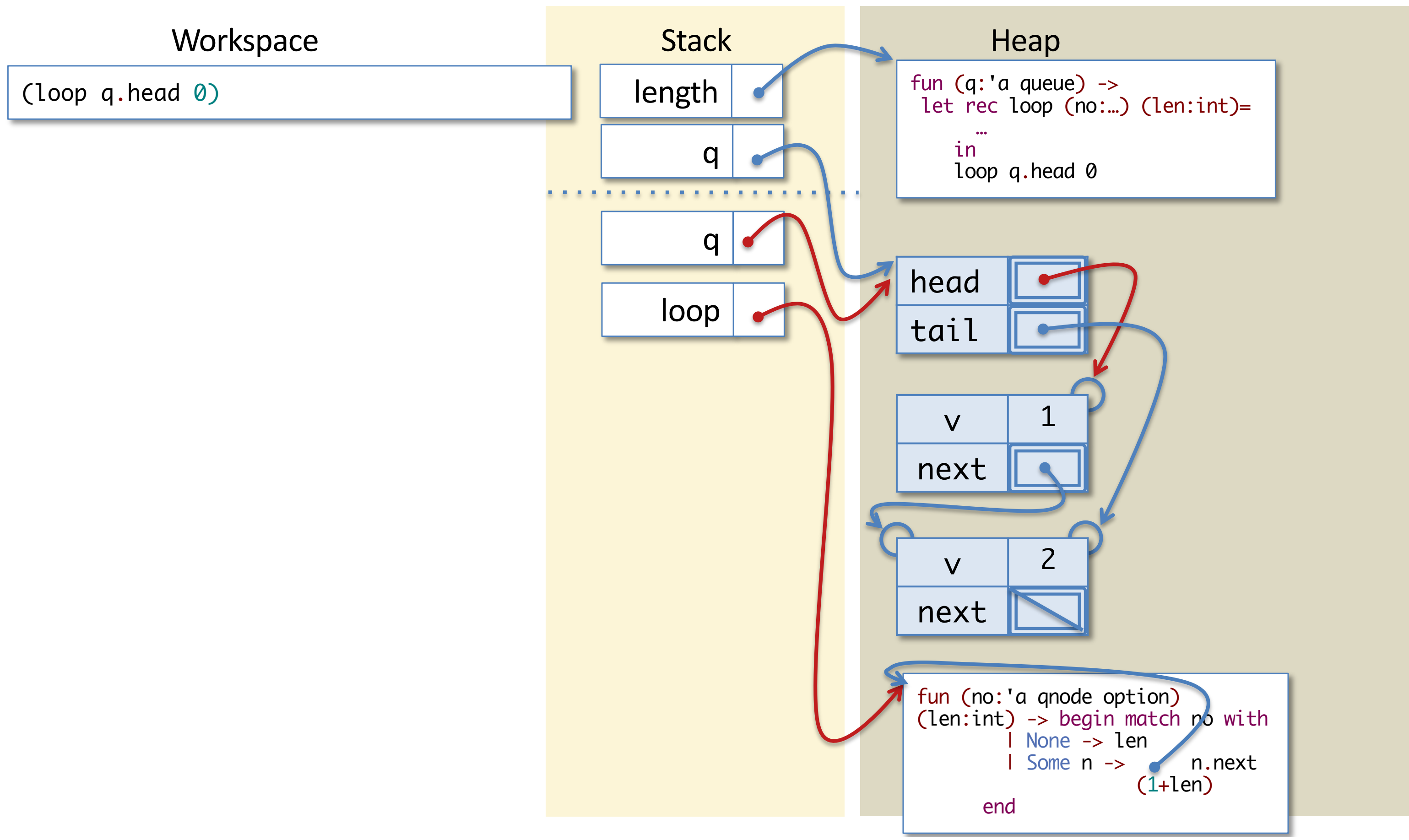


Note:

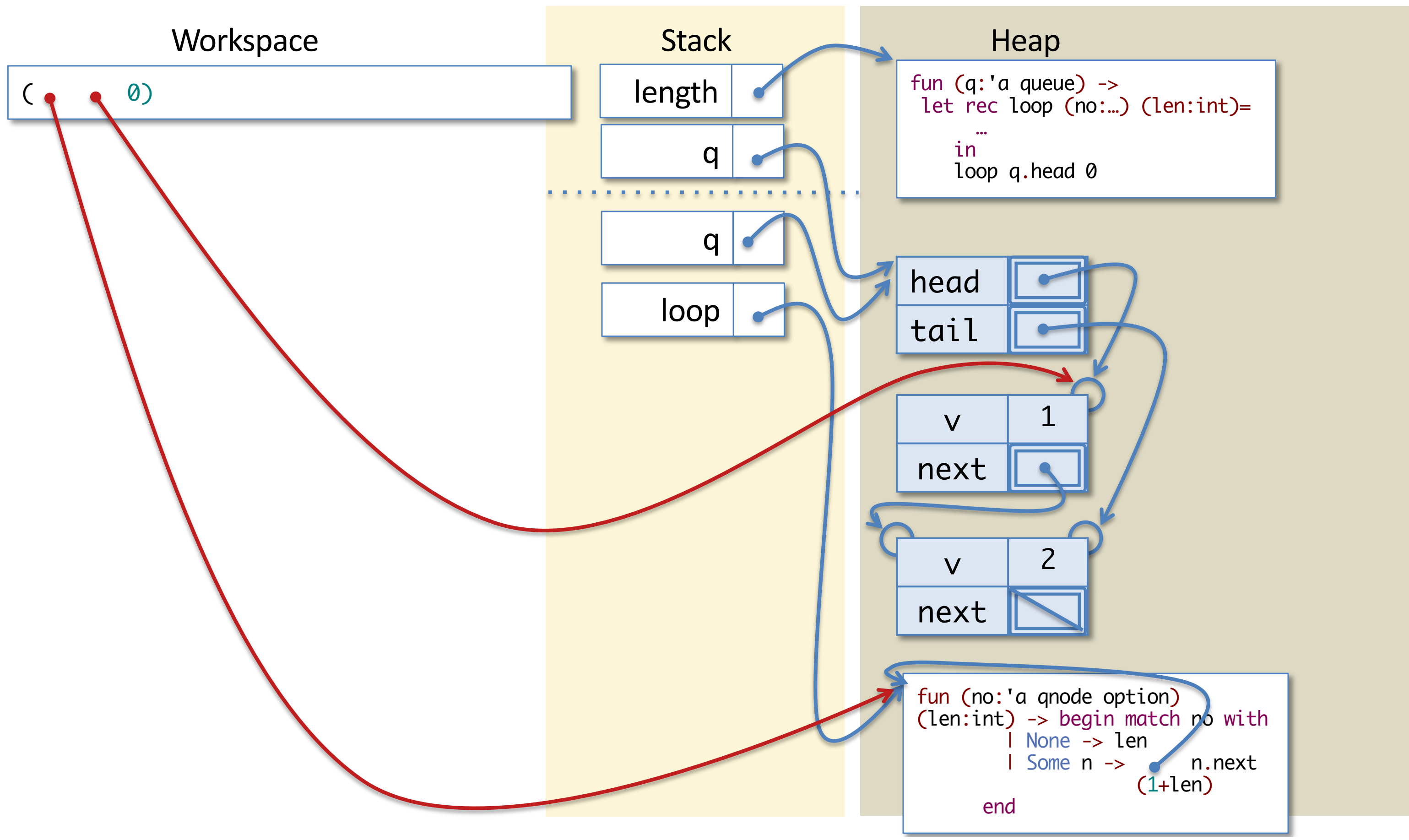
(1) No workspace is saved – there is no need do to that for tail calls

