Programming Languages and Techniques (CIS1200)

Lecture 14

Mutable State, Aliasing, and the Abstract Stack Machine

Chapters 14 and 15

Announcements

- If you have not watched lecture 13, please do so ASAP
- HW04 (mutable queues) available
  - Due in one week (next Tuesday)

Review: Options

Example: list_max

A function that returns the maximum value of a list as an option

- Returns None if the list is empty

```
let list_max (l:'a list) : 'a option =
  begin match l with
    | [] -> None
    | x::tl -> Some (fold max x tl)
  end
```

Option Types

- Define a generic datatype of *optional values*:

```
type 'a option =
  | None
  | Some of 'a
```

- A “partial” function returns an option

```
let list_max (l:'a list) : 'a option = …
```

- Safer than “null” (a legal value of *any type* in Java) or "None" in Python
  - Caller must pattern match to access the value

- Modern language designs (e.g. Apple's Swift, Mozilla's Rust) distinguish between the types String (definitely not null) and String? (optional string)

Review: Records

Records

Records are like tuples with named fields:

```
(* a type for representing colors *)
type rgb = {r:int; g:int; b:int}

(* some example rgb values *)
let red   : rgb = {r=255; g=0;   b=0}
let blue  : rgb = {r=0;   g=0;   b=255}
let green : rgb = {r=0;   g=255; b=0}
let black : rgb = {r=0;   g=0;   b=0}
let white : rgb = {r=255; g=255; b=255}
```

Curly braces around record. Semicolons between record components.

- The type rgb is a record with three fields: r, g, and b
  - fields can have any types; they don't all have to be the same
- Record values are created using this notation:

```
        {field1=val1; field2=val2;…}
```

Field Projection

- The value in a record field can be obtained by using “dot” notation: `record.field`

```
(* a type for representing colors *)
type rgb = {r:int; g:int; b:int}

(* using 'dot' notation to project out components,
   calculate the average of two colors… *)
let average_rgb (c1:rgb) (c2:rgb) : rgb =
  {r = (c1.r + c2.r) / 2;
   g = (c1.g + c2.g) / 2;
   b = (c1.b + c2.b) / 2}
```

Review: Mutable State

Records

- By default, all record fields are *immutable*—once initialized, they can never be modified.

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

let p0 = {x=0; y=0}
;; do_something_with p0
;; print_endline ("p0.x = " ^ (string_of_int p0.x))

let p1 = {x=(p0.x + 1); y=(p0.y + 1)}
;; do_something_with p1
;; print_endline ("p1.x = " ^ (string_of_int p1.x))
```

This will always be 0, no matter what "do_something_with" does

This will always be 1…

Mutable Record Fields

- By default, all record fields are *immutable*—once initialized, they can never be modified.
- OCaml also supports *mutable* fields that can be imperatively updated by the “set” command: `record.field <- val`

note the ‘mutable’ keyword

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

let p0 = {x=0; y=0}
(* set the x coord of p0 to 17 *)
;; p0.x <- 17
;; print_endline ("p0.x = " ^ (string_of_int p0.x))

p0.x = 17
```

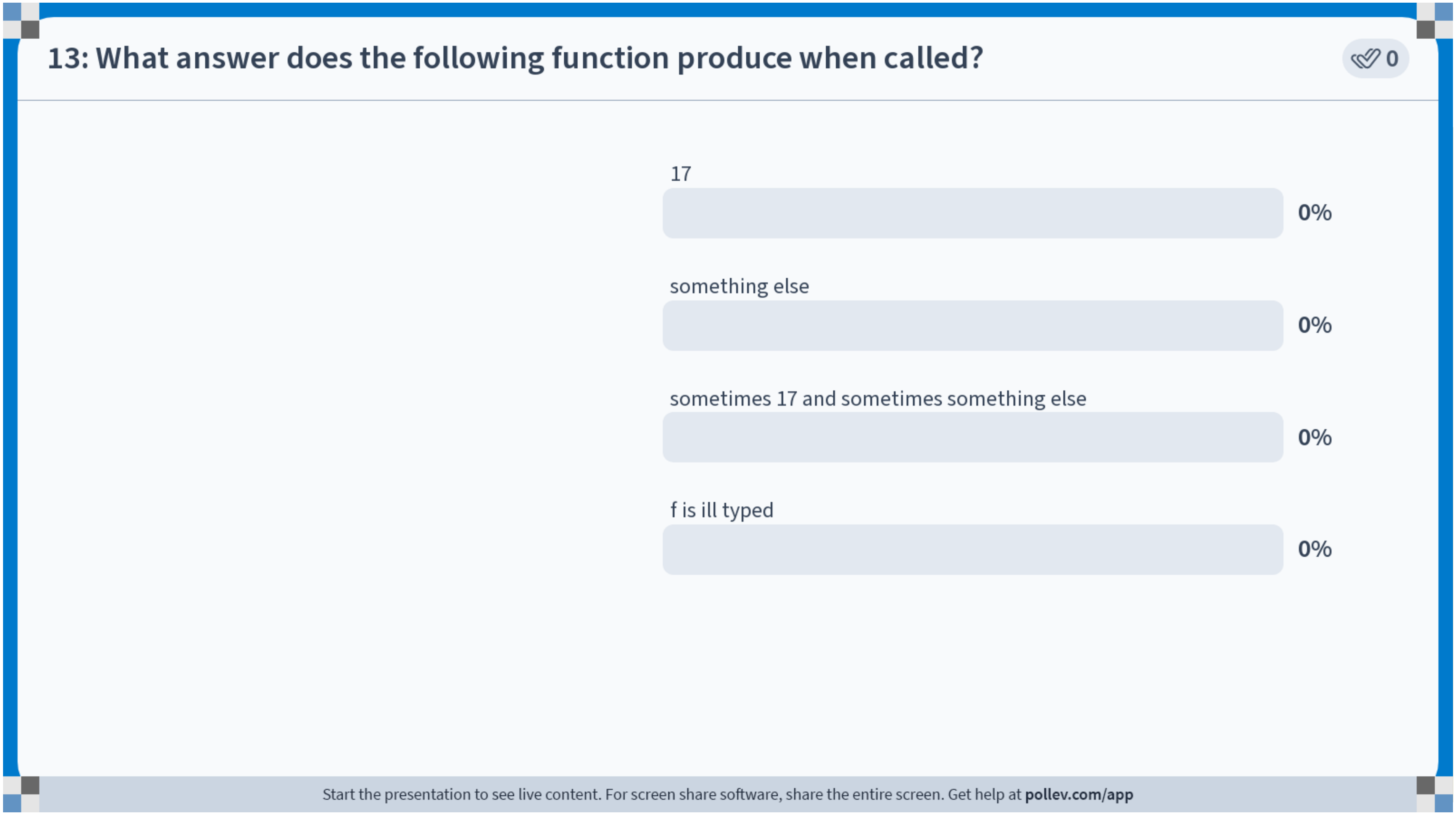
in-place update of p0.x

Record Update

- Functions can assign to mutable record fields
- Note that the return type of ‘<-’ is unit
  - i.e., it is a *command*

