Programming Languages and Techniques (CIS120)

Lecture 14

September 30th , 2015

The Abstract Stack Machine

Announcements

Midterm 1

- In class on *Friday, October 2nd*
  - Last names A – L Leidy Labs 10 (here)
  - Last names M – Z Cohen G17
- Covers lecture material through Sept. 23
  - Pure, value-oriented programming up to option Types
  - Chapters 1 – 11 in the UPDATED notes
- Review materials (old exams) on course website
- Contact me if you need to take the make-up exam

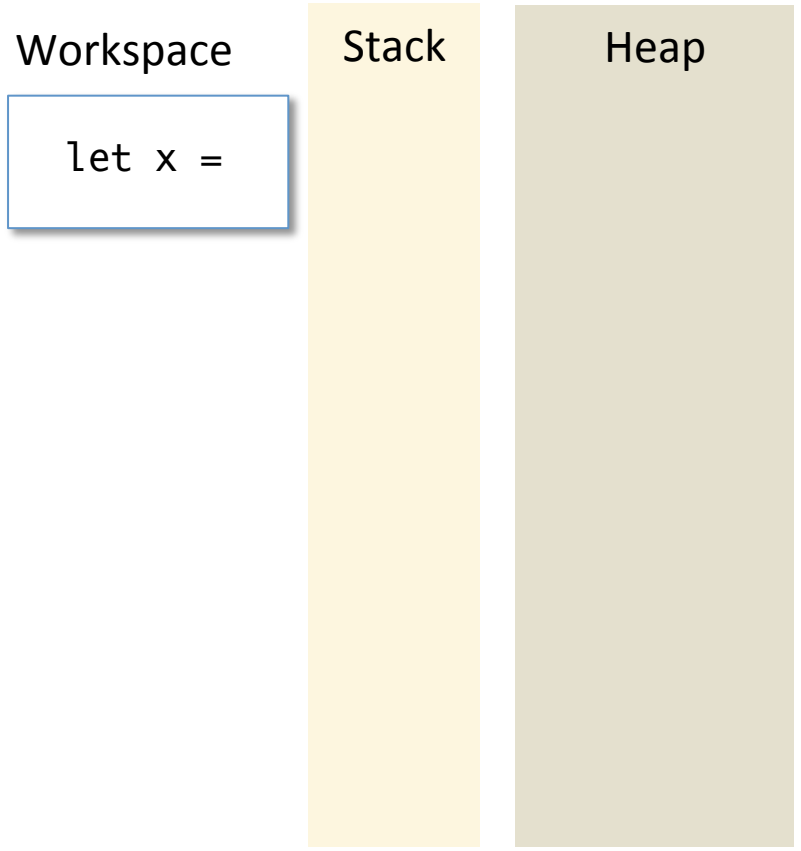
- Review Session:
  TONIGHT 9:00-11:00 Levine 100 (Wu & Chen)

Modeling State

Location, Location, Location!

Abstract Stack Machine

- Three "spaces"
  - workspace
    - the expression the computer is currently working with
  - stack
    - temporary storage for `let` bindings and partially simplified expressions
  - heap
    - storage area for large data structures
- Initial state:
  - workspace contains whole program
  - stack and heap are empty
- Machine operation:
  - In each step, choose next part of the workspace expression and simplify it
  - Stop when there are no more simplifications



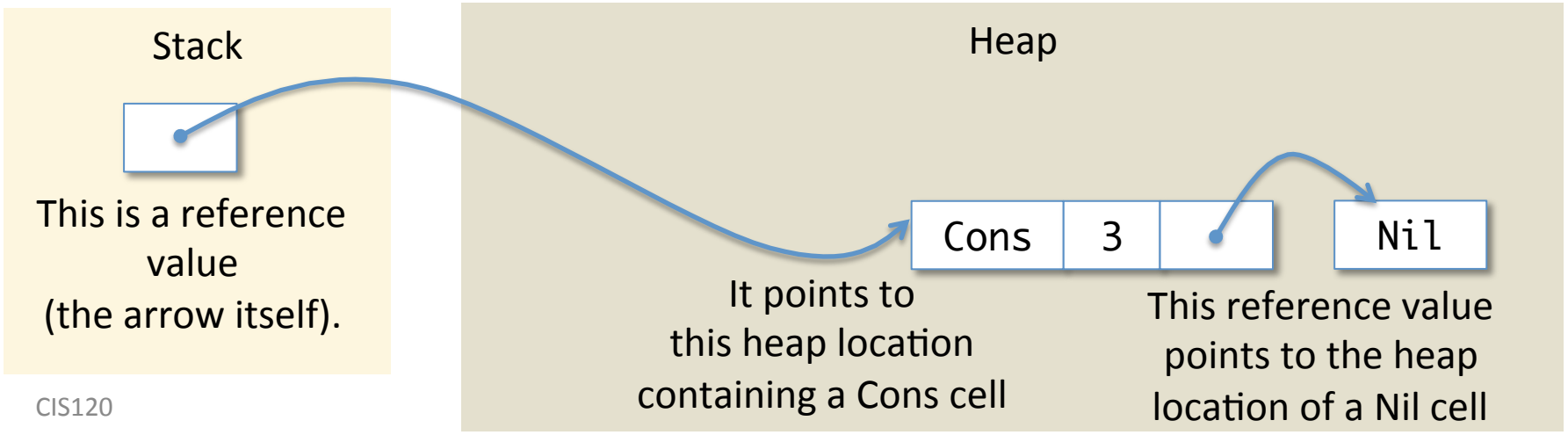
Values and References

A *value* is either:

- a *primitive value* like an integer, or,
- a *reference* to a location in the heap

A reference is the *address* of a piece of data in the heap. We draw a reference value as an "arrow":

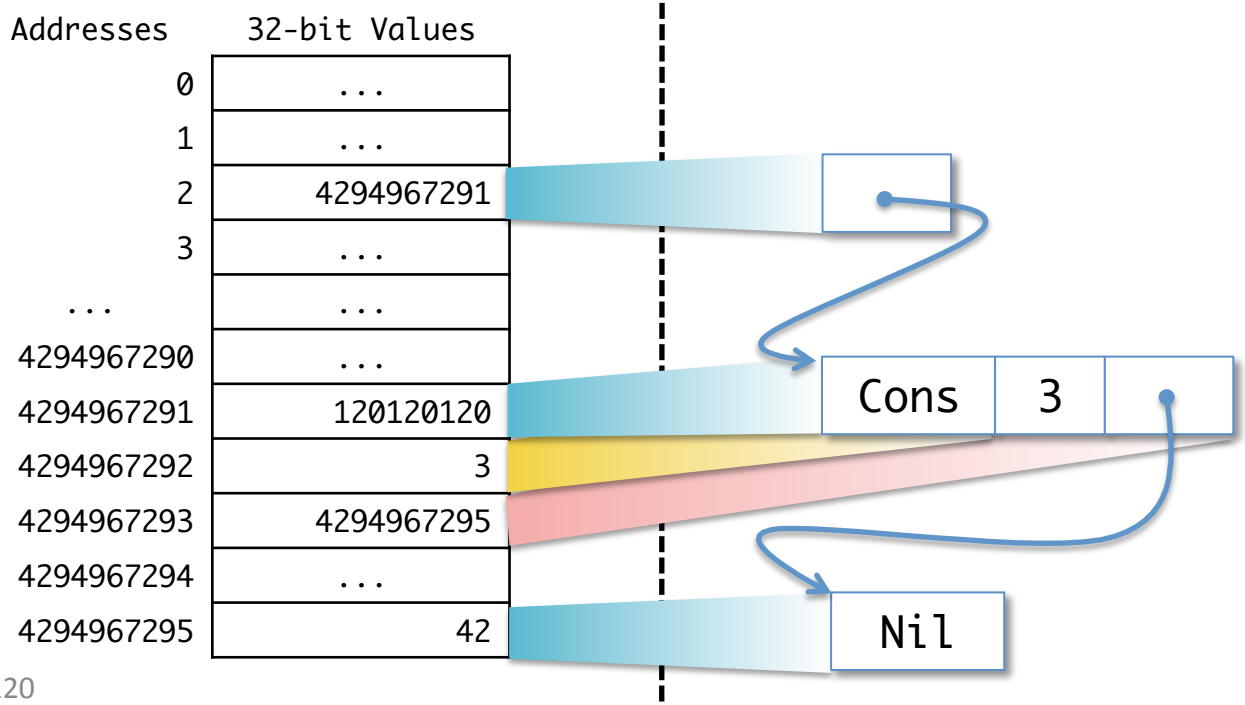
- The start of the arrow is the reference itself (i.e. the address).
- The arrow "points" to the value located at the reference's address.



References as an Abstraction

- In a real computer, the memory consists of an array of 32-bit words, numbered 0 … 2^{32}-1 (for a 32-bit machine)
  - A reference is just an address that tells you where to look up a value
  - Data structures are usually laid out in contiguous blocks of memory
  - Constructor tags are just numbers chosen by the compiler
    e.g. Nil = 42 and Cons = 120120120

The "real" heap.



How we picture it.

The ASM:
let, variables, operators,
and if expressions

Simplification

Workspace

```
let x = 10 + 12 in
let y = 2 + x in
  if x > 23 then 3 else 4
```

Stack

Heap

Simplification

Workspace

```
let x = 10 + 12 in
let y = 2 + x in
  if x > 23 then 3 else 4
```

Stack

Heap

Simplification

Workspace

```
let x = 22 in
let y = 2 + x in
  if x > 23 then 3 else 4
```

Stack

Heap

Simplification

Workspace

```
let x = 22 in
let y = 2 + x in
  if x > 23 then 3 else 4
```

Stack

Heap

Simplification

Workspace

```
let y = 2 + x in
  if x > 23 then 3 else 4
```

Stack

x	22

Heap

Simplification

Workspace

```
let y = 2 + x in
  if x > 23 then 3 else 4
```

Stack

x	22

Heap

x is not a value: so look it up in the stack

Simplification

Workspace

```
let y = 2 + 22 in
  if x > 23 then 3 else 4
```

Stack

x	22

Heap

Simplification

Workspace

```
let y = 2 + 22 in
  if x > 23 then 3 else 4
```

Stack

x	22

Heap

Simplification

Workspace

```
let y = 24 in
  if x > 23 then 3 else 4
```

Stack

x	22

Heap

Simplification

Workspace

```
let y = 24 in
  if x > 23 then 3 else 4
```

Stack

x	22

Heap

Simplification

Workspace

```
if x > 23 then 3 else 4
```

Stack

x	22

y	24

Heap

Simplification

Workspace

```
if x > 23 then 3 else 4
```

Stack

x	22
y	24

Heap

Looking up x in the stack proceed from most recent entries to the least recent entries – the "top" (most recent part) of the stack is toward the bottom of the diagram.

Simplification

Workspace

```
if 22 > 23 then 3 else 4
```

Stack

x	22

y	24

Heap

Simplification

Workspace

```
if 22 > 23 then 3 else 4
```

Stack

x	22

y	24

Heap

Simplification

Workspace

```
if false then 3 else 4
```

Stack

x	22

y	24

Heap

Simplification

Workspace

```
if false then 3 else 4
```

Stack

x	22
y	24

Heap

Simplification

Workspace

4

Stack

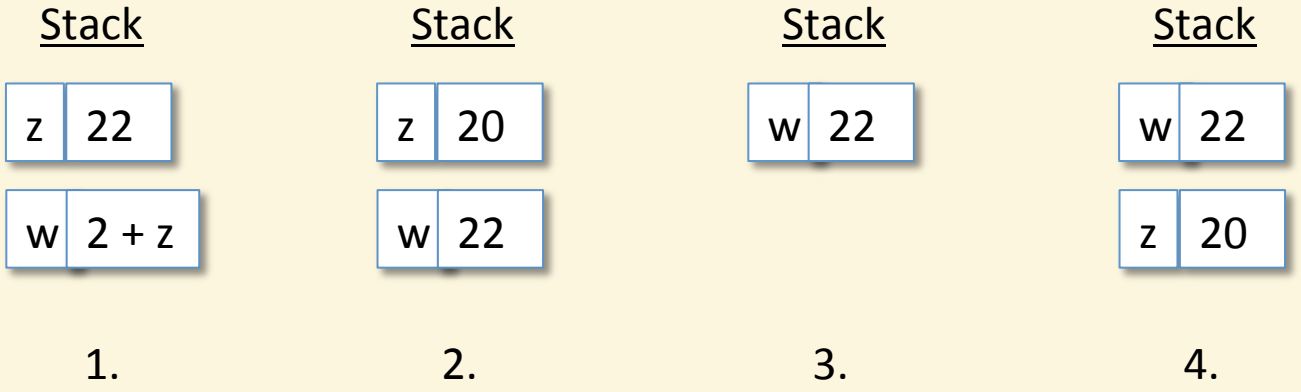
x	22

y	24

Heap

What does the Stack look like after simplifying the following code on the workspace?

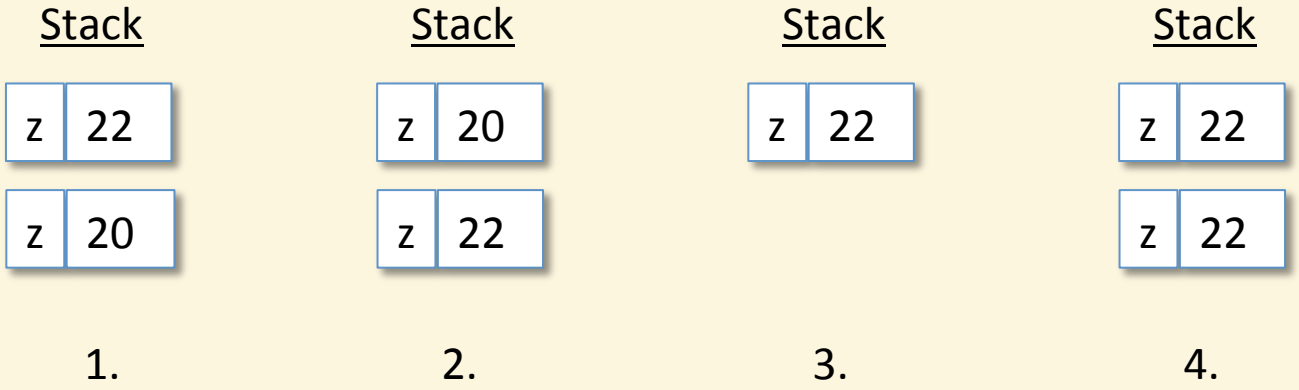
```
let z = 20 in
let w = 2 + z in
  w
```



ANSWER: 2

What does the Stack look like after simplifying the following code on the workspace?

```
let z = 20 in
let z = 2 + z in
  z
```



ANSWER: 2

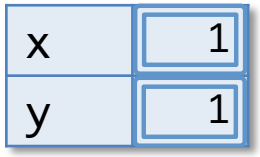
Mutable Records

- The reason for introducing all the ASM stuff is to make the model of heap locations and sharing *explicit*.
  - Now we can say what it means to mutate a heap value *in place*.

```
type point = {mutable x:int; mutable y:int}

let p1 : point = {x=1; y=1;}
let p2 : point = p1
let ans : int = (p2.x <- 17; p1.x)
```

- We draw a record in the heap like this:
  - The doubled outlines indicate that those cells are mutable
  - Everything else is immutable
  - (field names don't actually take up space)



A point record in the heap.

Allocate a Record

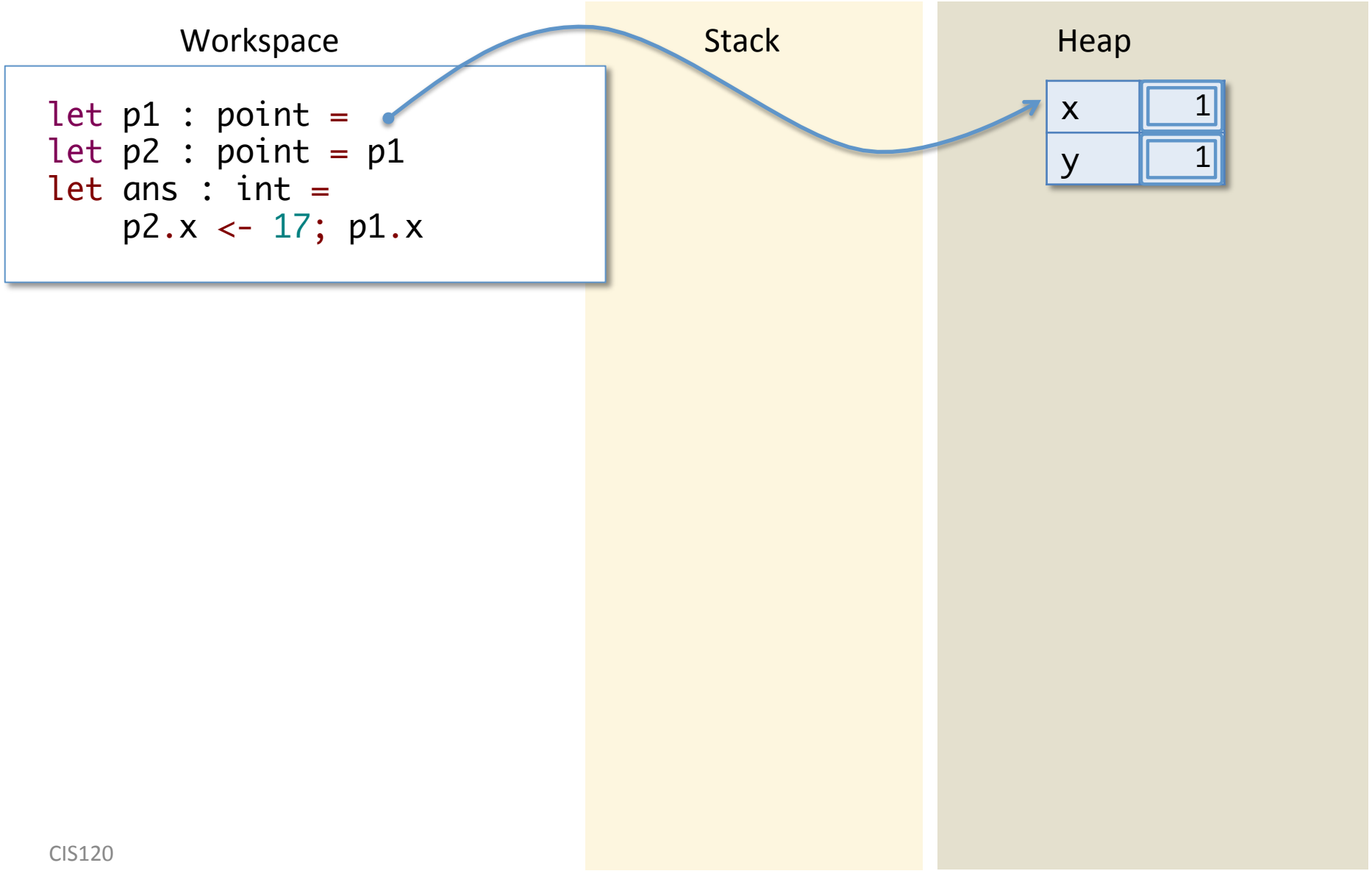
Workspace

```
let p1 : point = {x=1; y=1;}
let p2 : point = p1
let ans : int =
    p2.x <- 17; p1.x
```

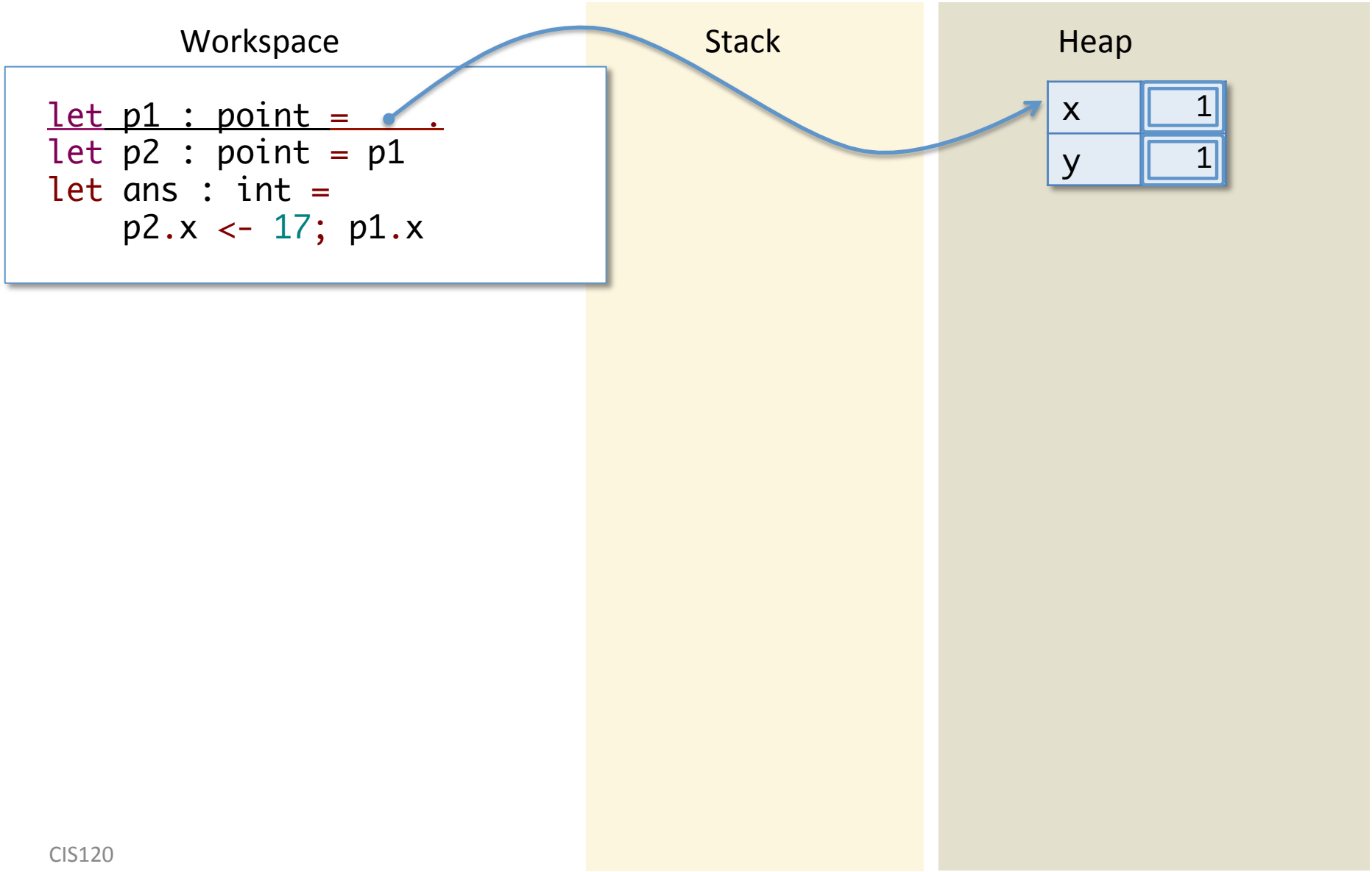
Stack

Heap

Allocate a Record



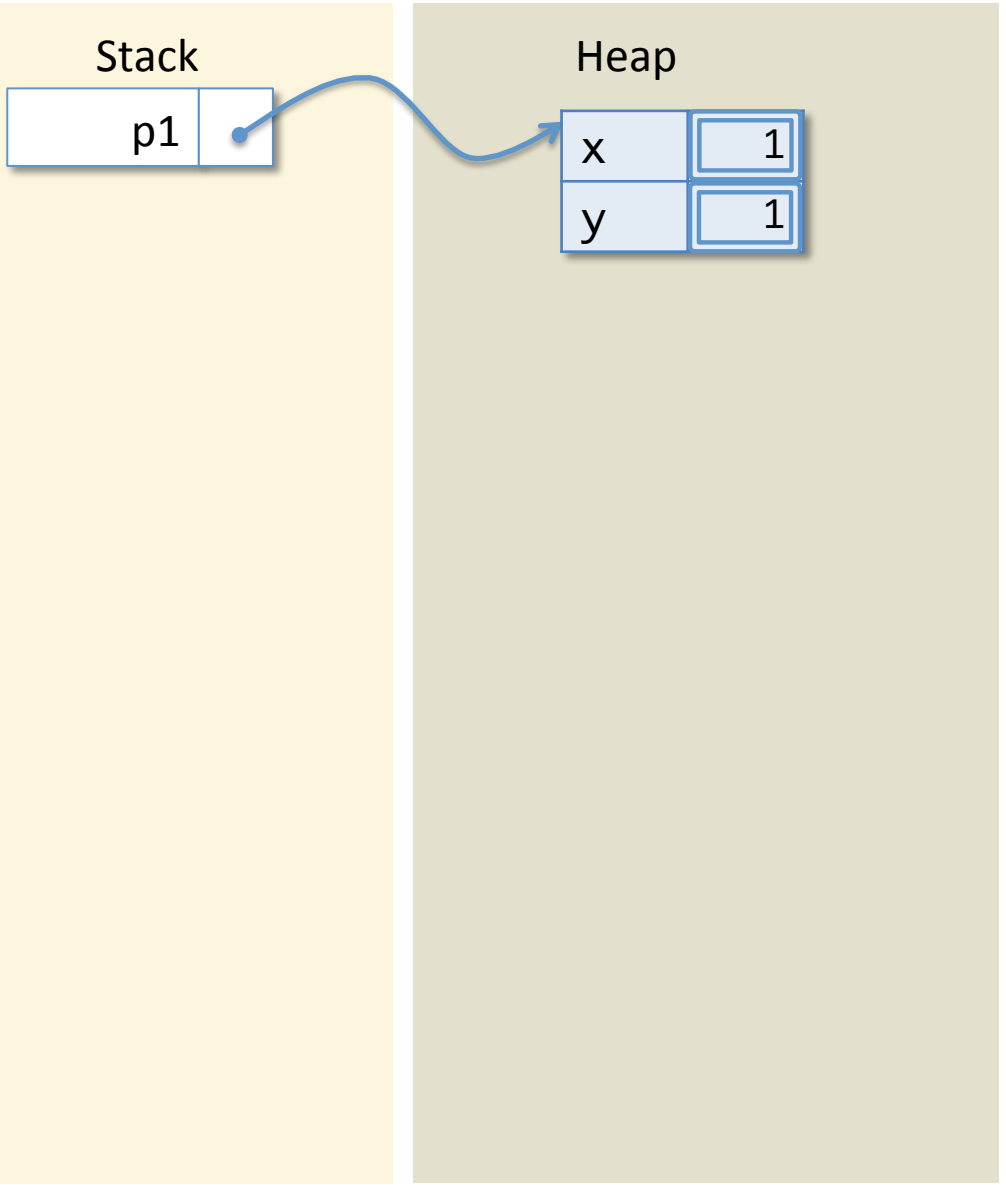
Let Expression



Push p1

Workspace

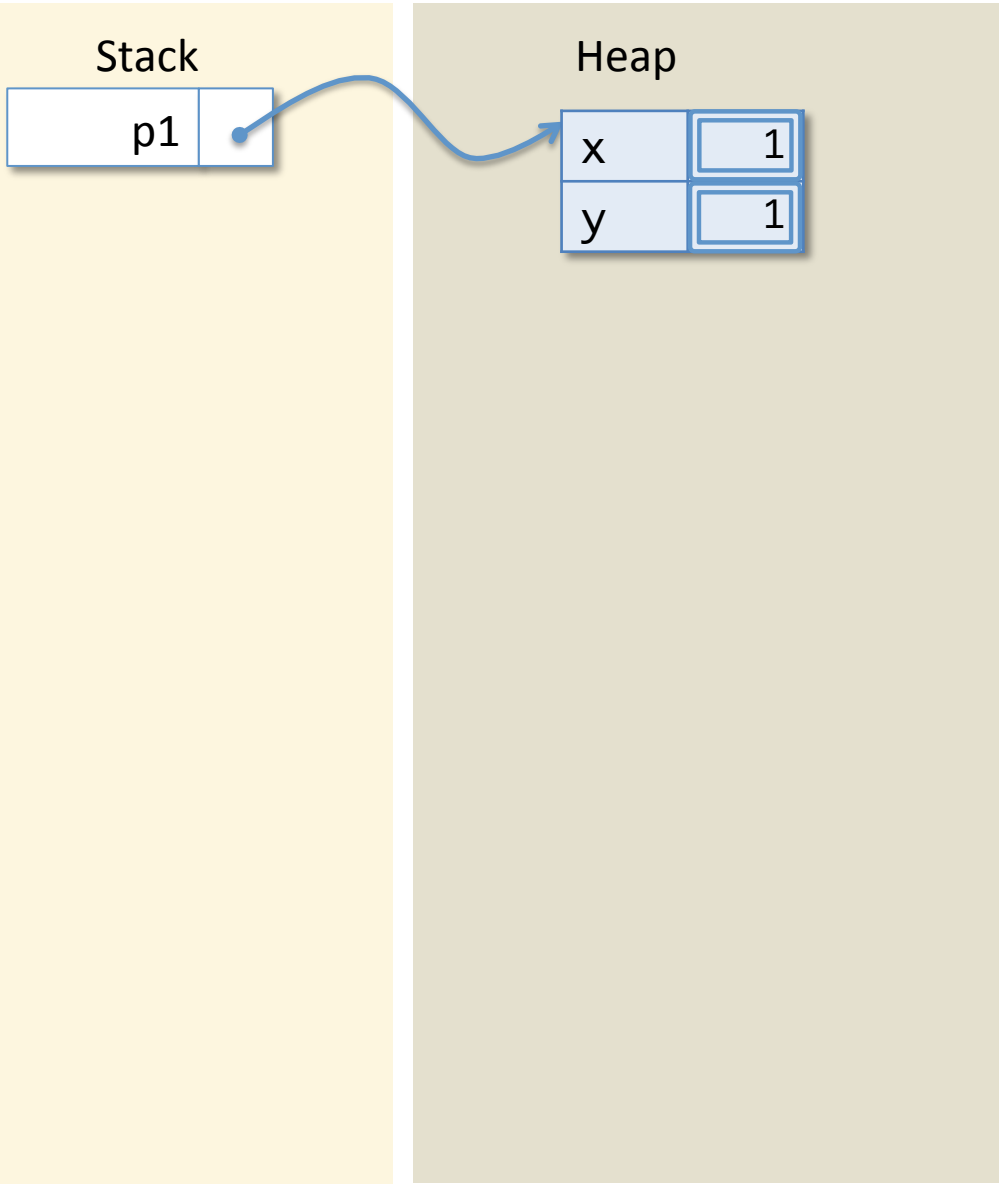
```
let p2 : point = p1
let ans : int =
    p2.x <- 17; p1.x
```



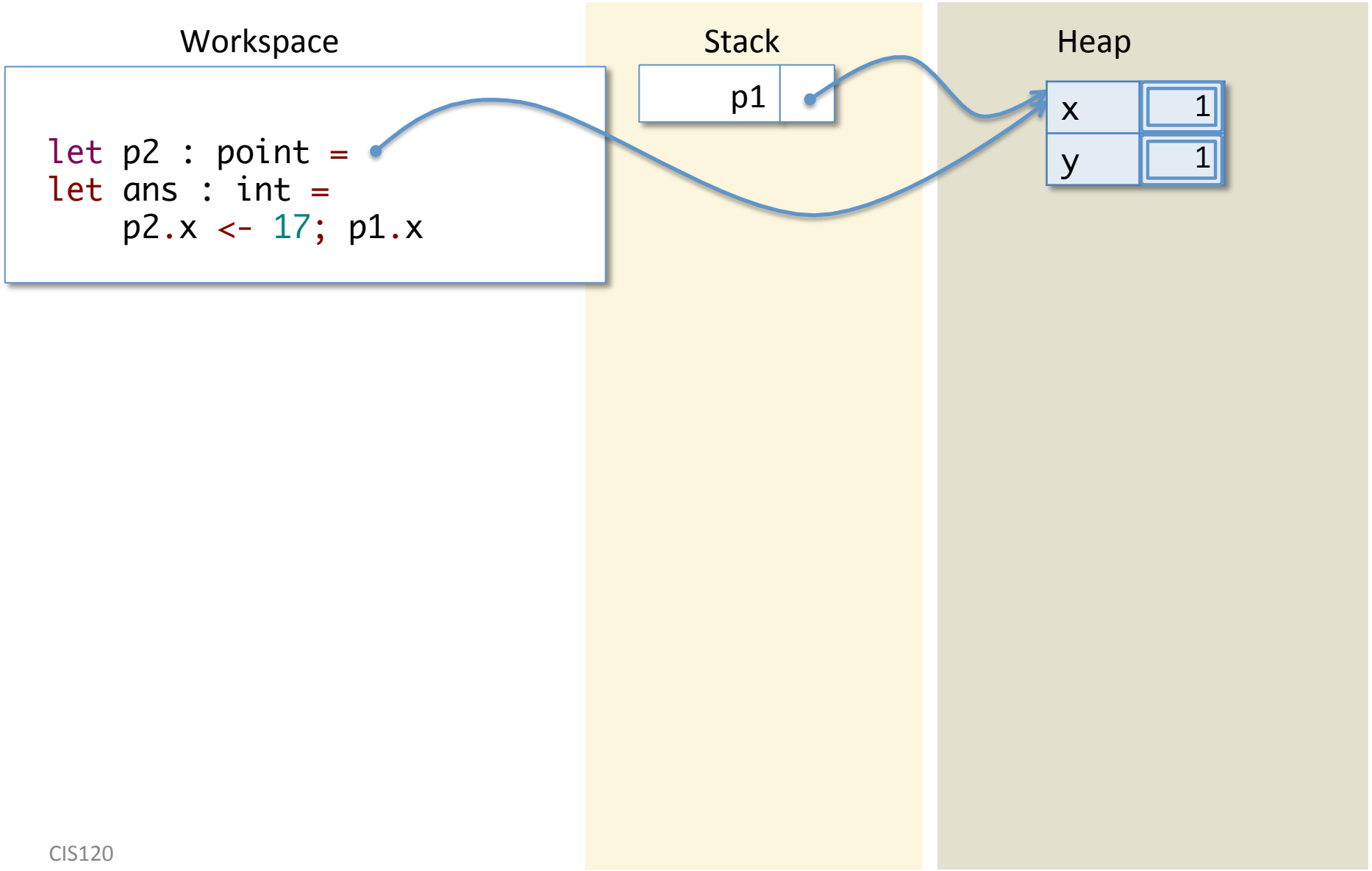
Look Up ‘p1’

Workspace

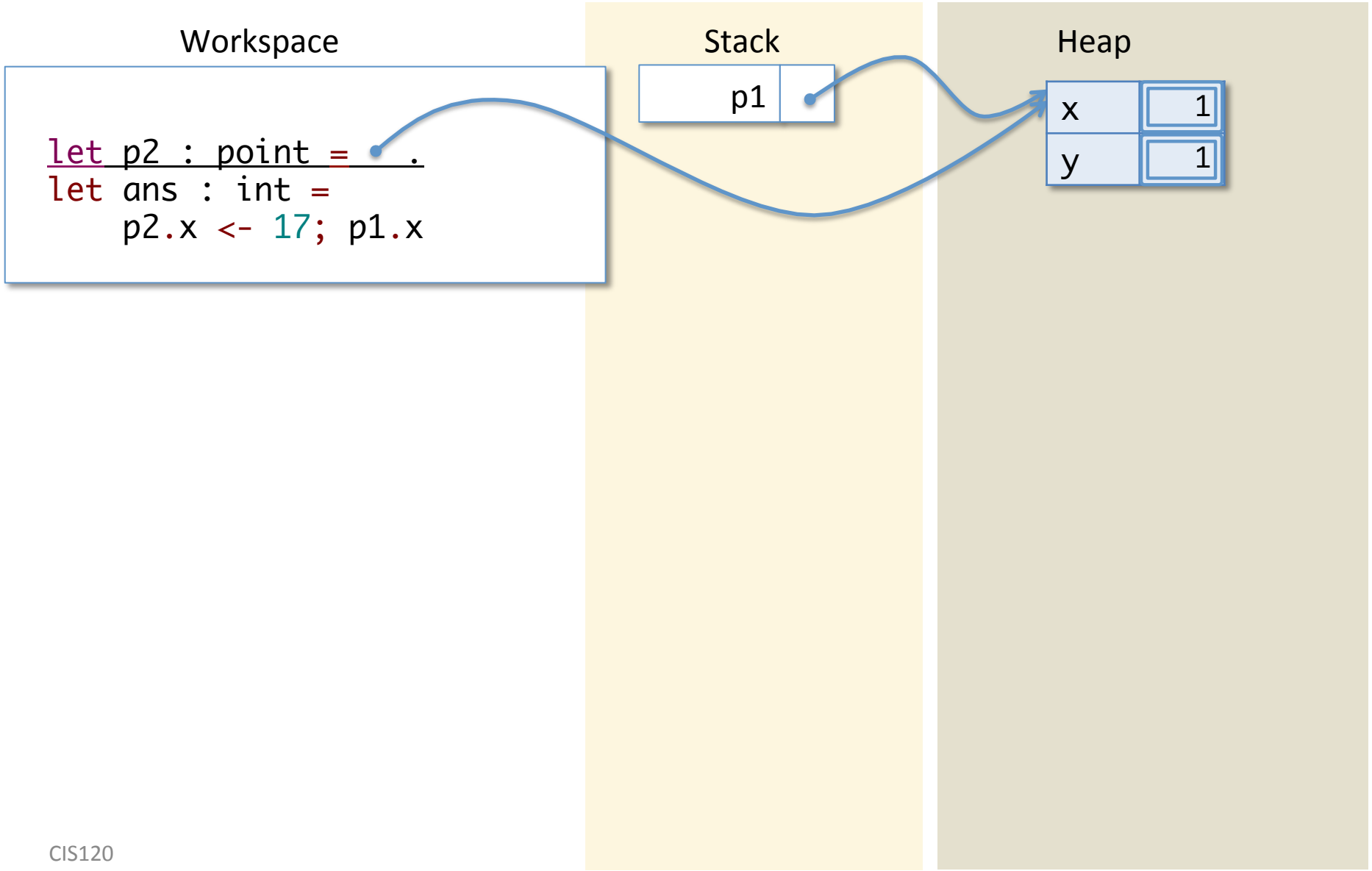
```
let p2 : point = p1
let ans : int =
    p2.x <- 17; p1.x
```



Look Up ‘p1’


Workspace
let p2 : point =
let ans : int =
p2.x <- 17; p1.x
Stack
p1
Heap
x
1
y
1

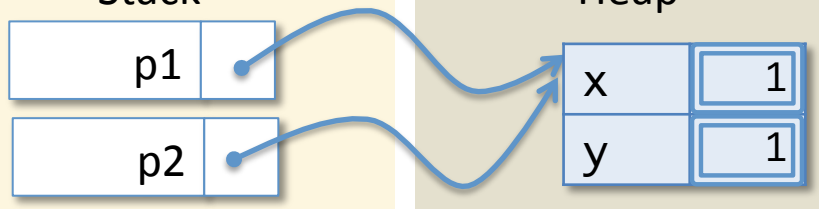
Let Expression



Push p2

Workspace

```
let ans : int =
    p2.x <- 17; p1.x
```

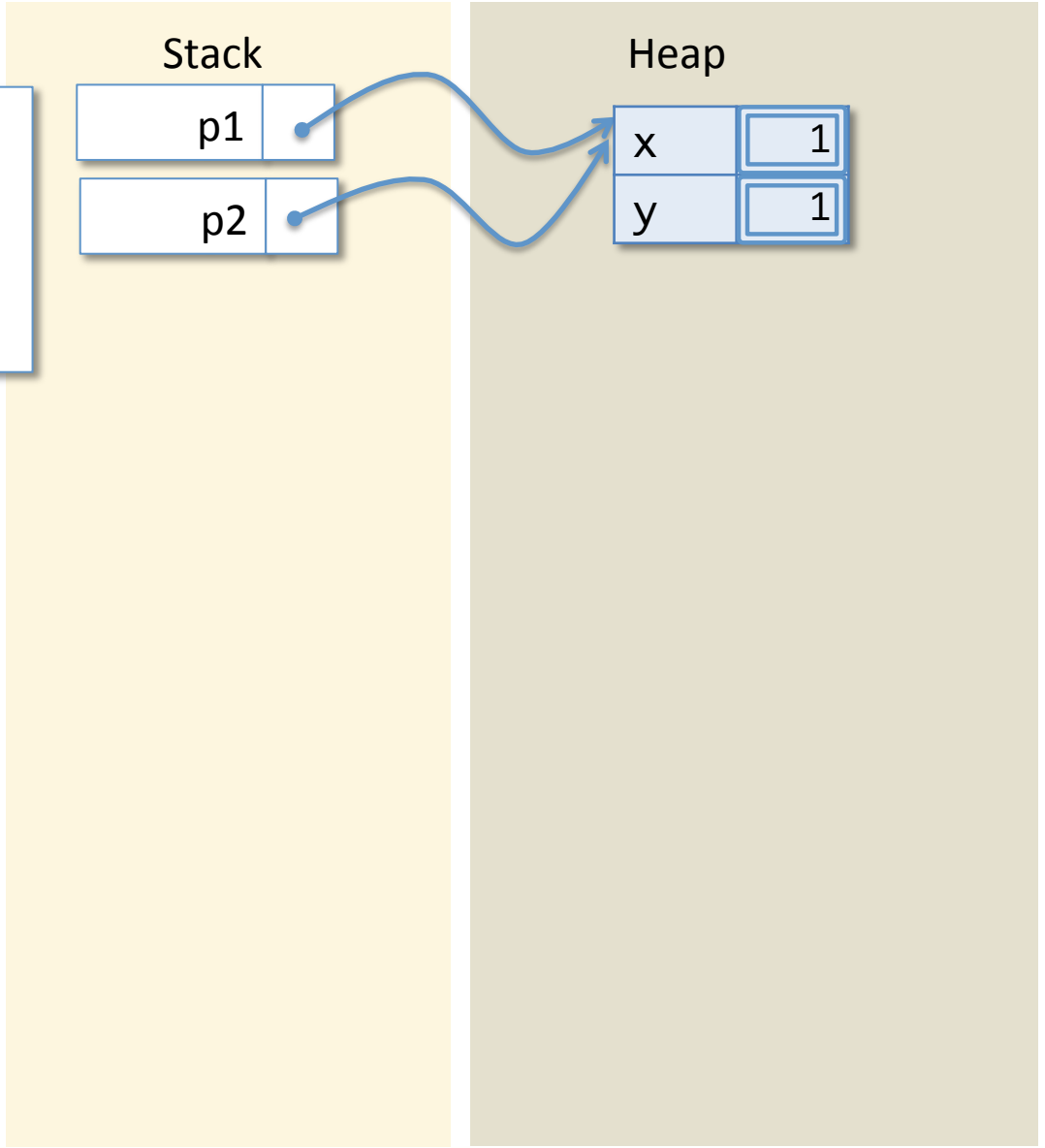


Note: p1 and p2 are references to the *same* heap record.
They are *aliases* – two different names for the *same thing*.

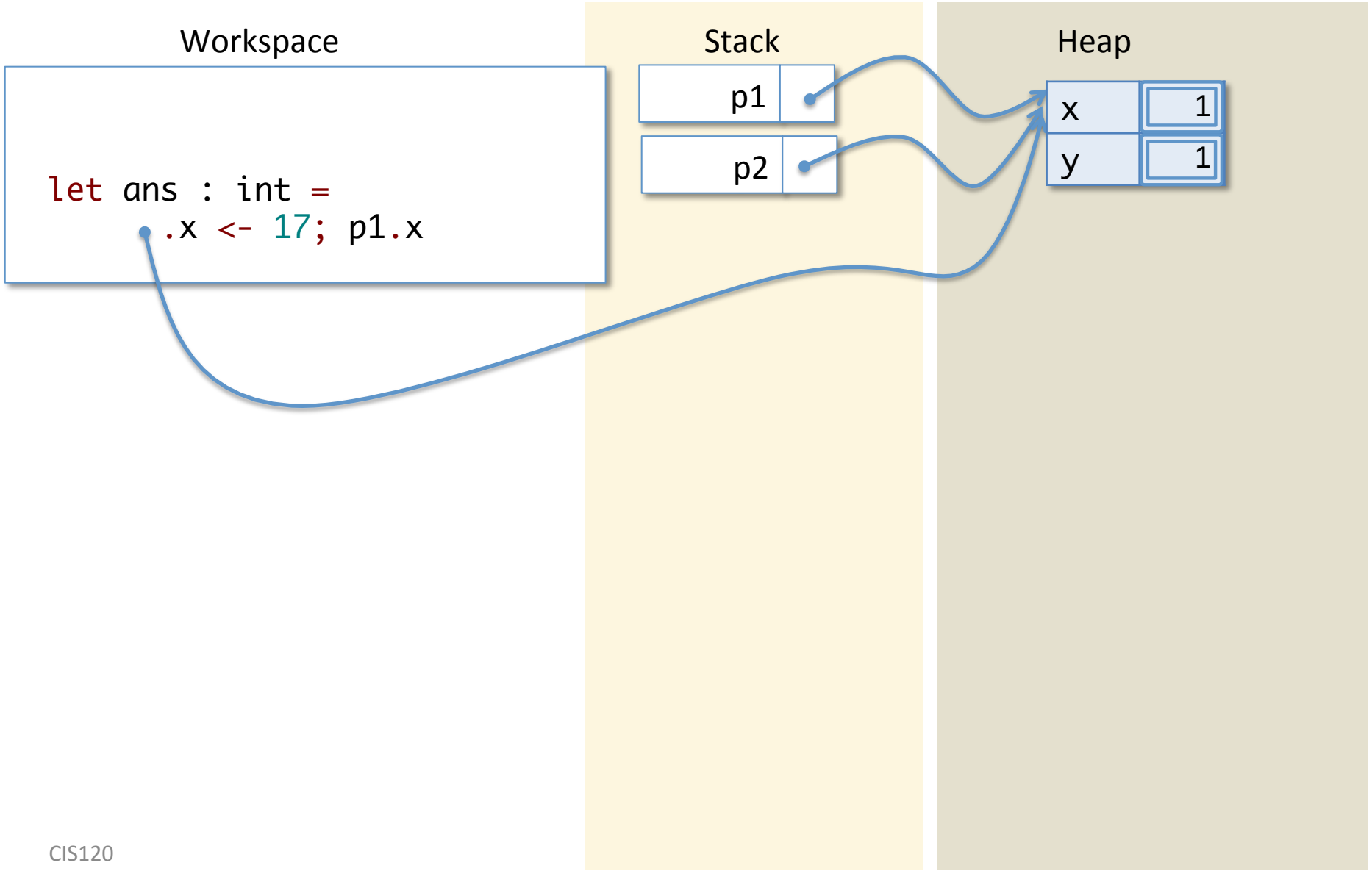
Look Up ‘p2’

Workspace

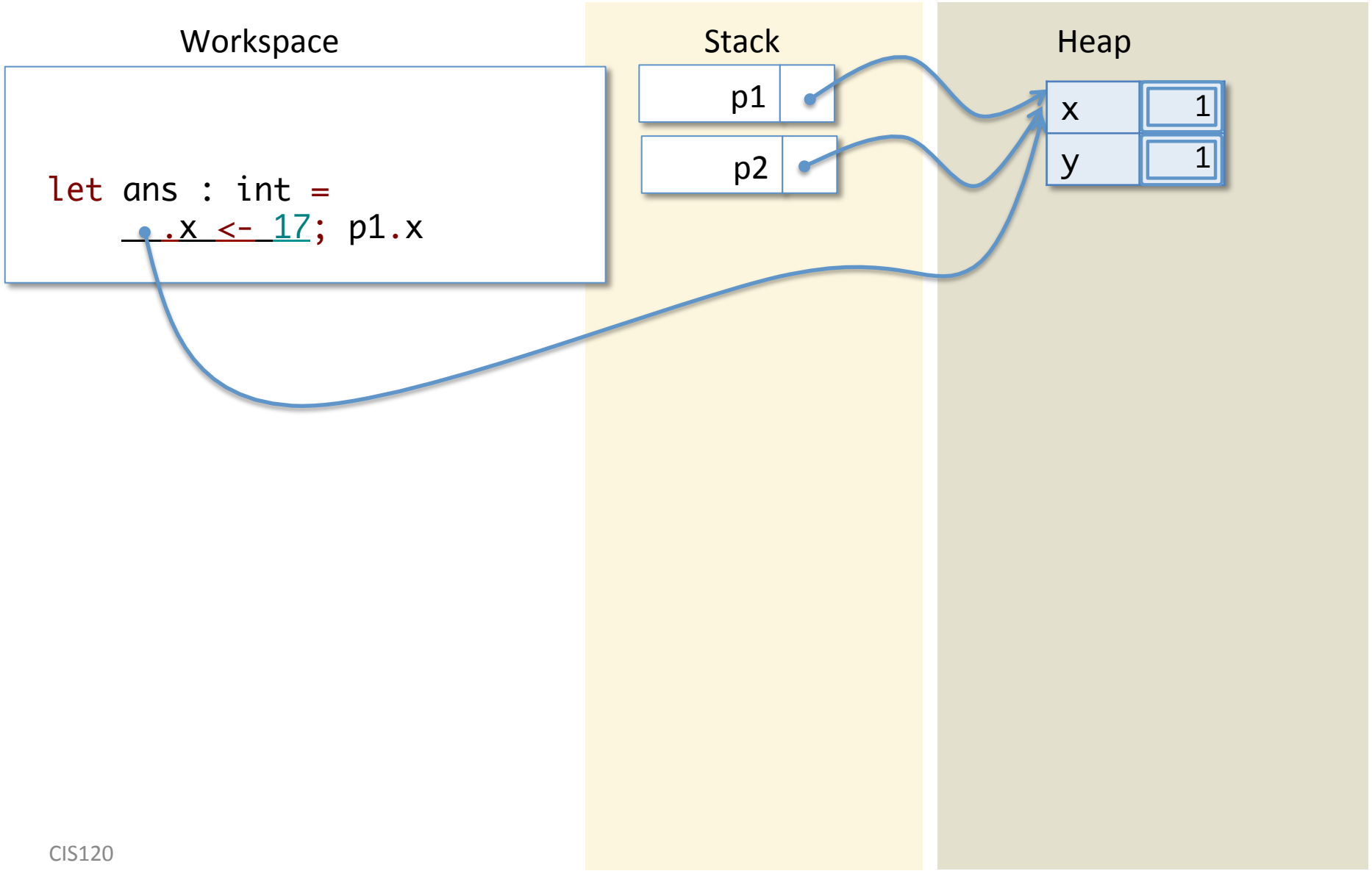
```
let ans : int =
    p2.x <- 17; p1.x
```



Look Up ‘p2’



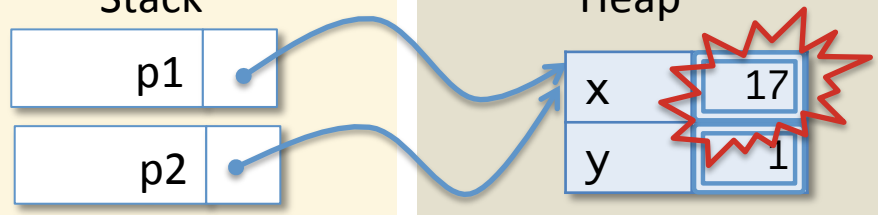
Assign to x field



Assign to x field

Workspace

```
let ans : int =
    (); p1.x
```

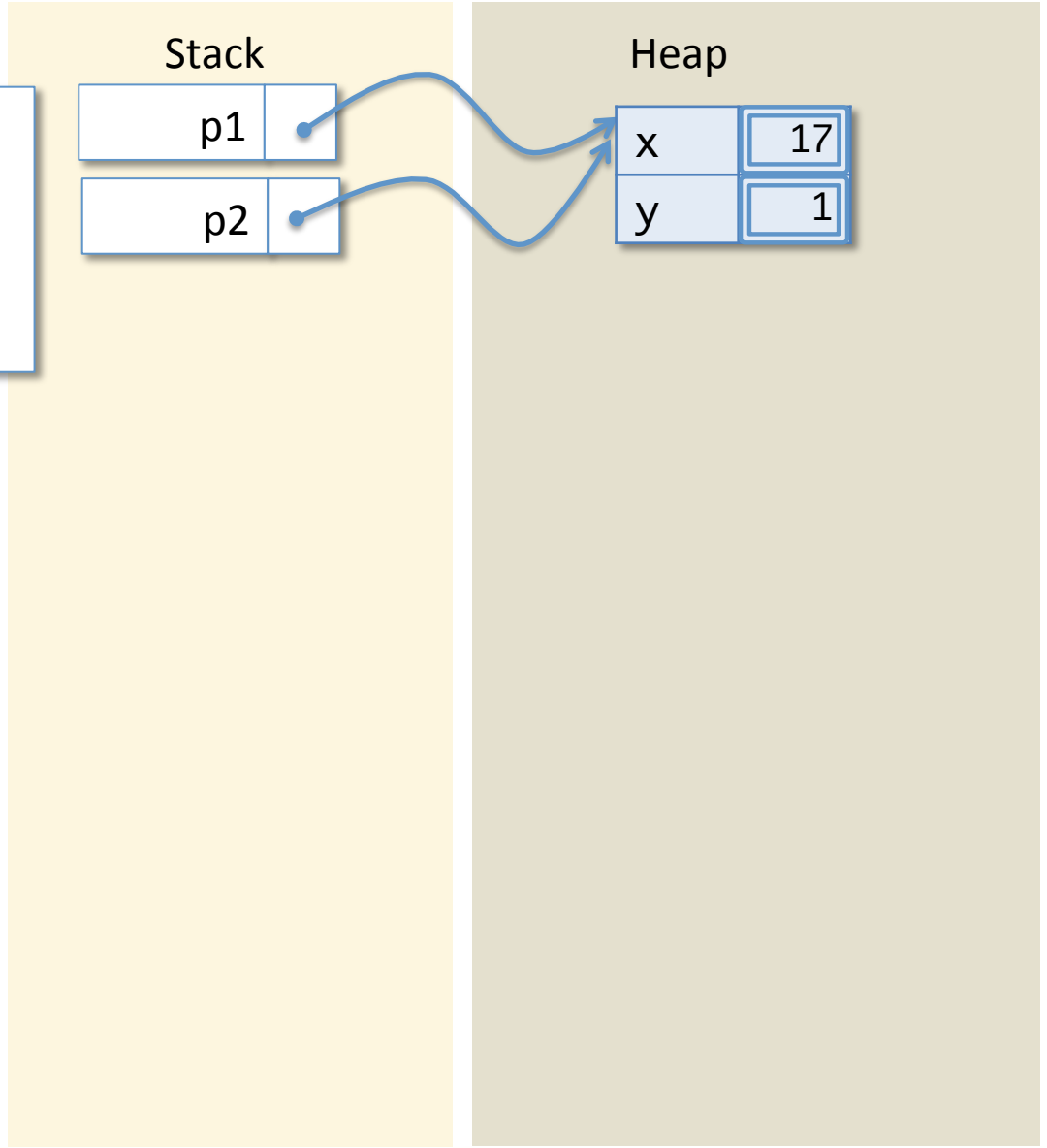


This is the step in which the 'imperative' update occurs. The mutable field x has been modified in place to contain the value 17.