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

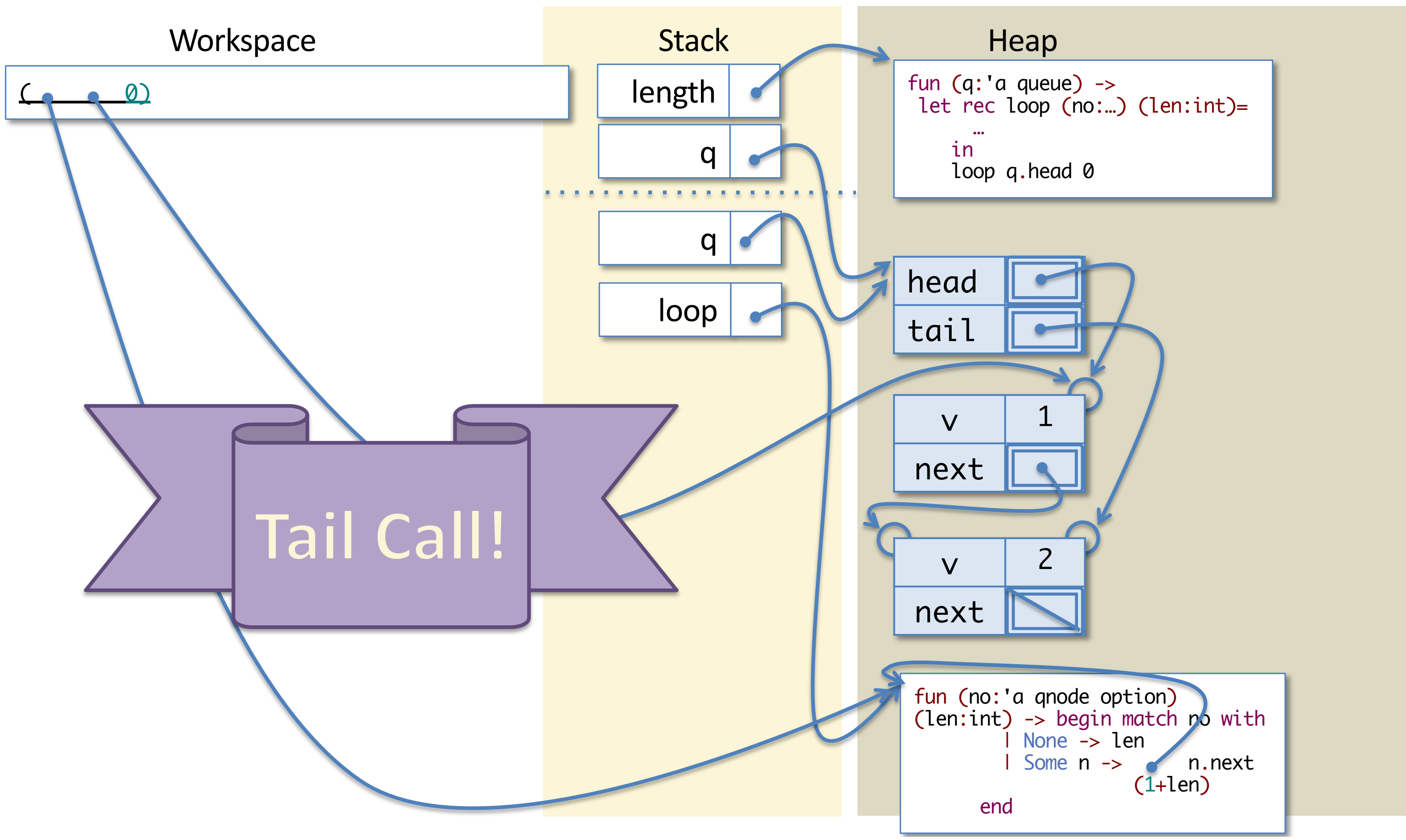
Tail Calls and Iterative length



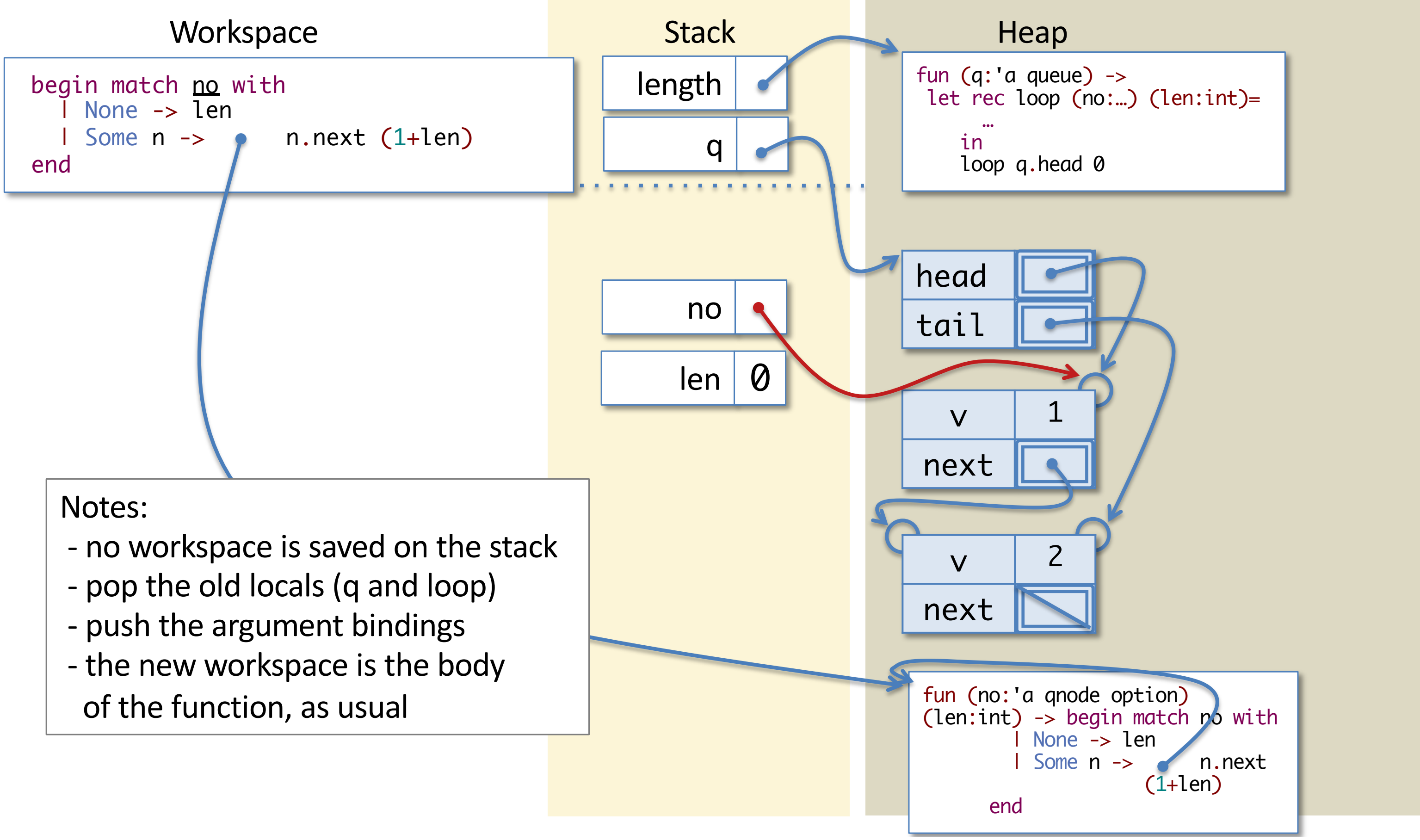
Tail Calls and Iterative length



Tail Calls and Iterative length

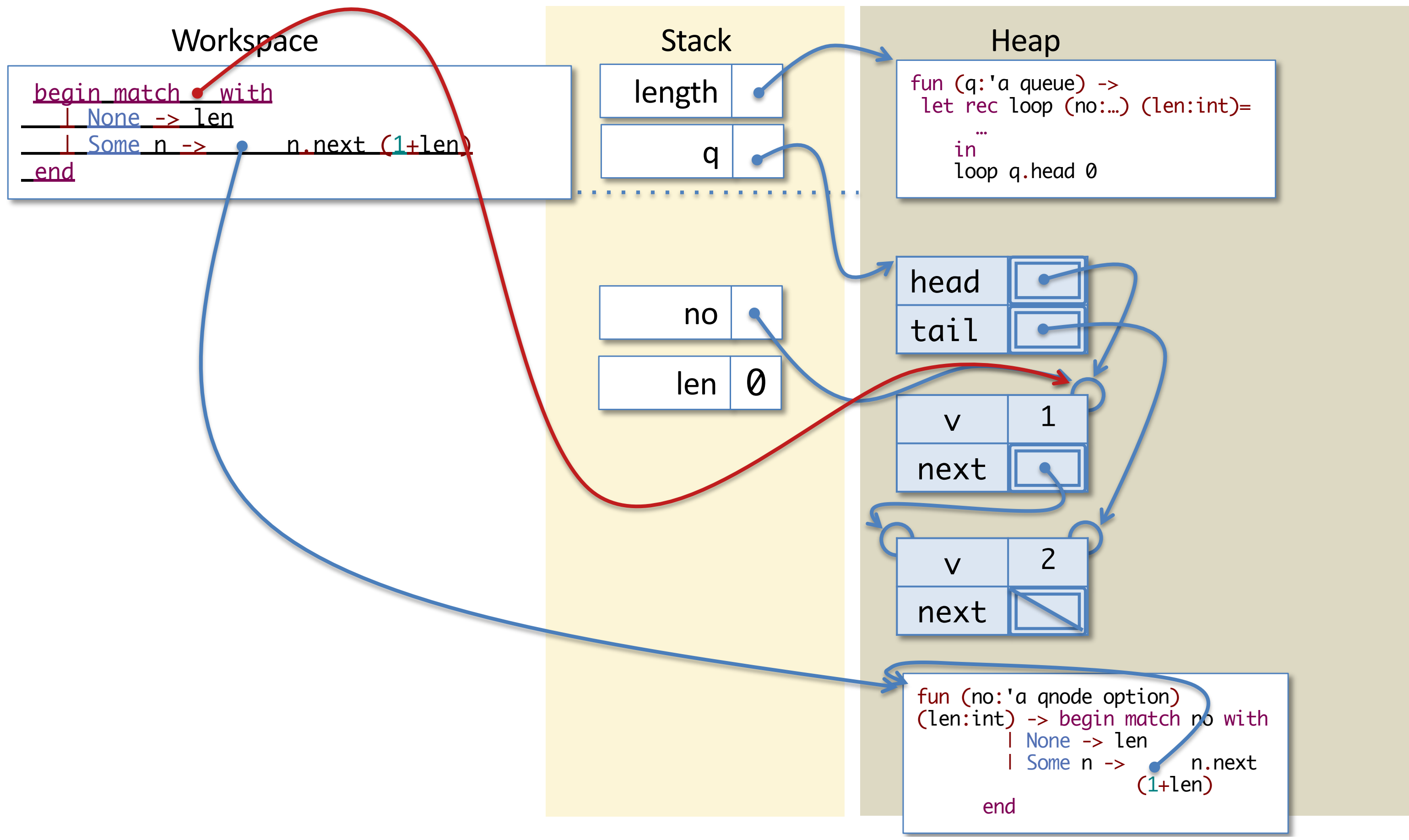


Tail Calls and Iterative length

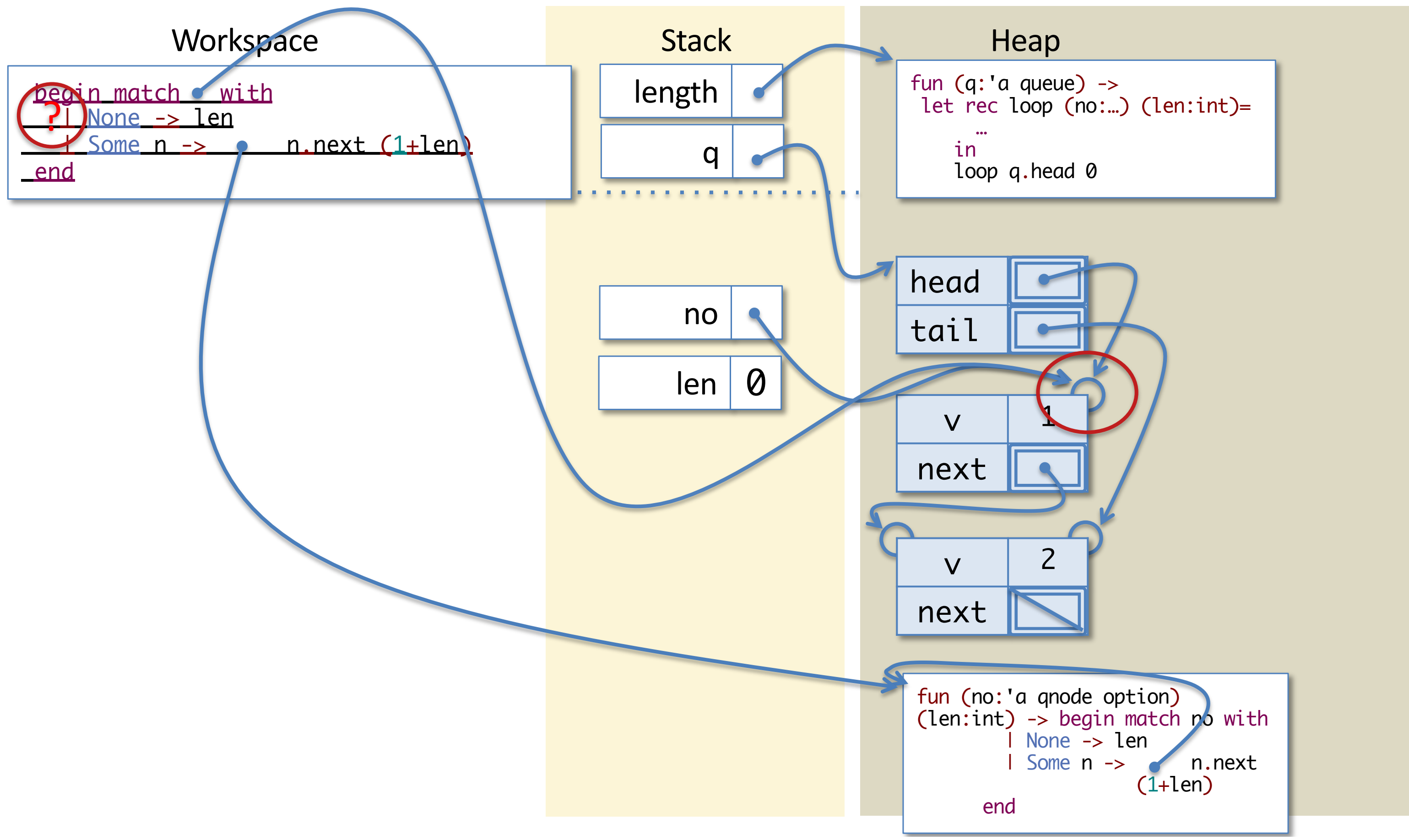


Notes:
- no workspace is saved on the stack
- pop the old locals (q and loop)
- push the argument bindings
- the new workspace is the body of the function, as usual