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

(* a command to shift a point by dx,dy *)
let shift (p:point) (dx:int) (dy:int) : unit =
  p.x <- p.x + dx;
  p.y <- p.y + dy
```

- the result type of `shift` is also unit
  - i.e., `shift` is a *user-defined* command



13: What answer does the following function produce when called?

17

0%

something else

0%

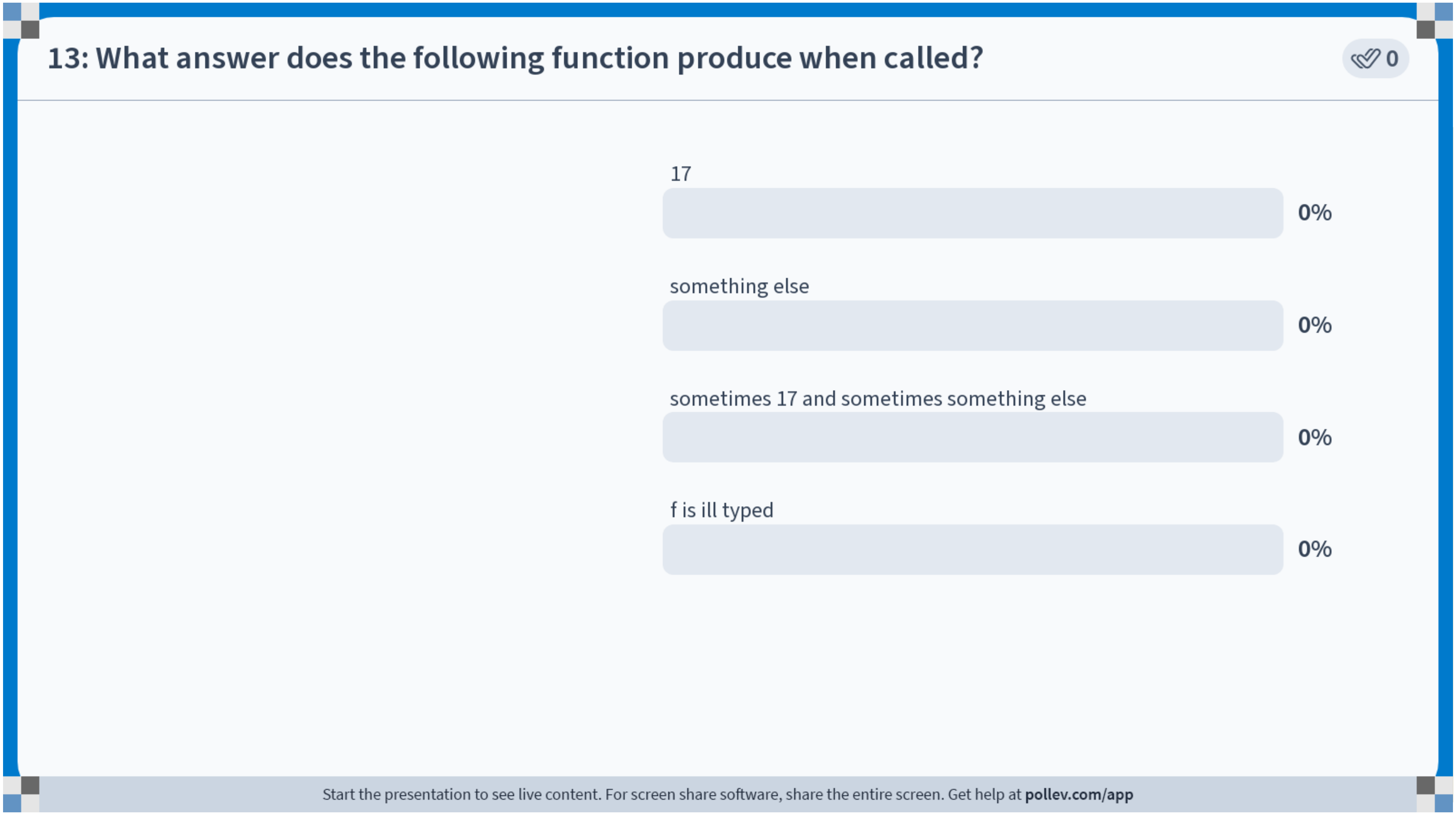
sometimes 17 and sometimes something else

0%

f is ill typed

0%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**



13: What answer does the following function produce when called?

✓ 0

17

0%

something else

0%

sometimes 17 and sometimes something else

0%

f is ill typed

0%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

What answer does the following function produce when called?

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

let f (p1:point) (p2:point) : int =
  p1.x <- 17;
  p2.x <- 42;
  p1.x
```

1. `17`
2. something else
3. sometimes `17` and sometimes something else
4. `f` is ill typed

ANSWER: 3

The Challenge of Mutable State: Aliasing

```
let f (p1:point) (p2:point) : int =
  p1.x <- 17;
  p2.x <- 42;
  p1.x
```

Consider this call to f:

```
let p0 = {x=0; y=0} in
  f p0 p0
```

Two identifiers are said to be *aliases* if they both name the *same* mutable record. Inside f, the identifiers p1 and p2 might or might not be aliased, depending on which arguments are passed in.

SEE THE COURSE NOTES FOR MORE ON THIS EXAMPLE

The Abstract Stack Machine

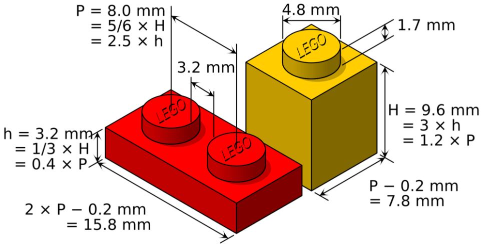
A model of imperative computation

or,

Location, Location, Location!

We Need a New Computation Model

- The simple model of computation we’ve used so far works well for pure *value-oriented* programming
  - "Observable behavior” of a value is completely determined by its structure
  - Two different calls to the same function with the same arguments always yield the same results
  - These properties justify "replace equals by equals" reasoning

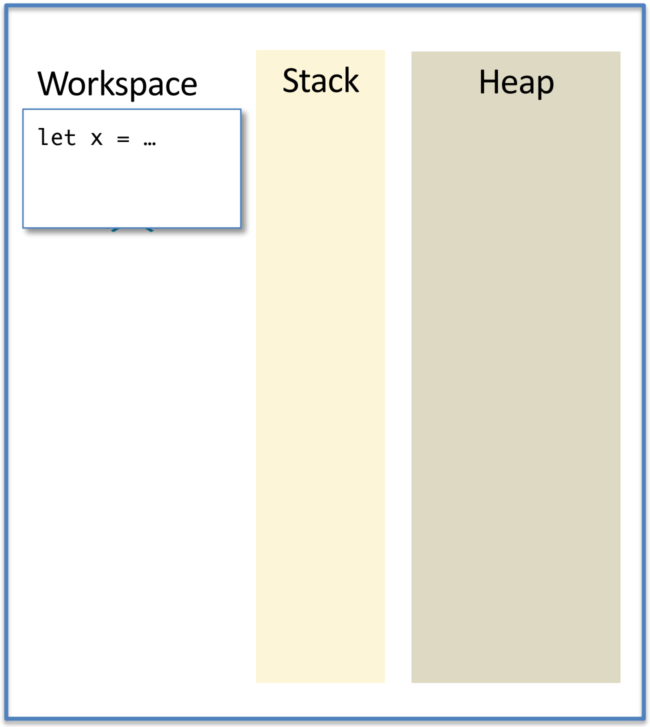


- But with mutable state…
  - The *location* of values matters, not just their structure
  - Results returned by functions are *not* fully determined by their arguments — can also depend on “hidden” mutable state

Abstract Stack Machine

Three "spaces"…

- workspace
  - the expression the computer is currently simplifying
  - abstraction of the CPU
- stack
  - temporary storage for local variables and saved work
  - abstraction of (part of) RAM
- heap
  - storage area for large data structures
  - abstraction of (part of) RAM



```
let x = …
```

Abstract stack machine

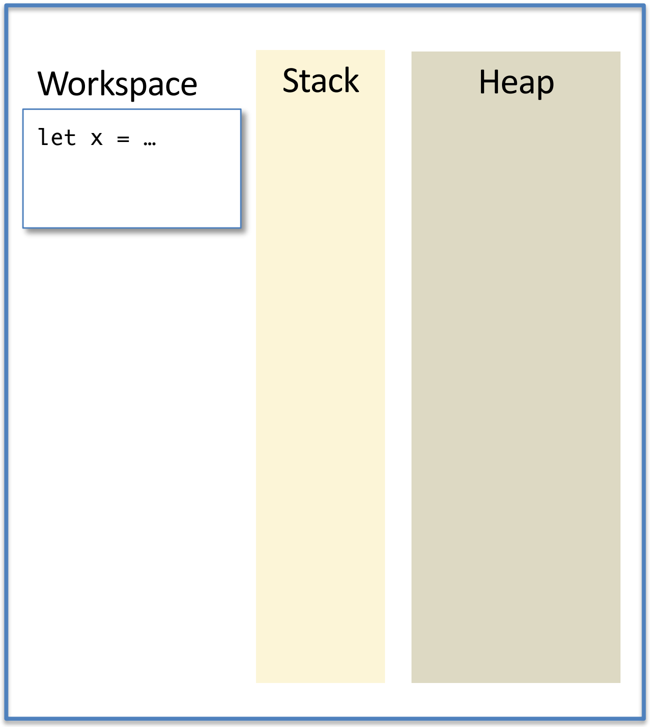
Abstract Stack Machine

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
- (Sometimes this will change the stack and/or heap)
- Stop when there are no more simplifications to be done