Sequence ';' Discards Unit

Workspace

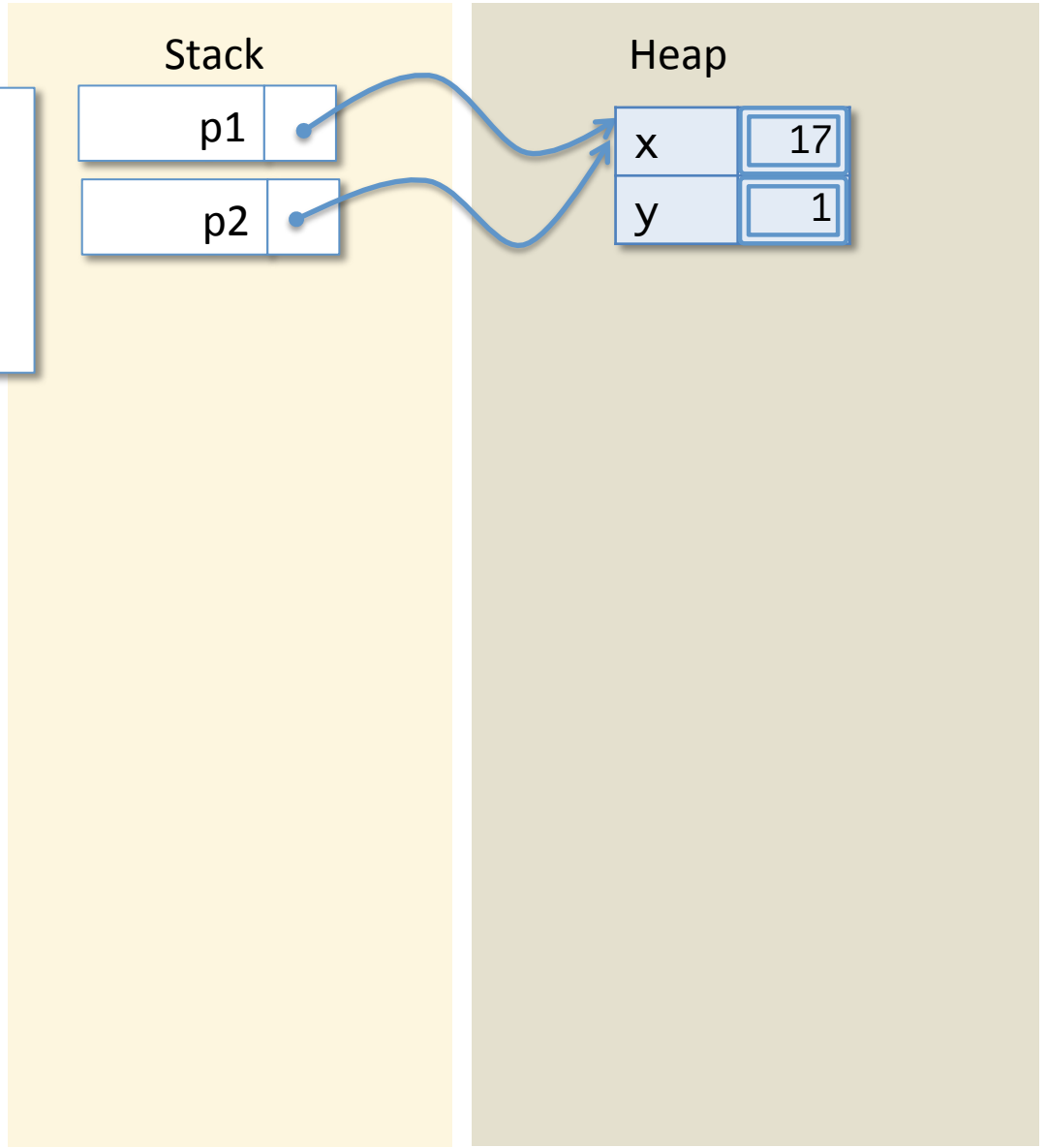
```
let ans : int =
    (); p1.x
```



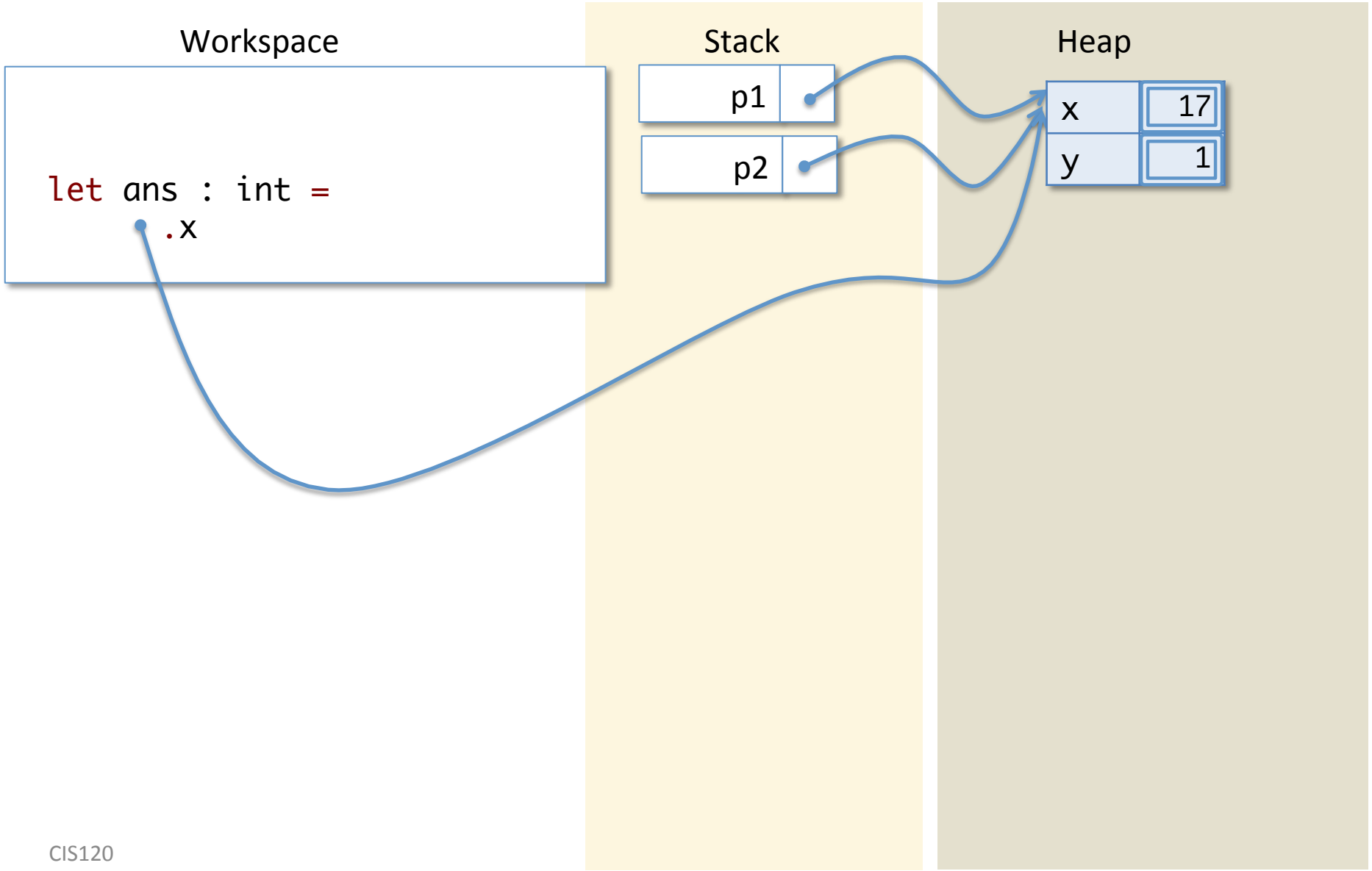
Look Up ‘p1’

Workspace

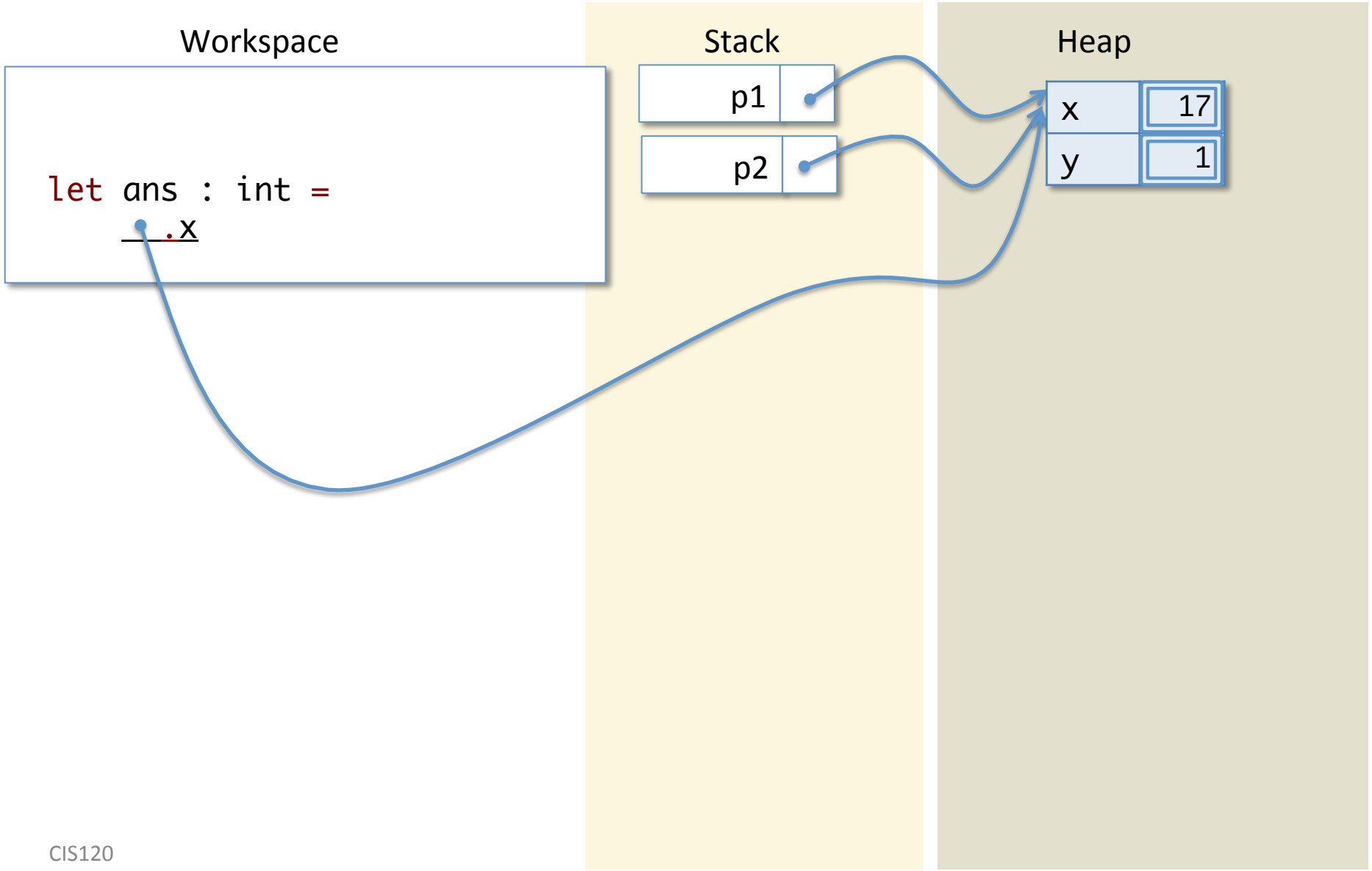
```
let ans : int =
    p1.x
```



Look Up ‘p1’



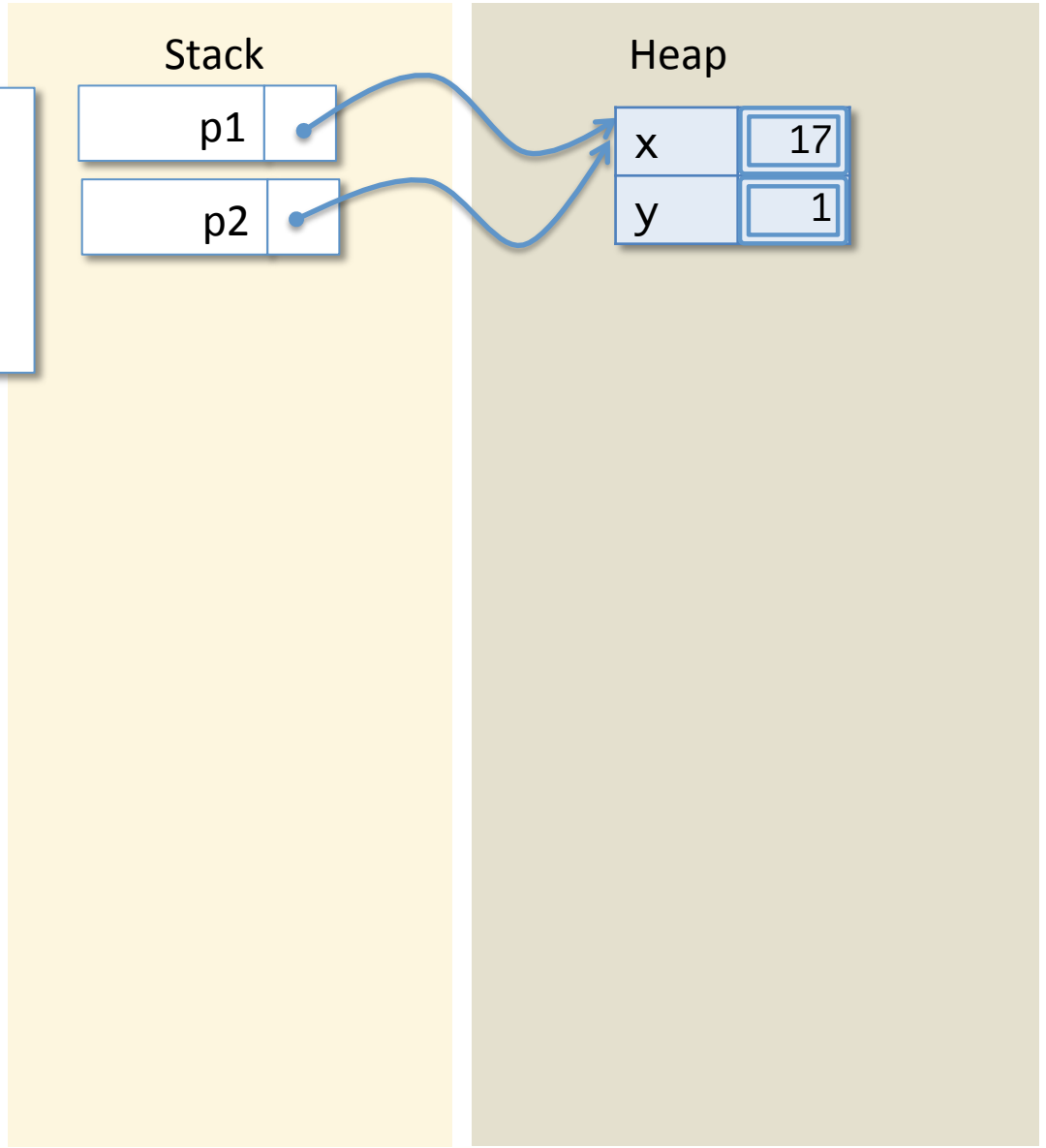
Project the ‘x’ field



Project the ‘x’ field

Workspace

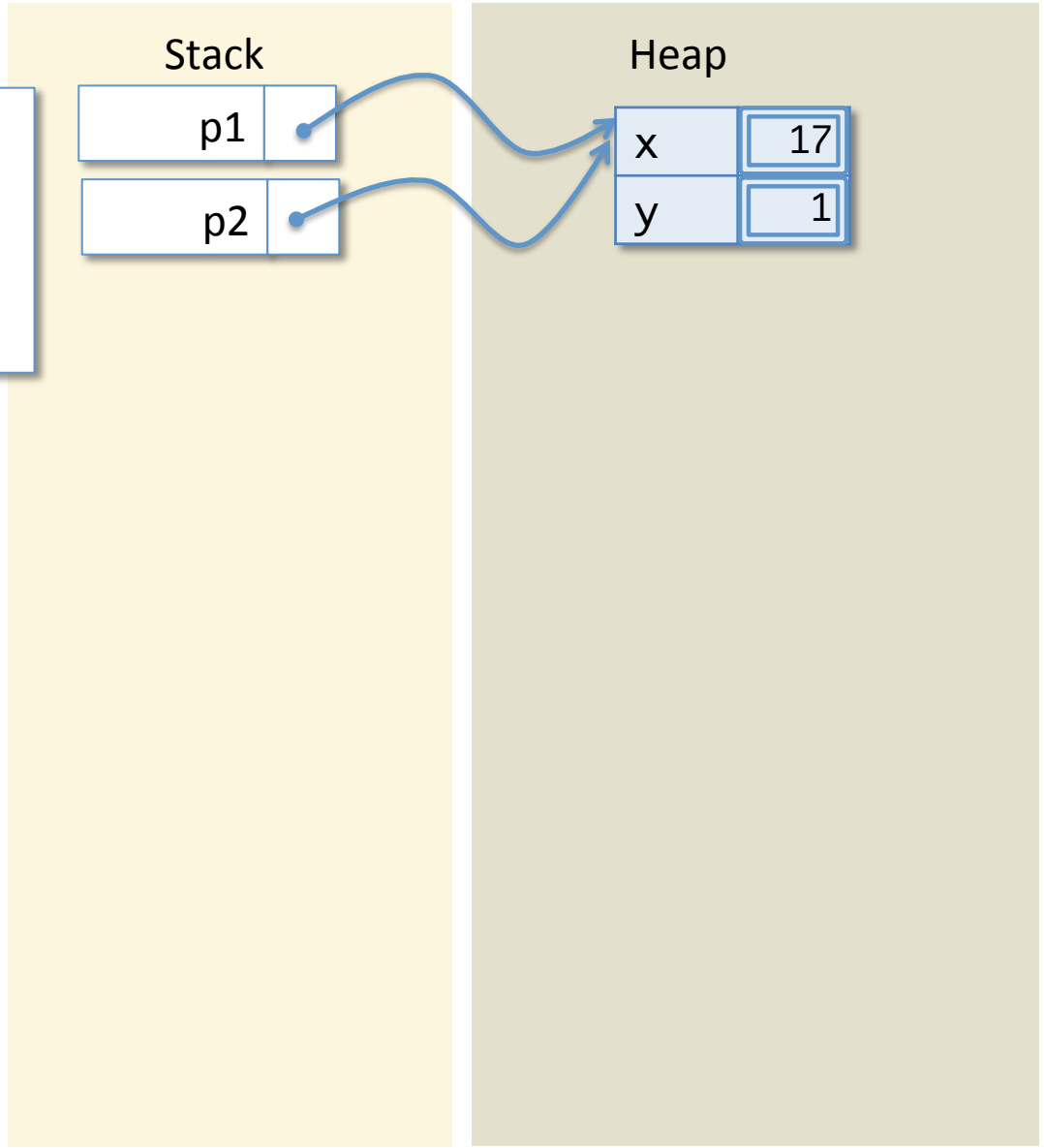
```
let ans : int =
    17
```



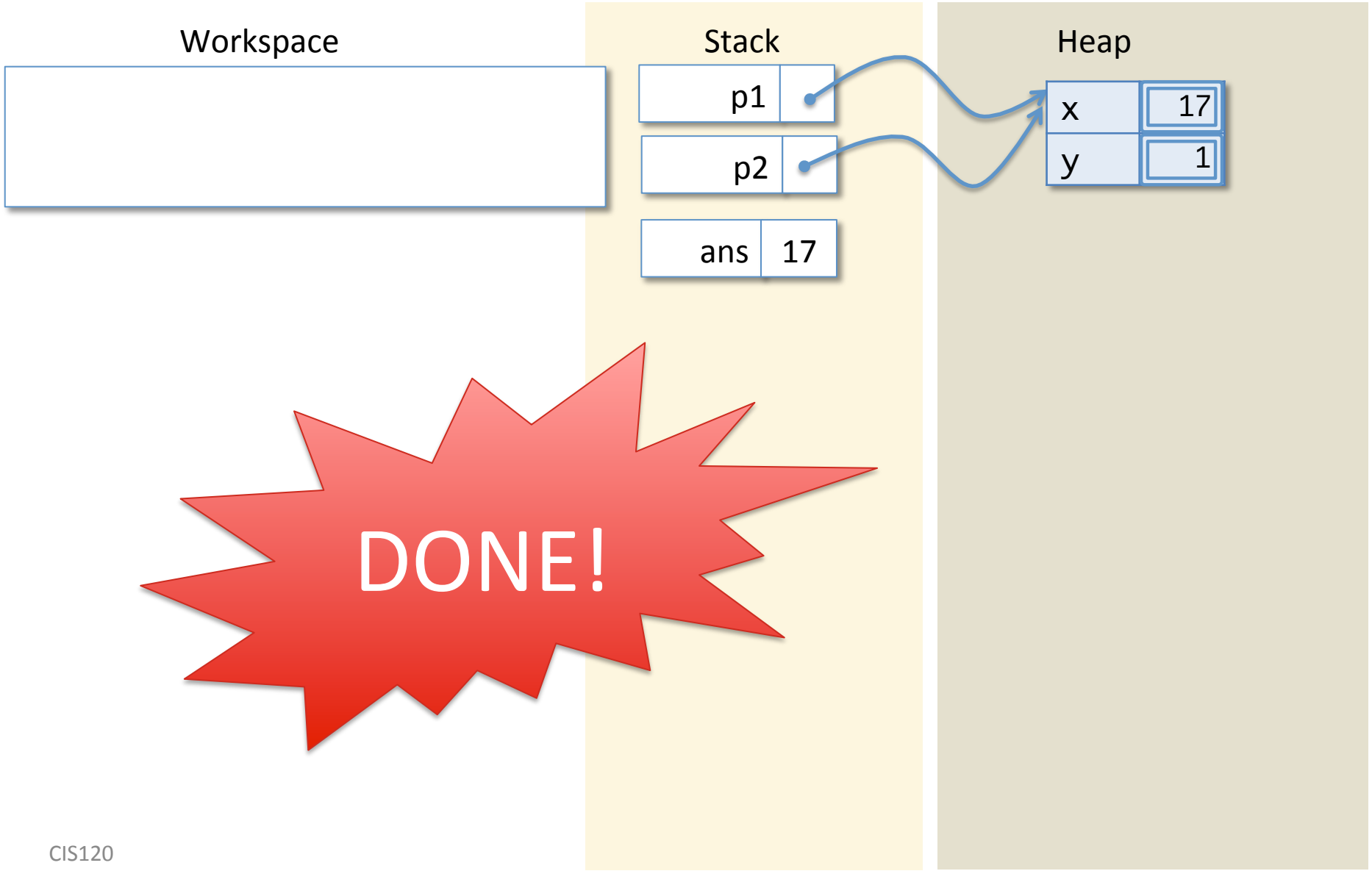
Let Expression

Workspace

```
let ans : int =
    17
```

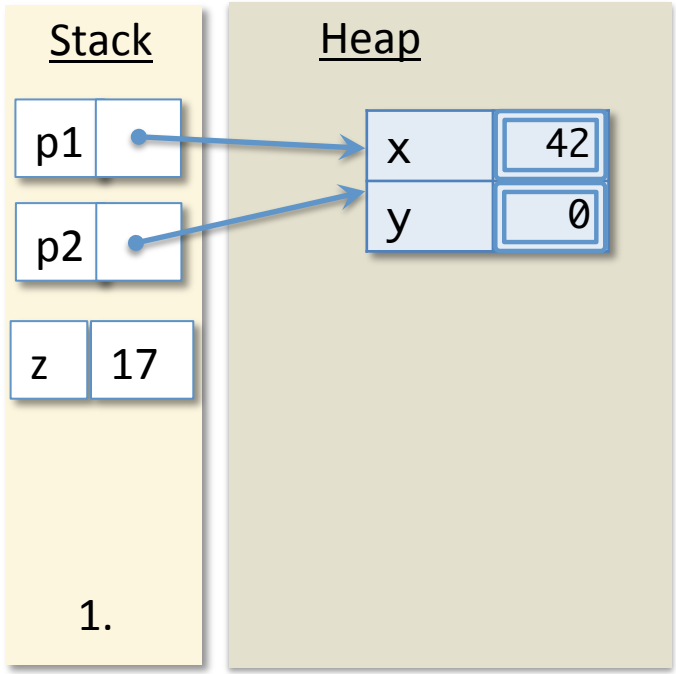


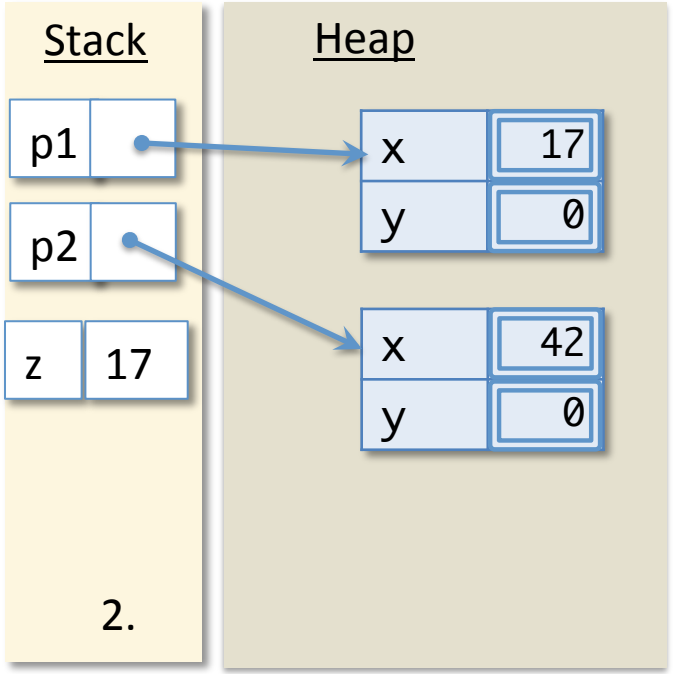
Push ans



What do the Stack and Heap look like after simplifying the following code on the workspace?

```
let p1 = {x=0; y=0} in
let p2 = p1 in
p1.x <- 17;
let z = p1.x in
p2.x <- 42;
p1.x
```





Answer: 1

Simplifying
lists and datatypes using the heap

Simplification

Workspace

```
1::2::3::[]
```

Stack

Heap

For uniformity, we'll pretend lists are declared like this:

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list
```

Simplification

Workspace

```
Cons (1,Cons (2,Cons (3,Nil)))
```

Stack

Heap

For uniformity, we'll pretend lists are declared like this:

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list
```

Simplification

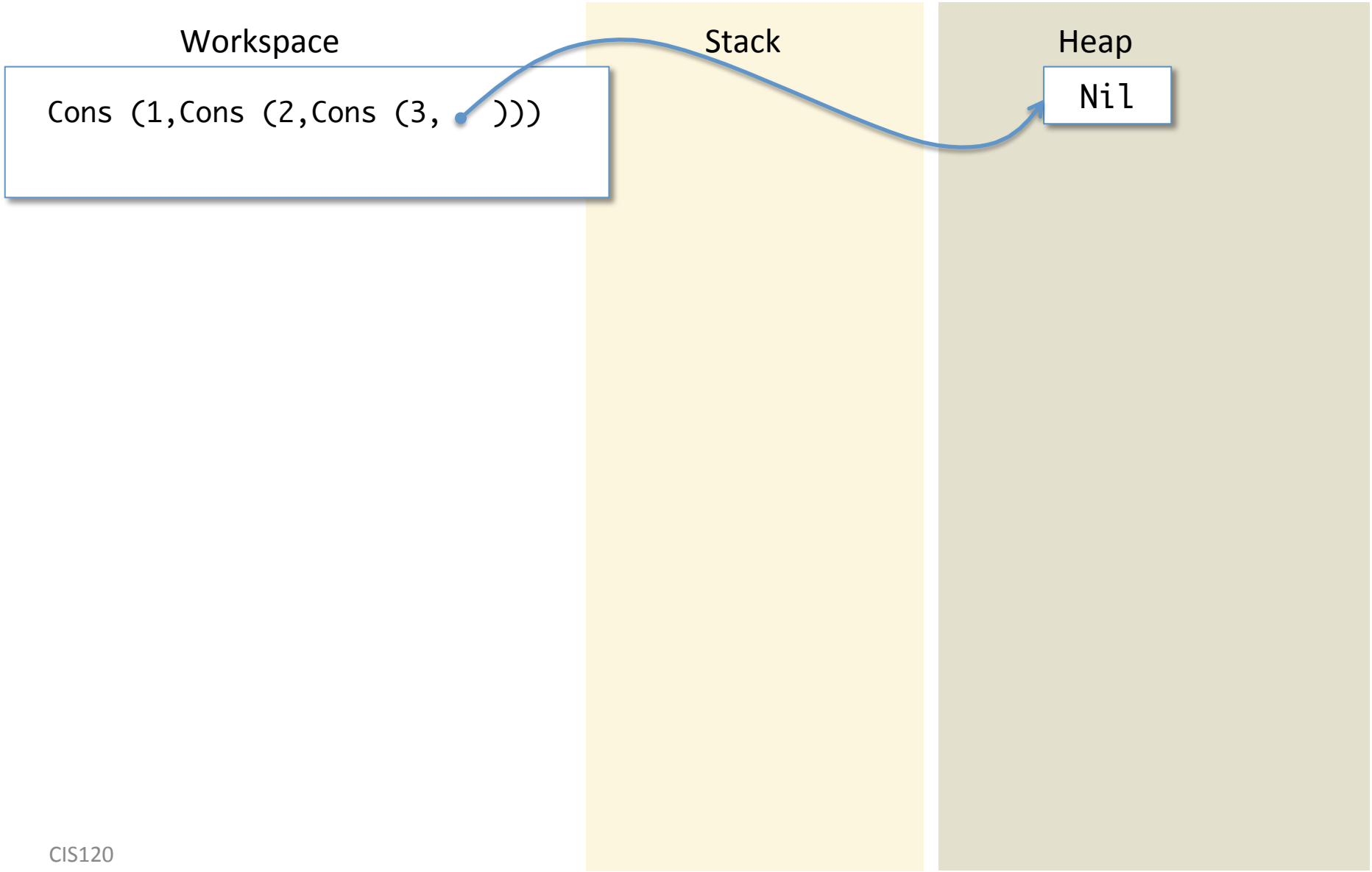
Workspace

```
Cons (1,Cons (2,Cons (3,Nil)))
```