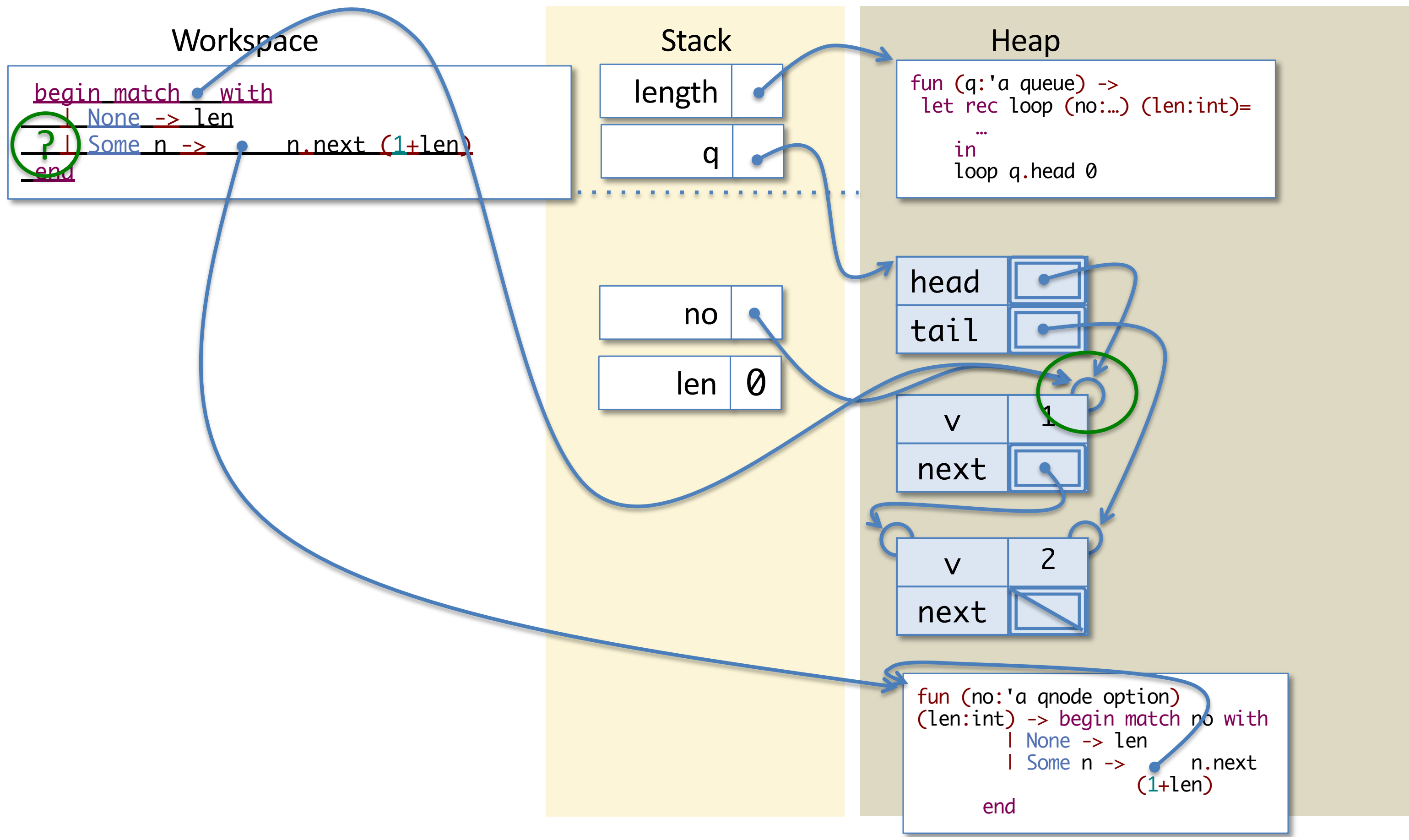
Tail Calls and Iterative length



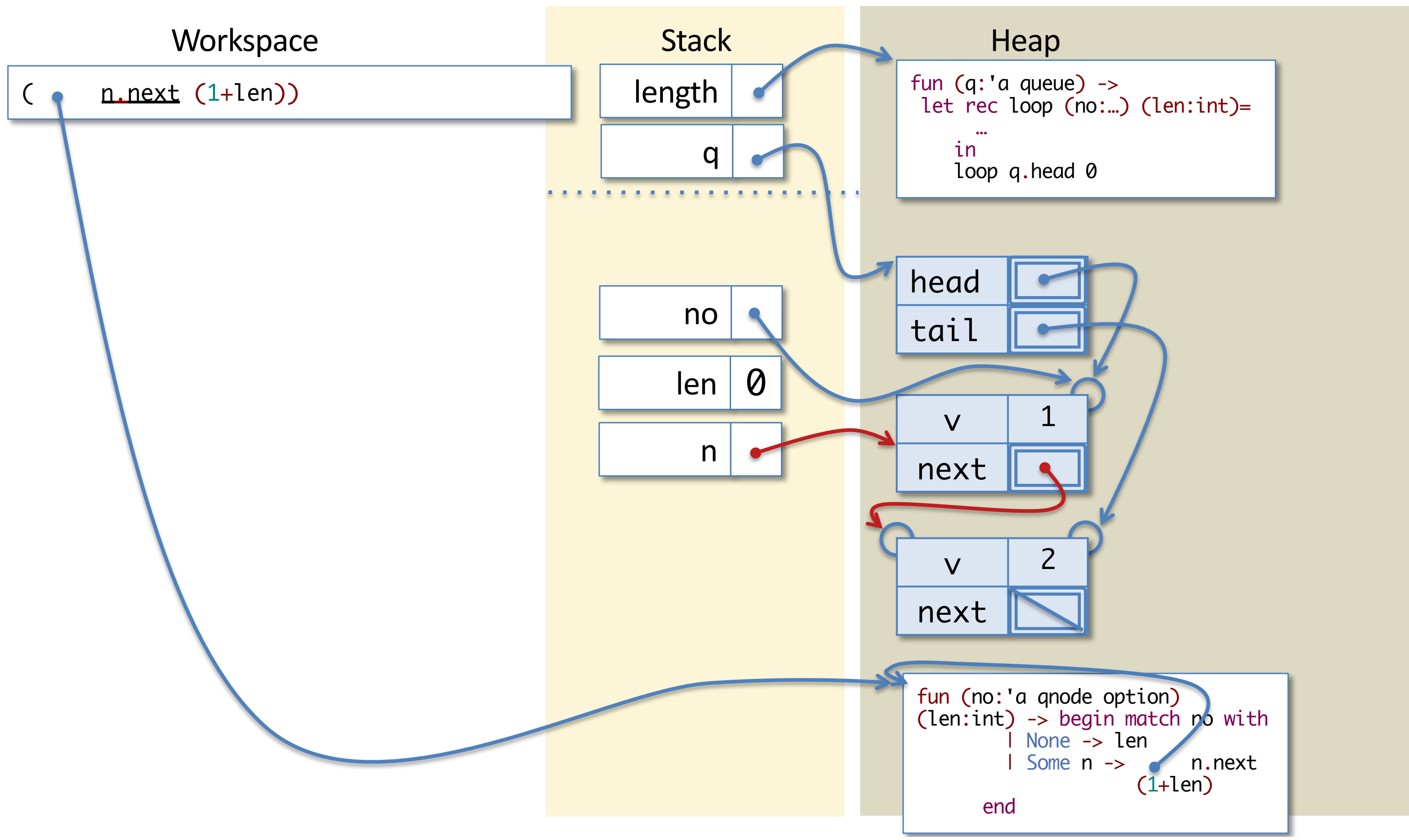
Tail Calls and Iterative length



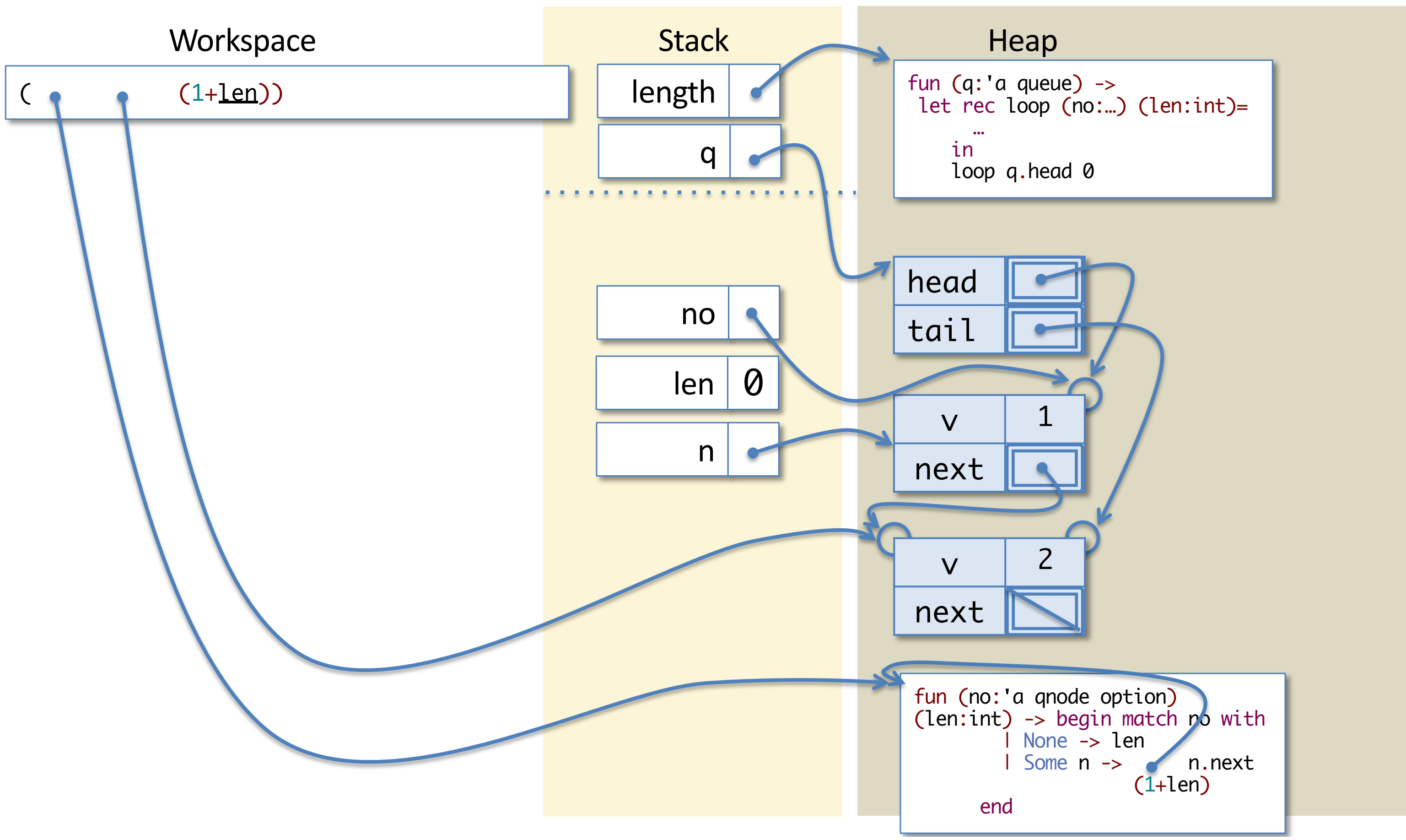
Tail Calls and Iterative length



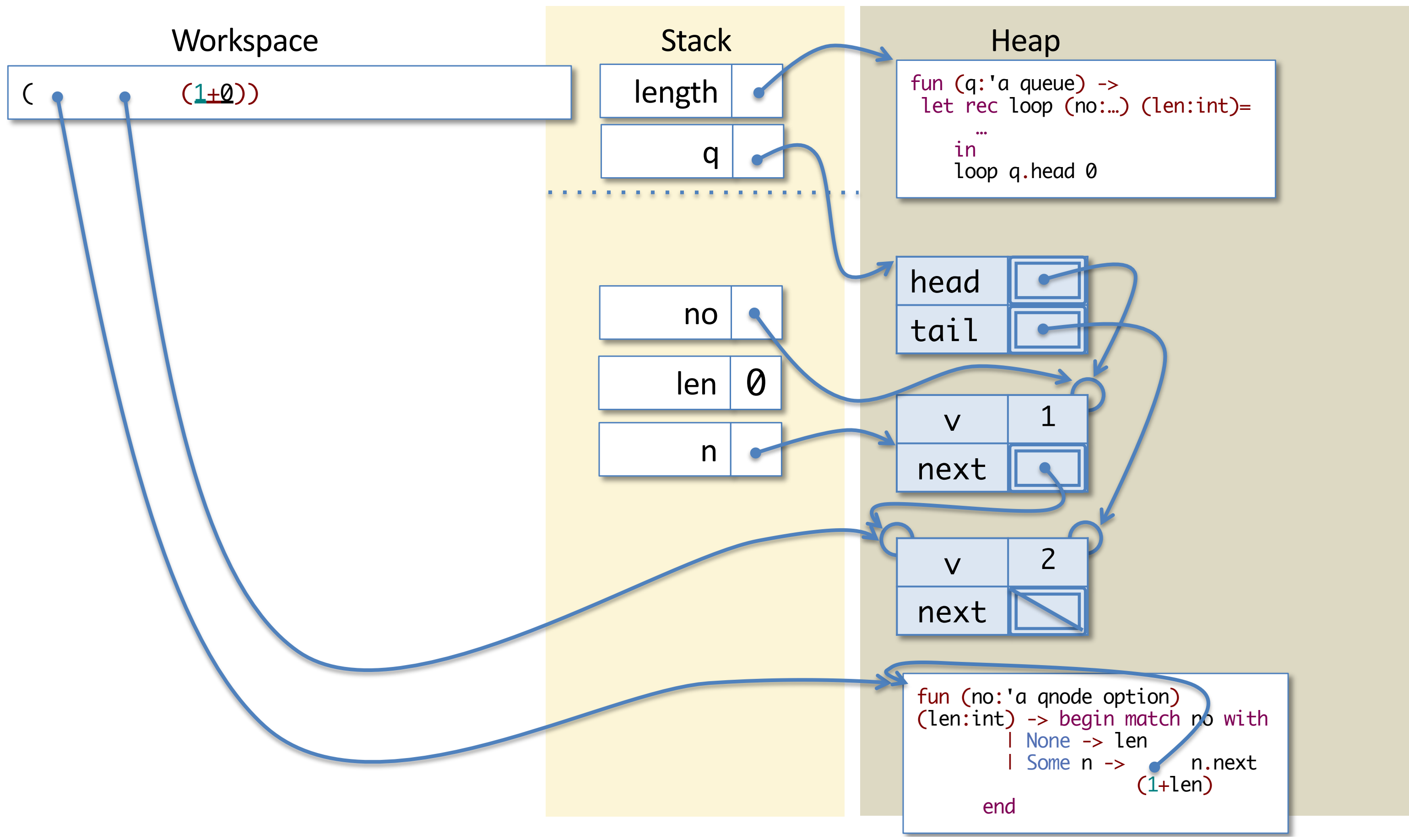
Tail Calls and Iterative length



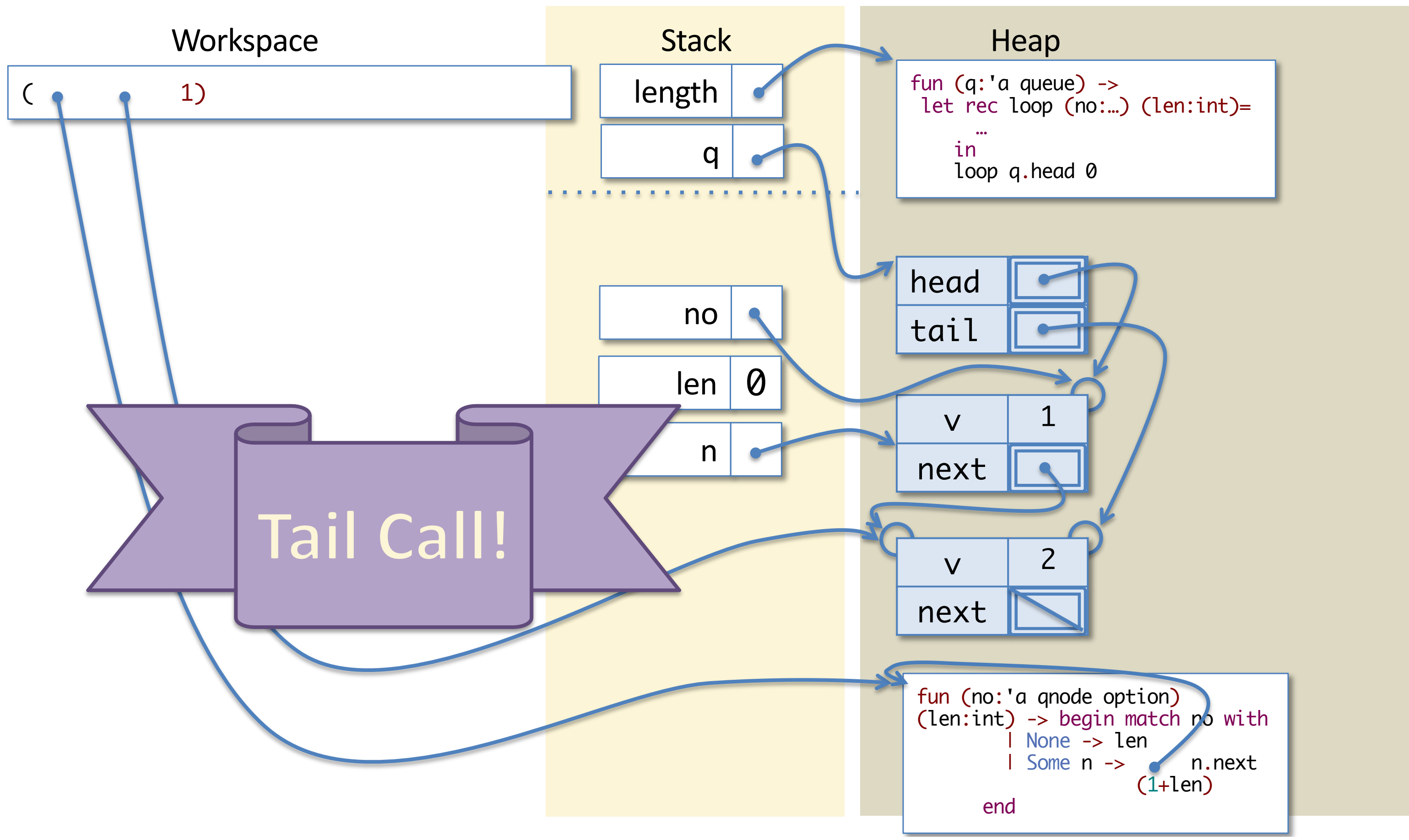