Abstract stack machine

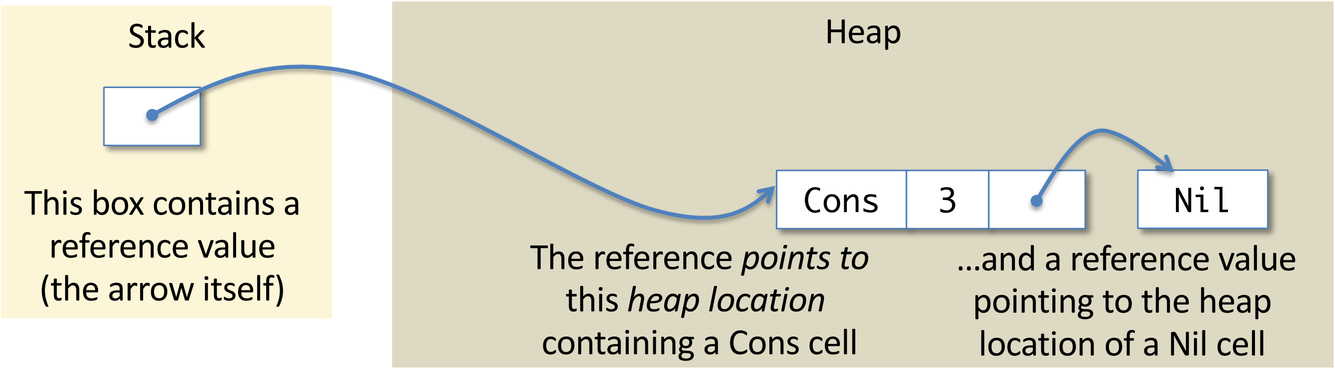
Values and References

A *value* is either:

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

A reference value is the *address (location)* of data in the heap.

We draw a reference value as an arrow pointing to the data "located at" this address



This box contains a reference value (the arrow itself)

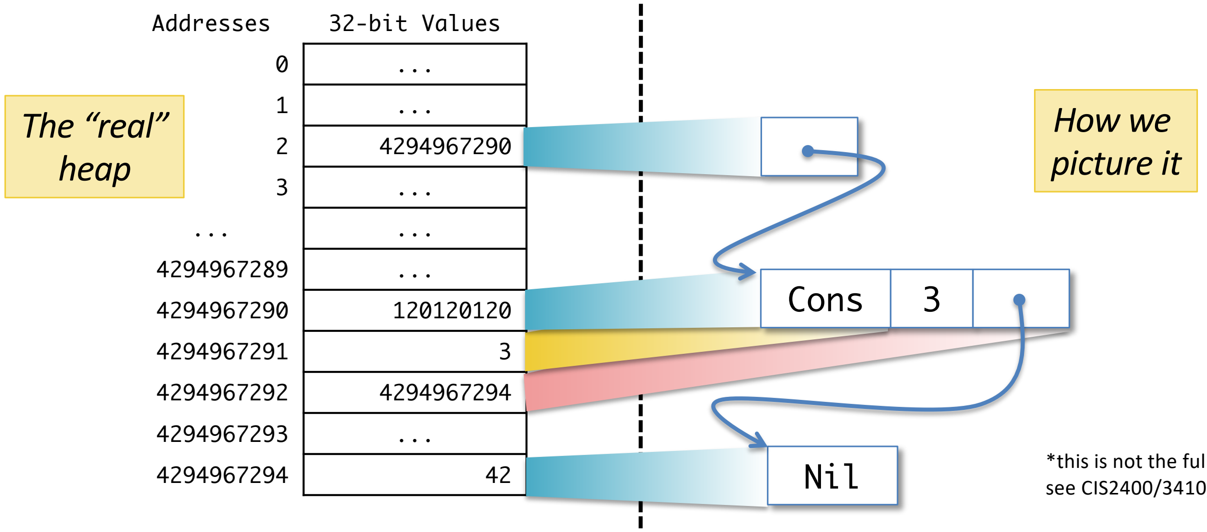
The reference *points to* this *heap location* containing a Cons cell

...and a reference value pointing to the heap location of a Nil cell

References are 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* (*pointer*) is an address indicating *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

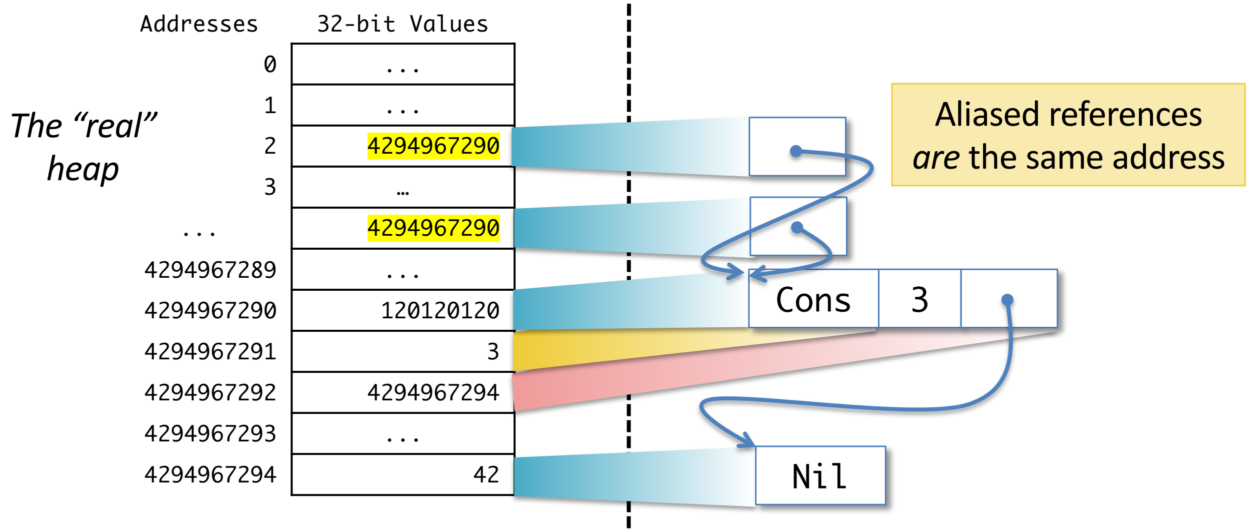


Addresses	32-bit Values
0	...
1	...
2	4294967290
3	...
...	...
4294967289	...
4294967290	120120120
4294967291	3
4294967292	4294967294
4294967293	...
4294967294	42

*this is not the full story;
see CIS2400/3410/3800/4710

References are an Abstraction

- Usually, the specific addresses chosen for where to place data don't matter
  - programmers don't want to think at that level of detail
  - *aliasing* (i.e., sharing the same location) is what matters



The “real” heap

Addresses	32-bit Values
0	...
1	...
2	4294967290
3	…
...	4294967290
4294967289	...
4294967290	120120120
4294967291	3
4294967292	4294967294
4294967293	...
4294967294	42

