Programming Languages and Techniques (CIS120)

Lecture 13

February 12th, 2016

ASM

Chapter 14

Announcements

- Midterm 1 will be Tuesday evening
  - ROOMS announced on Monday
  - TIME: 6:15PM
  - Covers up to Feb 10th and HW 3
    - no options, records, or Abstract Stack Machine!

- Review session Sunday, Feb 14th 6-8PM in Towne 100

Has this situation ever happened to you?

1. yes
2. no

Have you used the substitution model to reason about how functions evaluate?

```
filter is_even [1;2]
⟼ if is_even 1 then 1 :: filter is_even [2]
   else filter is_even [2]
⟼ if false then 1 :: filter is_even [2]
   else filter is_even [2]
⟼ filter is_even [2]
⟼ if is_even 2 then 2 :: filter is_even []
   else filter is_even []
⟼ 2 :: filter is_even []
⟼ 2 :: []
```

```
let filter (f : 'a -> bool)
           (l : 'a list) : 'a list =
  begin match l with
  | []       -> []
  | hd :: tl ->
    if f hd then hd :: filter f tl
      else filter f tl
  end
```

1. yes, every single step
2. yes, but skipping some steps
3. no, it seems useless to me
4. what is the substitution model?

Modeling (Stateful) Computation

Models of Computation

- Explain the behavior of your program.
  - i.e. the *meaning* or *semantics*

- Substitution model works for pure functional programs…

…but:

- How do we *implement* the substitution model?
- How do we explain behaviors like:

```
Stack overflow during evaluation (looping recursion?).
```

- Where do the lists and trees we construct live in memory?

Towards Imperative Programs

- What about program features that update state:
- E.g. in Java

```
int x = 3;
x = x + 1;          //  what does this do?
```

```
int x = 3;
int y = x;
y = y + 1;
```

vs.

```
DataObj x = new DataObj();
DataObj y = x;
y.field = y.field + 1;
```

- Other features: Exceptions, Dynamic Dispatch, Threads, etc.

Mutable Records

- *Mutable* (updateable) state means that the *locations* of values becomes important.

```
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
```

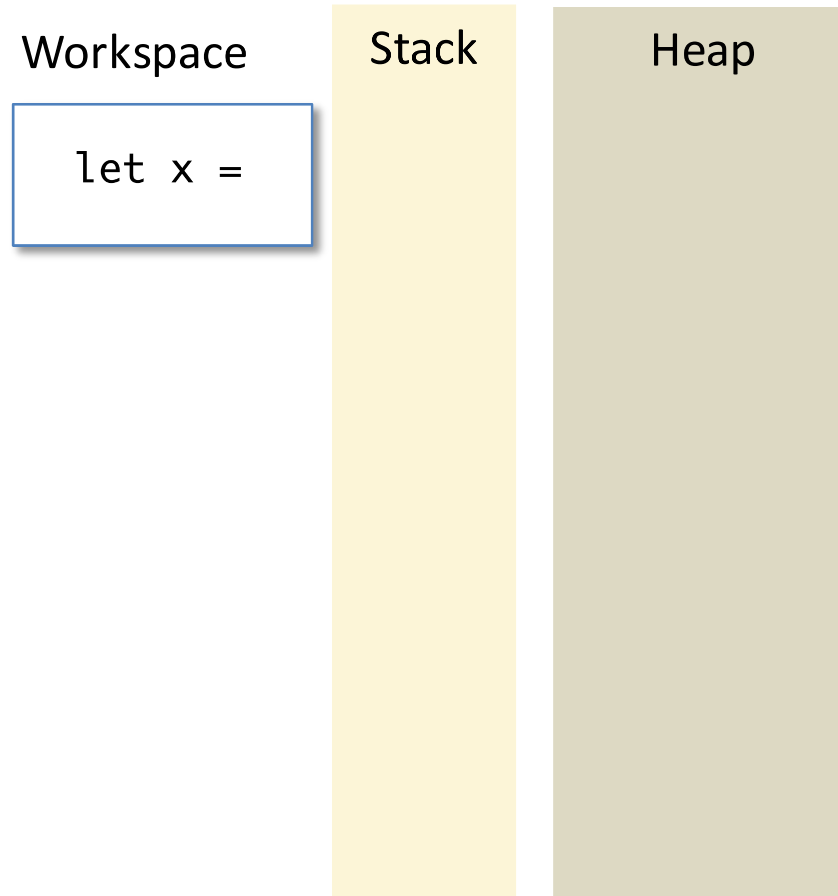
- The simple substitution model of program evaluation breaks down – it doesn't account for locations

- We need to refine our model of how to understand programs.

The Abstract Stack Machine

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



Abstract Stack Machine

The abstract stack machine operates by simplifying the expression in the workspace…

… but instead of substitution, it records the values of variables on the stack

… values themselves are divided into primitive values (also on the stack) and reference values (on the heap).

For immutable structures, this model is just a complicated way of doing substitution

… but we need the extra complexity to understand mutable state.

We'll go through examples here, read Chapter 14 of the lecture notes for general rules

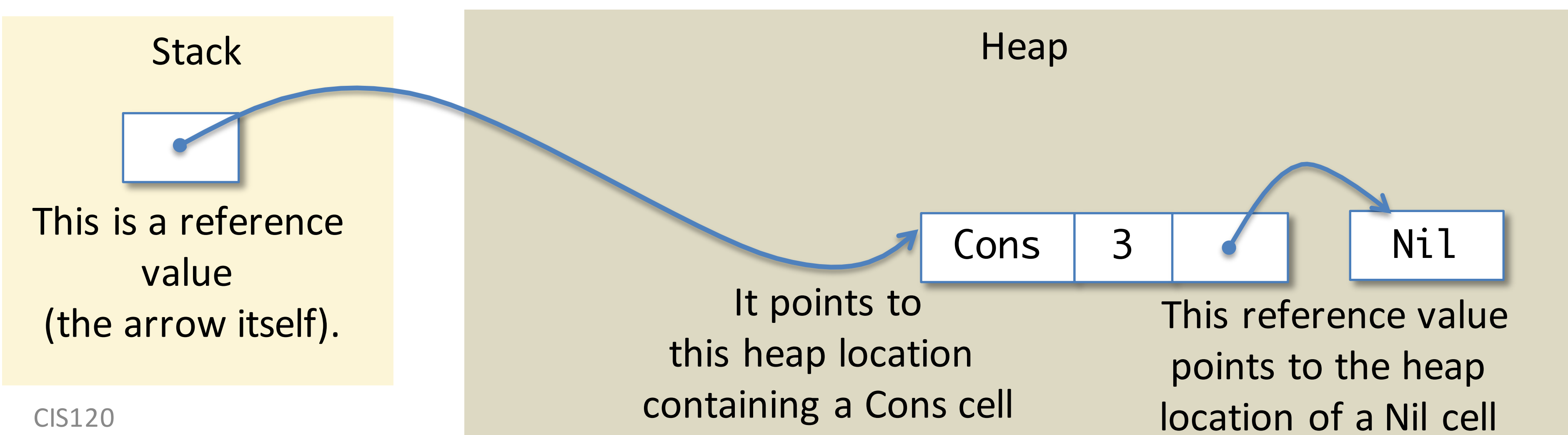
Values and References

A *value* is either:

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

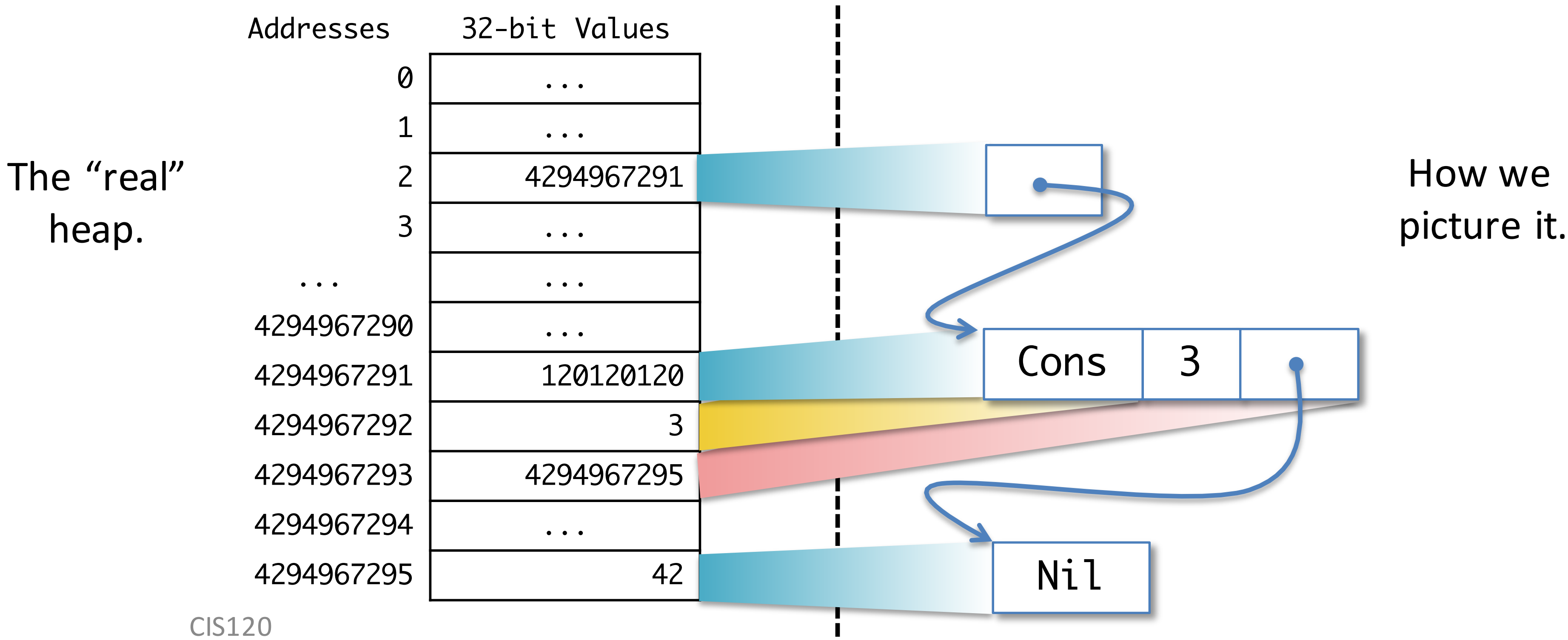
A reference is the *address* of a piece of data in the heap. We draw a reference value as an "arrow":

- 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 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

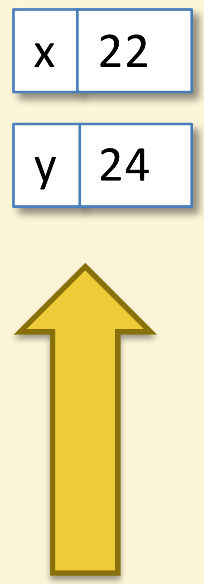
Heap

Simplification

Workspace

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

Stack



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
```

1\. Stack

z	22
w	2 + z

2\. Stack

z	20
w	22

3\. Stack

w	22

4\. Stack

w	22
z	20

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

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

1. Stack

z	22
z	20

2. Stack

z	20
z	22

3. Stack

z	22

4. Stack

z	22
z	22

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

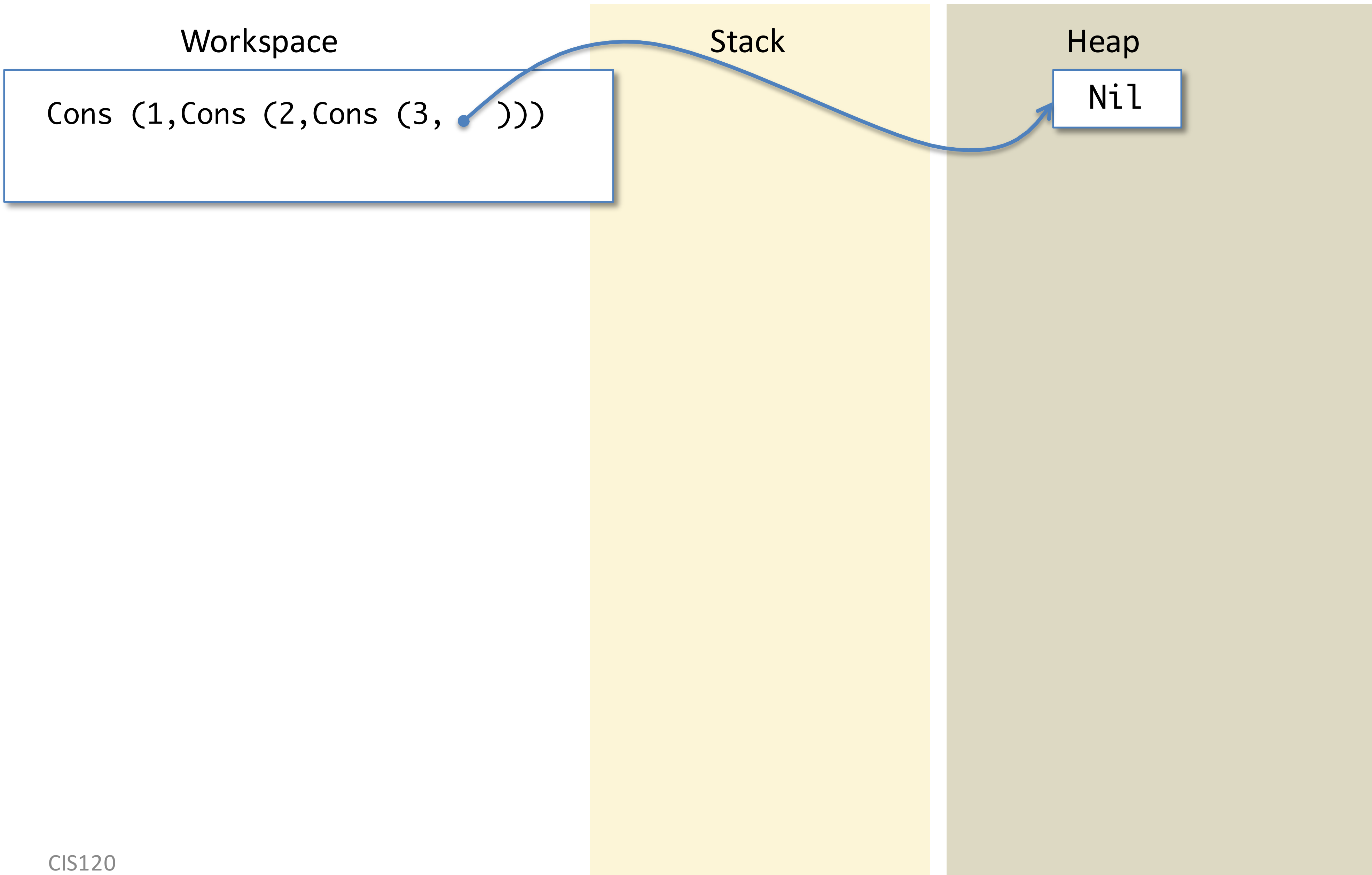
Workspace

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

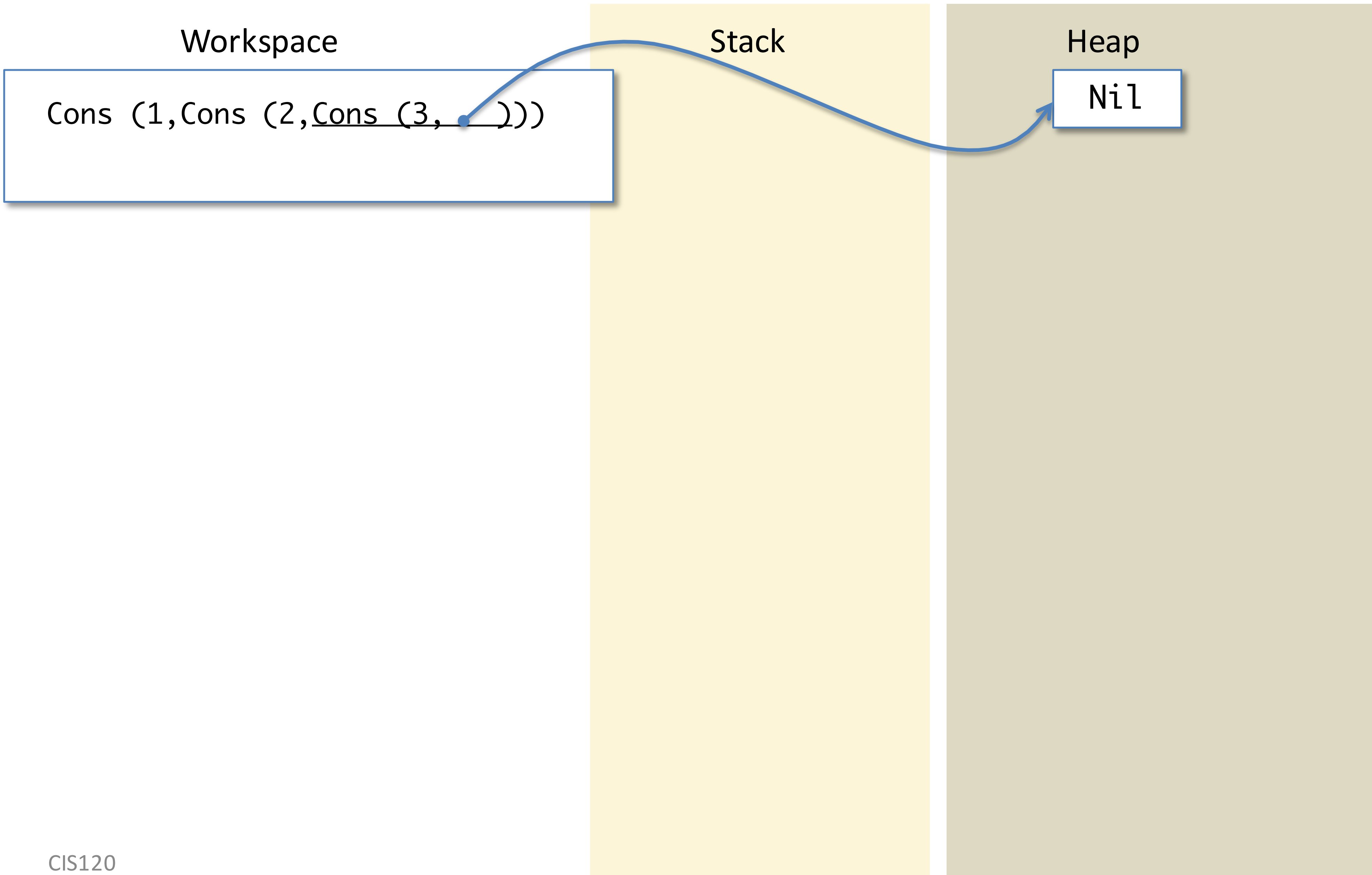
Stack

Heap

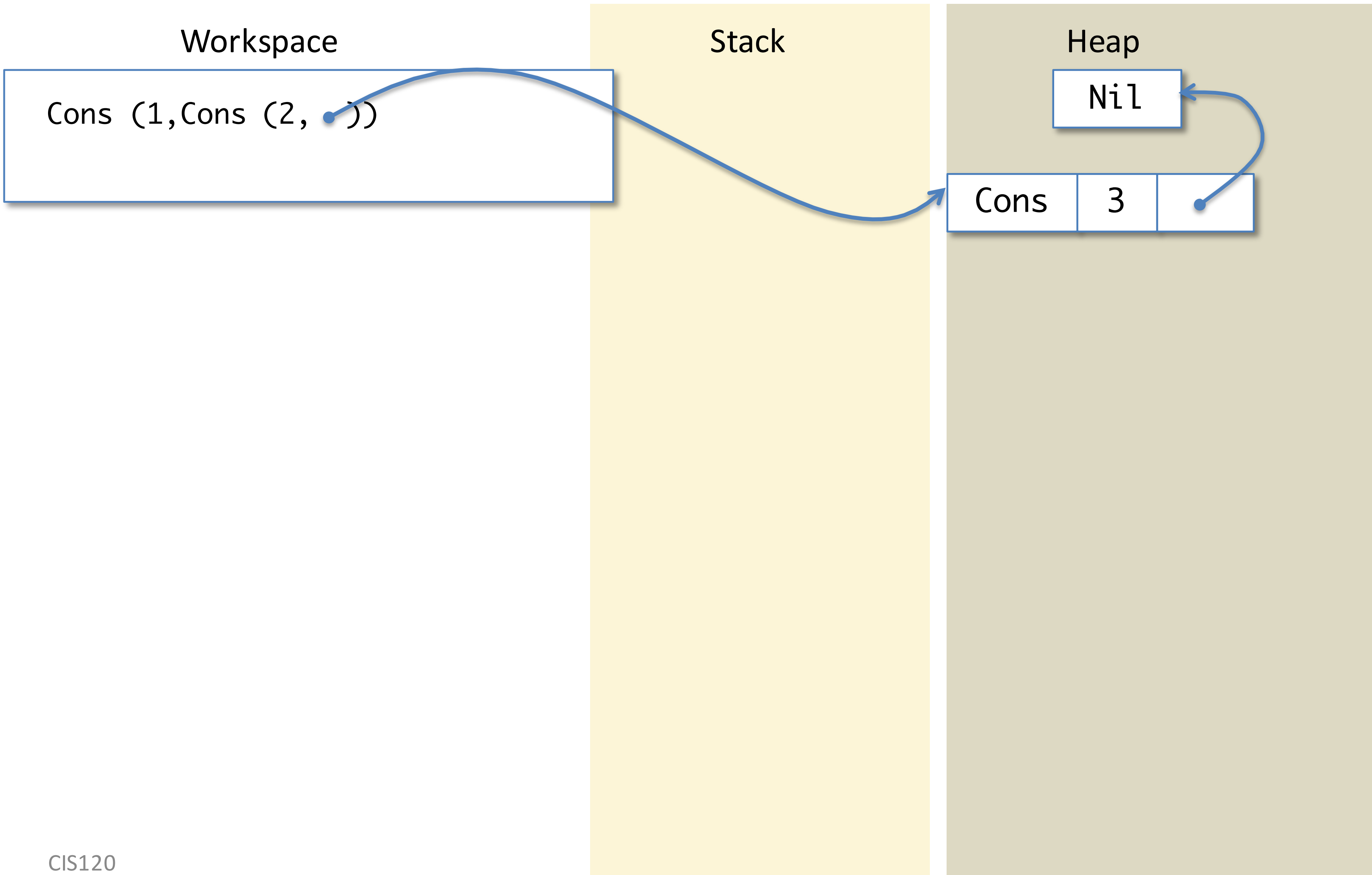
Simplification



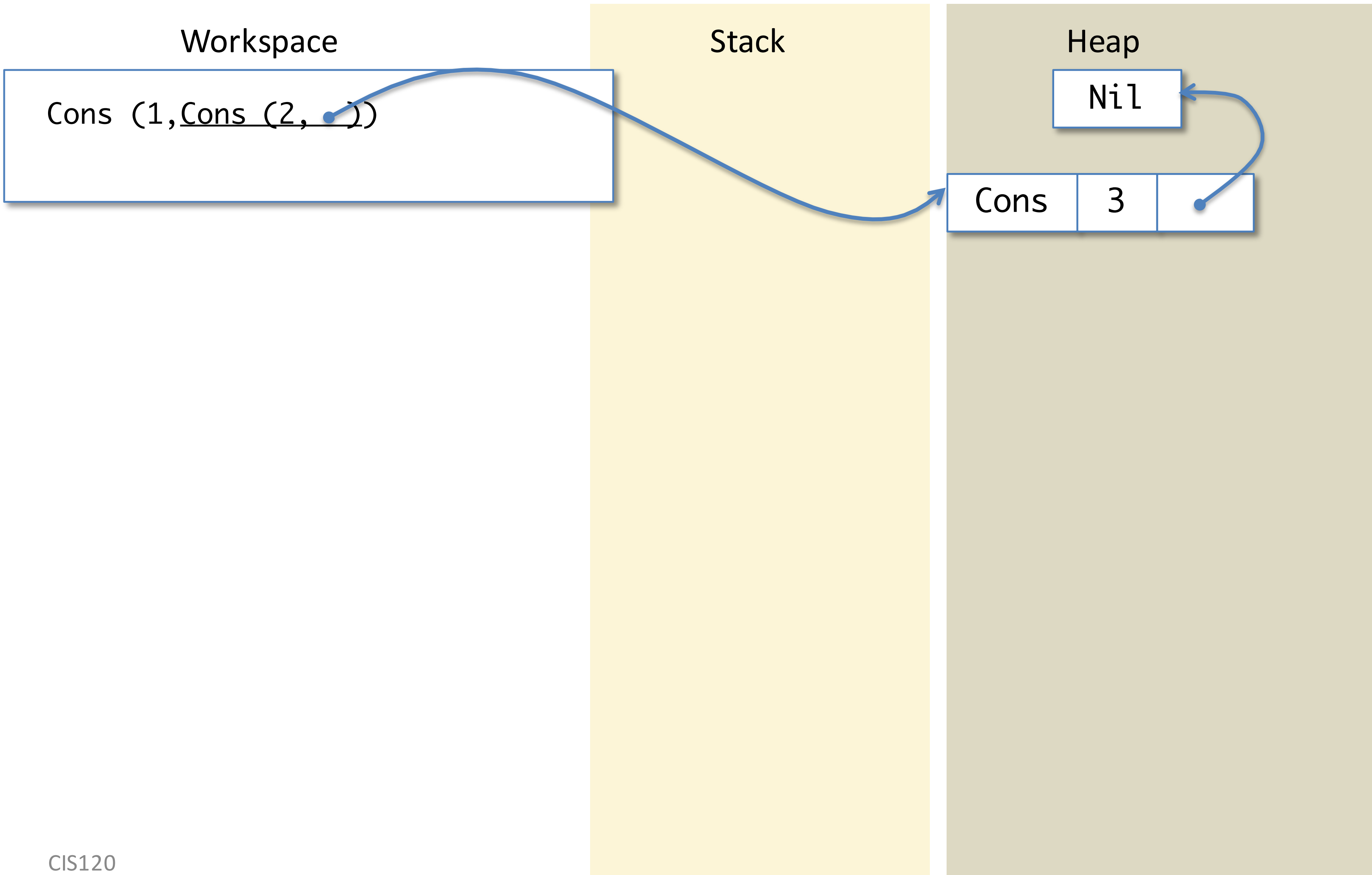
Simplification



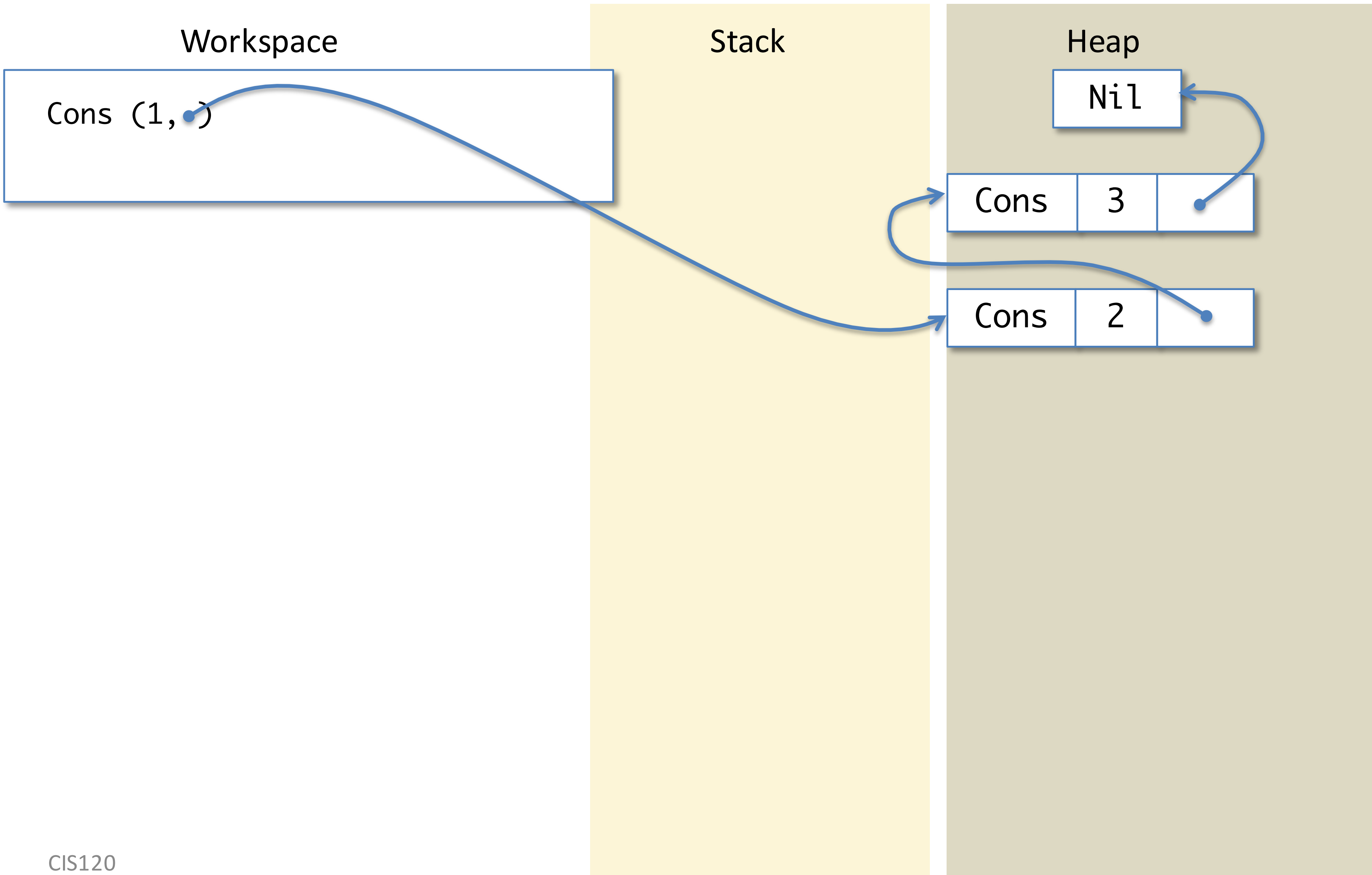
Simplification



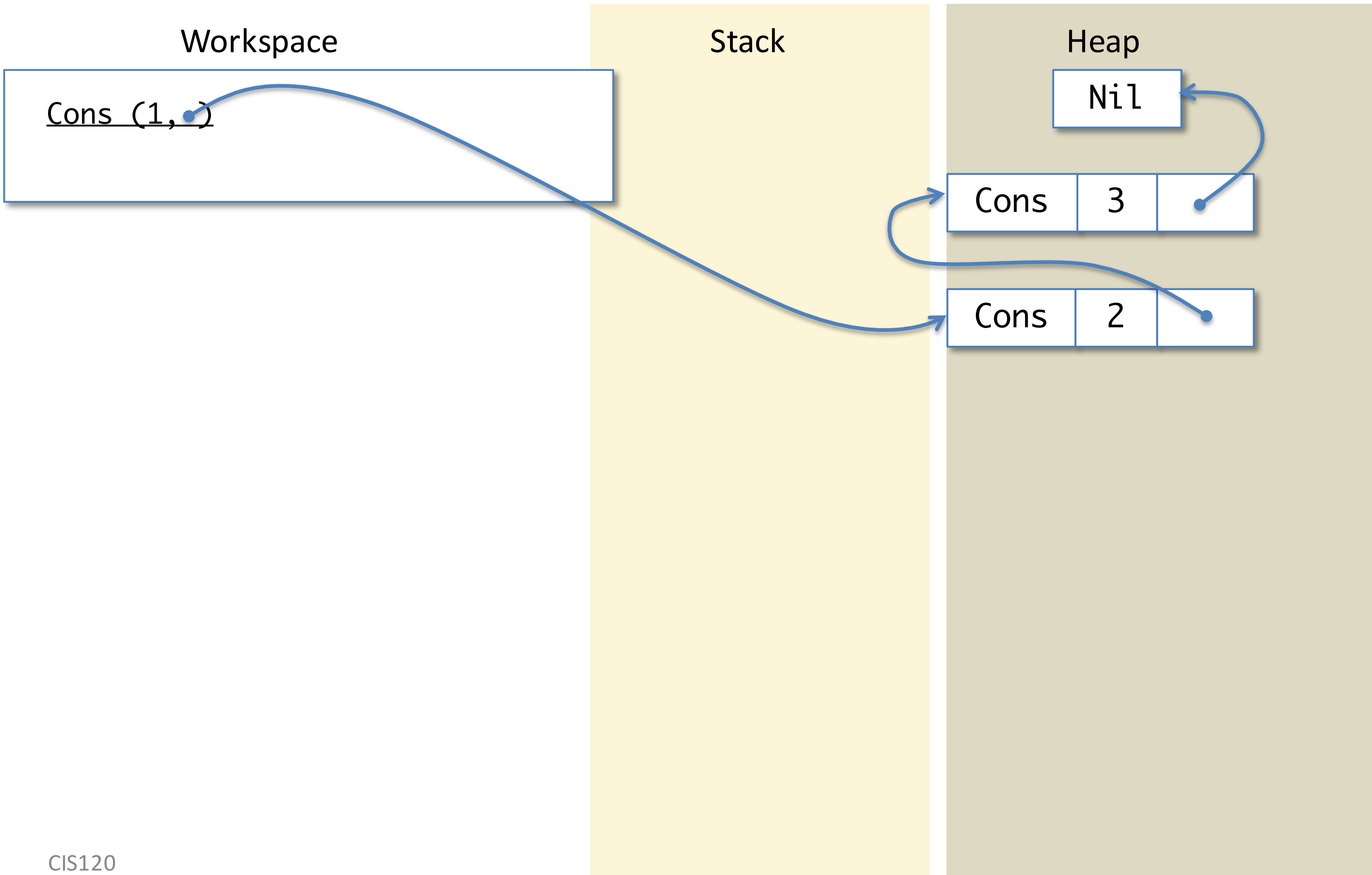
Simplification



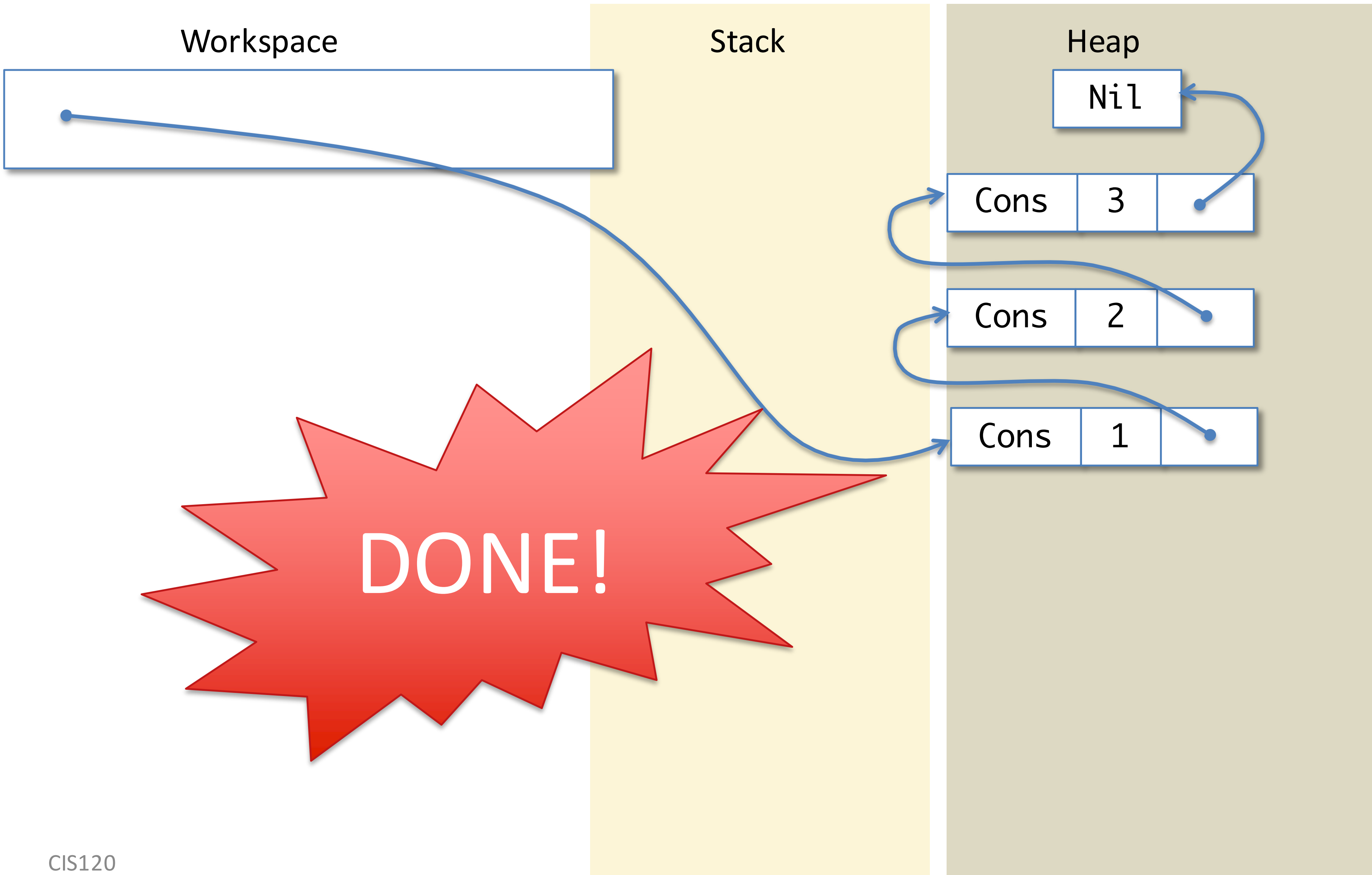
Simplification



Simplification



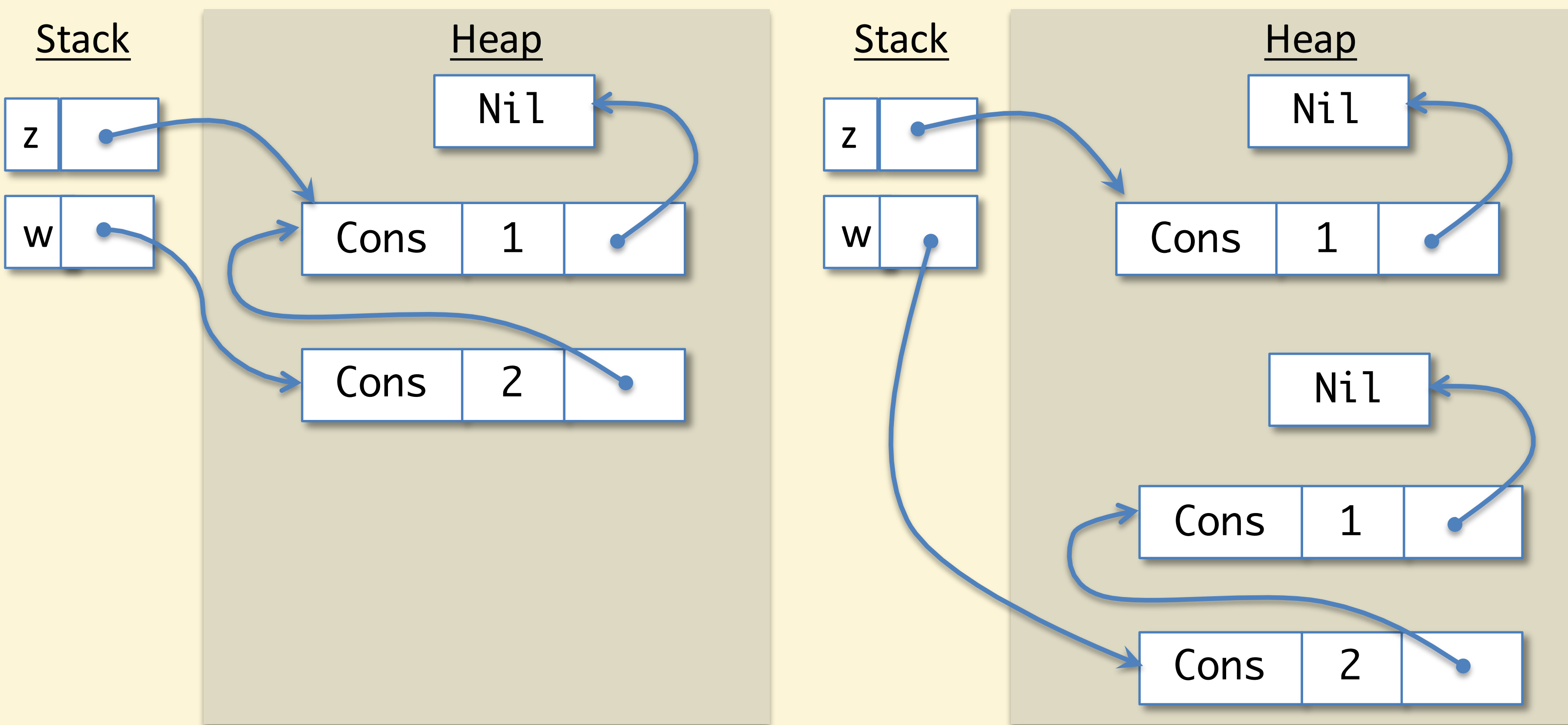
Simplification



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

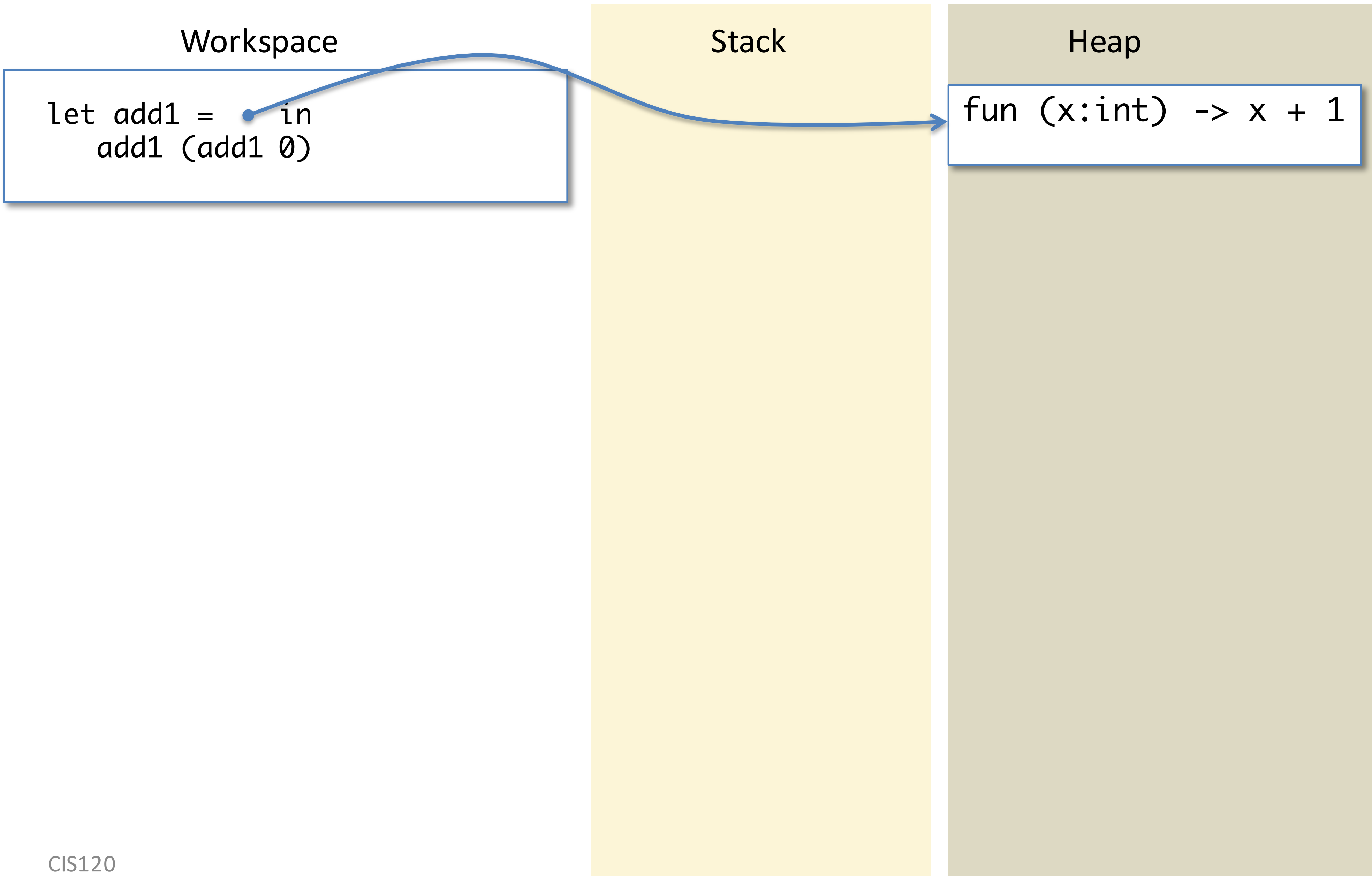
Workspace

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

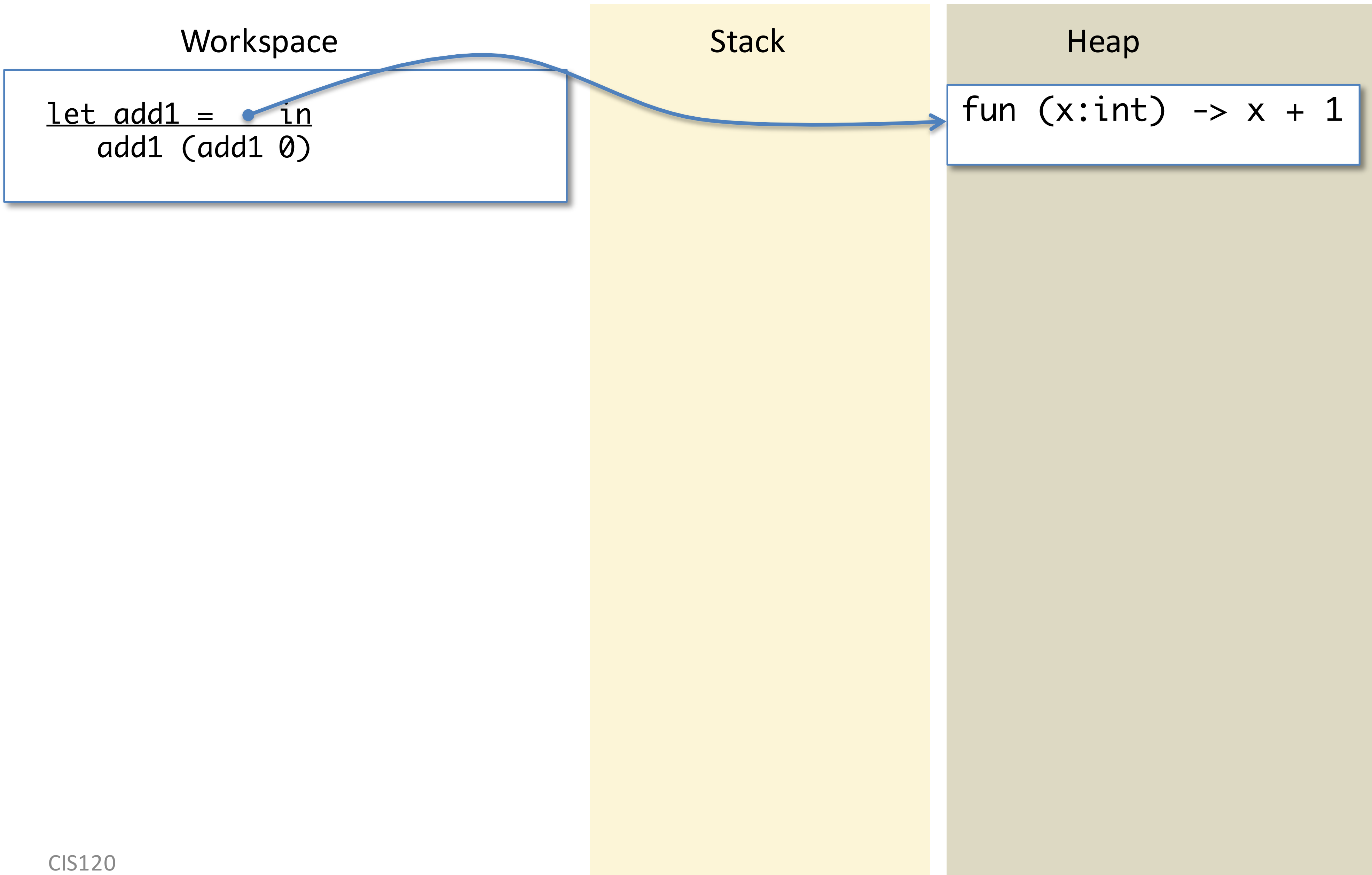
Stack

Heap

Function Simplification



Function Simplification



Function Simplification

Workspace