Tail Calls and Iterative length



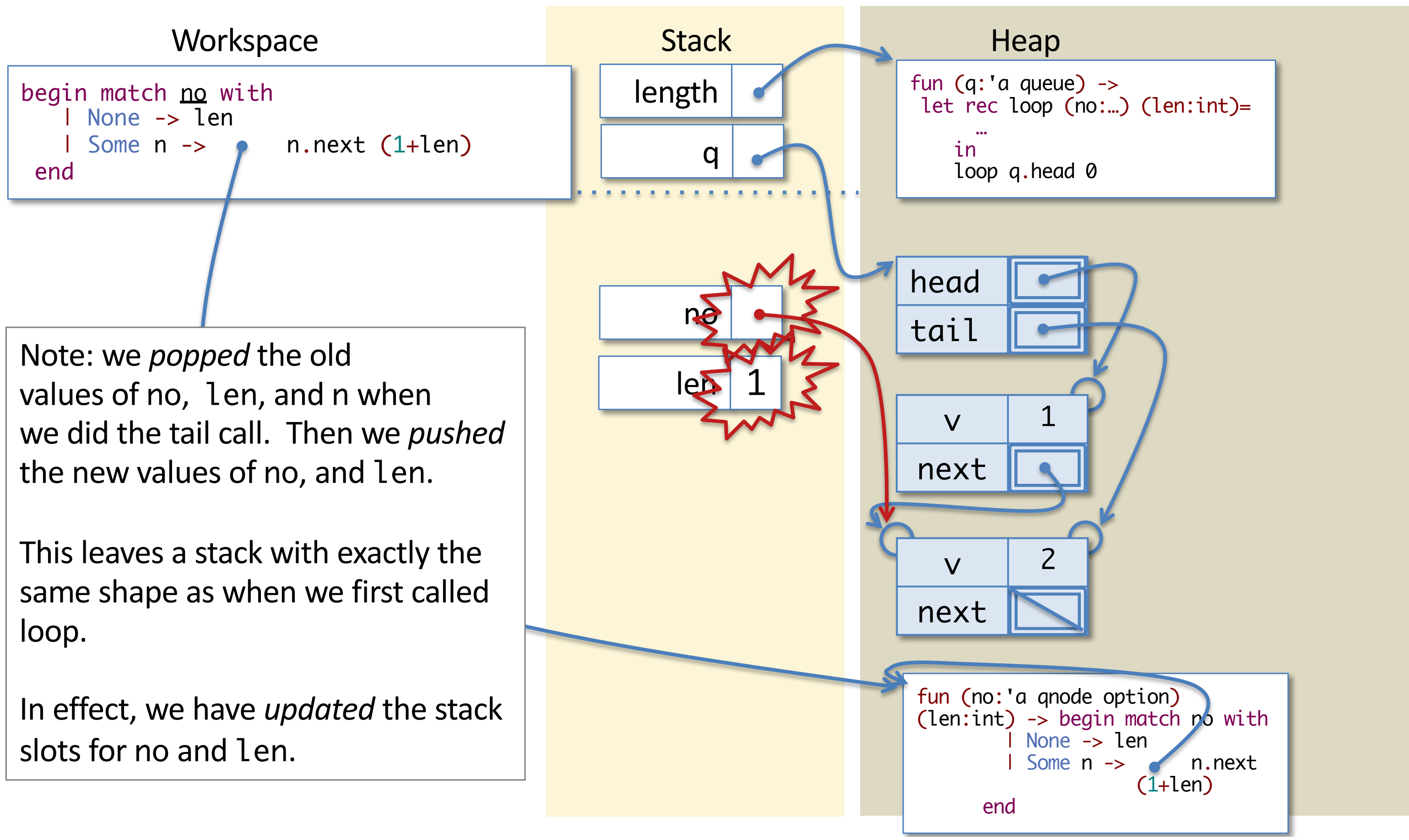
Tail Calls and Iterative length



Tail Calls and Iterative length



Tail Calls and Iterative length

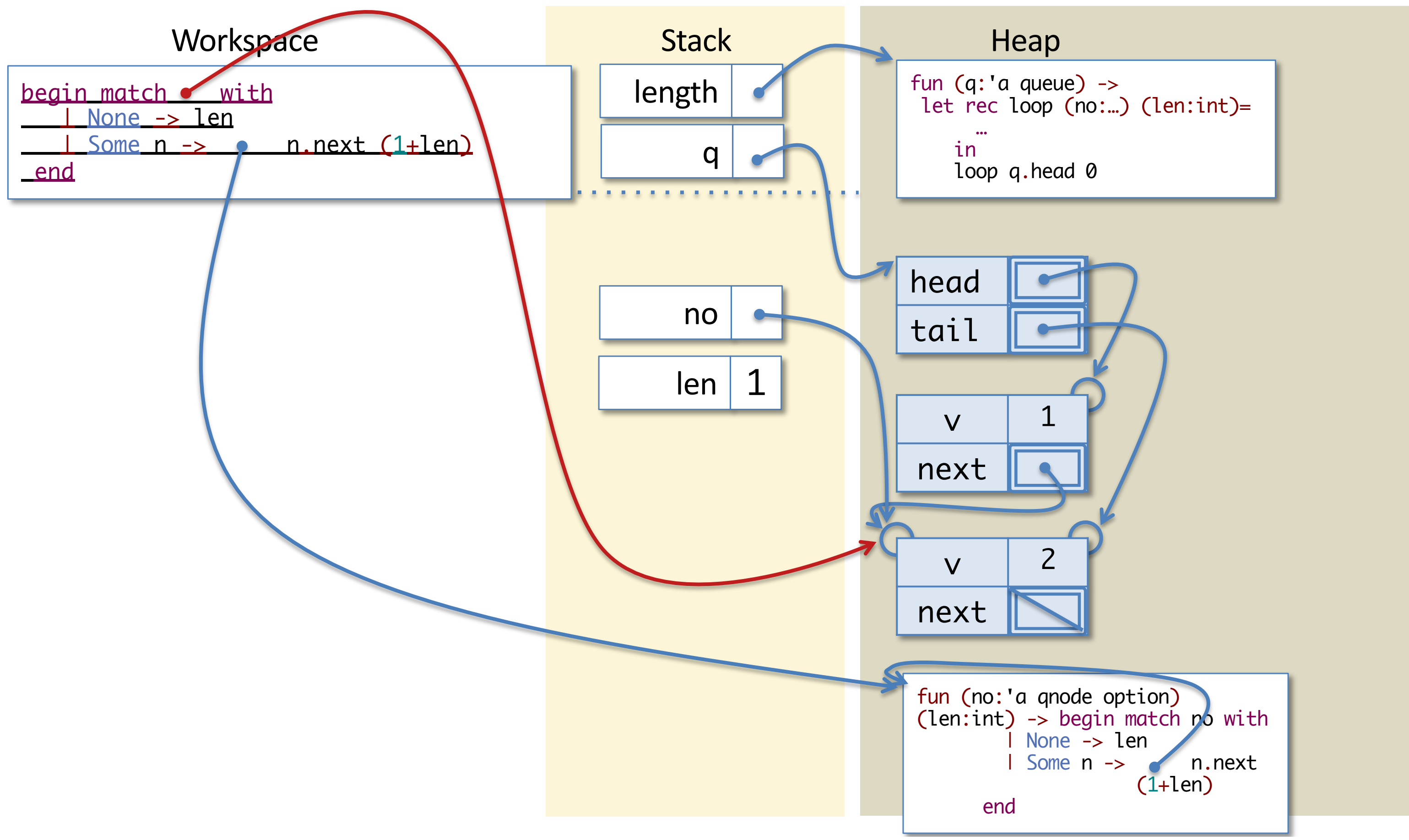


Note: we *popped* the old values of no, `len`, and n when we did the tail call. Then we *pushed* the new values of no, and `len`.