Aliased references *are* the same address

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

Using the stack instead of substitution

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

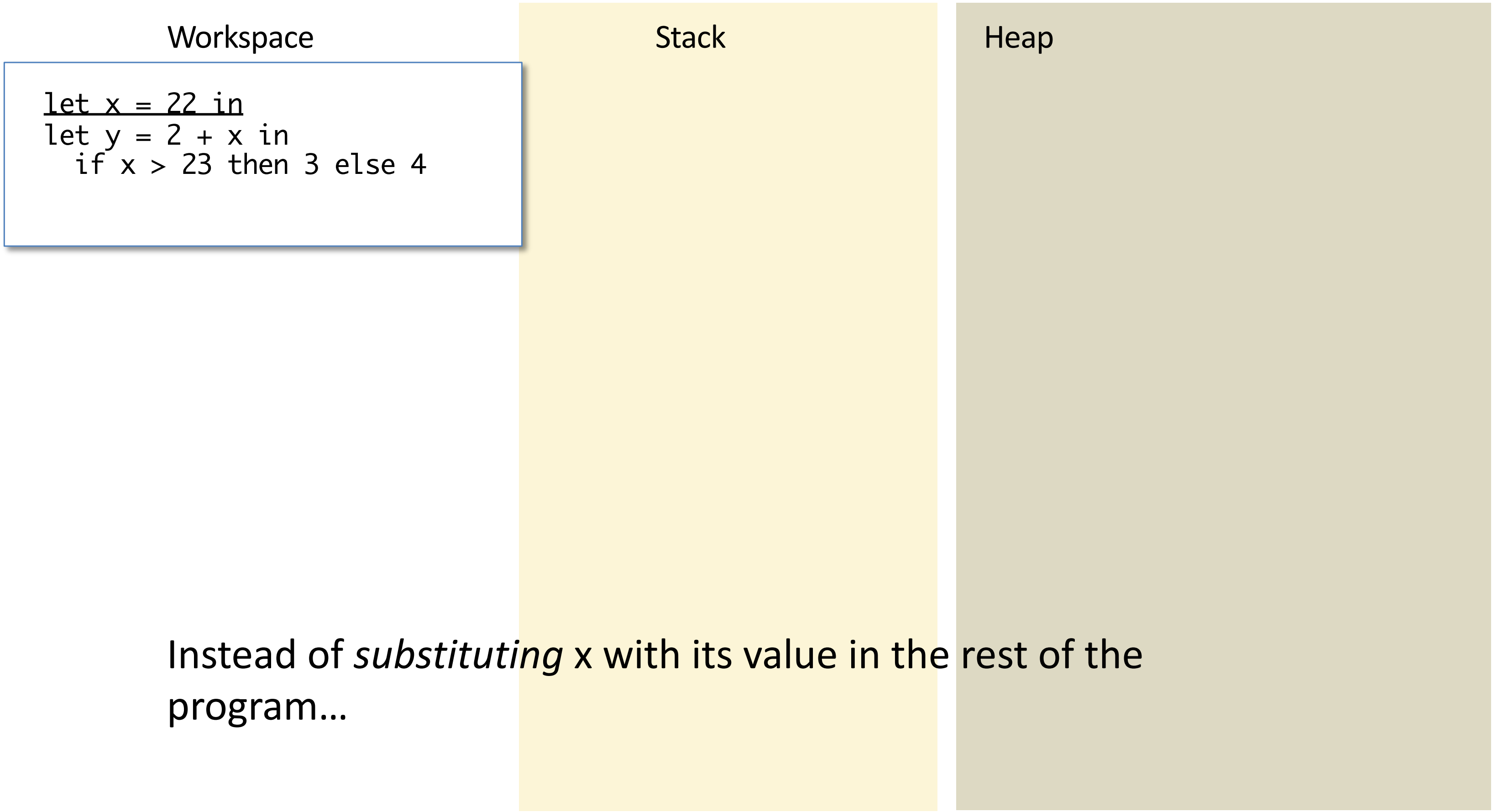
Workspace

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

Stack

Heap

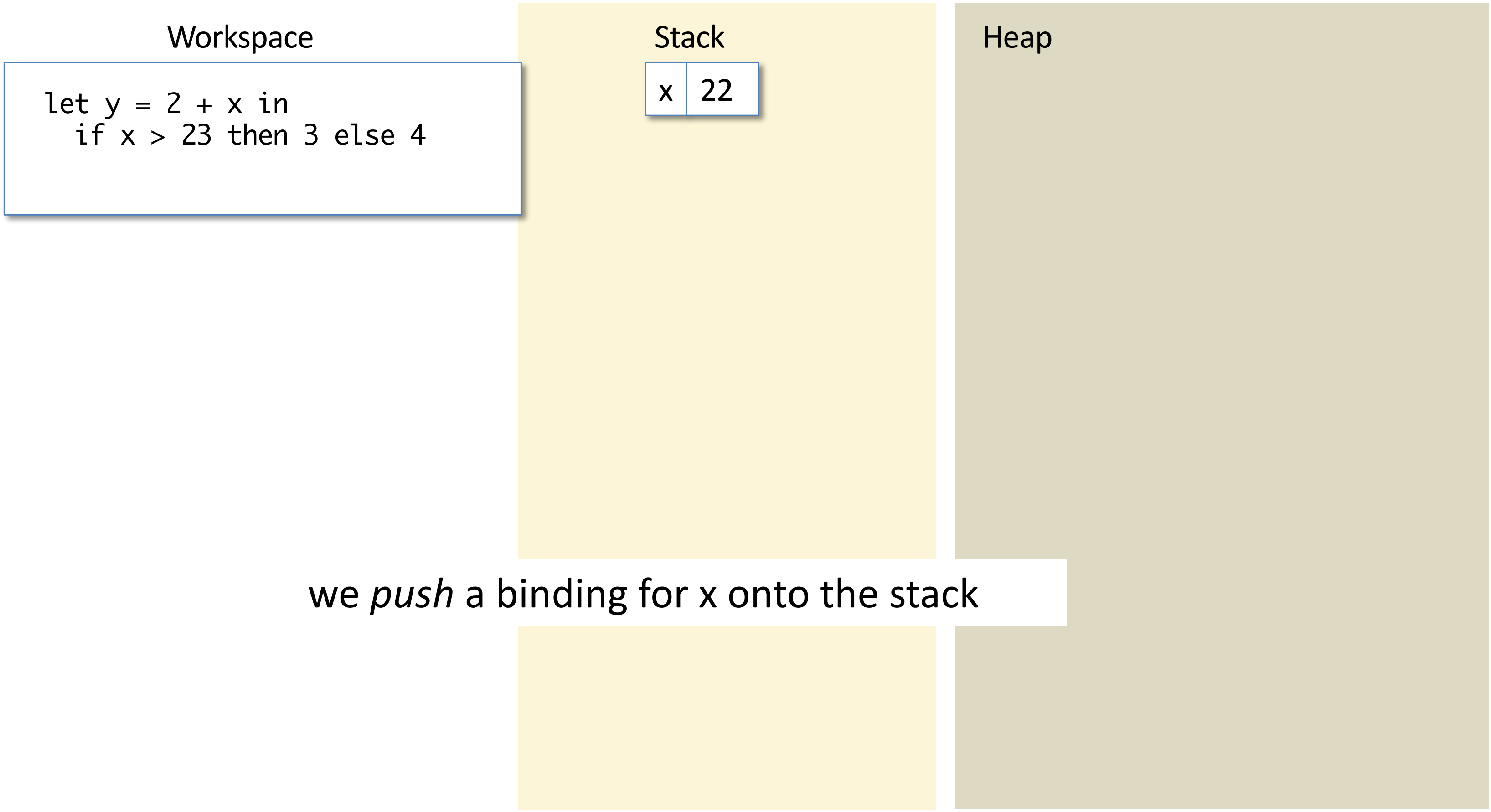
Simplification



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