```
add1 (add1 0)
```

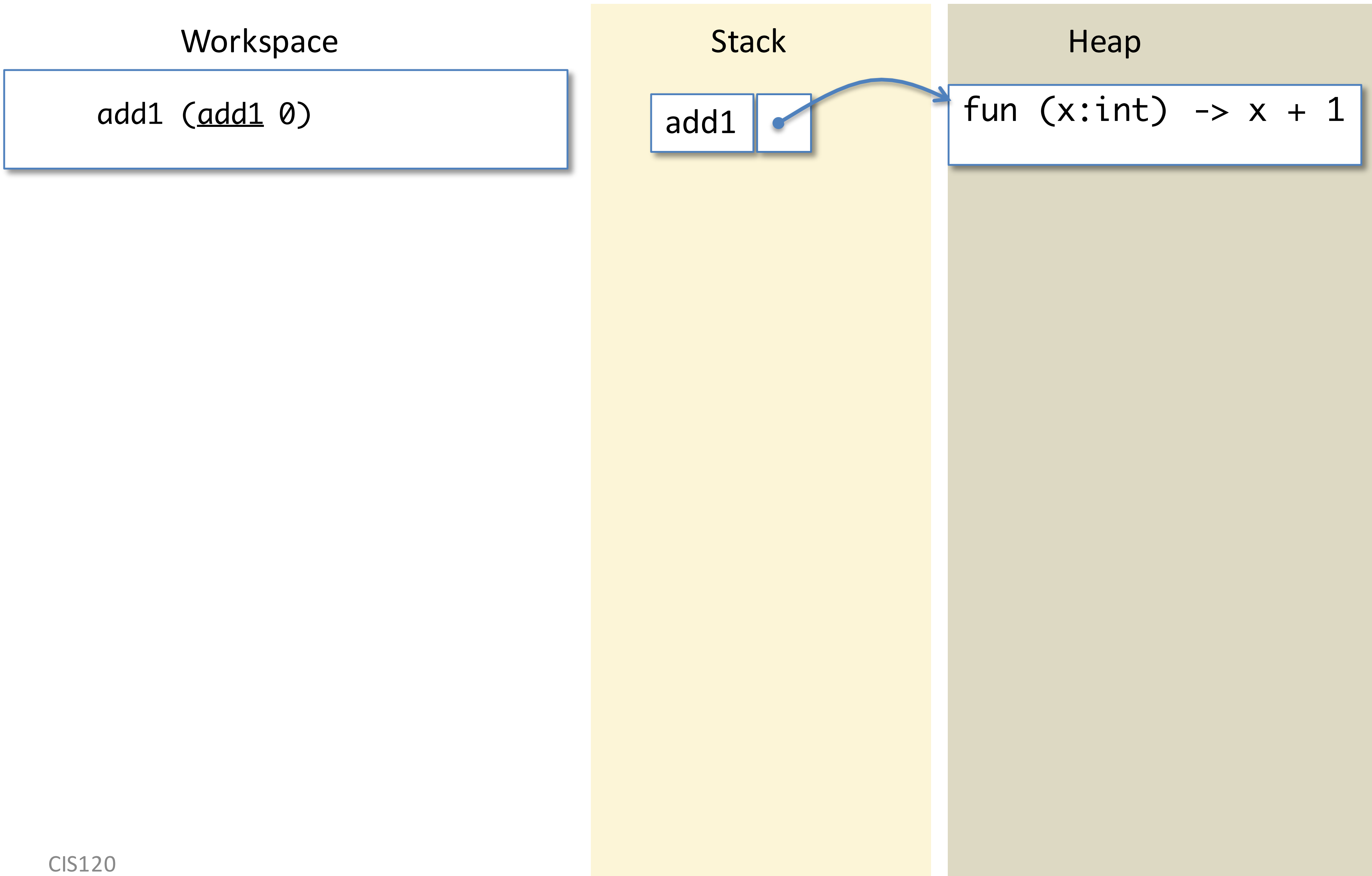
Stack

```
add1 | •
```

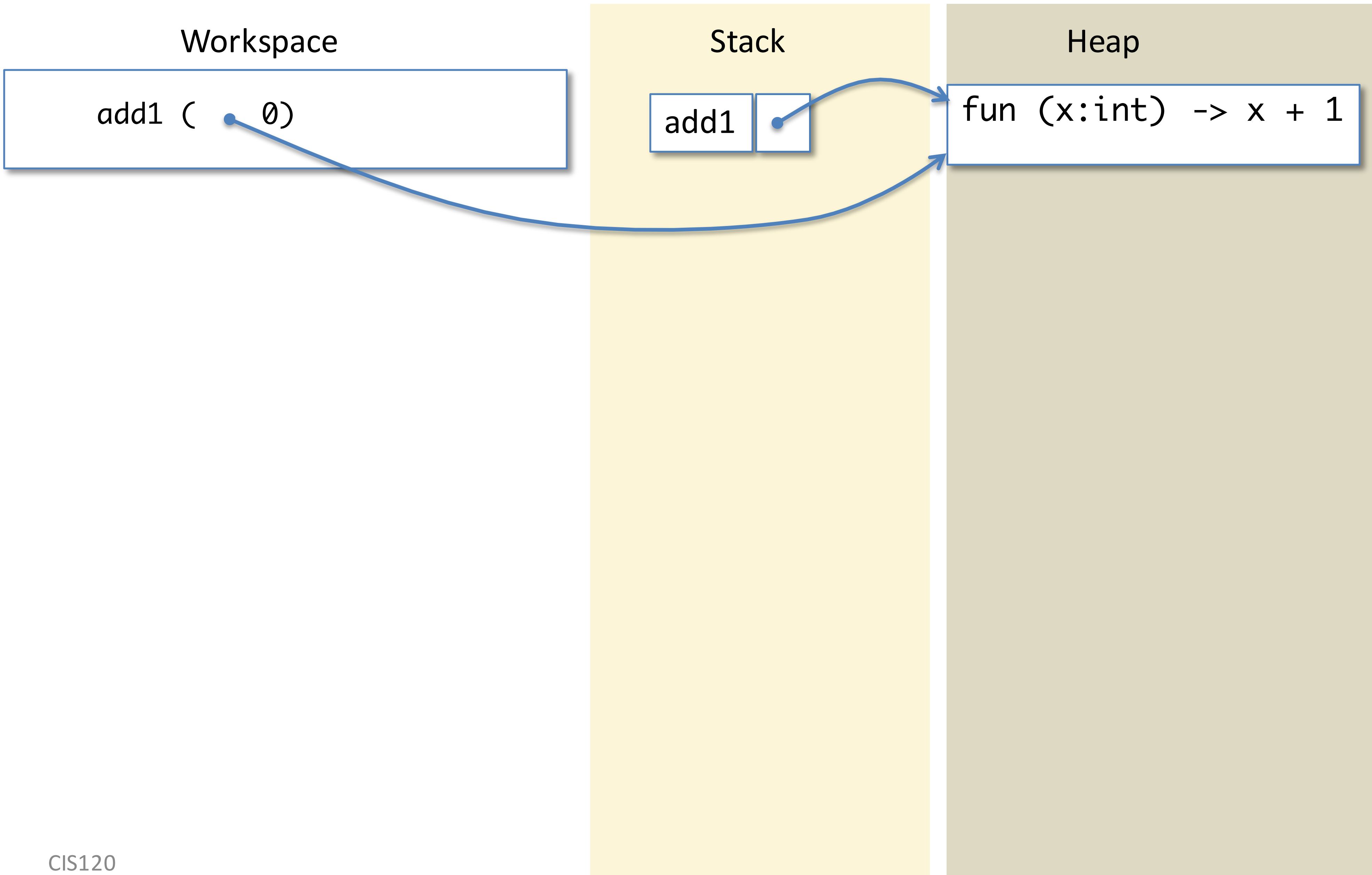
Heap

```
fun (x:int) -> x + 1
```

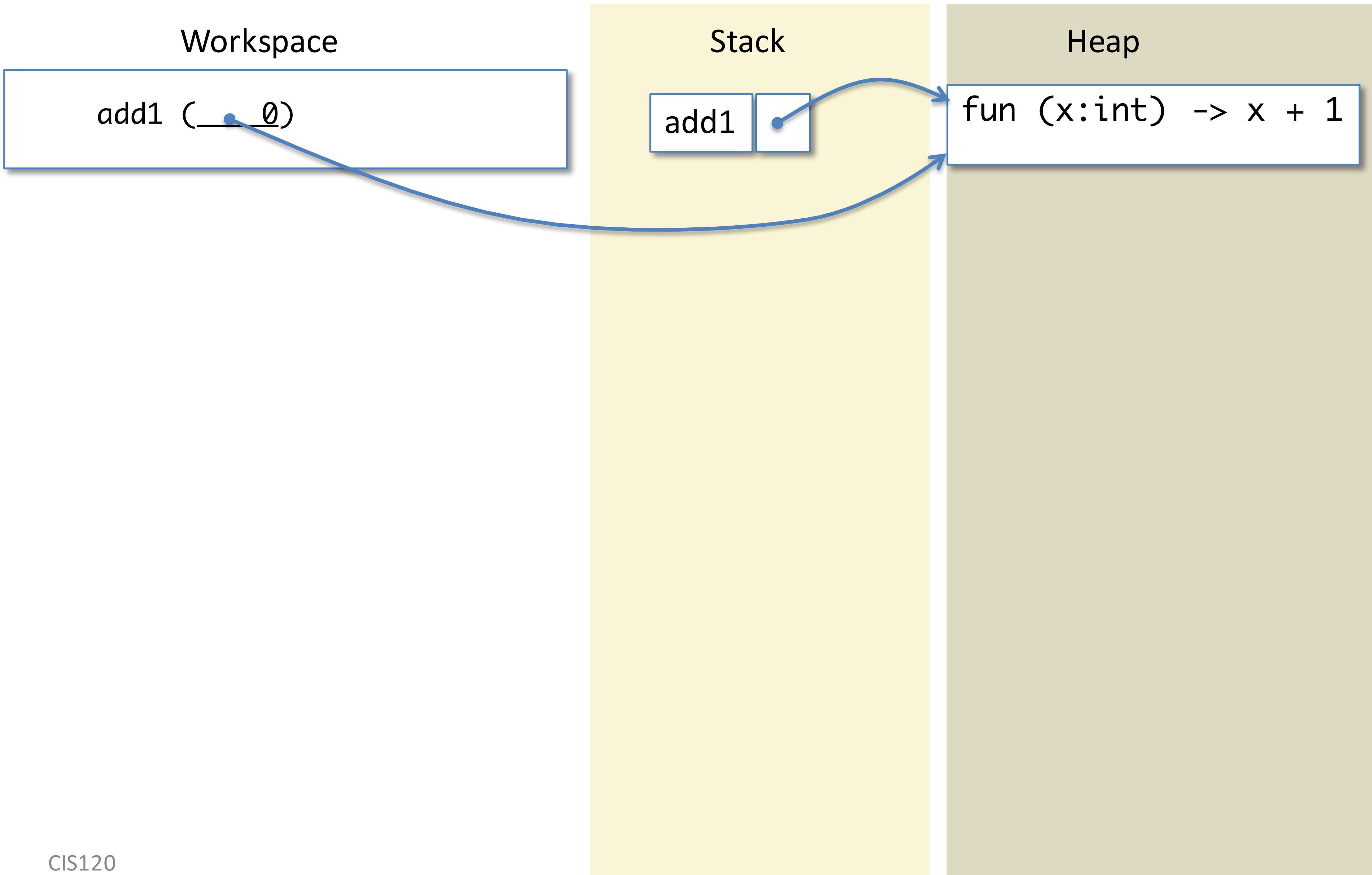
Function Simplification



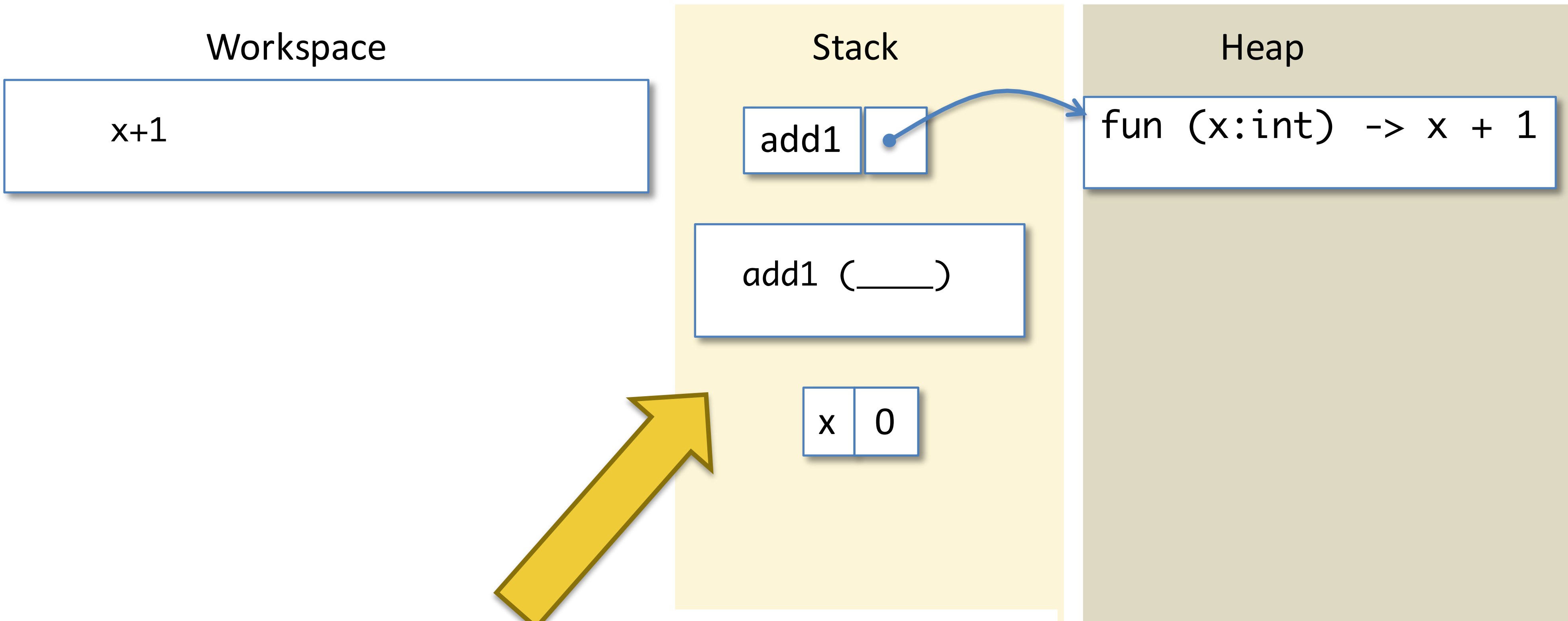
Function Simplification



Function Simplification



Do the Call, Saving the Workspace

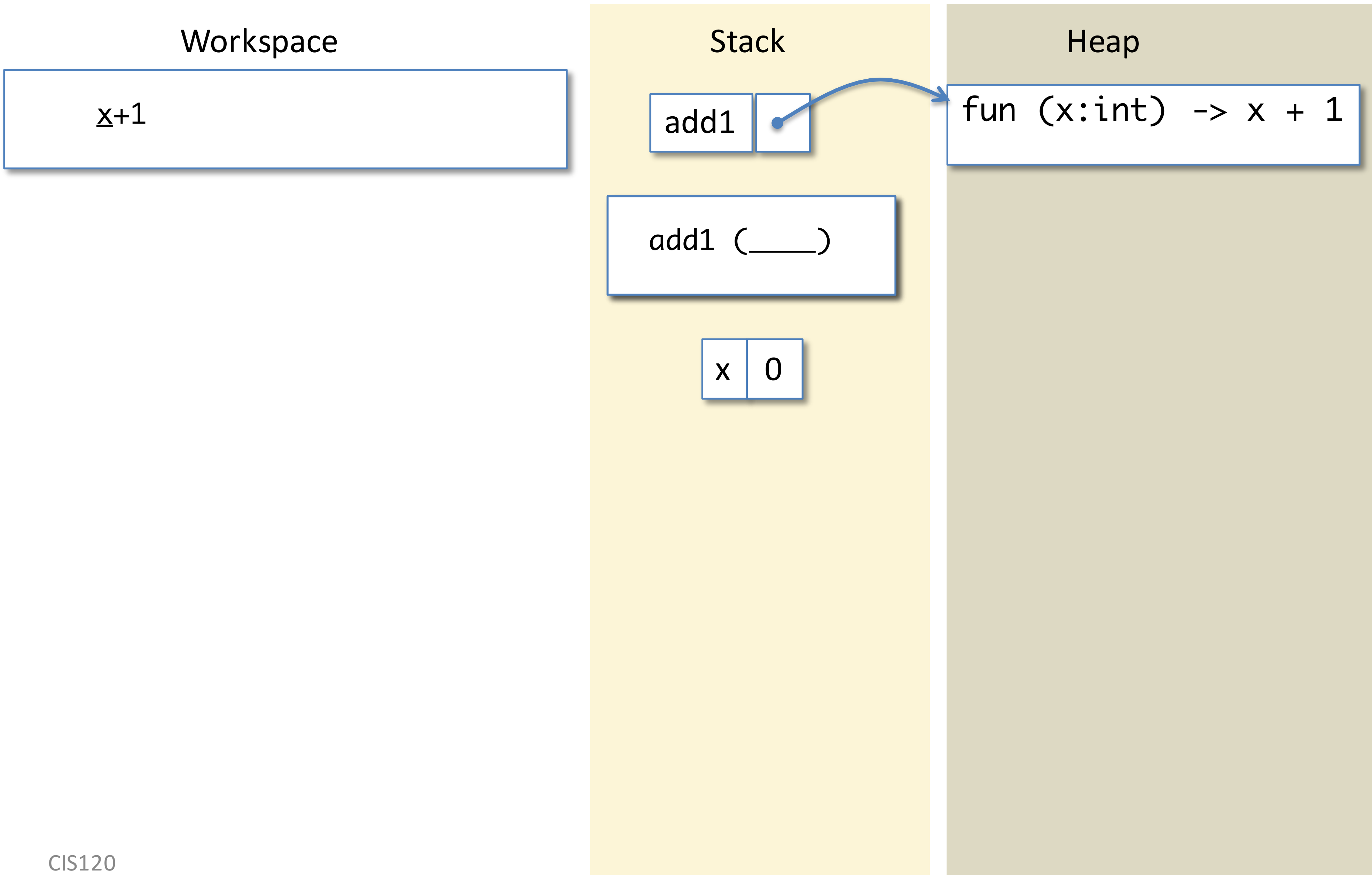


Note the saved workspace and pushed function argument.

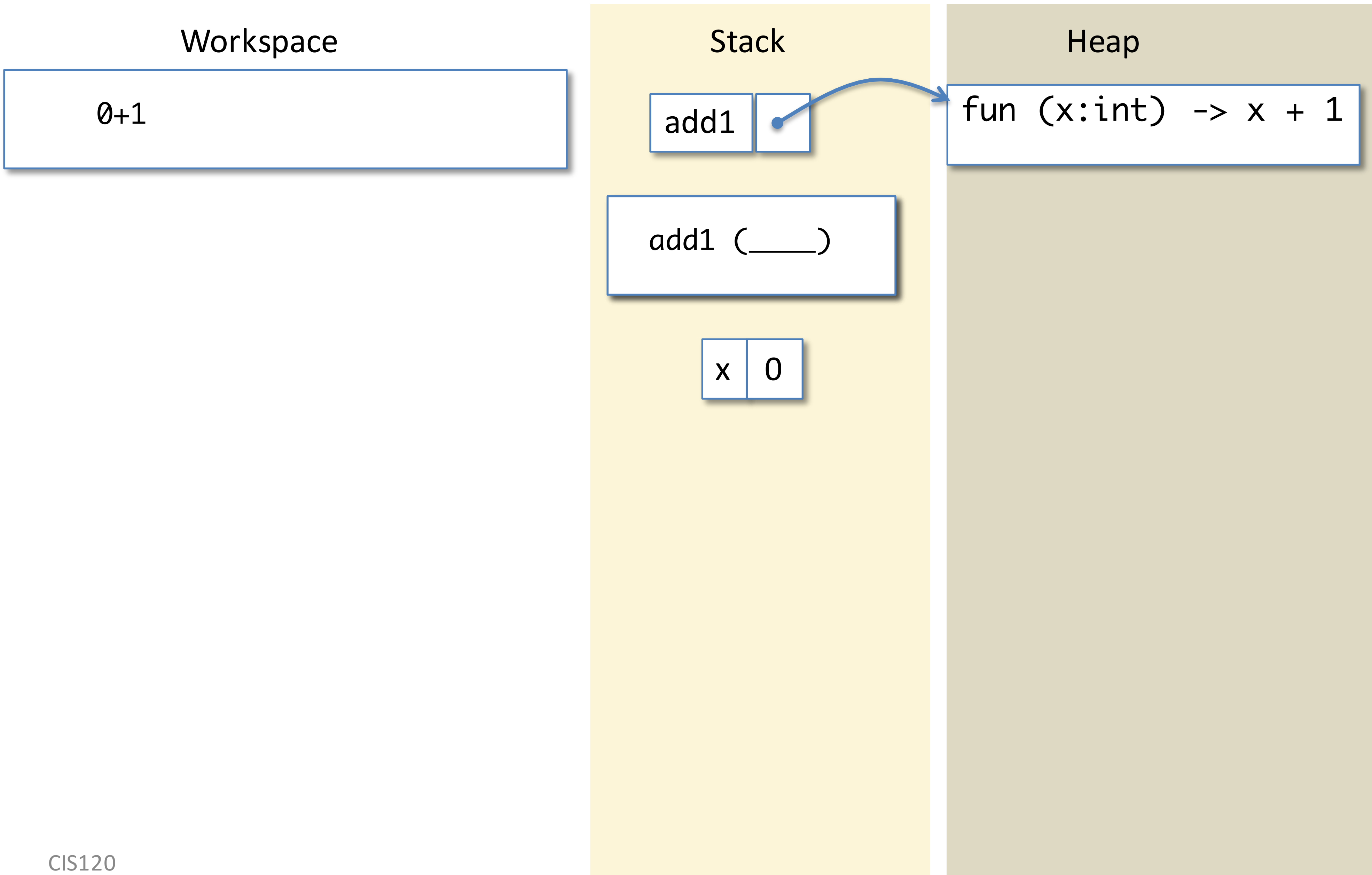
- compare with the workspace on the previous slide.
- the name 'x' comes from the name in the heap

The new workspace is the *body* of the function

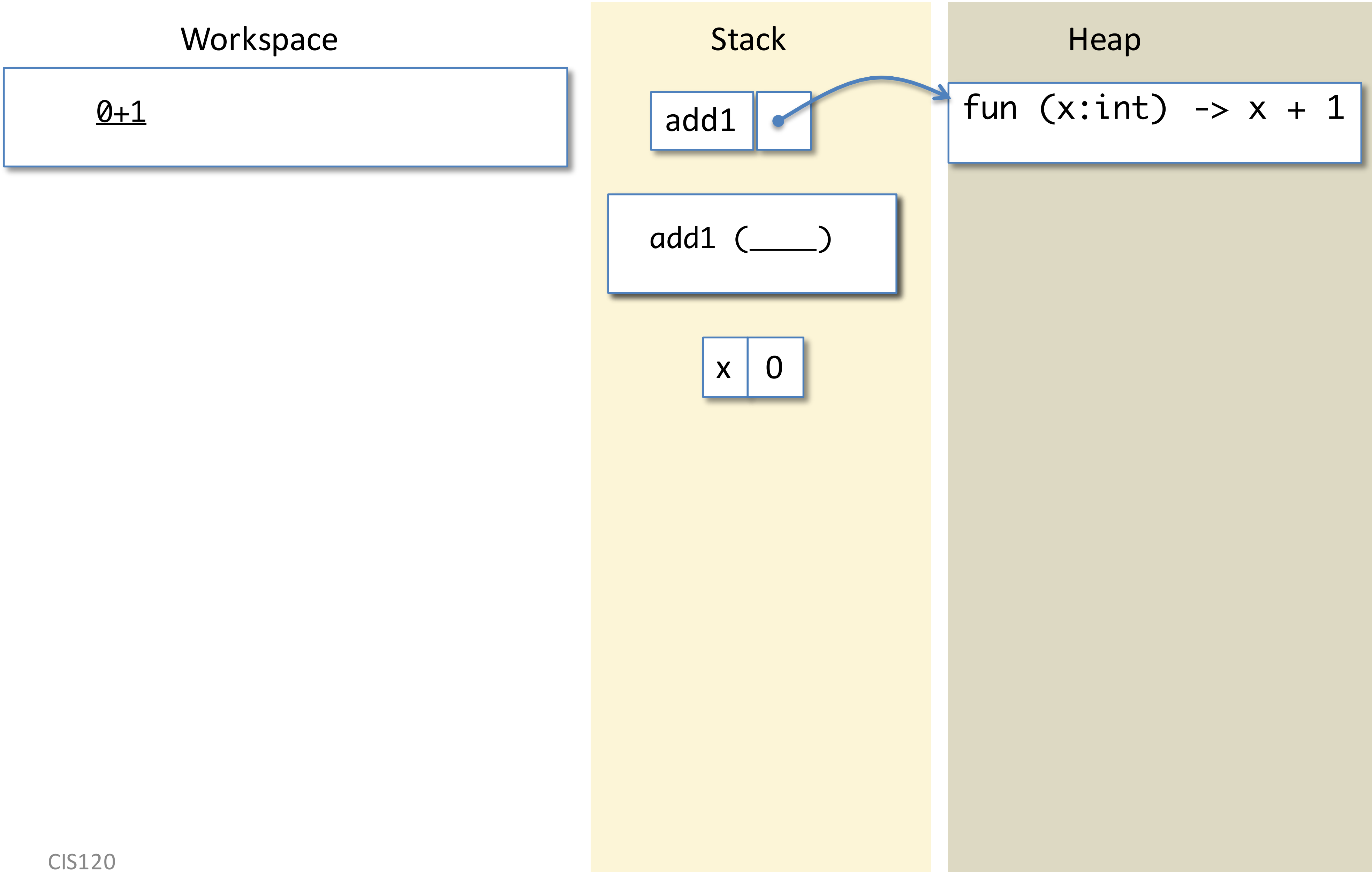
Function Simplification



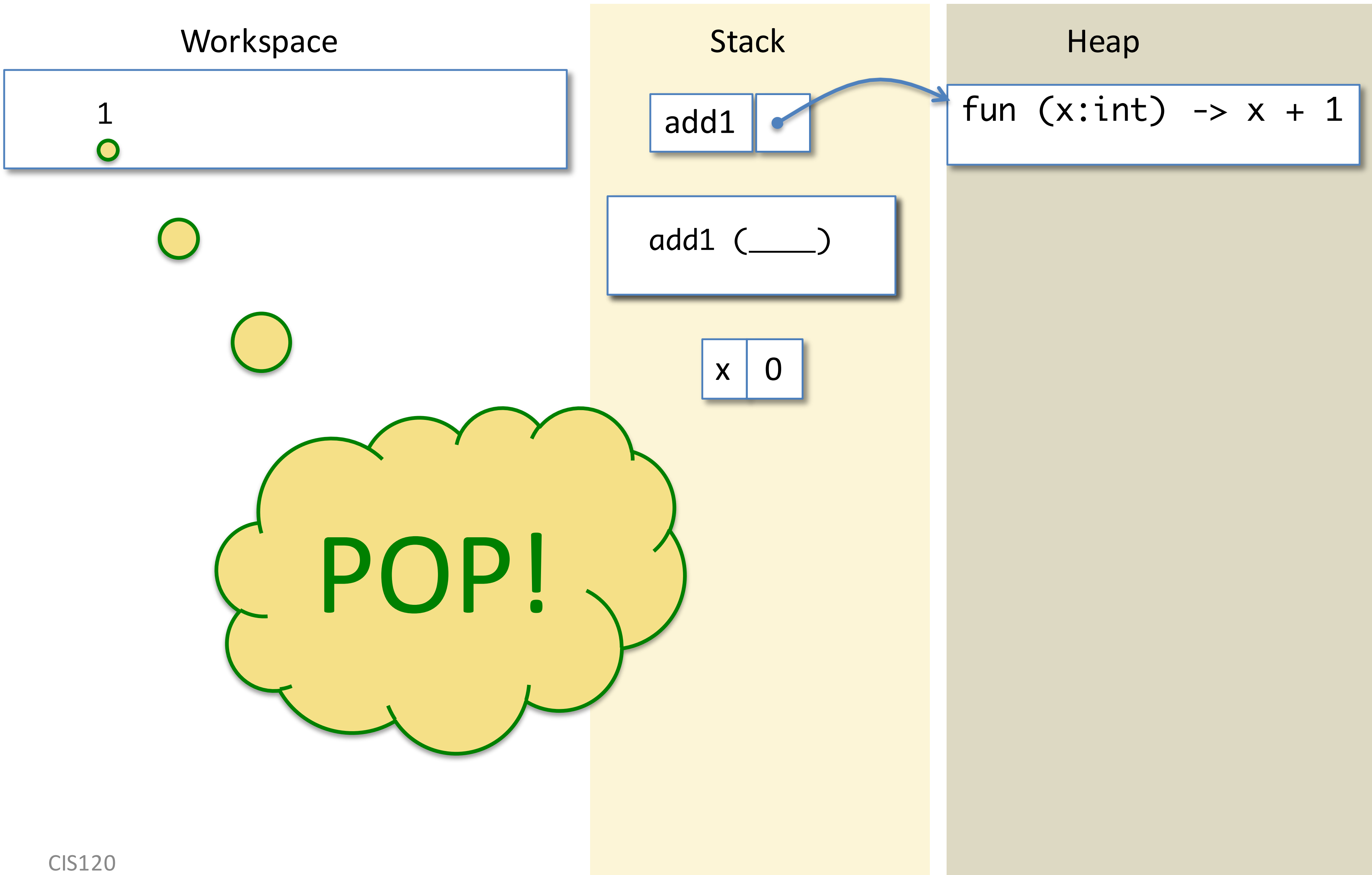
Function Simplification



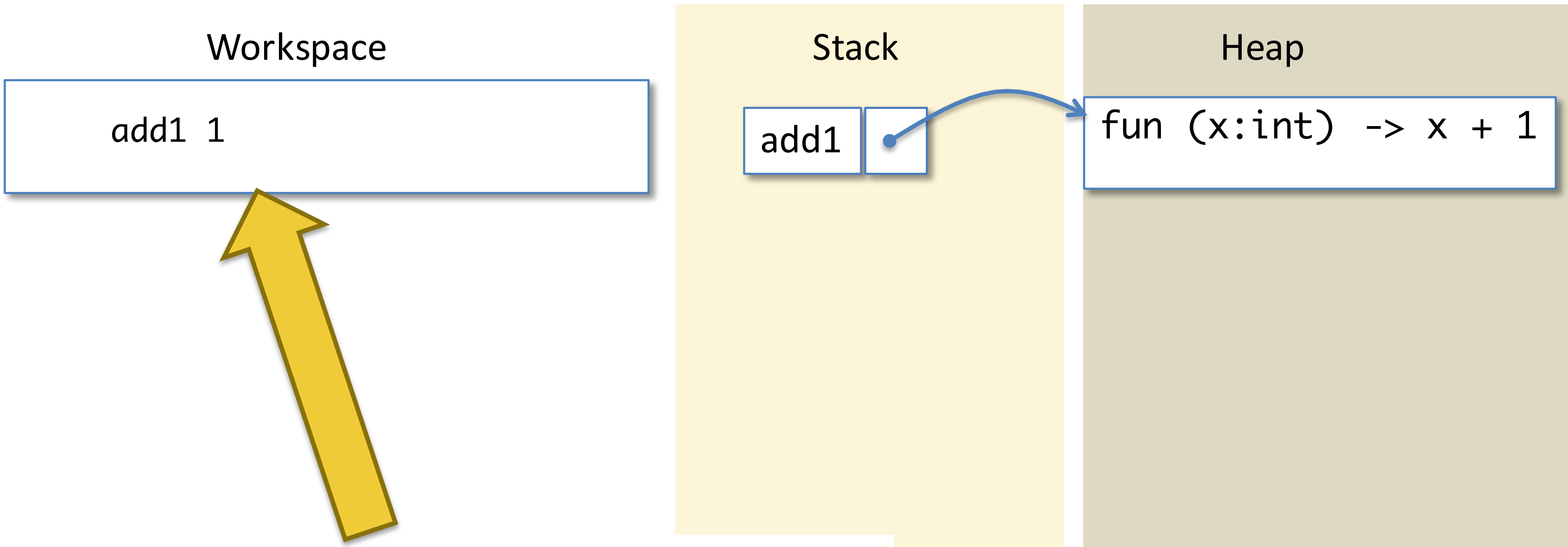
Function Simplification



Function Simplification

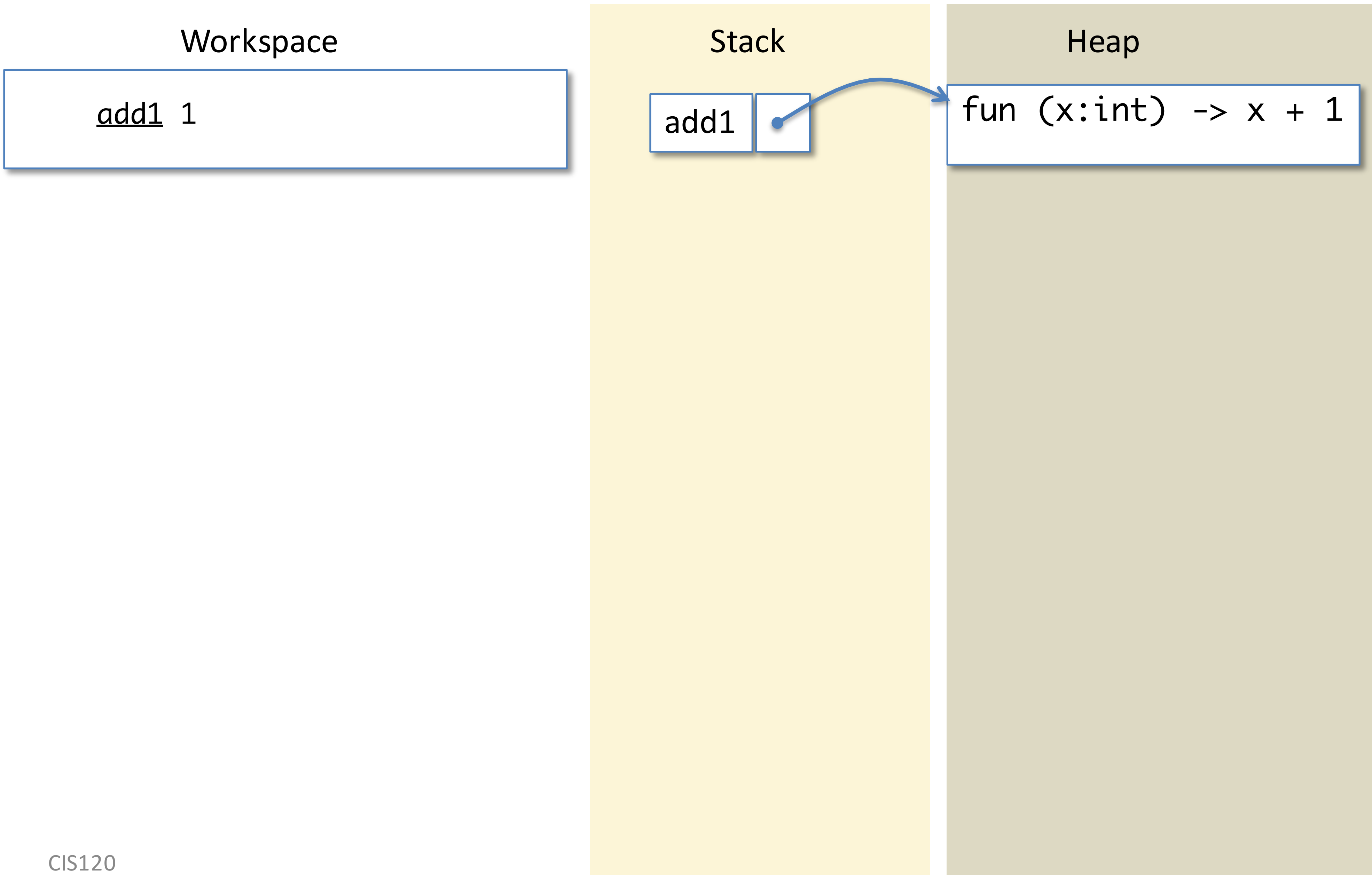


Function Simplification

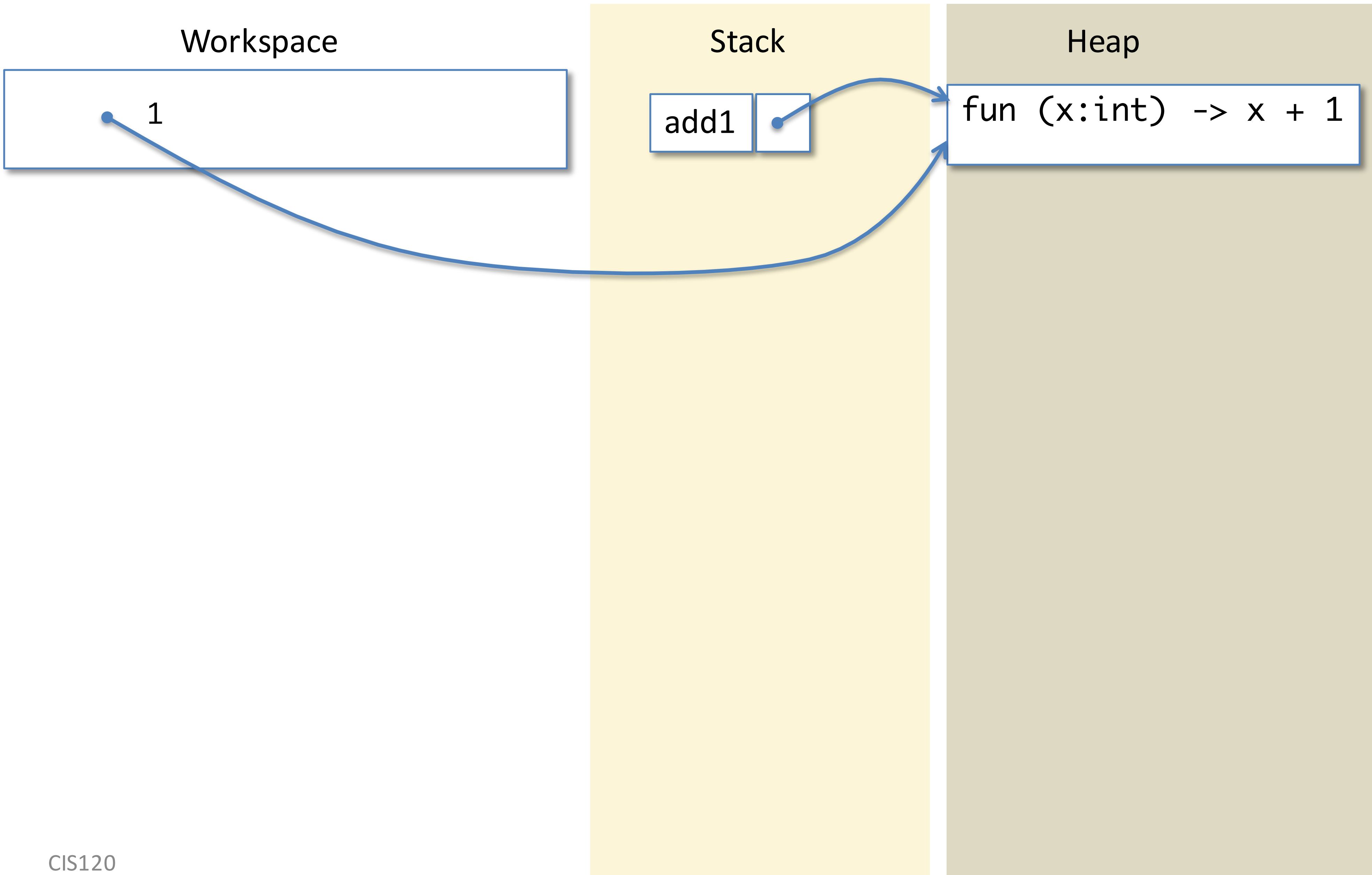


See how the ASM *restored* the saved workspace, replacing its `hole' with the value computed into the old workspace. (Compare with previous slide.)

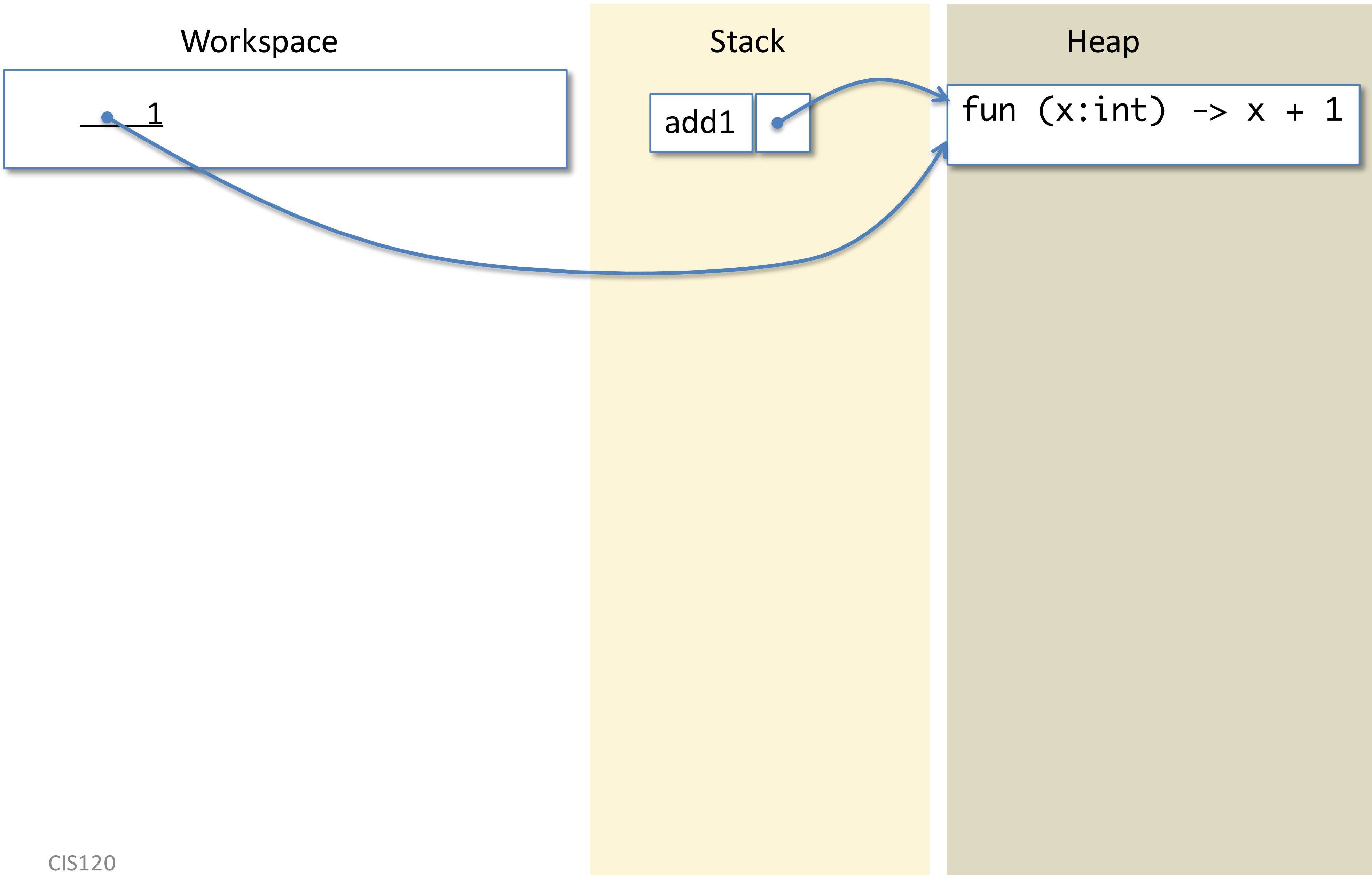
Function Simplification



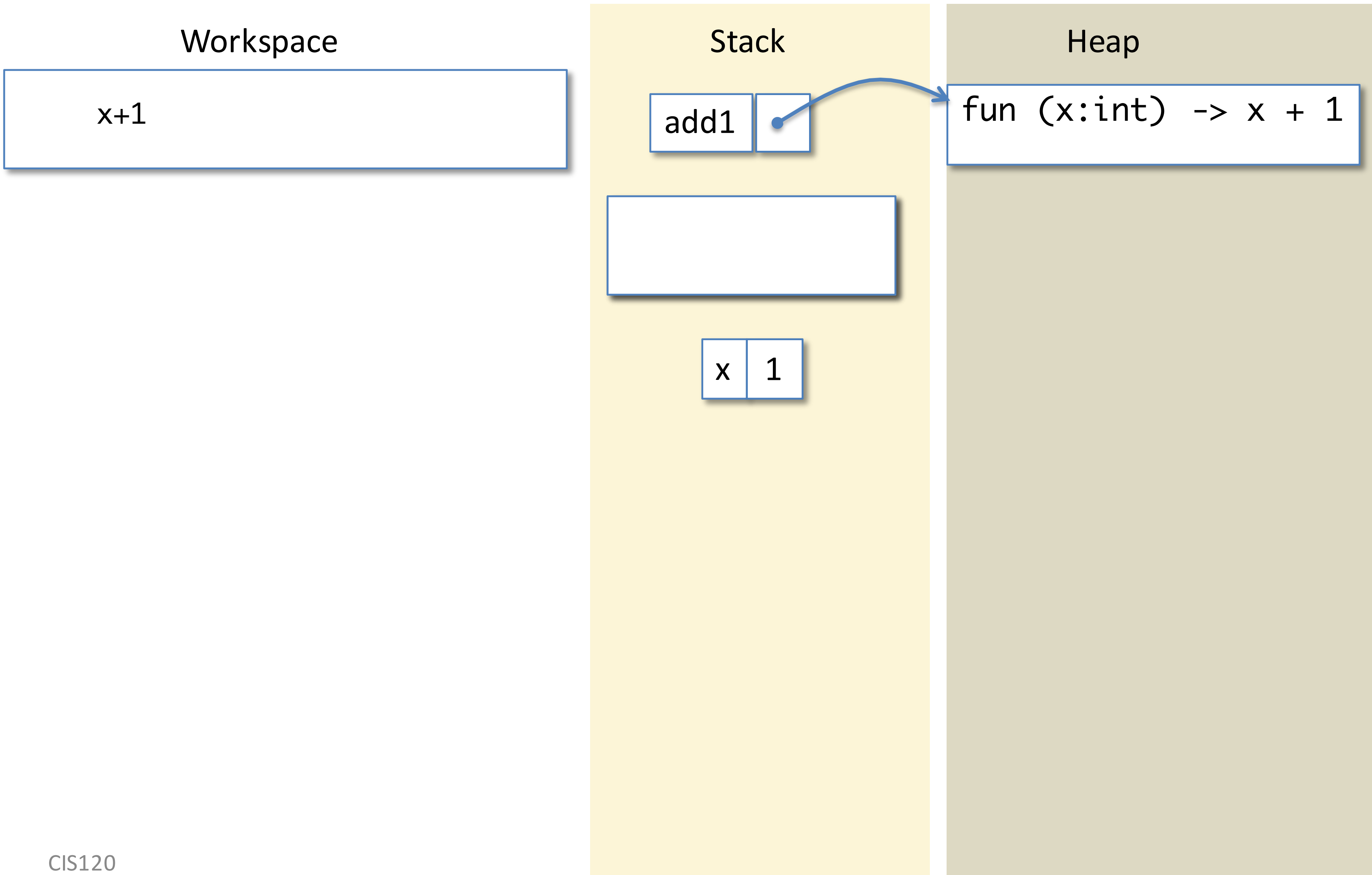
Function Simplification



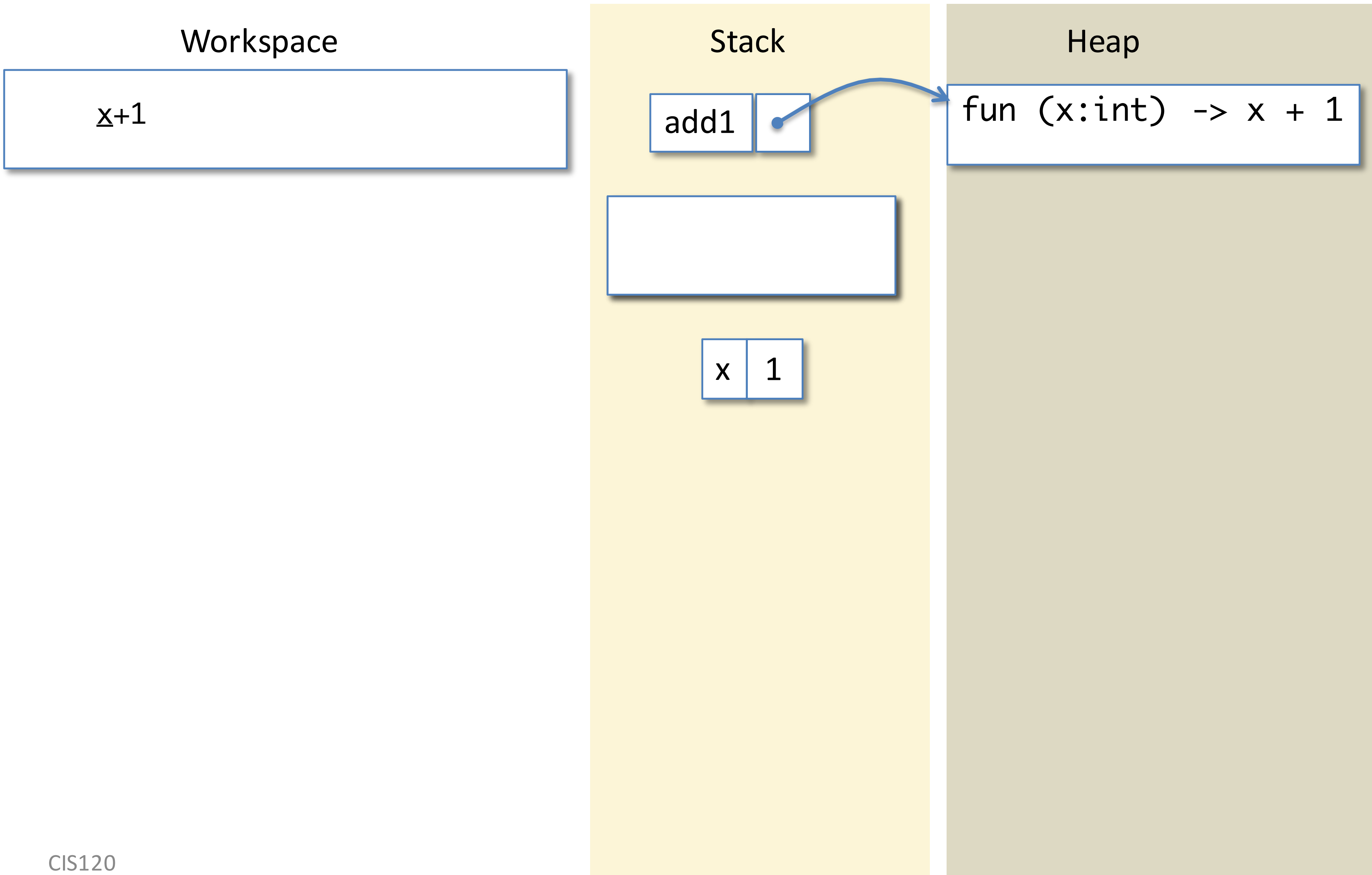
Function Simplification



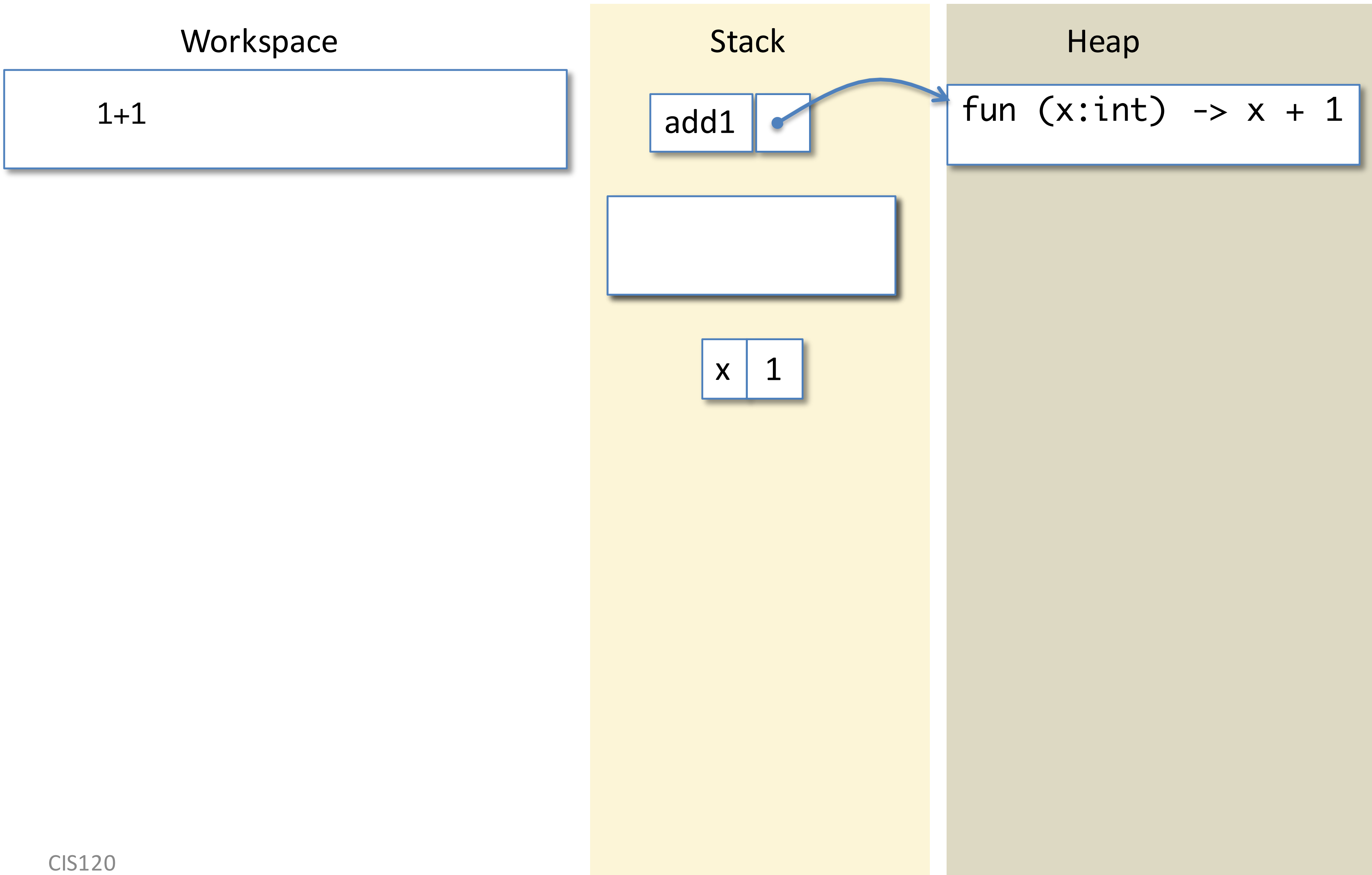
Function Simplification



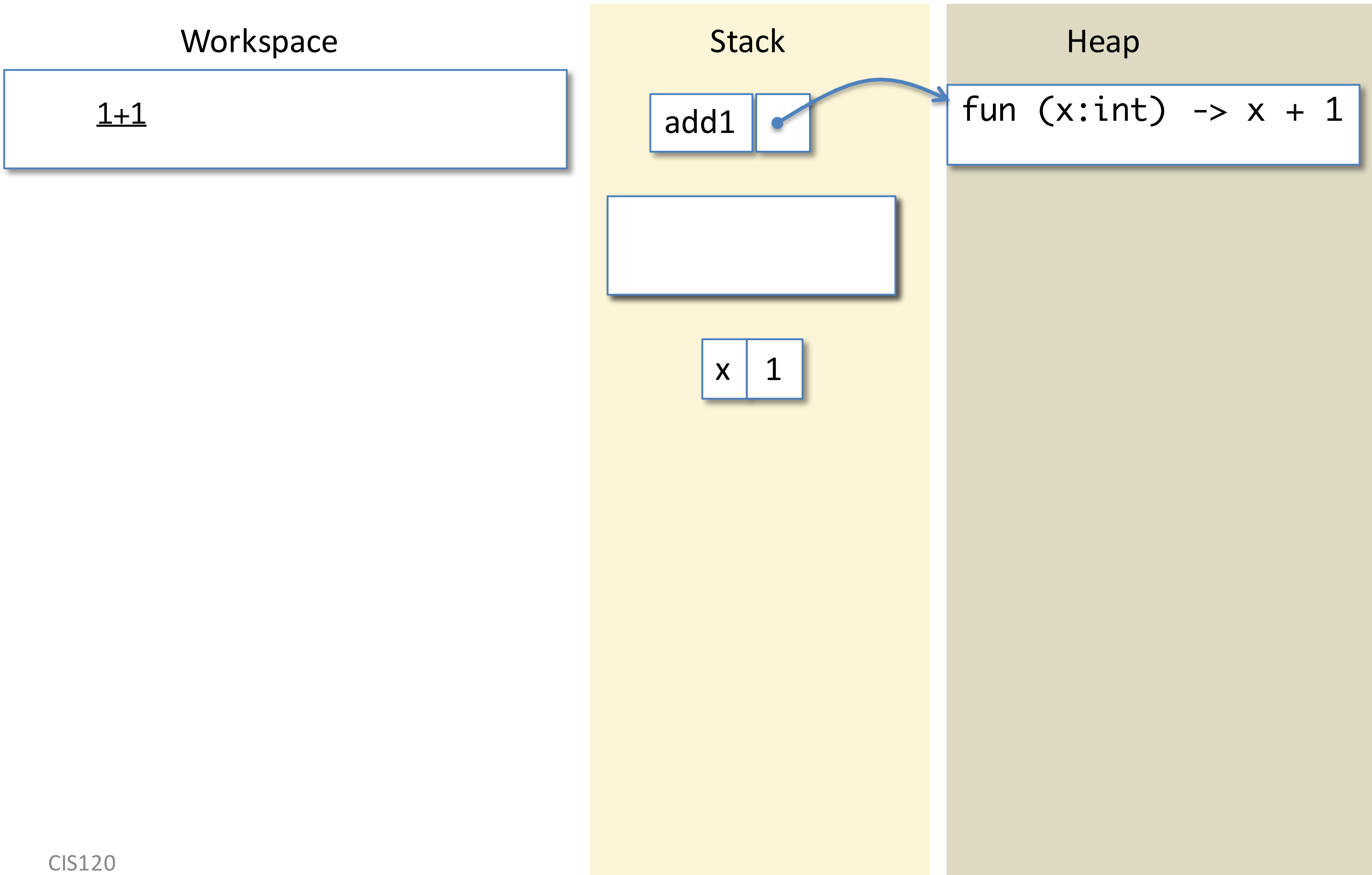
Function Simplification



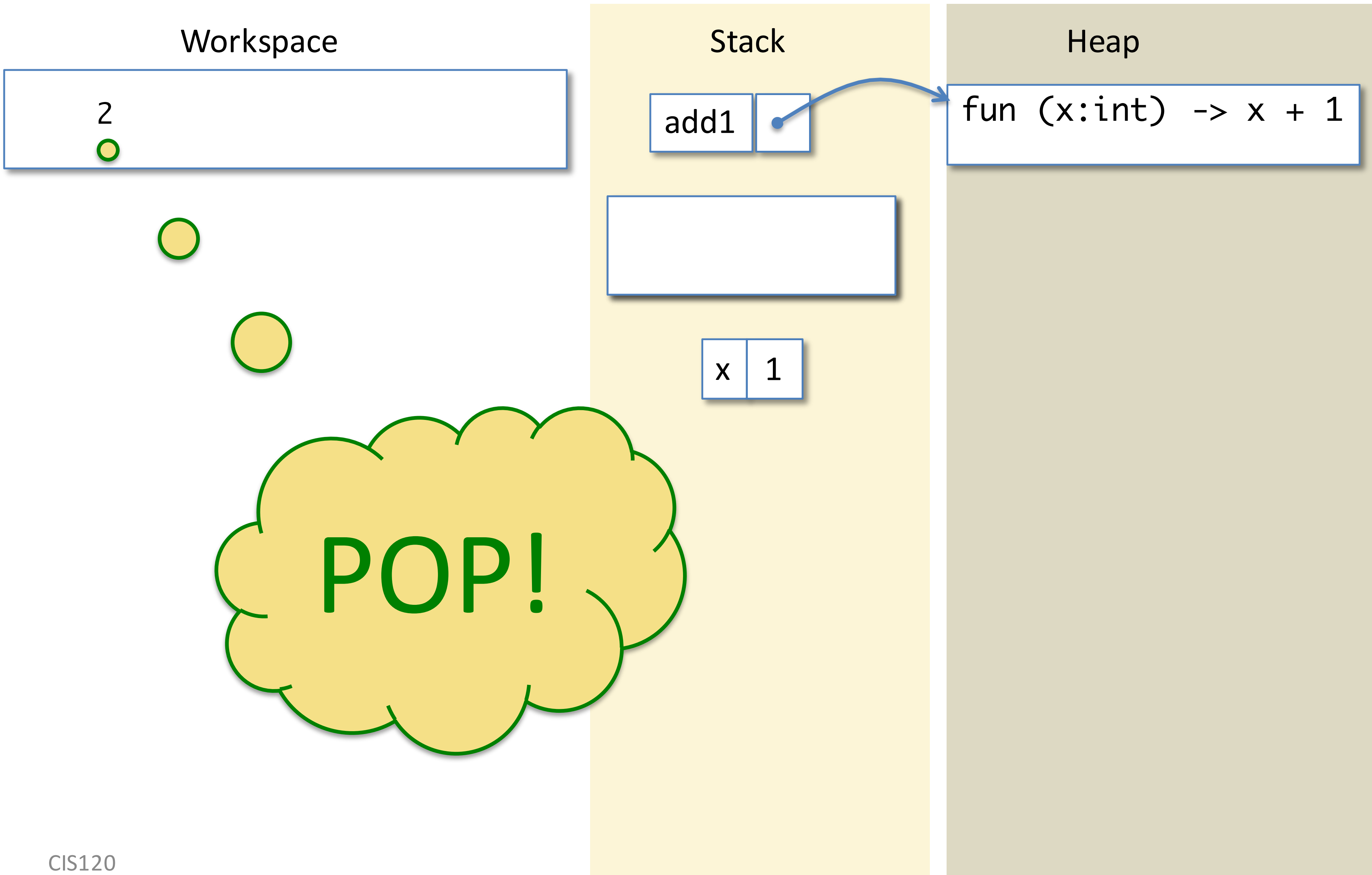
Function Simplification



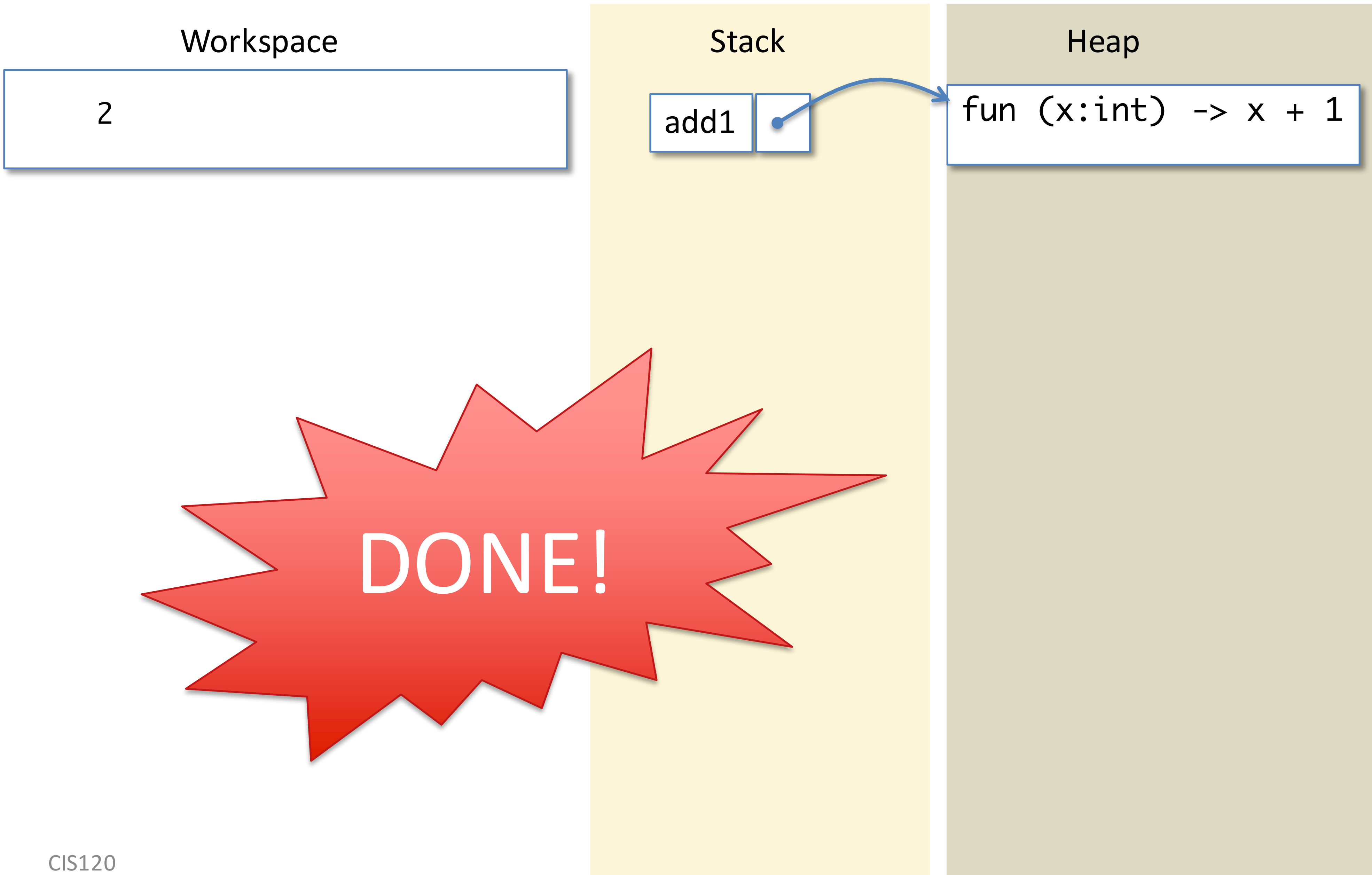
Function Simplification



Function Simplification



Function Simplification



Simplifying Functions

- A function definition "let rec f $(x_1:t_1)\ldots(x_n:t_n)$ = e in body" is always ready.
  - It is simplified by replacing it with "let f = fun $(x:t_1)\ldots(x:t_n)$ = e in body"

- A function "fun $(x_1:t_1)\ldots(x_n:t_n)$ = e" is always ready.
  - It is simplified by moving the function to the heap and replacing the function expression with a pointer to that heap data.

- A function *call* is ready if the function and its arguments are all values
  - it is simplified by
    - saving the current workspace contents on the stack
    - adding bindings for the function's parameter variables (to the actual argument values) to the end of the stack
    - copying the function's body to the workspace

Function Completion

When the workspace contains just a single value, we *pop the stack* by removing everything back to (and including) the last saved workspace contents.

The value currently in the workspace is substituted for the function application expression in the saved workspace contents, which are put back into the workspace.

If there aren't any saved workspace contents in the stack, the whole computation is finished and the value in the workspace is its final result.

What is your current level of comfort with the Abstract Stack Machine?

1. got it well under control
2. OK but need to work with it a little more
3. a little puzzled
4. very puzzled
5. very *very* puzzled :-)

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

Workspace