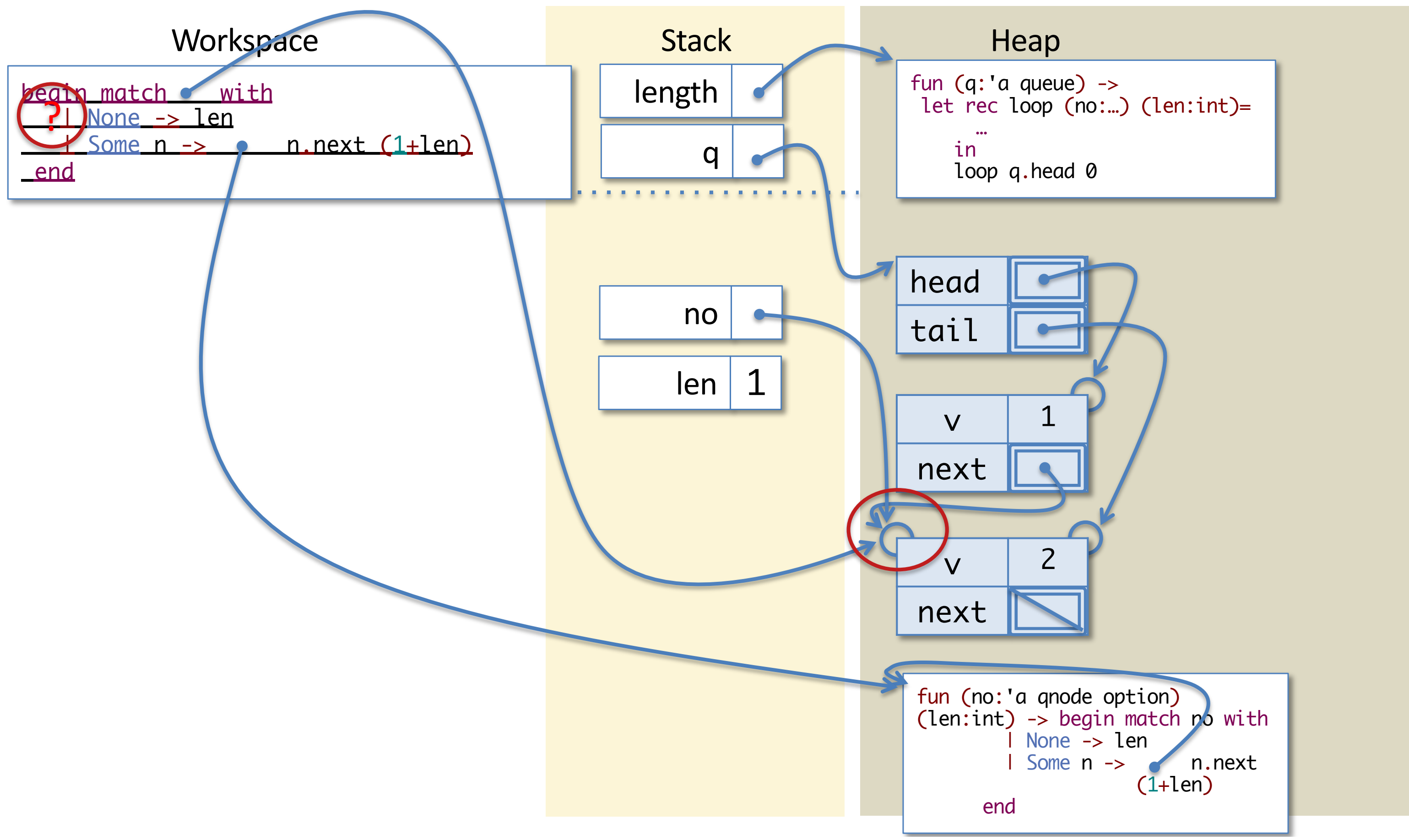
This leaves a stack with exactly the same shape as when we first called loop.