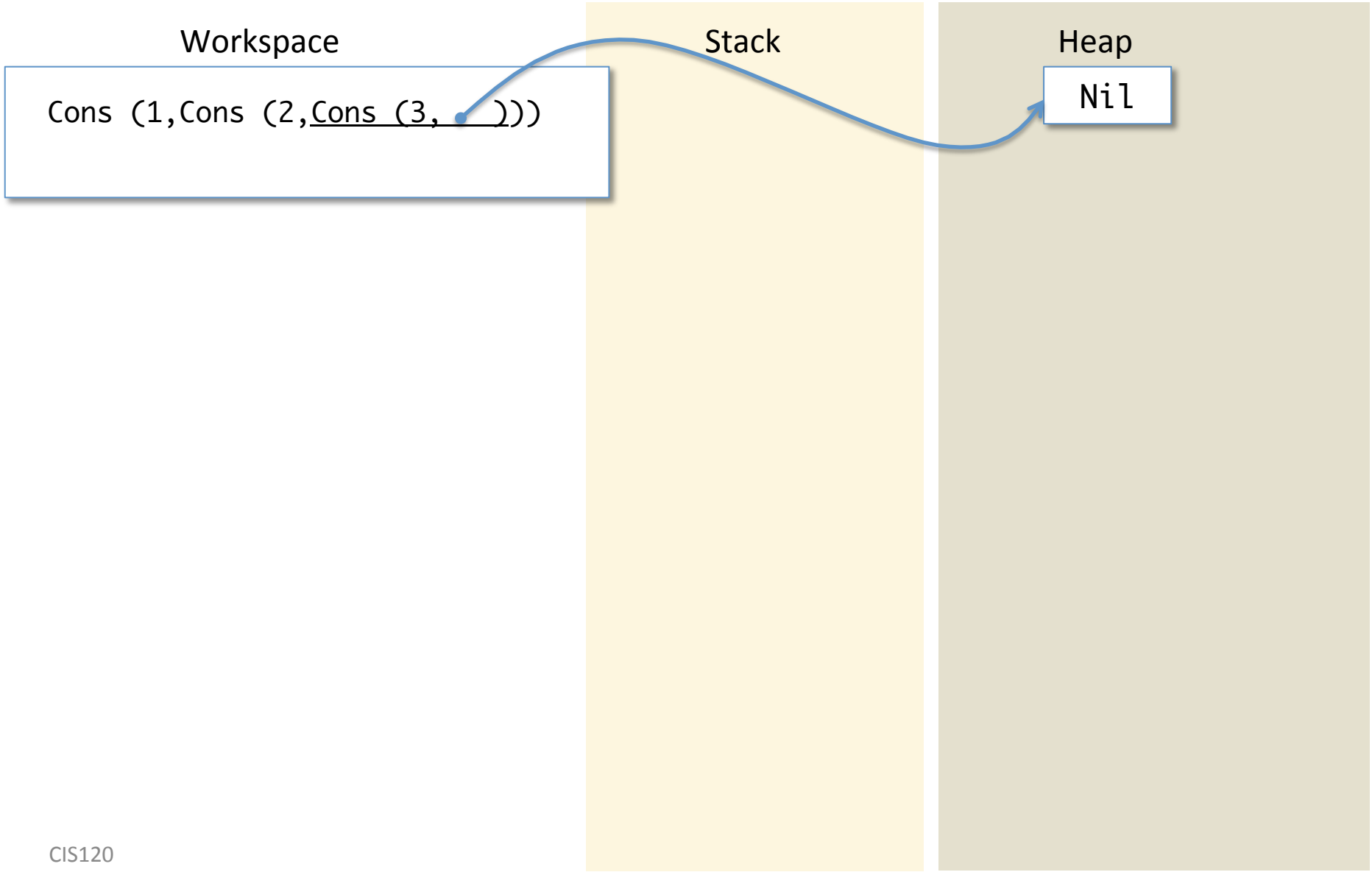
Stack

Heap

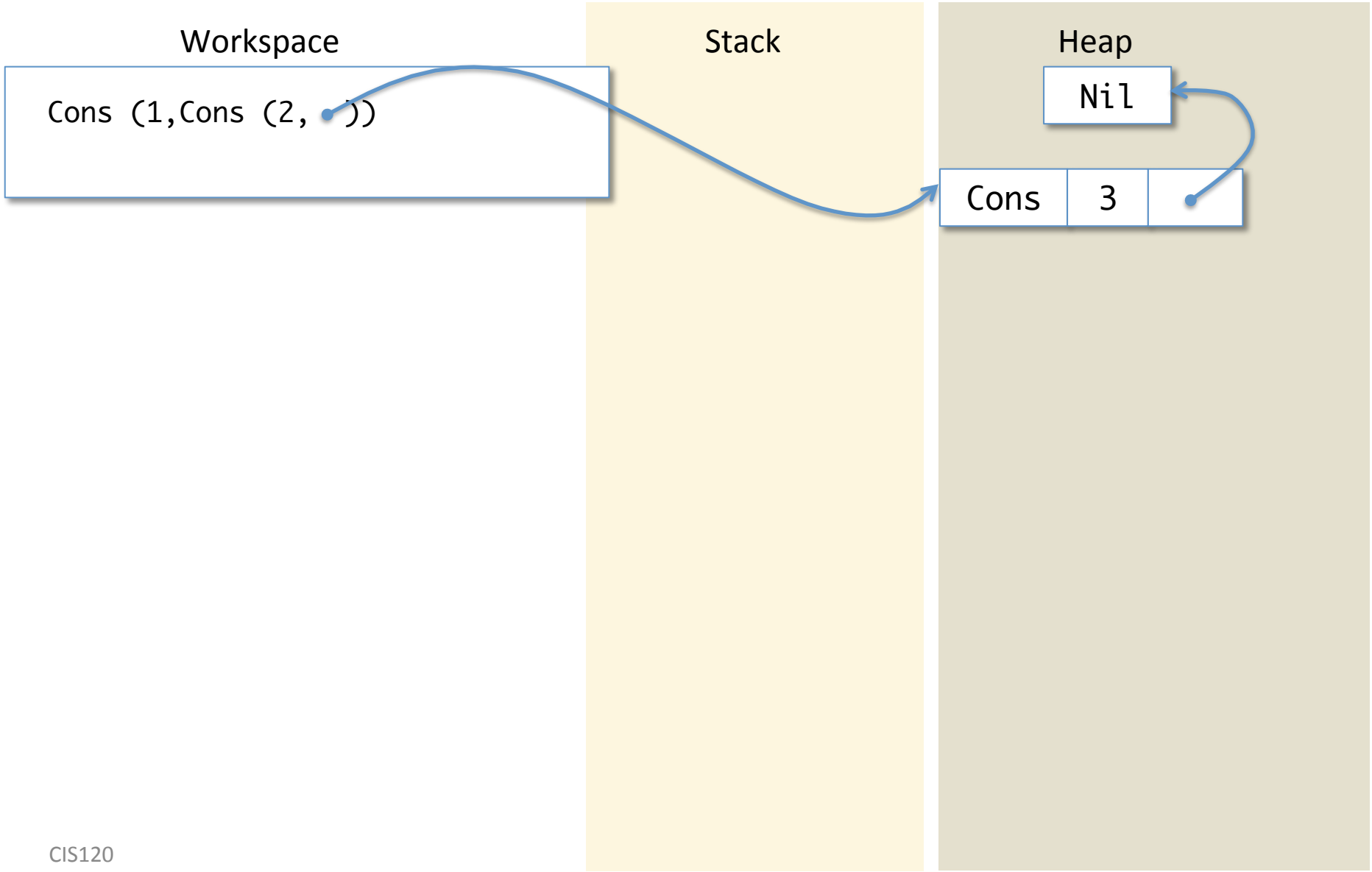
Simplification



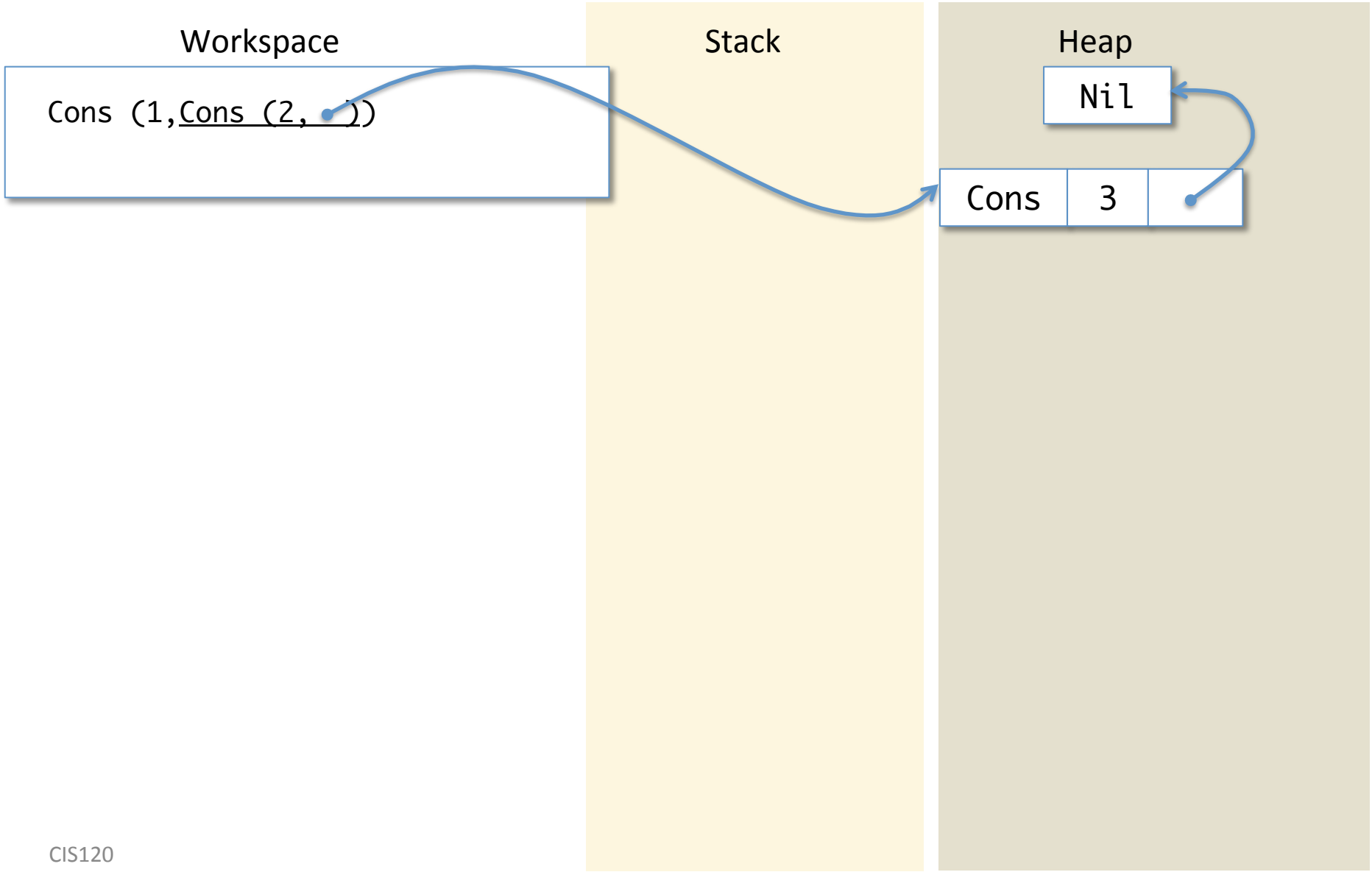
Simplification



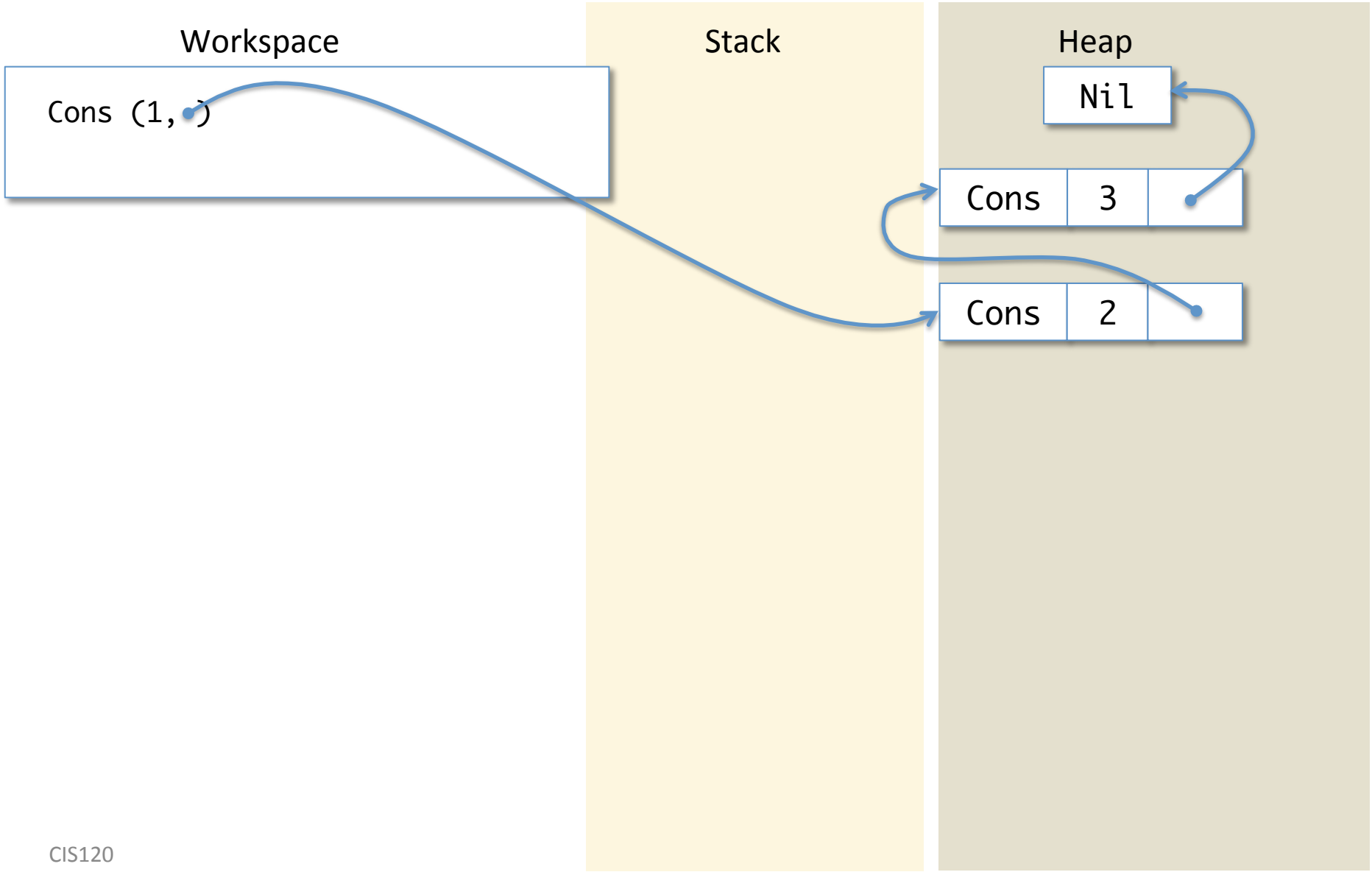
Simplification



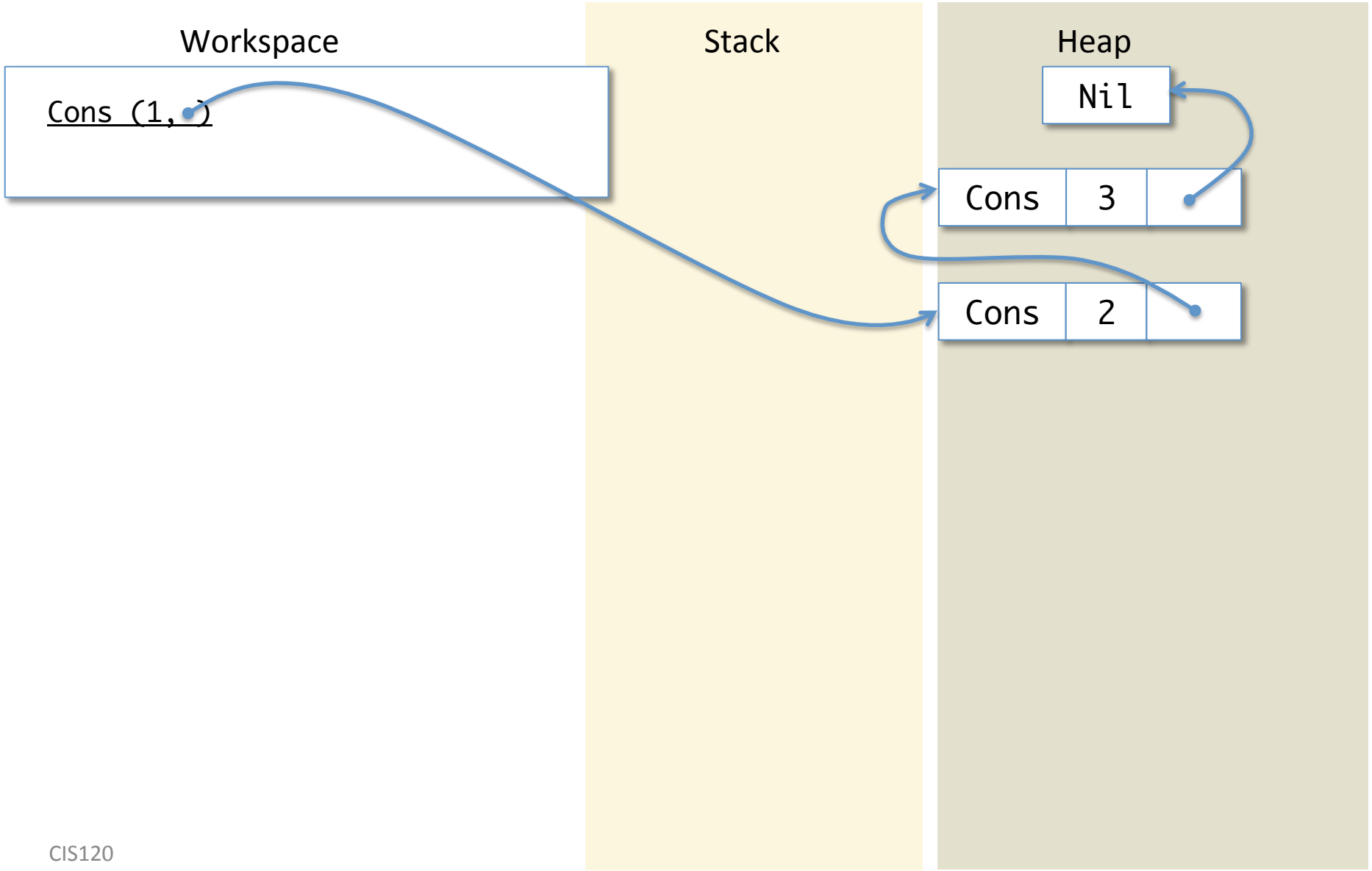
Simplification



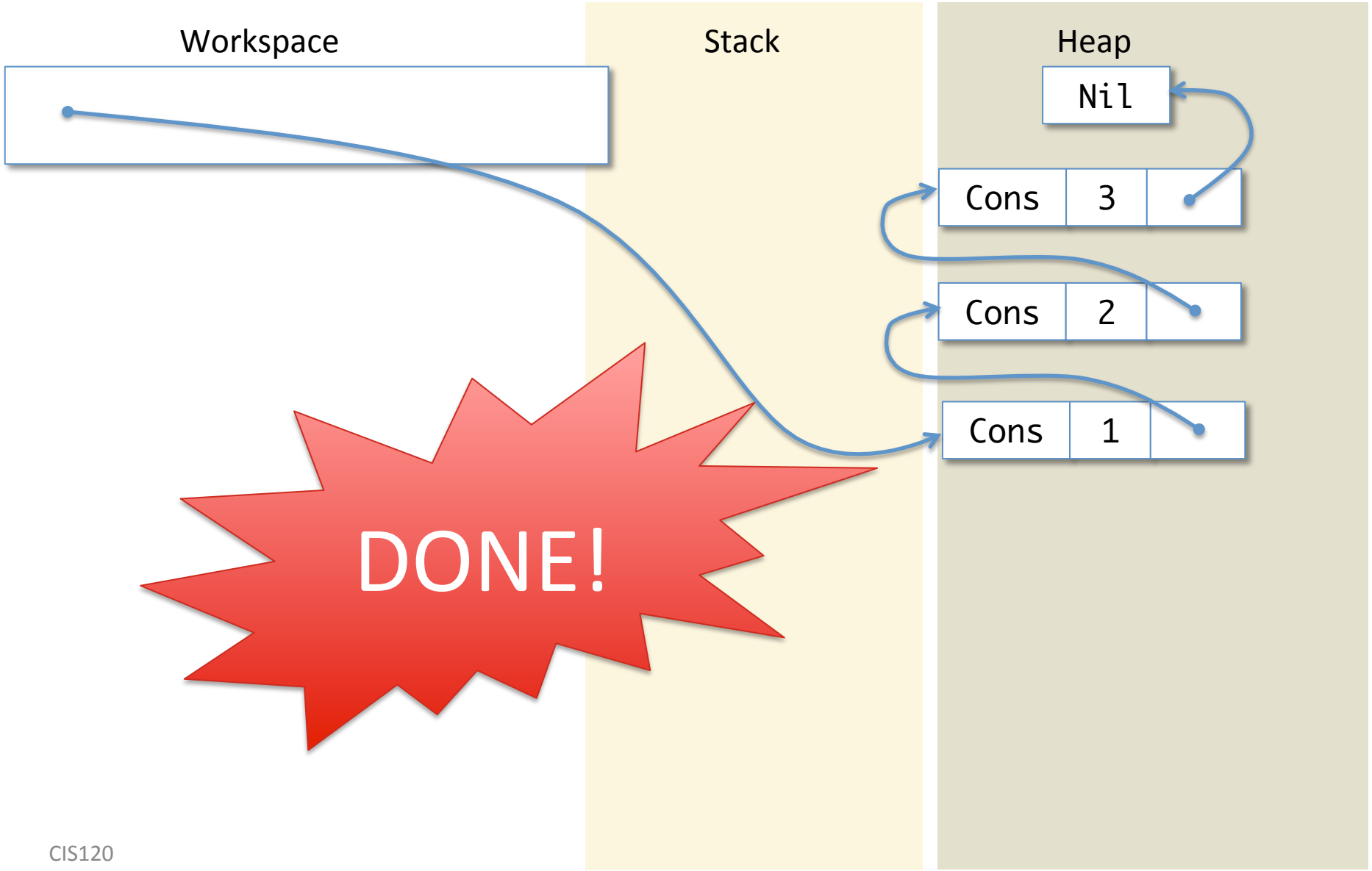
Simplification



Simplification



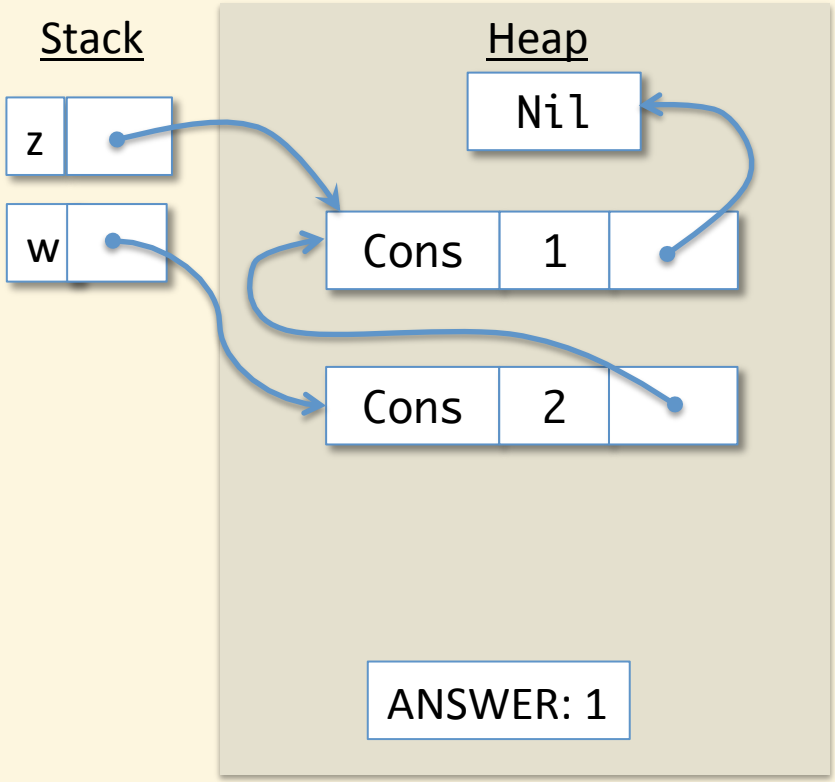
Simplification



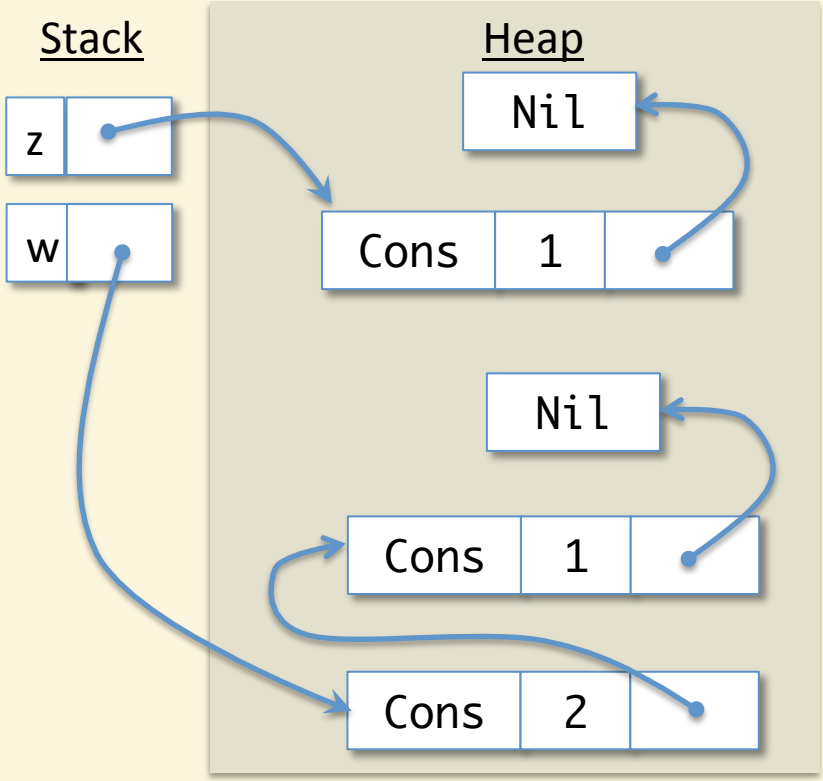
What do the Stack and Heap look like after simplifying the following code on the workspace?

```
let z = Cons (1, Nil) in
let w = Cons (2, z) in
    w
```

1.



2.



Simplifying functions

Function Simplification

Workspace

```
let add1 (x : int) : int =
  x + 1 in
add1 (add1 0)
```

Stack

Heap

Function Simplification

Workspace

```
let add1 (x : int) : int =
  x + 1 in
add1 (add1 0)
```

Stack

Heap

Function Simplification

Workspace

```
let add1 : int ->  int =
  fun (x:int) -> x + 1 in
add1 (add1 0)
```

Stack

Heap

Function Simplification

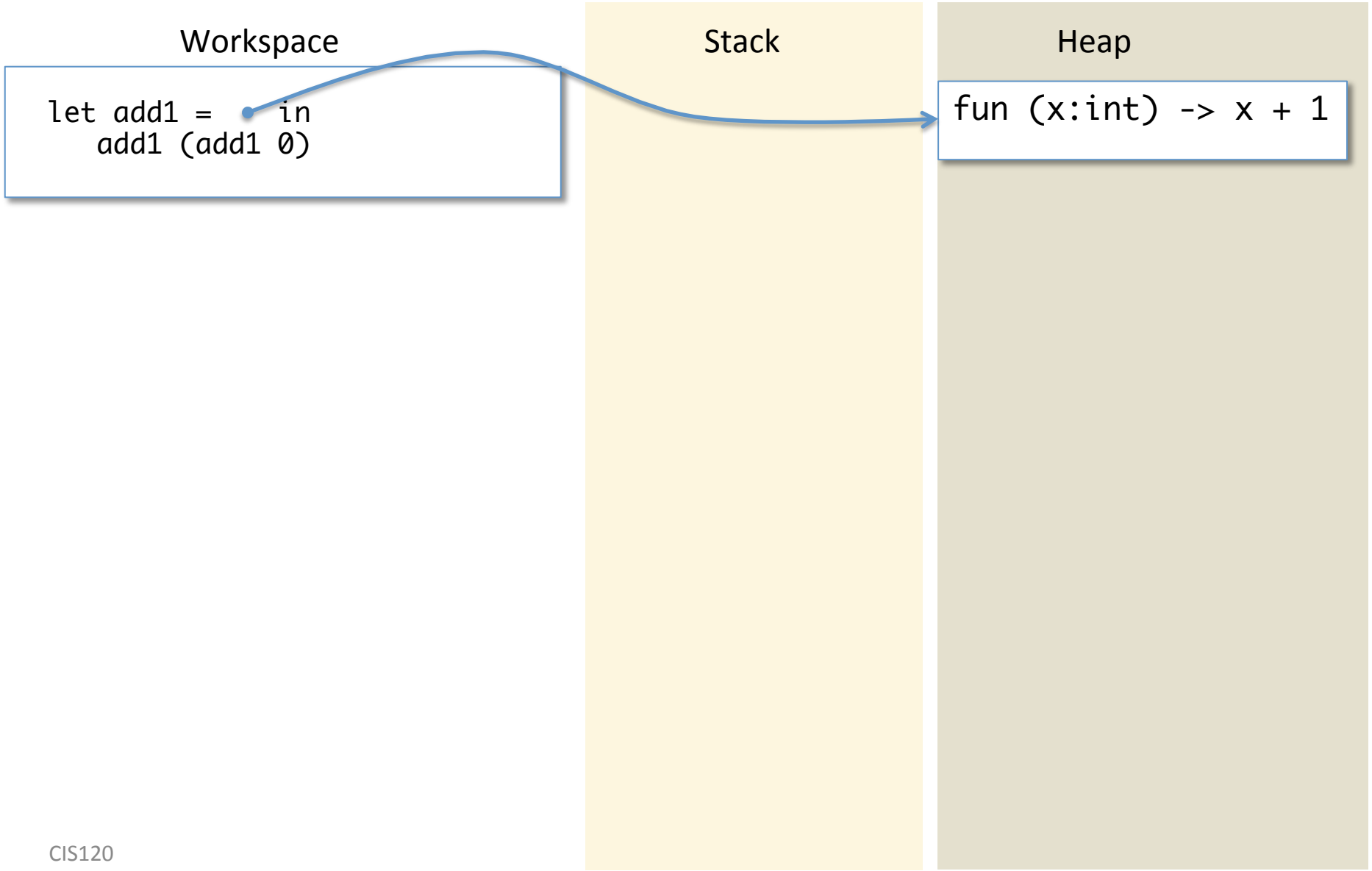
Workspace

```
let add1 : int ->  int =
  fun (x:int) -> x + 1 in
add1 (add1 0)
```

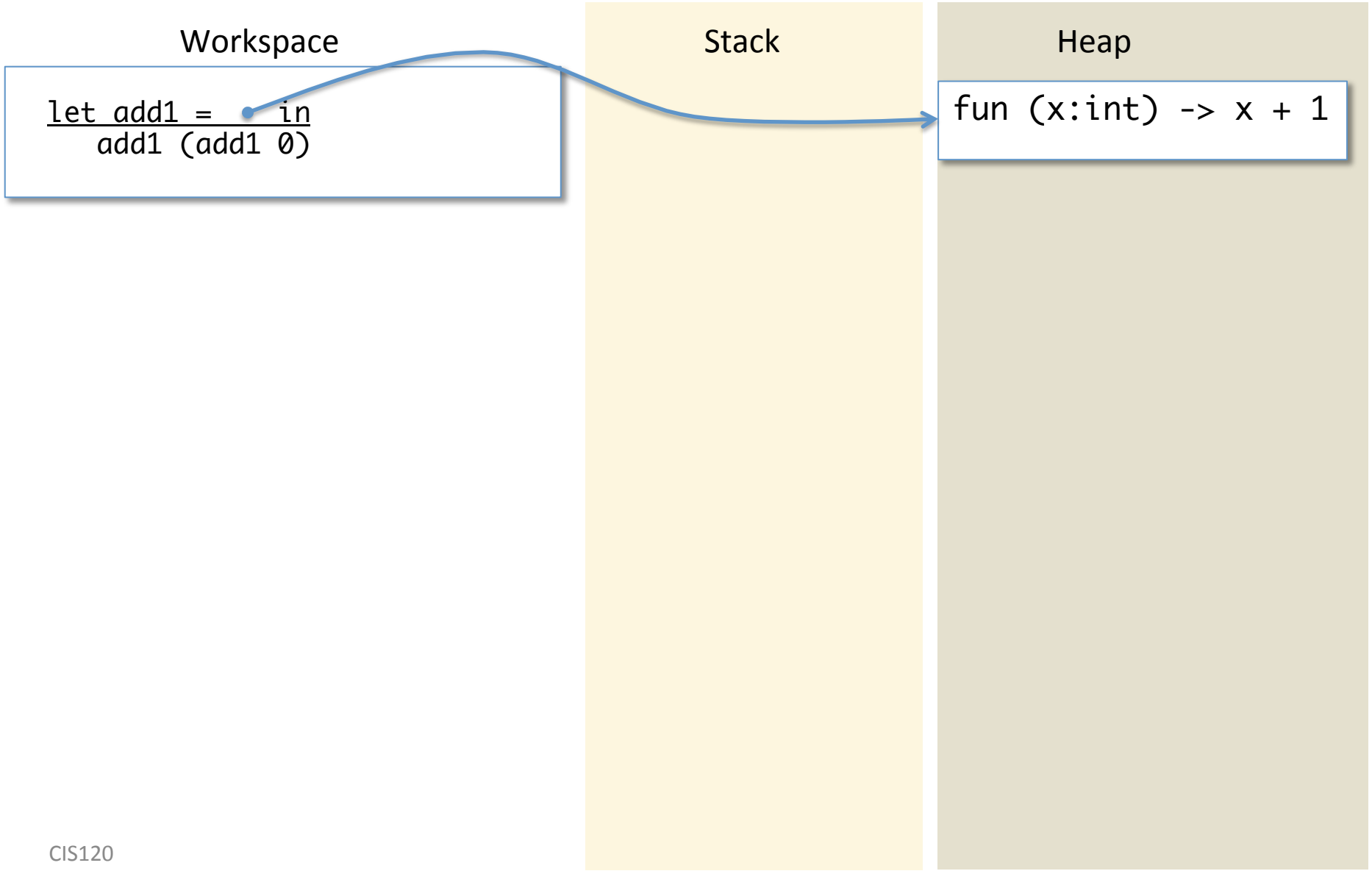
Stack

Heap

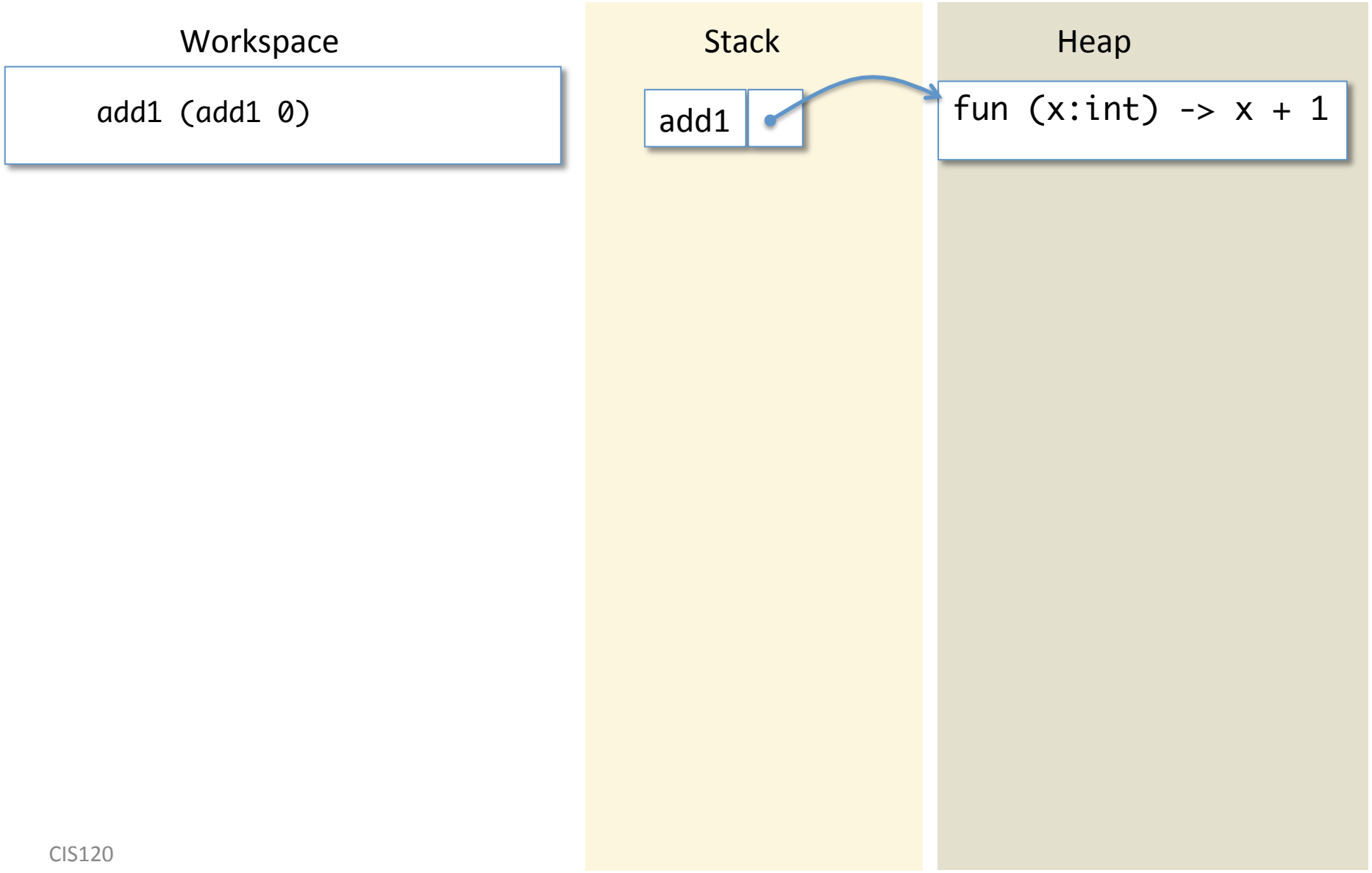
Function Simplification



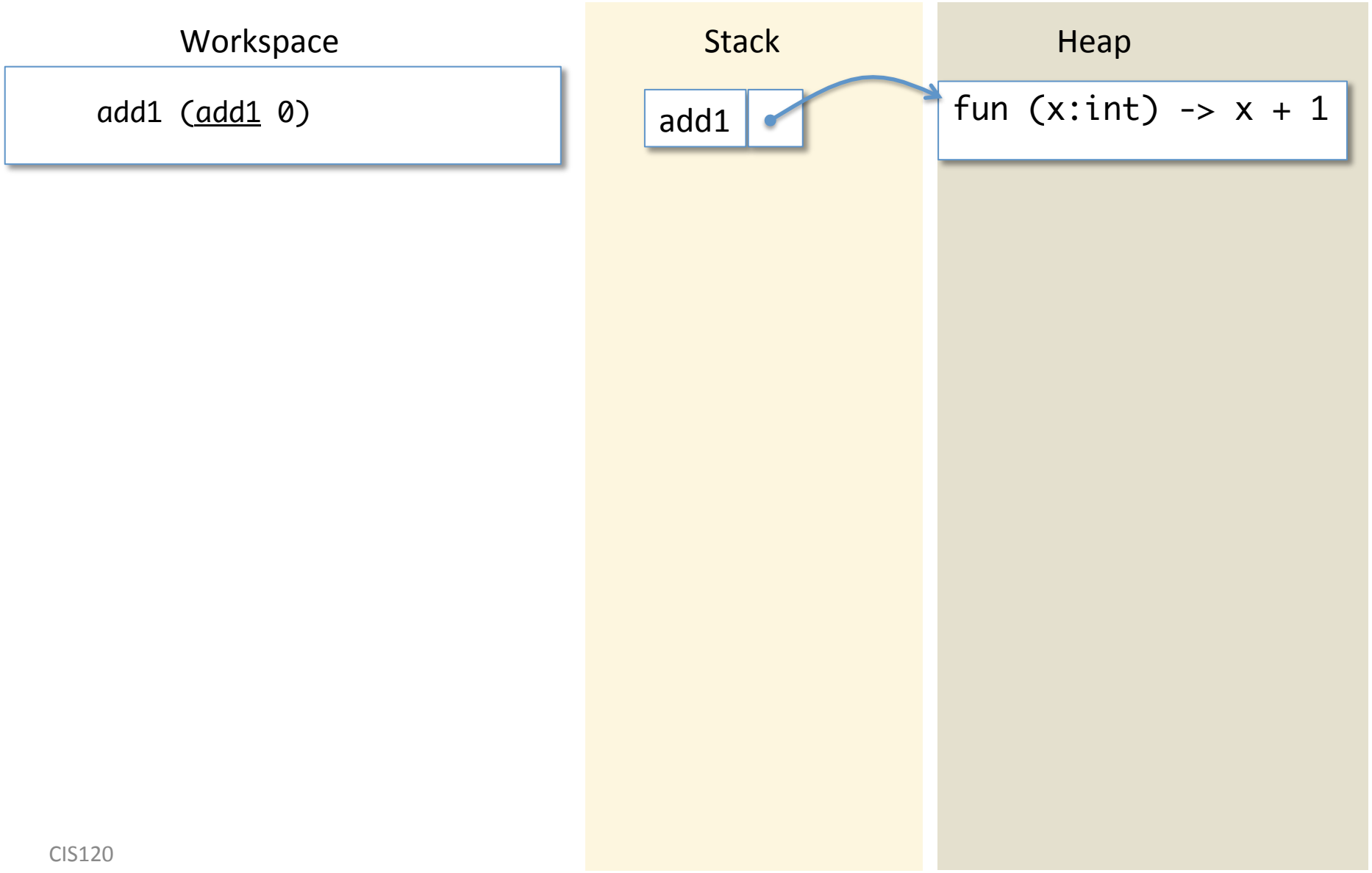
Function Simplification



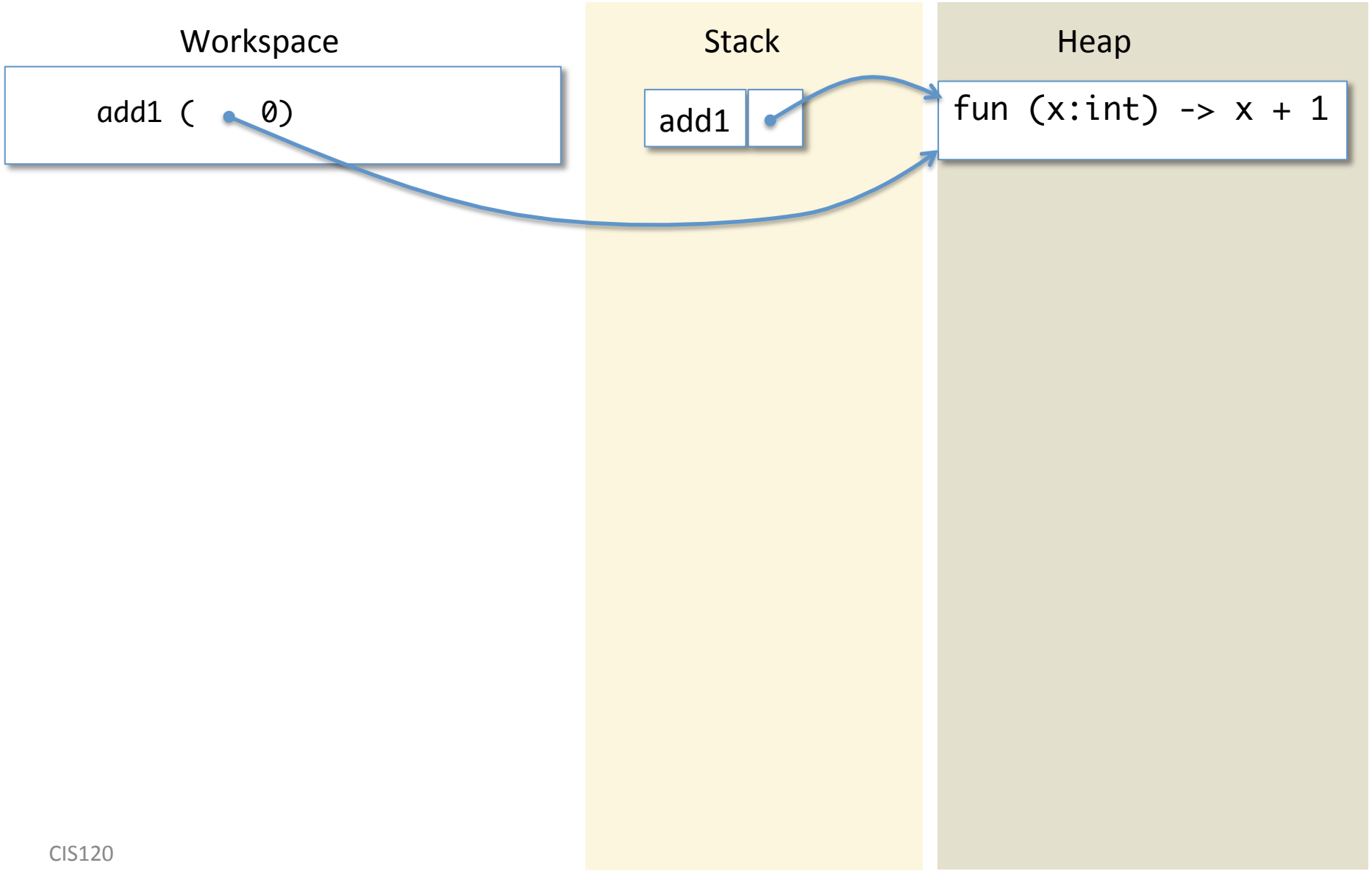
Function Simplification



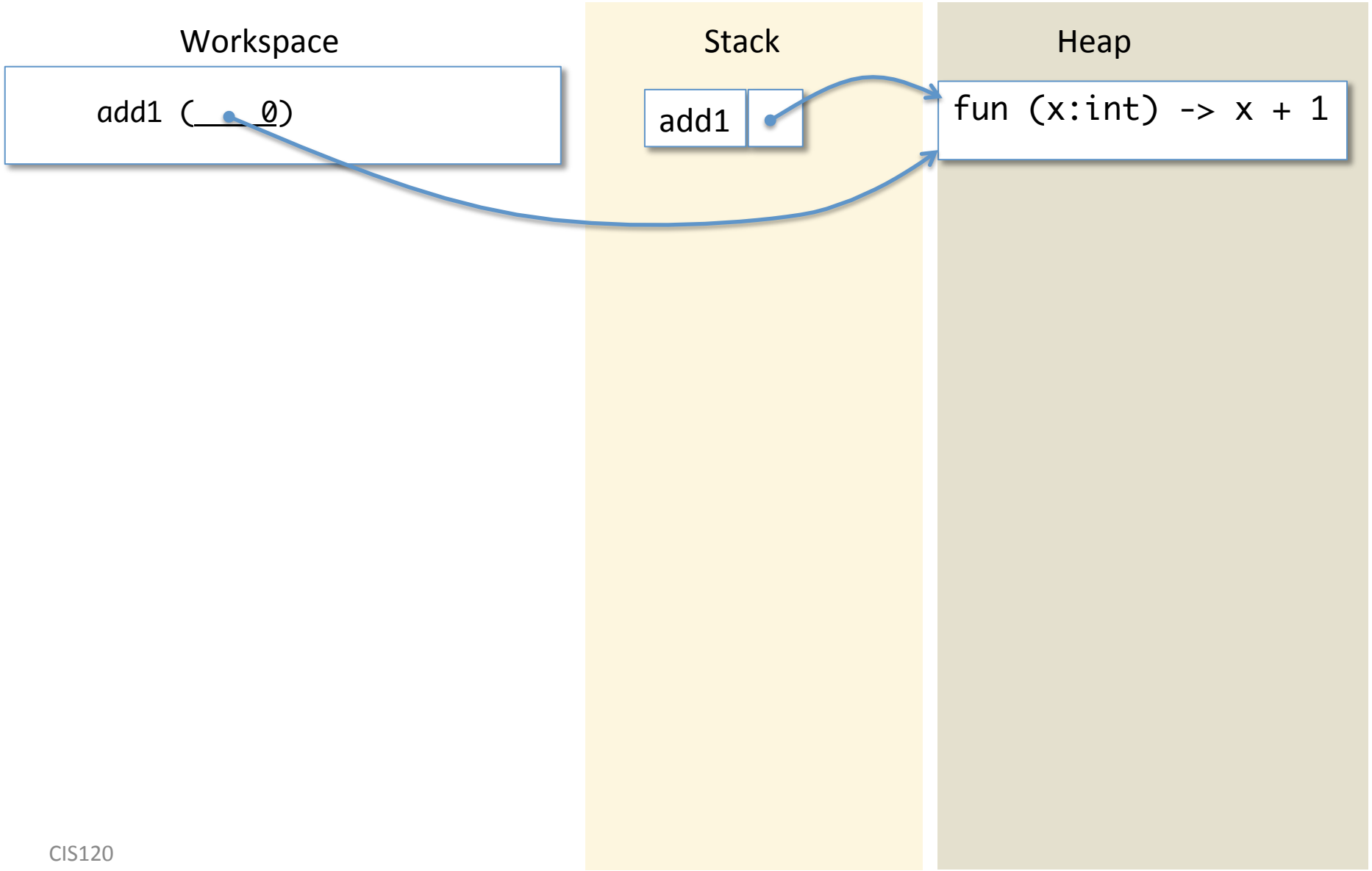
Function Simplification



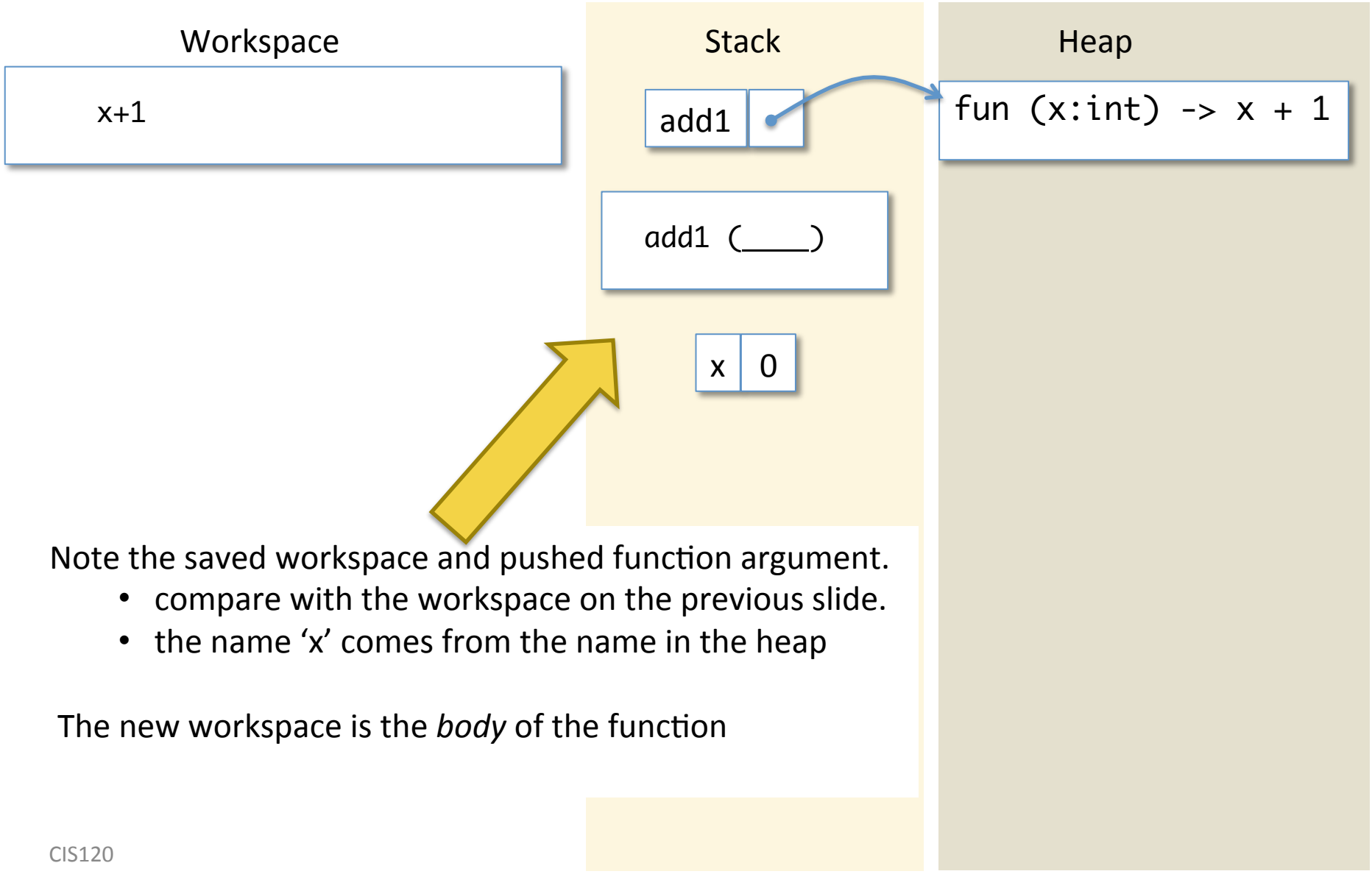
Function Simplification



Function Simplification



Do the Call, Saving the Workspace

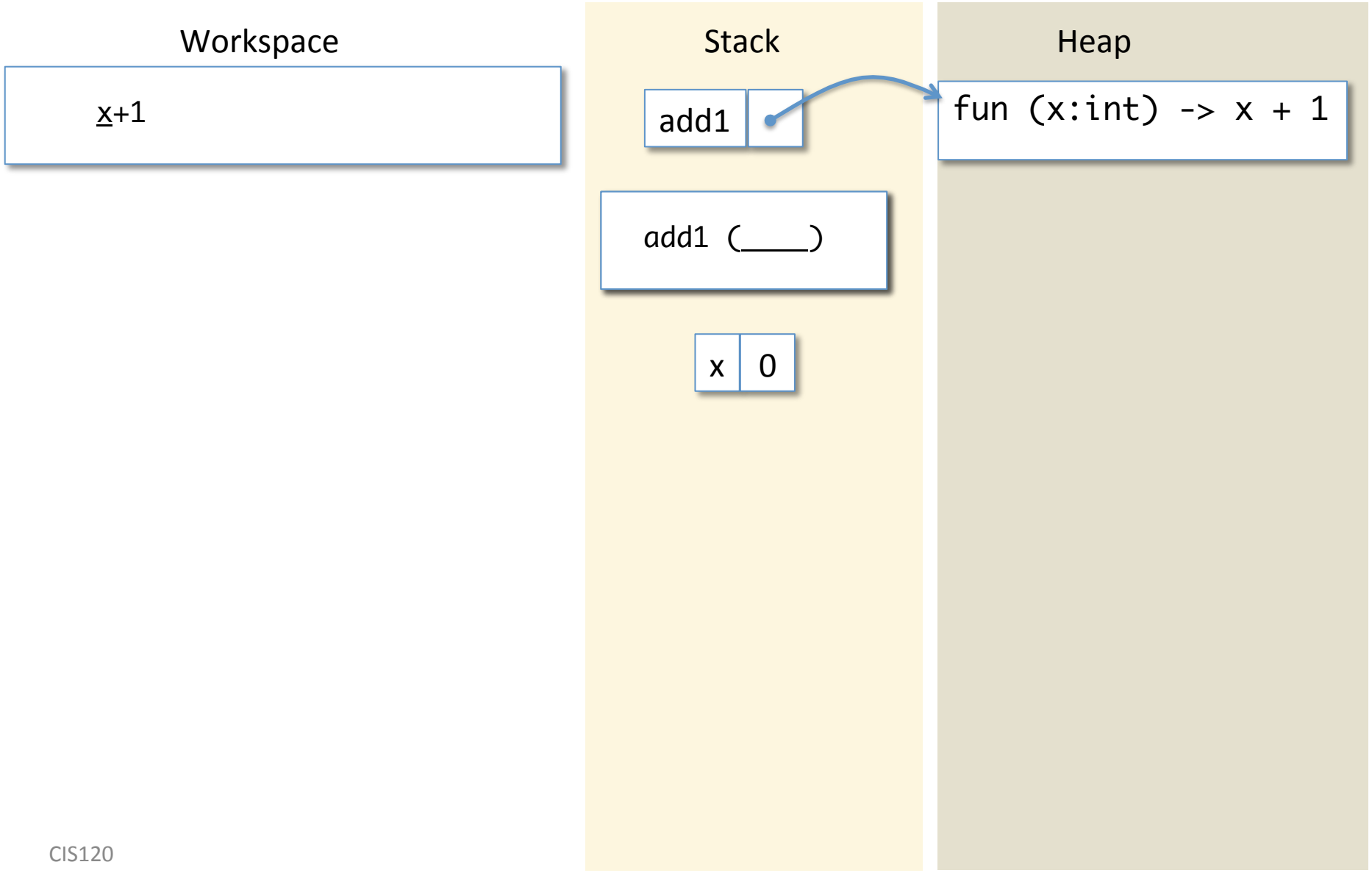


Note the saved workspace and pushed function argument.