```
let append =
   in
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
```

Let Expression

Workspace

```
let append =
    in
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
```

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

append

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
```

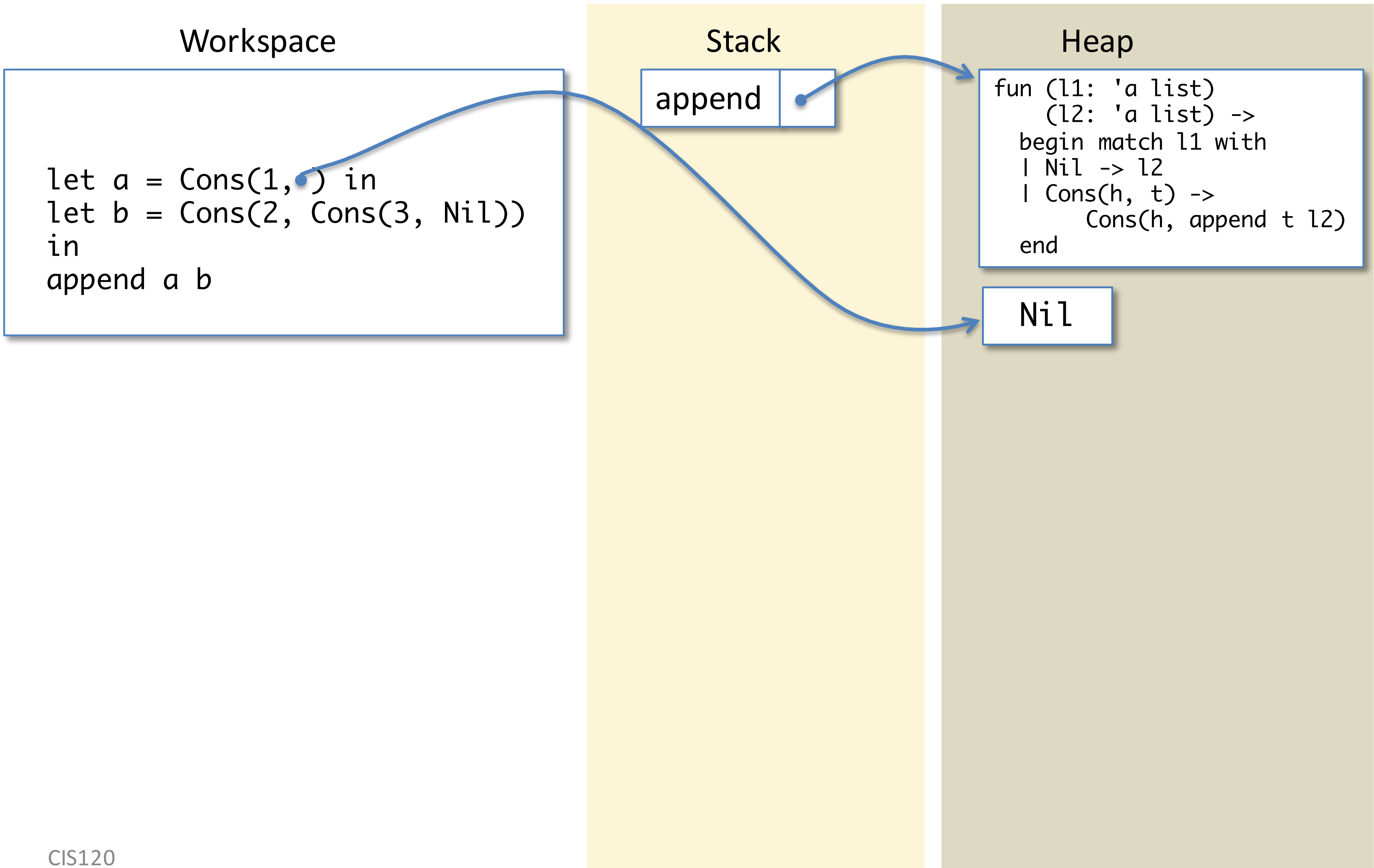
Stack

append	•

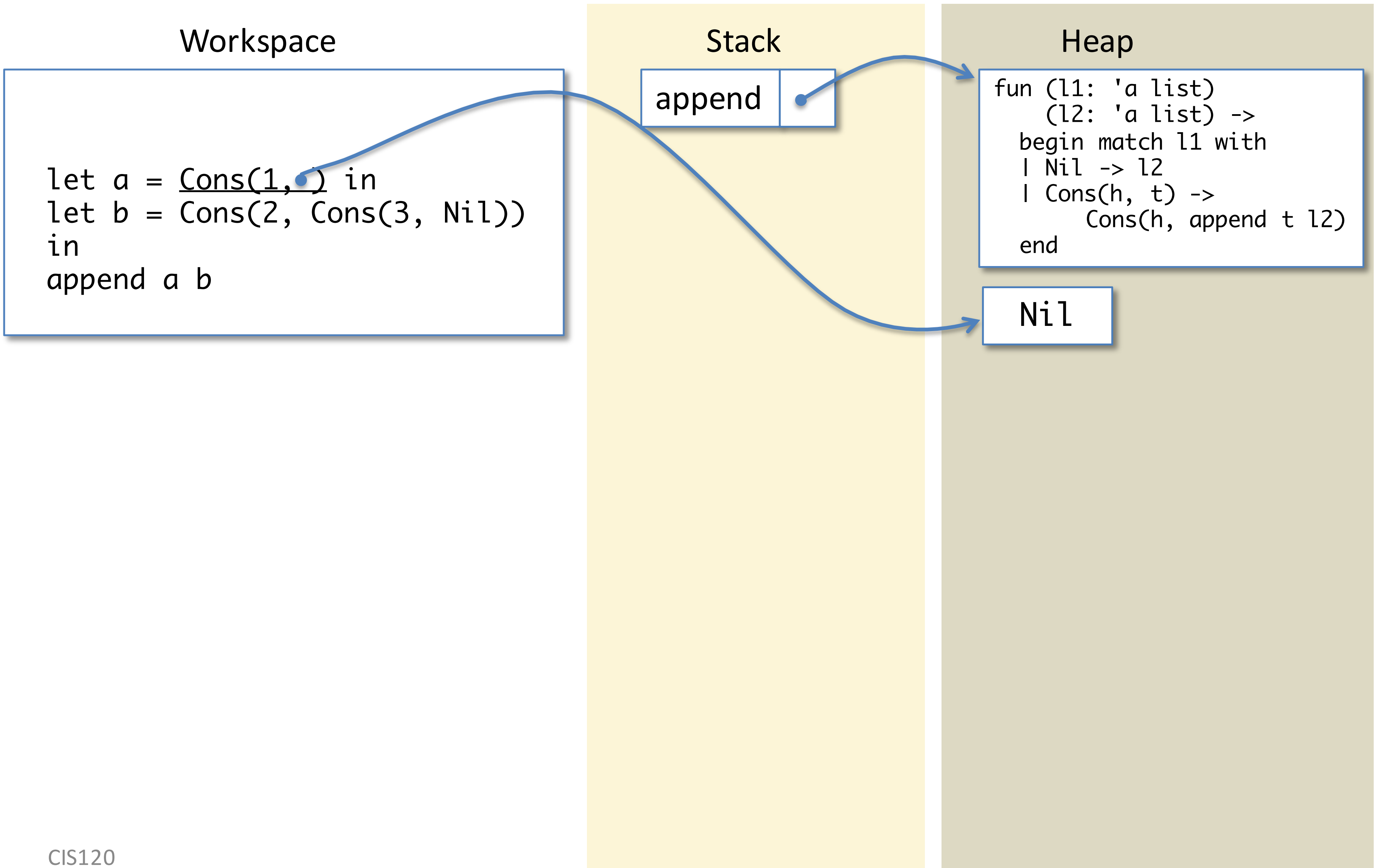
Heap

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

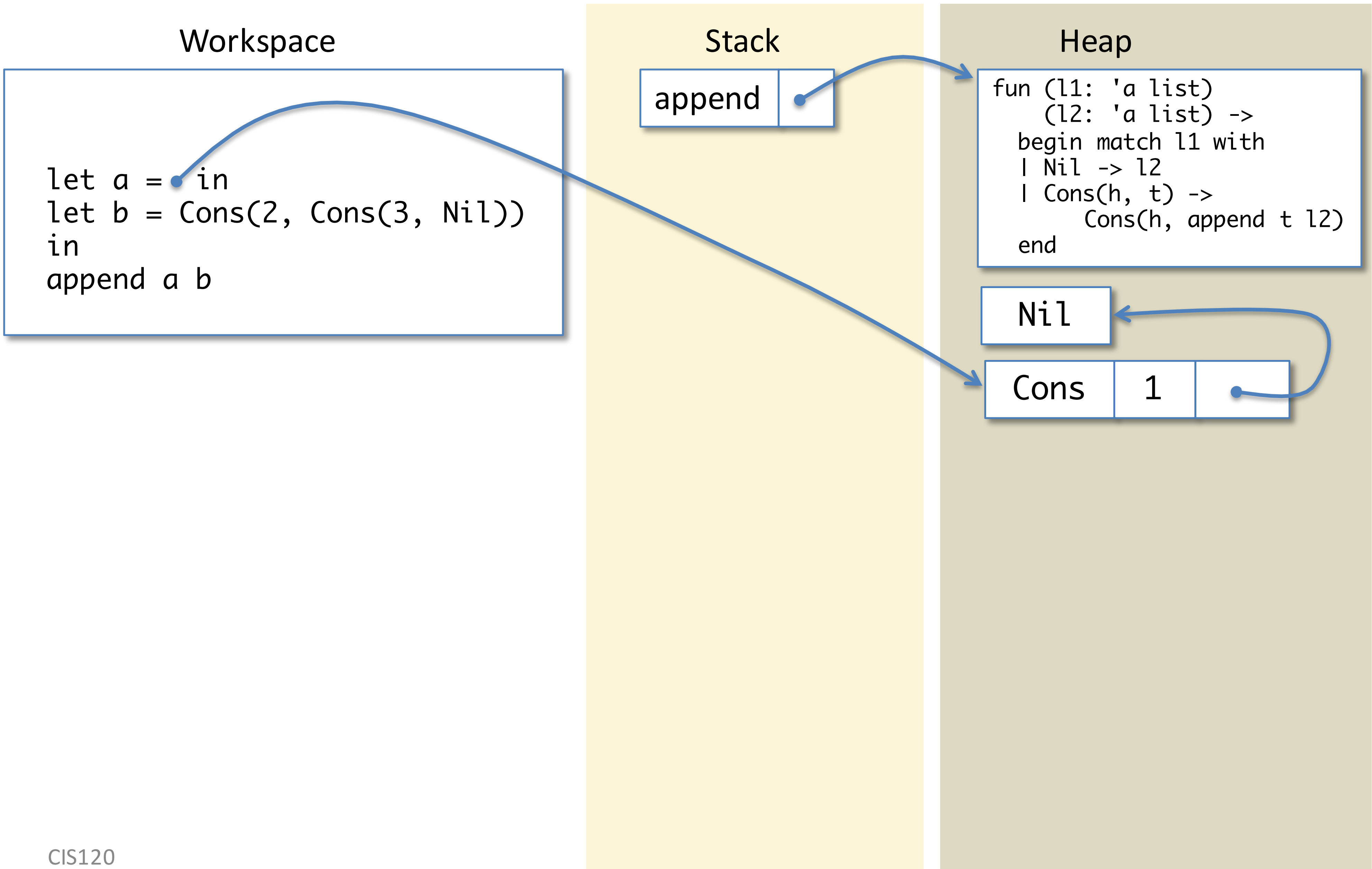
Allocate a Nil cell



Allocate a Cons cell



Allocate a Cons cell



Let Expression

Workspace

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

Stack

append	•

Heap

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

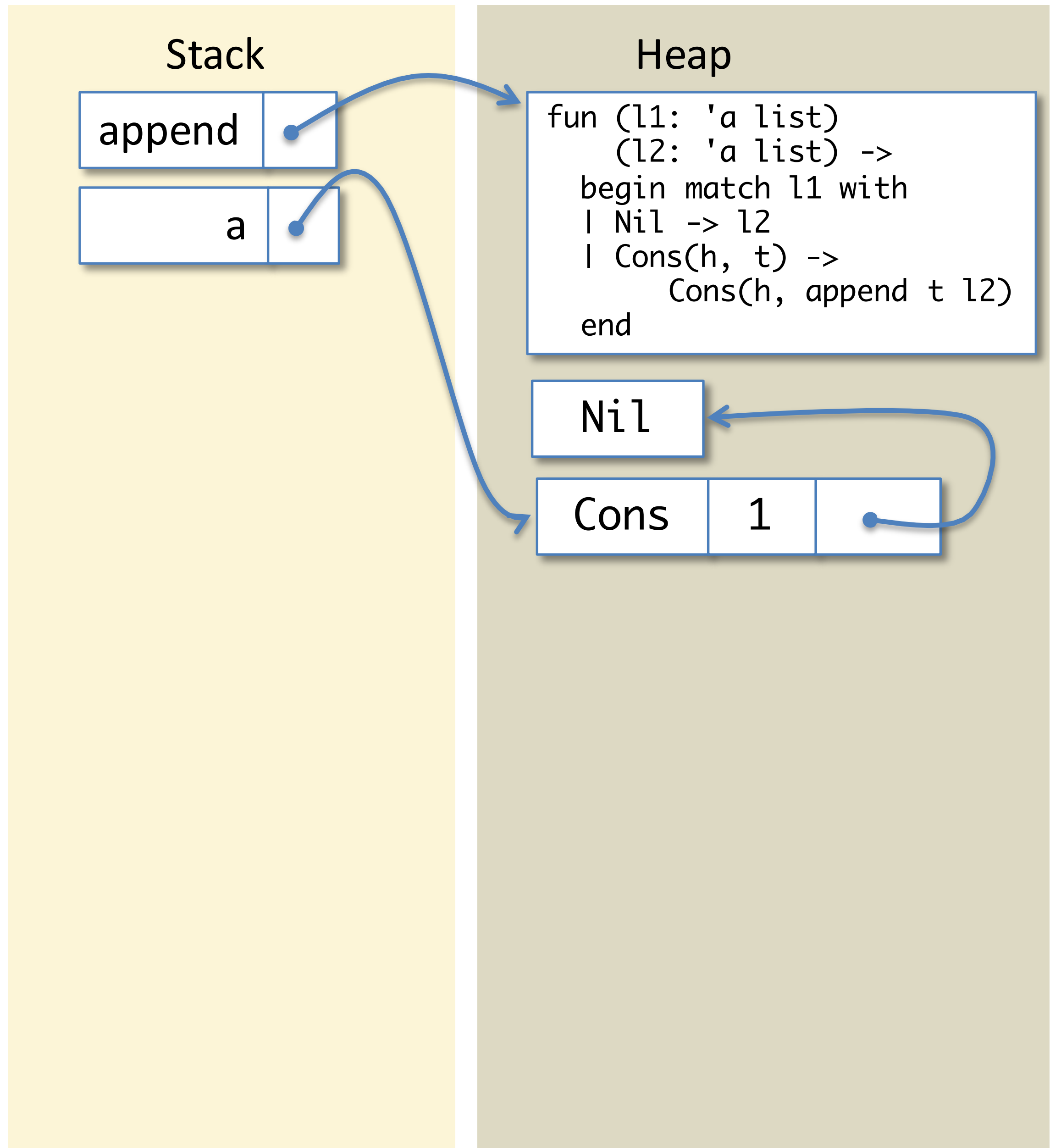
Nil

Cons	1	•

Create a Stack Binding

Workspace

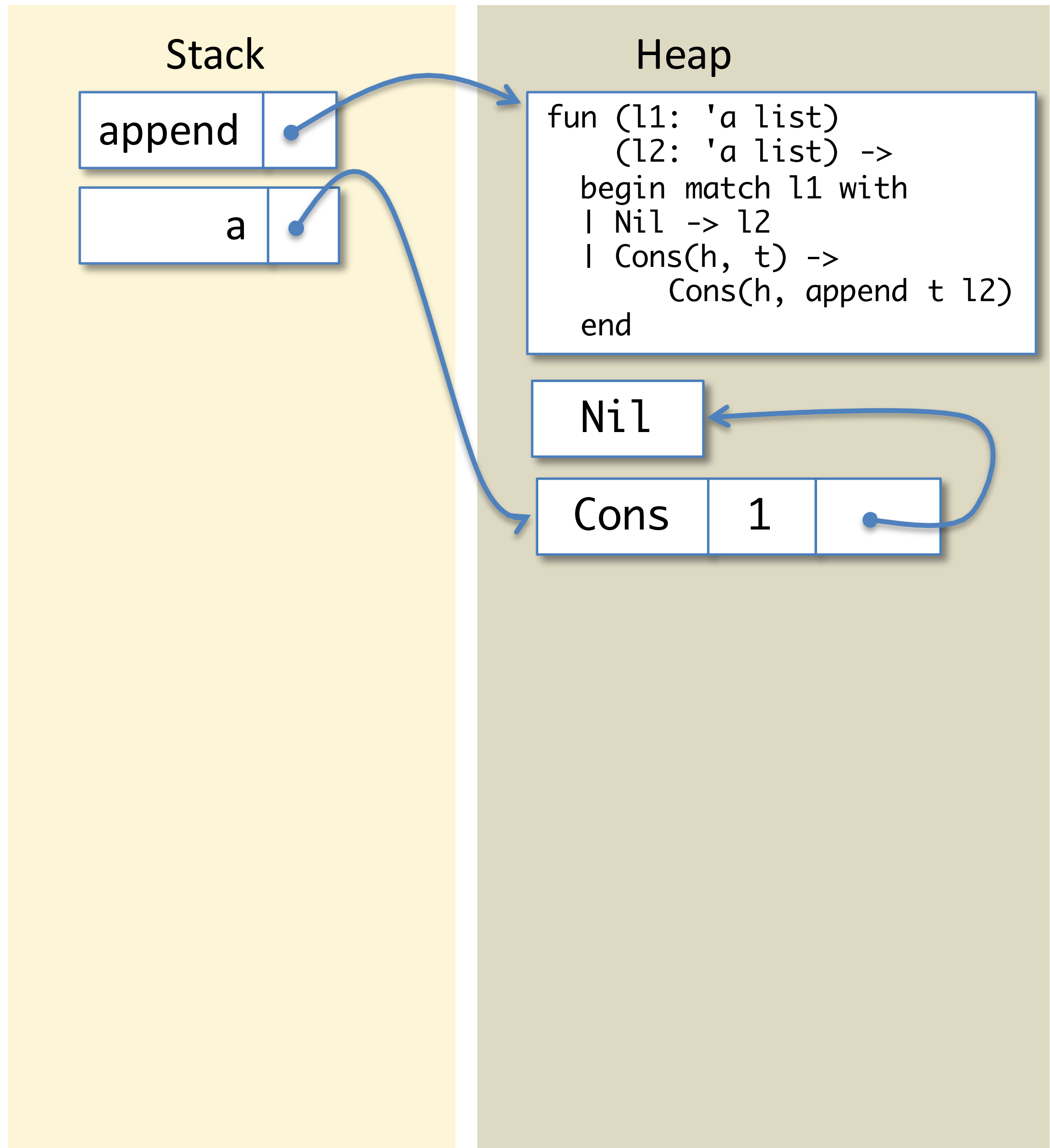
```
let b = Cons(2, Cons(3, Nil))
in
append a b
```



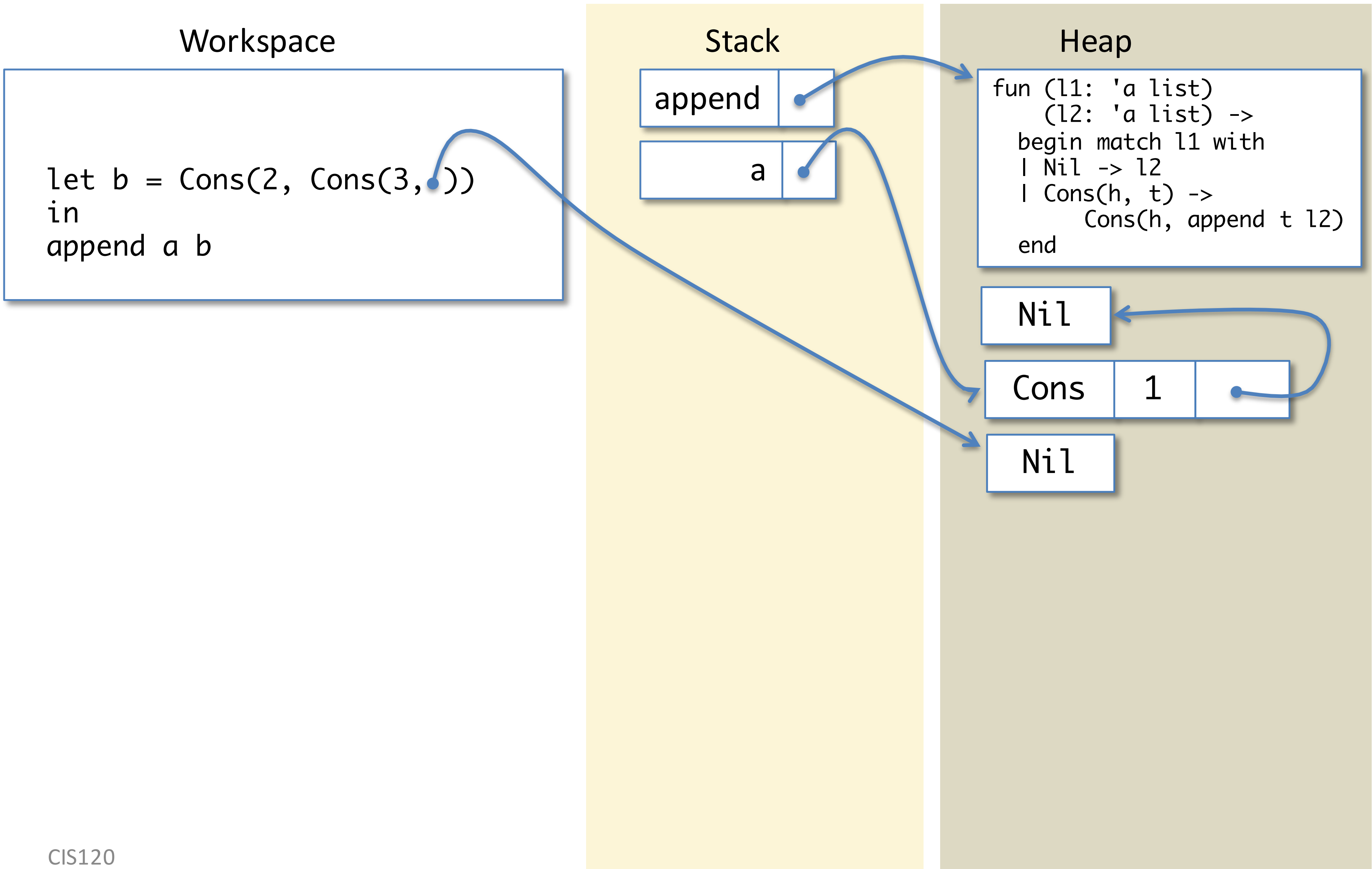
Allocate a Nil cell

Workspace

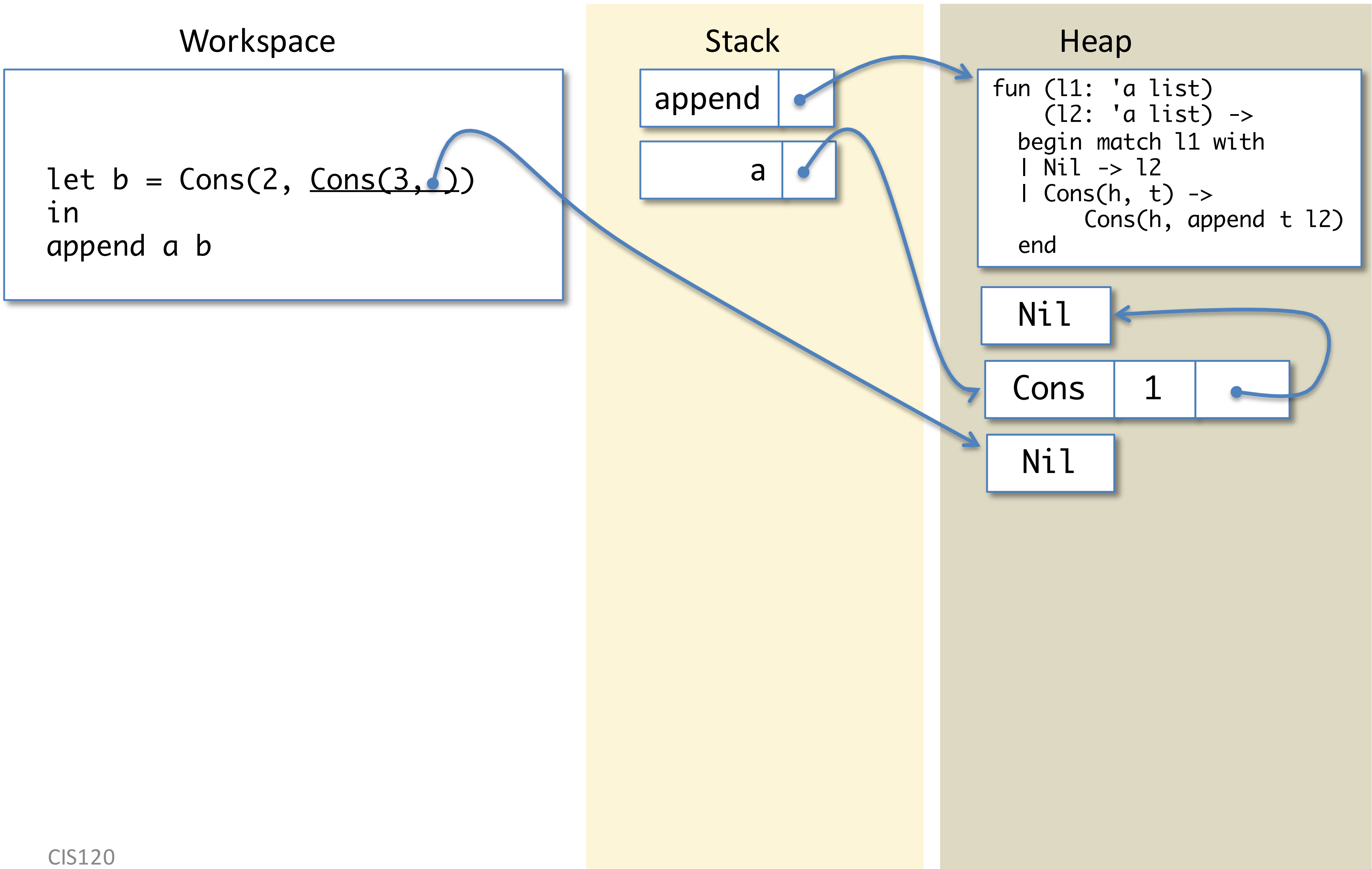
```
let b = Cons(2, Cons(3, Nil))
in
append a b
```