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

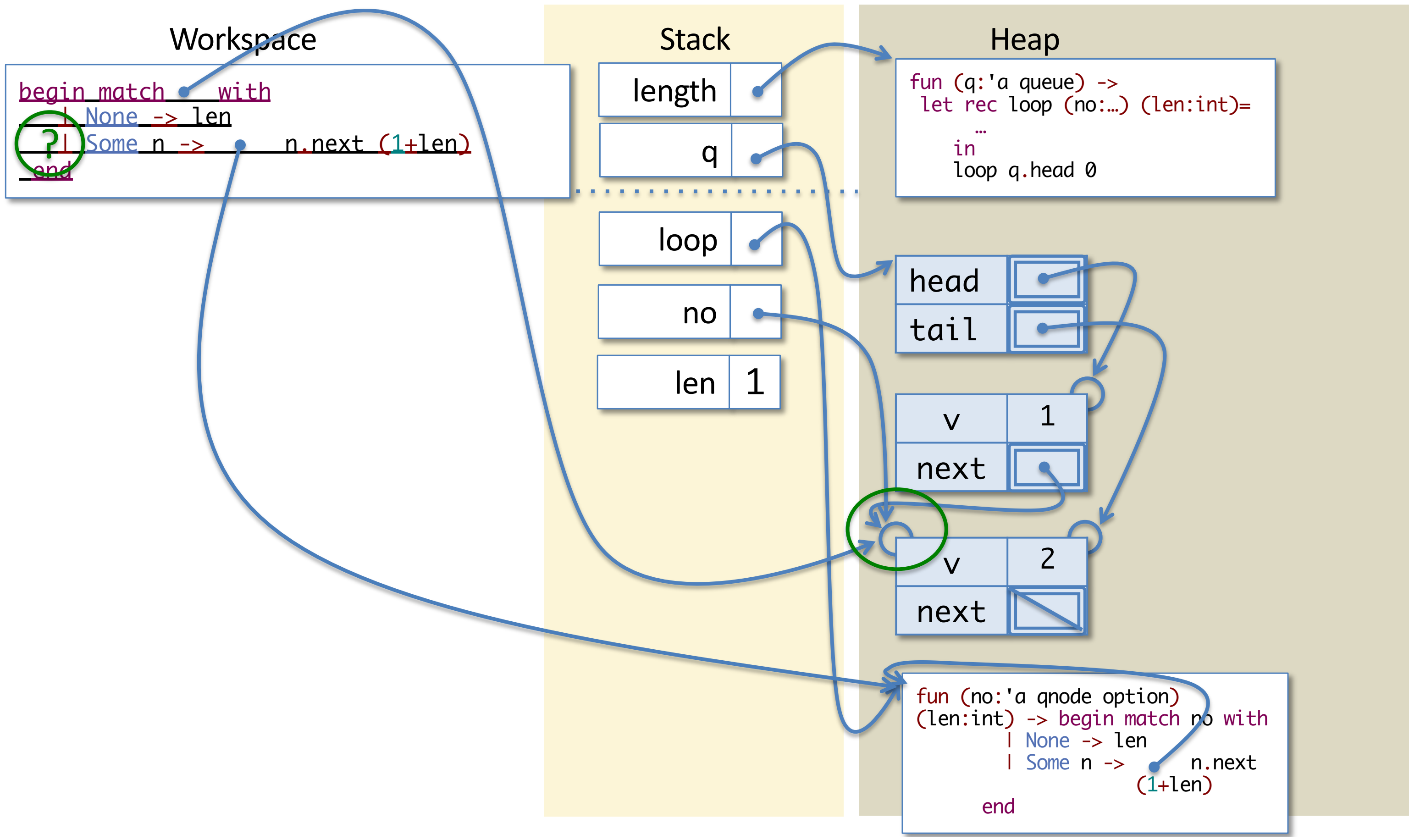
Tail Calls and Iterative length



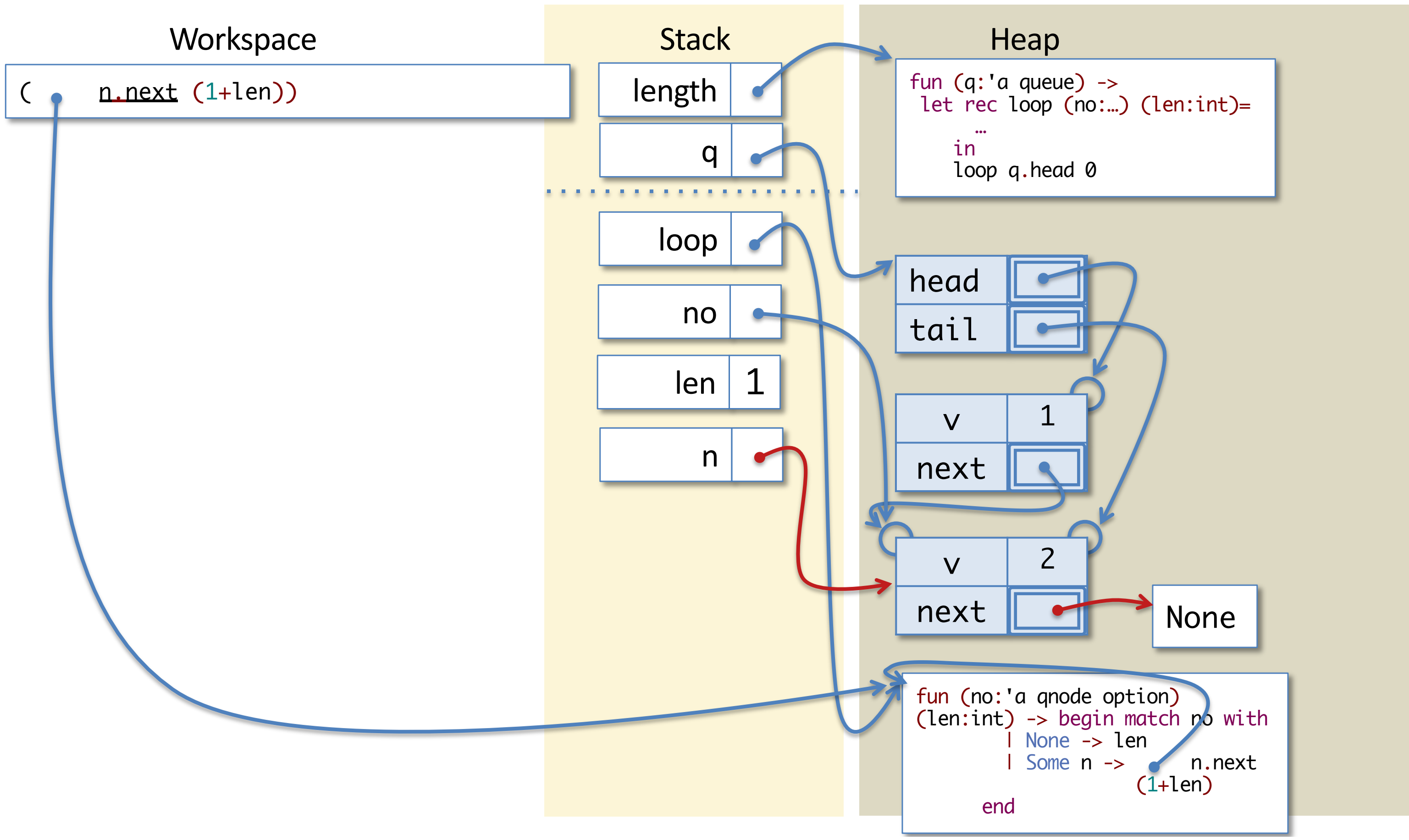
Tail Calls and Iterative length



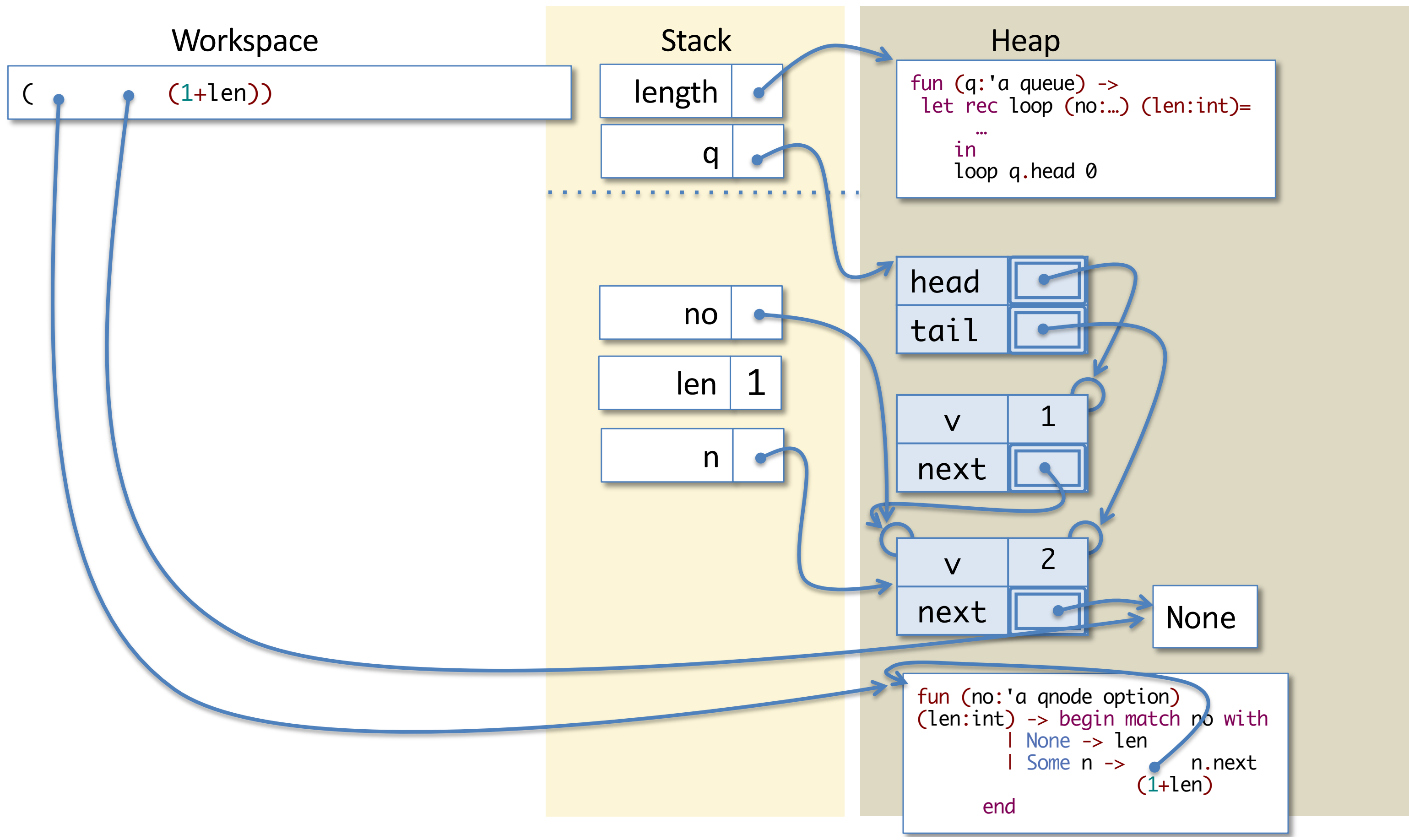
Tail Calls and Iterative length



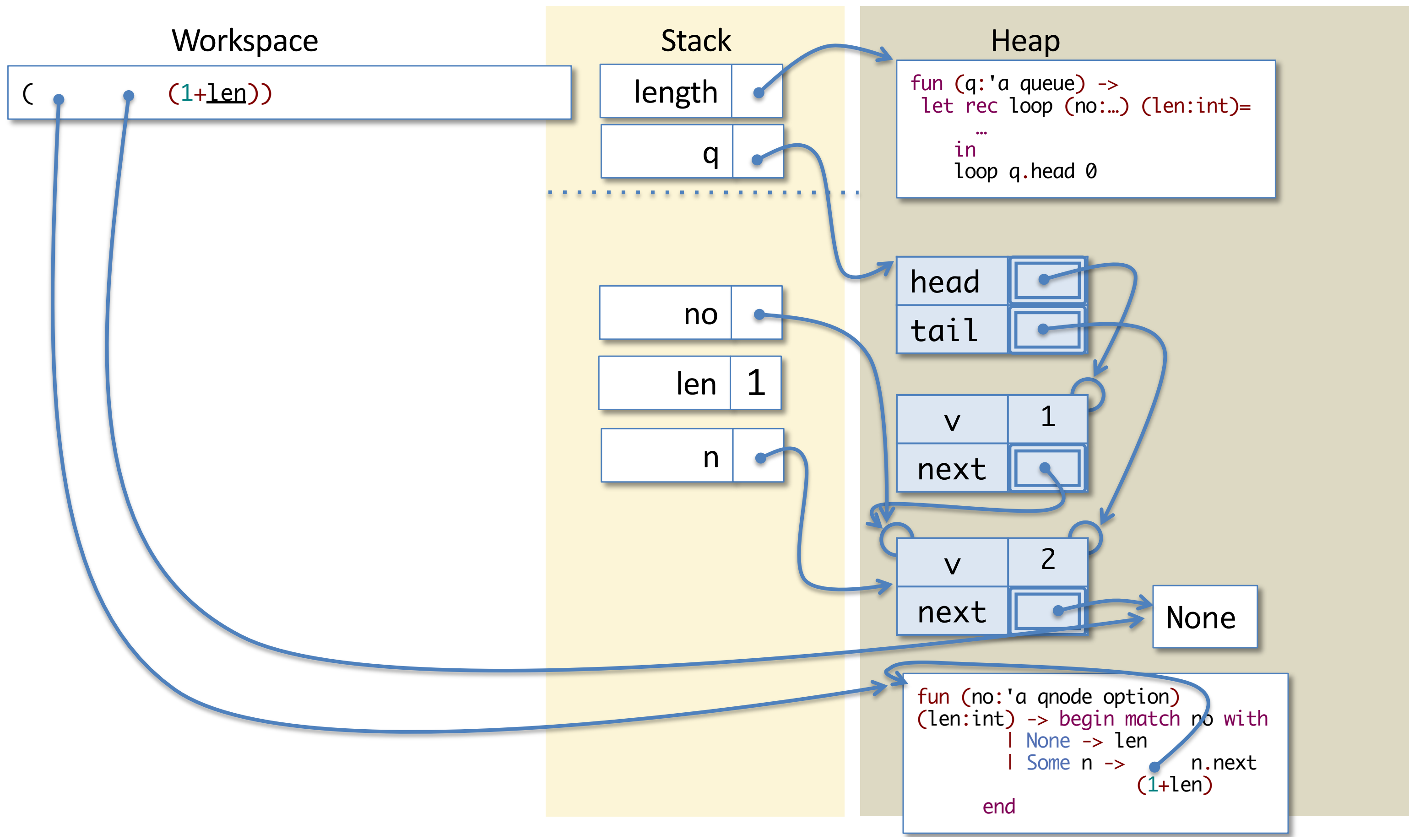
Tail Calls and Iterative length



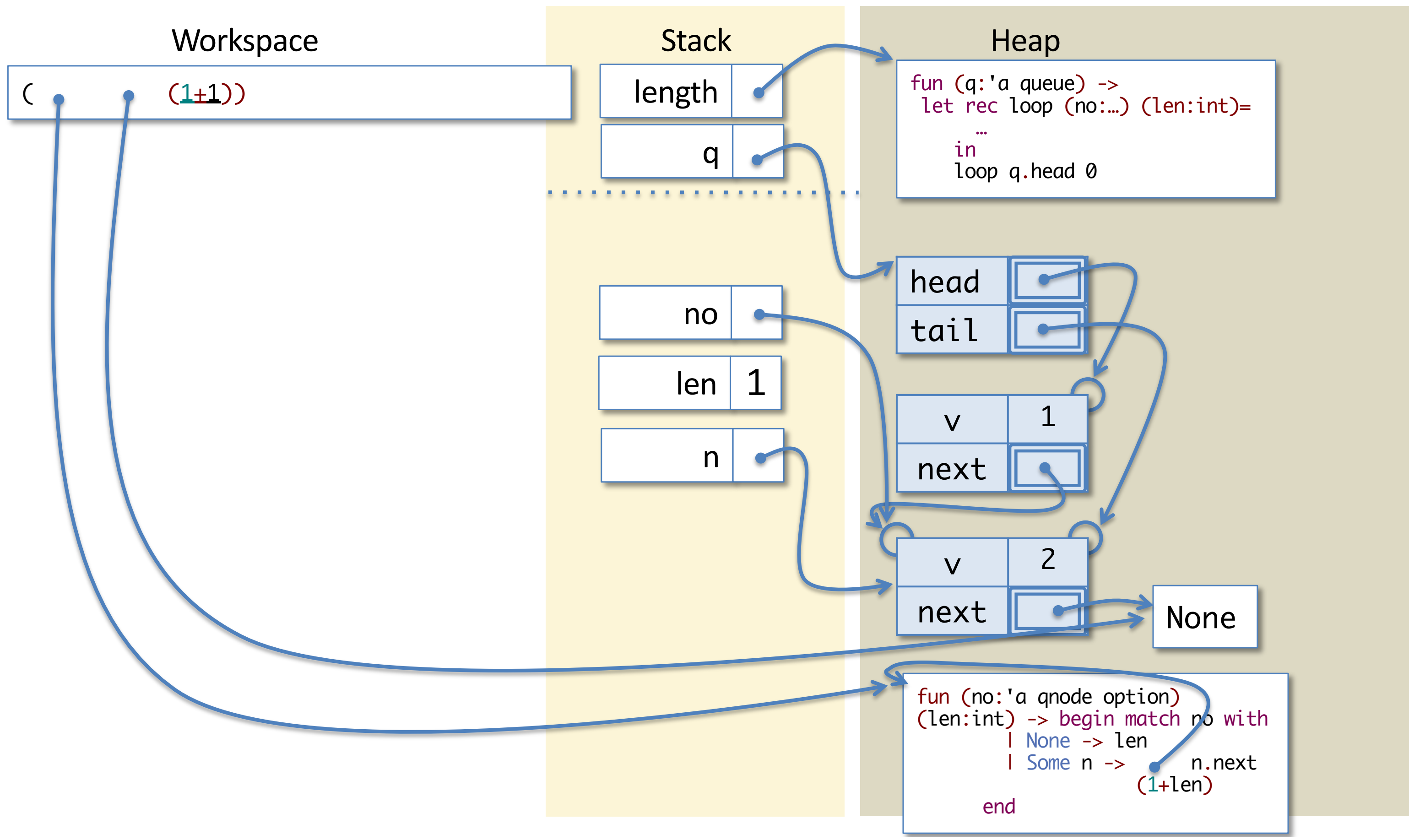
Tail Calls and Iterative length