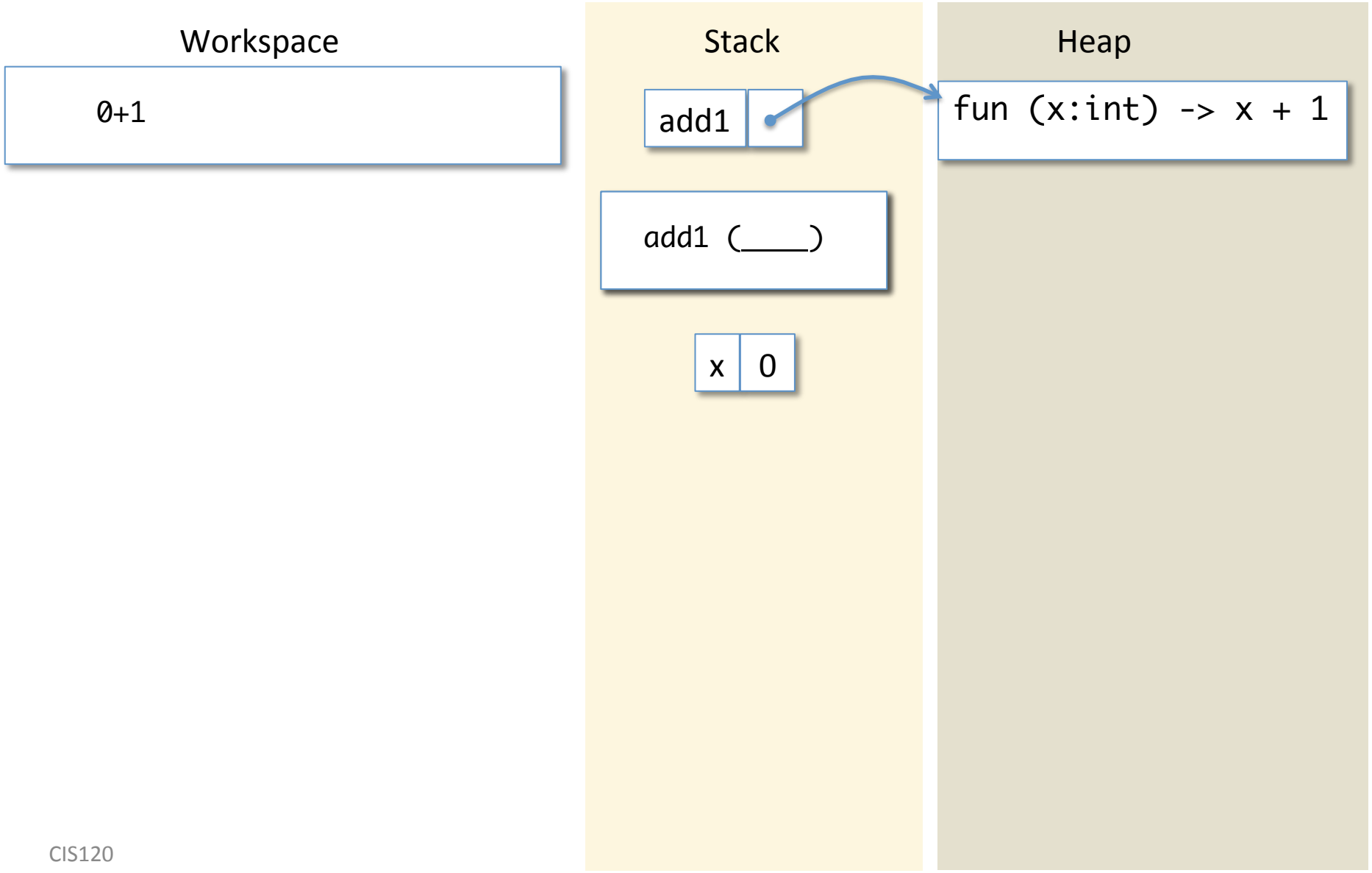
- compare with the workspace on the previous slide.
- the name 'x' comes from the name in the heap

The new workspace is the *body* of the function

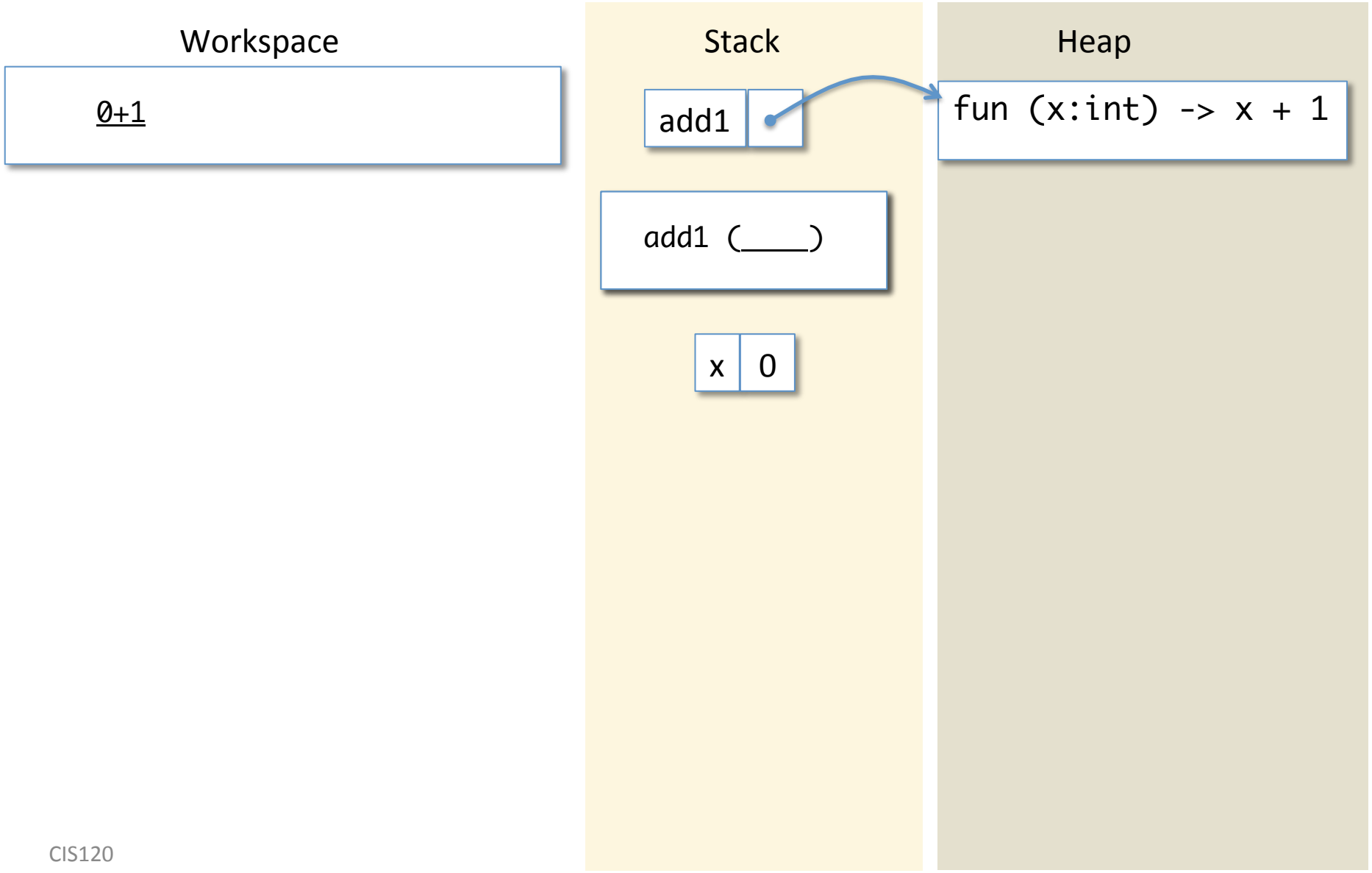
Function Simplification



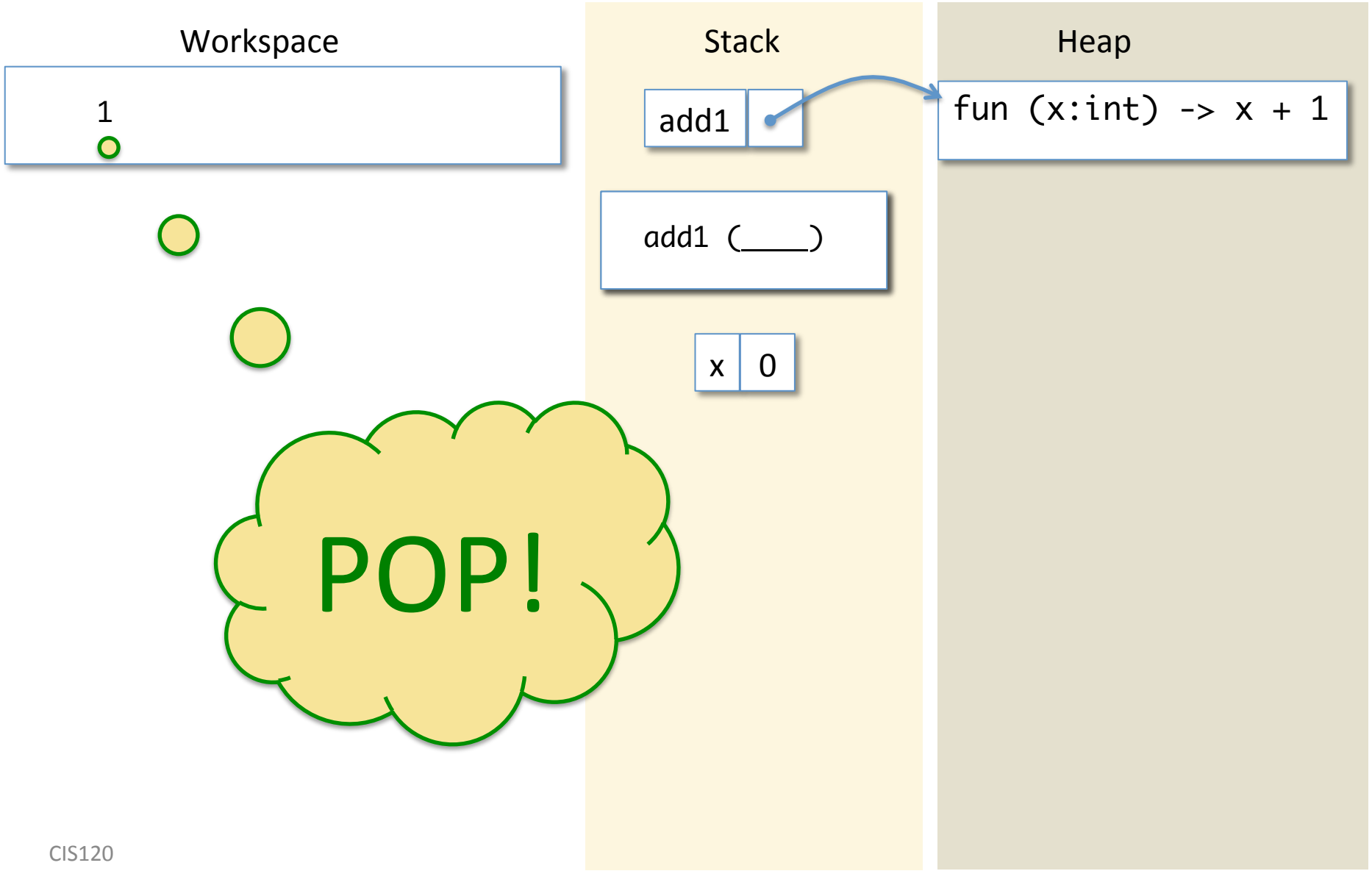
Function Simplification



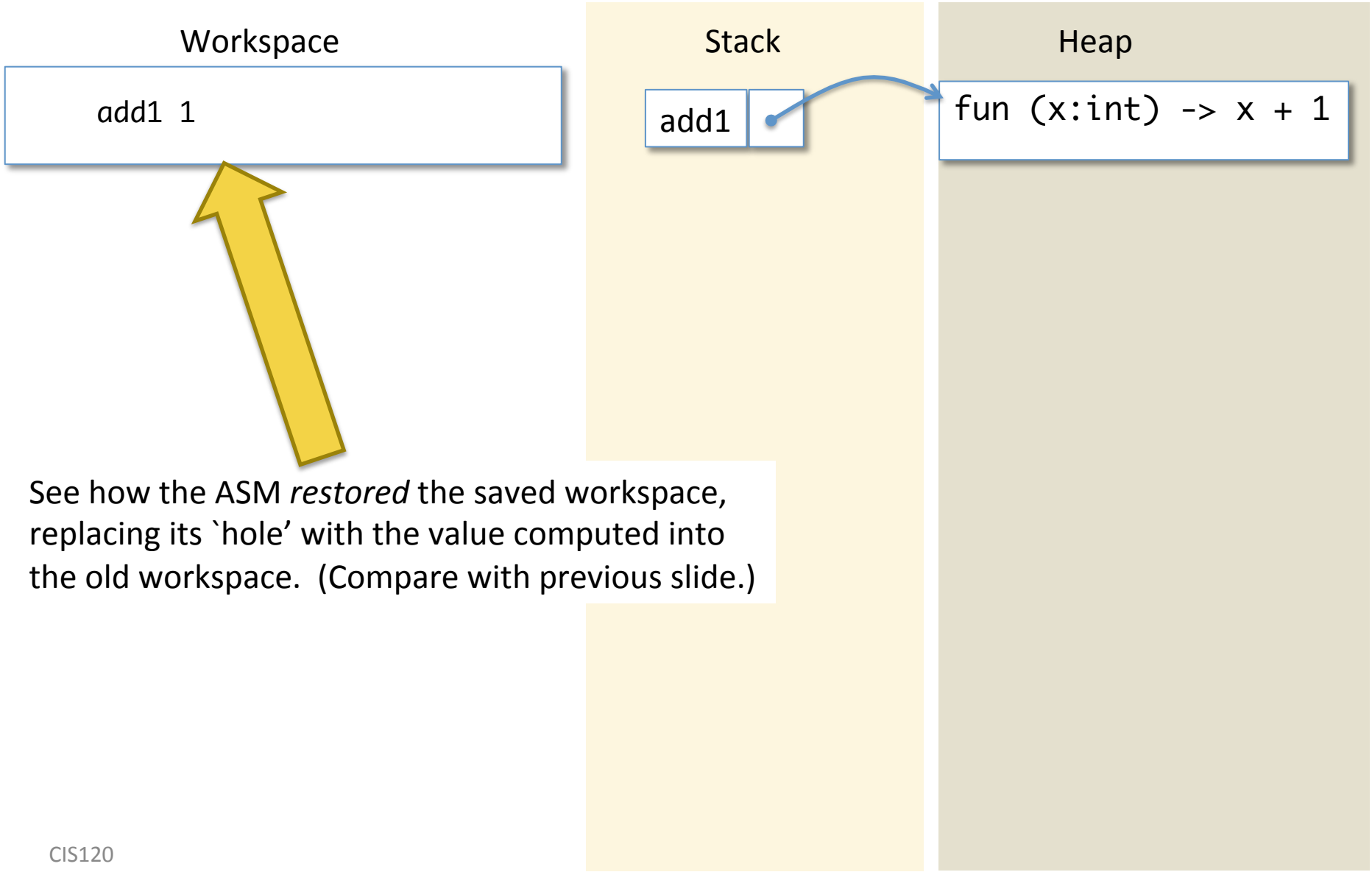
Function Simplification



Function Simplification

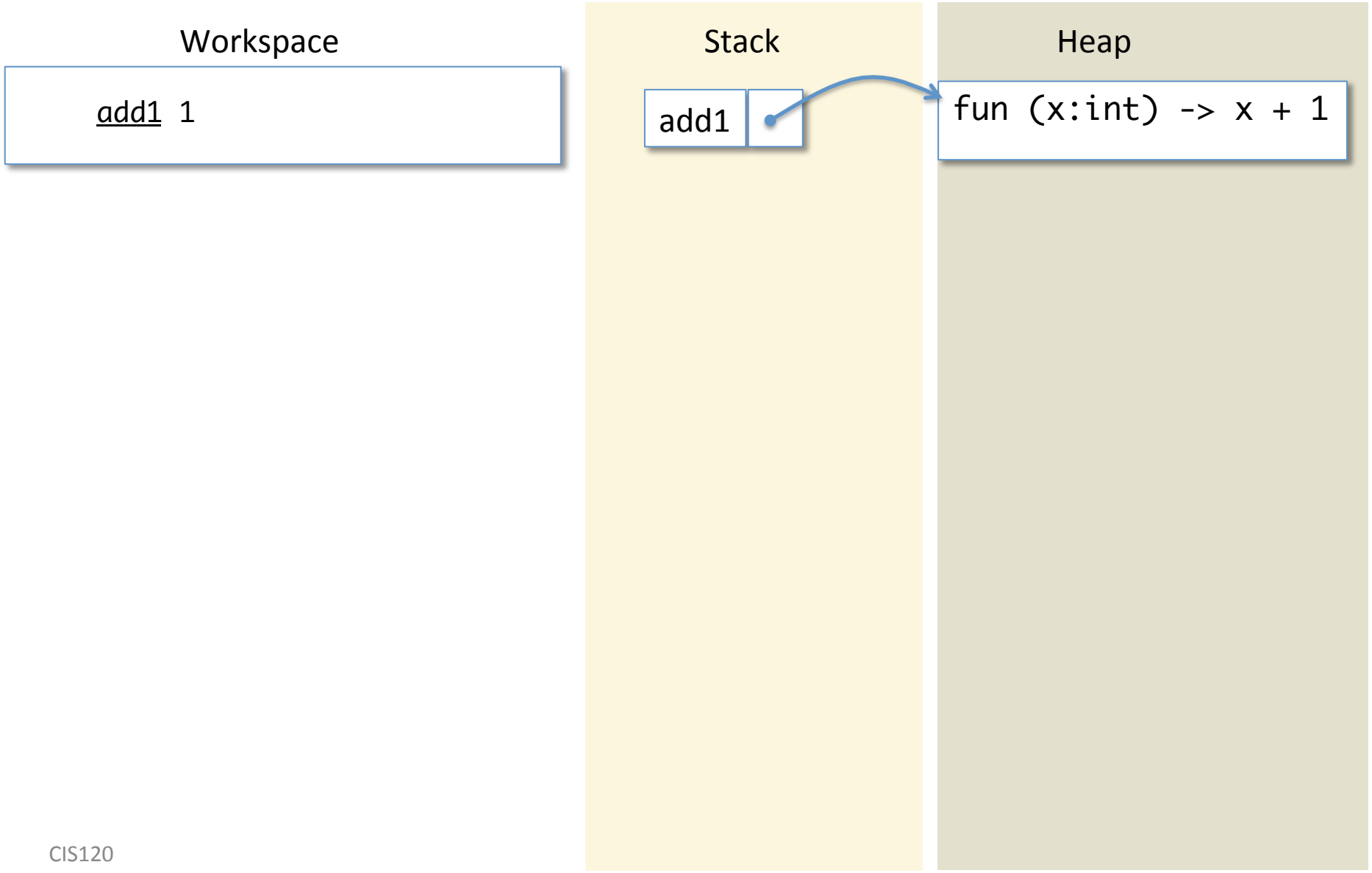


Function Simplification

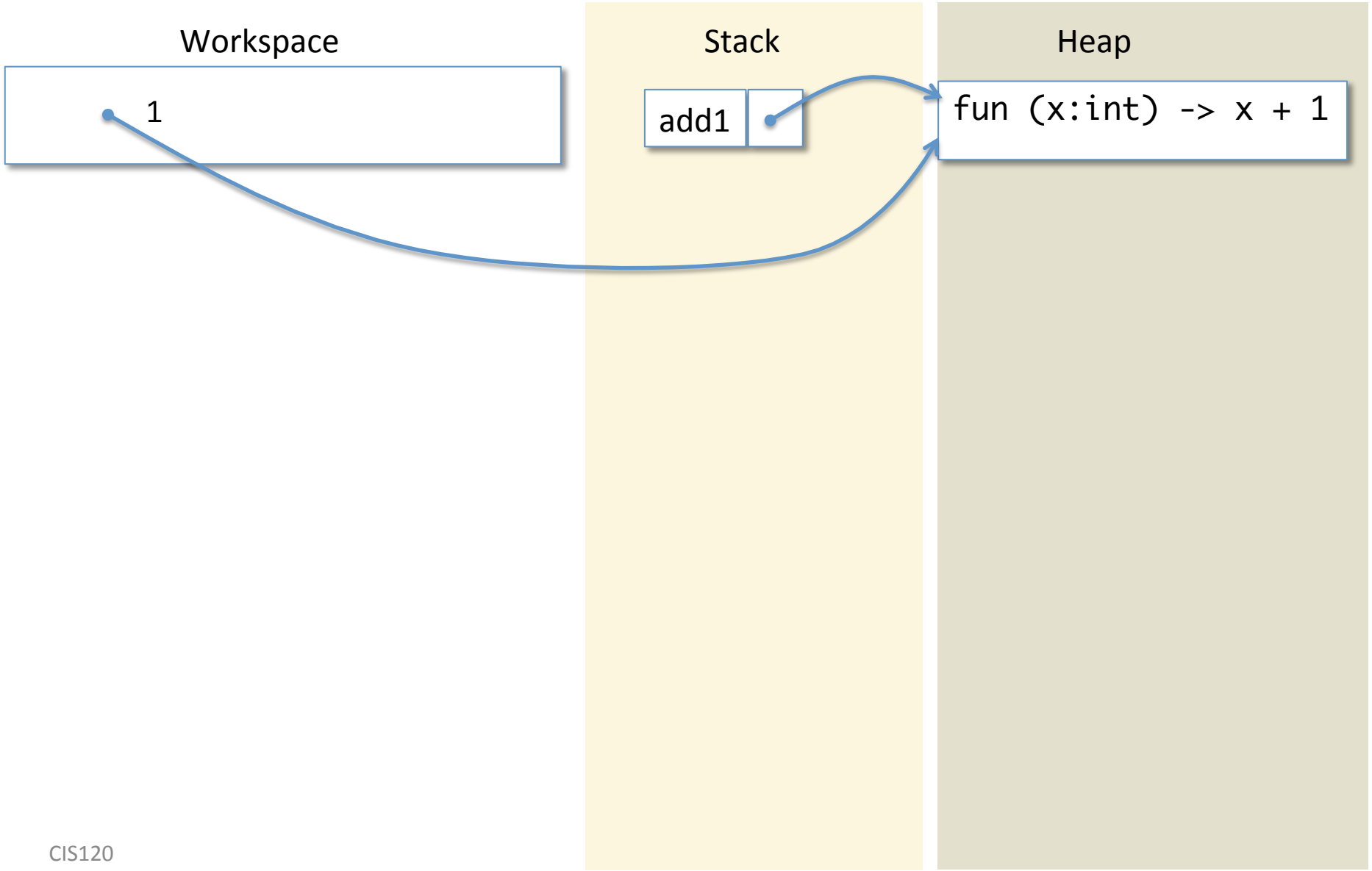


See how the ASM *restored* the saved workspace, replacing its `hole' with the value computed into the old workspace. (Compare with previous slide.)

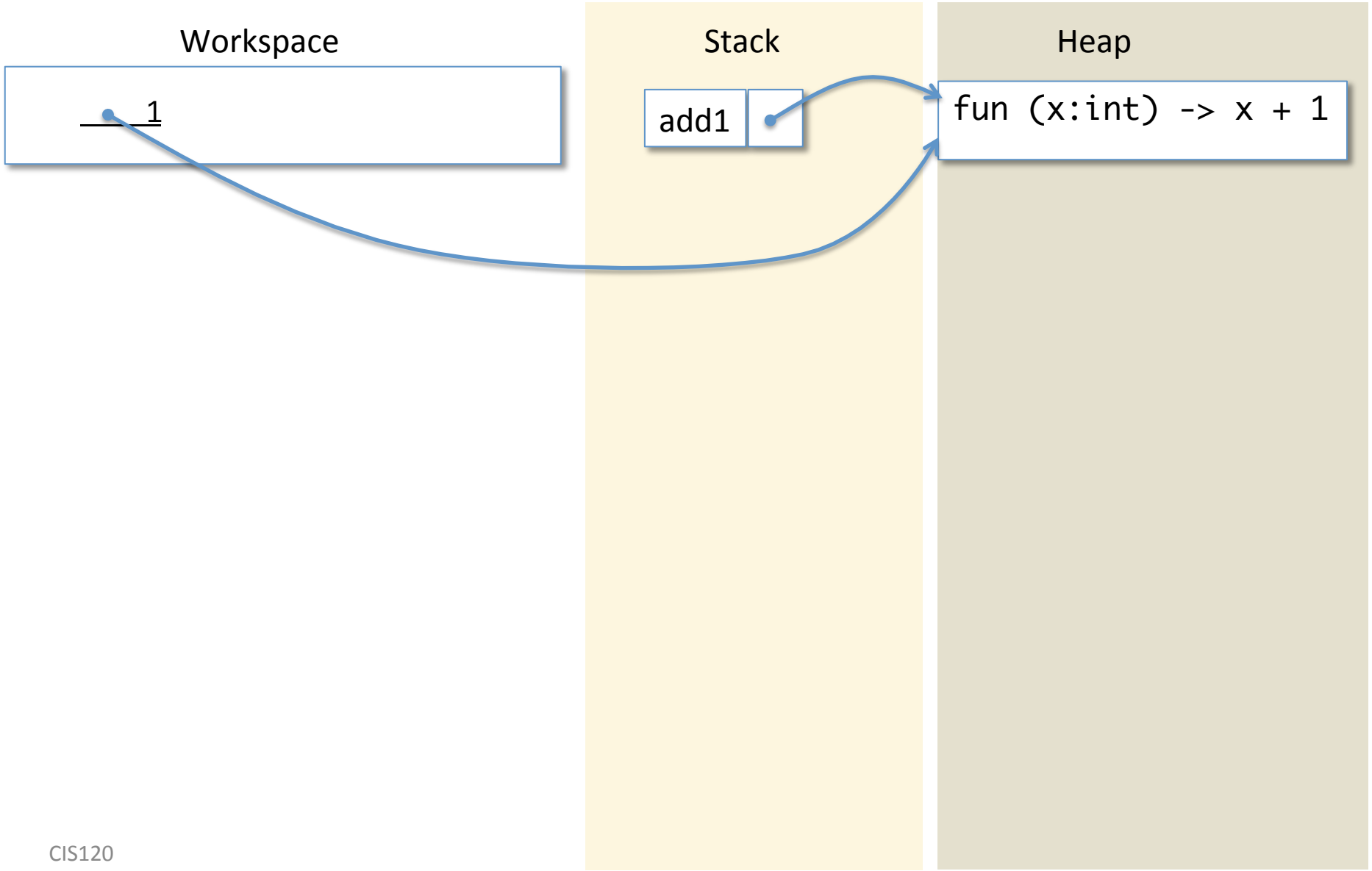
Function Simplification



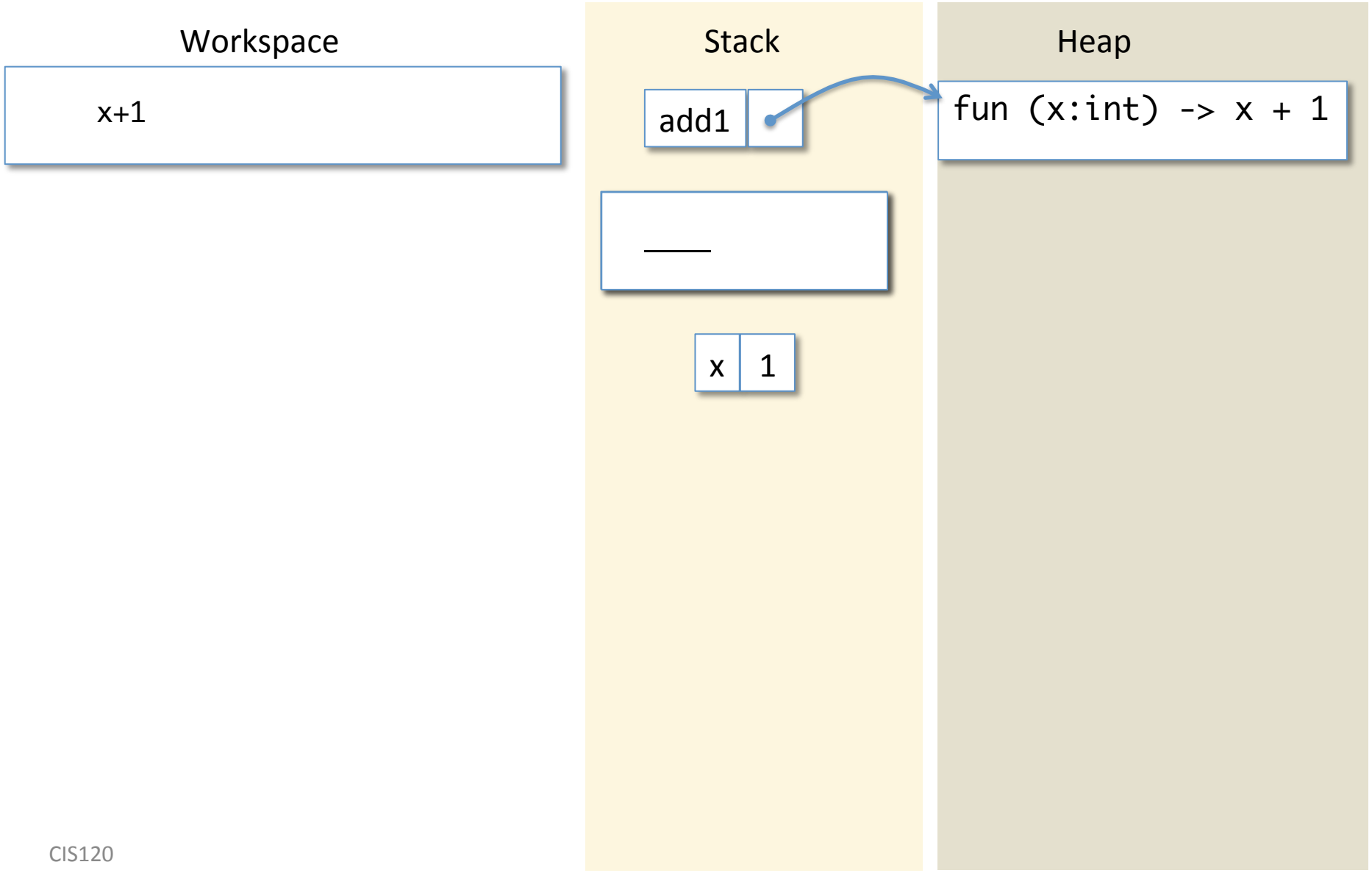
Function Simplification



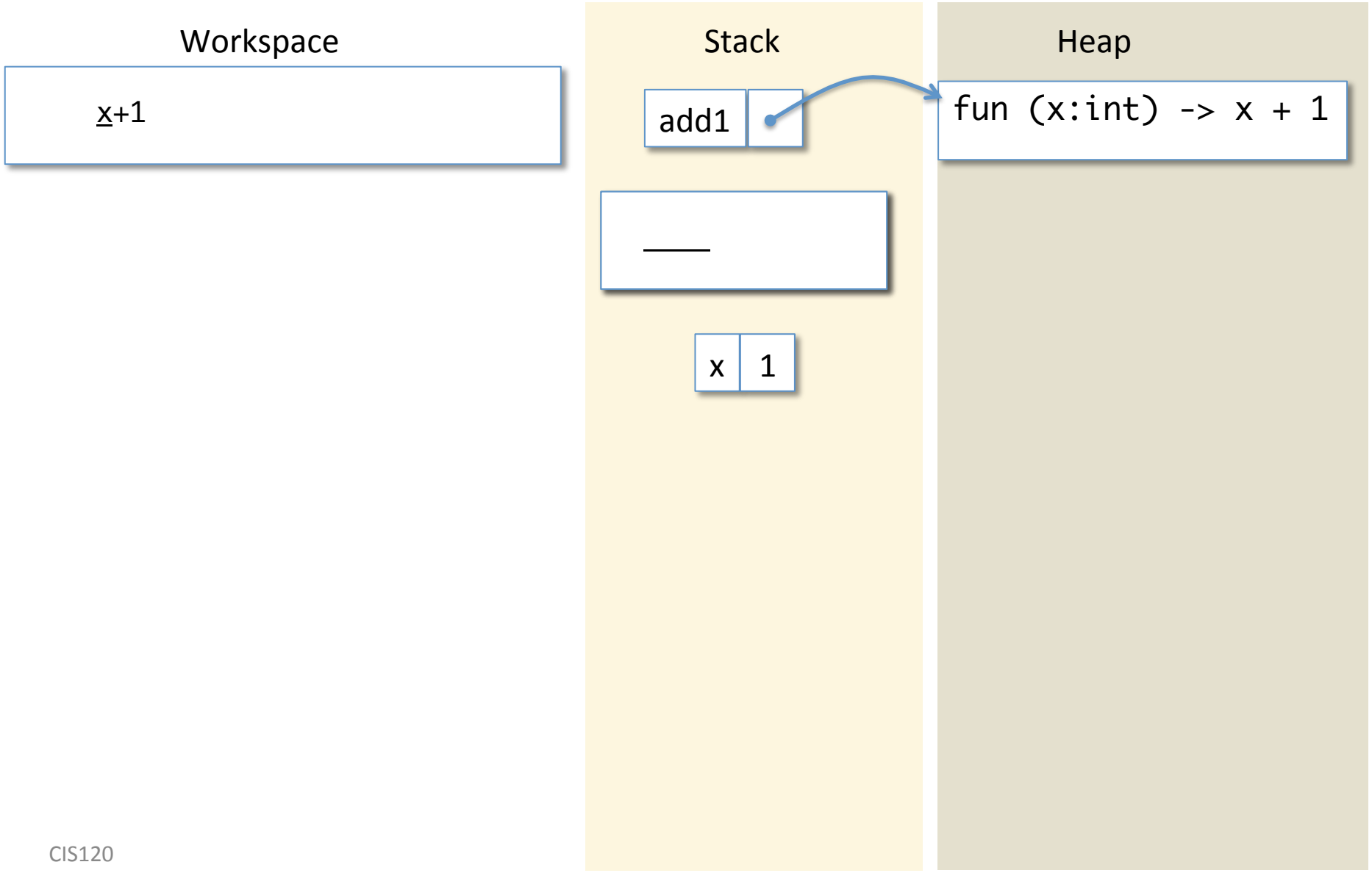
Function Simplification



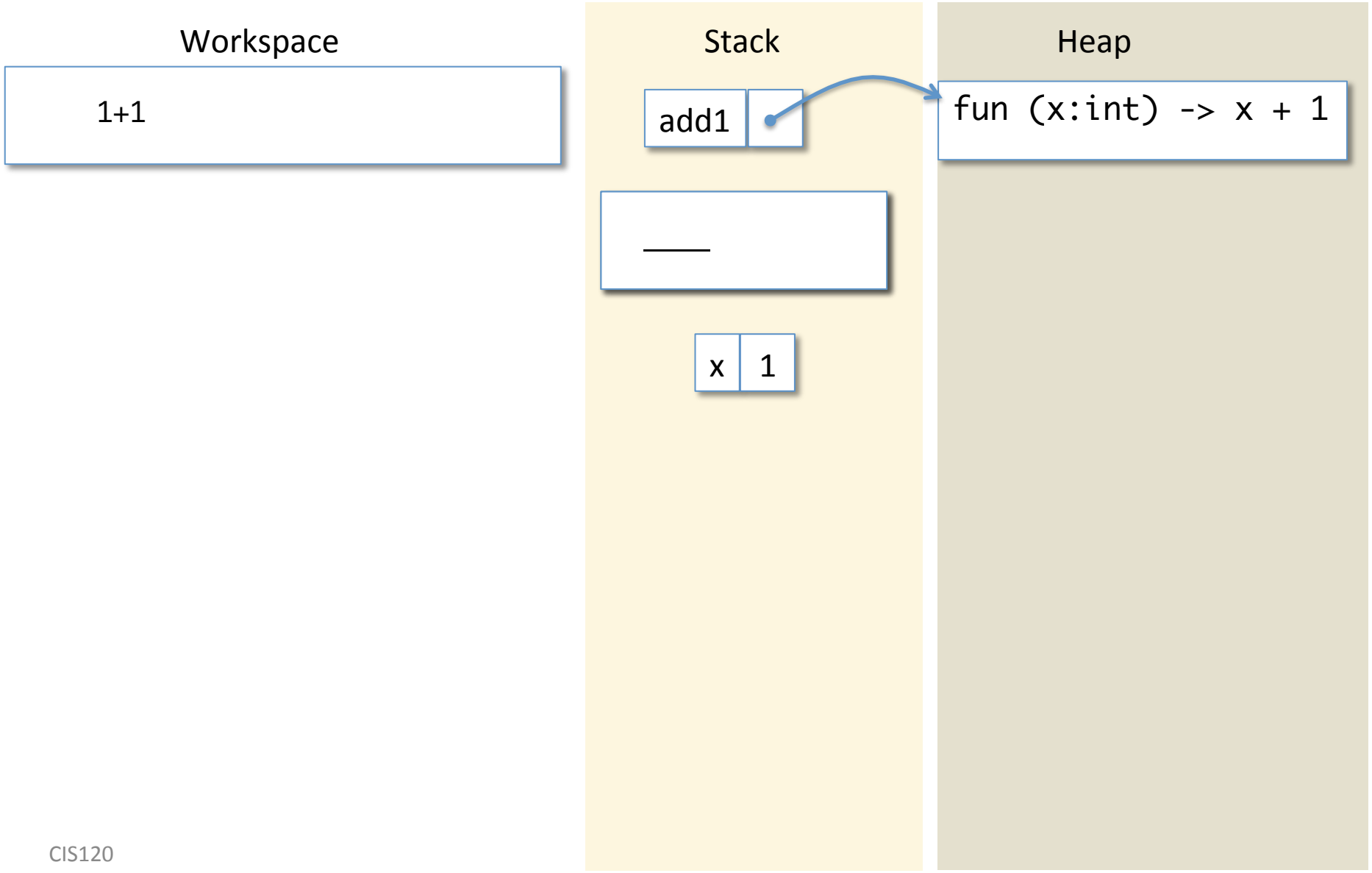
Function Simplification



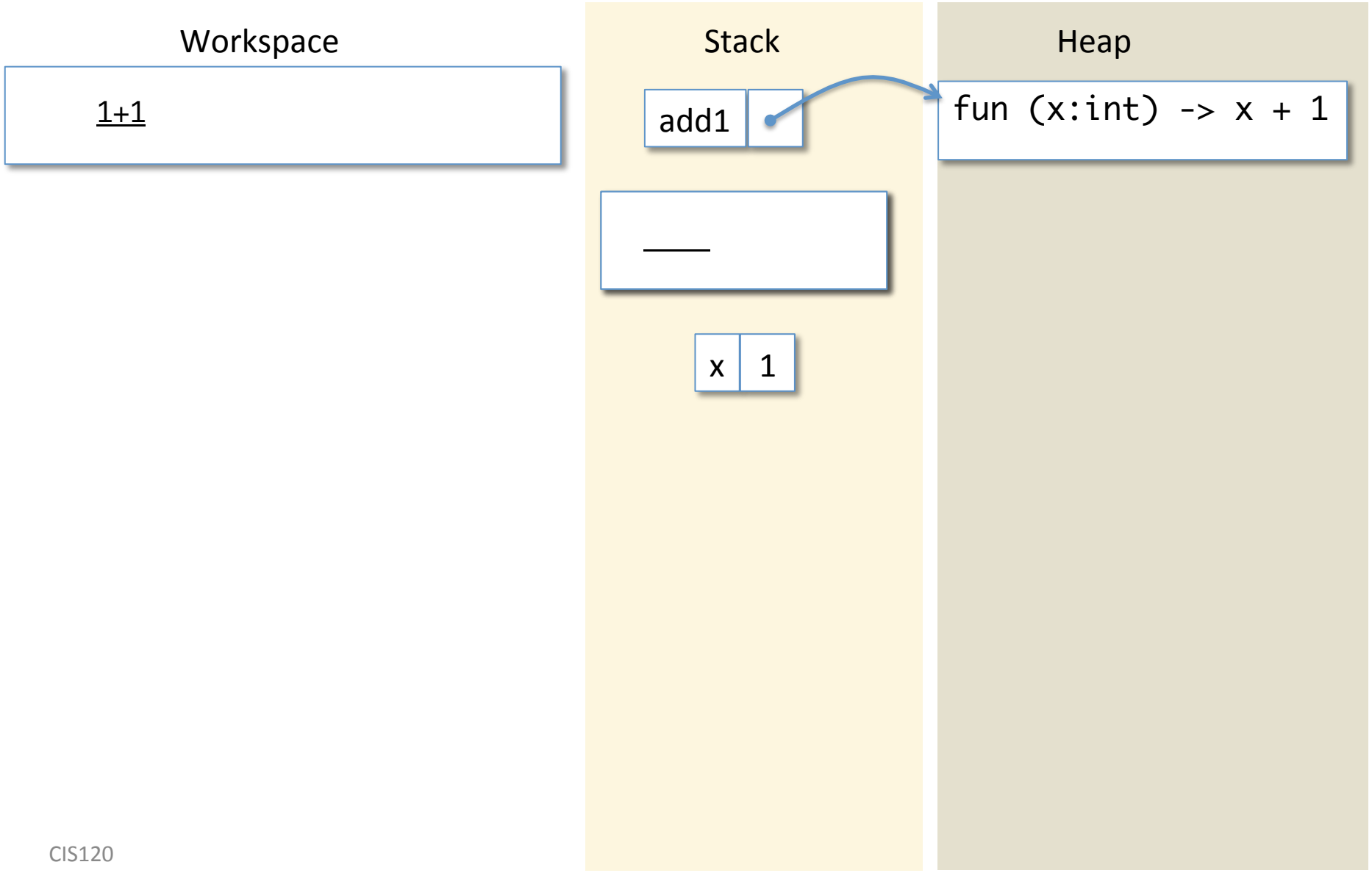
Function Simplification



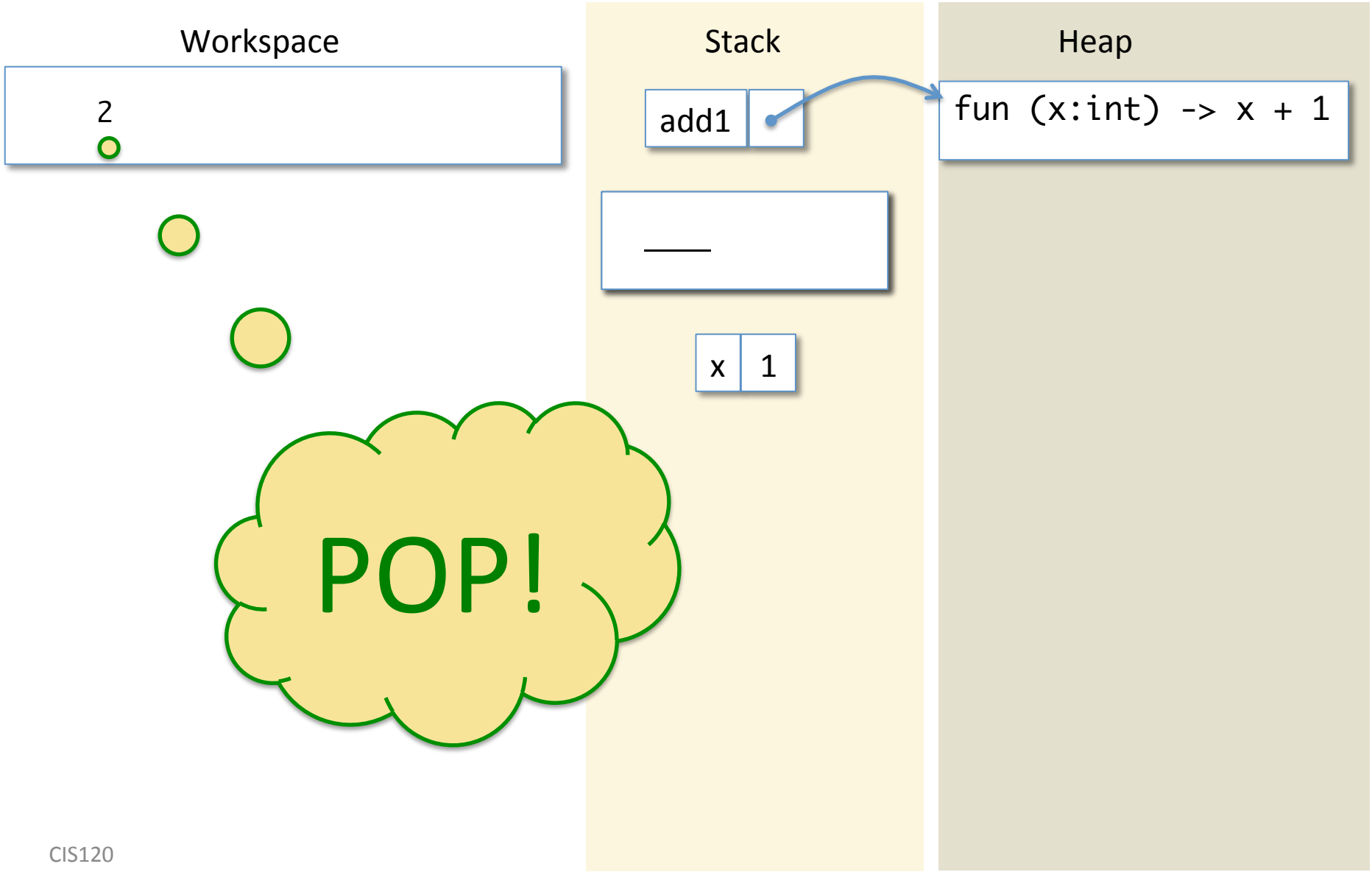
Function Simplification



Function Simplification



Function Simplification



Function Simplification

Simplifying
pattern matching & recursion

Example

```
let rec append (l1: 'a list) (l2: 'a list) : 'a list =
  begin match l1 with
  | Nil -> l2
  | Cons(h, t) -> Cons(h, append t l2)
  end in

let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil)) in

append a b
```

Simplification

Workspace

```
let rec append (l1: 'a list)
    (l2: 'a list) : 'a list =
  begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
       Cons(h, append t l2)
  end in
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil))
in
append a b
```

Stack

Heap

Function Definition

Workspace

```
let rec append (l1: 'a list)
    (l2: 'a list) : 'a list =
  begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
       Cons(h, append t l2)
  end in
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil))
in
append a b
```

Stack

Heap

Rewrite to a “fun”

Workspace

```
let append =
  fun (l1: 'a list)
    (l2: 'a list) ->
  begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
       Cons(h, append t l2)
  end in
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil))
in
append a b
```

Stack

Heap

Function Expression

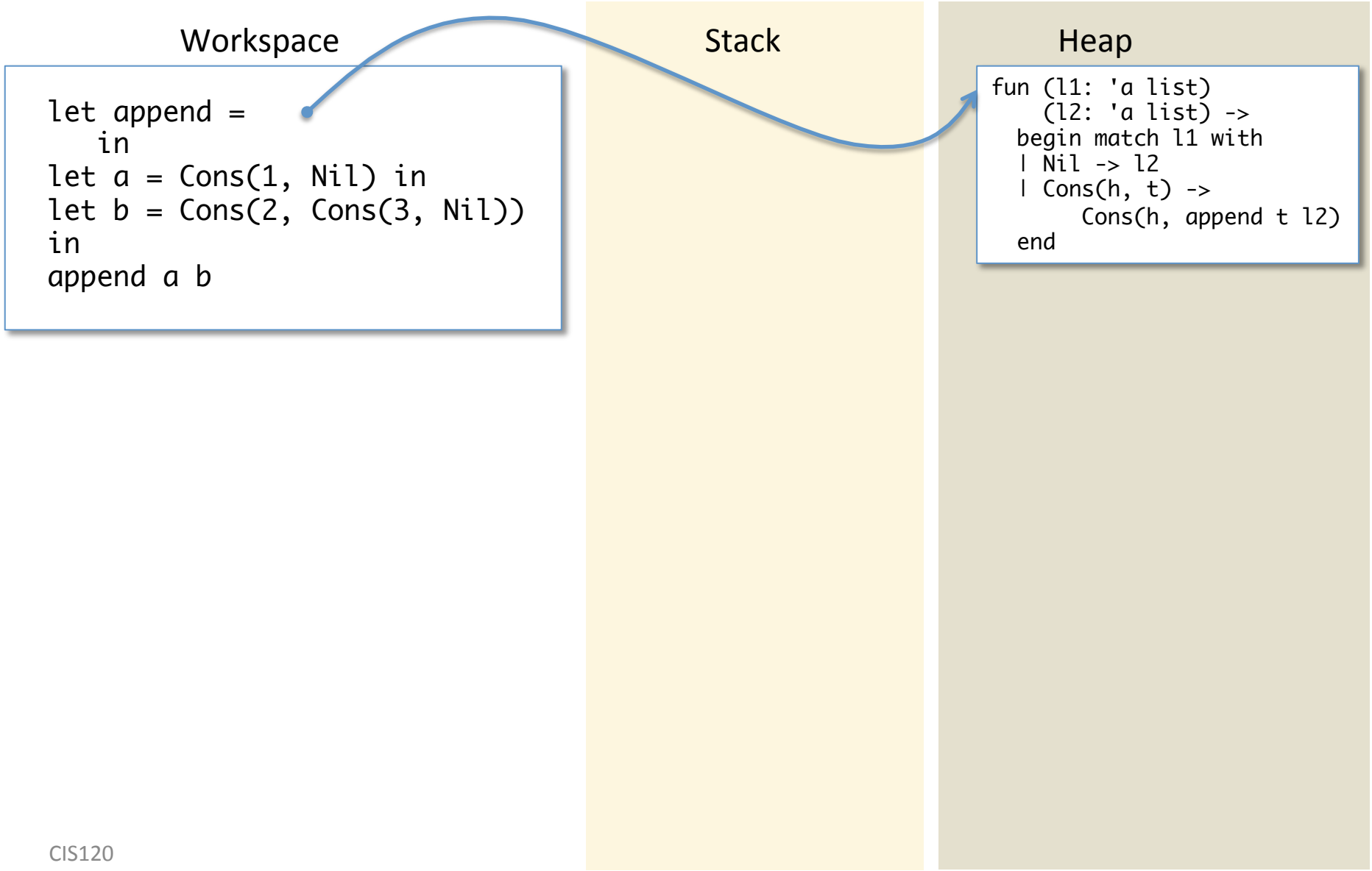
Workspace

```
let append =
  fun (l1: 'a list)
    (l2: 'a list) ->
  begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
       Cons(h, append t l2)
  end in
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil))
in
append a b
```

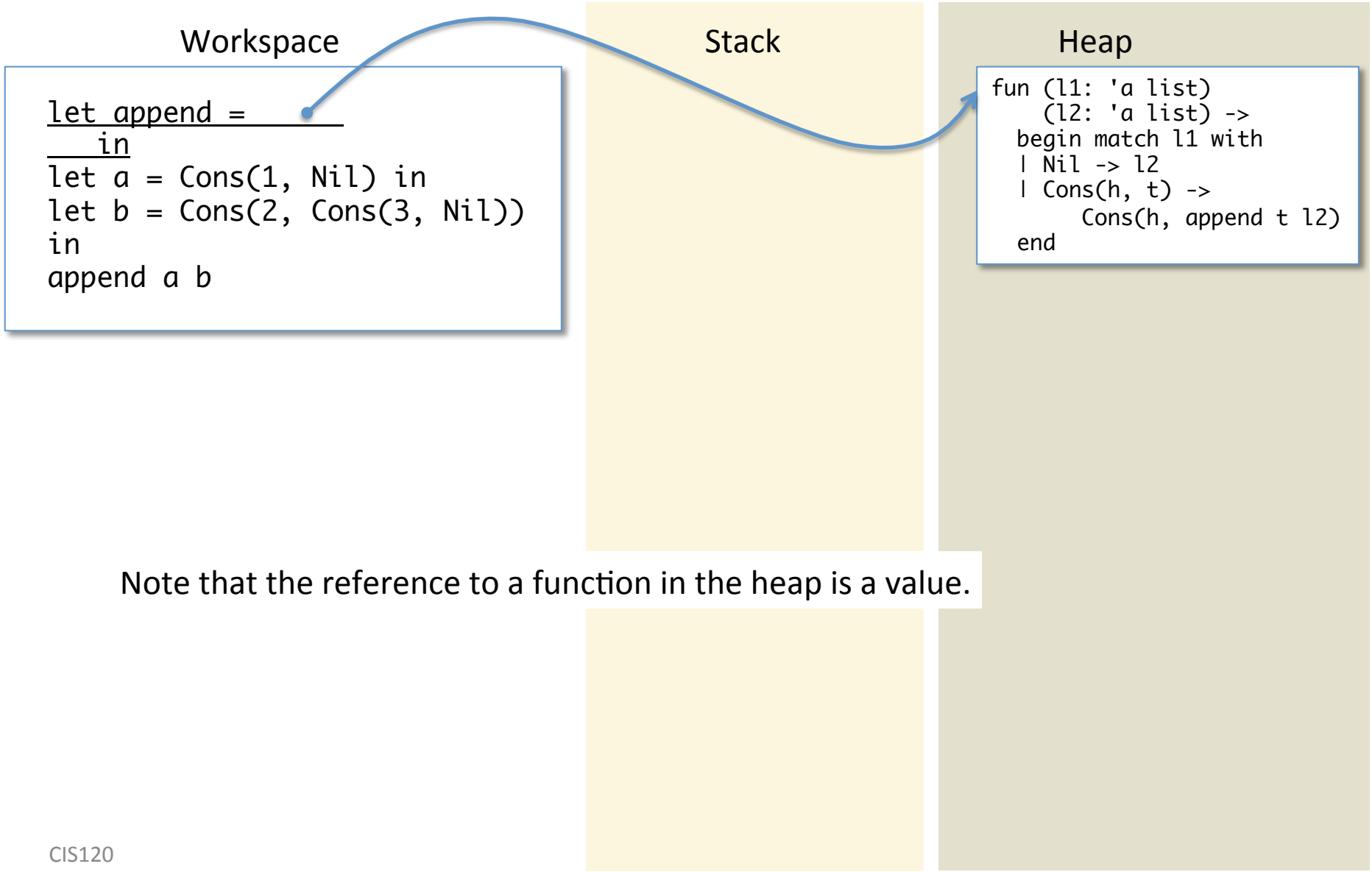
Stack

Heap

Copy to the Heap, Replace w/Reference



Let Expression



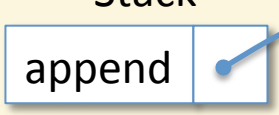
Note that the reference to a function in the heap is a value.