Instead of *substituting* x with its value in the rest of the program…

Simplification



Workspace

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

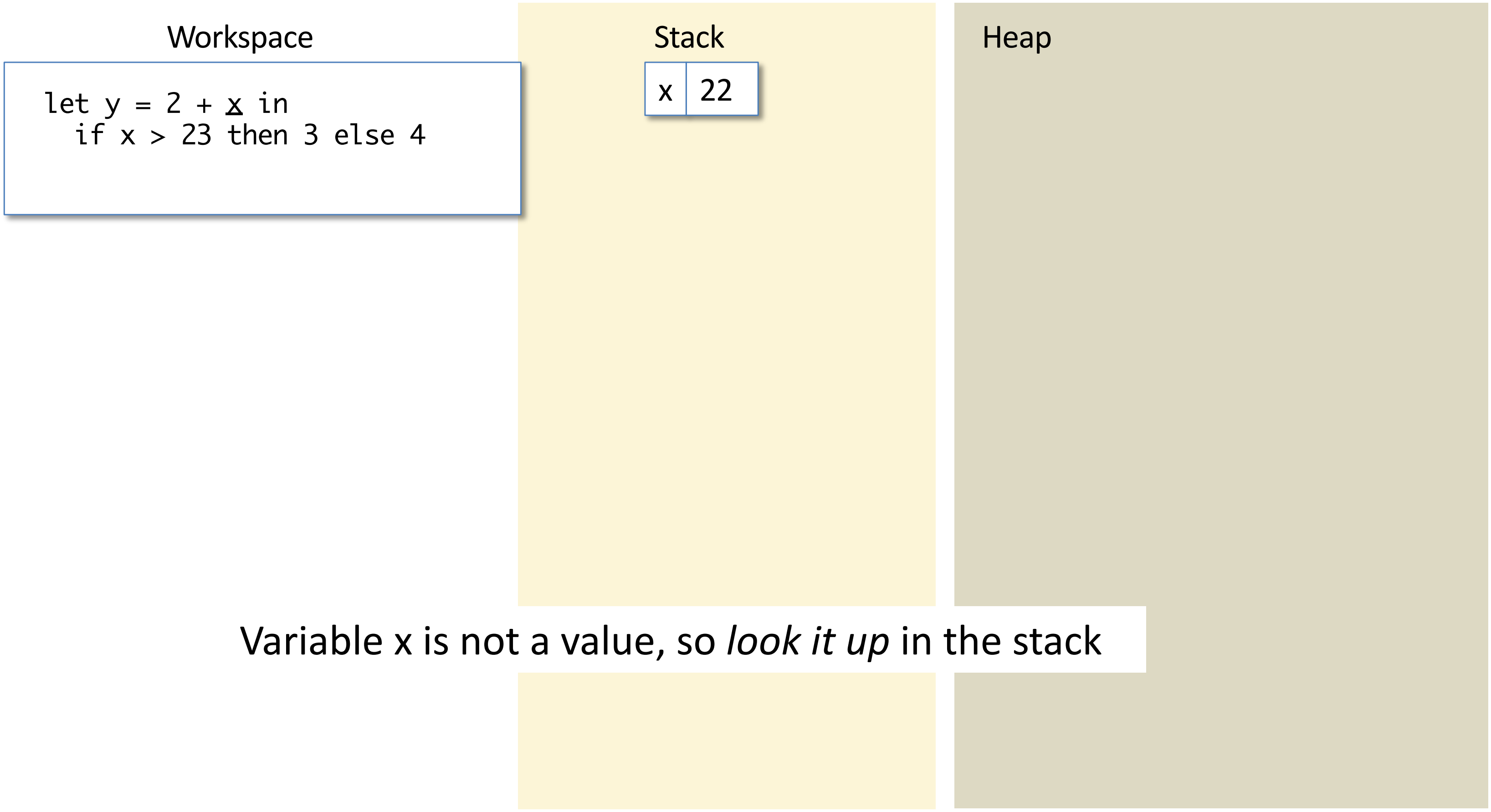
Stack

x	22

Heap

we *push* a binding for x onto the stack

Simplification



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

x	22

Variable 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

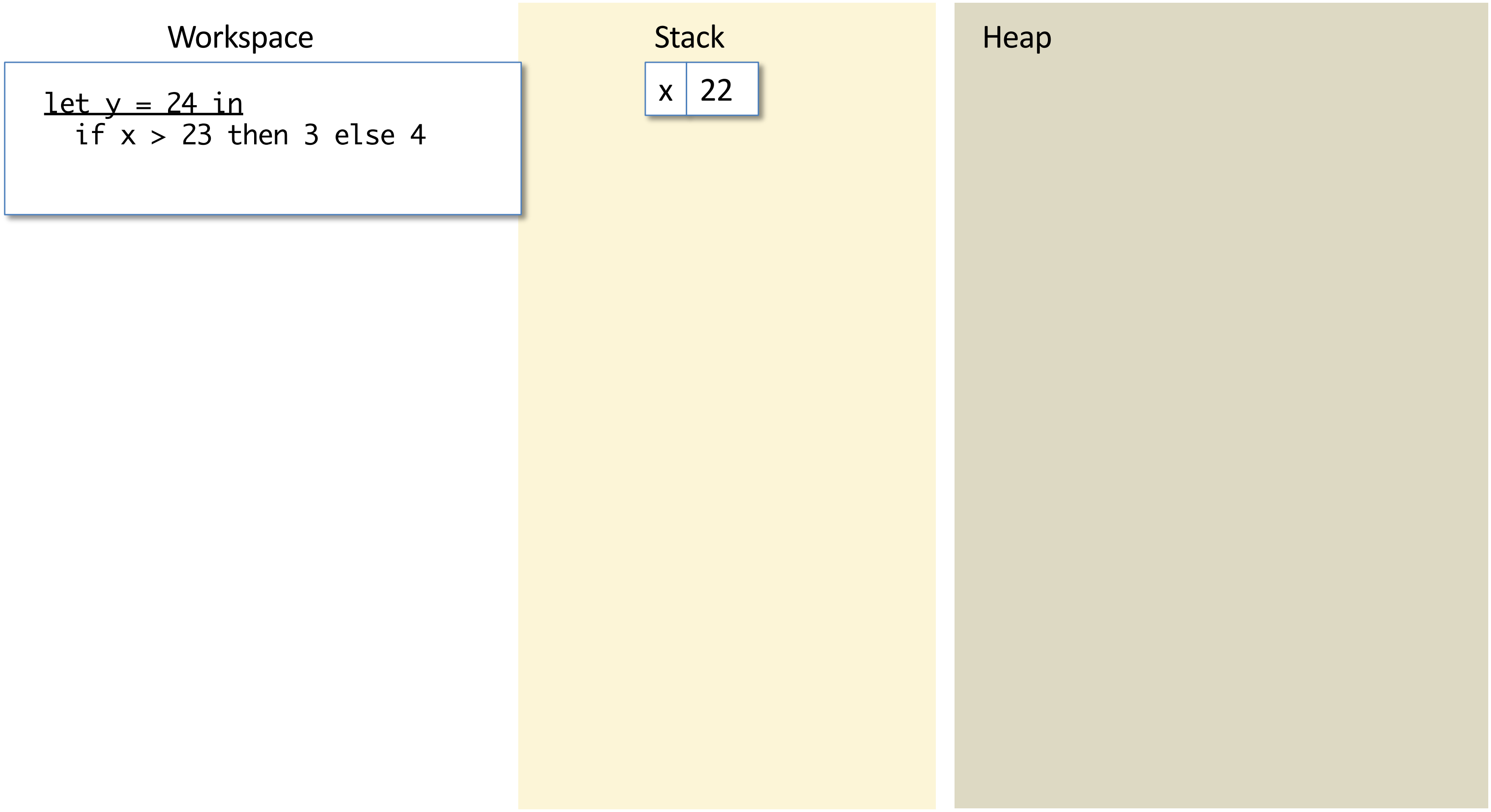
```
let y = 24 in
  if x > 23 then 3 else 4
```