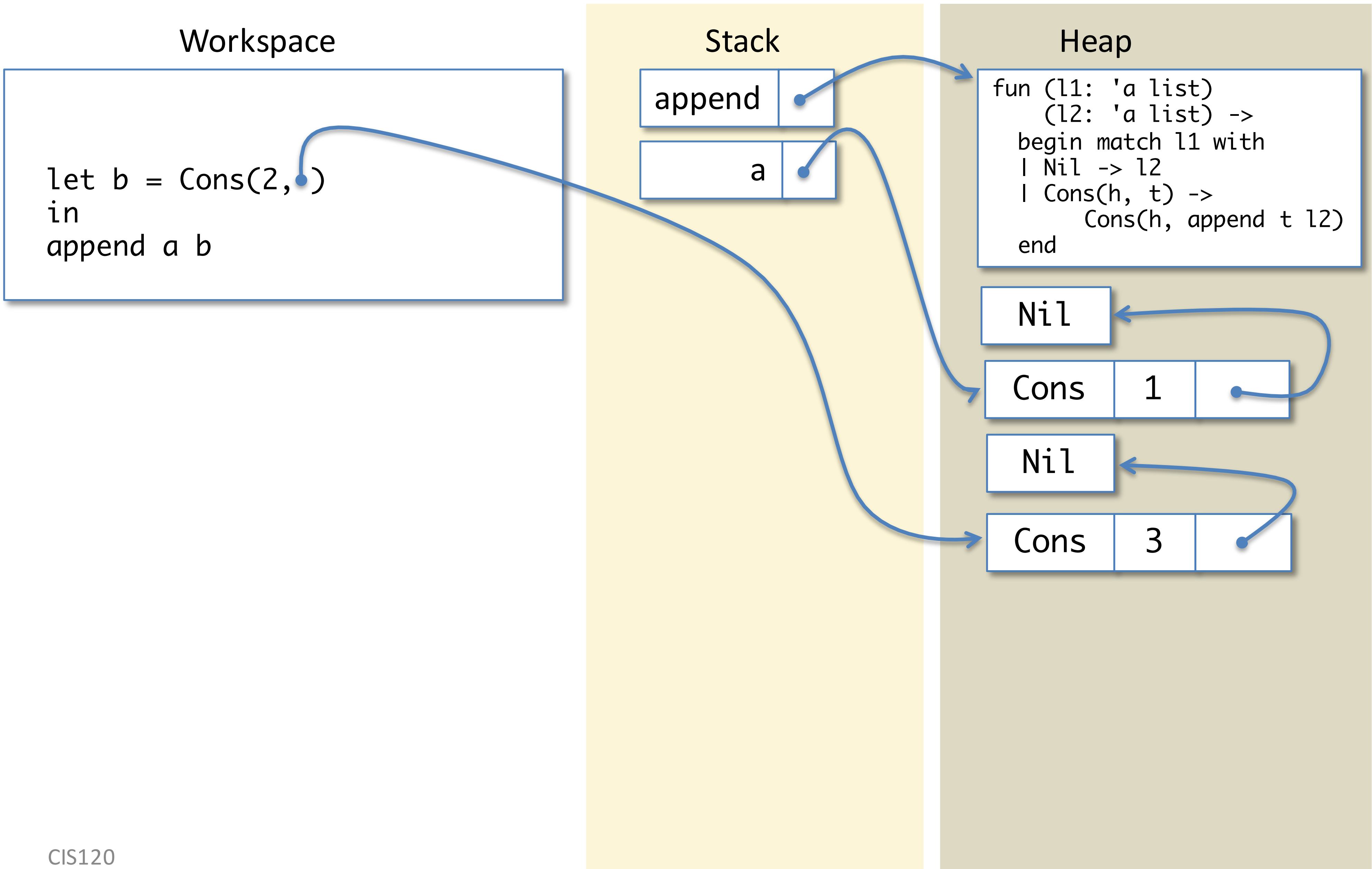
Allocate a Nil cell



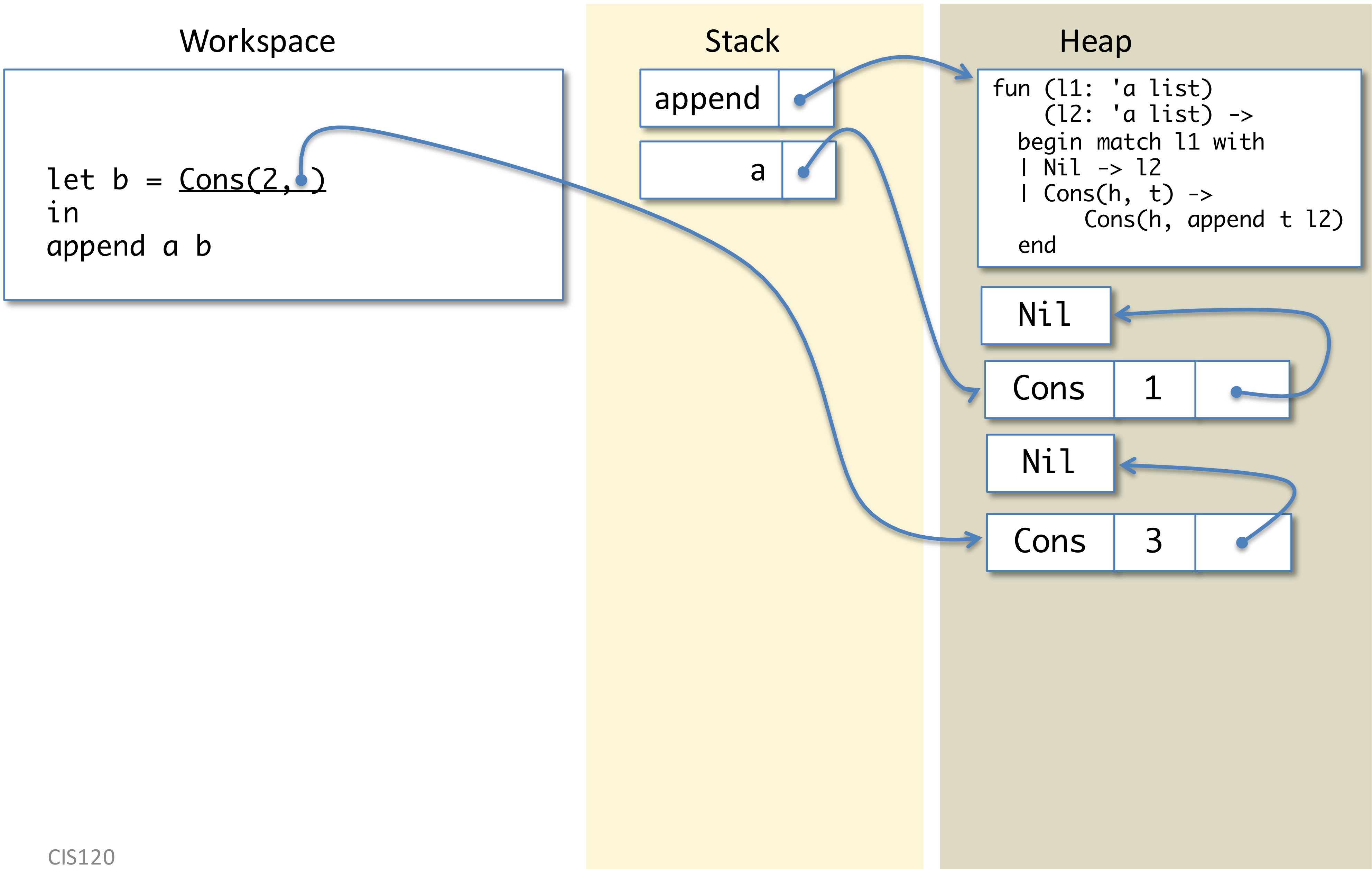
Allocate a Cons cell



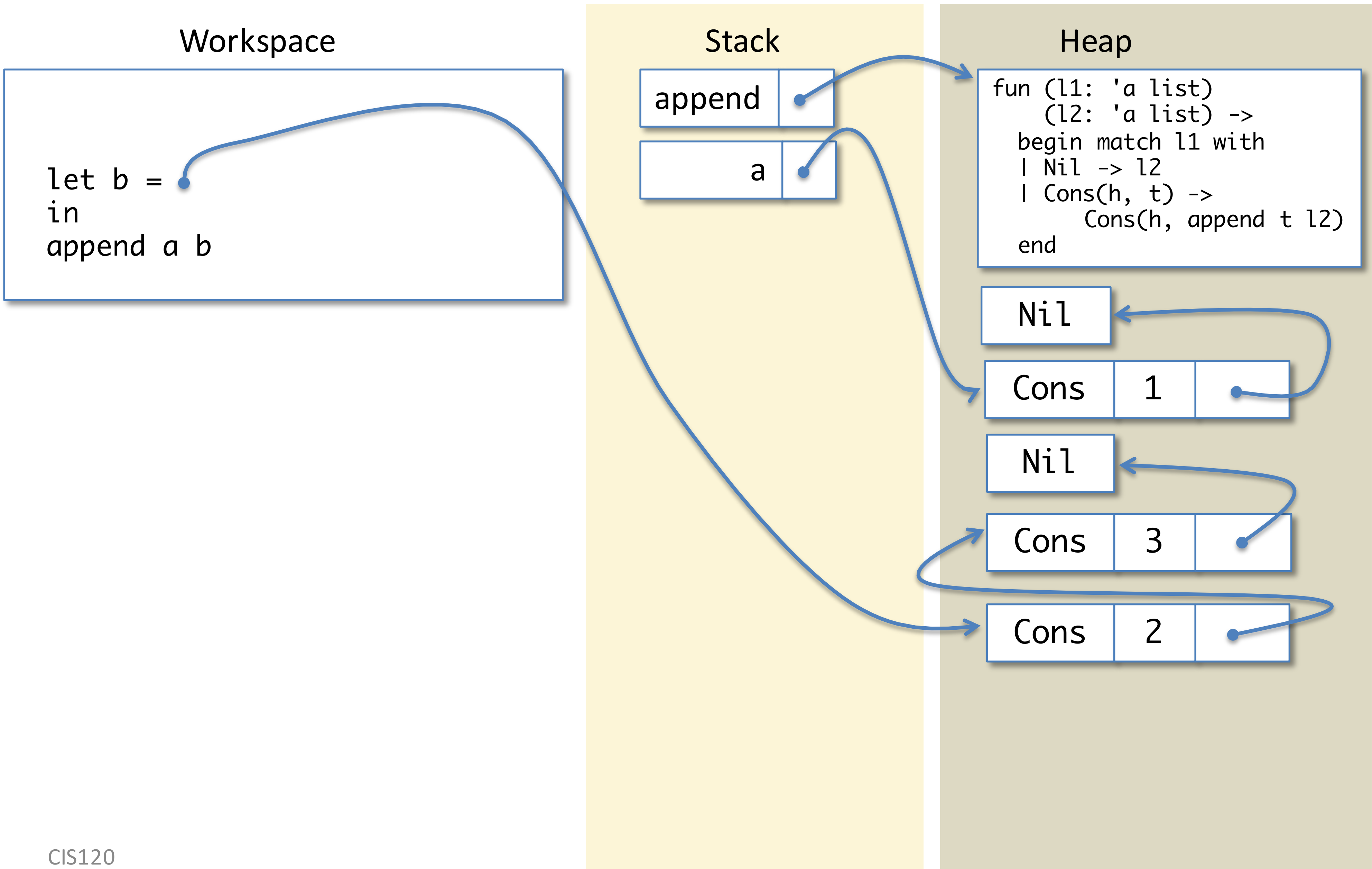
Allocate a Cons cell



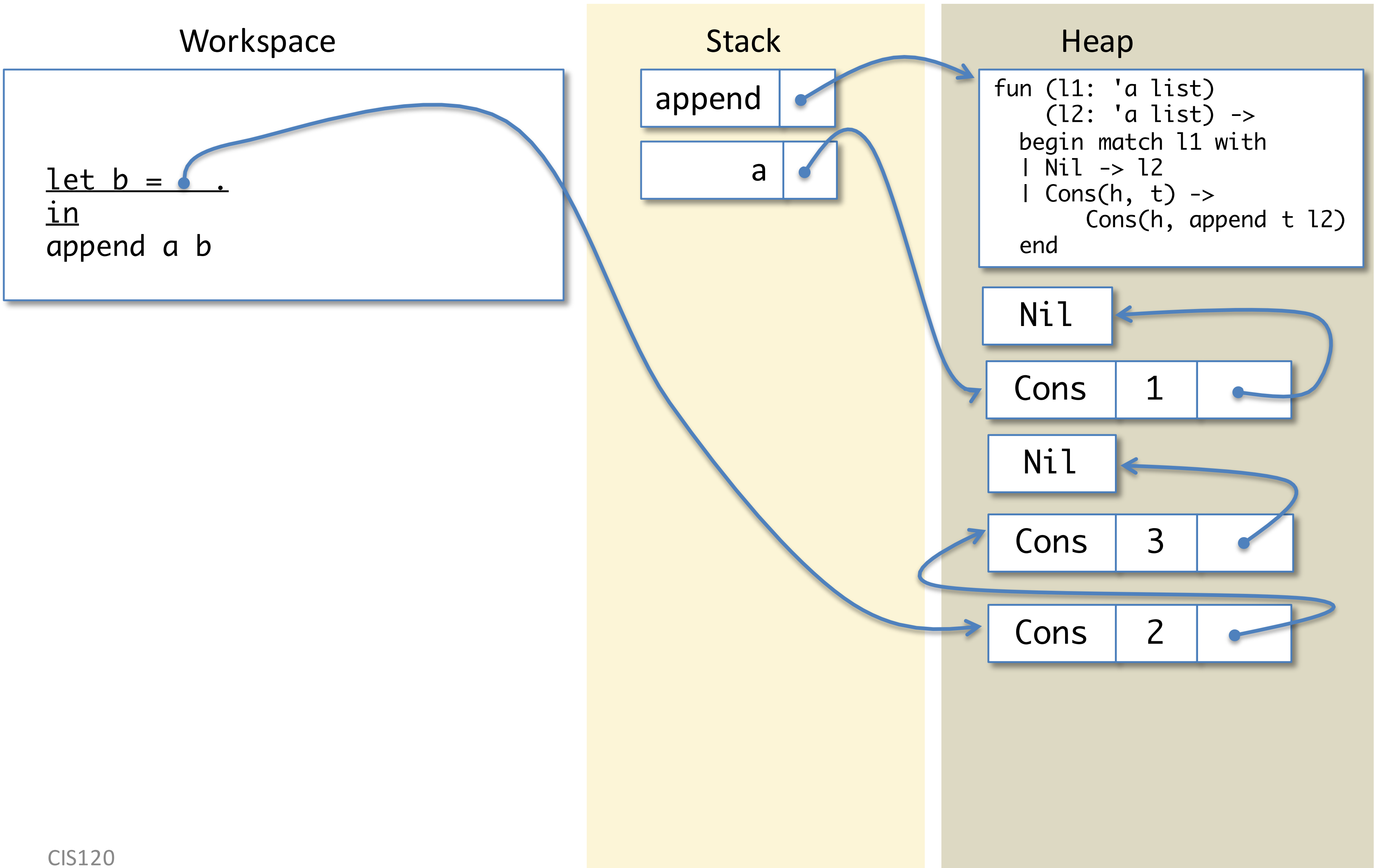
Allocate a Cons cell



Allocate a Cons cell



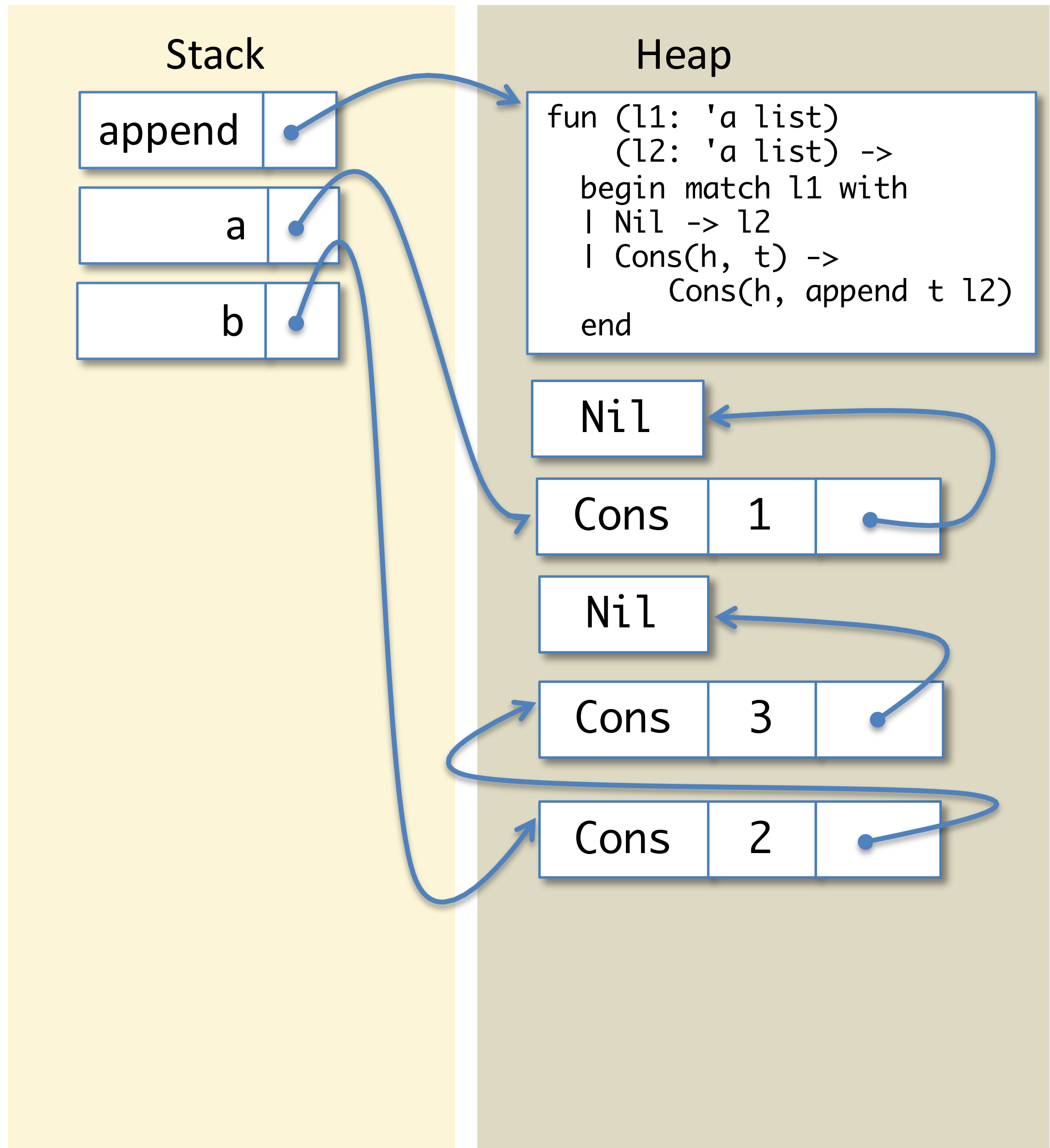
Let Expression



Create a Stack Binding

Workspace

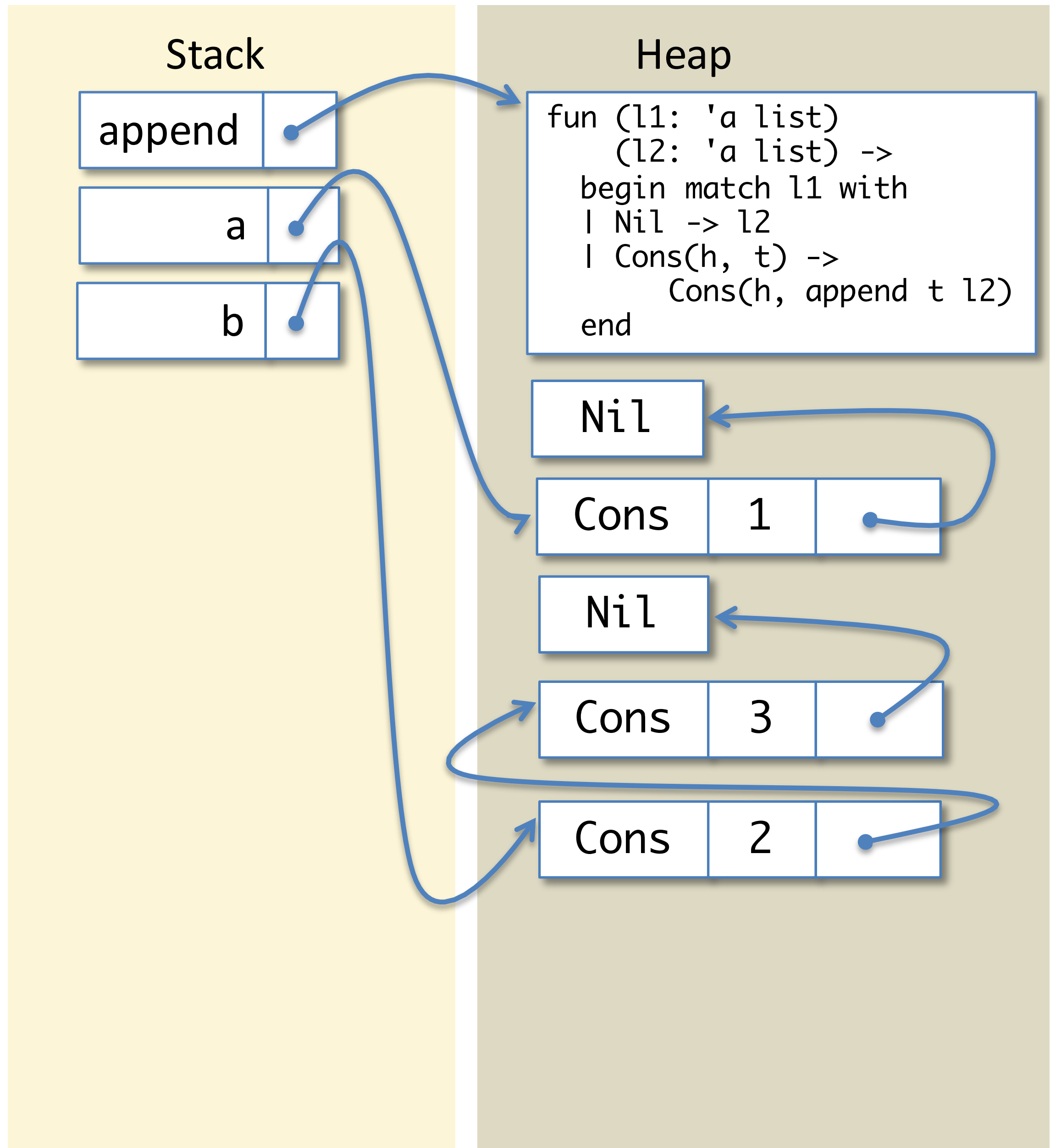
```
append a b
```



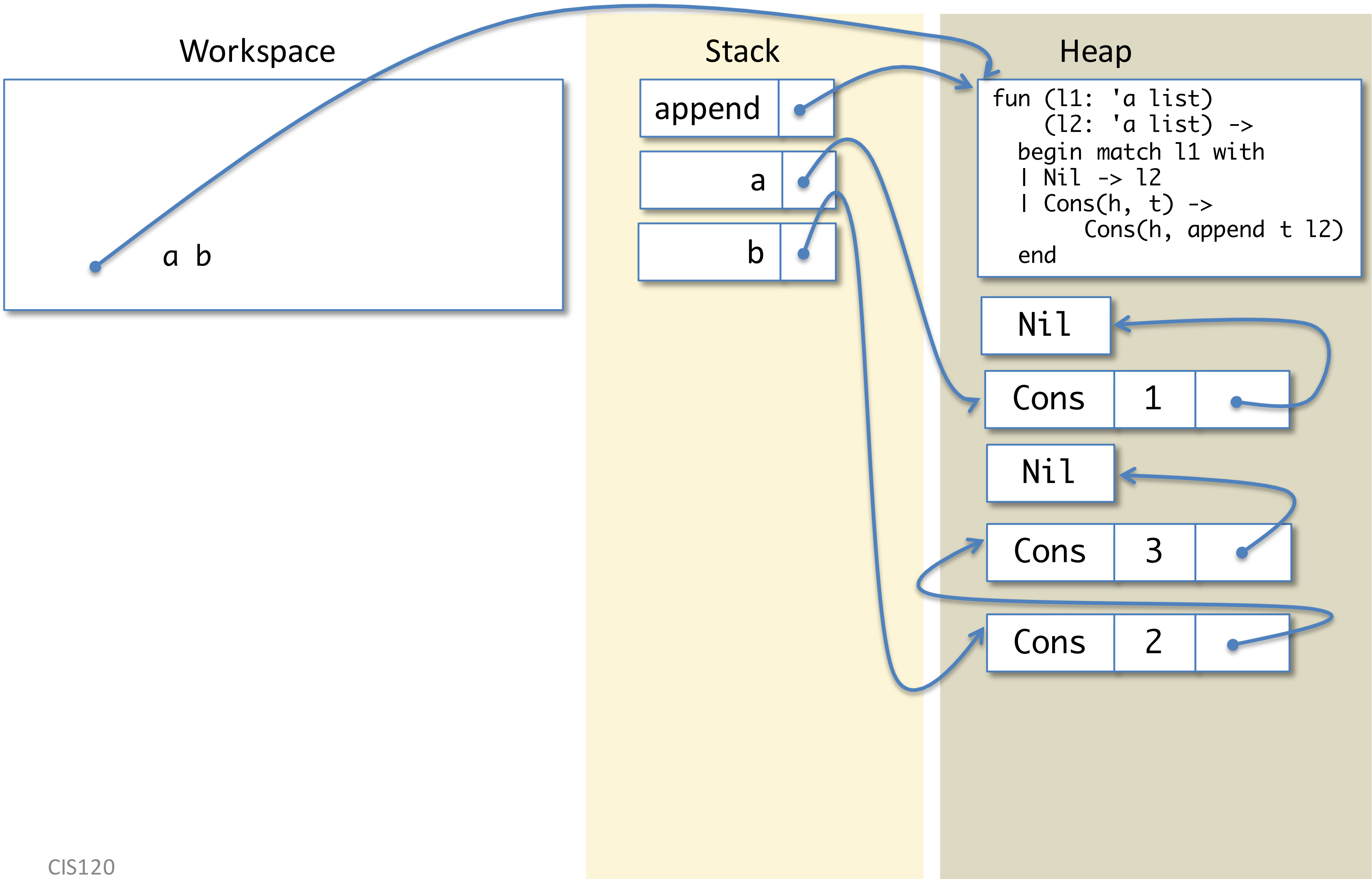
Lookup ‘append’

Workspace

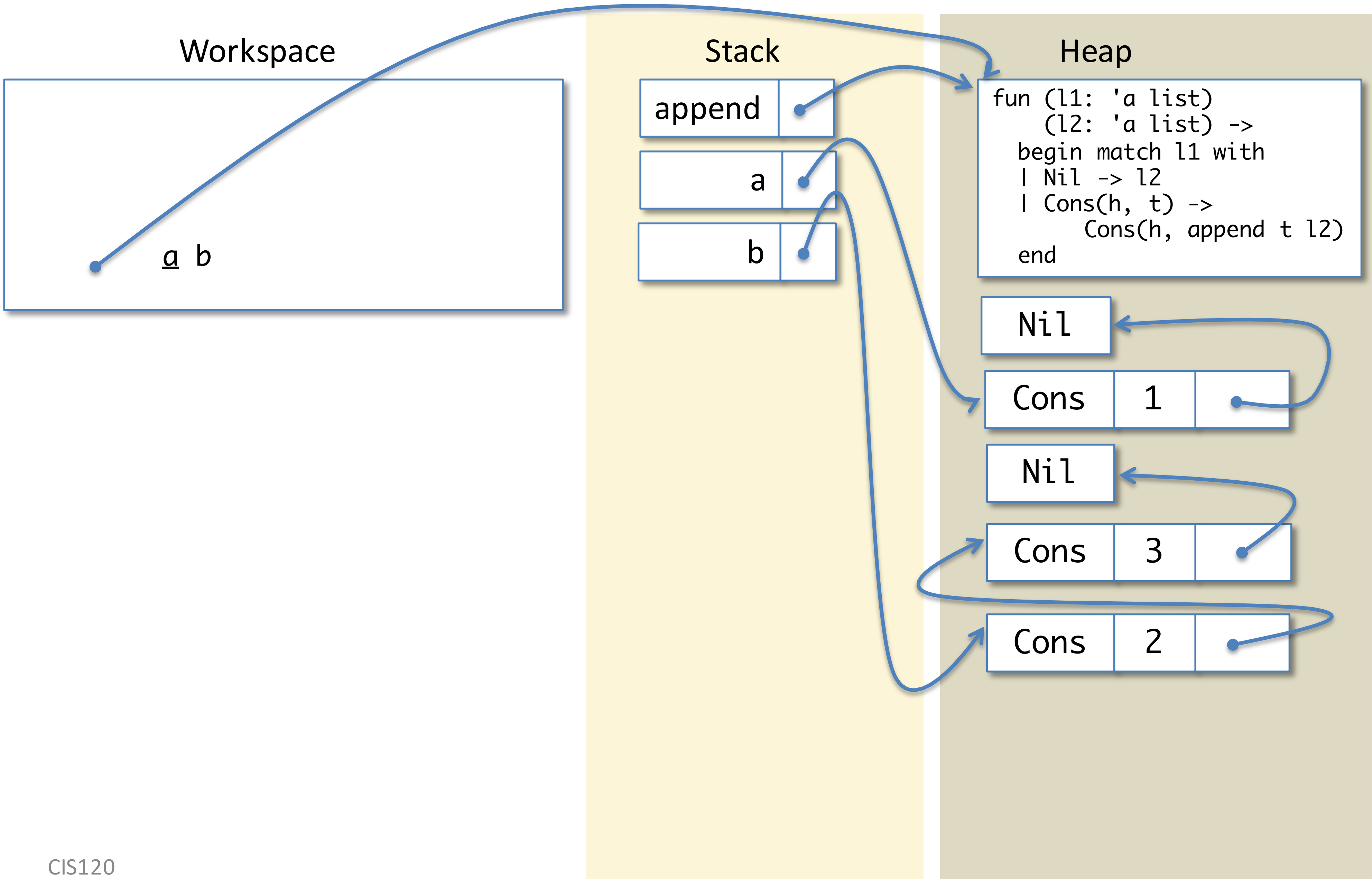
append a b



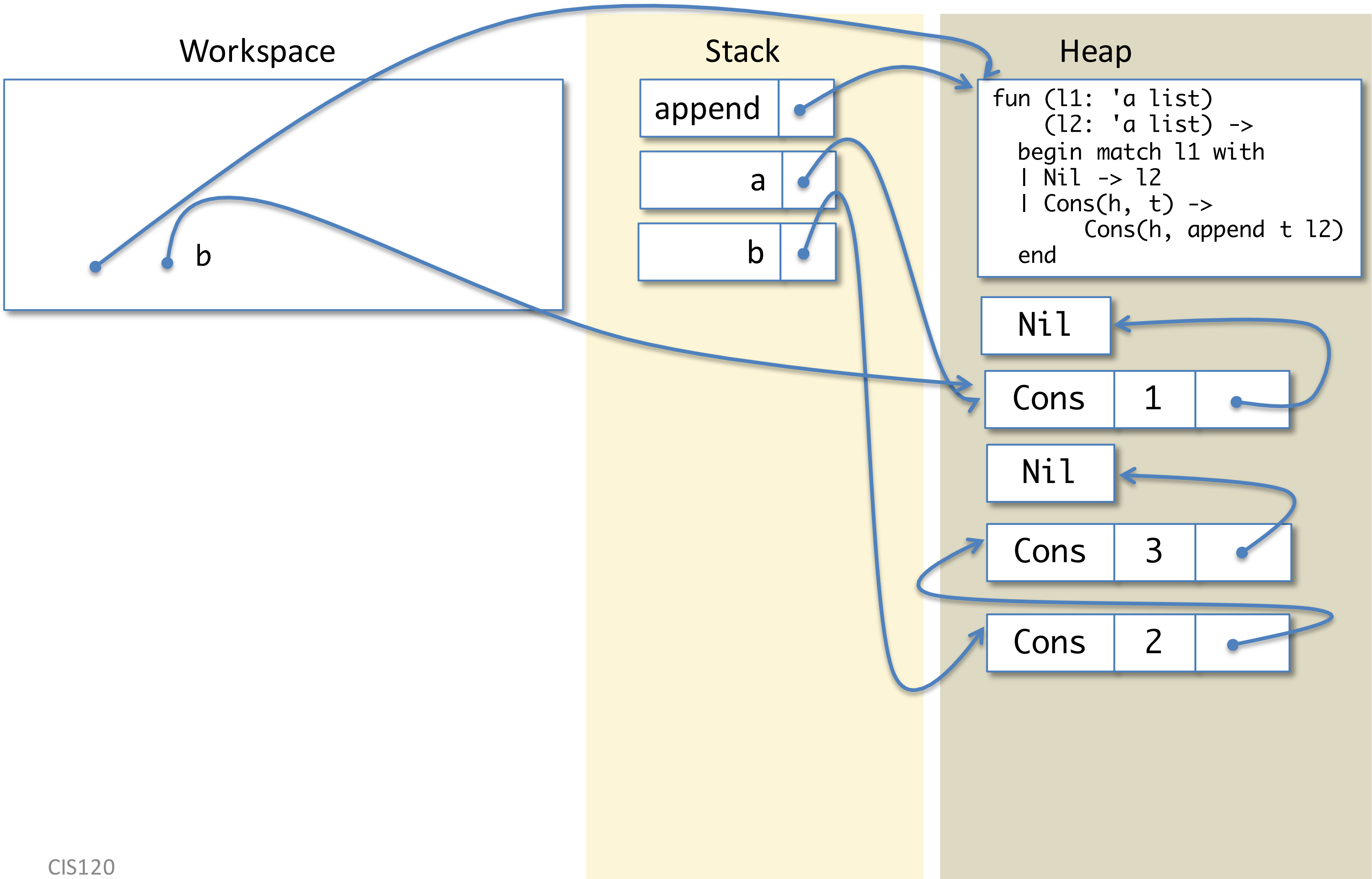
Lookup 'append'



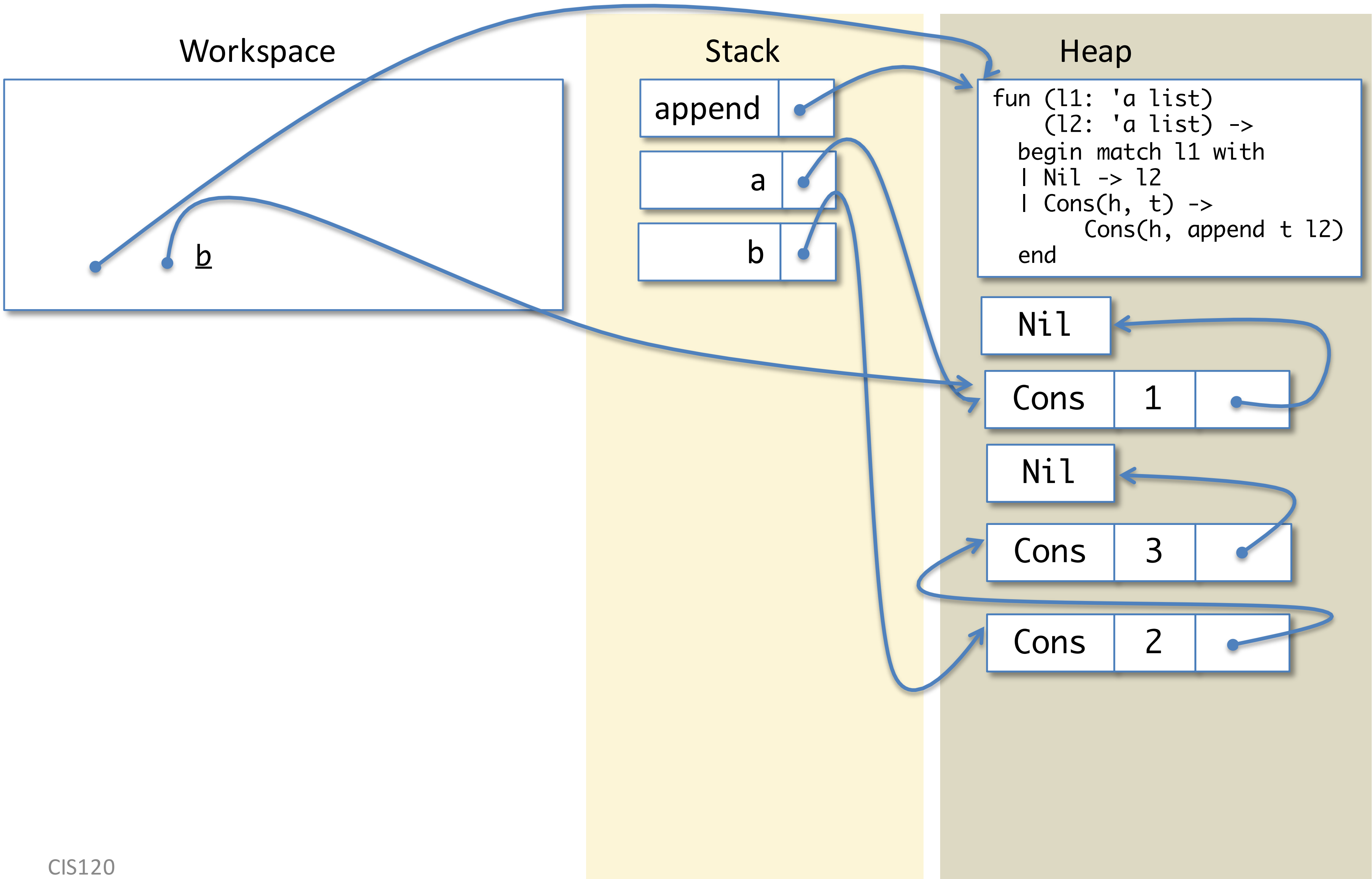
Lookup ‘a’


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

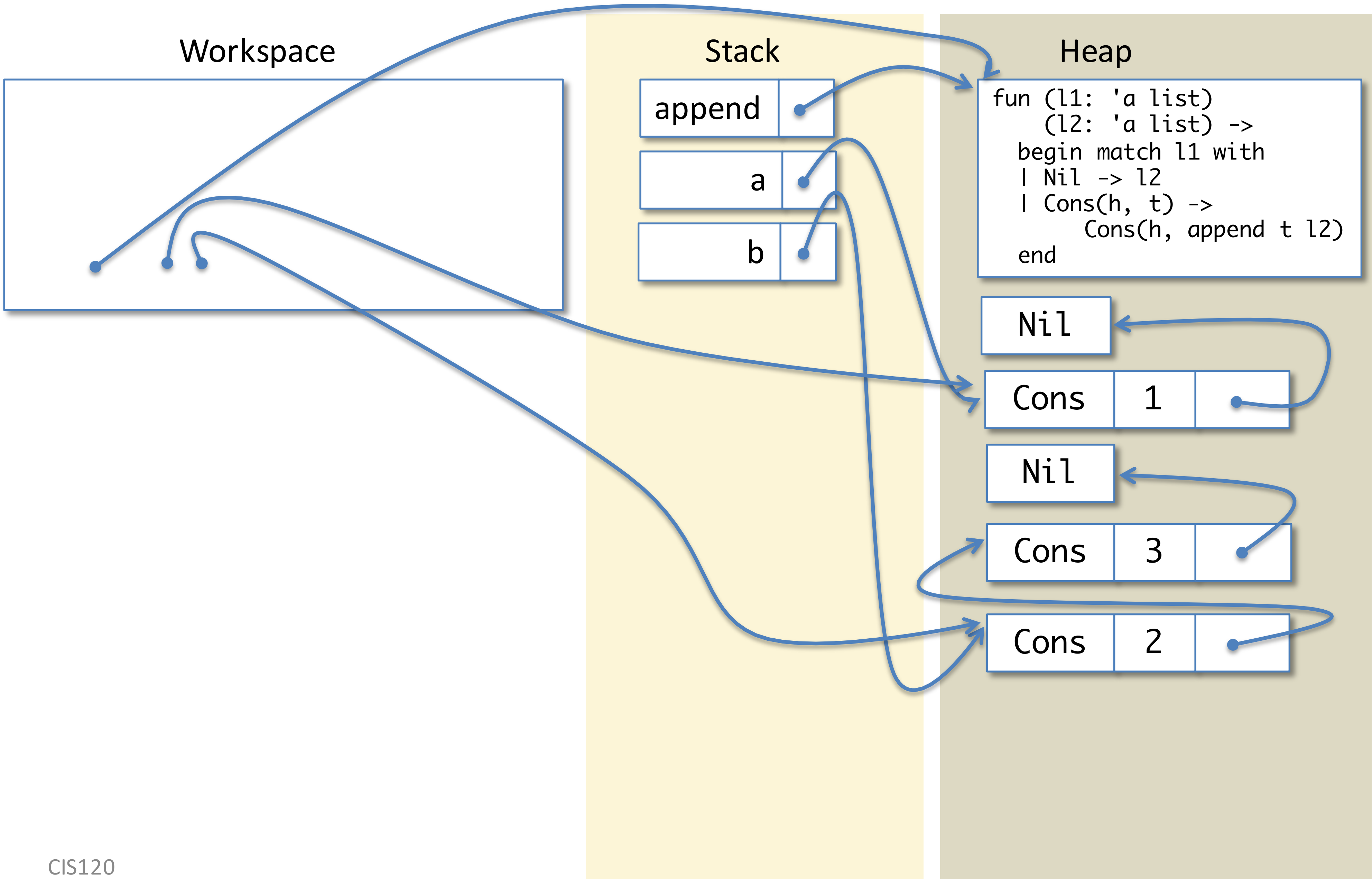
Lookup 'a'


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’

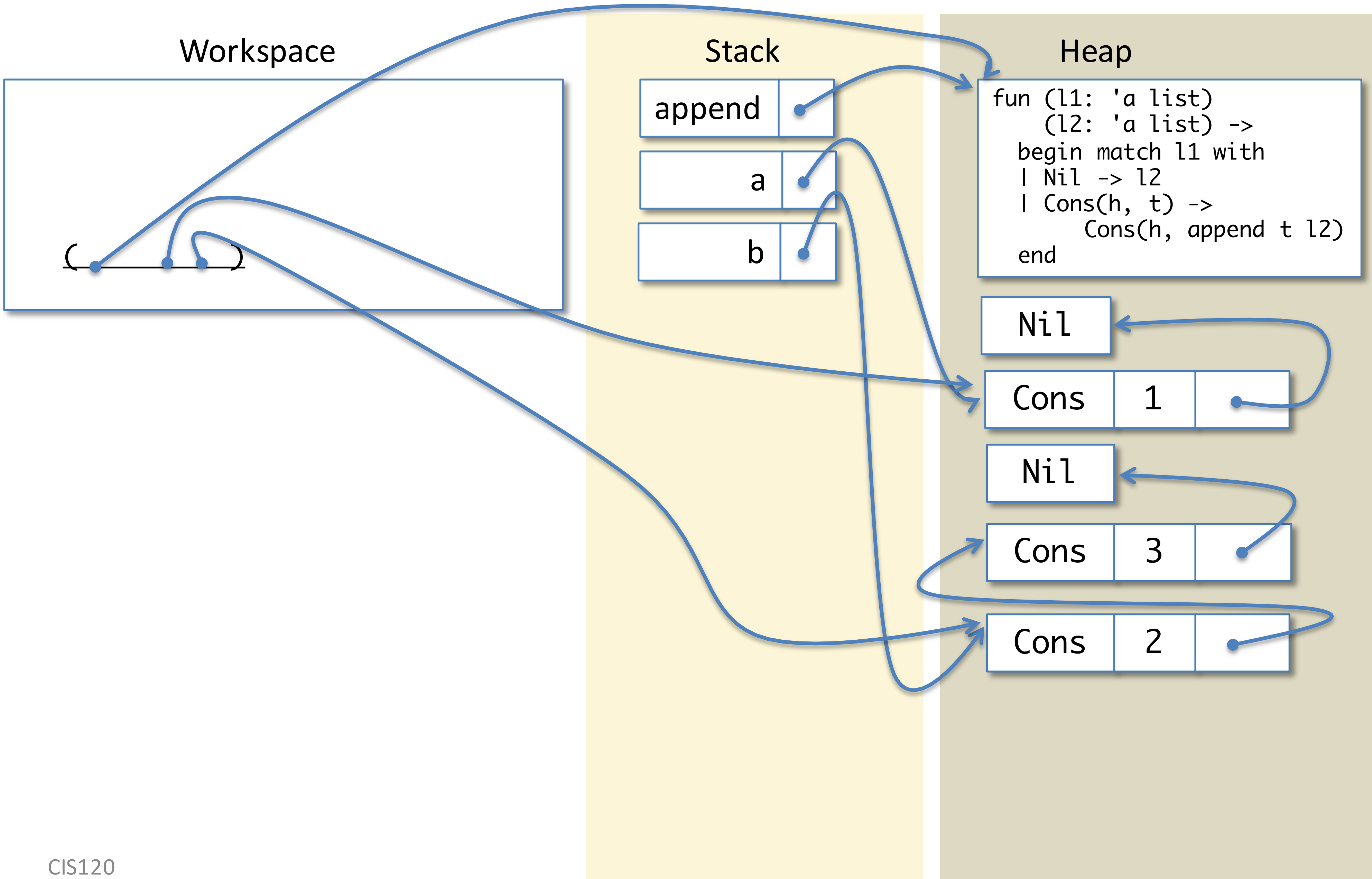

Workspace
b
Stack
append
a
b
Heap
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons
1
Nil
Cons
3
Cons
2