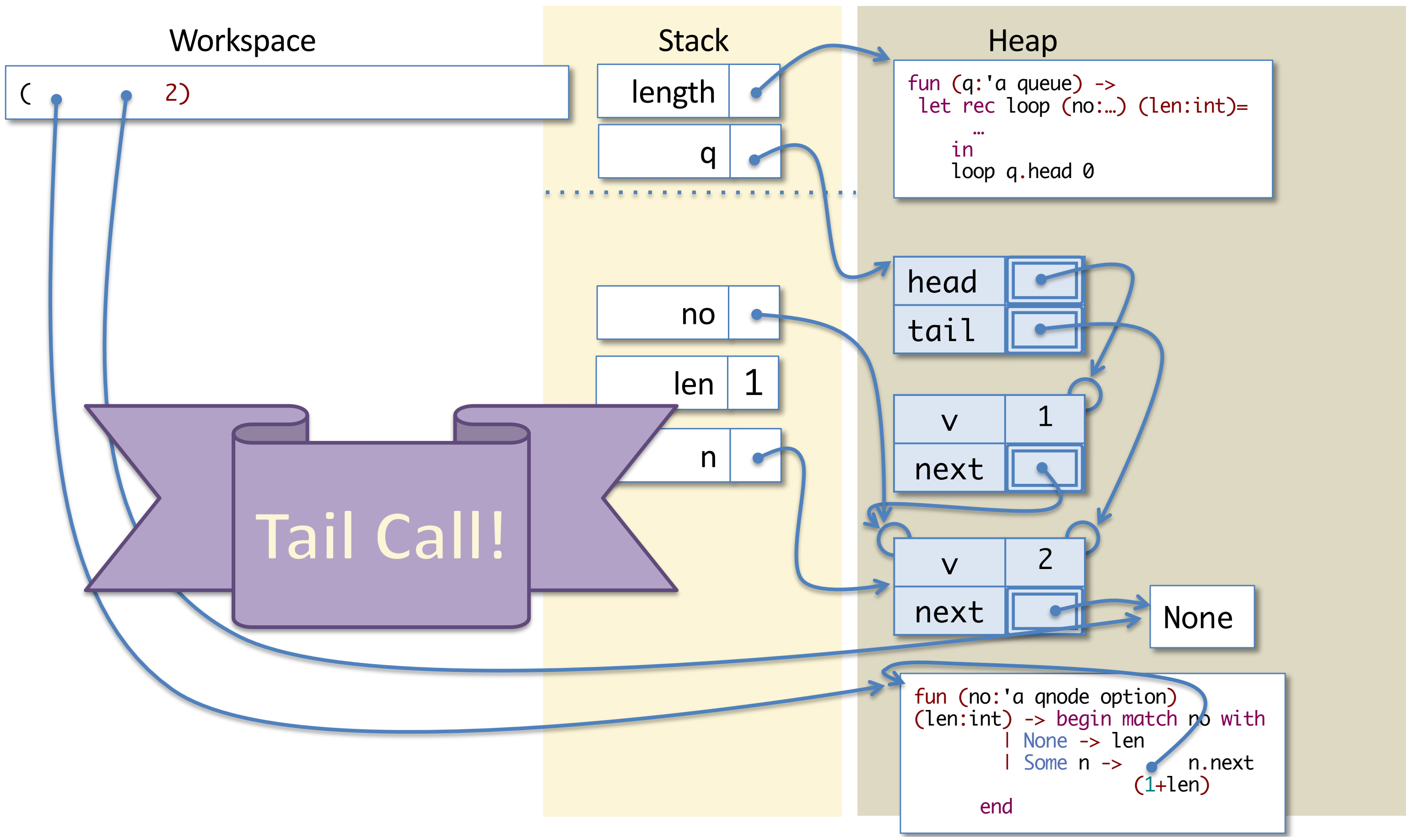
Tail Calls and Iterative length



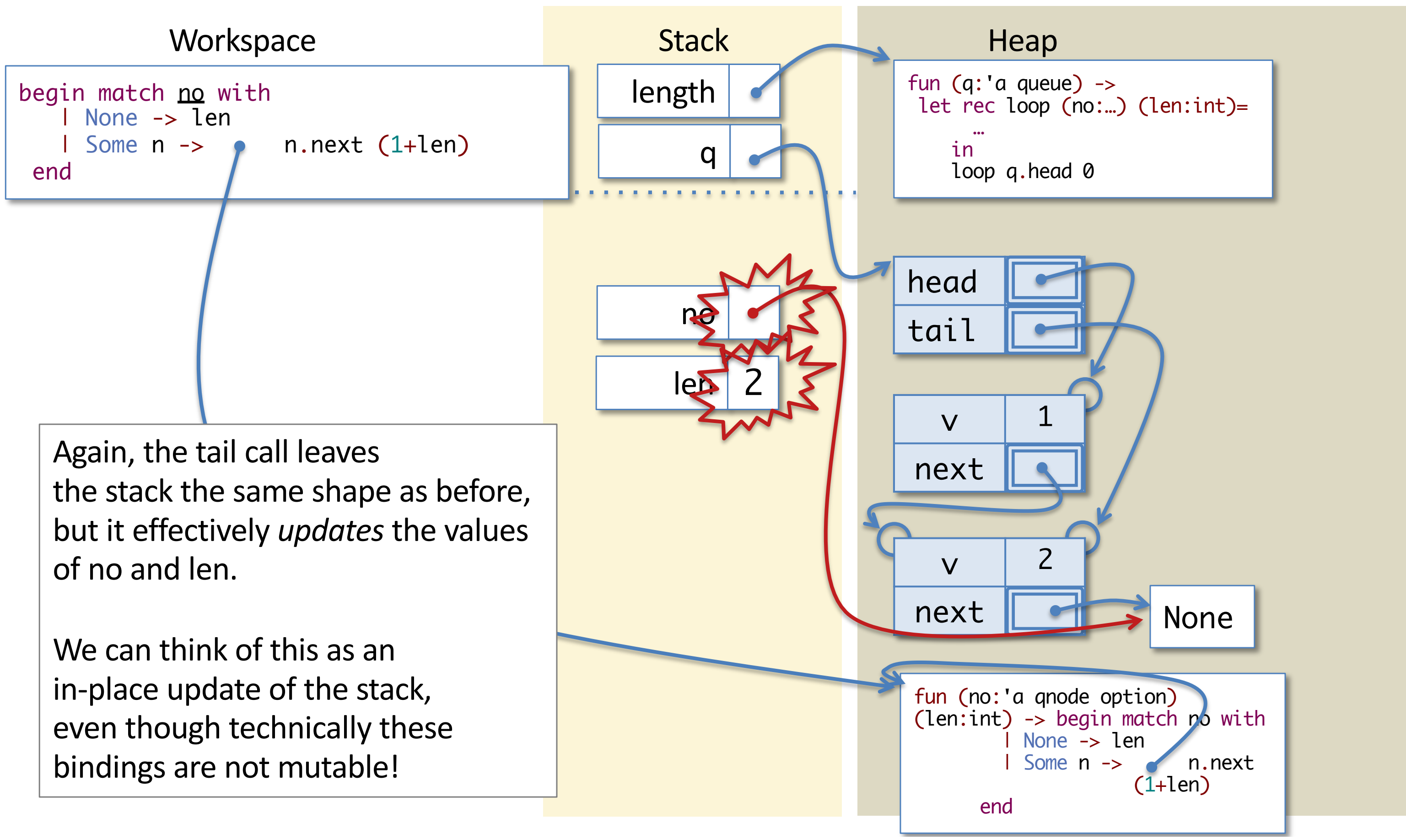
Tail Calls and Iterative length



Tail Calls and Iterative length



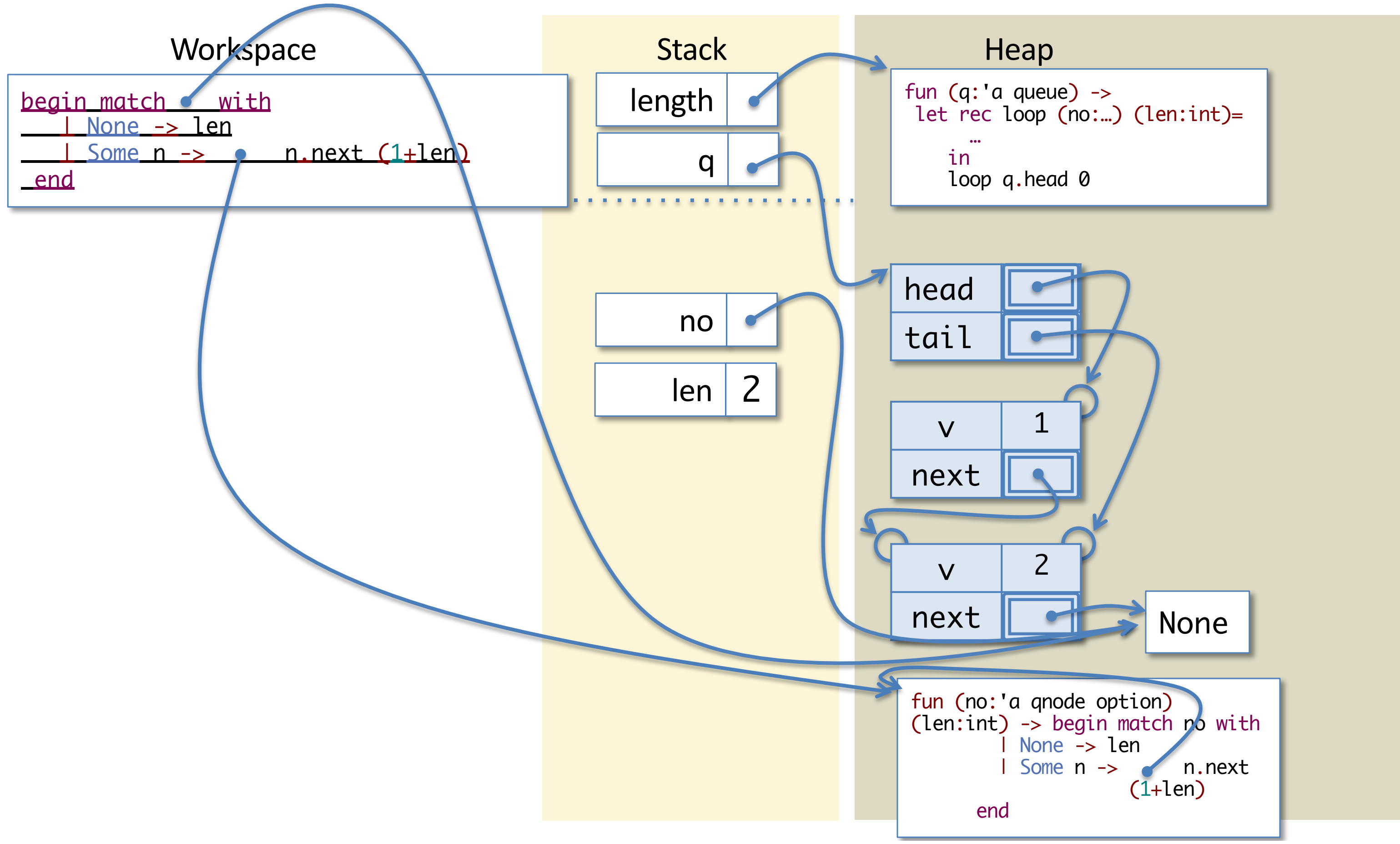
Tail Calls and Iterative length



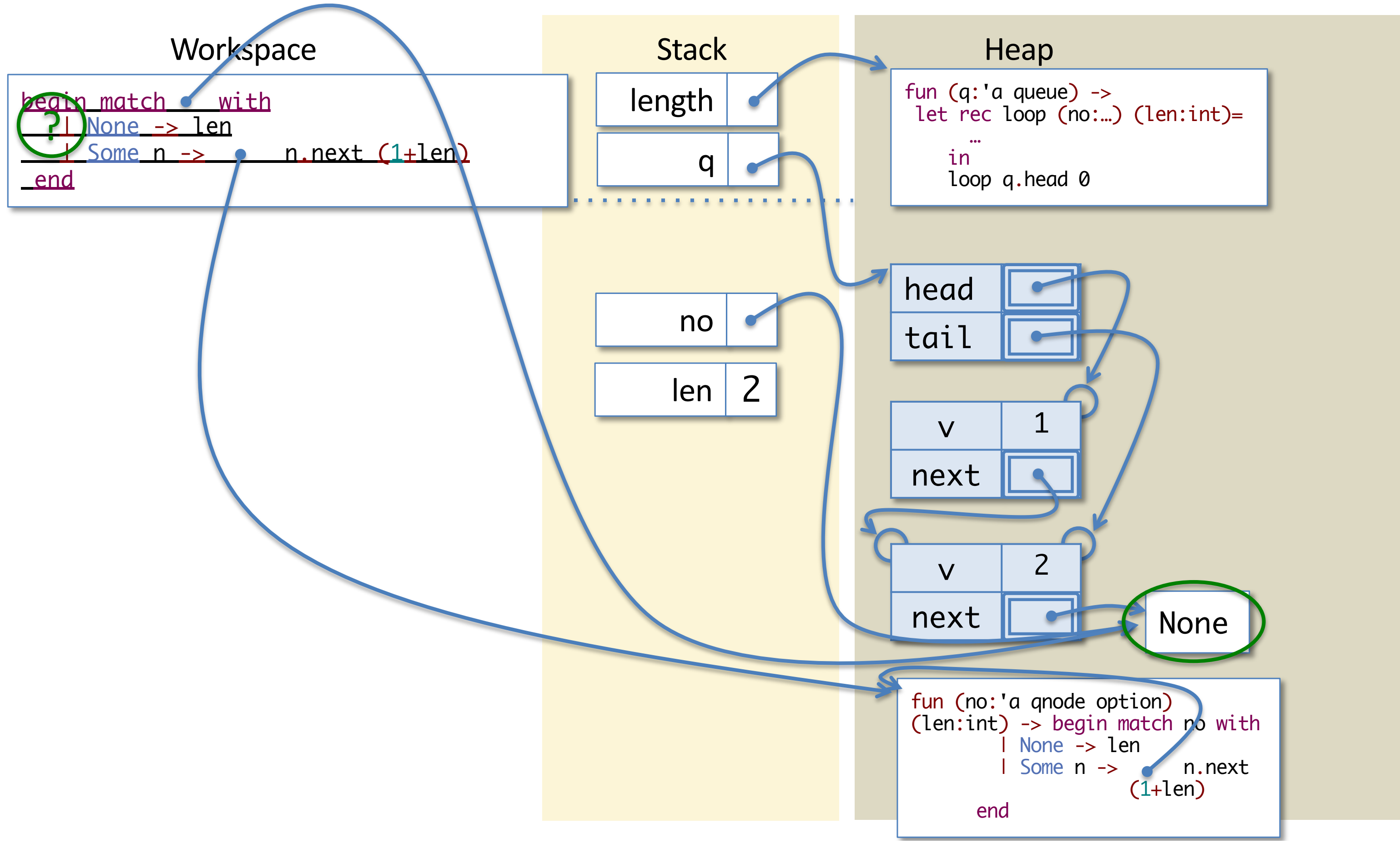
Again, the tail call leaves the stack the same shape as before, but it effectively *updates* the values of no and len.