Create a Stack Binding

Workspace

```
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil))
in
append a b
```

Stack



Heap

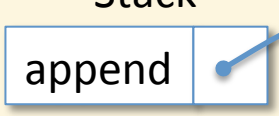
```
fun (l1: 'a list)
    (l2: 'a list) ->
  begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
       Cons(h, append t l2)
  end
```

Allocate a Nil cell

Workspace

```
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil))
in
append a b
```

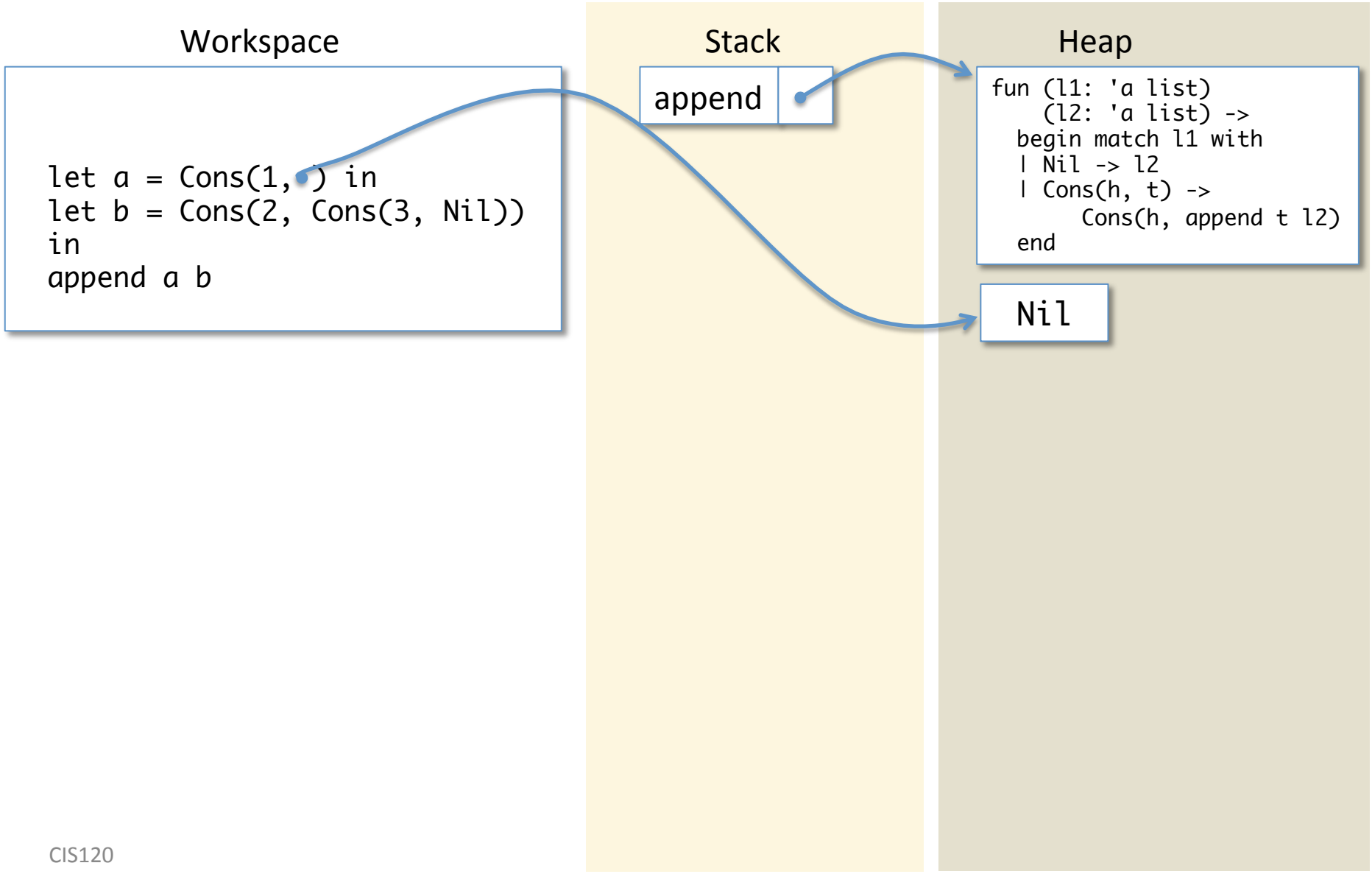
Stack



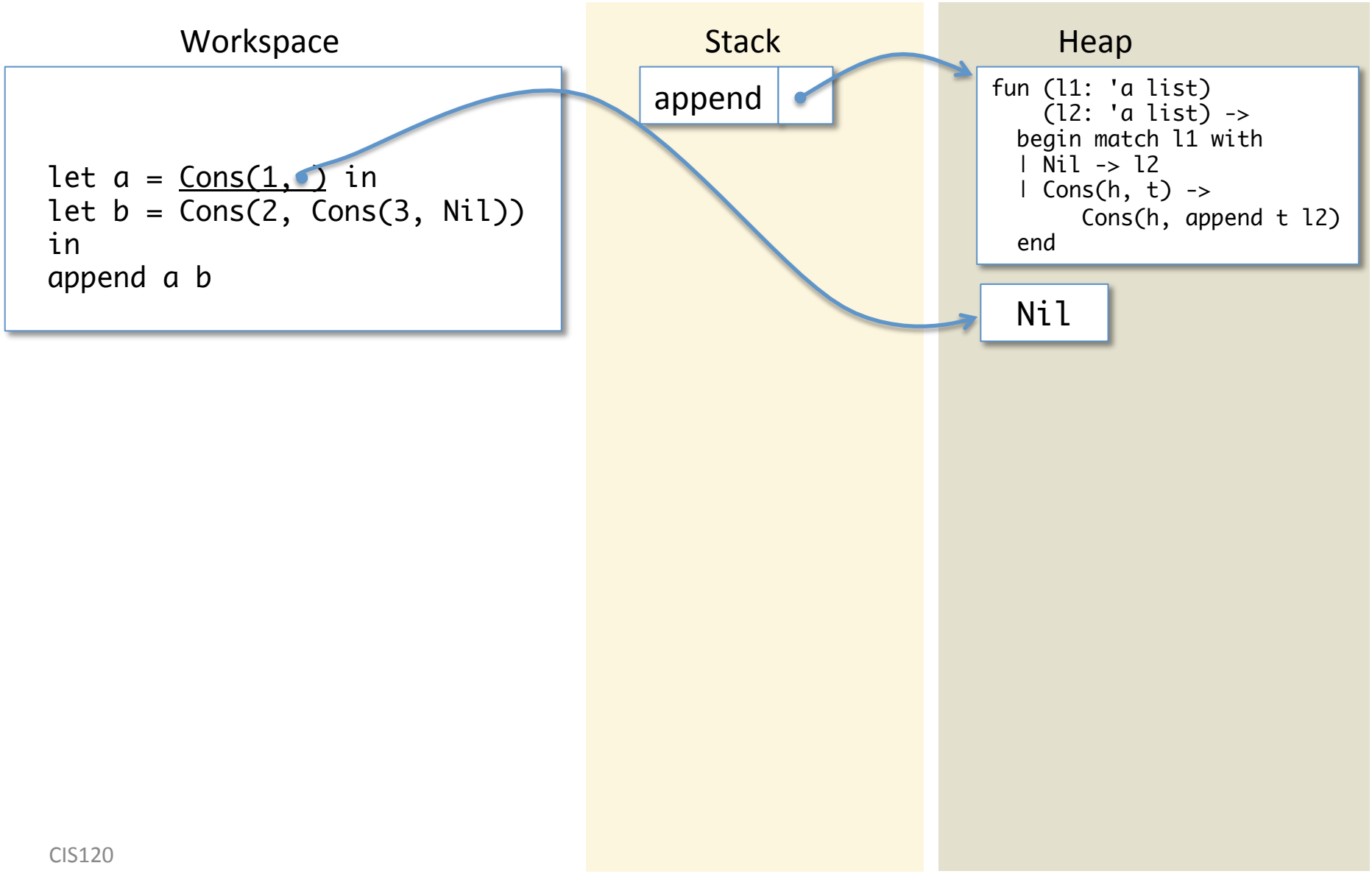
Heap

```
fun (l1: 'a list)
    (l2: 'a list) ->
  begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
       Cons(h, append t l2)
  end
```

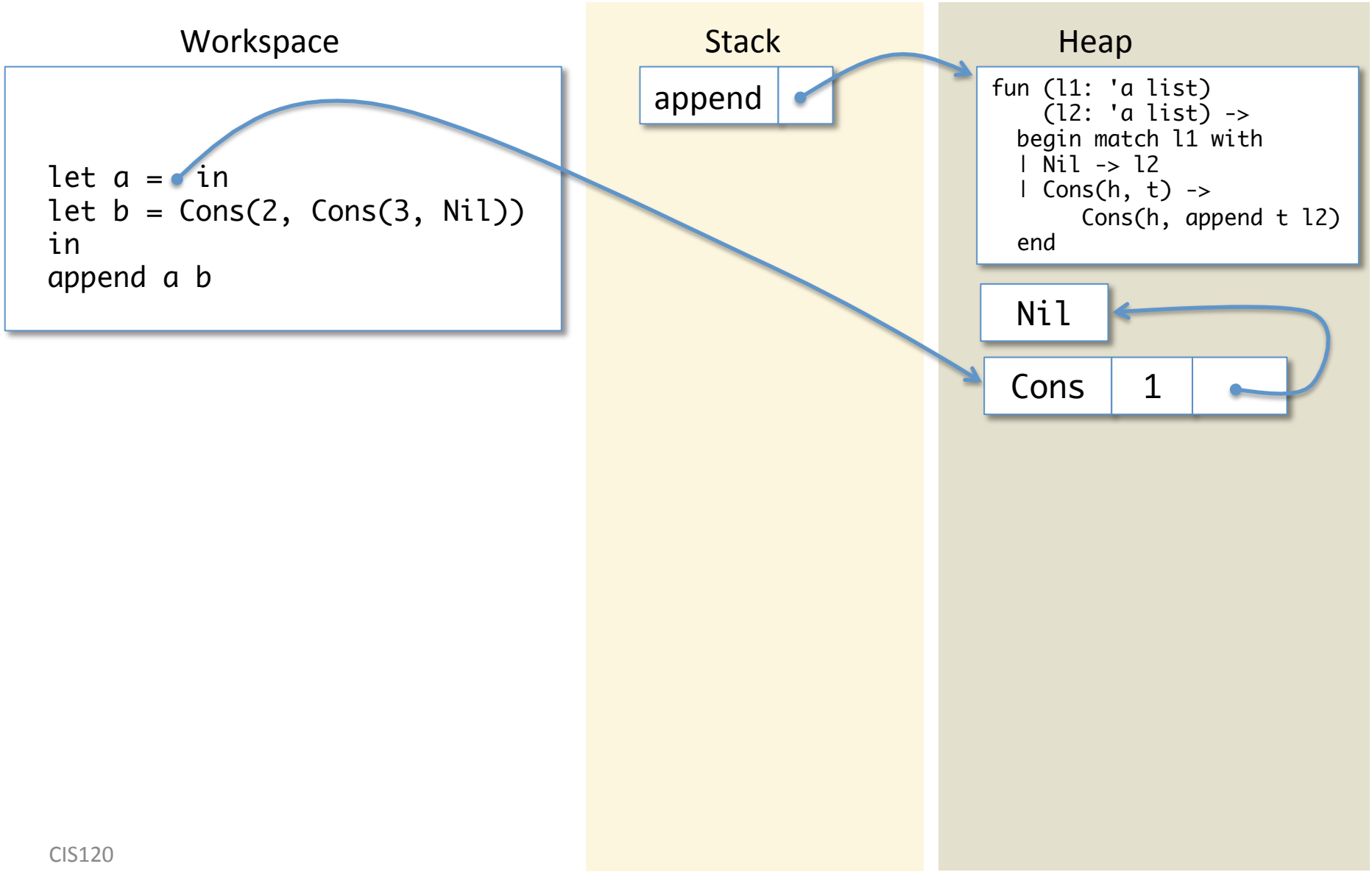
Allocate a Nil cell



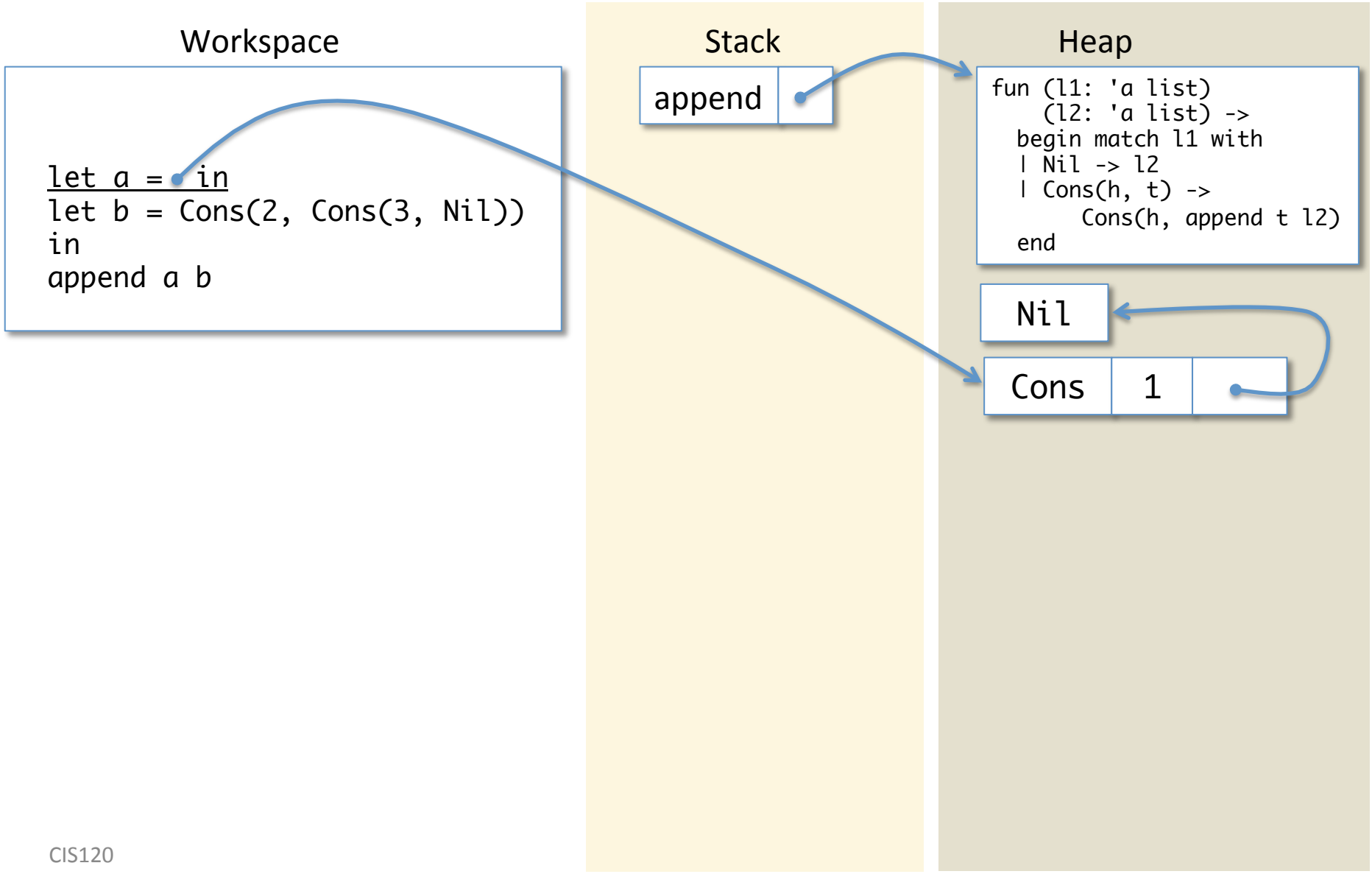
Allocate a Cons cell



Allocate a Cons cell



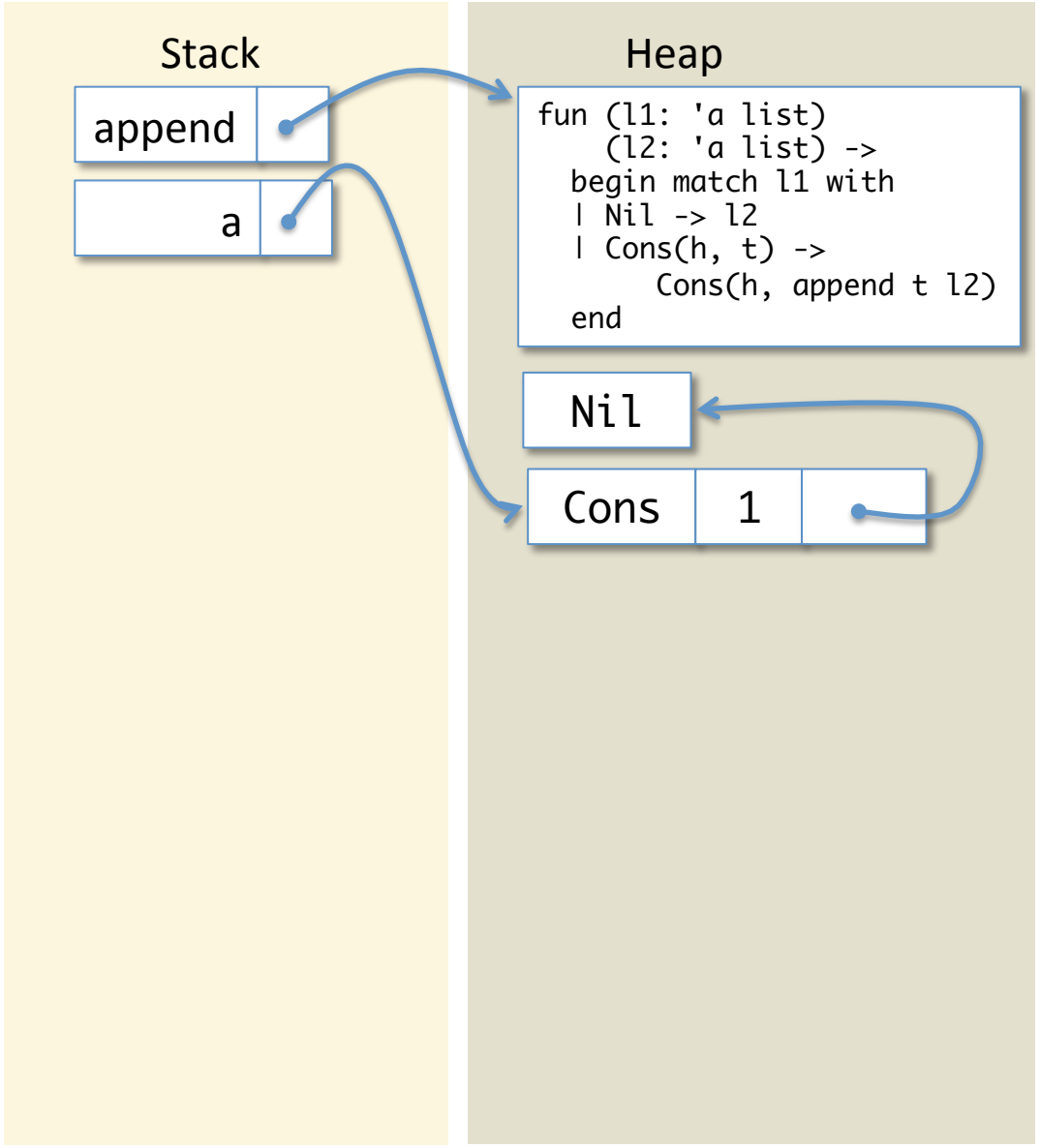
Let Expression



Create a Stack Binding

Workspace

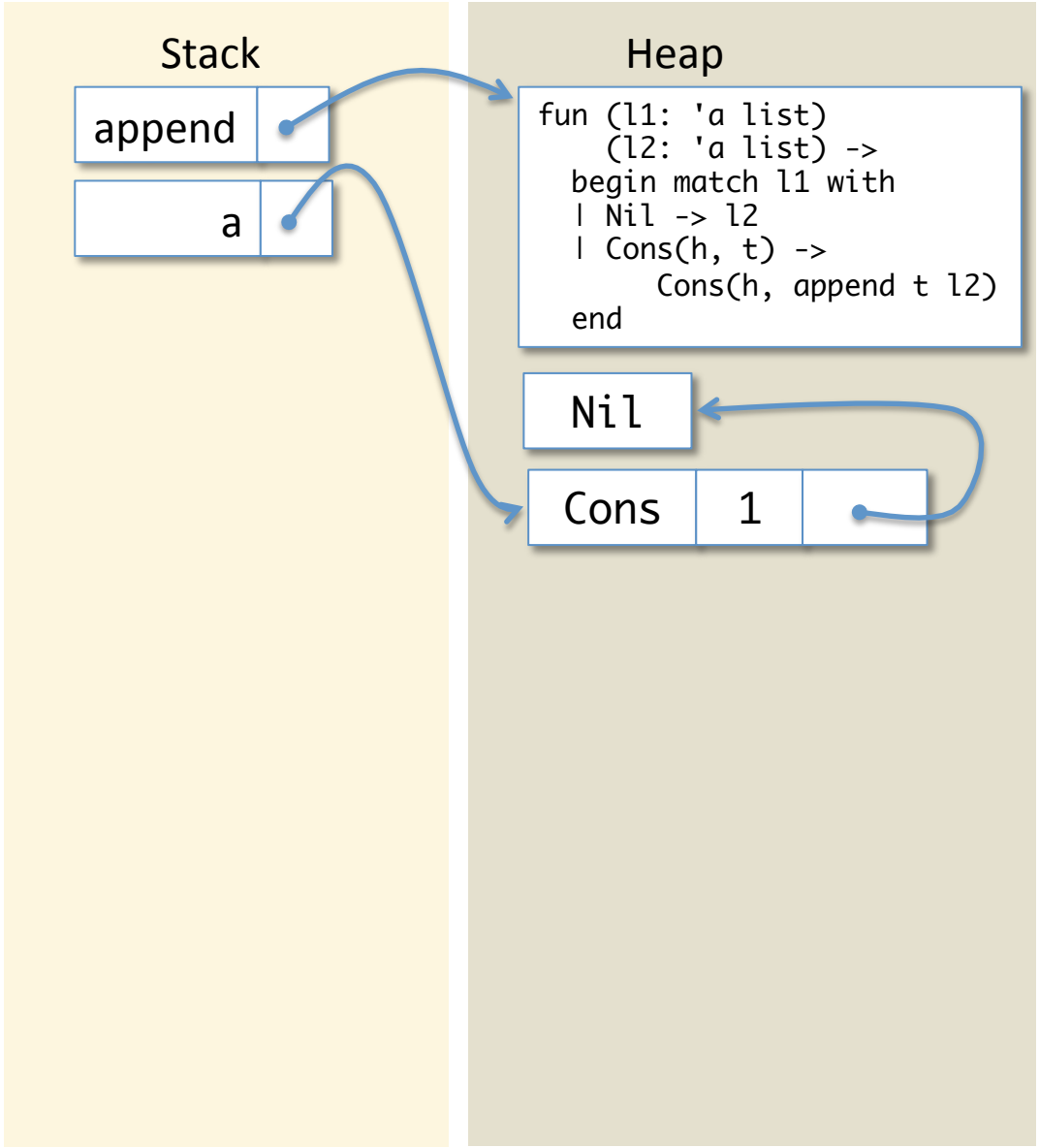
```
let b = Cons(2, Cons(3, Nil))
in
append a b
```



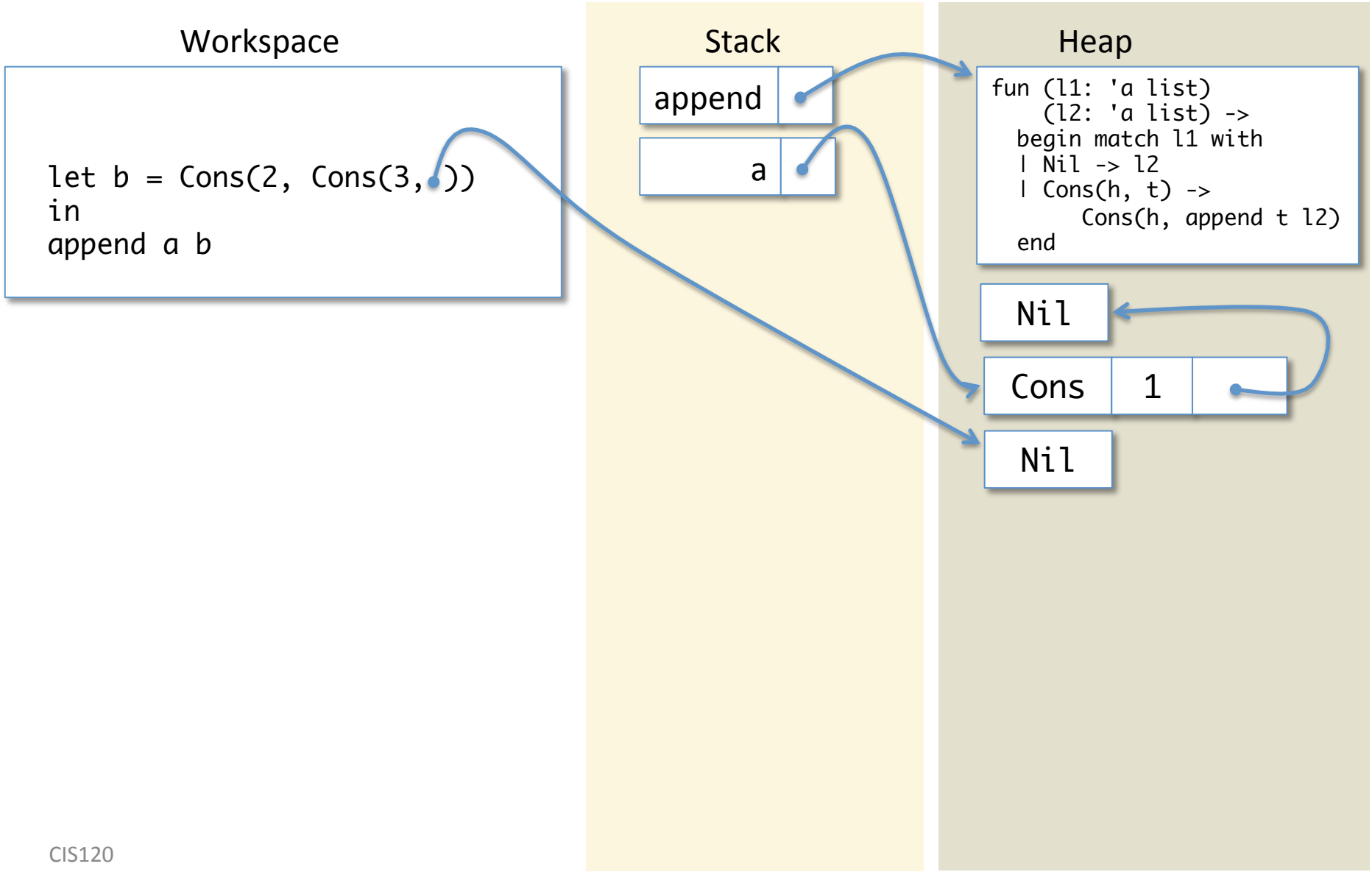
Allocate a Nil cell

Workspace

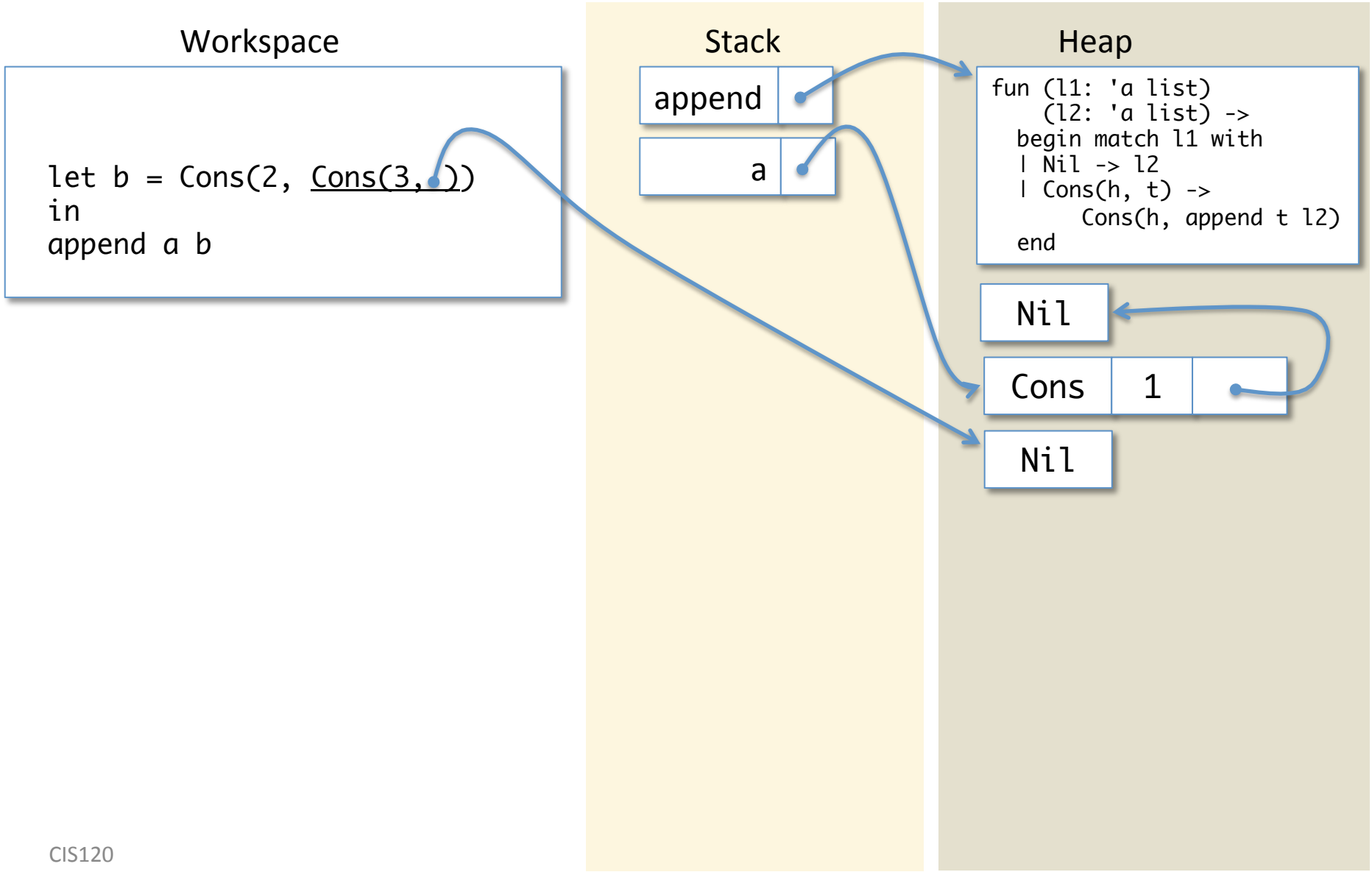
```
let b = Cons(2, Cons(3, Nil))
in
append a b
```