Stack

x	22

Heap

Simplification



Workspace

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

Simplification

Workspace

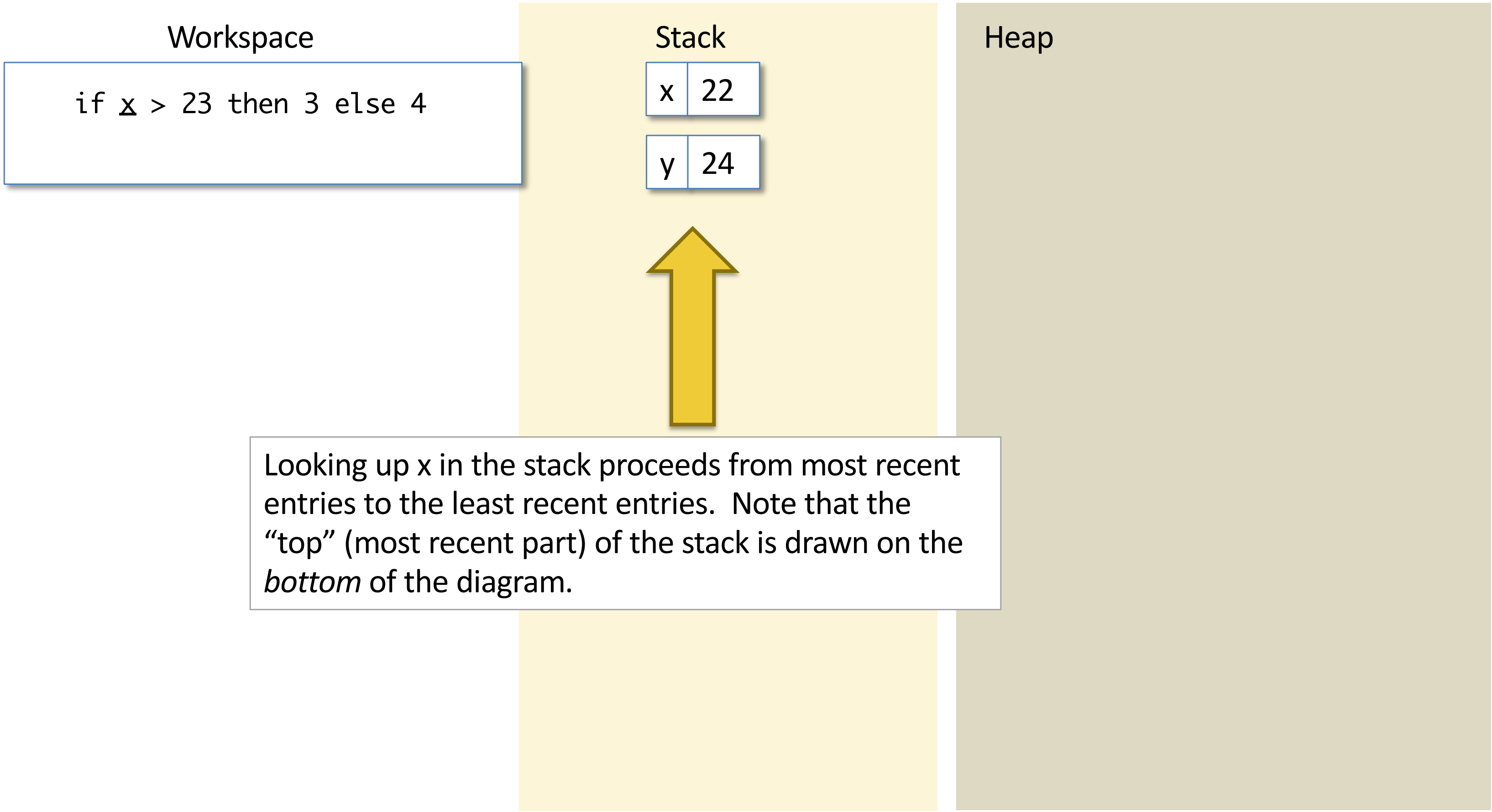
```
if x > 23 then 3 else 4
```

Stack

x	22
y	24

Heap

Simplification



Workspace

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

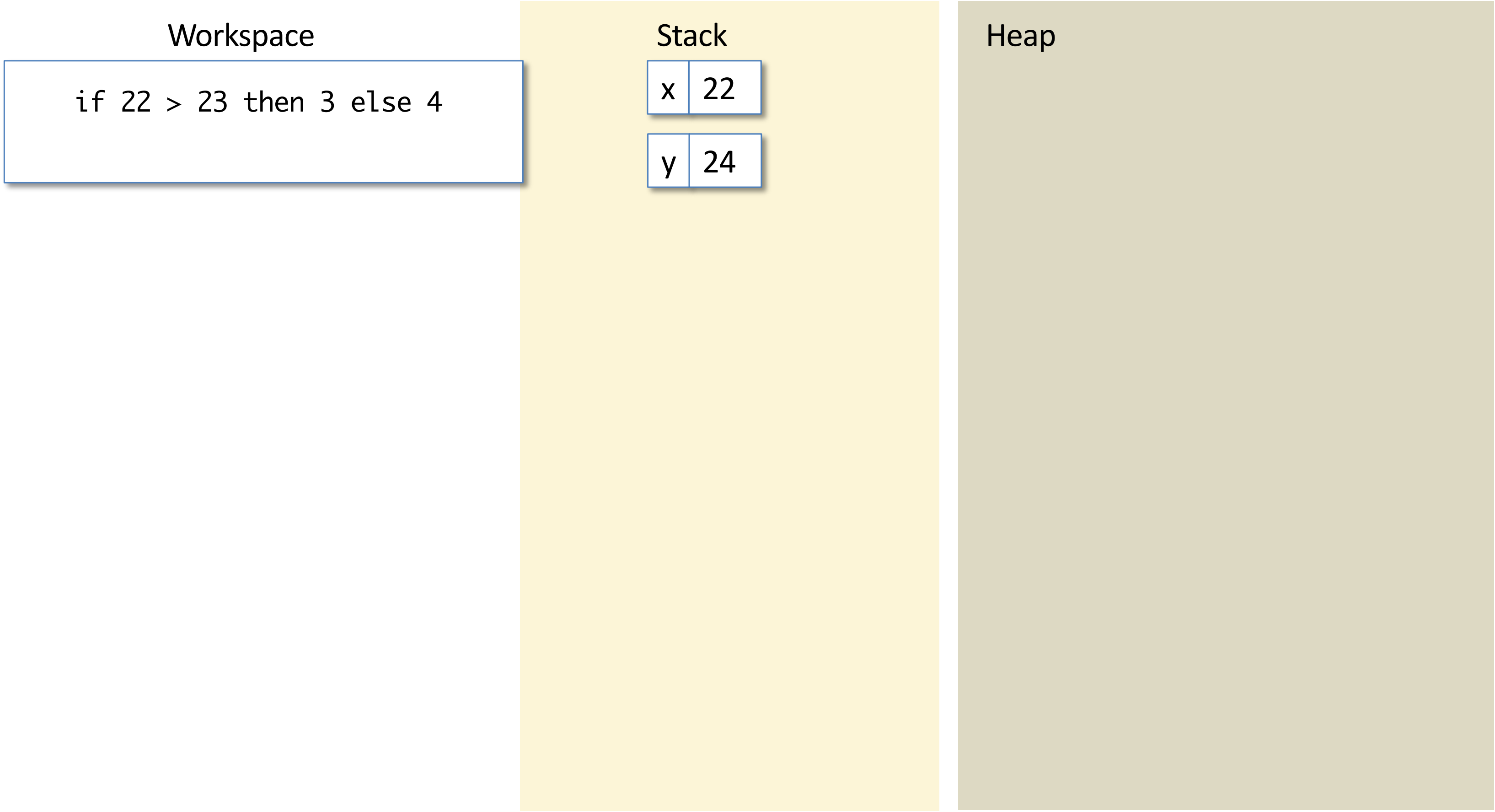
Stack

x	22
y	24

Heap

Looking up x in the stack proceeds from most recent entries to the least recent entries. Note that the “top” (most recent part) of the stack is drawn on the *bottom* of the diagram.