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

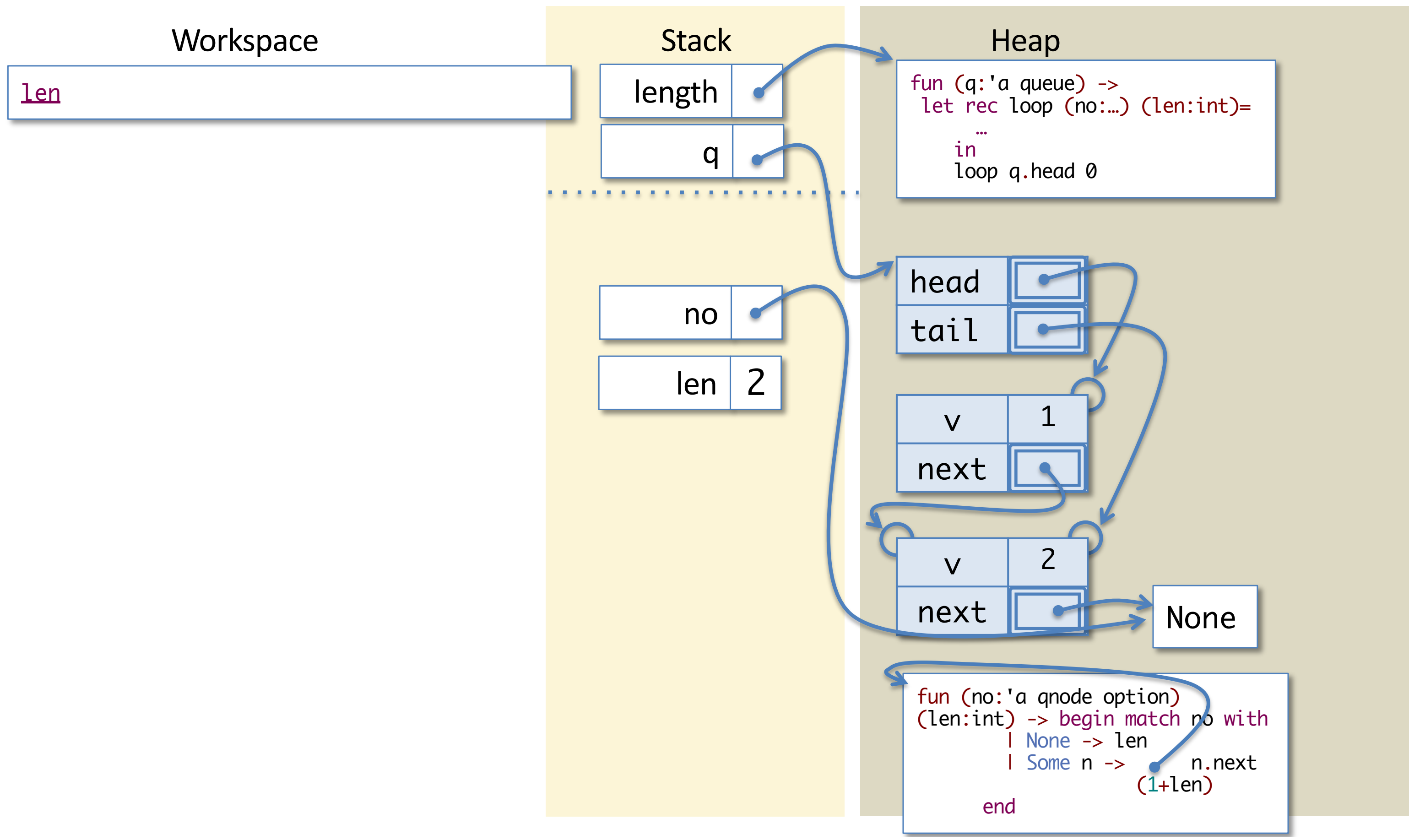
Tail Calls and Iterative length



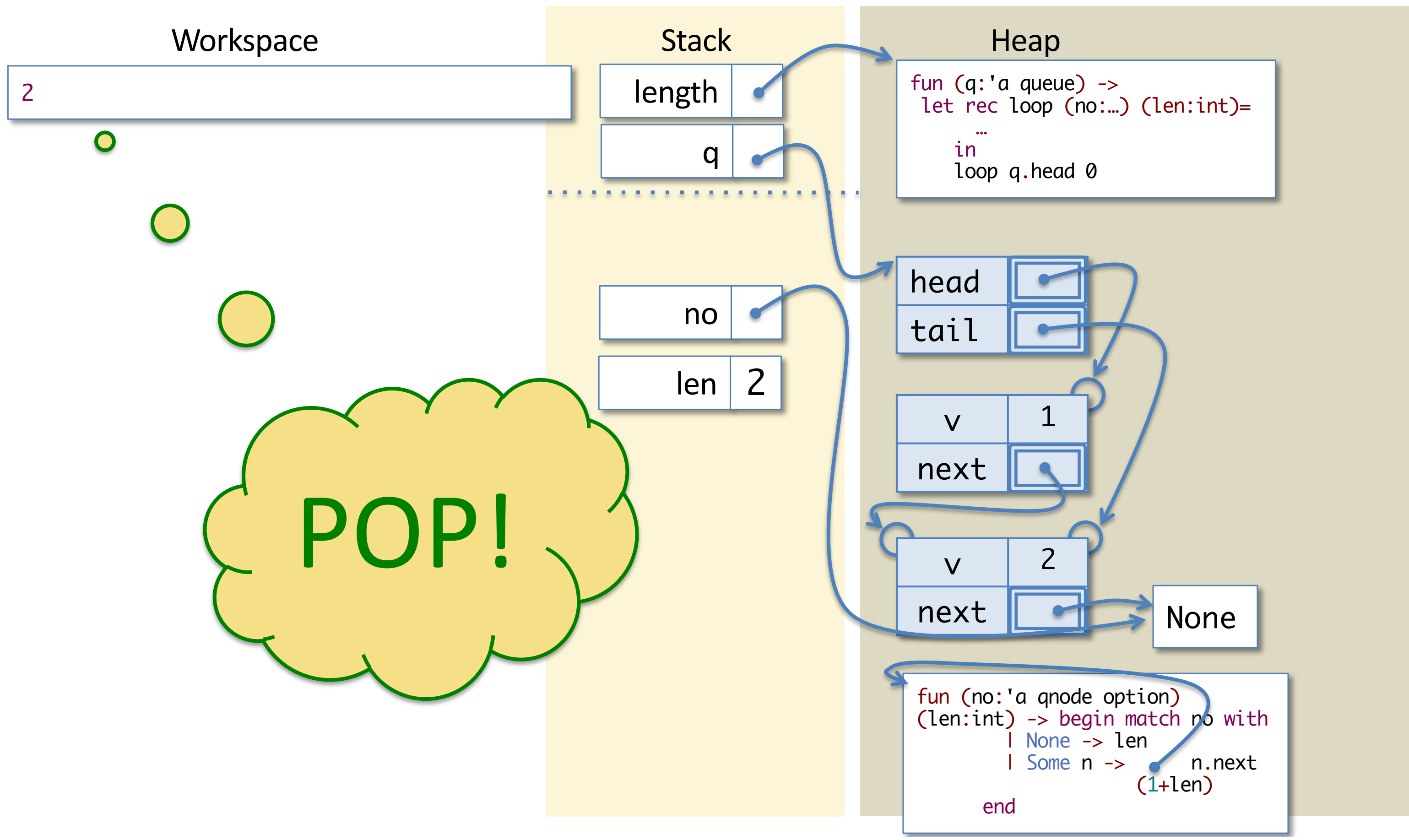
Tail Calls and Iterative length



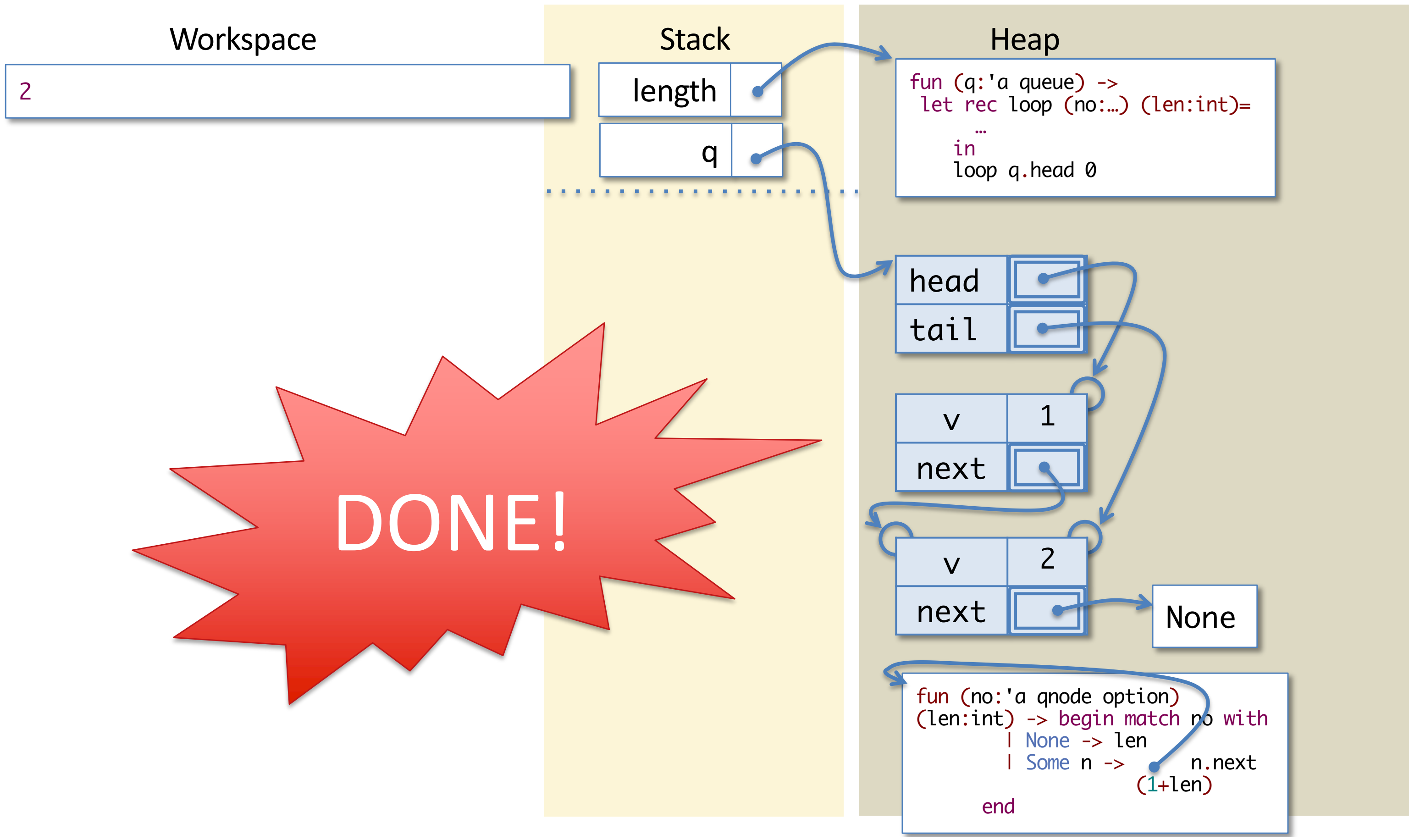
Tail Calls and Iterative length



Tail Calls and Iterative length



Tail Calls and Iterative length



Crucial Observations

- Tail call optimization lets the stack take only a *fixed amount of space*.
- The recursive call to `loop` effectively updates the stack bindings in place.
  - We can think of these bindings as the *state* being modified by each iteration of the loop.
- These two properties are the essence of iteration.
  - They are the difference between general recursion and iteration

16: What happens when you run this function on a (valid) queue containing 2 elements?

0

The value 2 is returned	0%
The value 0 is returned	0%
StackOverflow	0%
Your program hangs	0%

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

16: What happens when you run this function on a (valid) queue containing 2 elements?

0

The value 2 is returned	0%
The value 0 is returned	0%
StackOverflow	0%
Your program hangs	0%

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

chop

`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”.

chop (remembering "previous")

```
(* Removes all elements from q starting with the first
   occurrence of elt.  *)
let chop (elt:'a) (q:'a queue) : unit =
  let rec loop (curr: 'a qnode option)
               (prev: 'a qnode option) : unit =
    begin match curr with
      | None -> ()
      | Some n ->
         if n.v == elt
         then begin
           q.tail <- prev;     (* NOTE: update the tail with prev *)
           begin match prev with
           | None    -> q.head <- None (* We deleted everything! *)
           | Some n -> n.next <- None (* prev is the new tail *)
           end
         end else loop n.next curr
    end
  in loop q.head None
```

- Here, the state needed is the current "pointer" and (optionally) the previous node.
  - We need to keep track of the previous node to update the tail

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!

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)