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

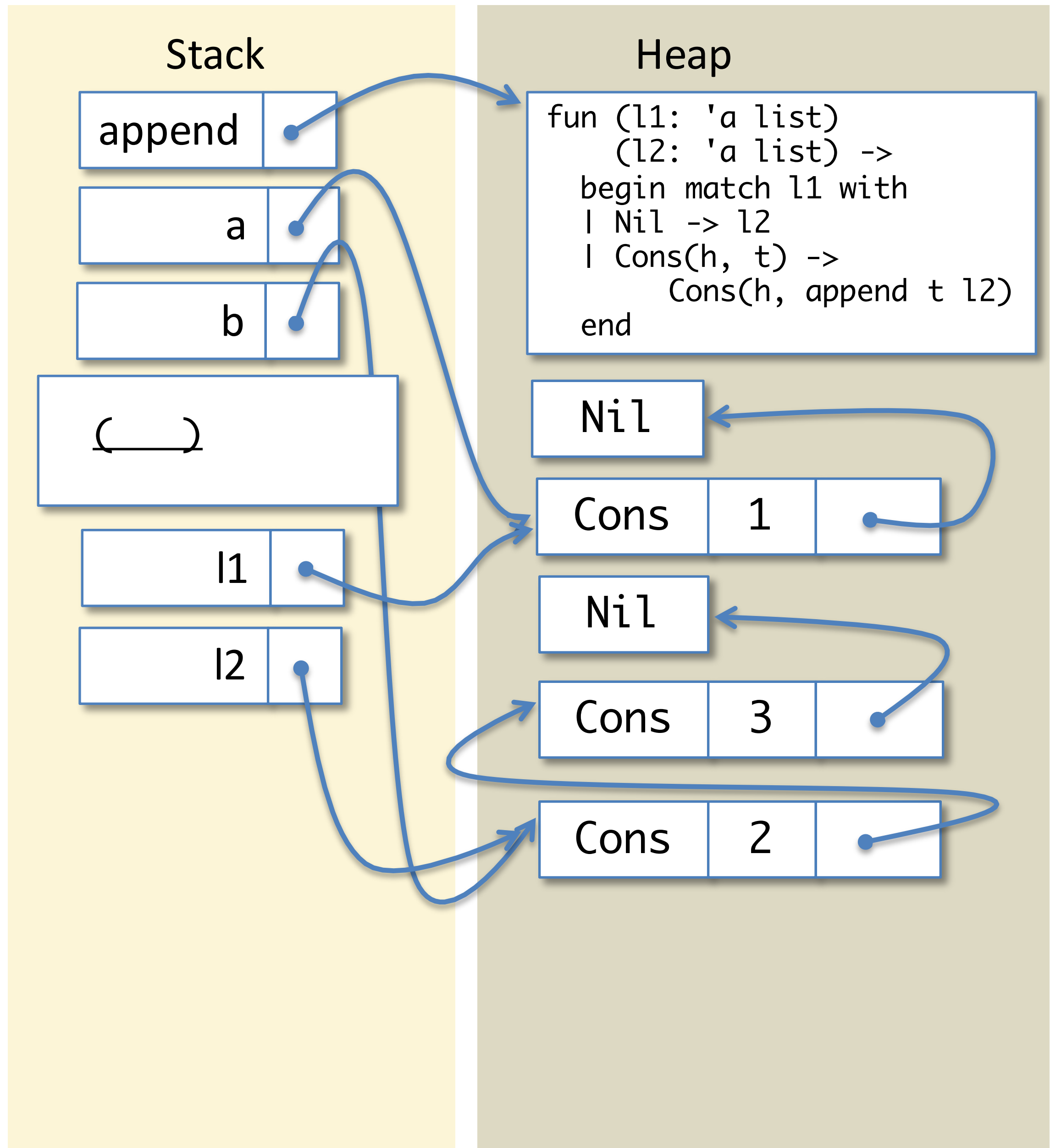
Do the Function call



Save Workspace; push l1, l2

Workspace

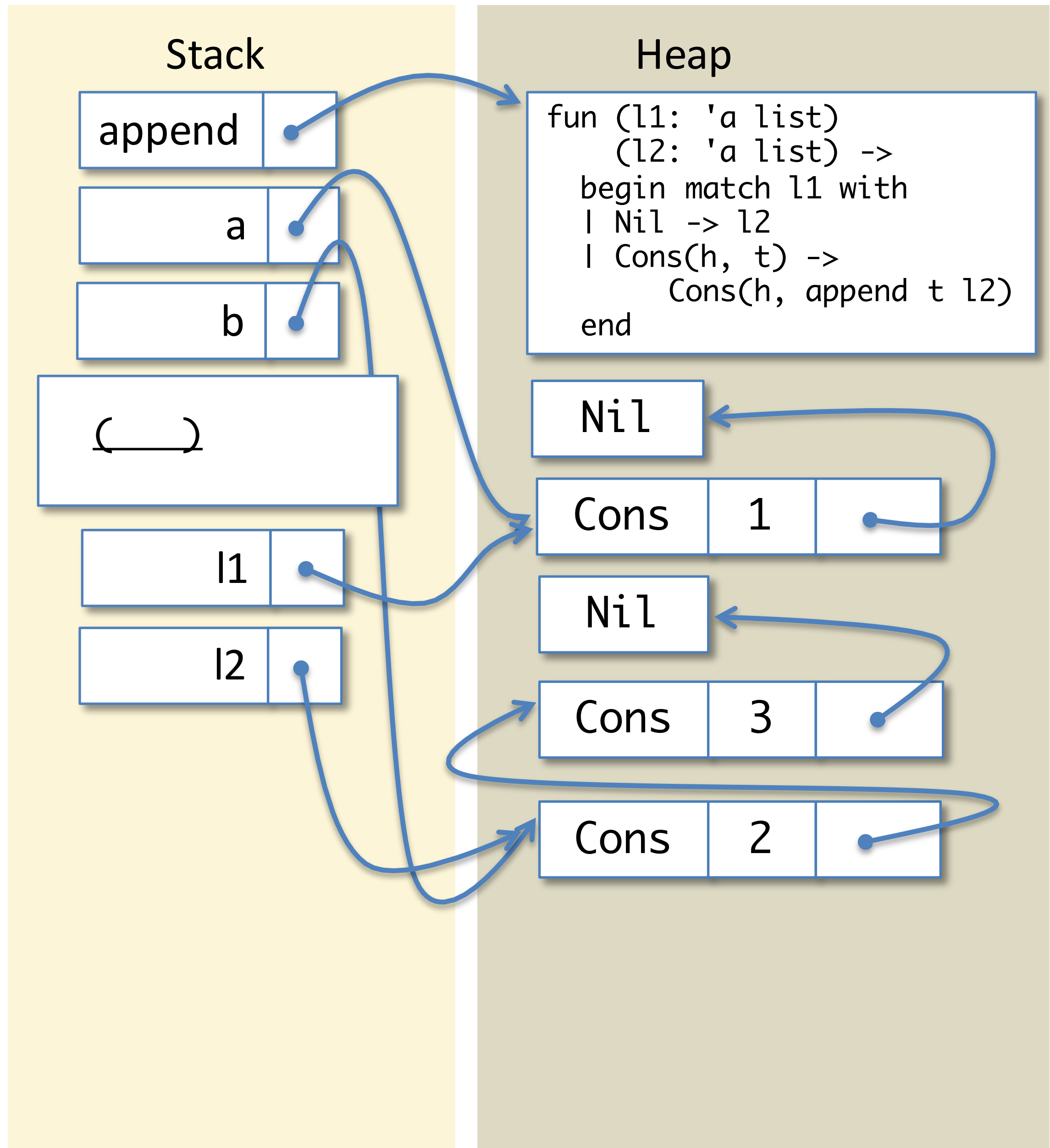
```
begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
       Cons(h, append t l2)
  end
```



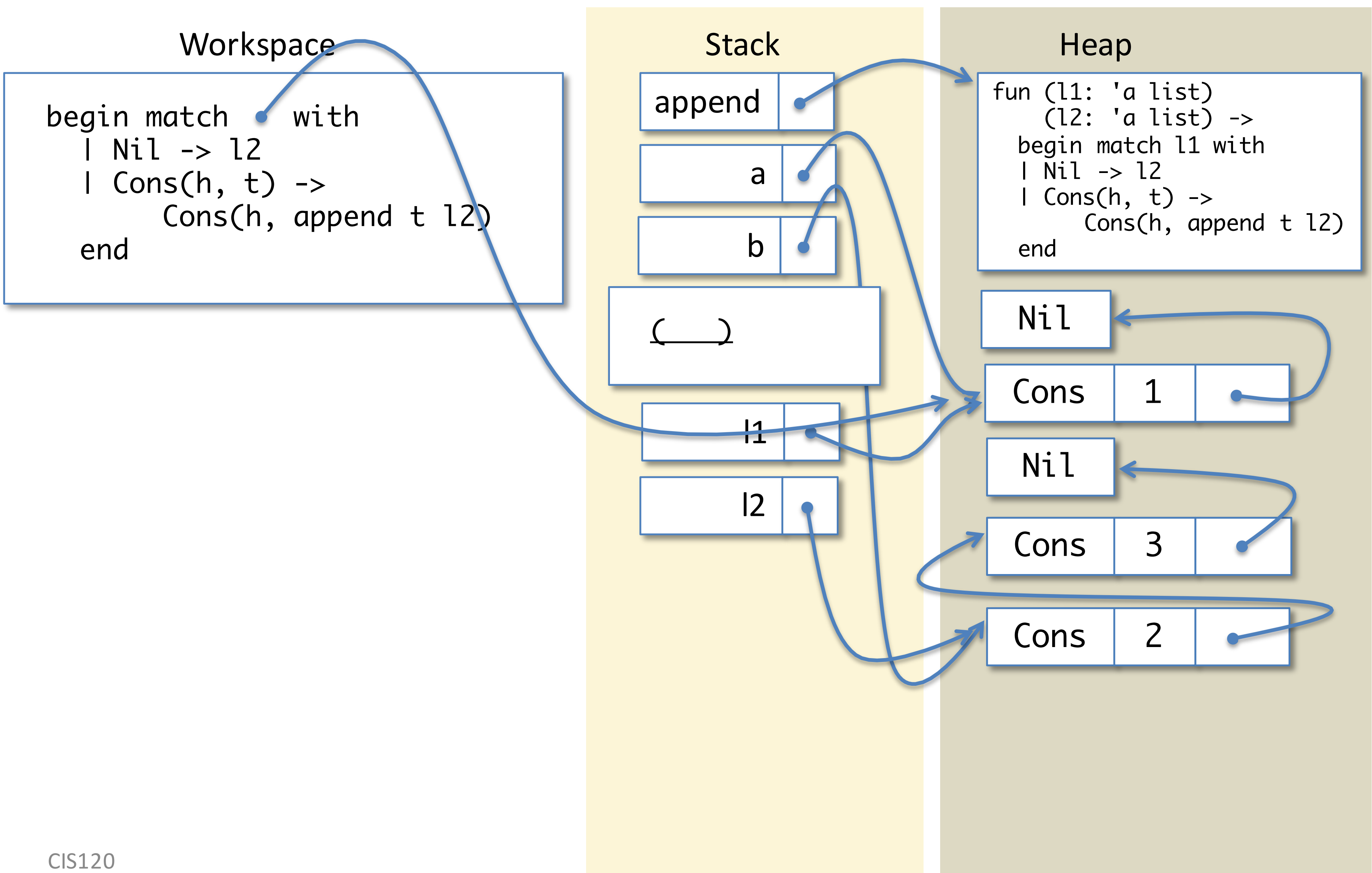
Lookup l1

Workspace

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

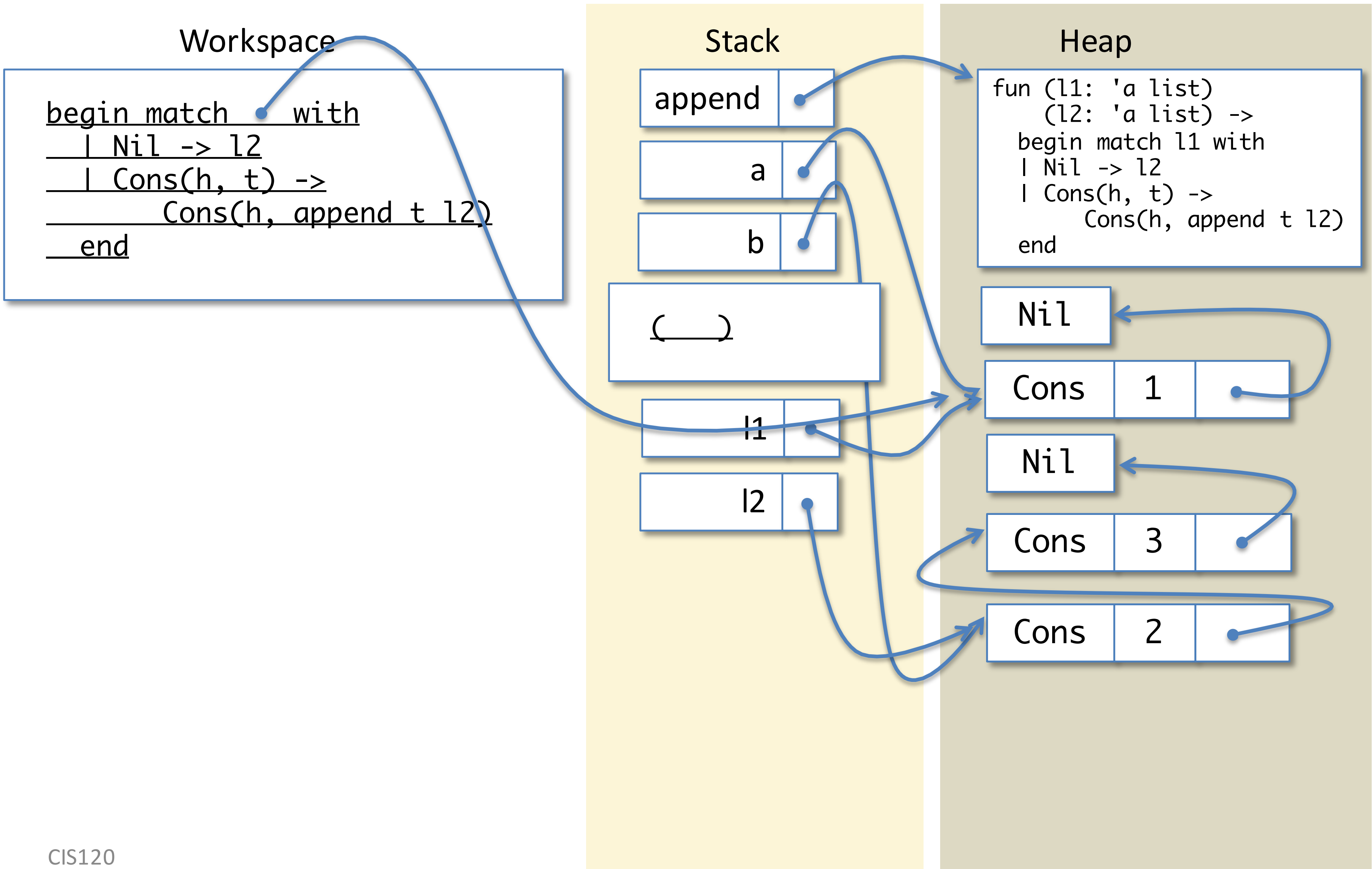


Lookup l1

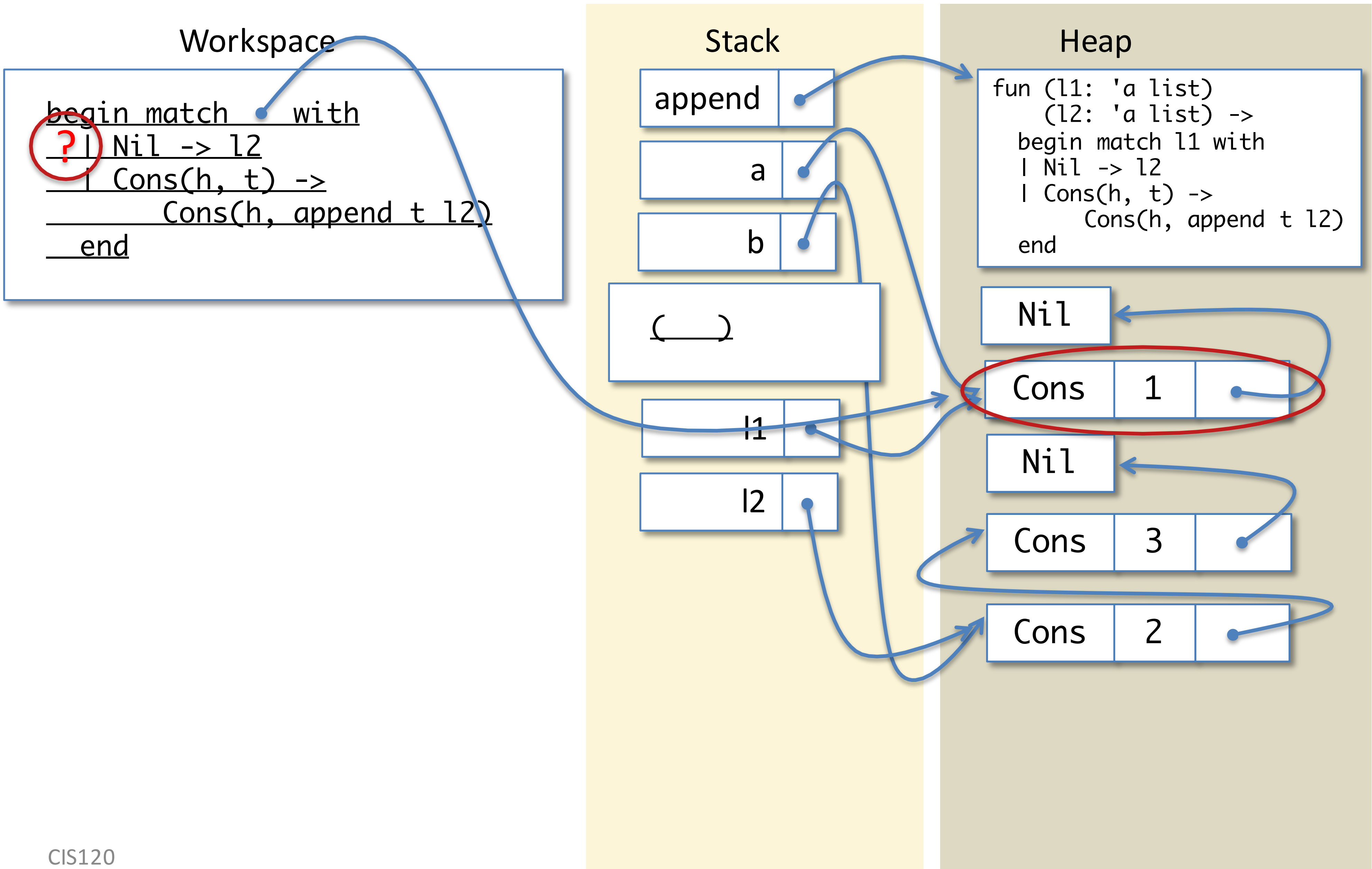


Workspace
begin match with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Stack
append
a
b
( )
l1
l2
Heap
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons
1
Nil
Cons
3
Cons
2

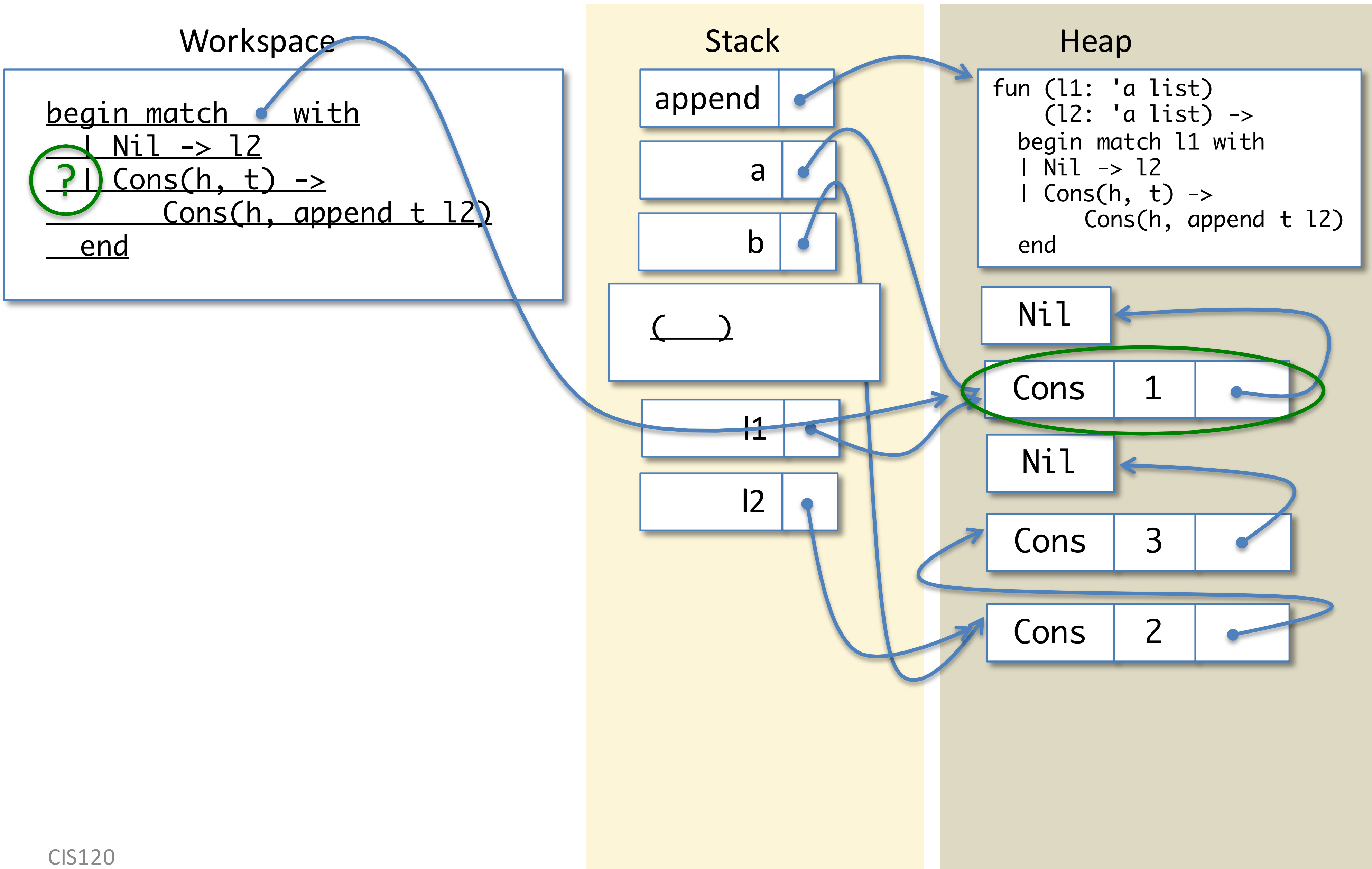
Match Expression



Nil case Doesn't Match



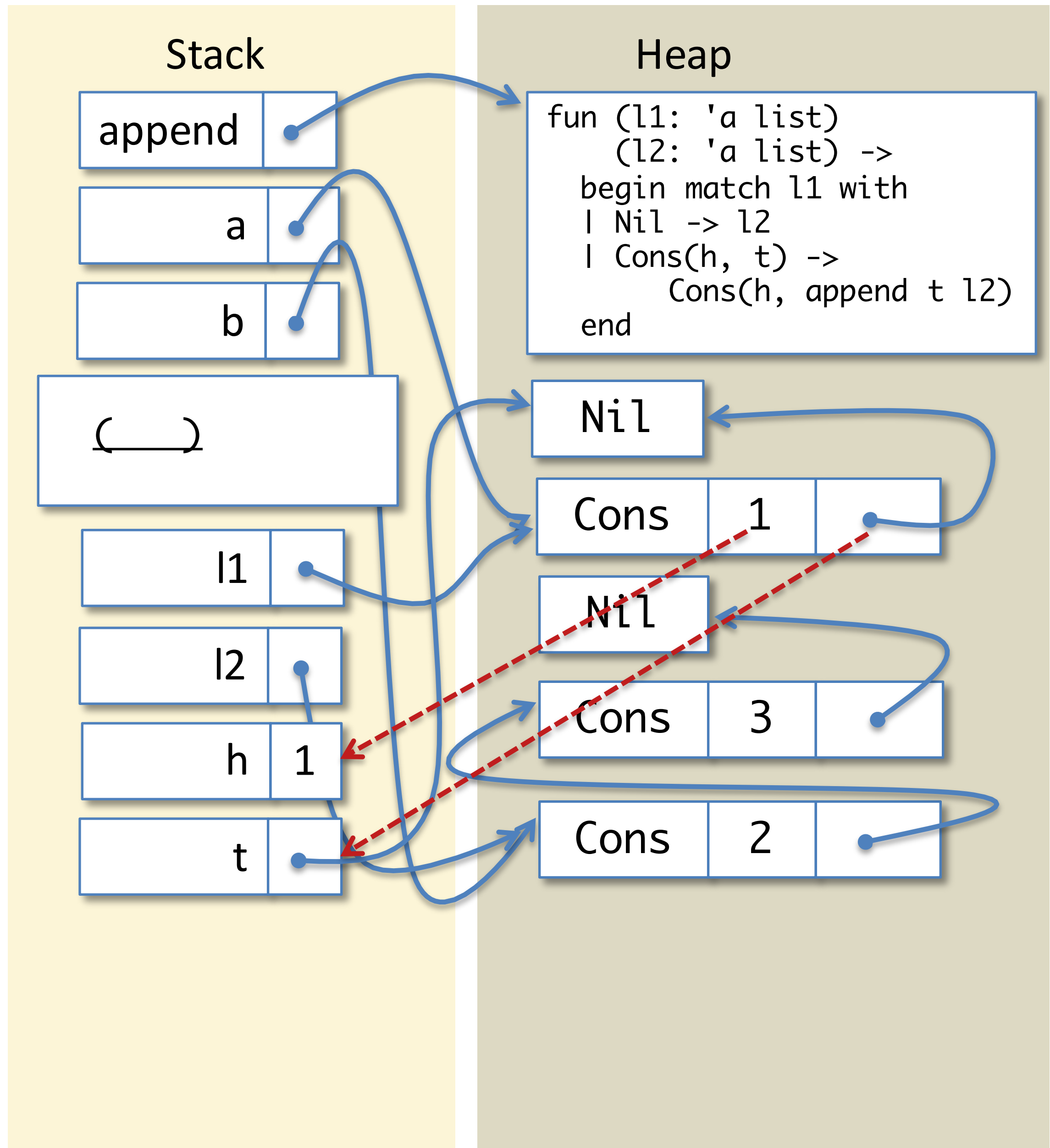
Cons case *Does* Match



Simplify the Branch: push h, t

Workspace

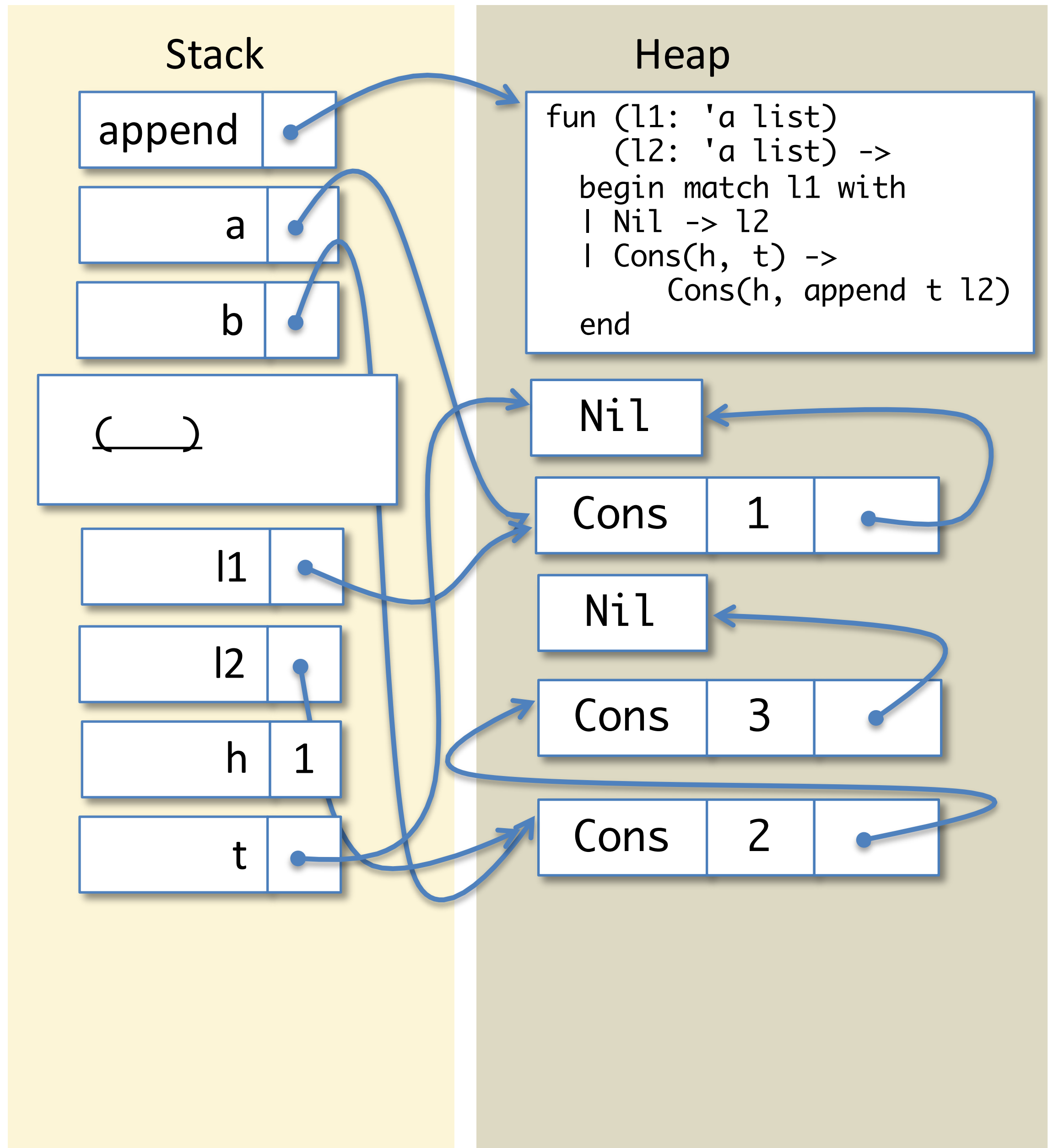
```
Cons(h, append t l2)
```



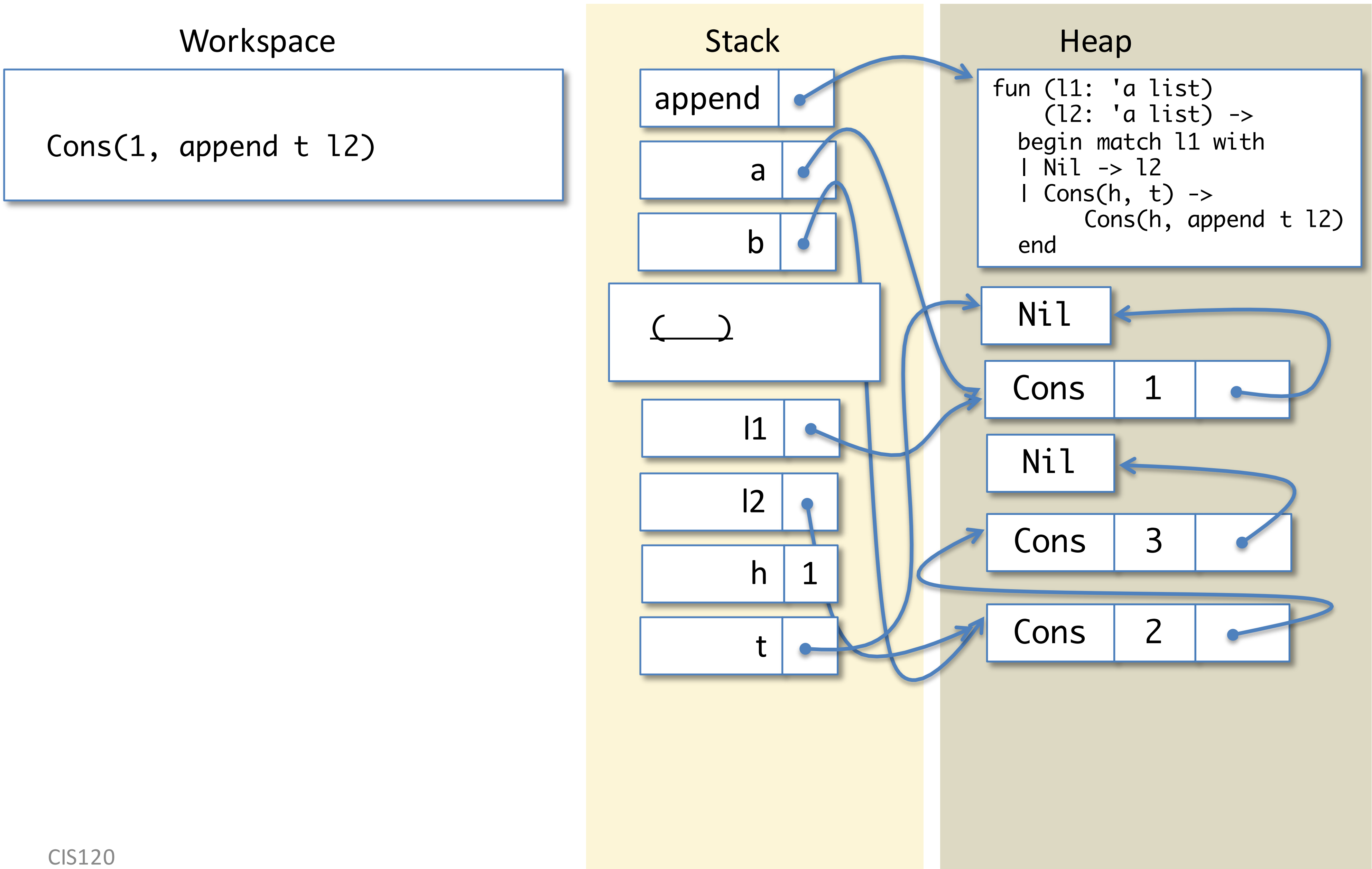
Lookup 'h'

Workspace

Cons(h, append t l2)



Lookup 'h'

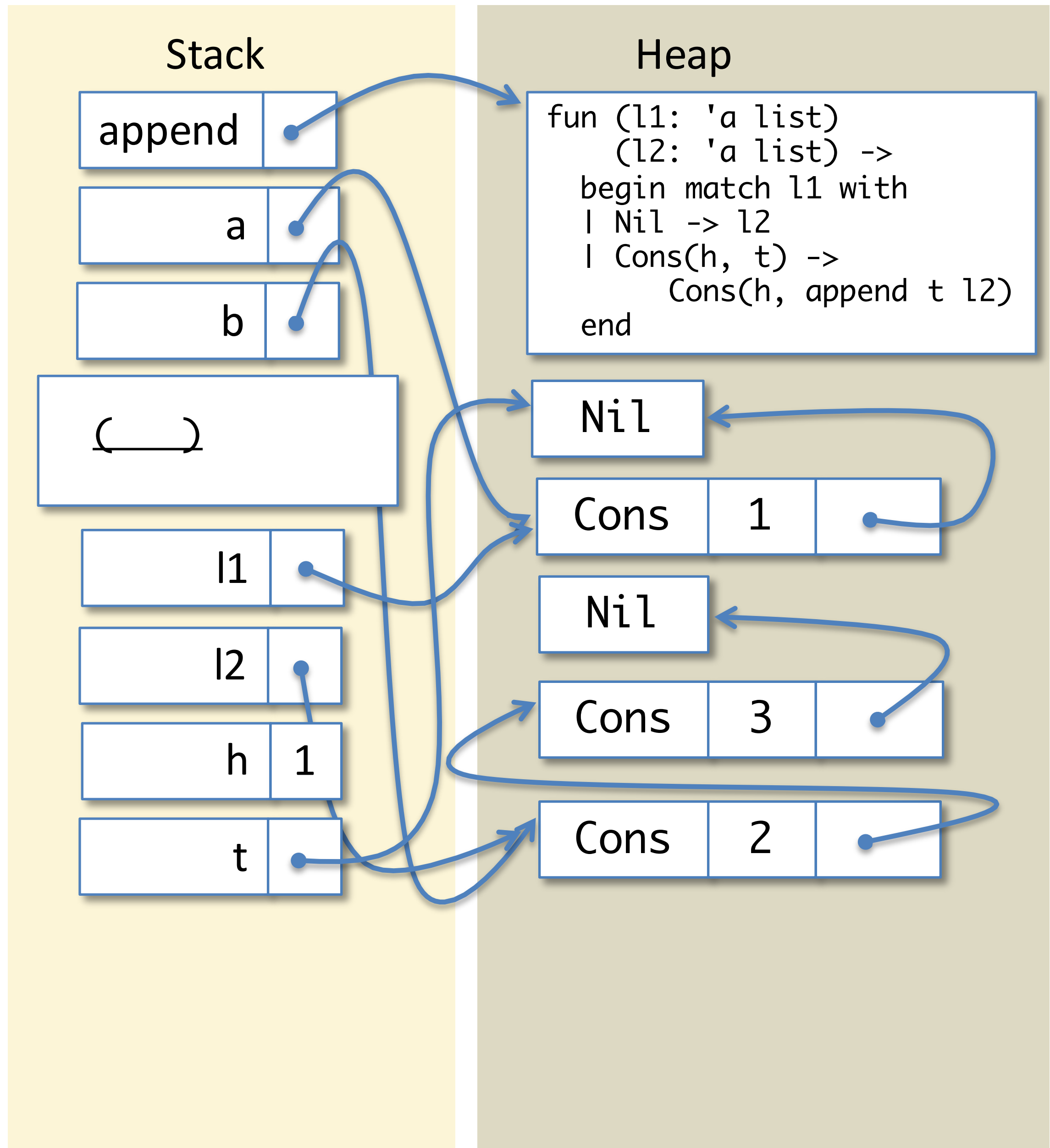


Workspace
Cons(1, append t l2)
Stack
append
a
b
(___)
l1
l2
h
1
t
Heap
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons
1
Nil
Cons
3
Cons
2

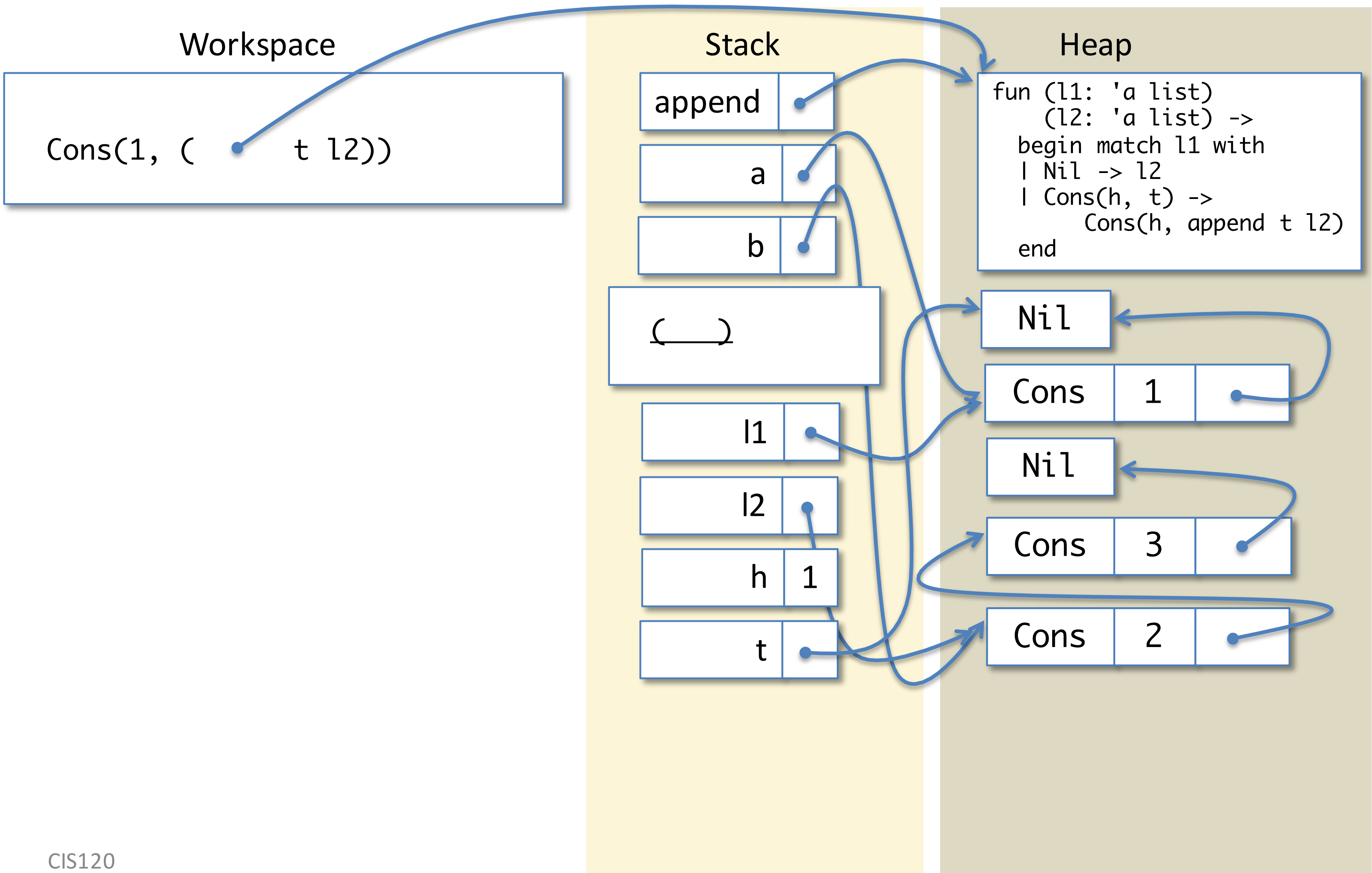
Lookup 'append'