Simplification



Workspace

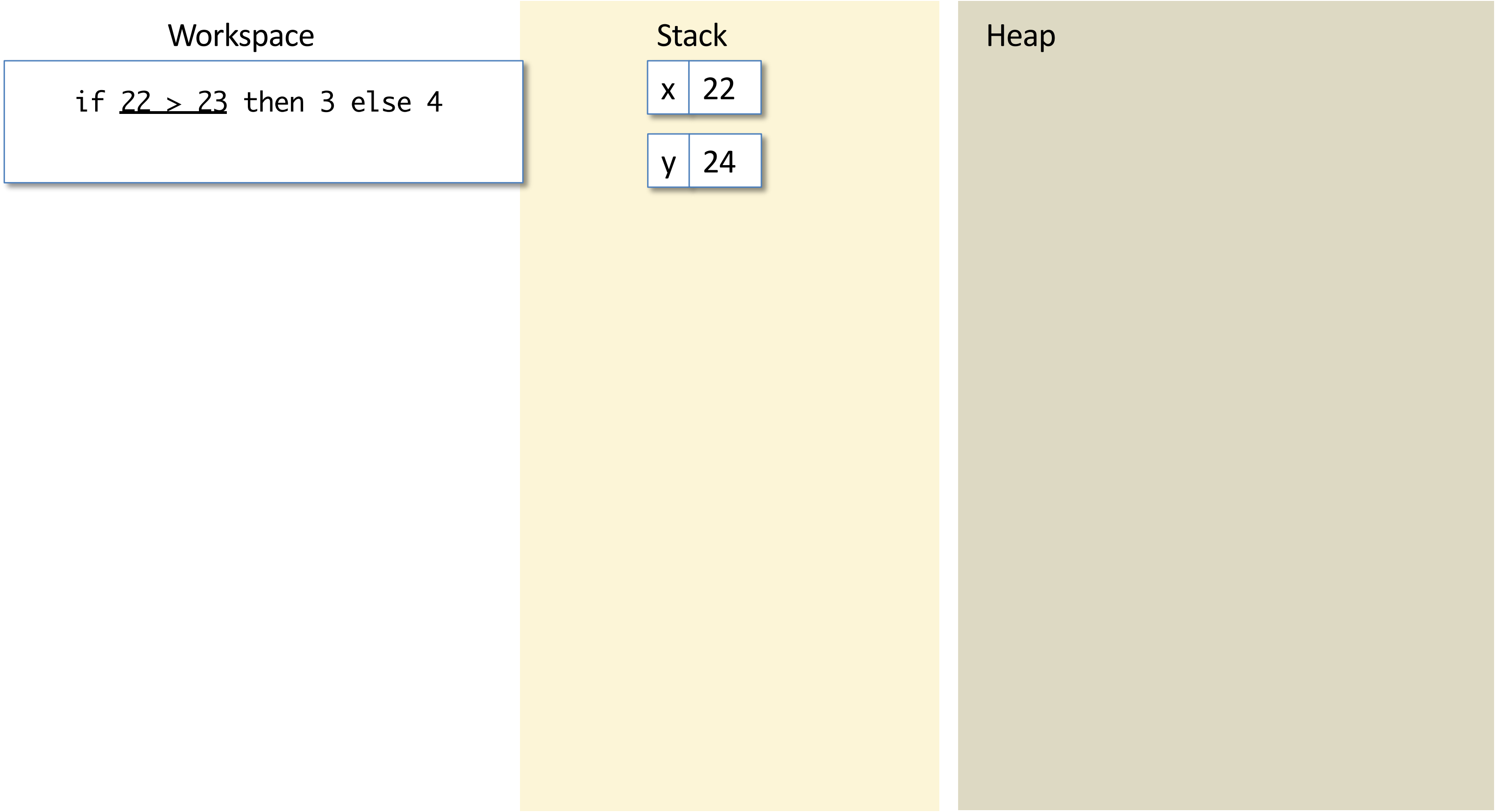
```
if 22 > 23 then 3 else 4
```

Stack

x	22
y	24

Heap

Simplification



Workspace

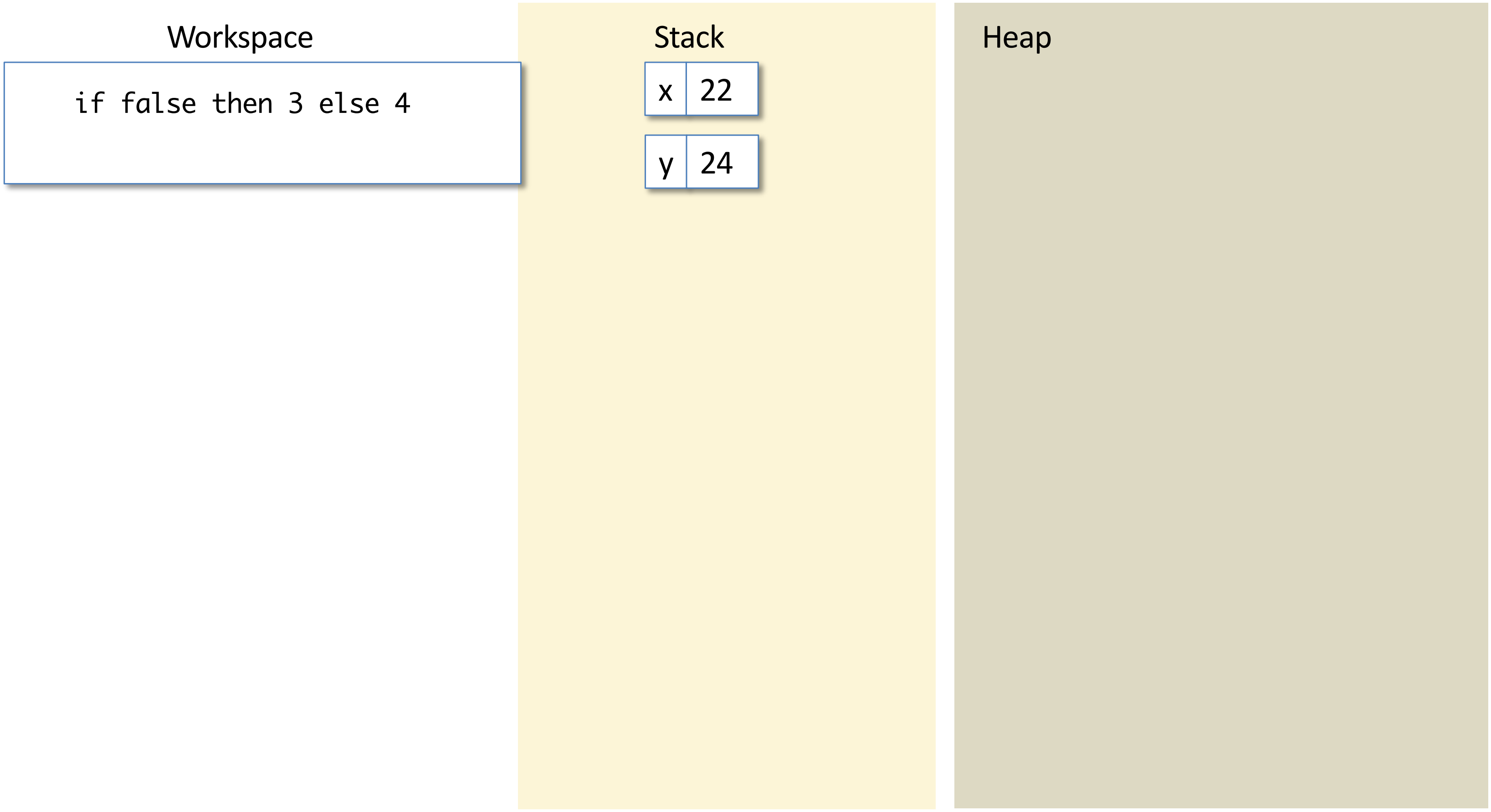
```
if 22 > 23 then 3 else 4
```

Stack

x	22
y	24

Heap

Simplification



Workspace

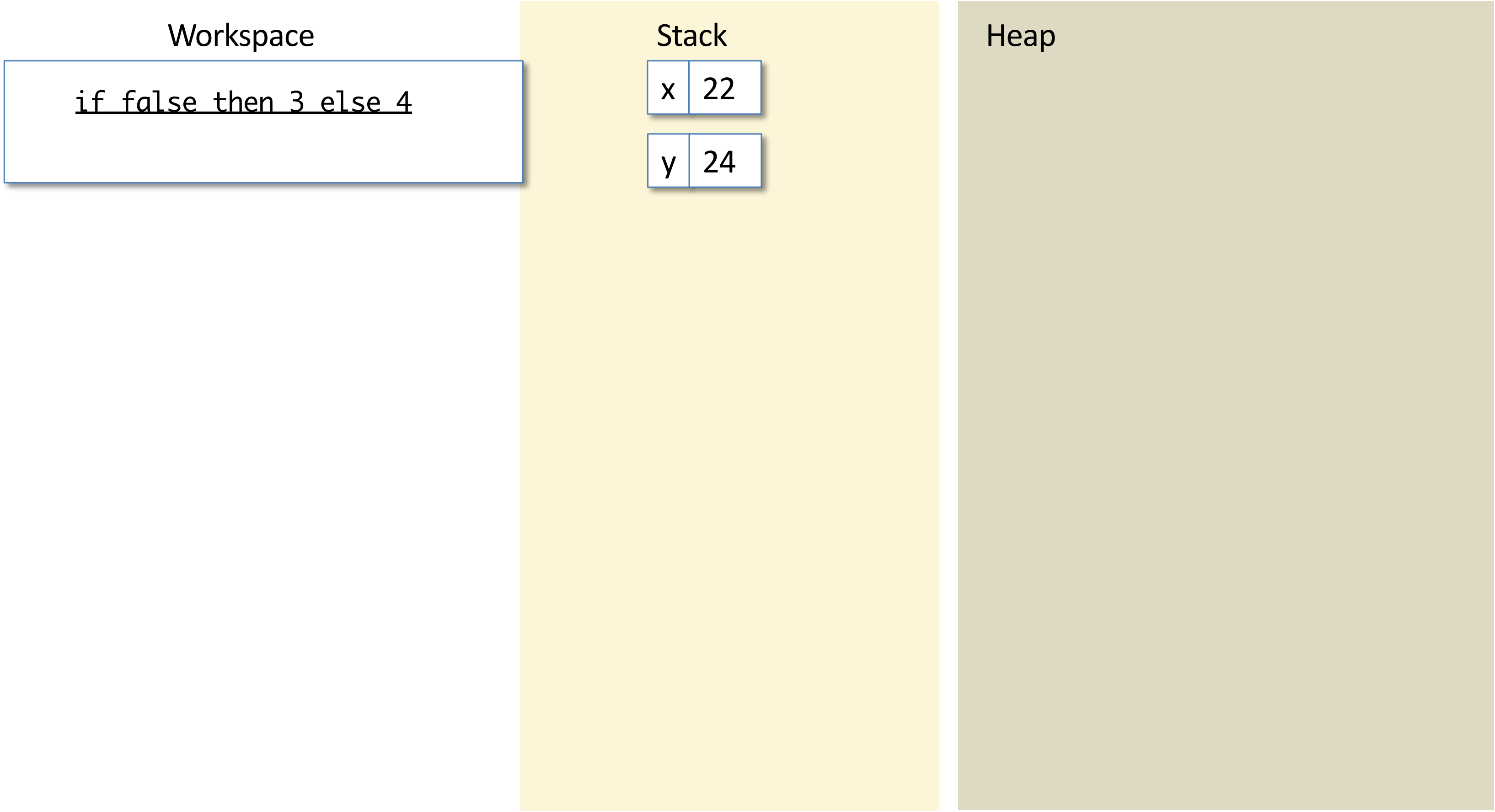