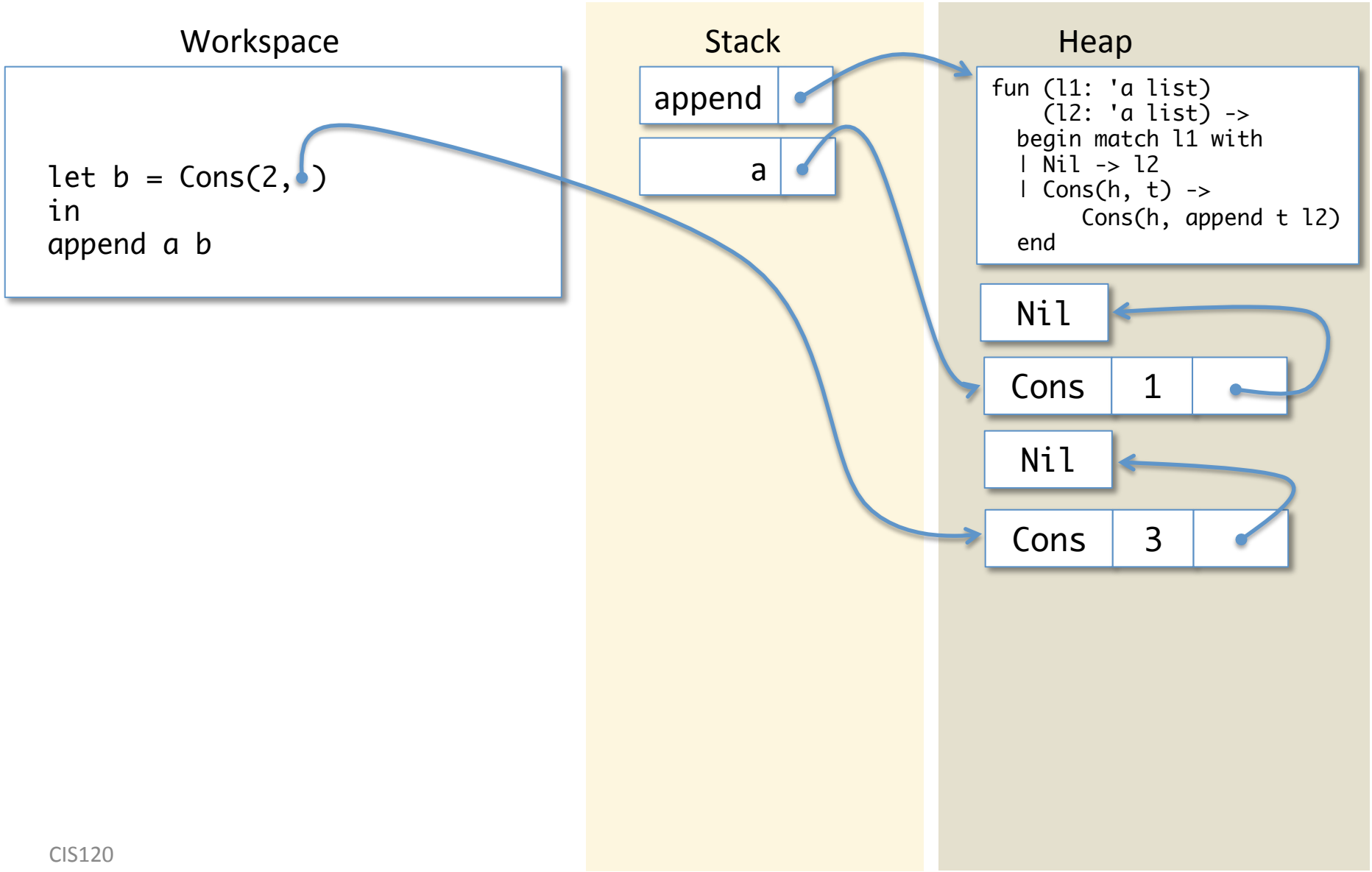
Allocate a Nil cell



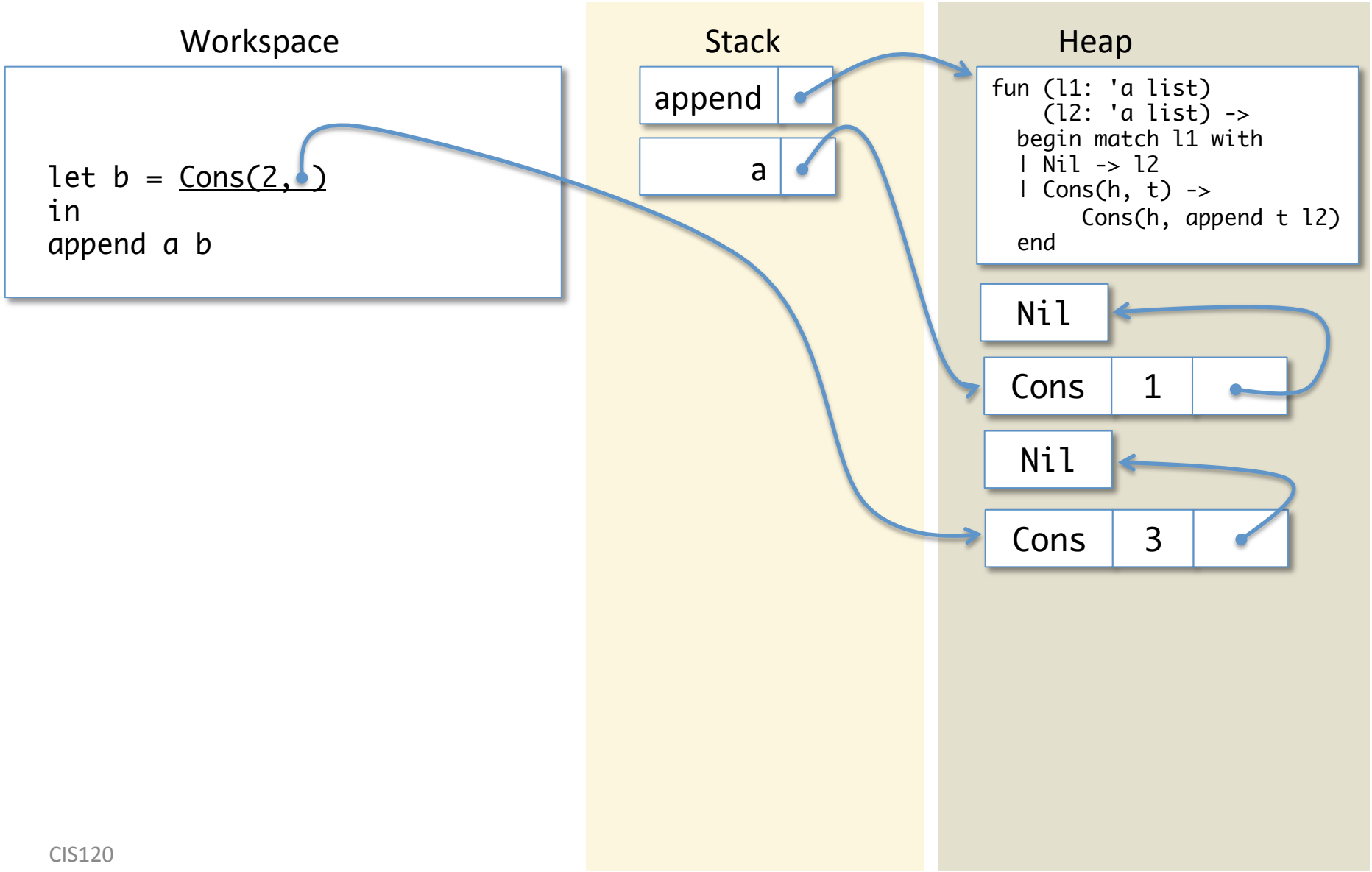
Allocate a Cons cell



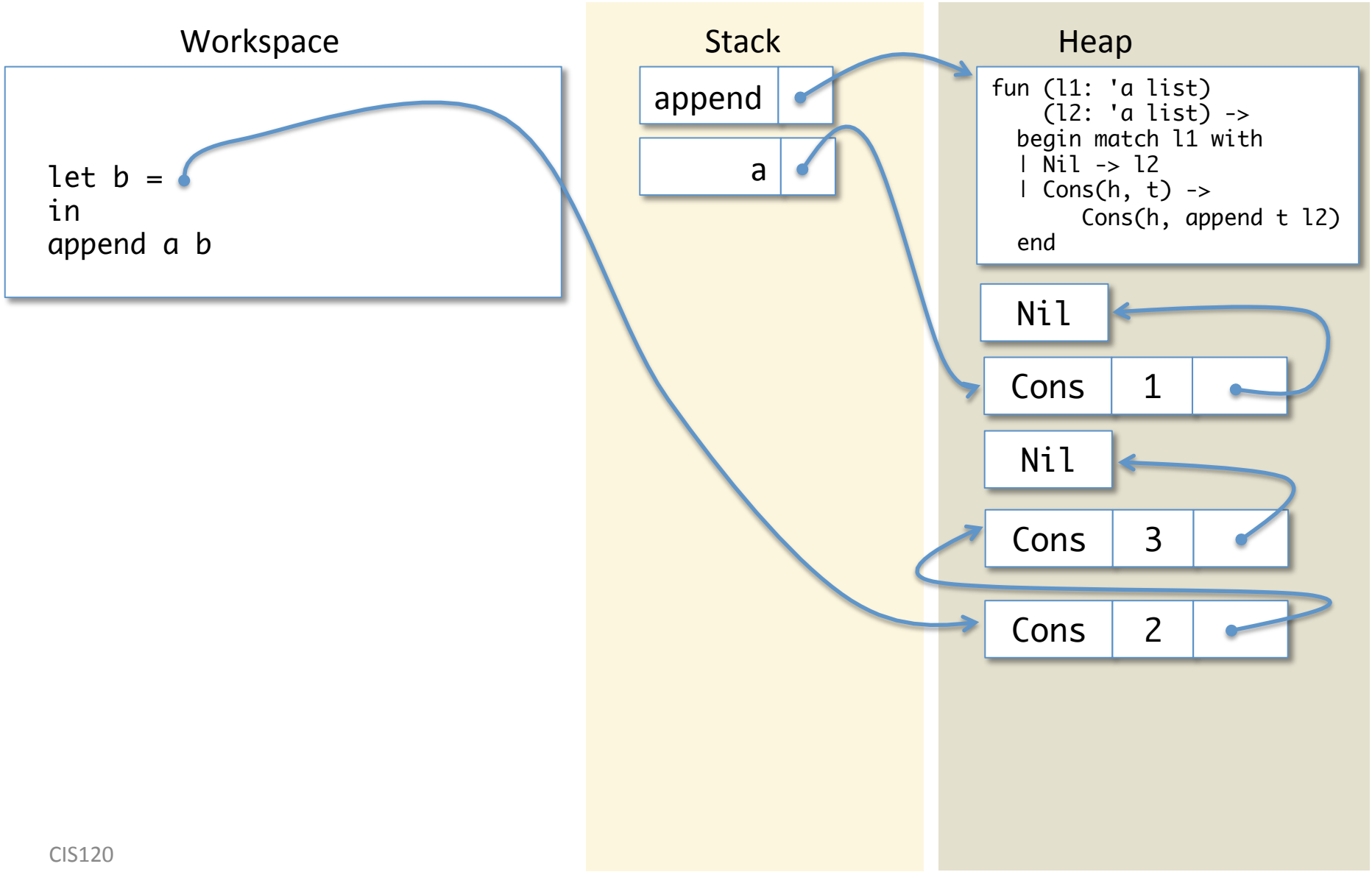
Allocate a Cons cell



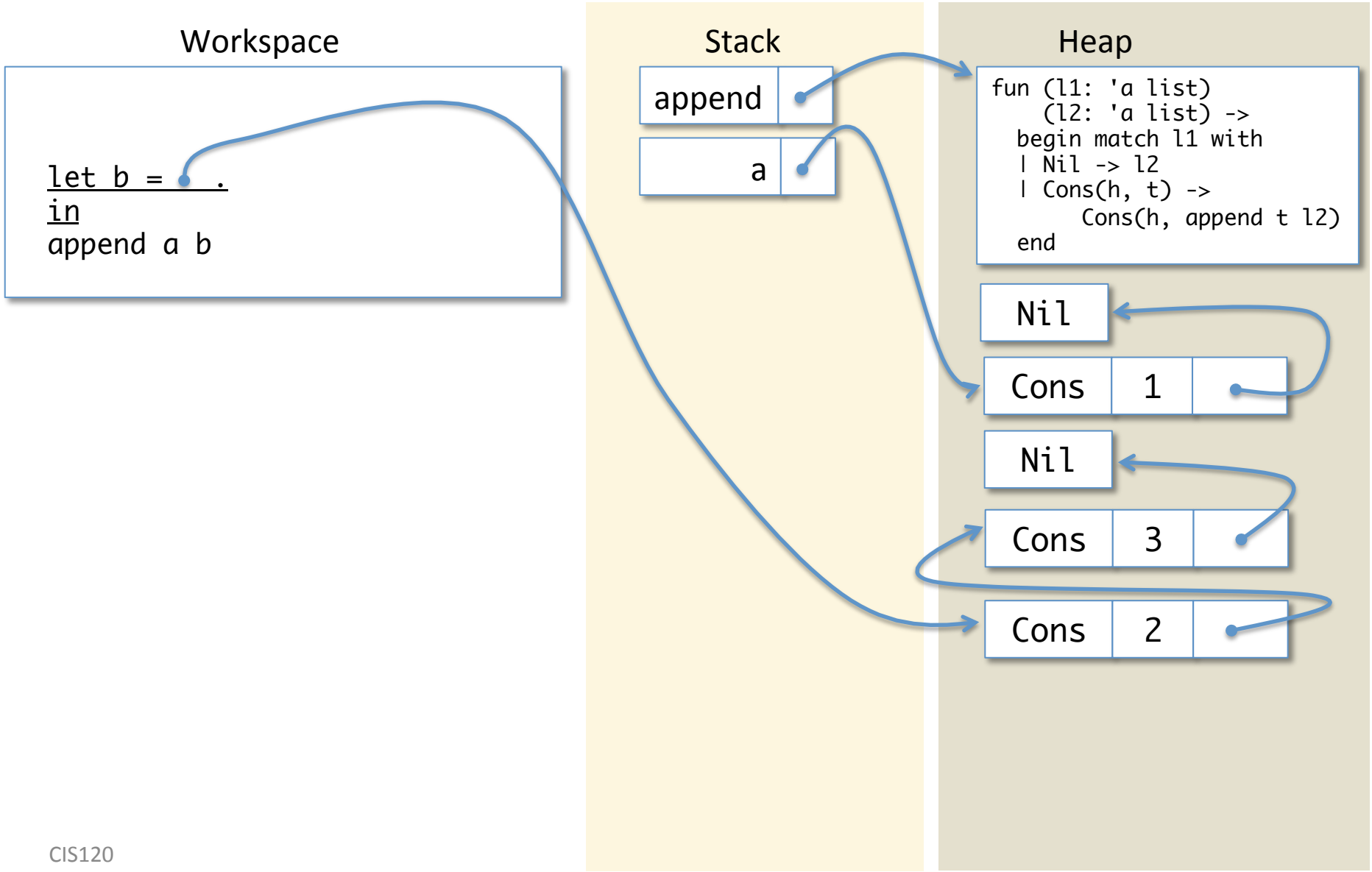
Allocate a Cons cell



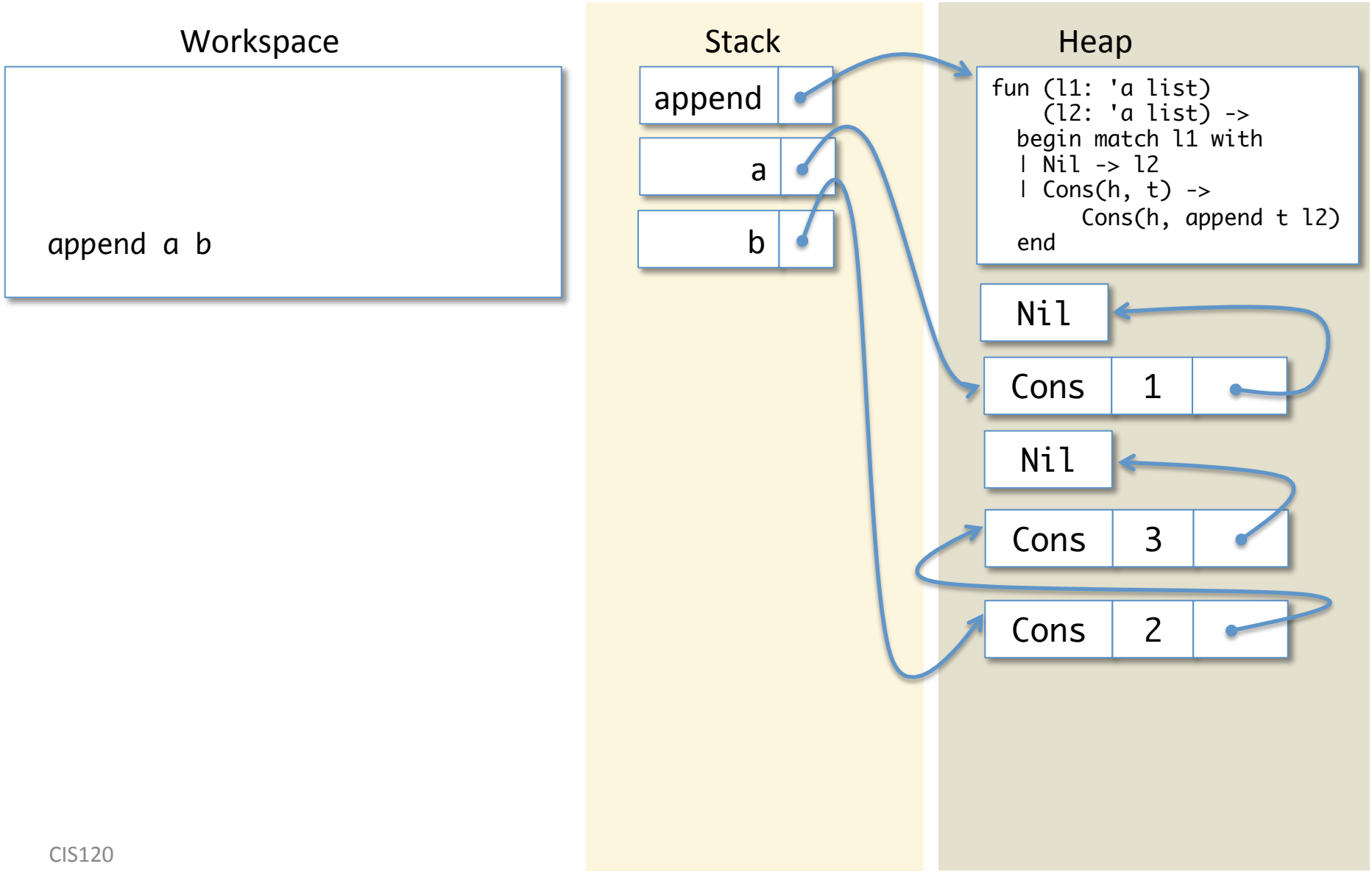
Allocate a Cons cell



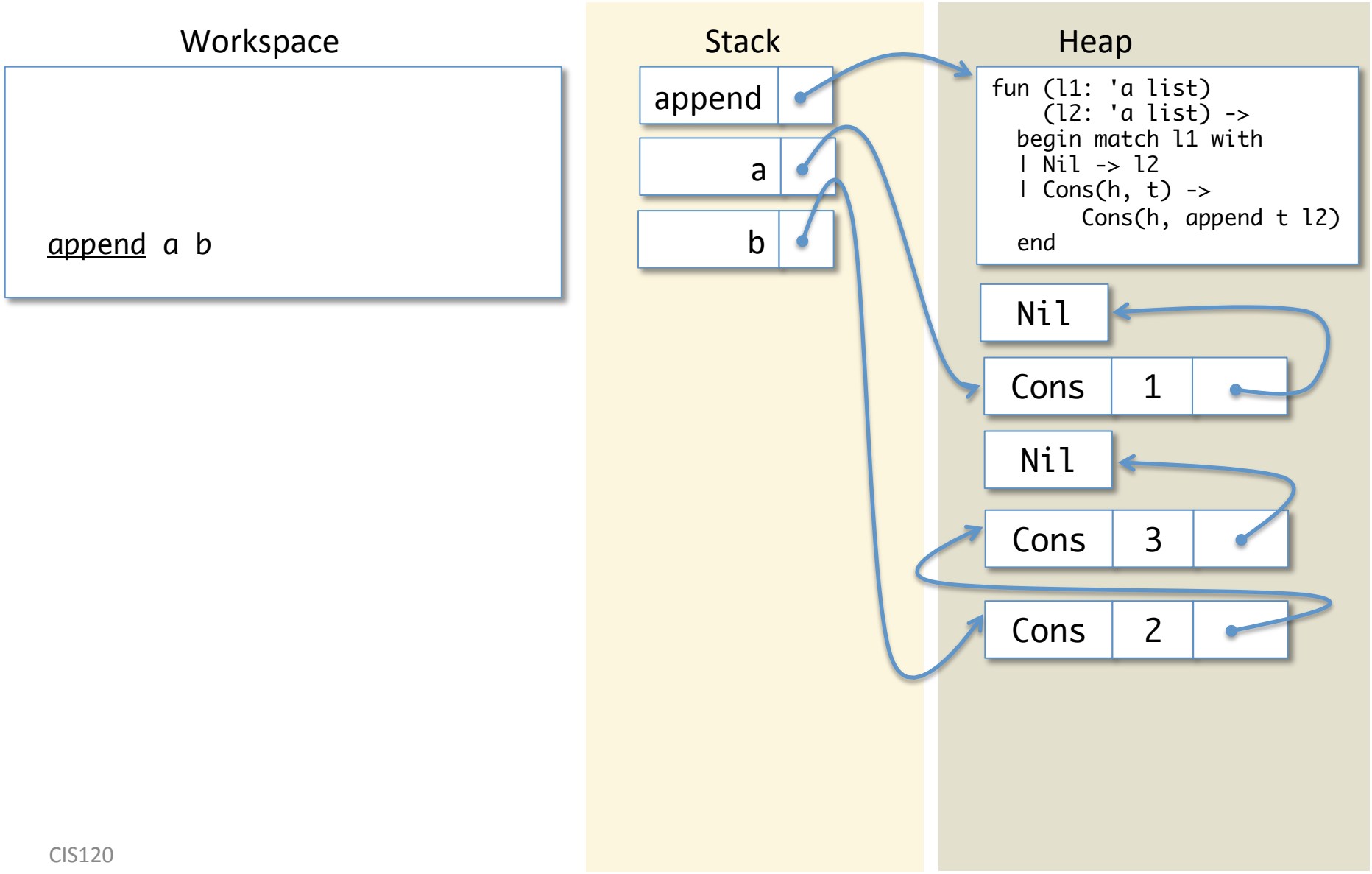
Let Expression



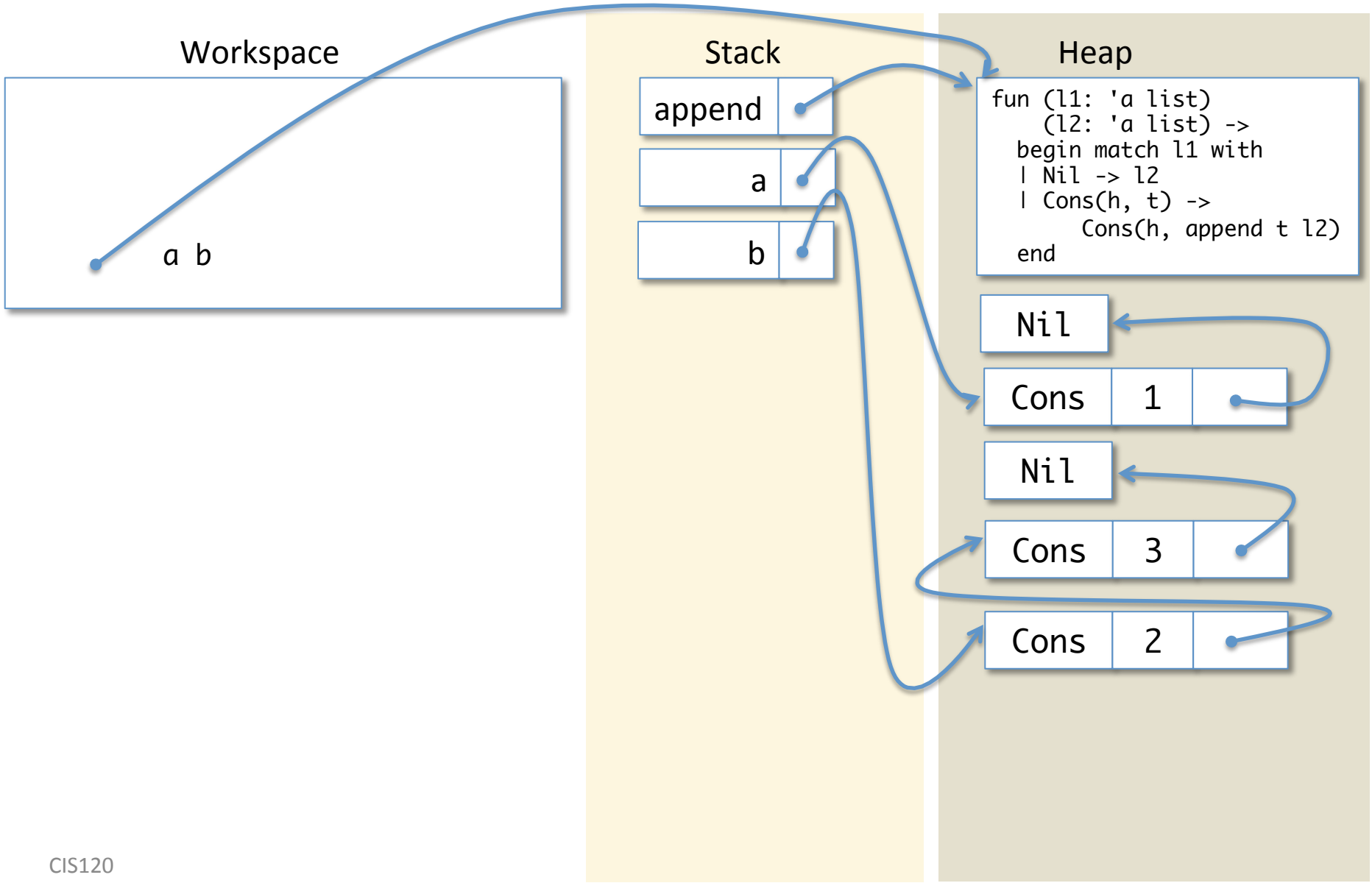
Create a Stack Binding



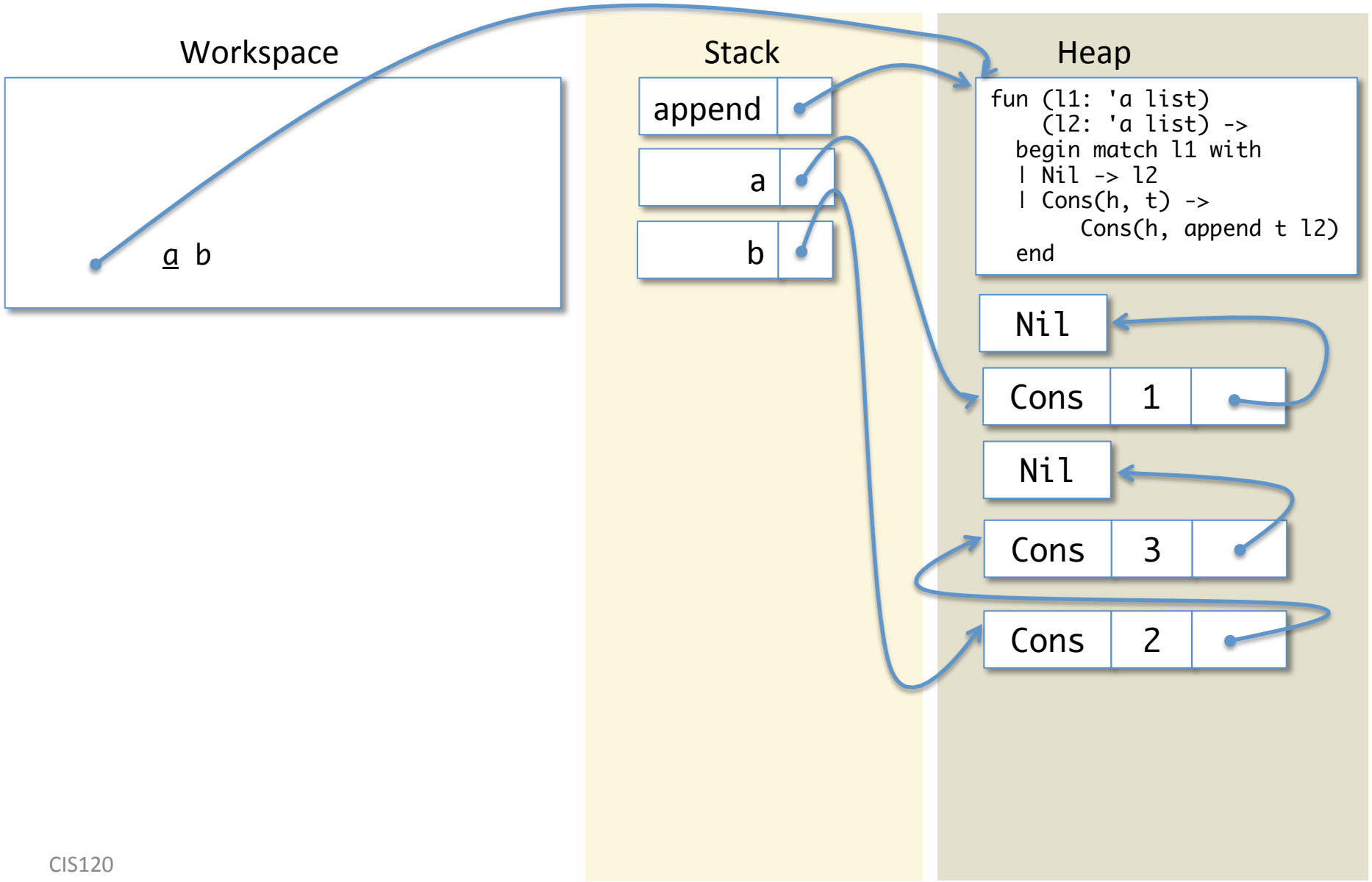
Lookup ‘append’



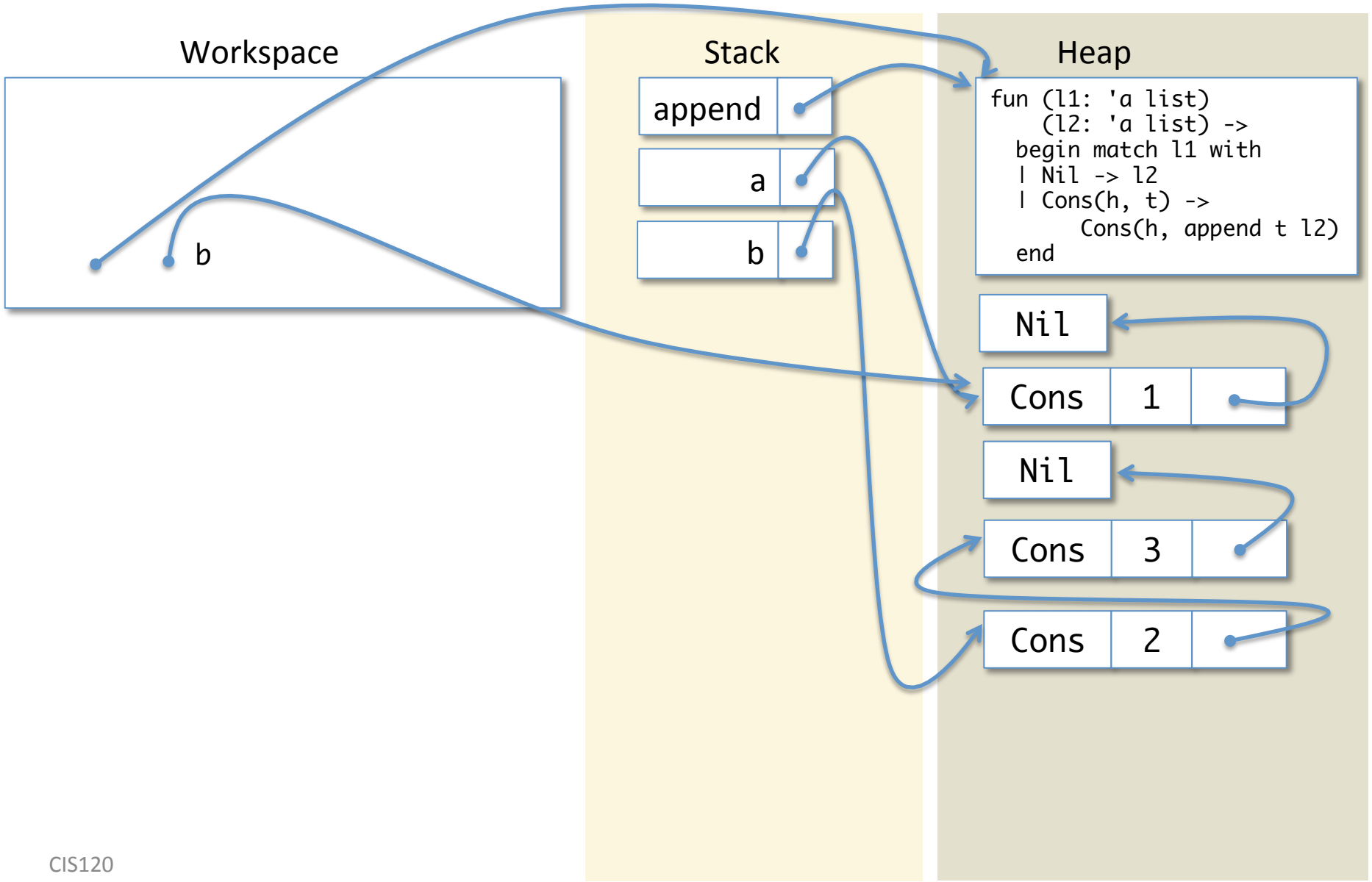
Lookup ‘append’



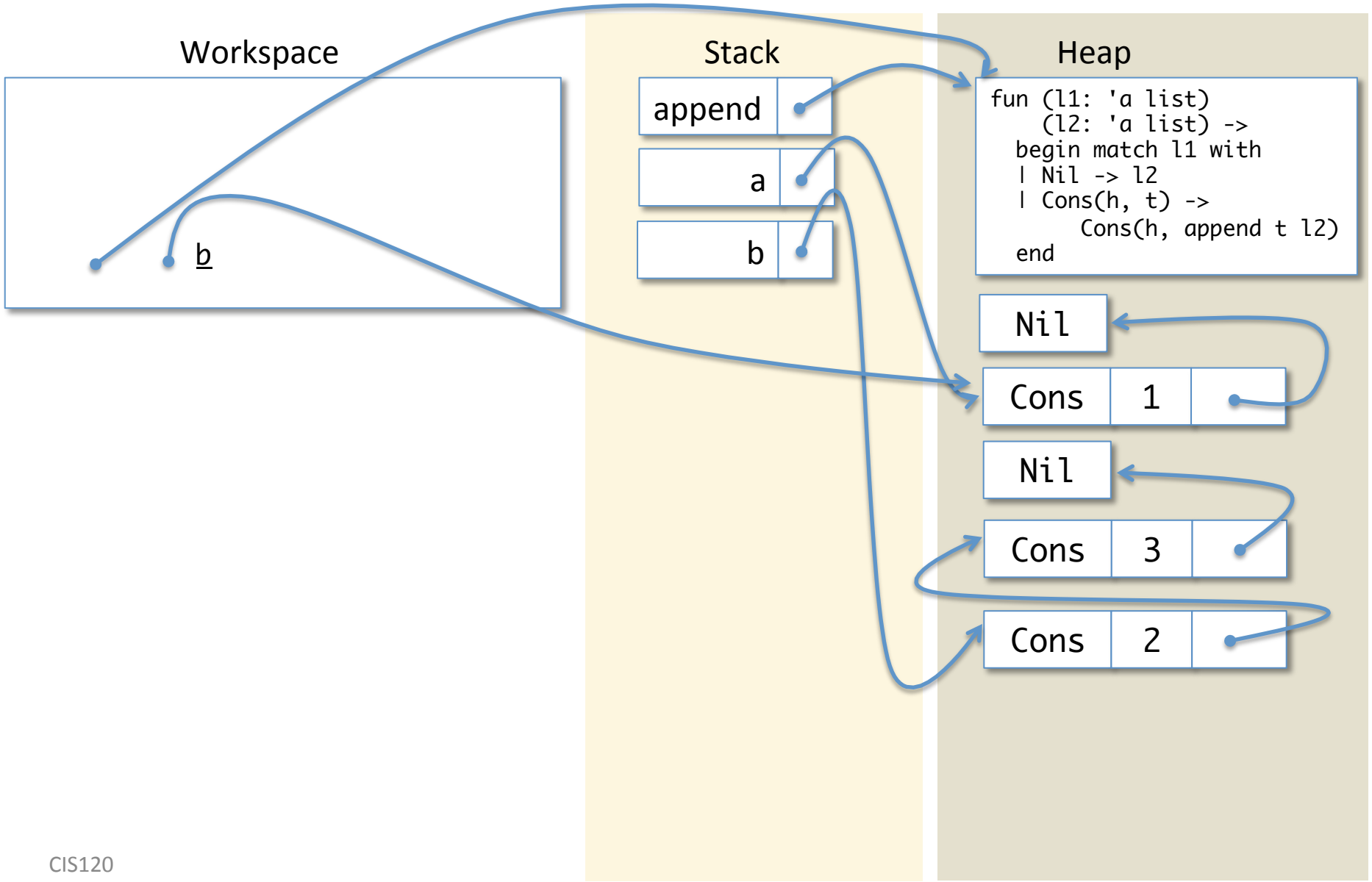
Lookup ‘a’


Workspace
a b
Stack
append
a
b
Heap
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

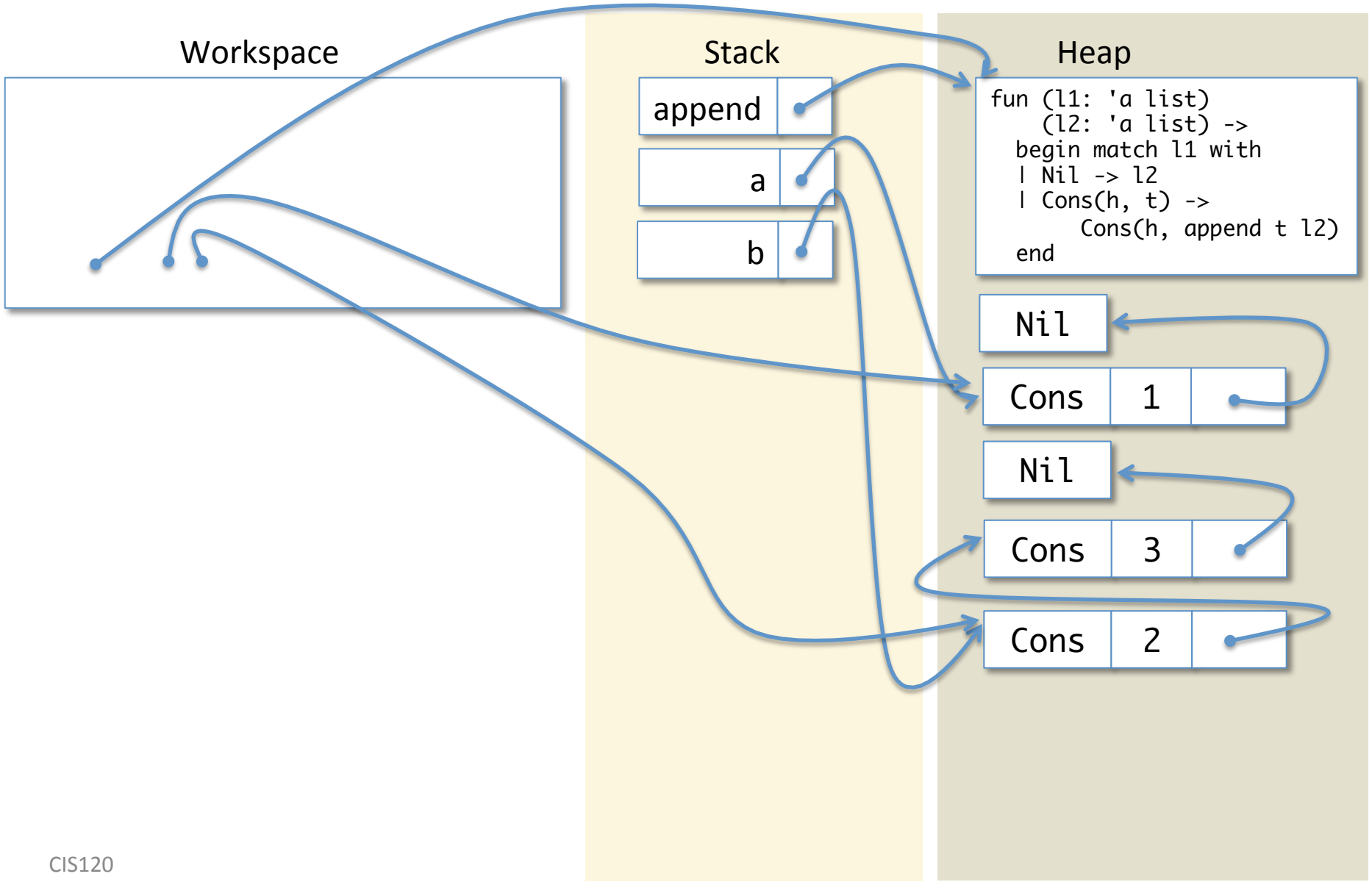
Lookup ‘a’


Workspace
b
Stack
append
a
b
Heap
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

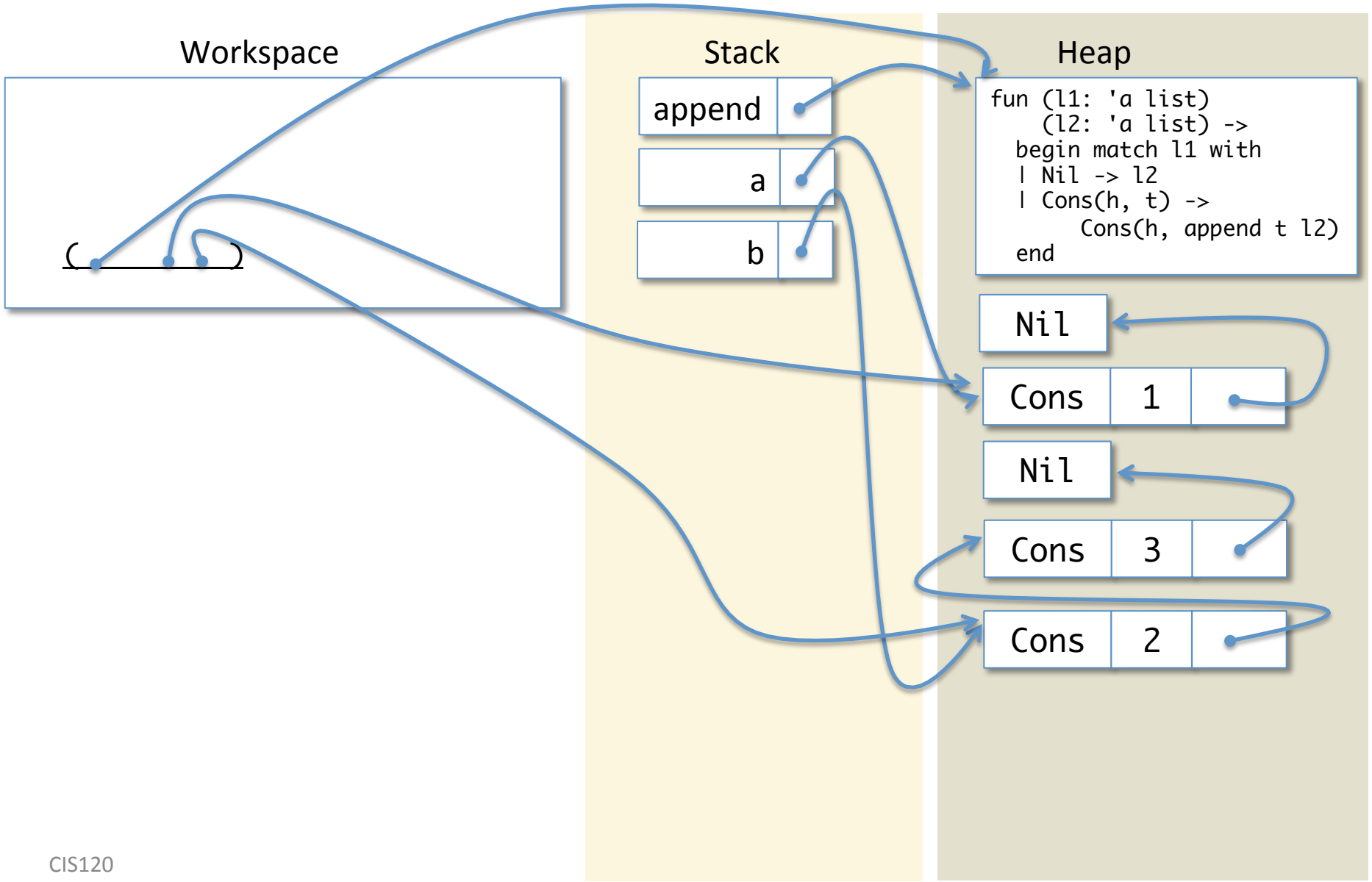
Lookup ‘b’


Workspace
b
Stack
append
a
b
Heap
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

Lookup ‘b’


Workspace
Stack
Heap
append
a
b
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

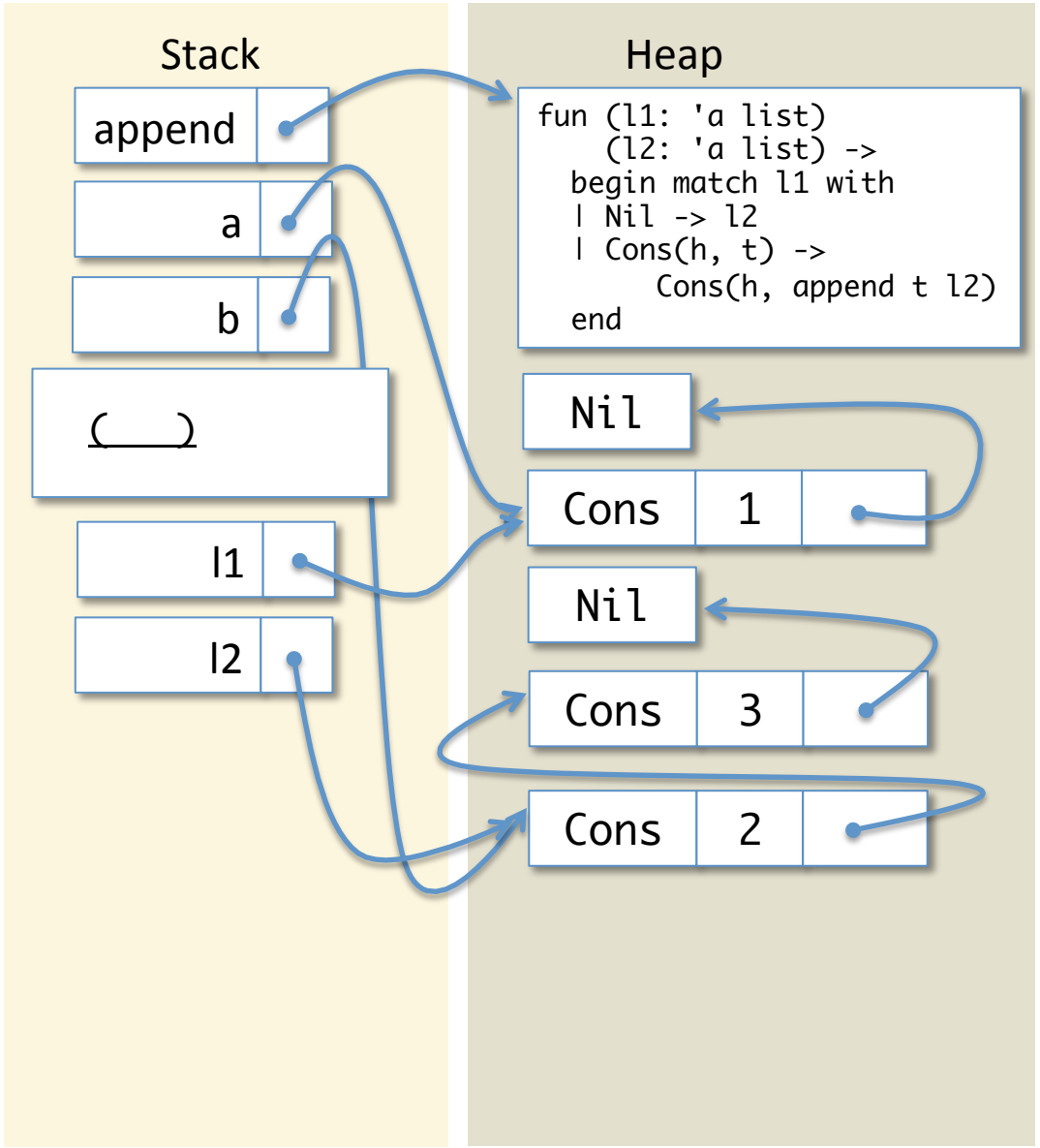
Do the Function call



Save Workspace; push l1, l2

Workspace

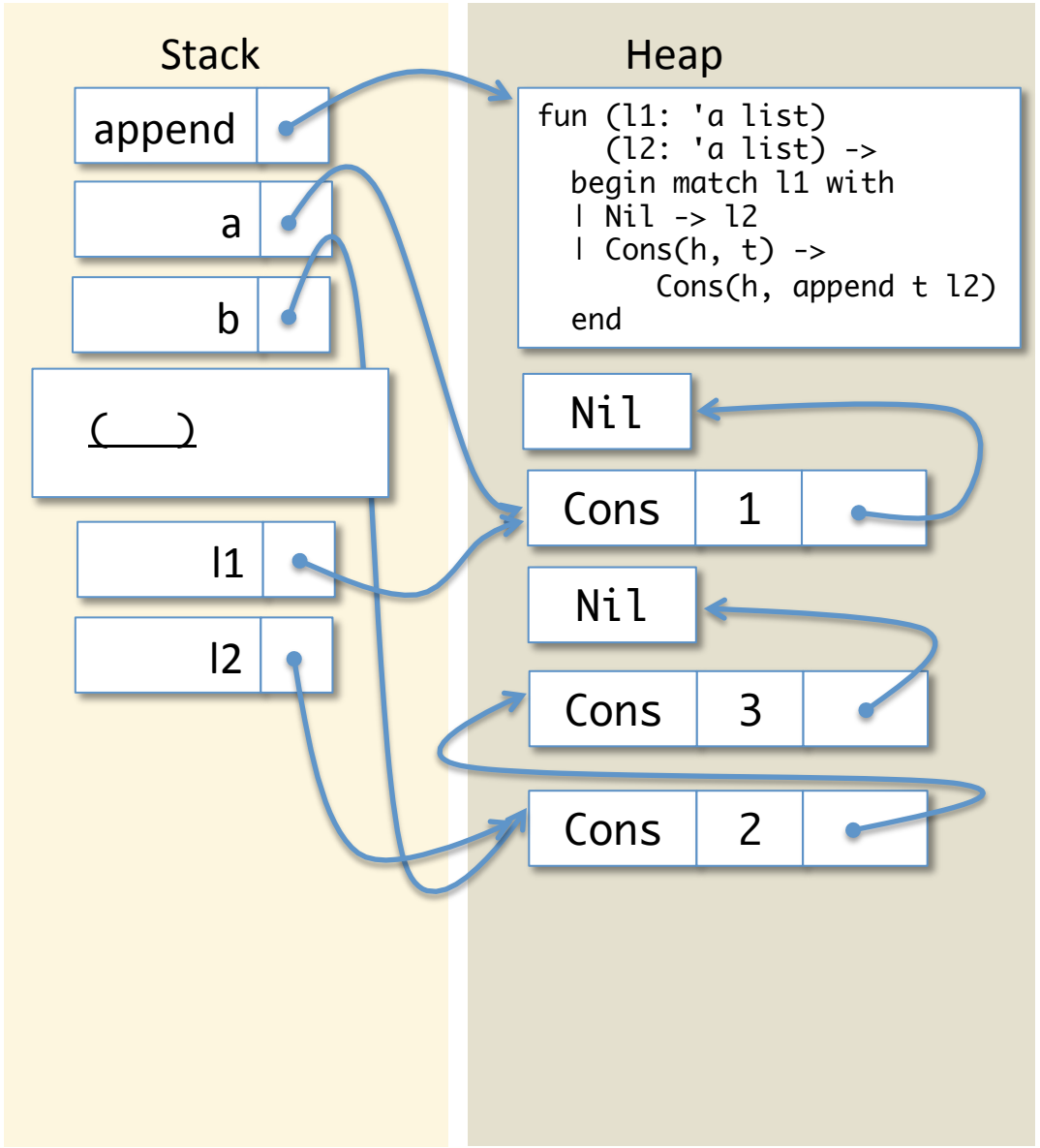
```
begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
       Cons(h, append t l2)
  end
```



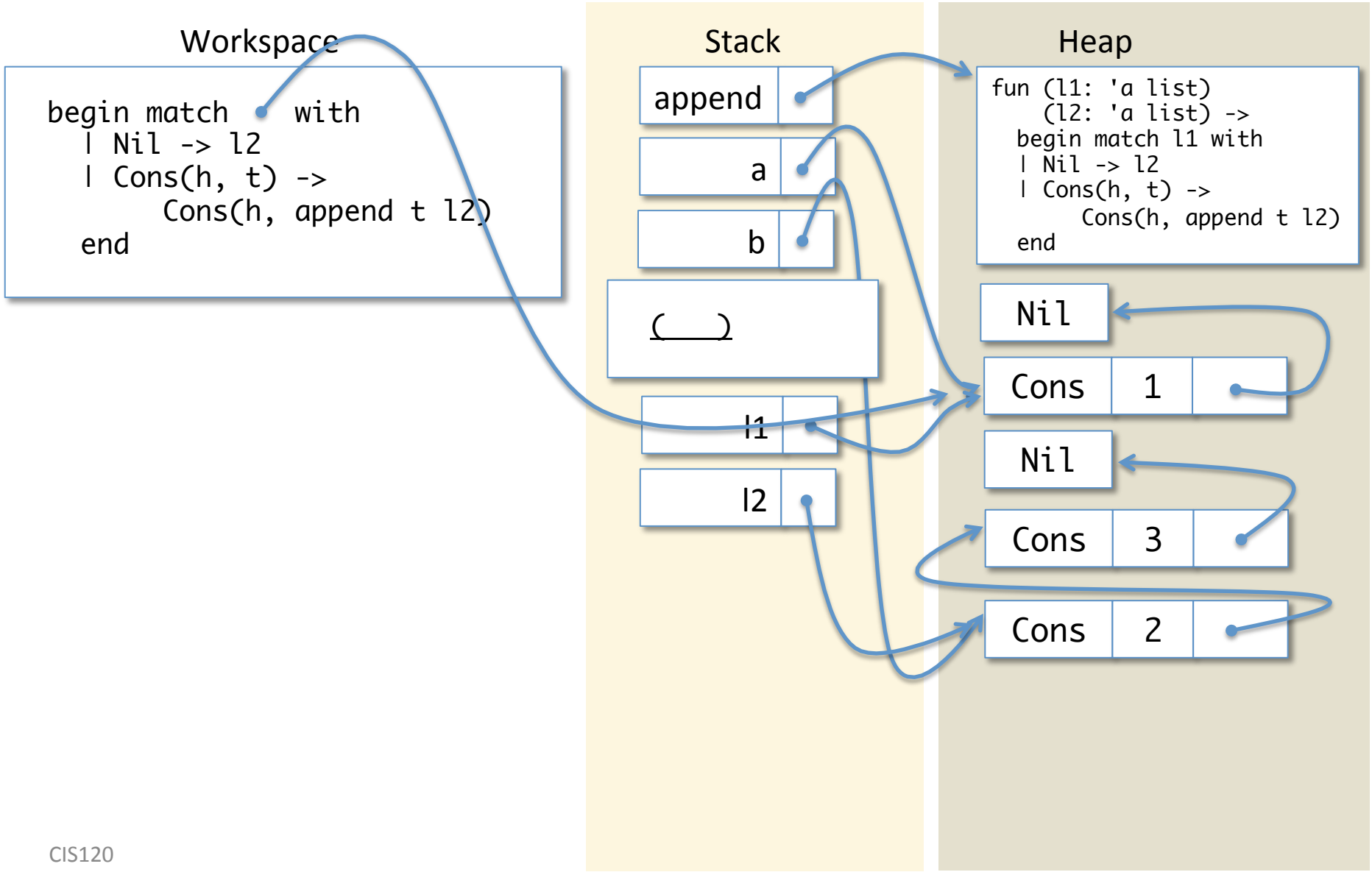
Lookup l1

Workspace

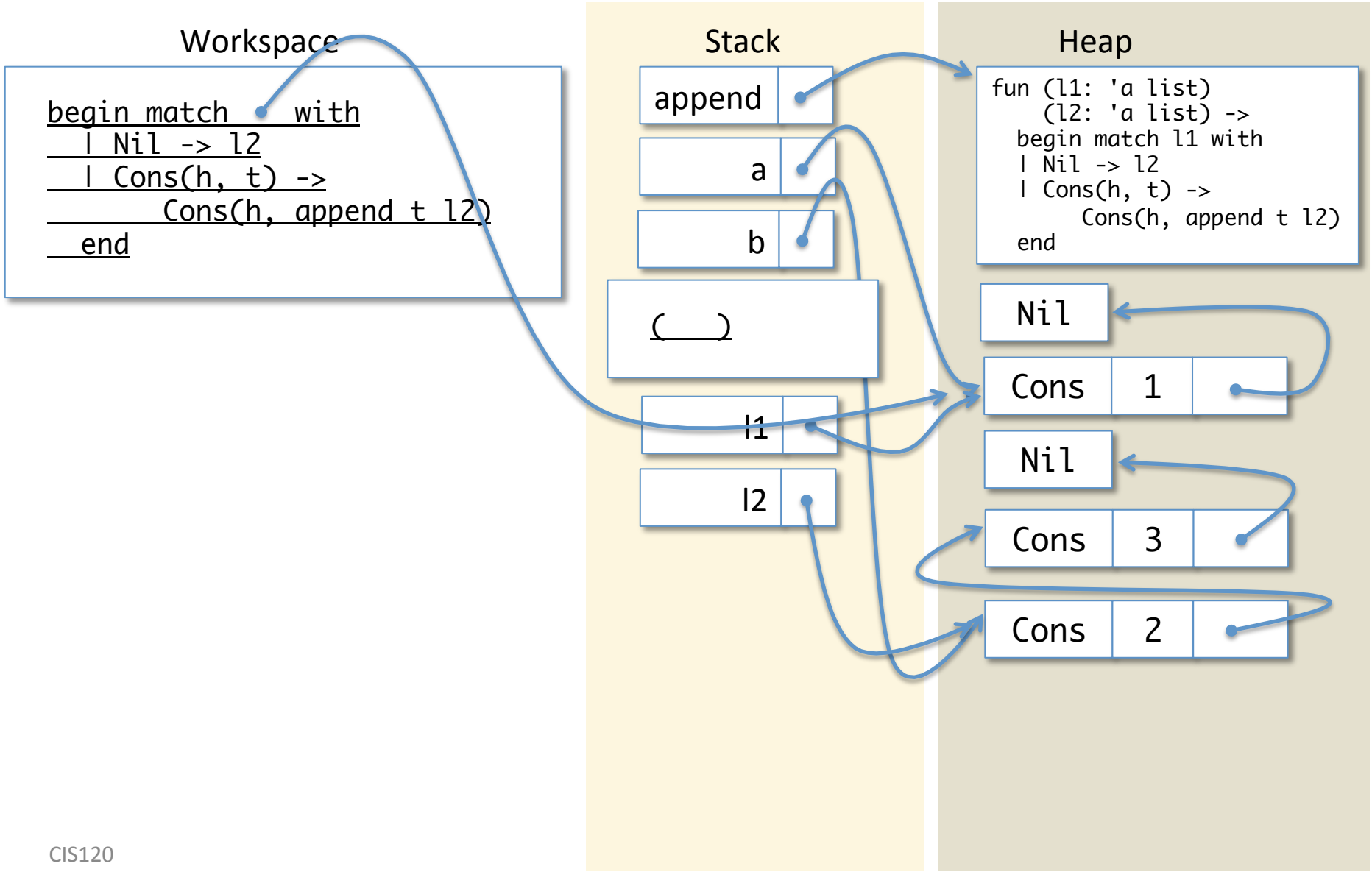
```
begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
       Cons(h, append t l2)
  end
```



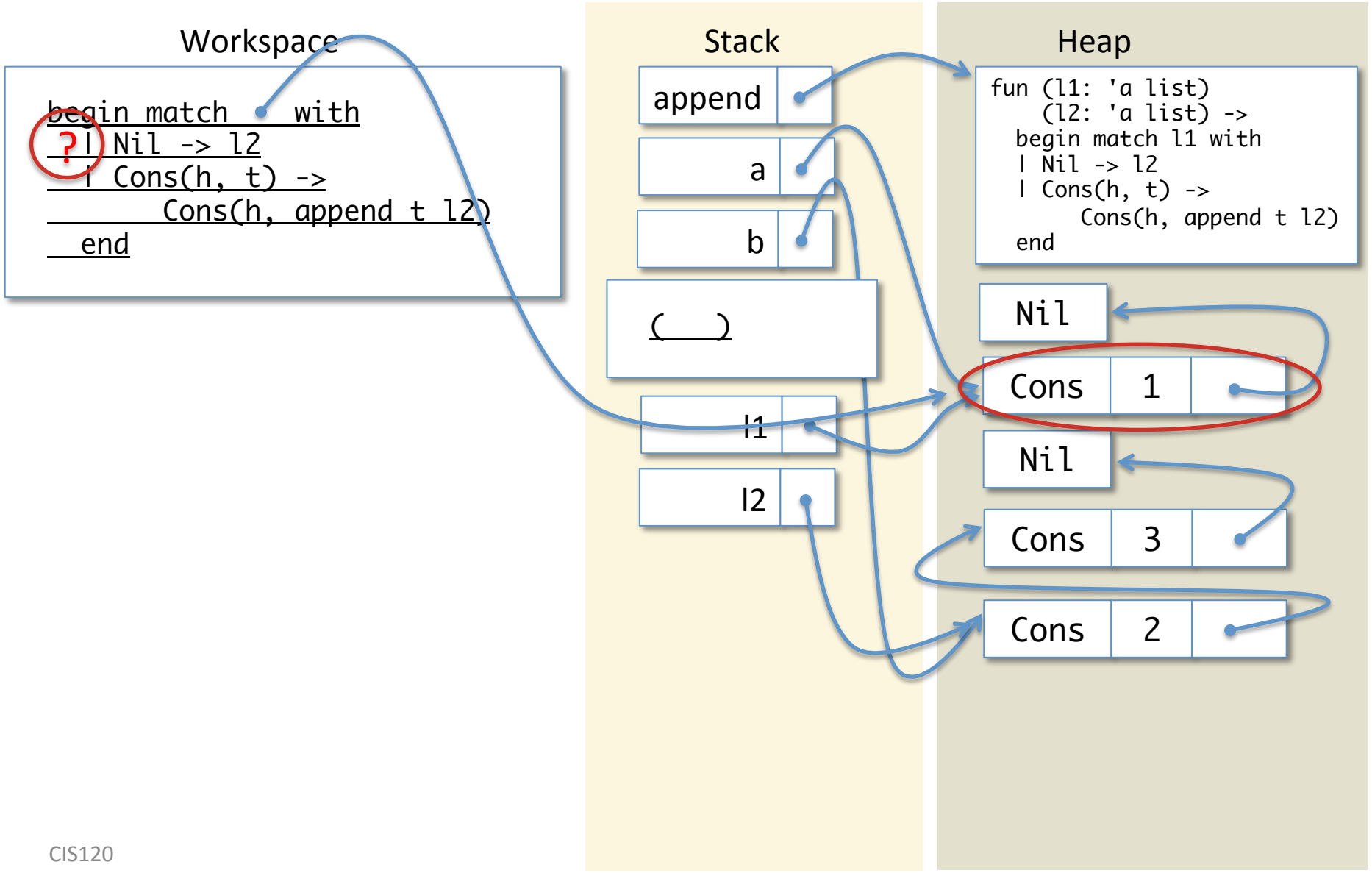
Lookup l1


Workspace
begin match with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Stack
append
a
b
( )
l1
l2
Heap
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