Workspace

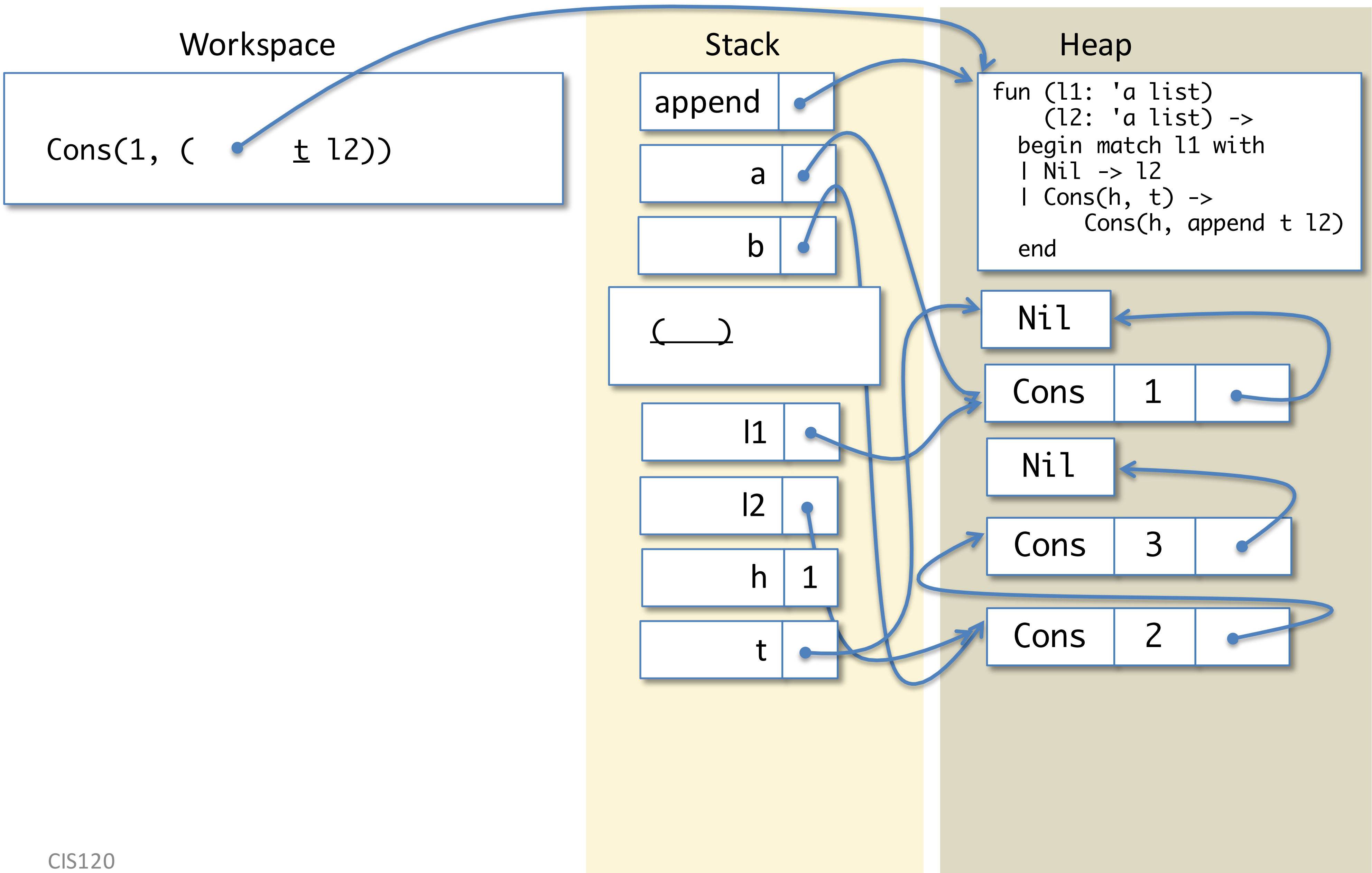
Cons(1, append t l2)



Lookup 'append'

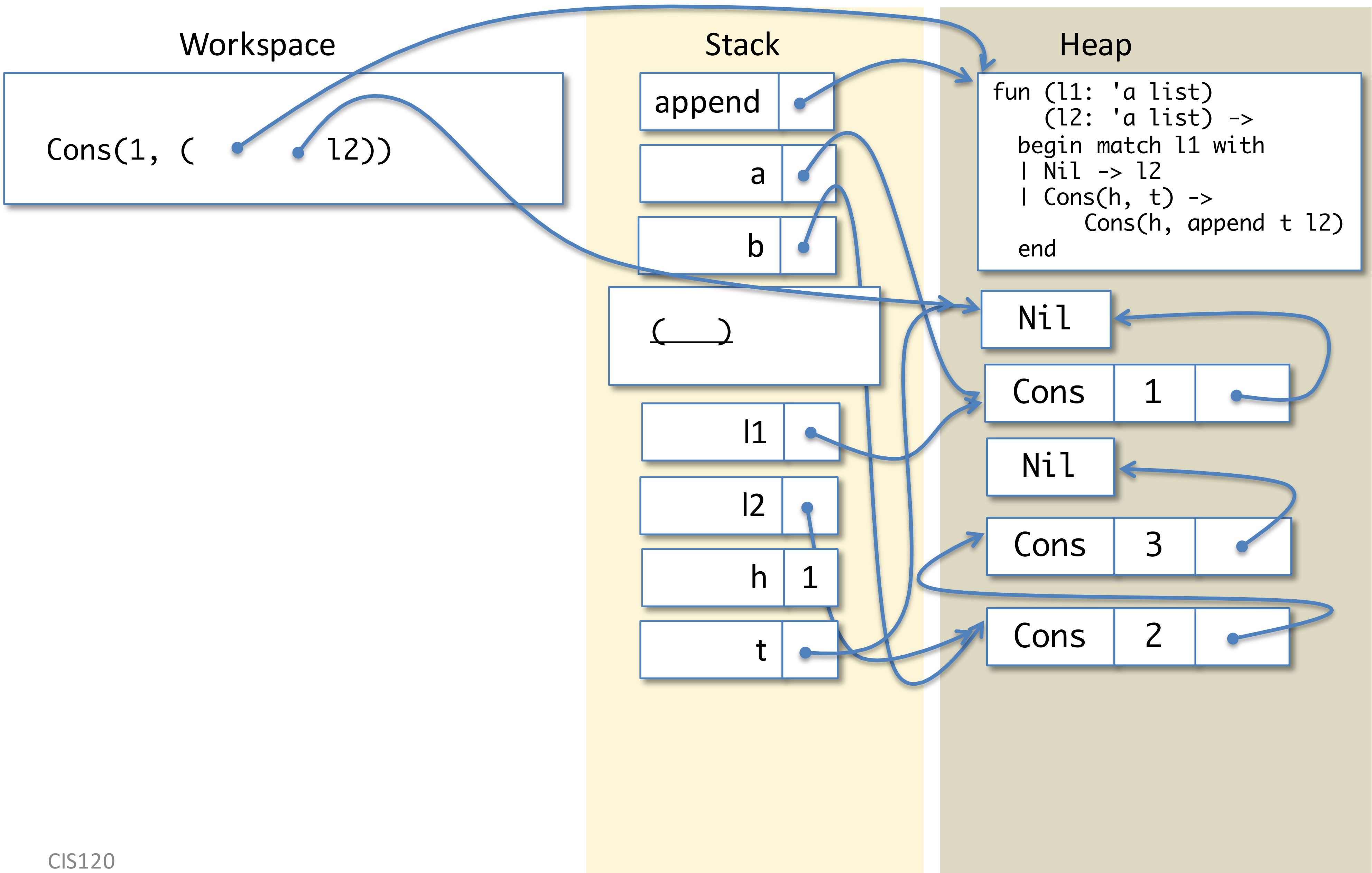


Lookup ‘t’

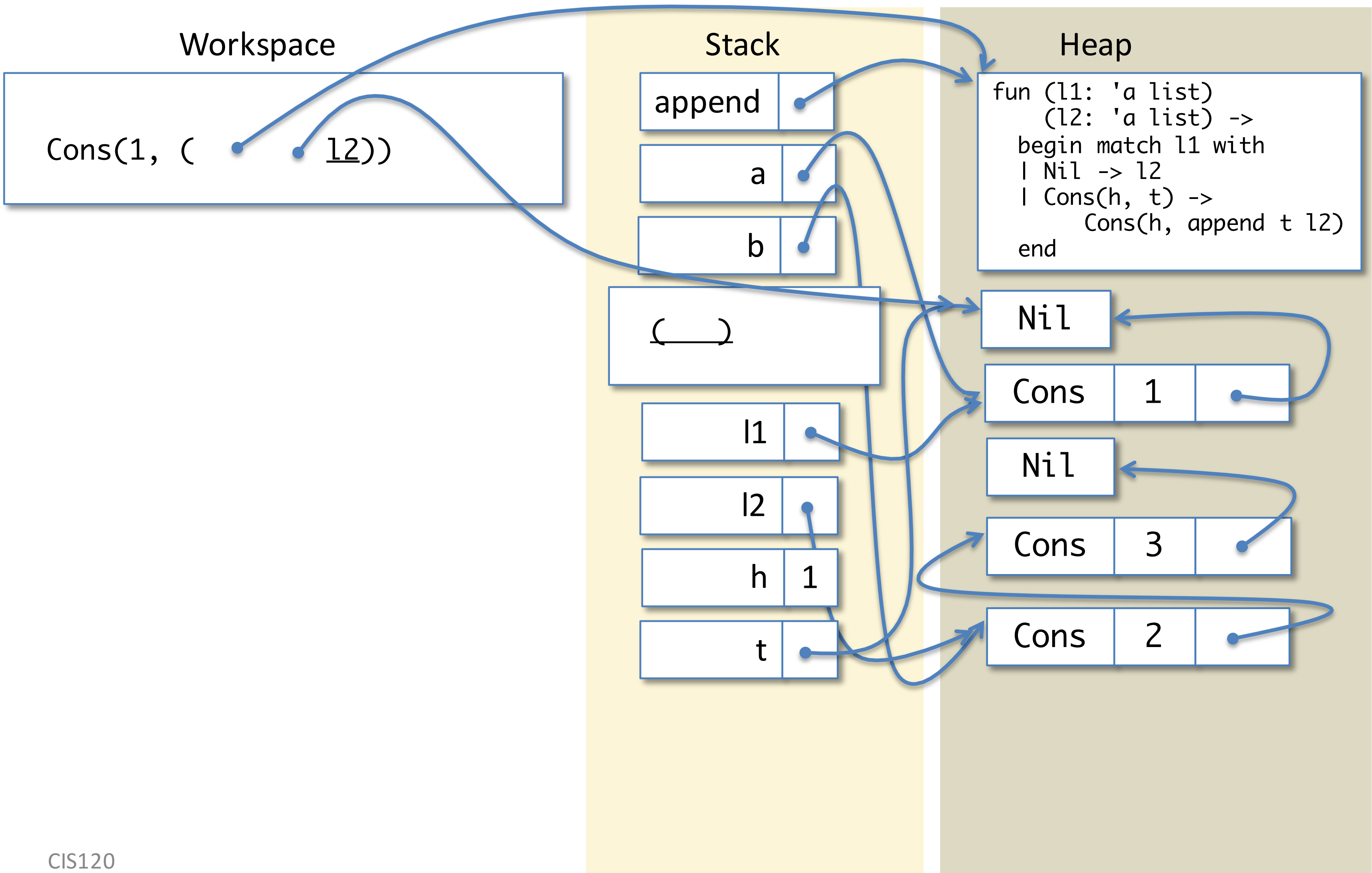


Workspace
Cons(1, (t l2))
Stack
append
a
b
( )
l1
l2
h 1
t
Heap
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

Lookup ‘t’

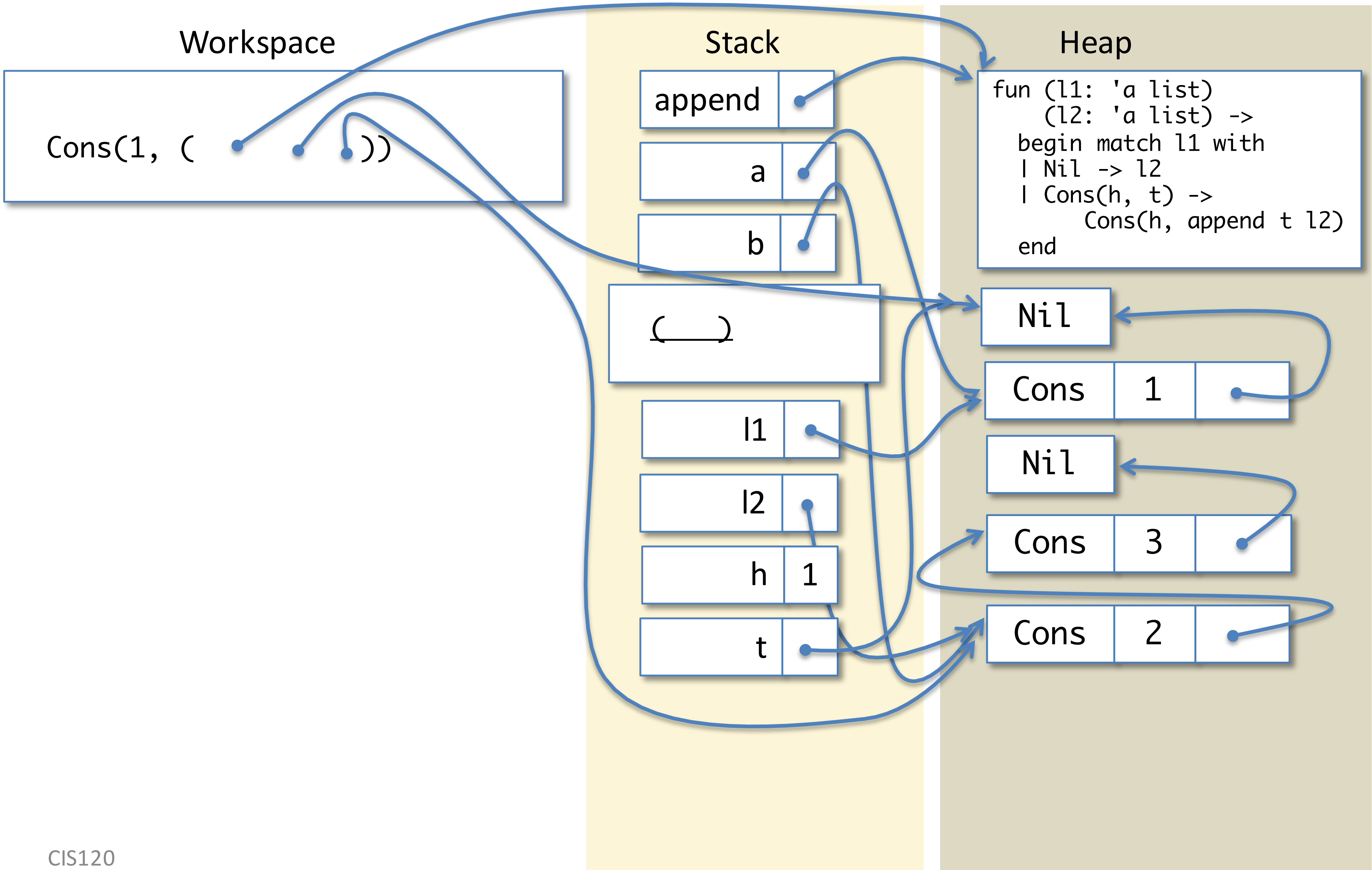

Workspace
Cons(1, (l2))
Stack
append
a
b
( )
l1
l2
h 1
t
Heap
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

Lookup ‘l2’



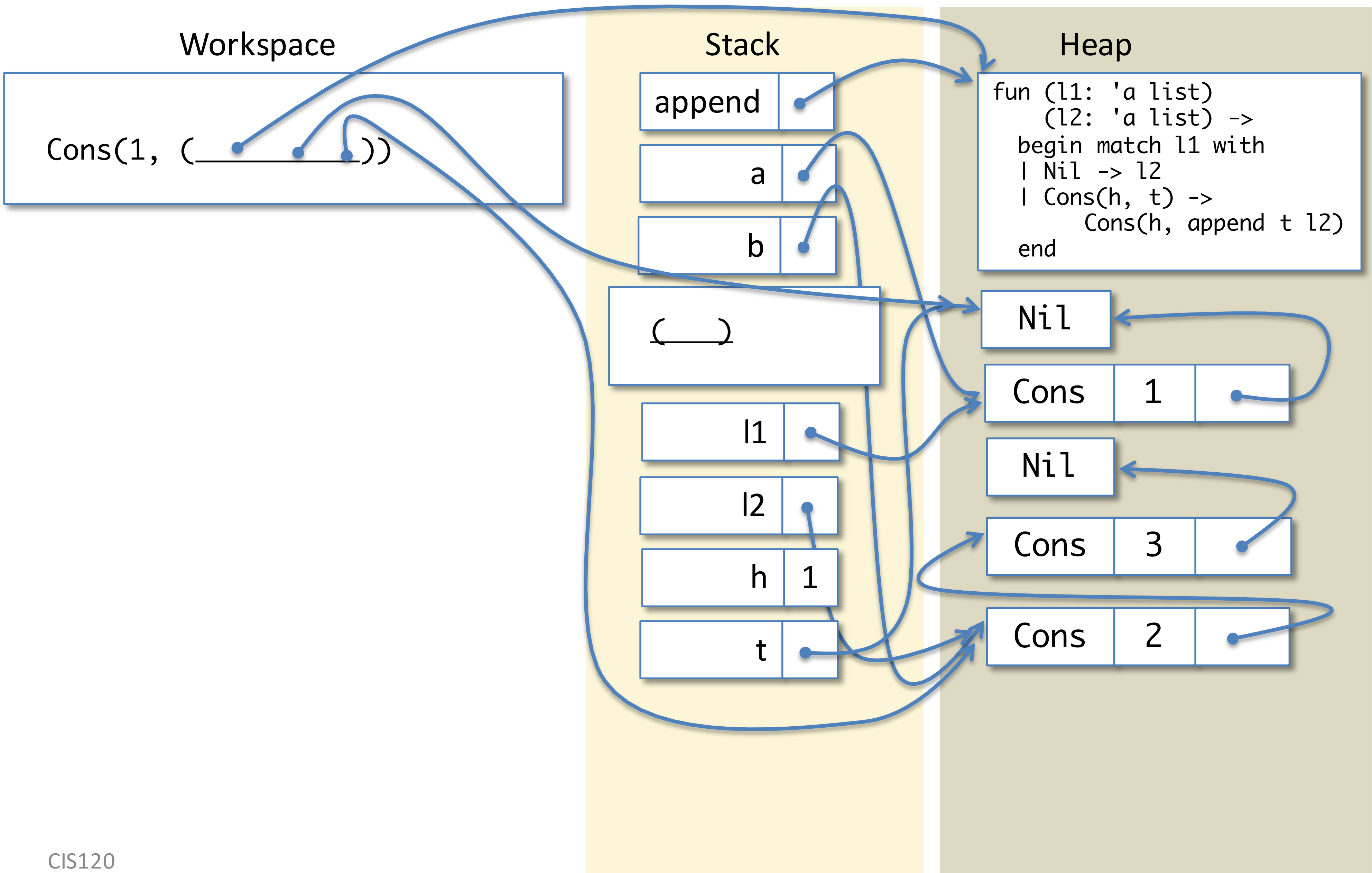
Workspace
Cons(1, (l2))
Stack
append
a
b
( )
l1
l2
h 1
t
Heap
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

Lookup 'l2'



Workspace
Stack
Heap
Cons(1, (   ))
append
a
b
(__)
l1
l2
h
1
t
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

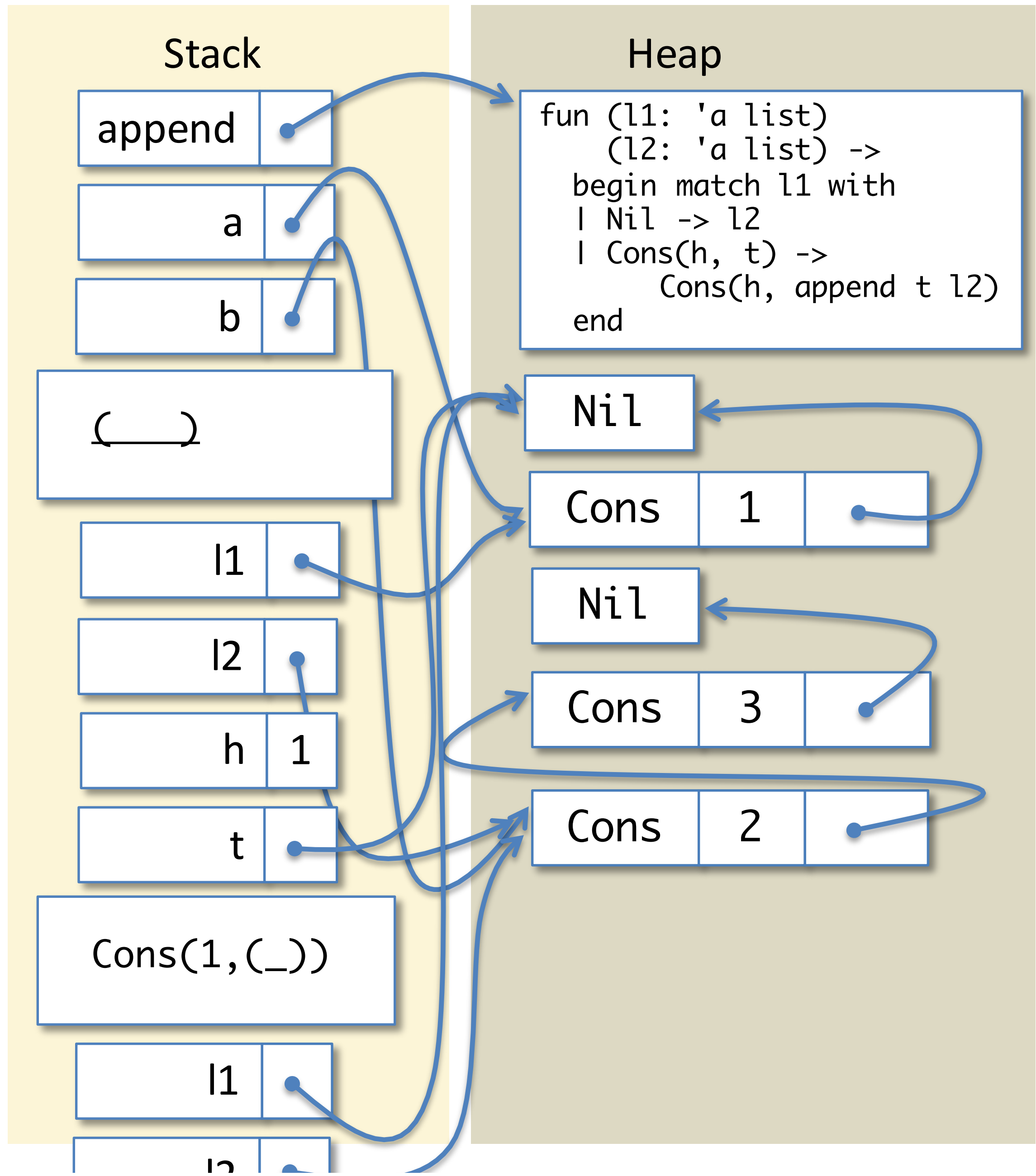
Do the Function Call



Save the Workspace; push l1, l2

Workspace

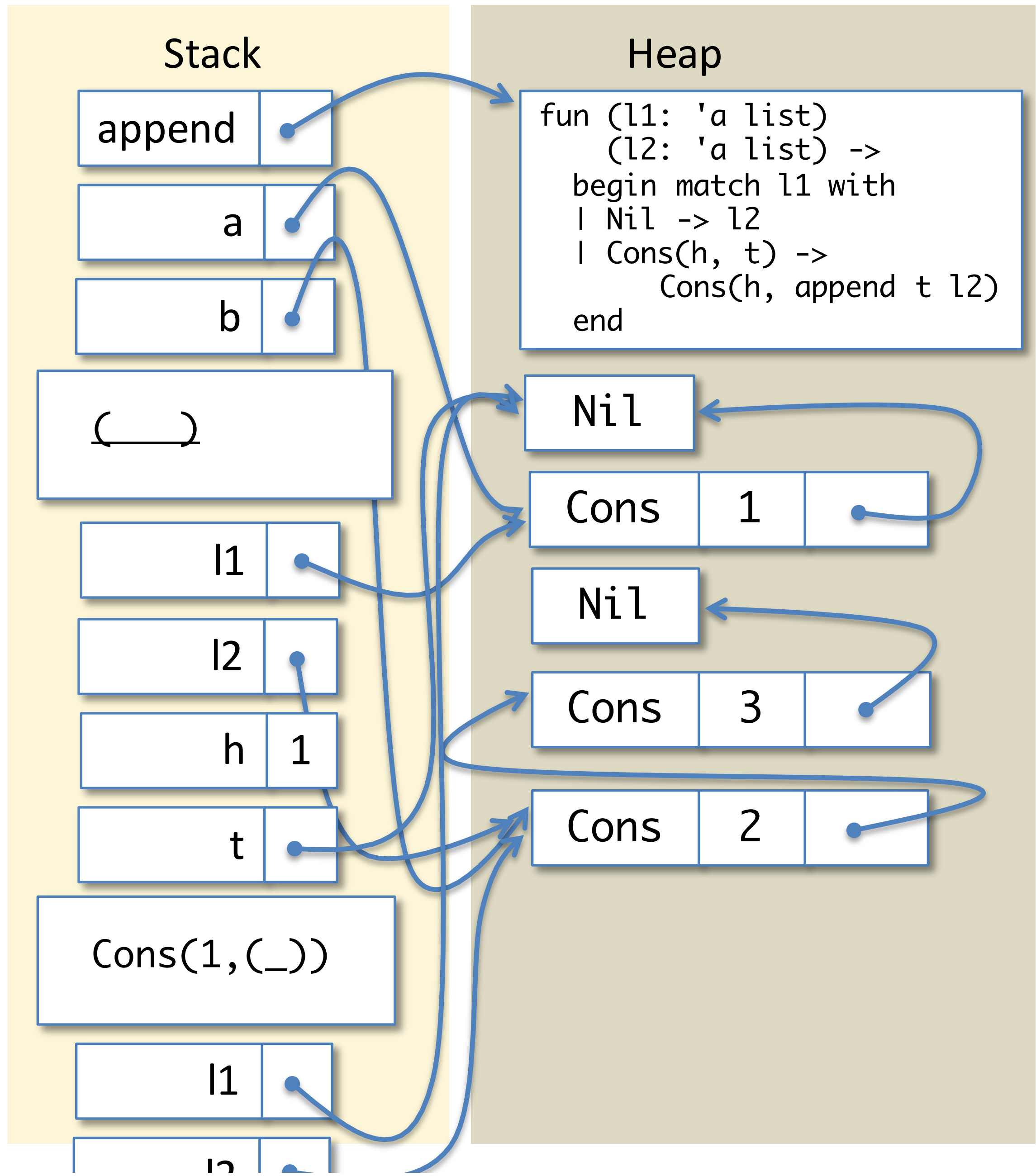
```
begin match l1 with
 | Nil -> l2
 | Cons(h, t) ->
     Cons(h, append t l2)
end
```



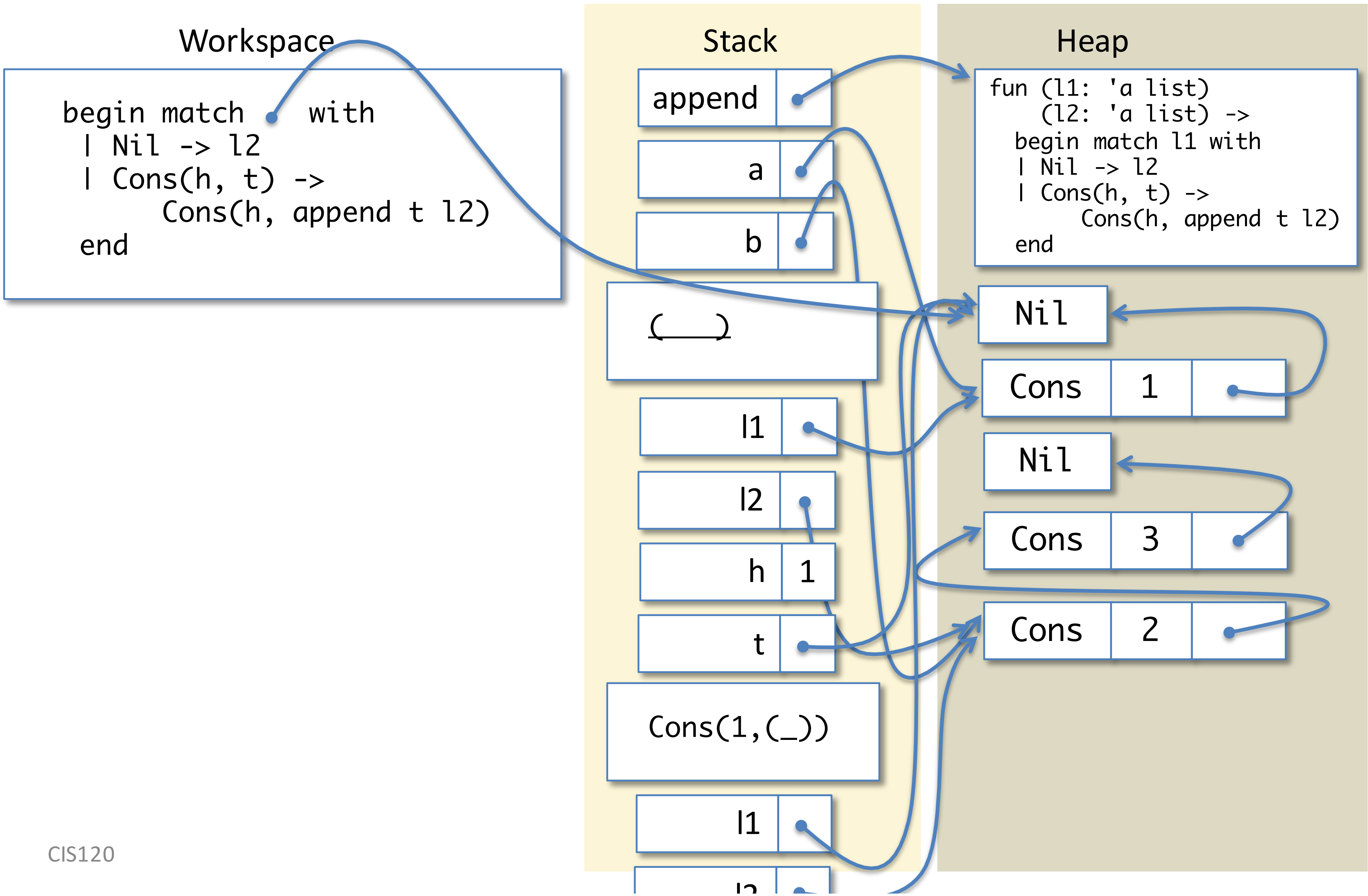
Lookup 'l1'

Workspace

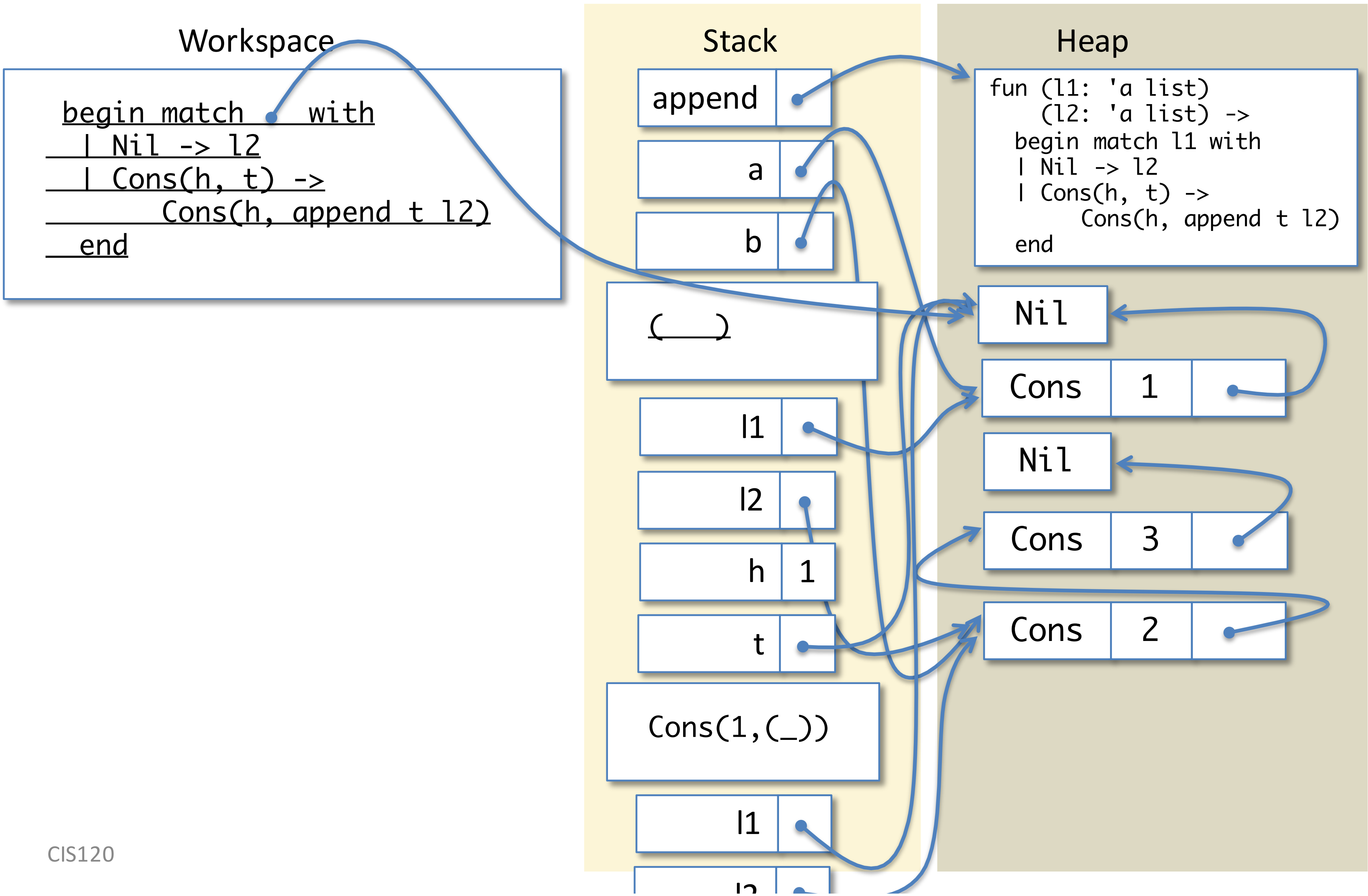
```
begin match l1 with
 | Nil -> l2
 | Cons(h, t) ->
      Cons(h, append t l2)
 end
```



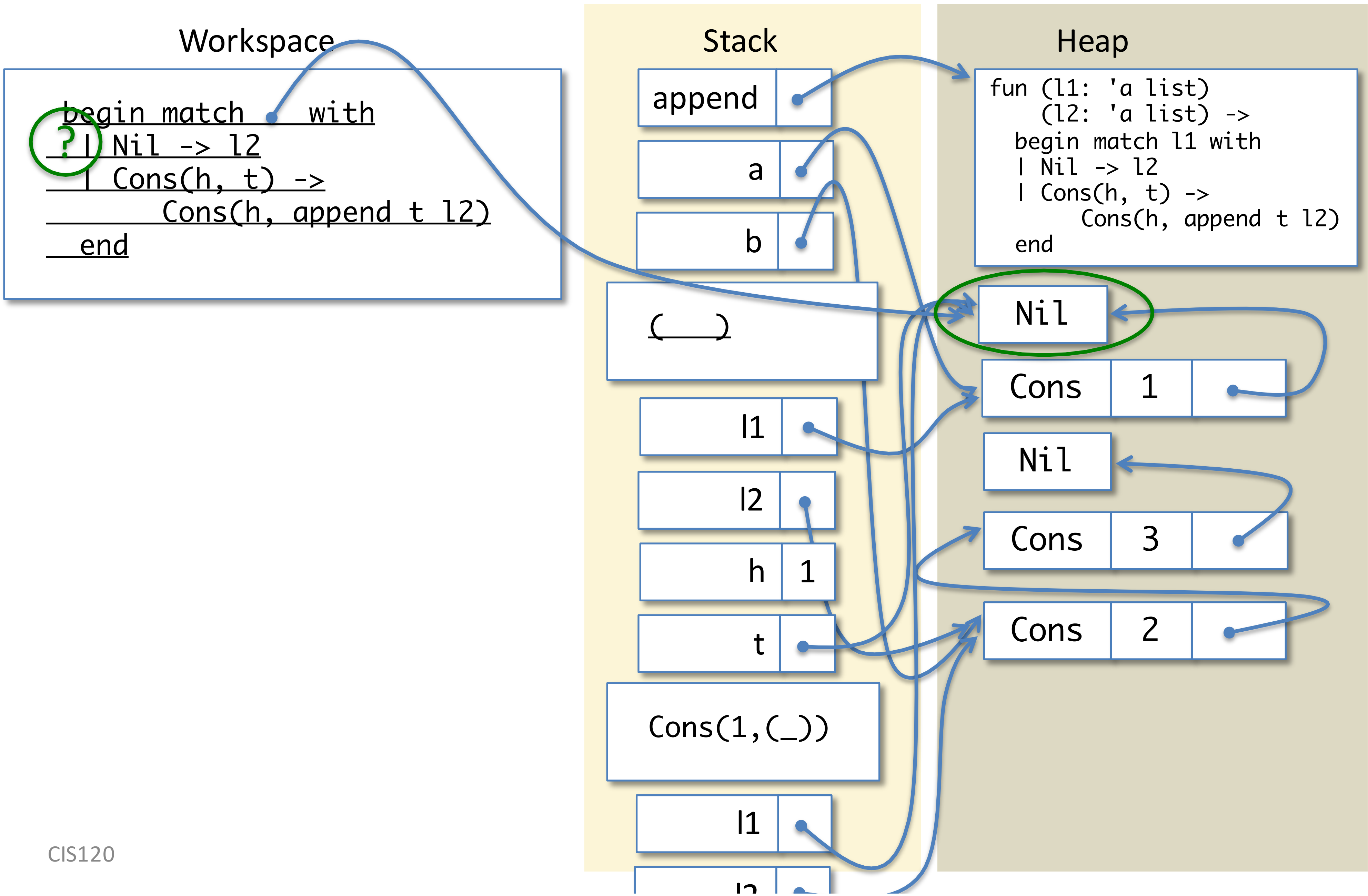
Lookup ‘l1’