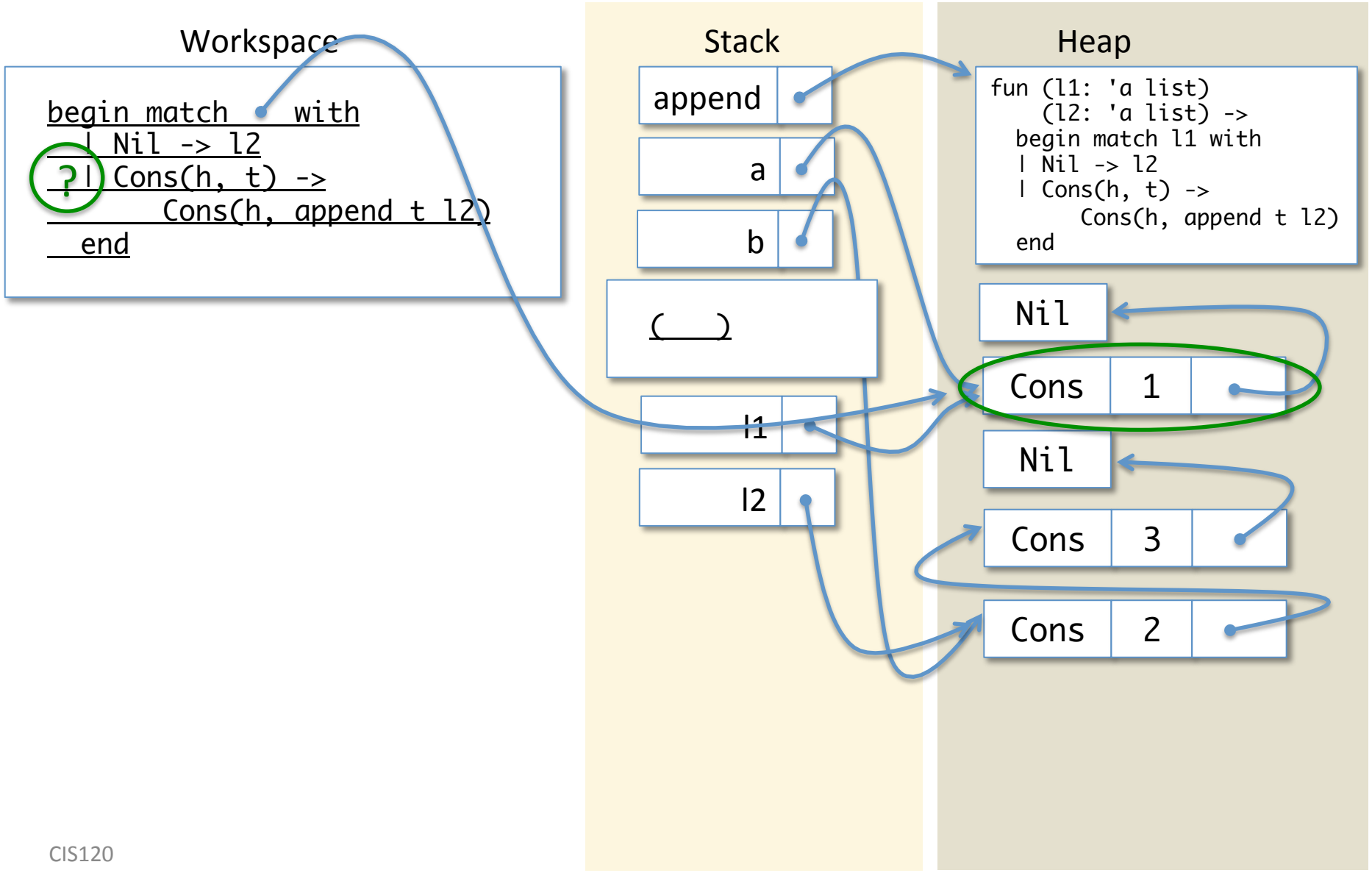
Match Expression



Nil case Doesn't Match



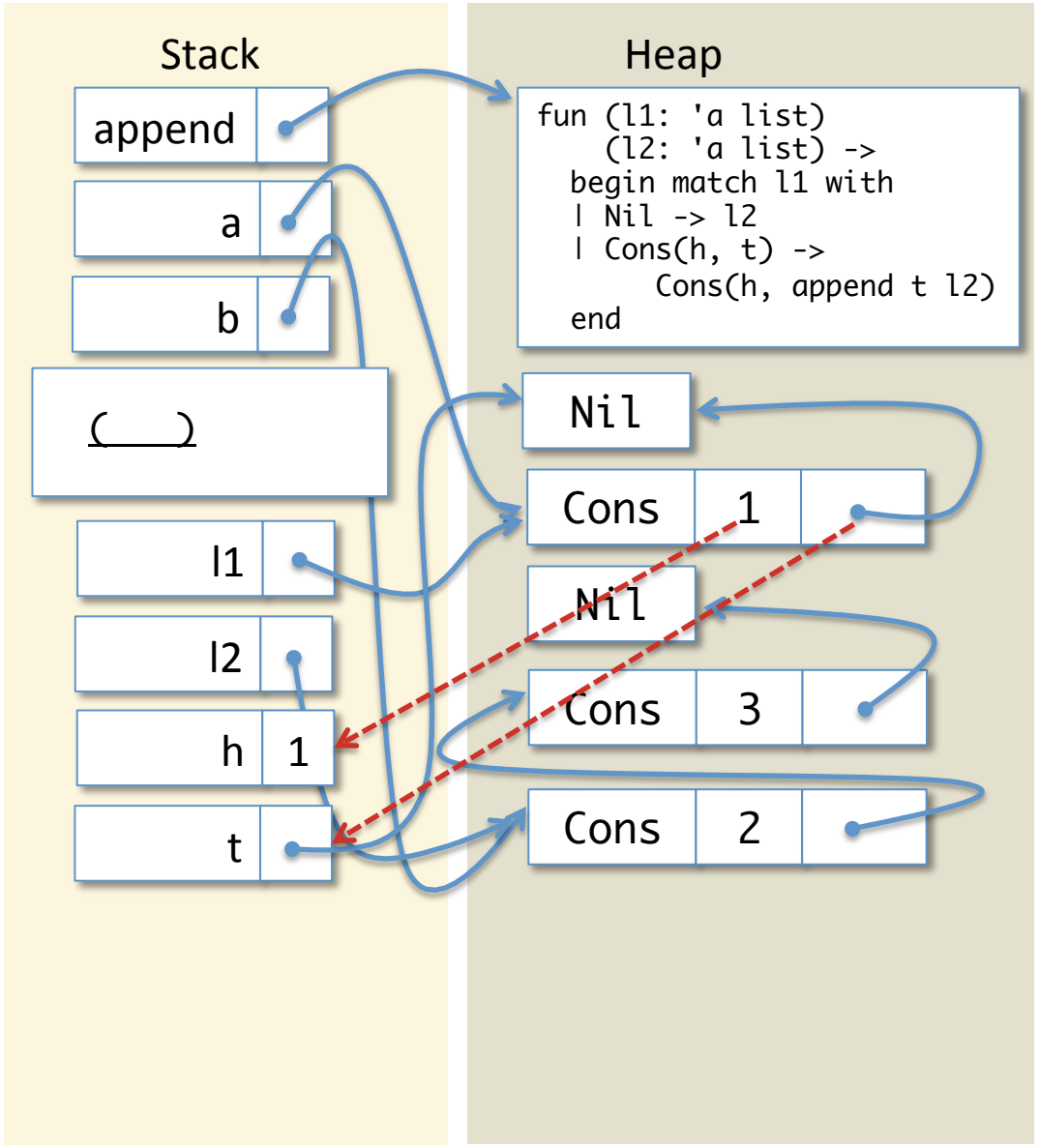
Cons case *Does* Match



Simplify the Branch: push h, t

Workspace

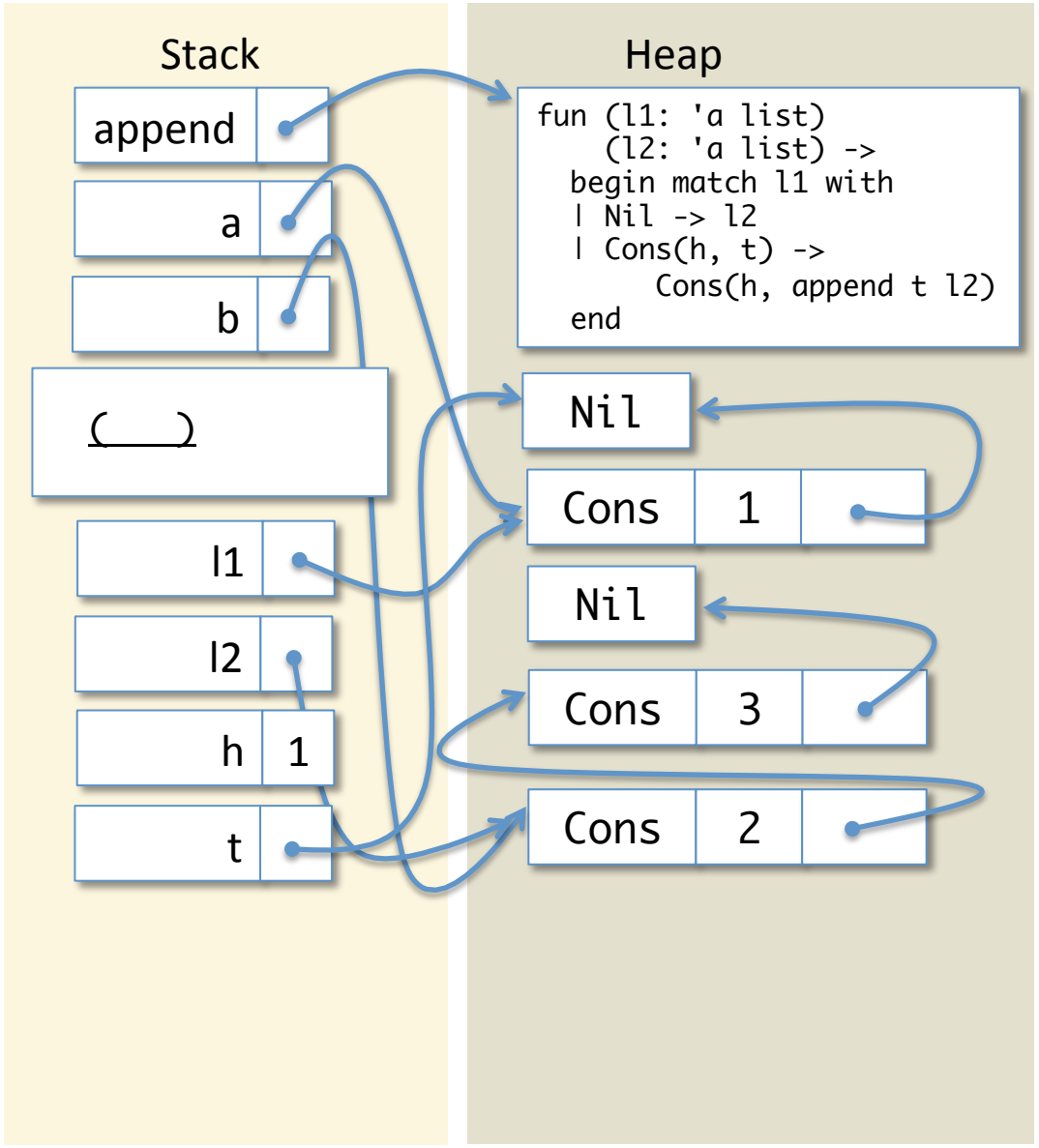
```
Cons(h, append t l2)
```



Lookup ‘h’

Workspace

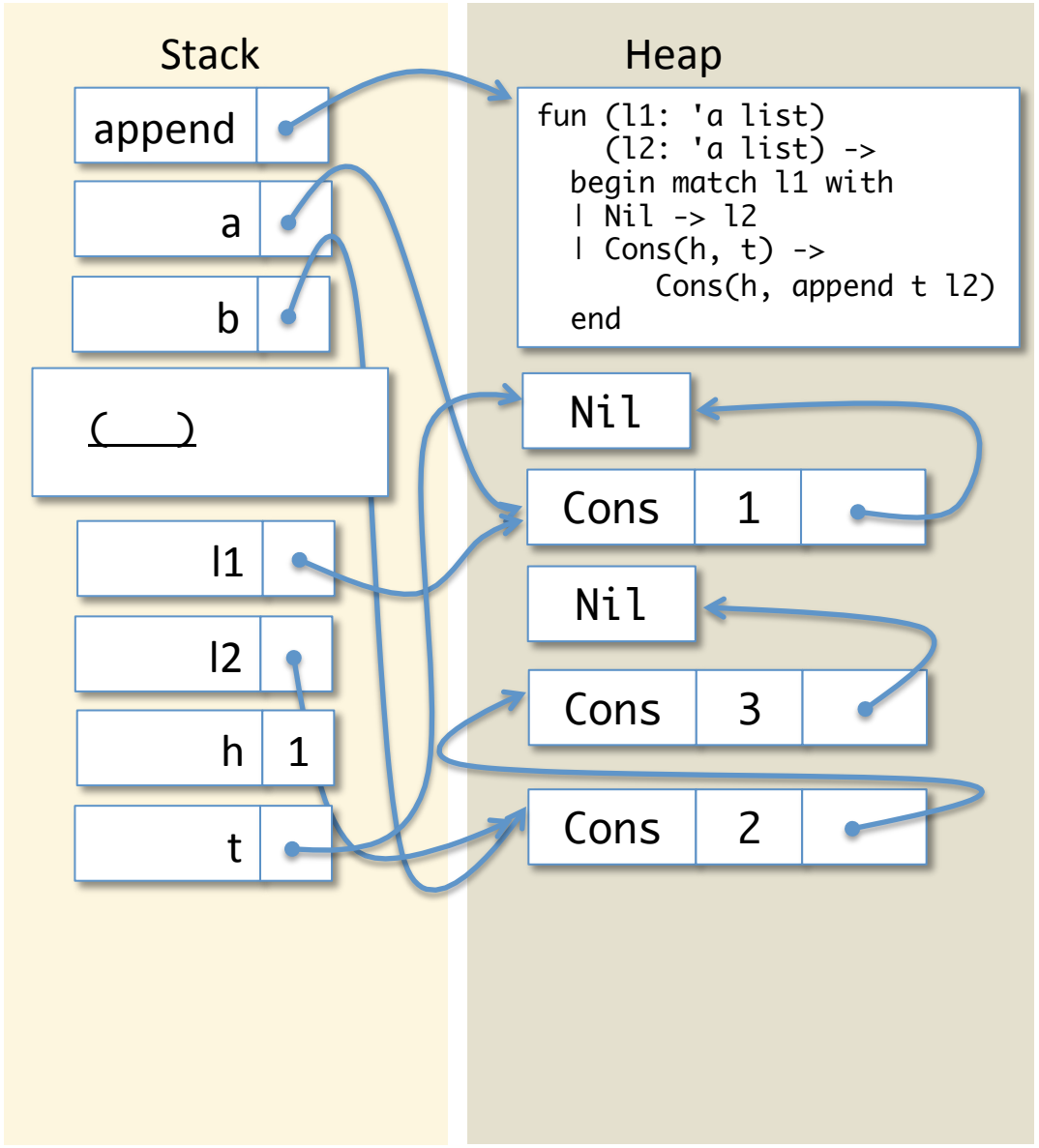
```
Cons(h, append t l2)
```



Lookup ‘h’

Workspace

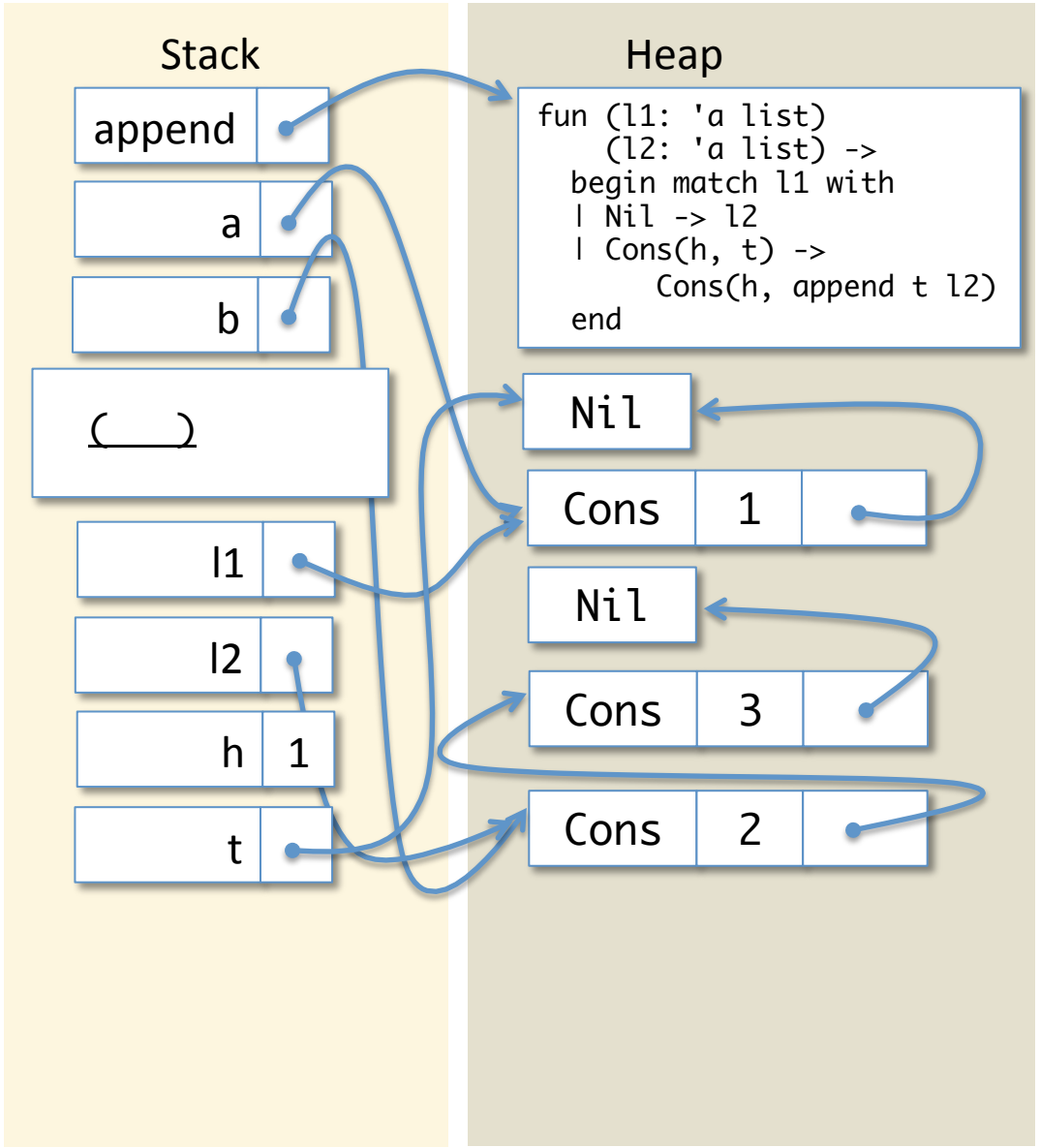
```
Cons(1, append t l2)
```



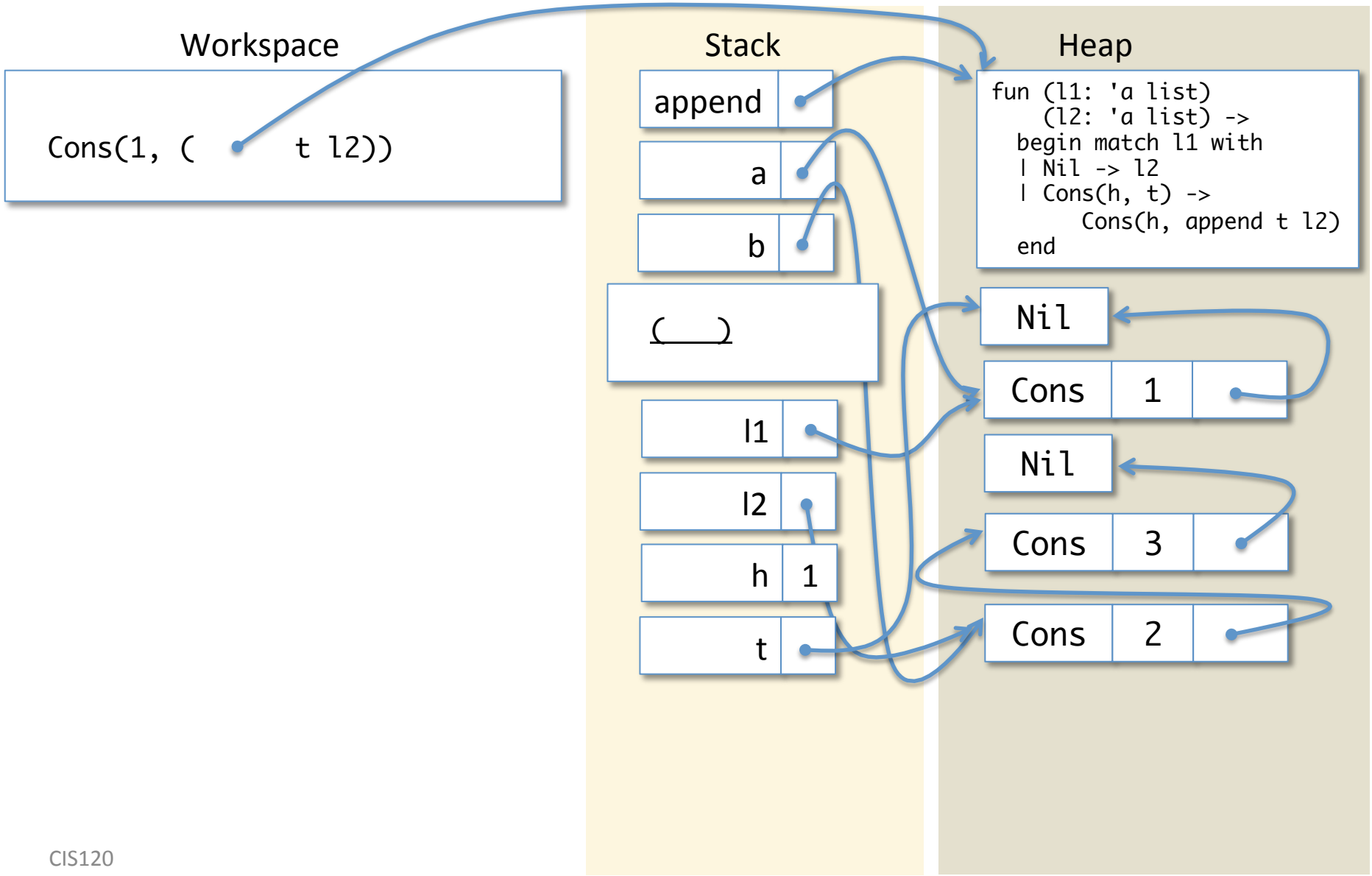
Lookup ‘append’

Workspace

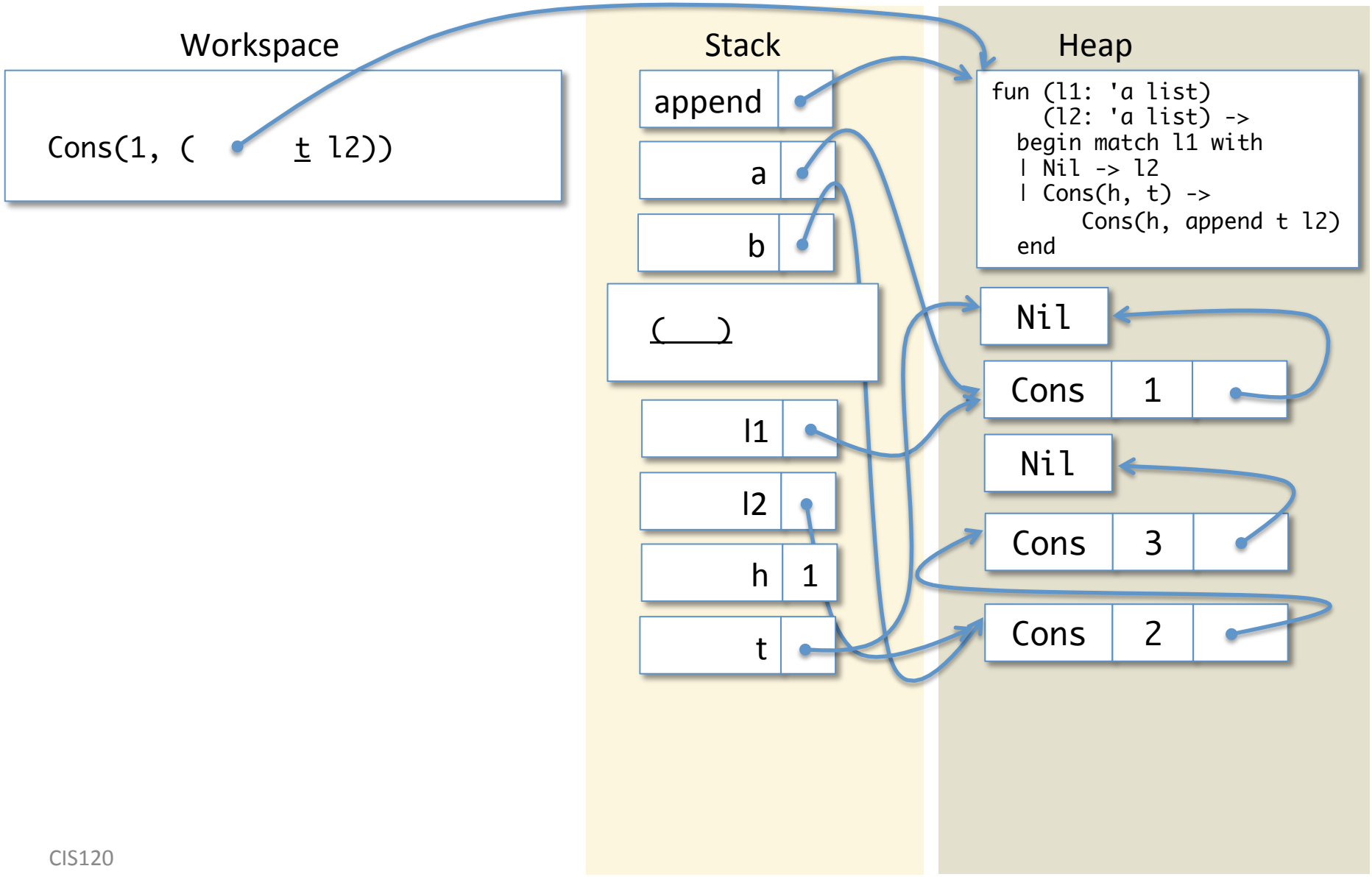
```
Cons(1, append t l2)
```



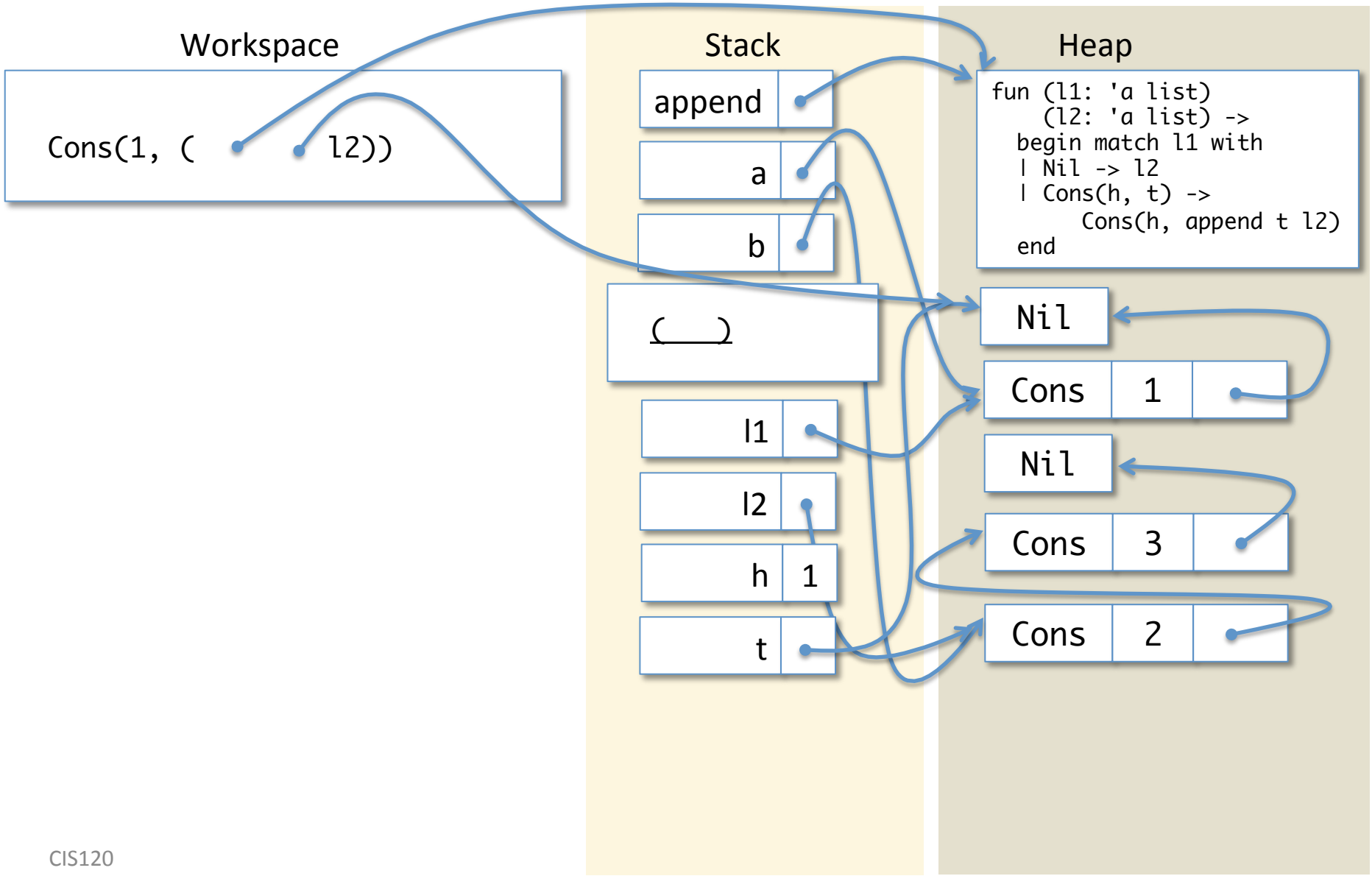
Lookup ‘append’



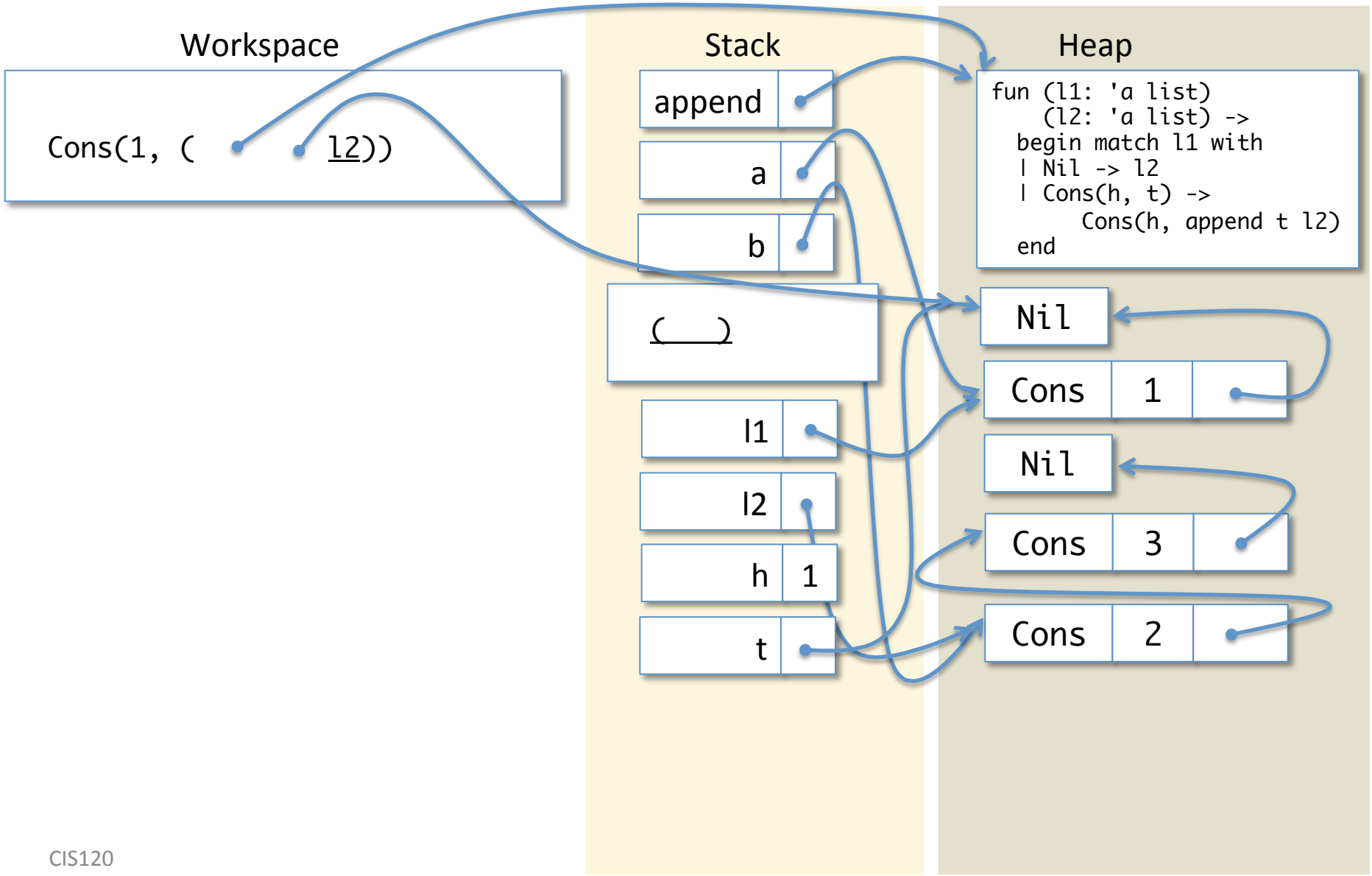
Lookup ‘t’


Workspace
Cons(1, (t l2))
Stack
append
a
b
( )
l1
l2
h 1
t
Heap
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

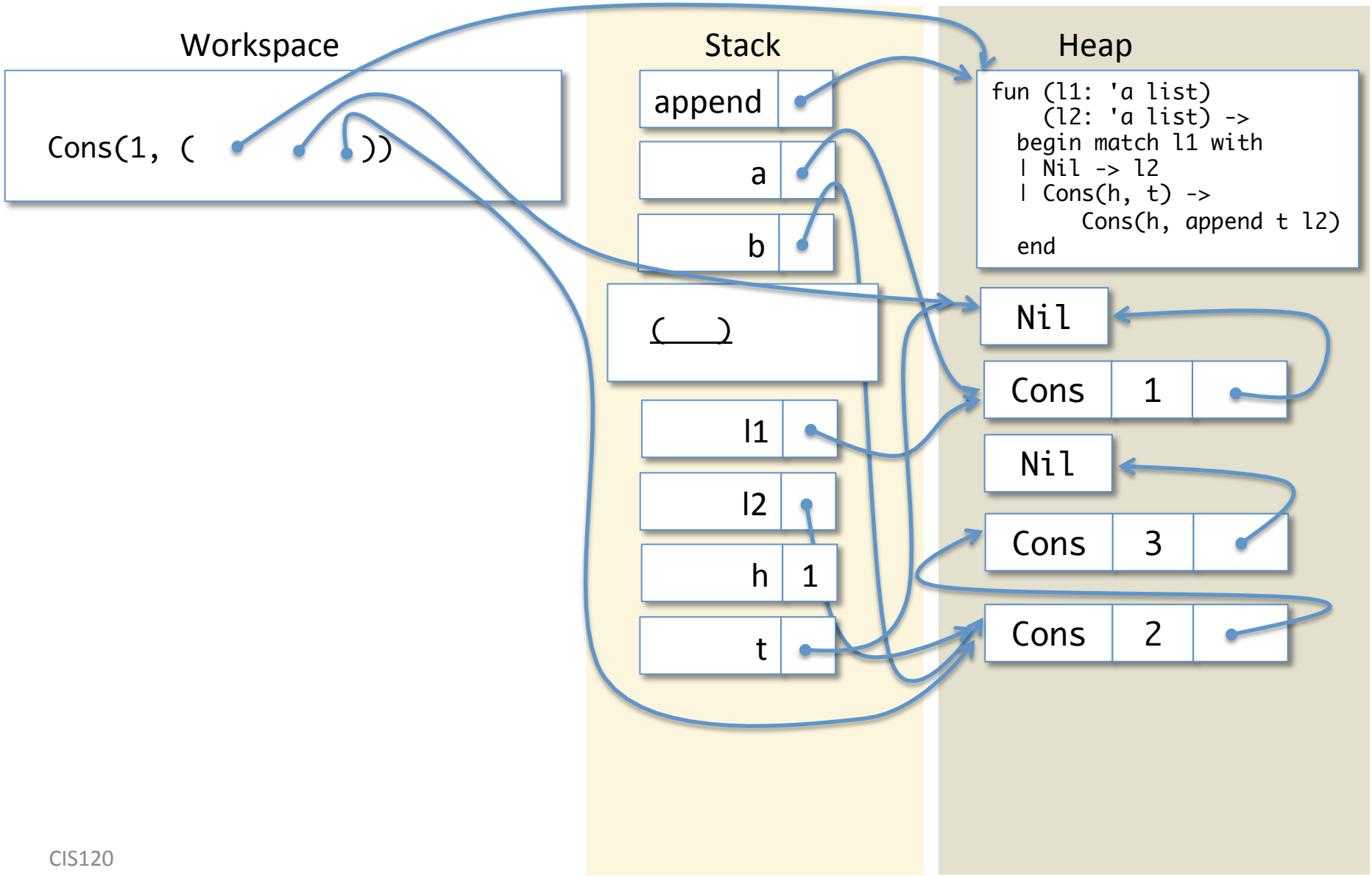
Lookup ‘t’


Workspace
Cons(1, (
l2))
Stack
append
a
b
( )
l1
l2
h
1
t
Heap
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons
1
Nil
Cons
3
Cons
2

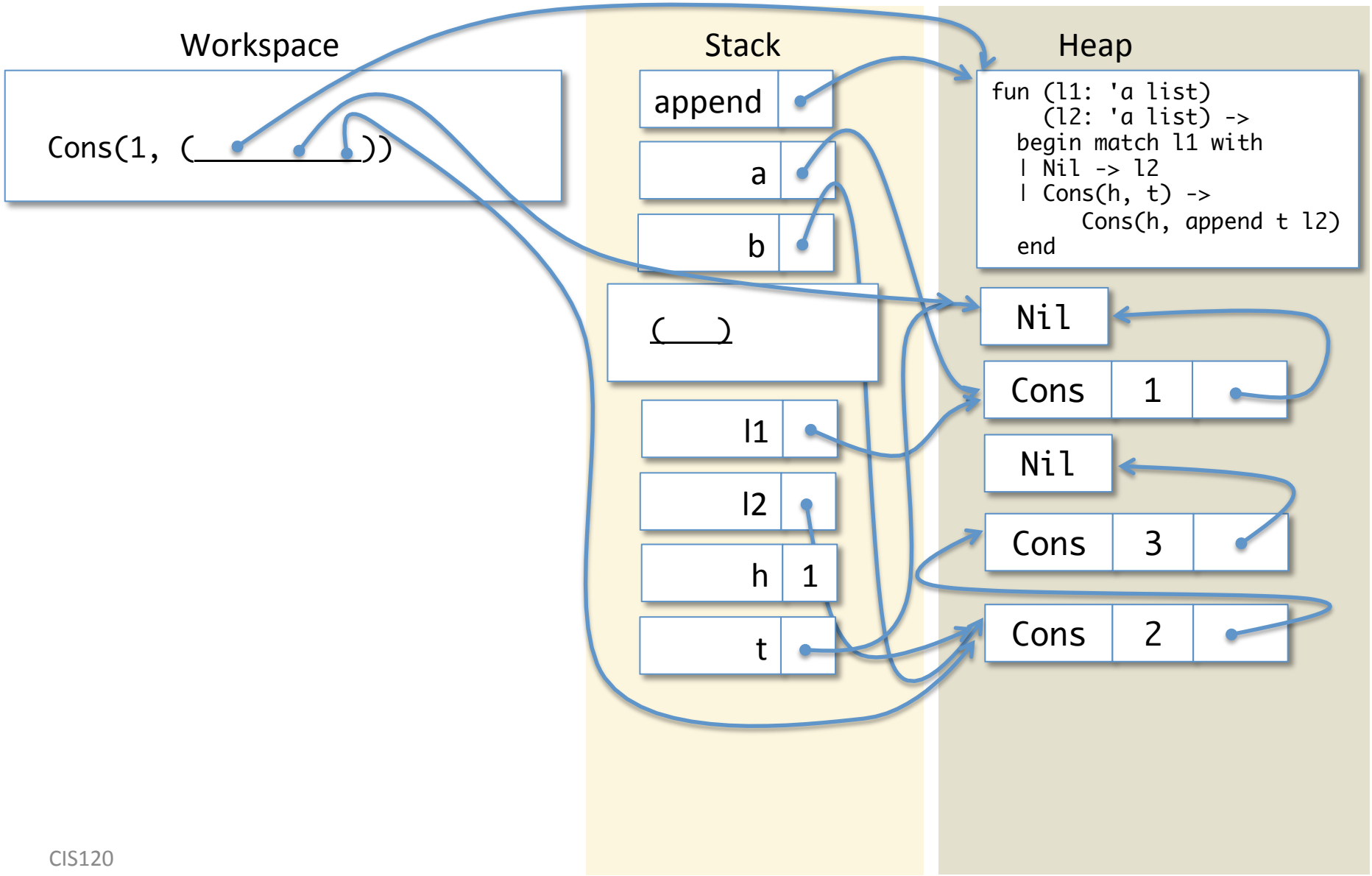
Lookup ‘l2’


Workspace
Cons(1, (l2))
Stack
append
a
b
( )
l1
l2
h 1
t
Heap
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

Lookup ‘l2’


Workspace
Cons(1, ( ))
Stack
append
a
b
( )
l1
l2
h 1
t
Heap
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

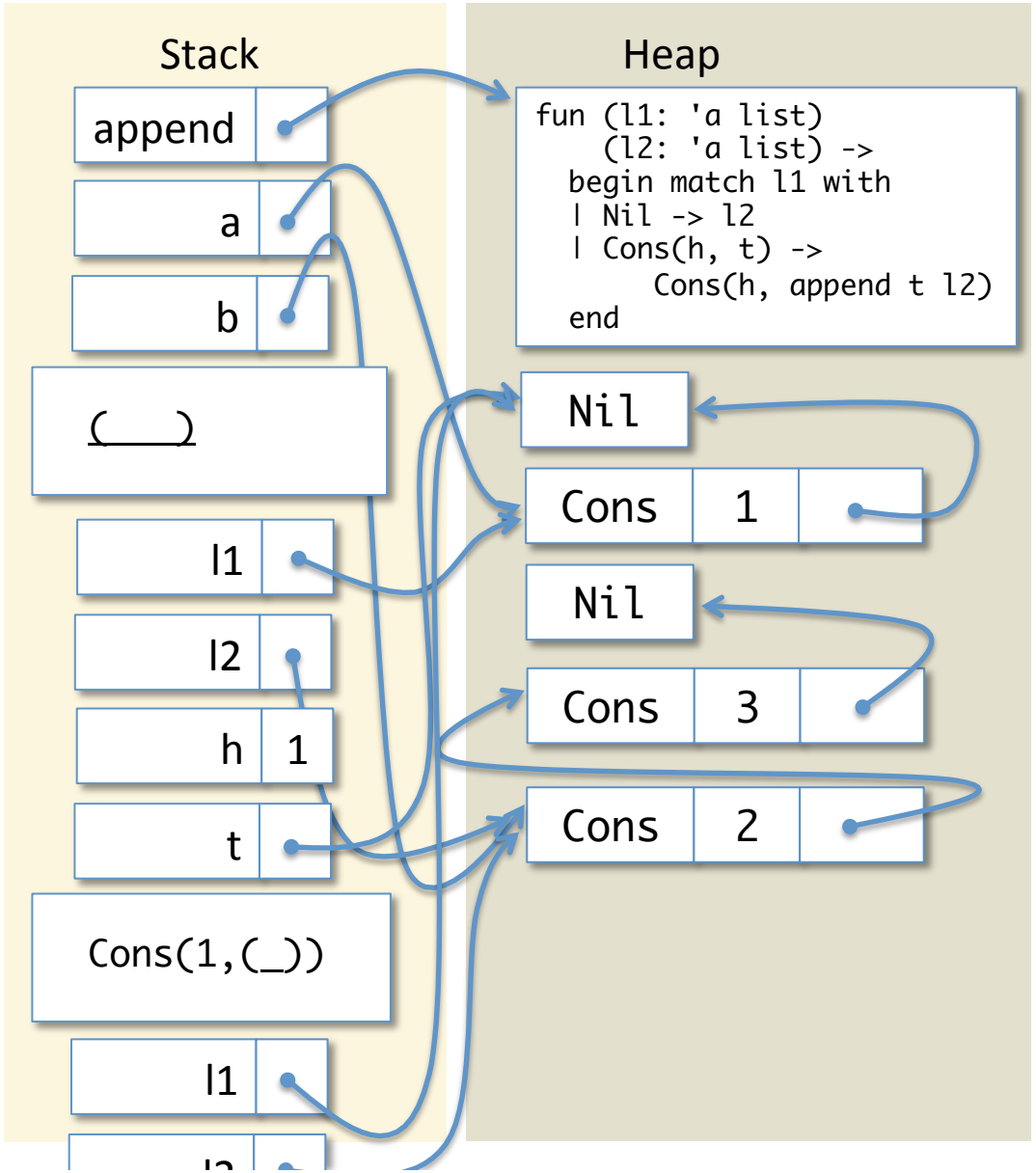
Do the Function Call



Save the Workspace; push l1, l2

Workspace

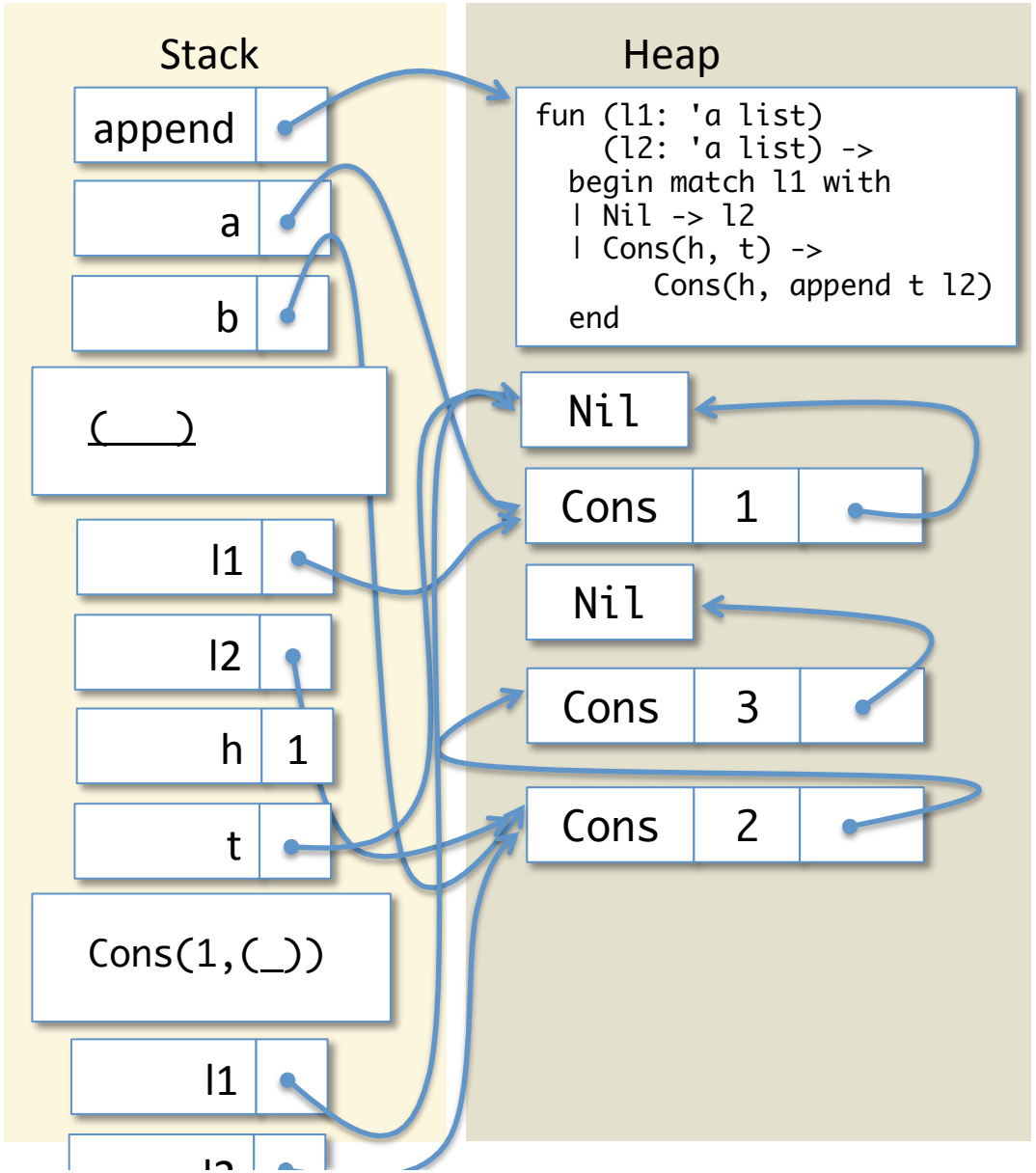
```
begin match l1 with
 | Nil -> l2
 | Cons(h, t) ->
      Cons(h, append t l2)
 end
```