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
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



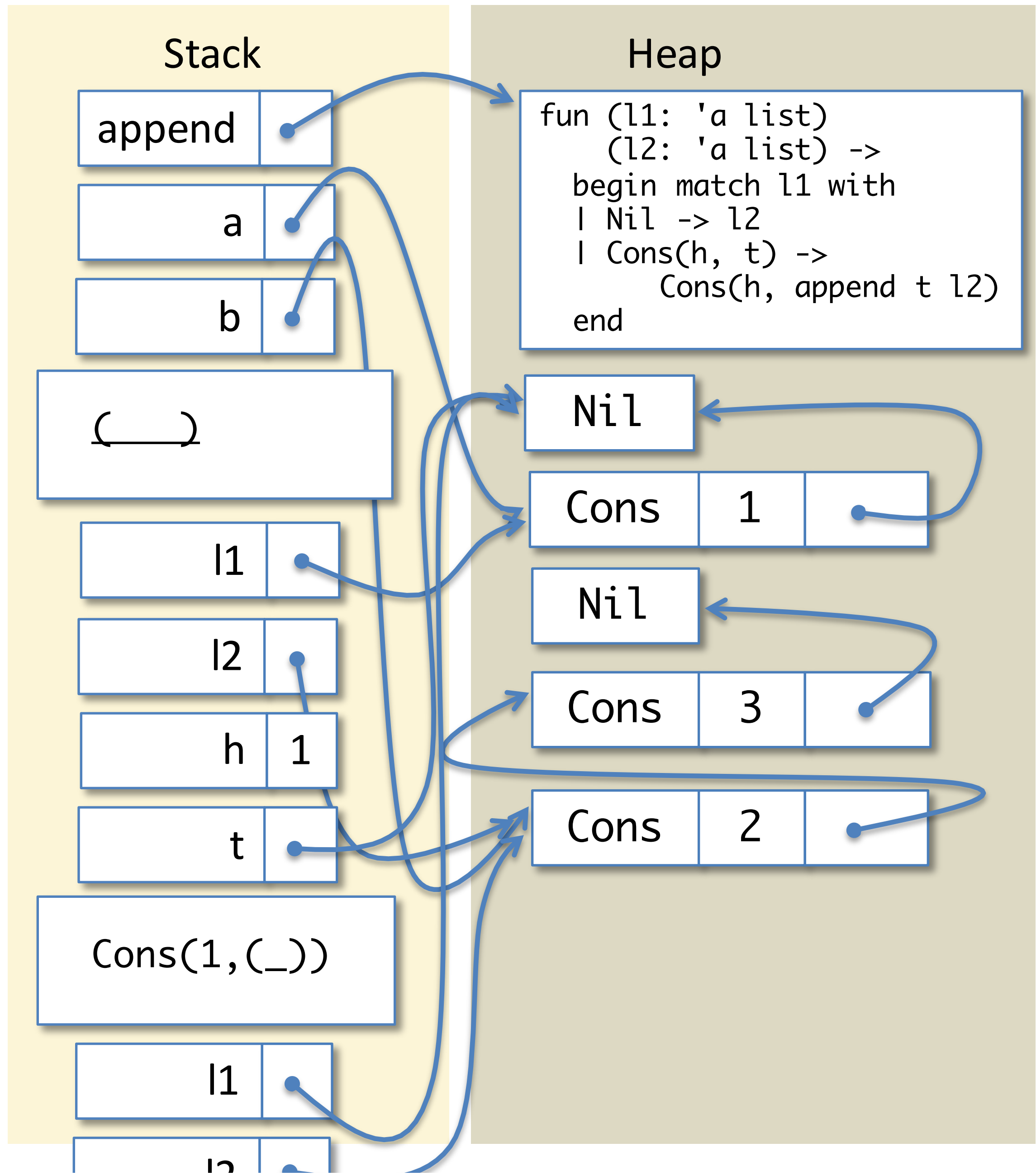
The Nil case Matches



Simplify the Branch (nothing to push)

Workspace

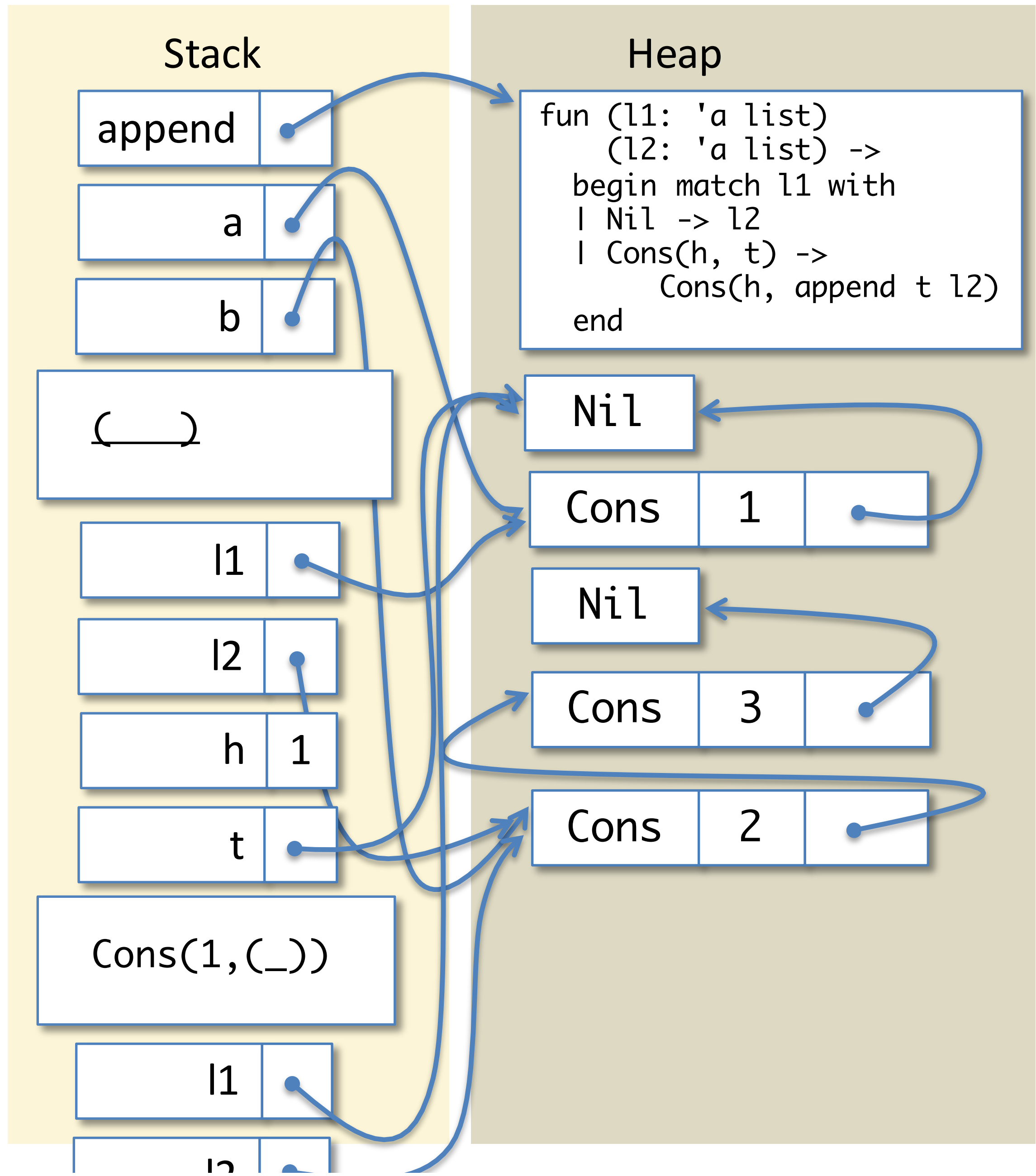
```
l2
```



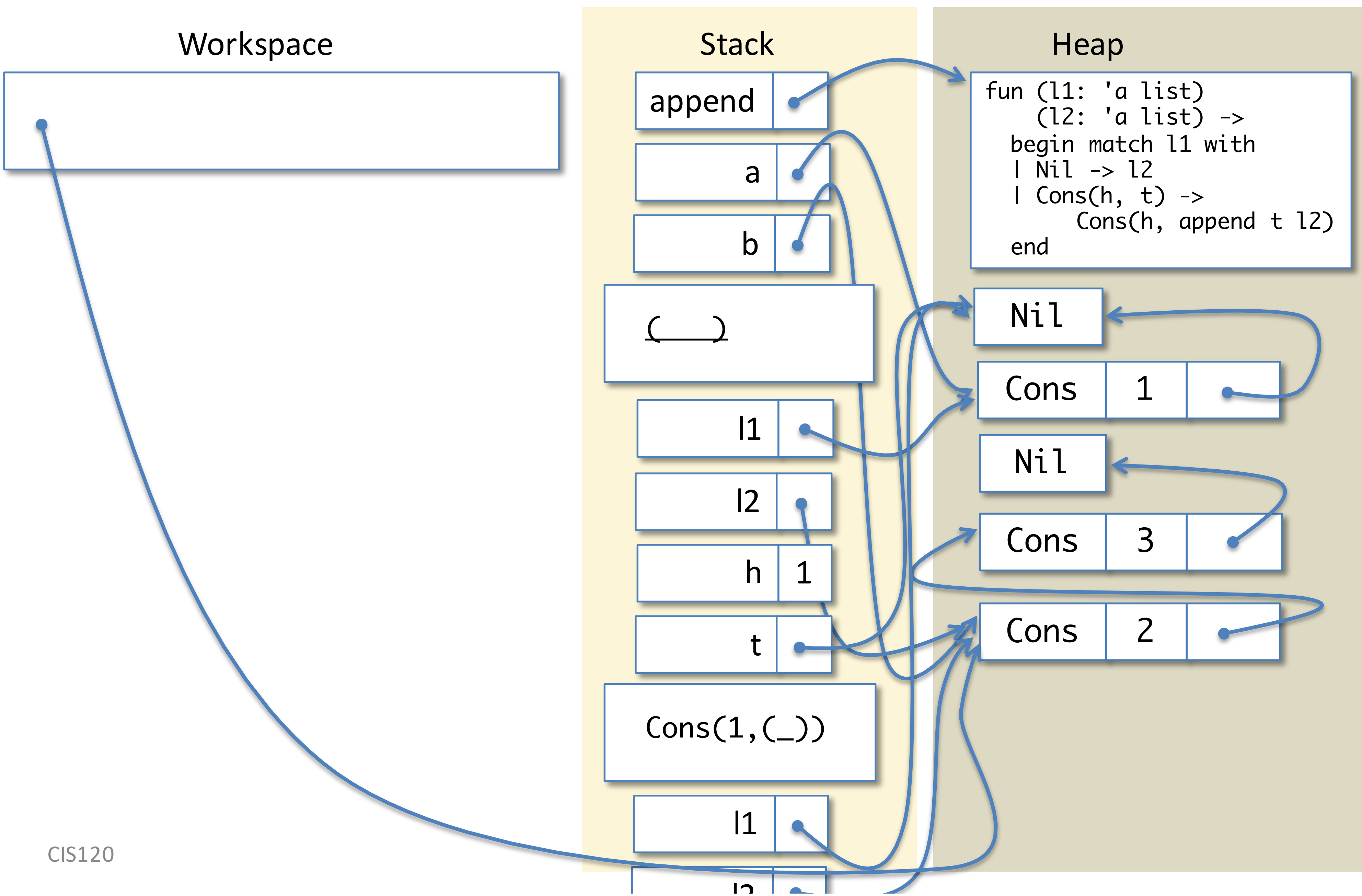
Lookup ‘l2’

Workspace

l2

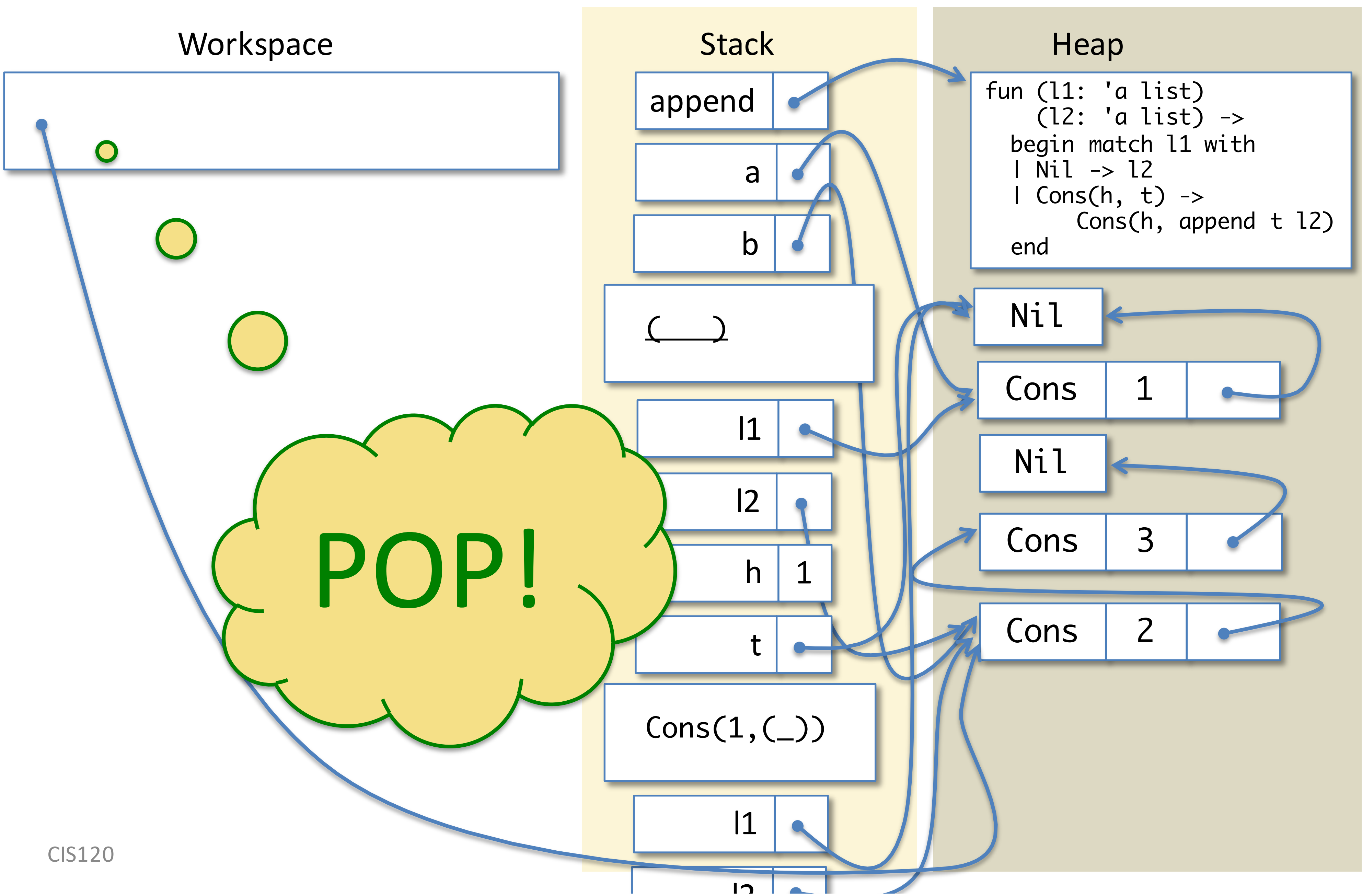


Lookup ‘l2’

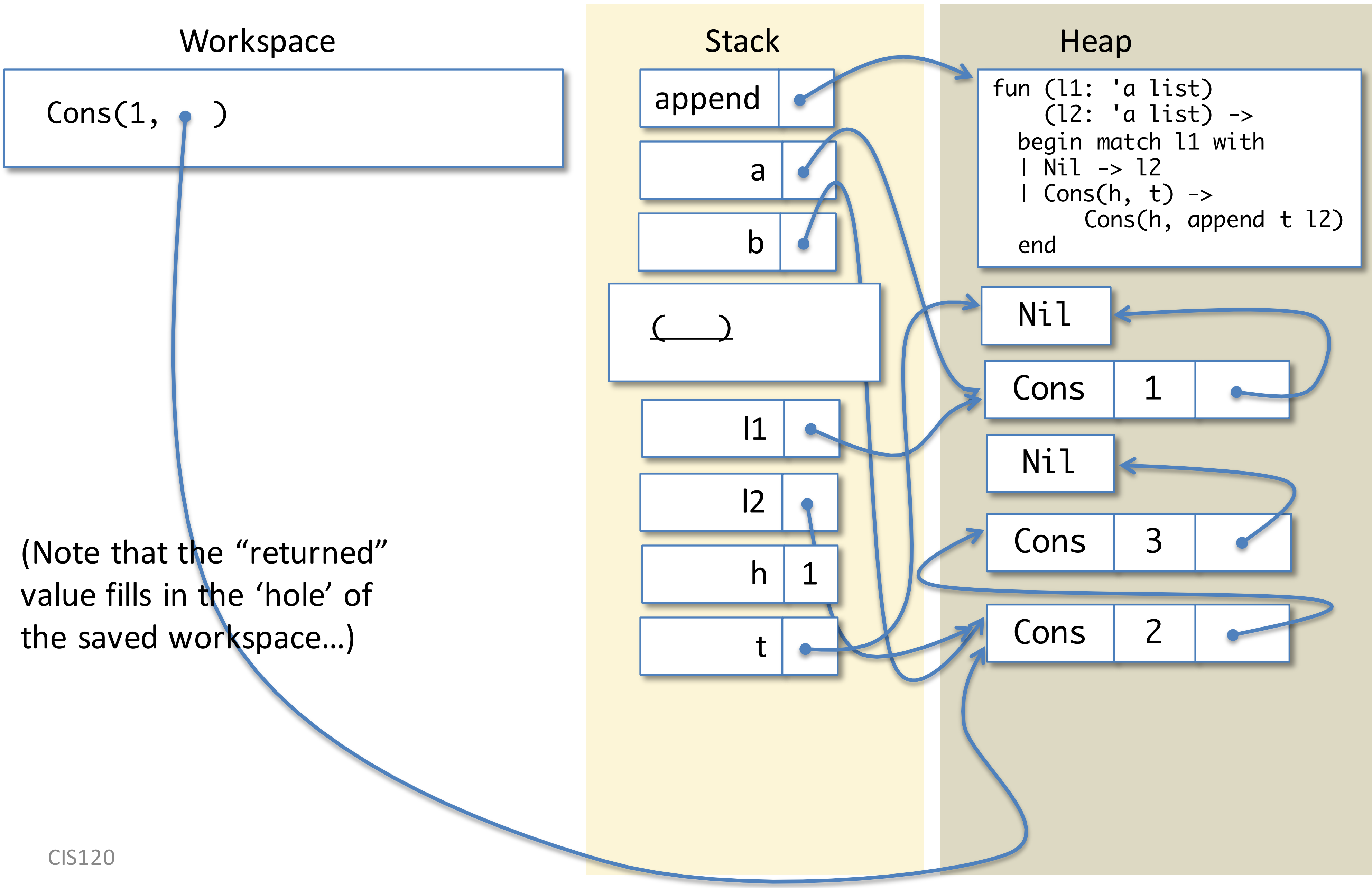


Workspace
Stack
Heap
append
a
b
(___)
l1
l2
h
1
t
Cons(1,(_))
l1
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons
1
Nil
Cons
3
Cons
2

Done! Pop stack to last Workspace

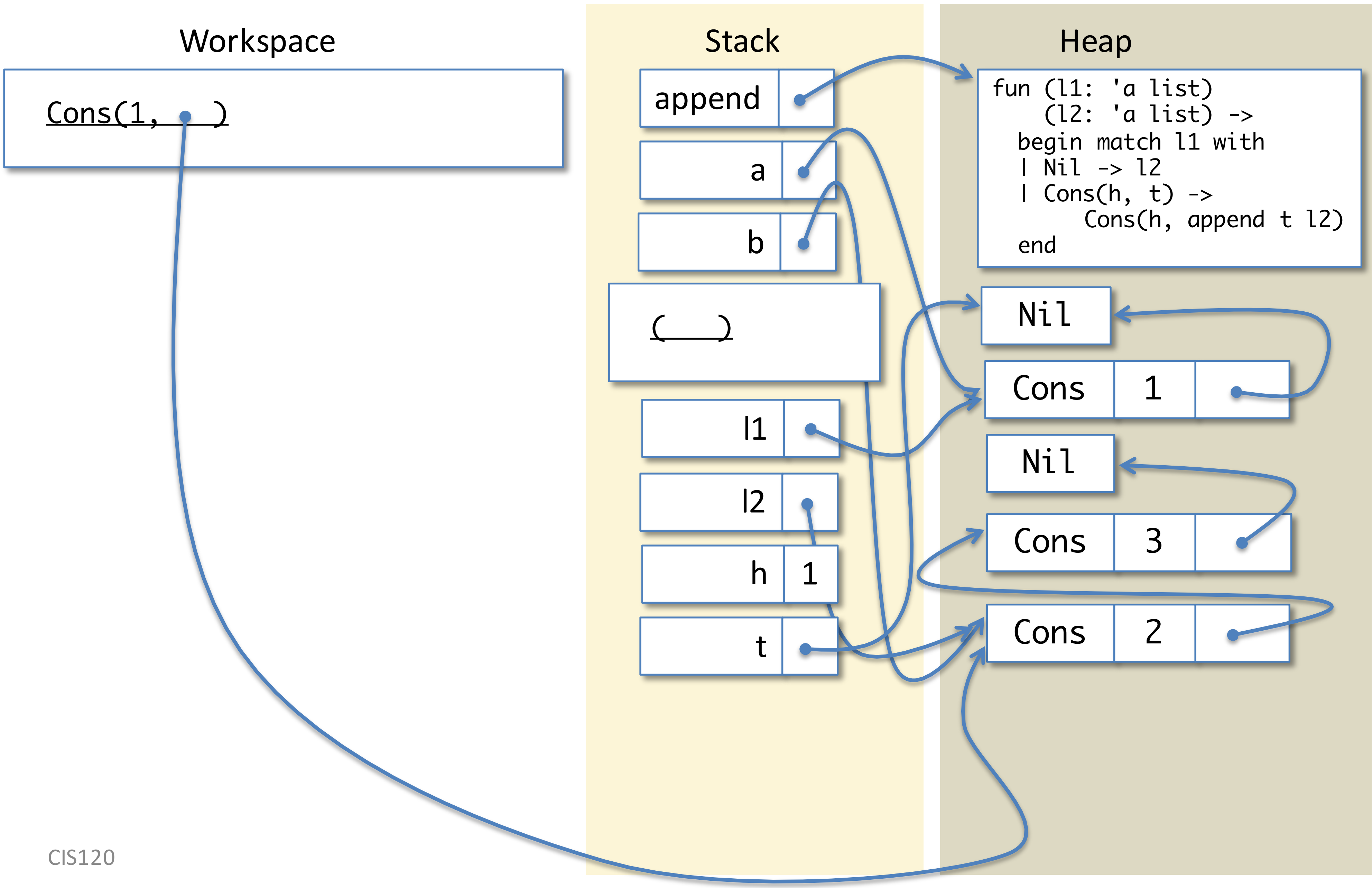


Done! Pop stack to last Workspace

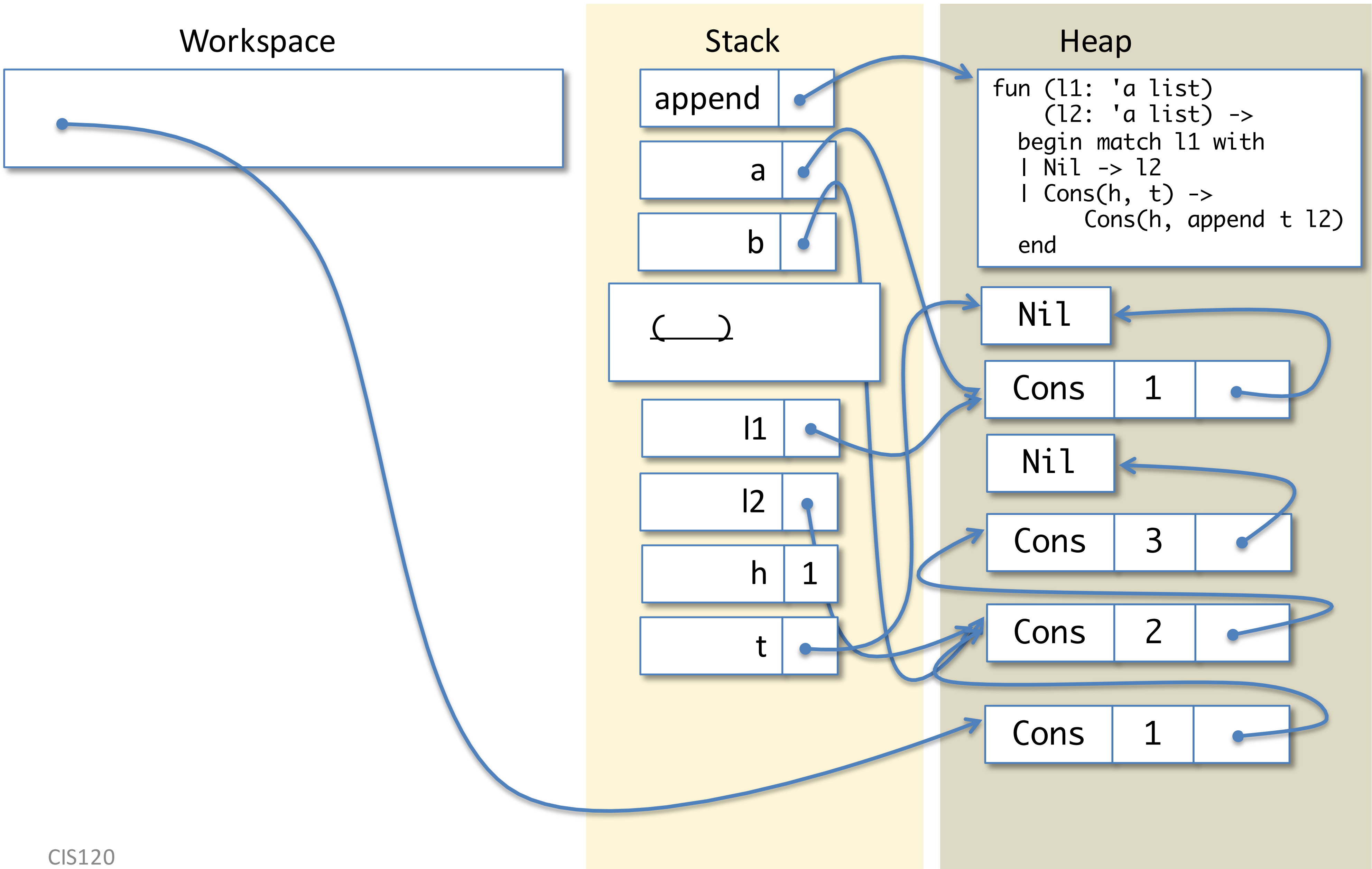


(Note that the "returned" value fills in the 'hole' of the saved workspace…)

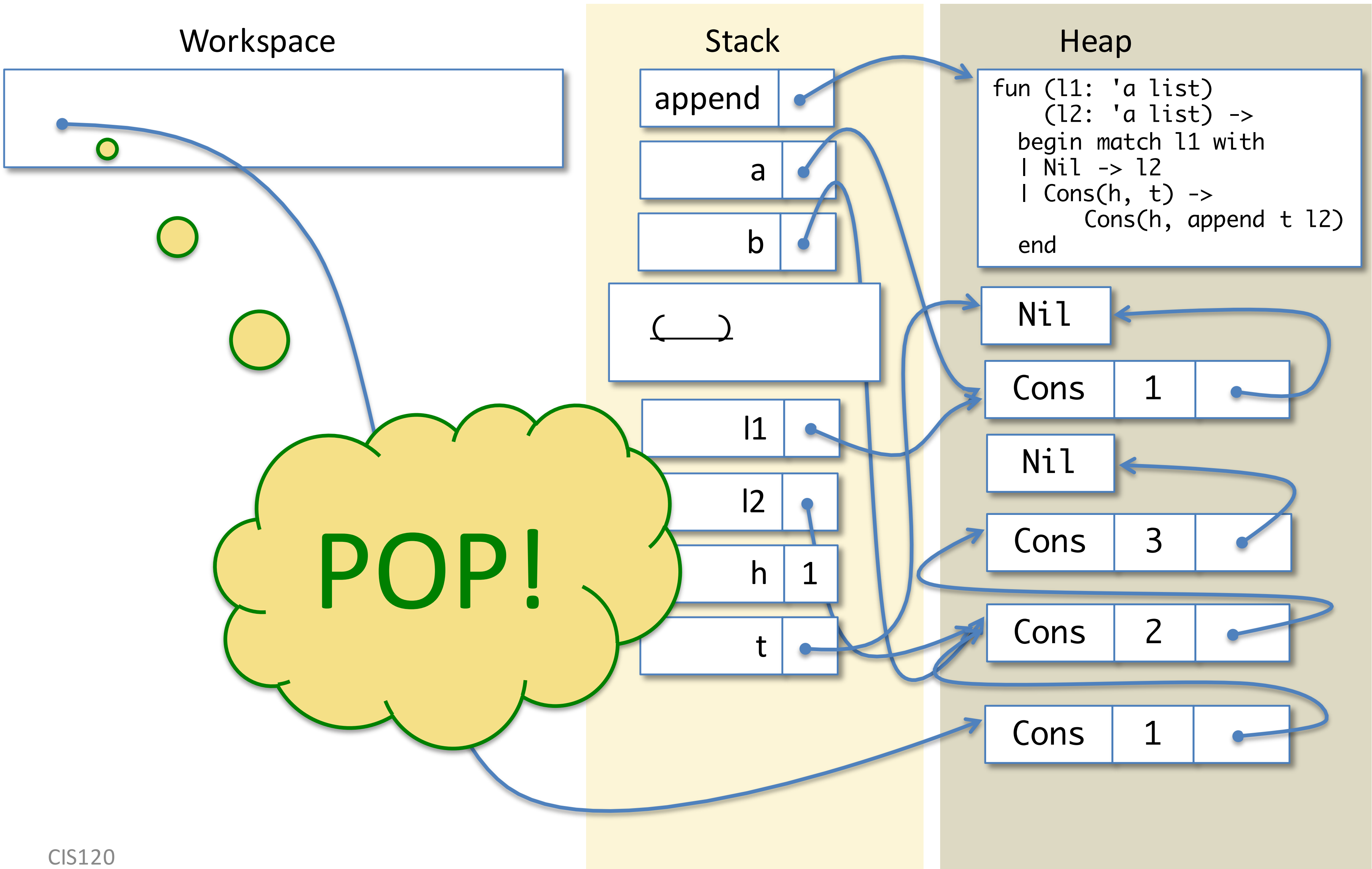
Allocate a Cons cell



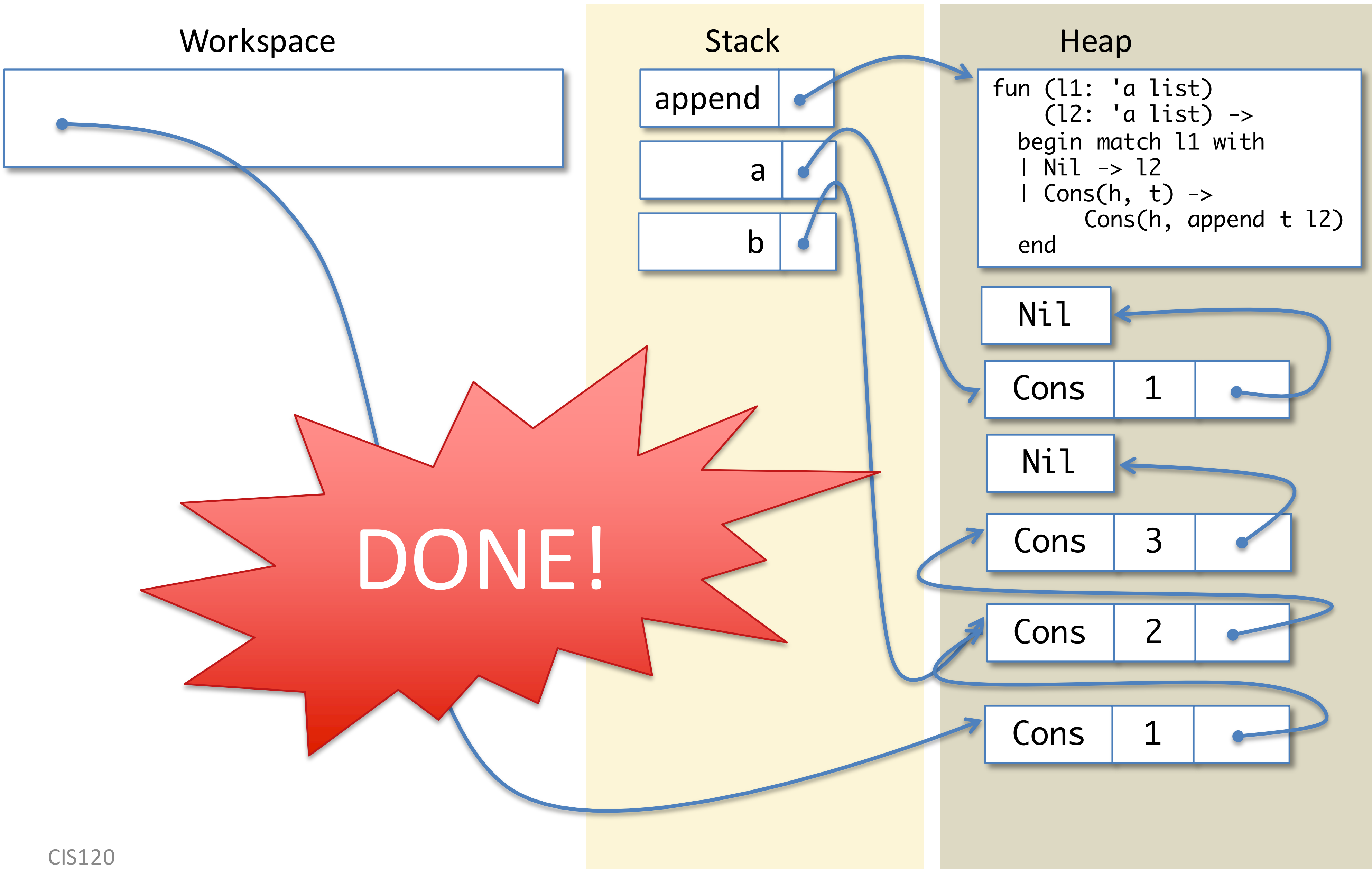
Allocate a Cons cell



Done! Pop stack to last Workspace

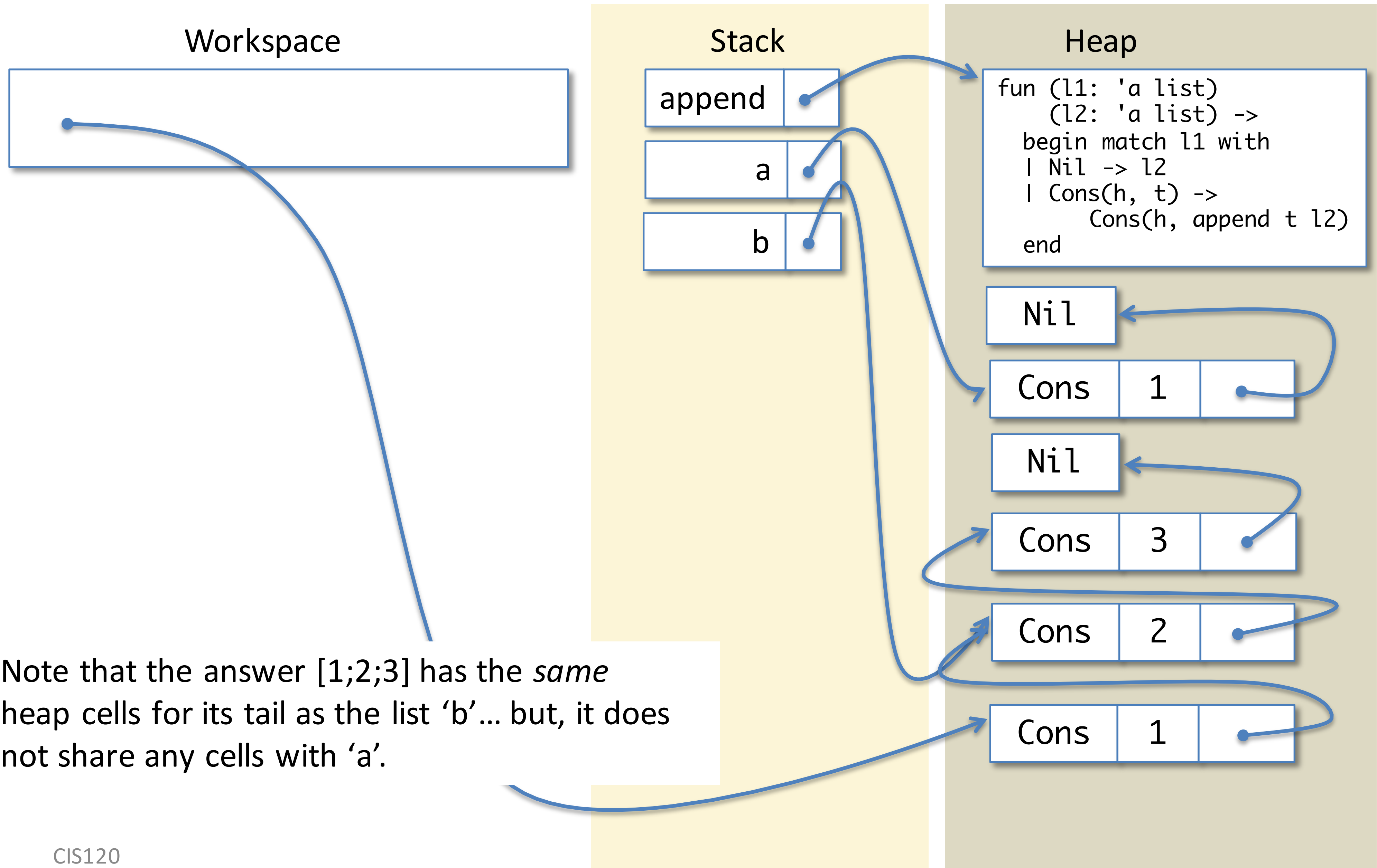


Done! (PHEW!)



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 $\texttt{pat}_i$ 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 $\texttt{branch}_i$

Did you attend class today?

1. Yes