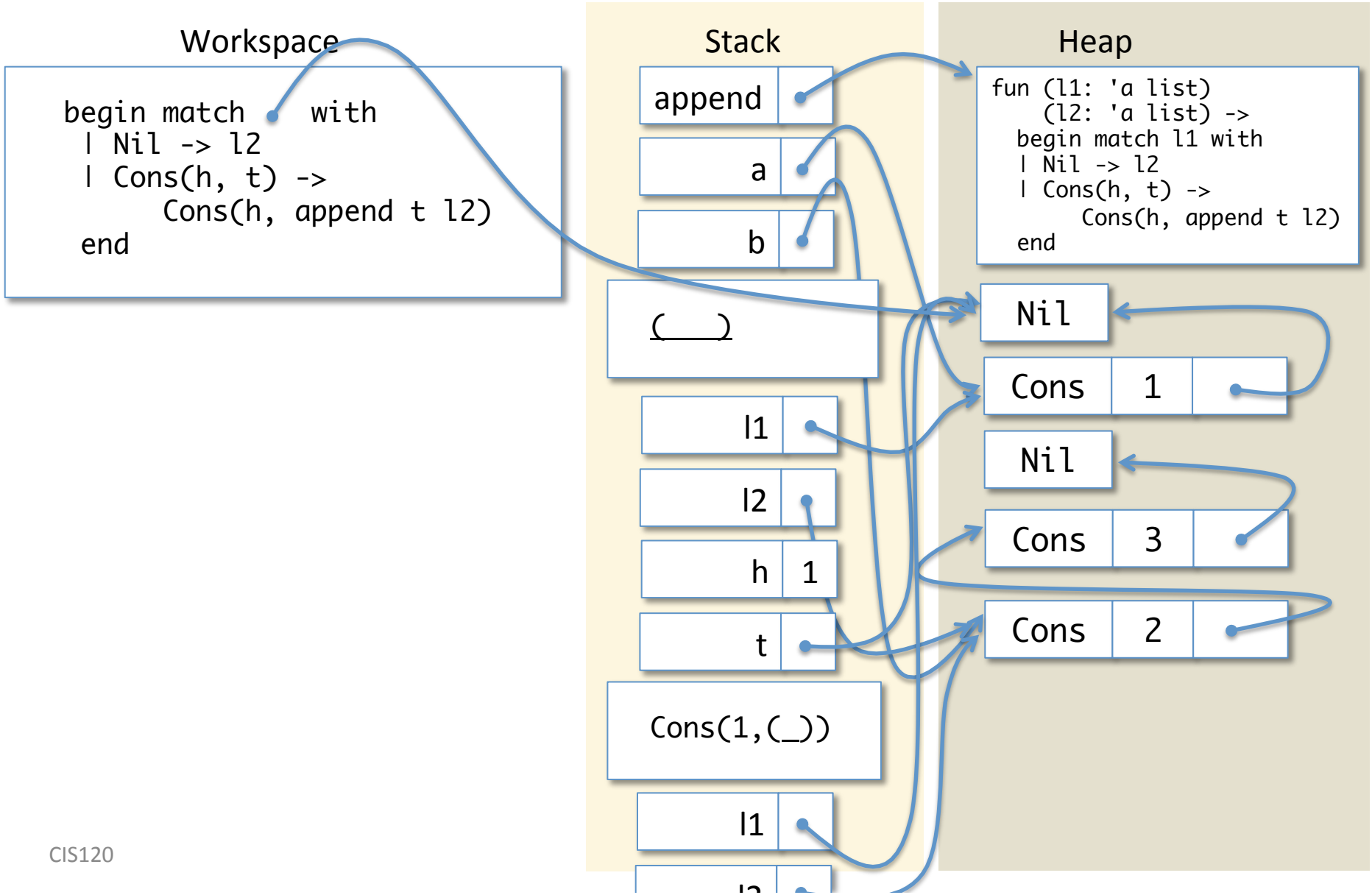
Lookup ‘l1’

Workspace

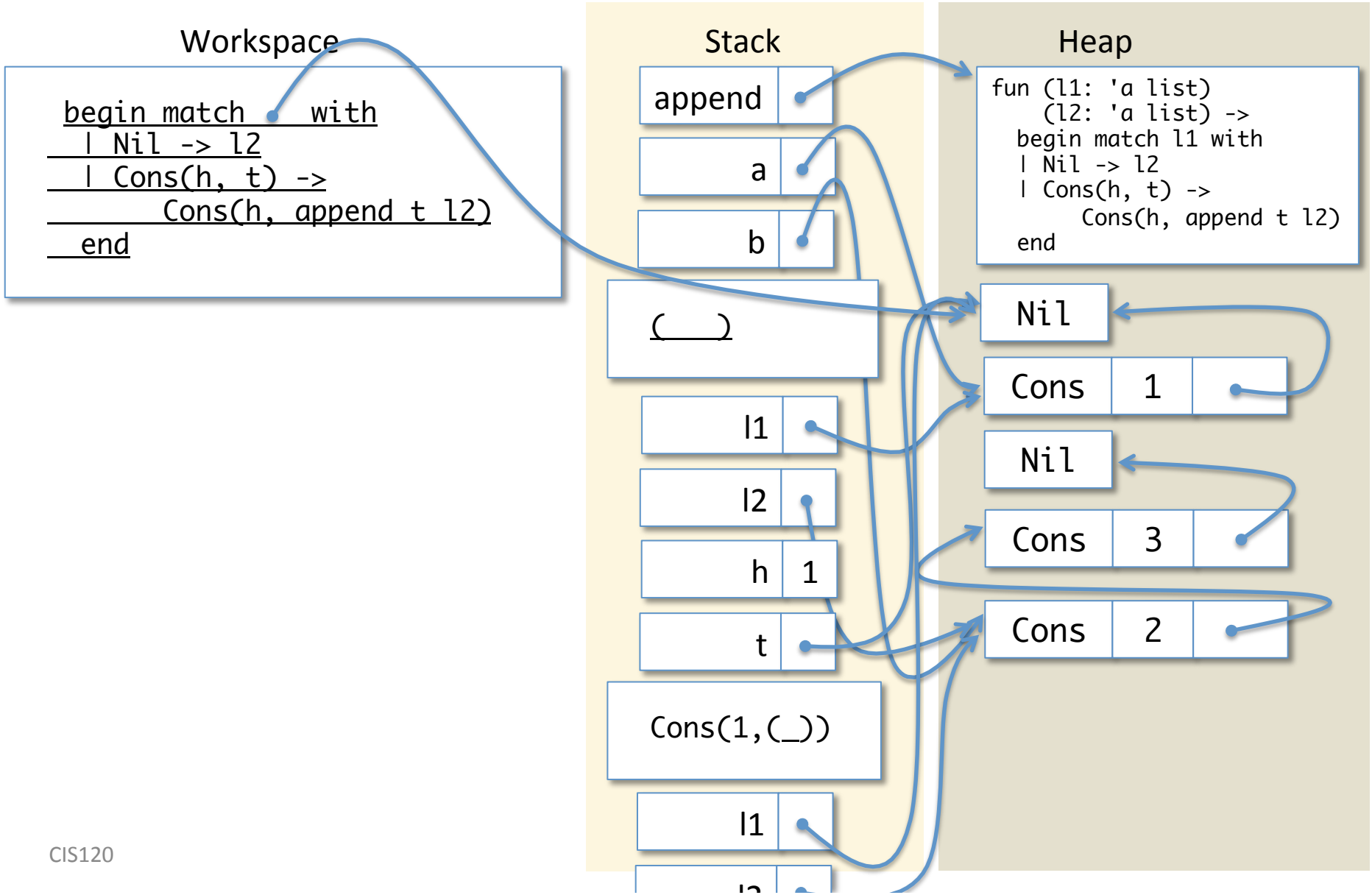
```
begin match l1 with
 | Nil -> l2
 | Cons(h, t) ->
      Cons(h, append t l2)
 end
```



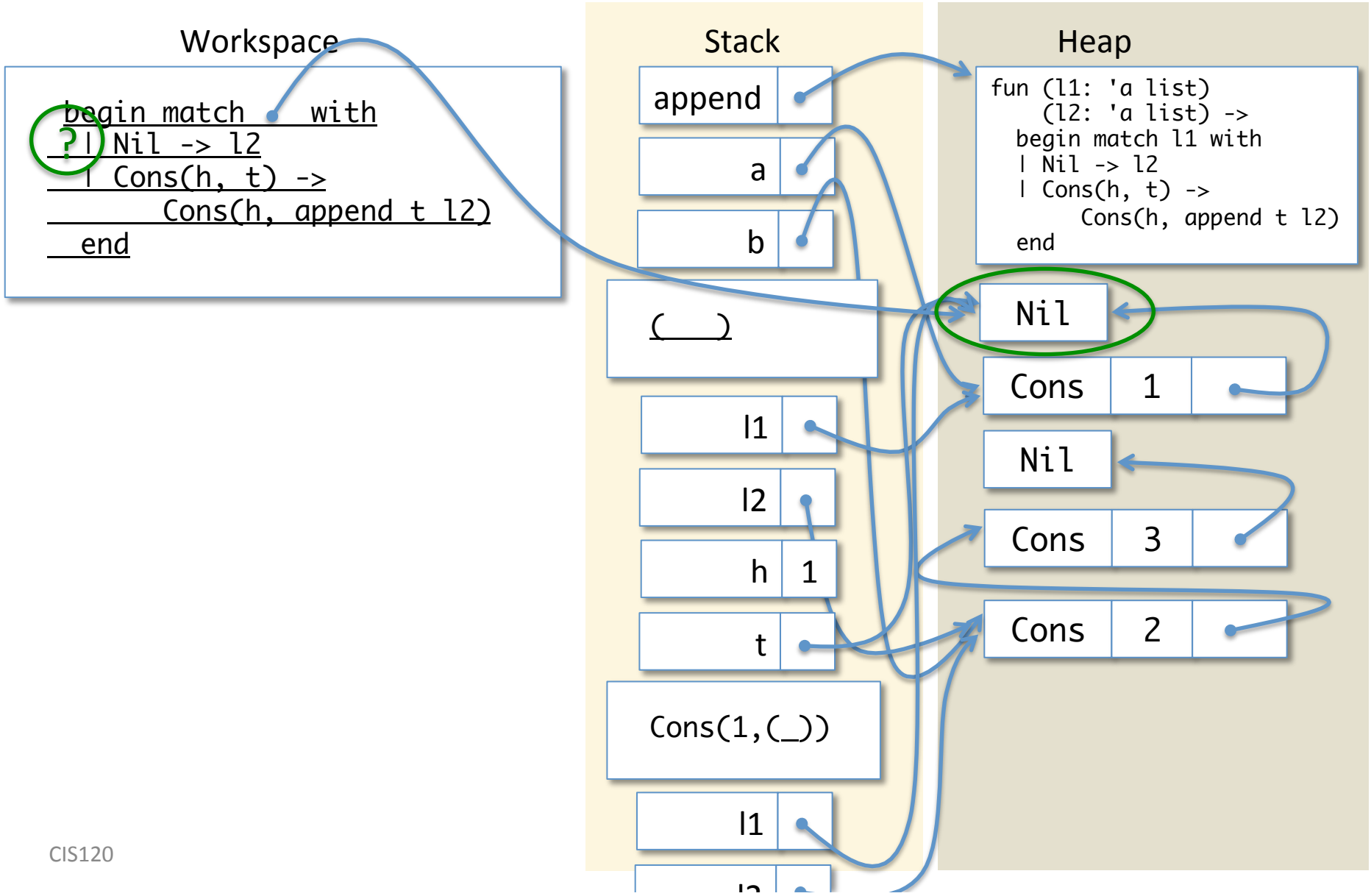
Lookup ‘l1’


Workspace
begin match with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Stack
append
a
b
(___)
l1
l2
h 1
t
Cons(1,(_))
l1
Heap
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

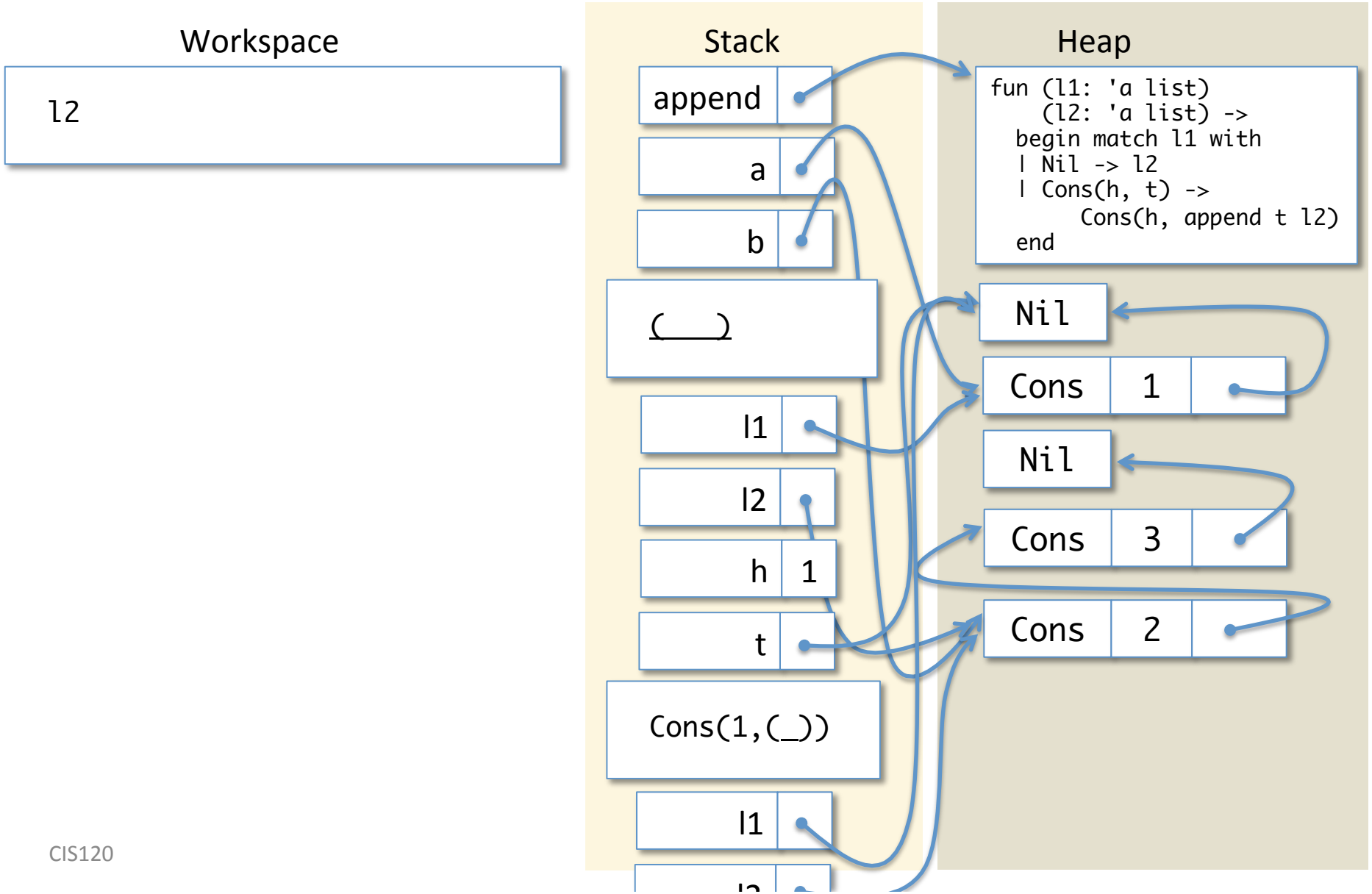
Match Expression



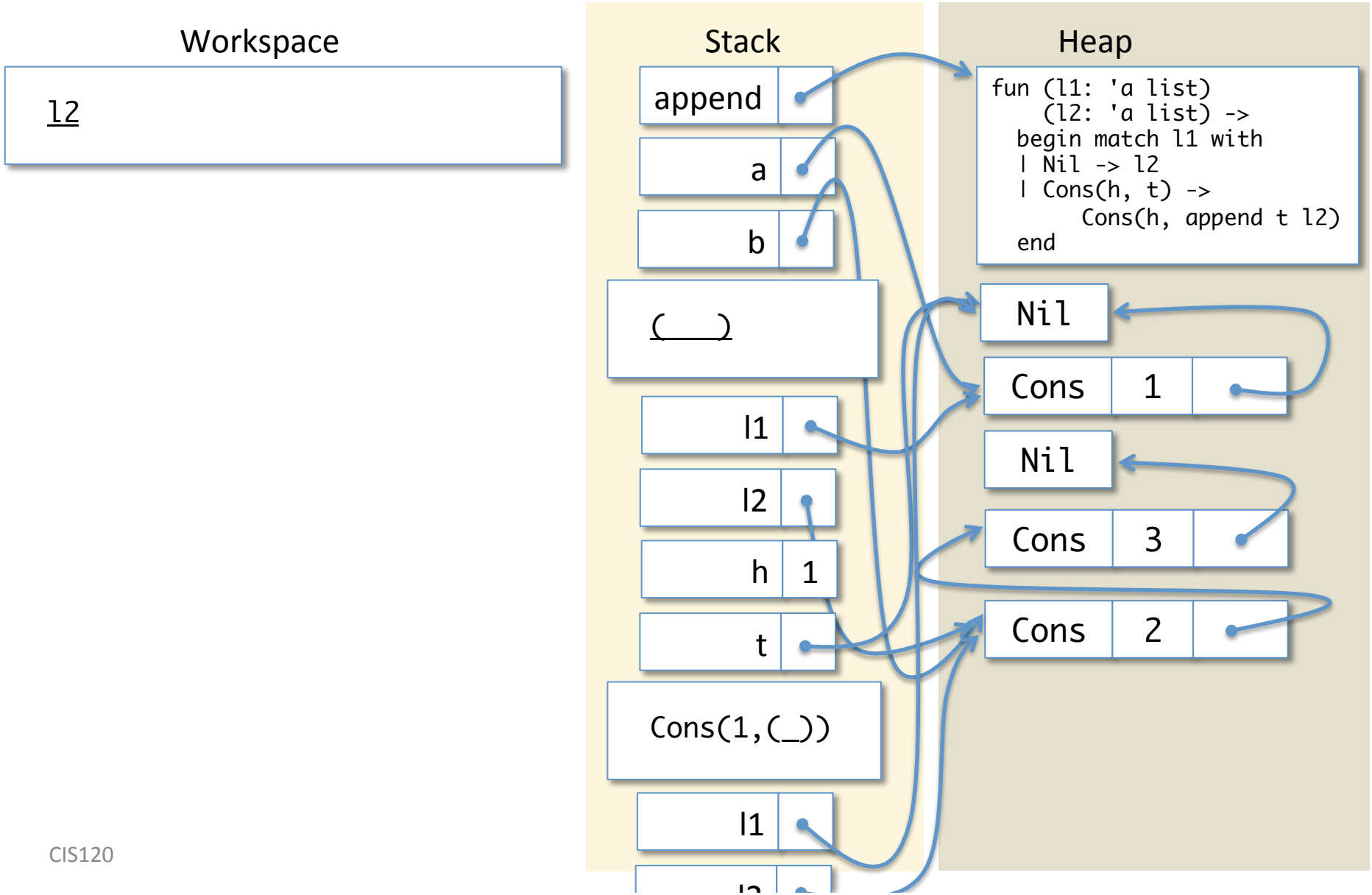
The Nil case Matches



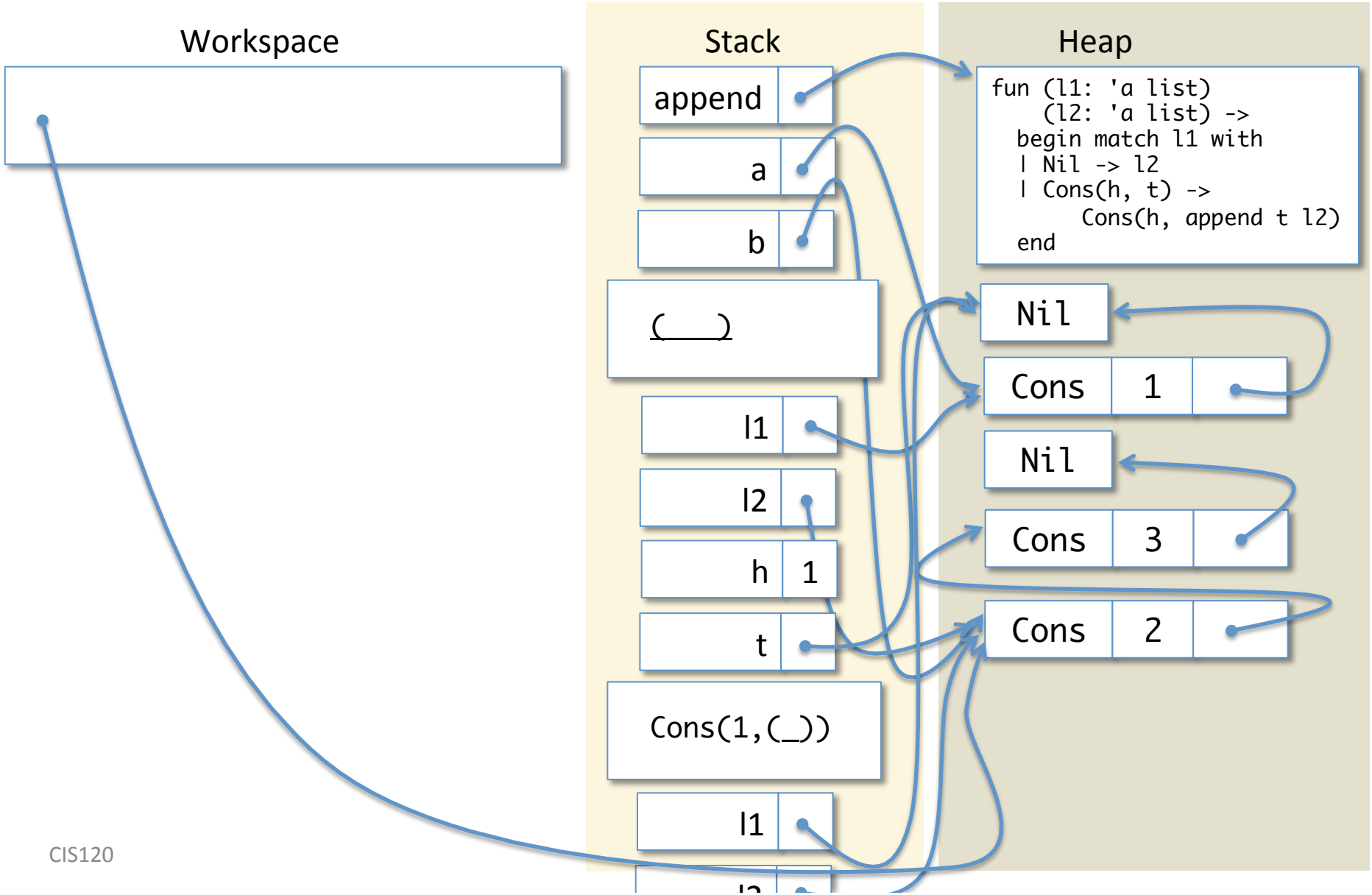
Simplify the Branch (nothing to push)



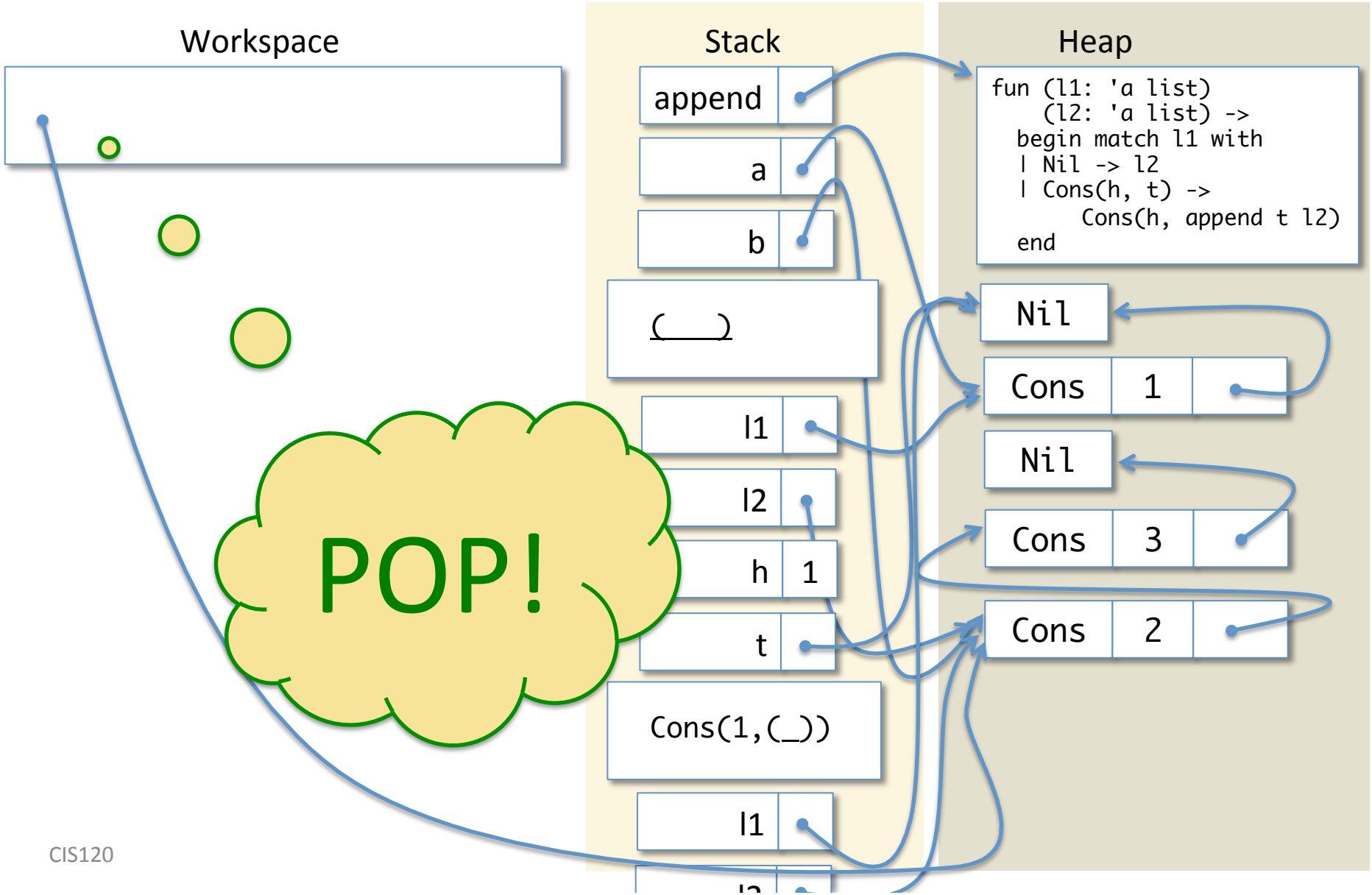
Lookup ‘l2’


Workspace
l2
Stack
append
a
b
(___)
l1
l2
h 1
t
Cons(1,(_))
l1
Heap
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

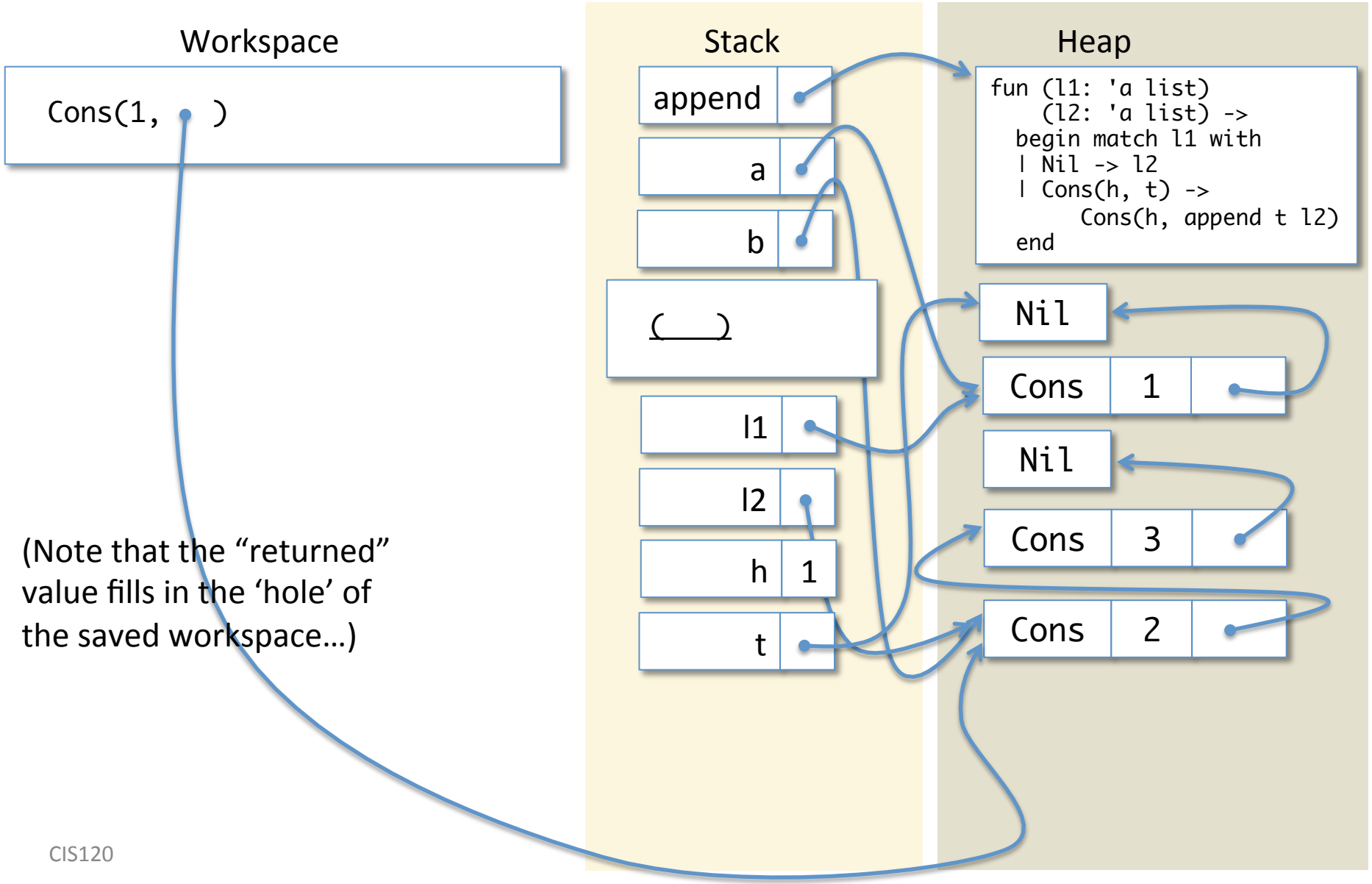
Lookup ‘l2’


Workspace
Stack
Heap
append
a
b
(___)
l1
l2
h 1
t
Cons(1,(_))
l1
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

Done! Pop stack to last Workspace

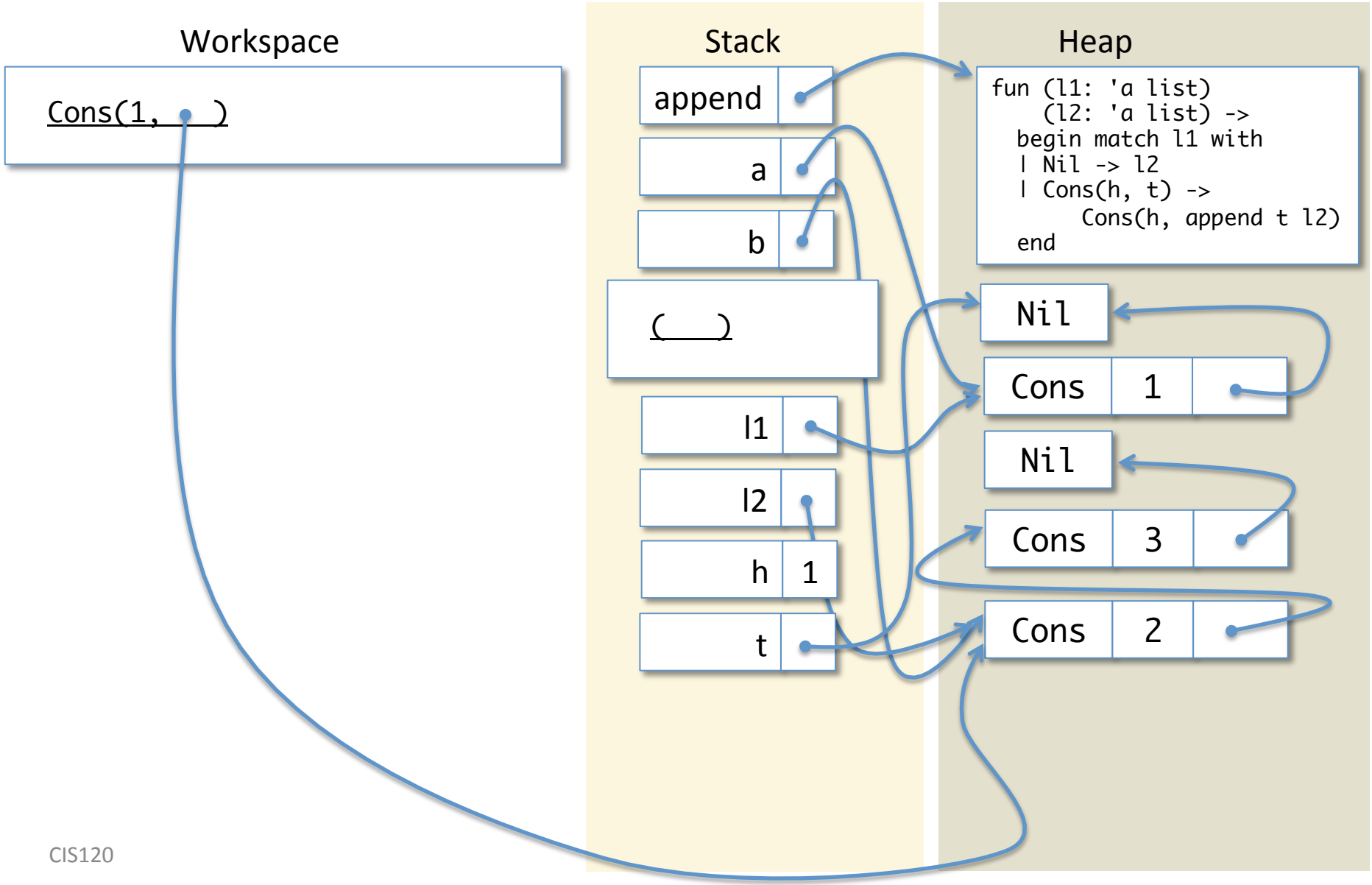


Done! Pop stack to last Workspace

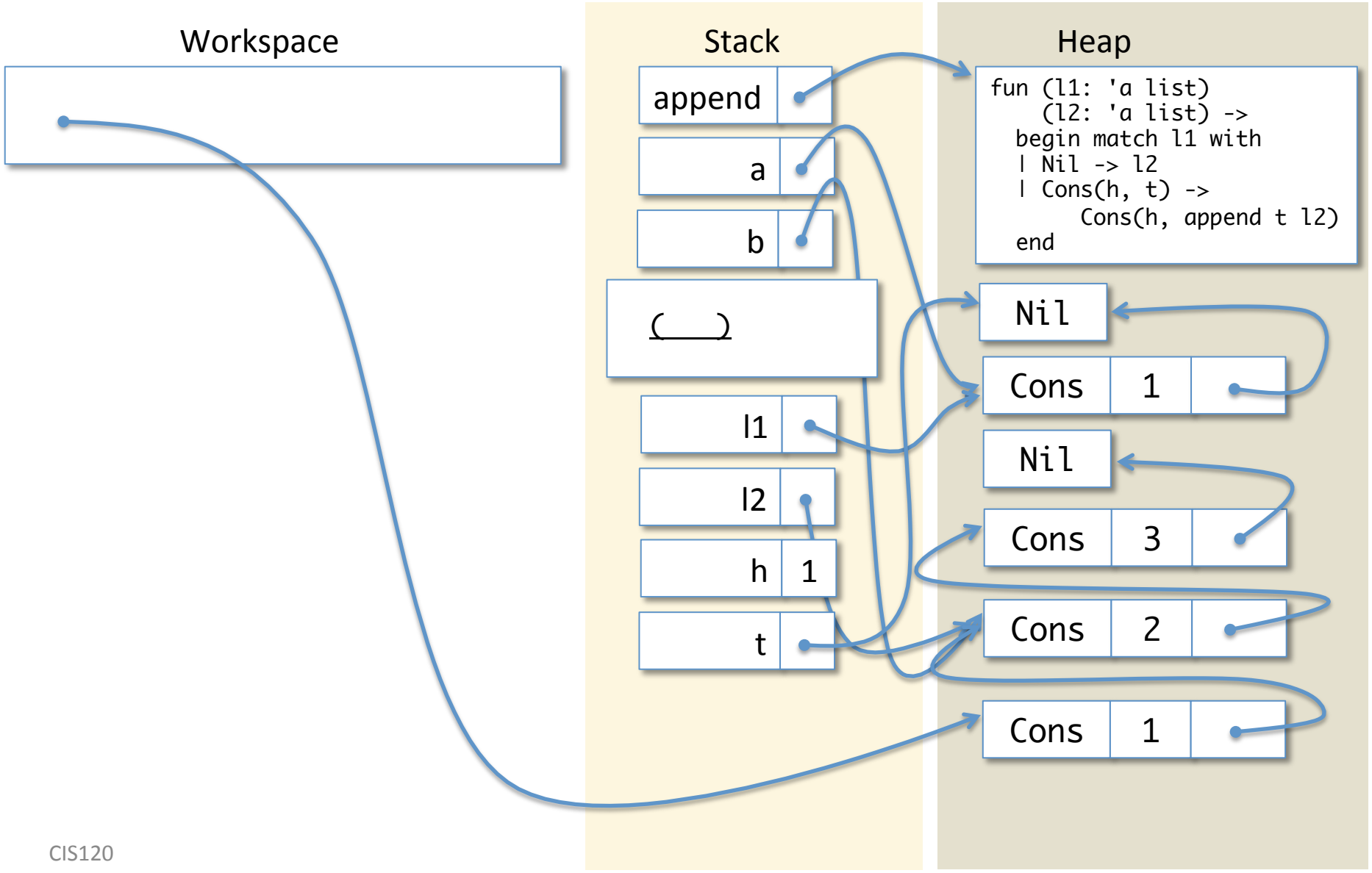


(Note that the "returned" value fills in the 'hole' of the saved workspace…)

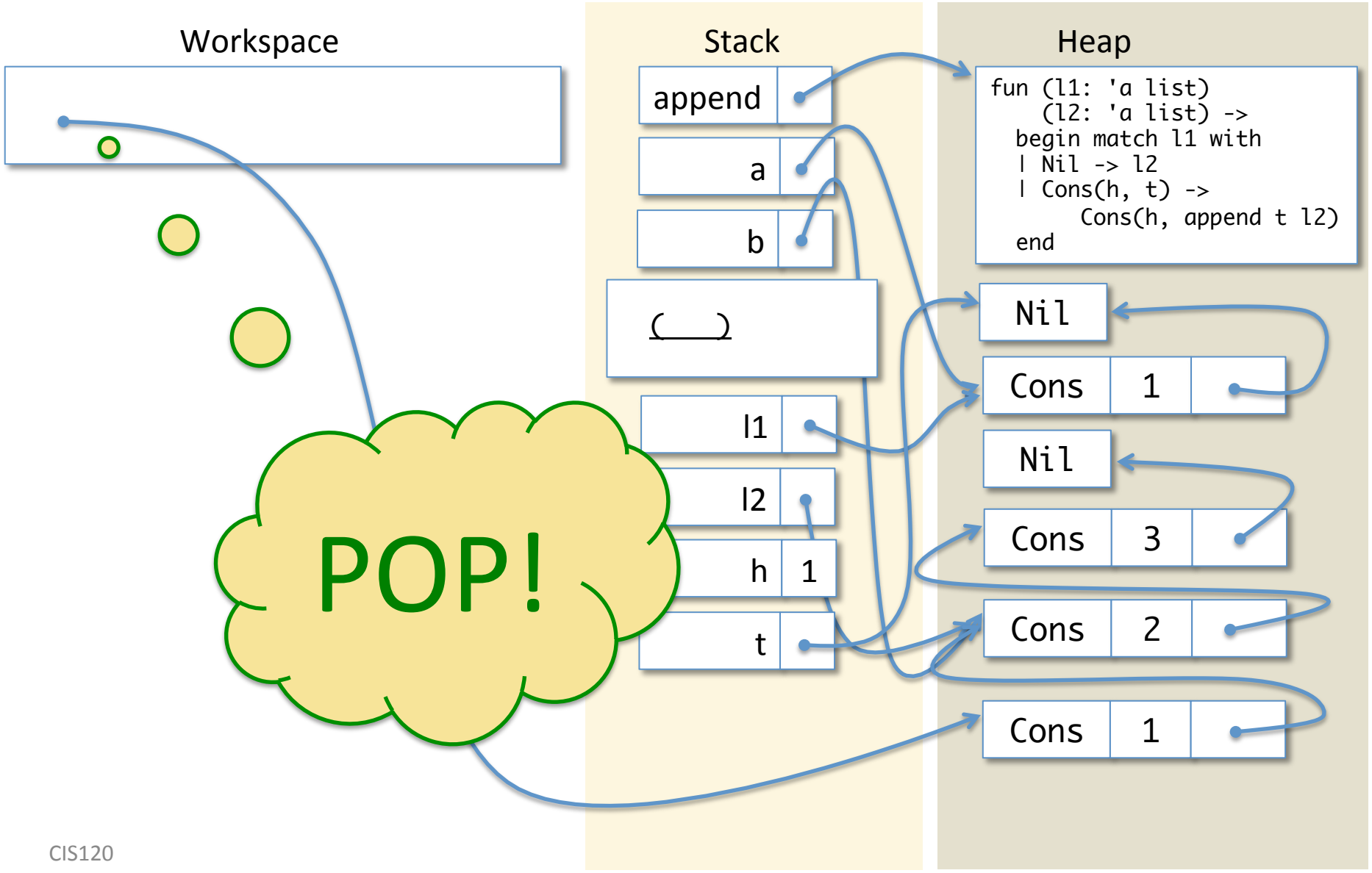
Allocate a Cons cell



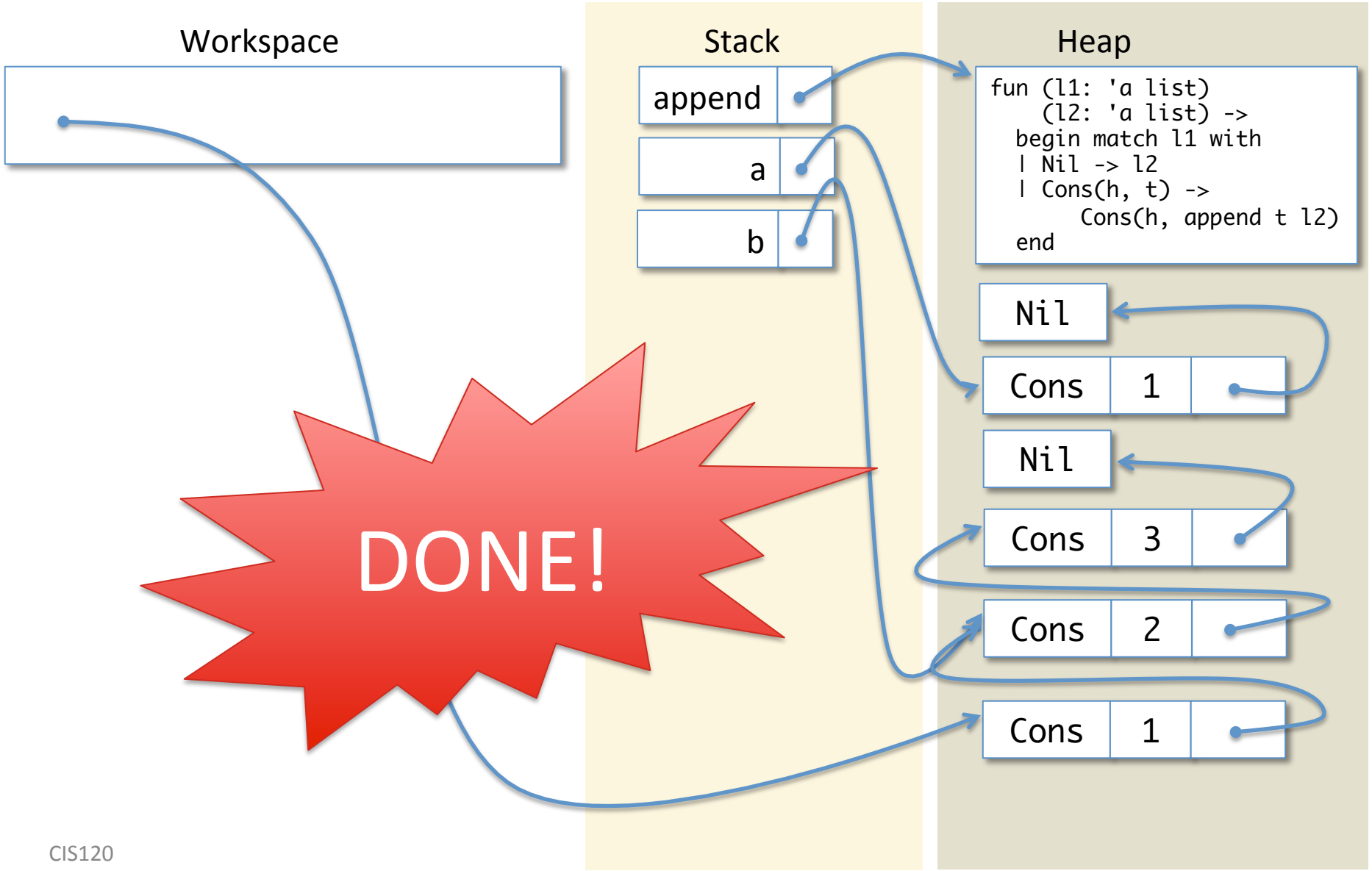
Allocate a Cons cell



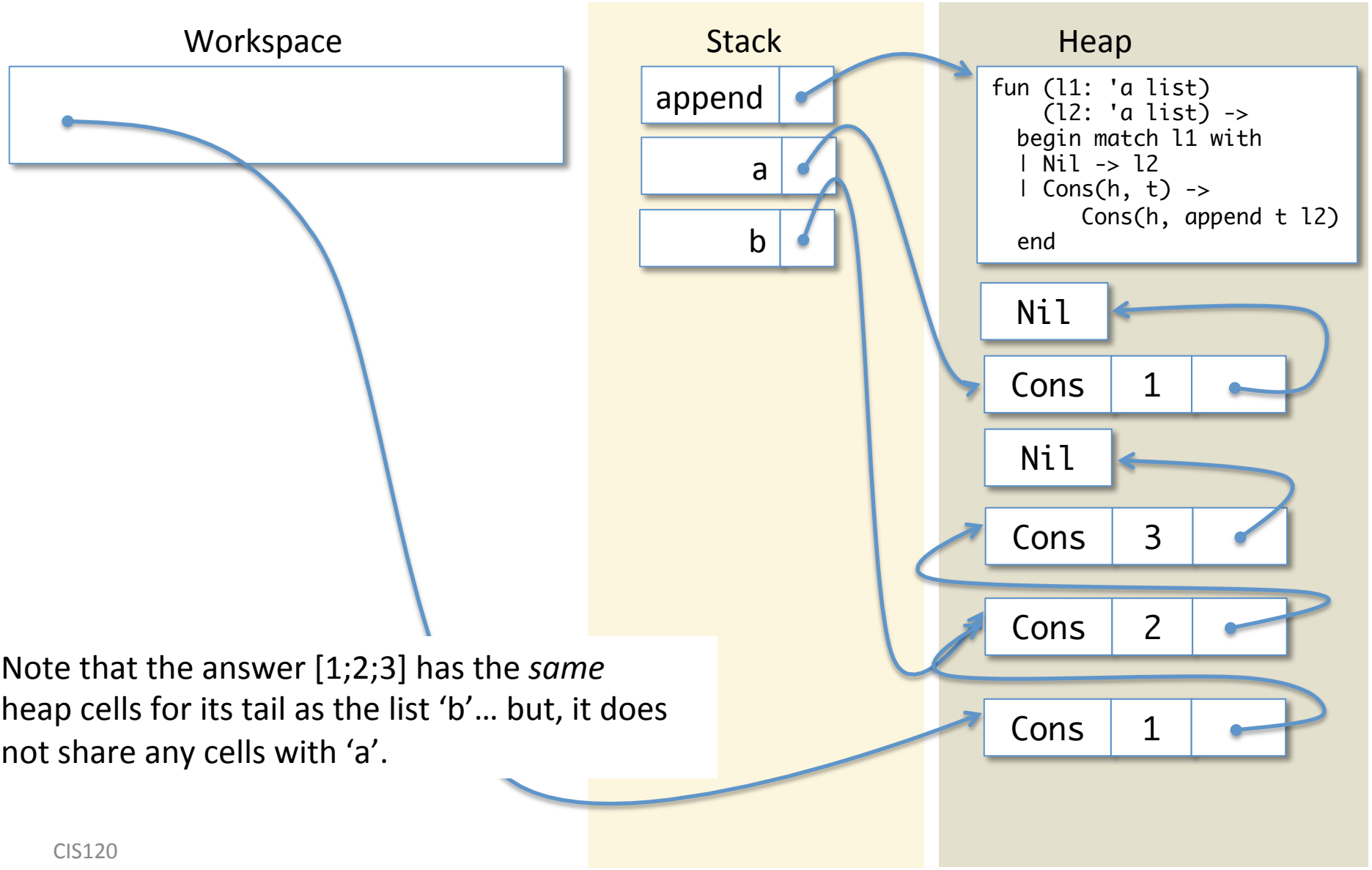
Done! Pop stack to last Workspace



Done! (PHEW!)


Workspace
Stack
Heap
append
a
b
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2
Cons 1
DONE!

Done! (PHEW!)



Note that the answer [1;2;3] has the *same* heap cells for its tail as the list 'b'… but, it does not share any cells with 'a'.

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 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 $branch_i$