```
if false then 3 else 4
```

Stack

x	22
y	24

Heap

Simplification



Workspace

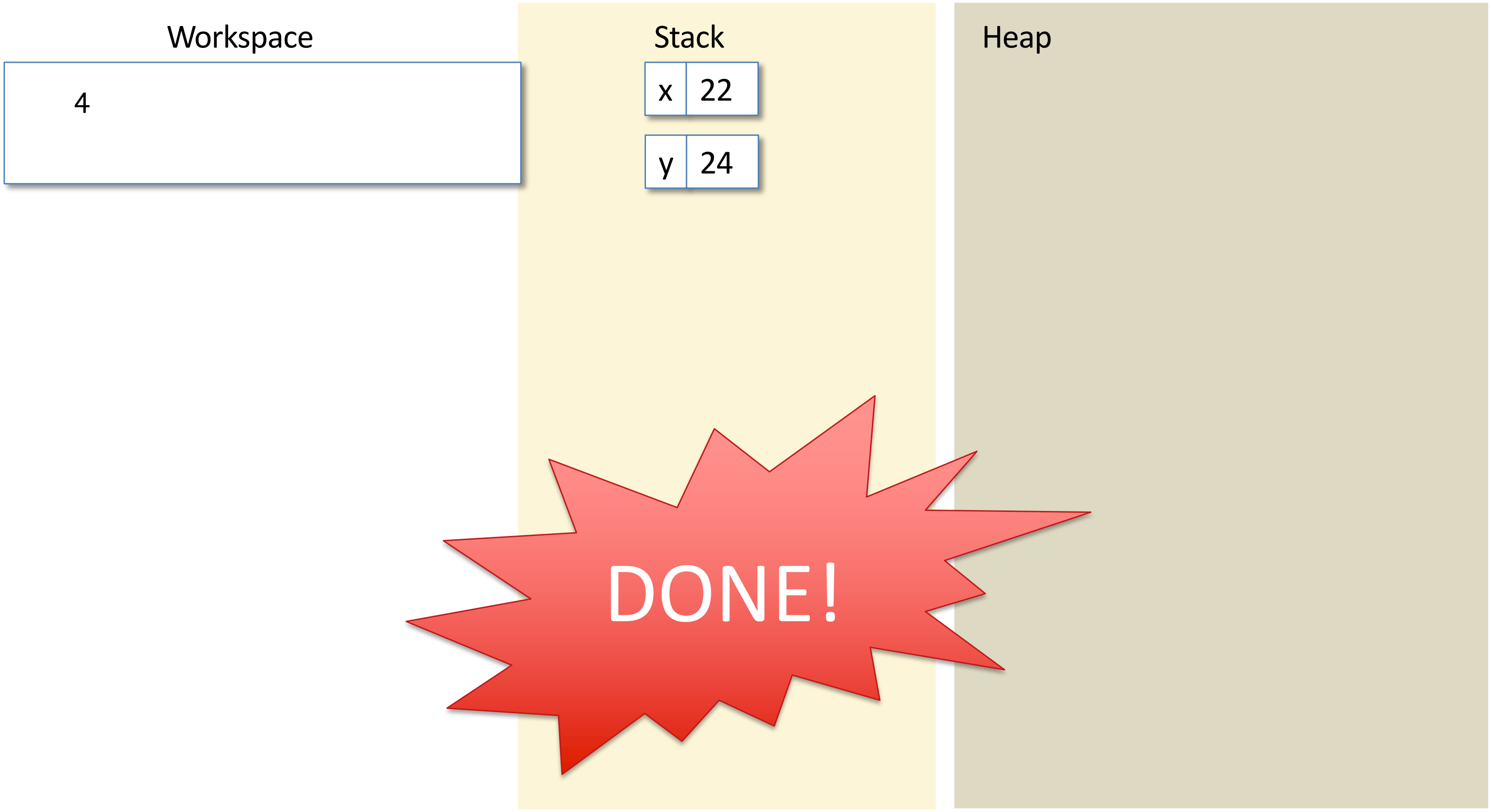
```
if false then 3 else 4
```

Stack

x	22
y	24

Heap

Simplification



Workspace

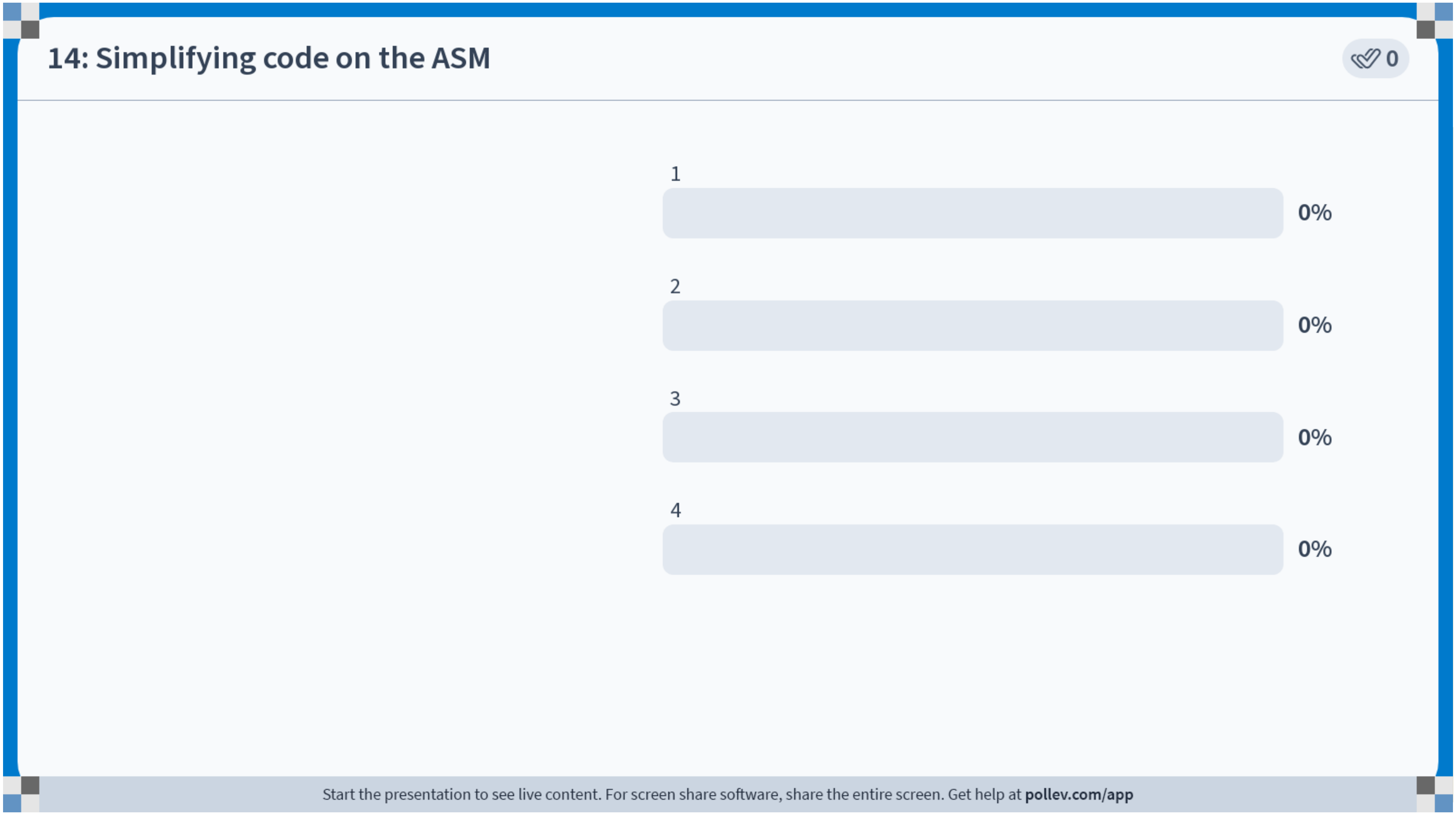
4

Stack

x	22
y	24

Heap

DONE!



14: Simplifying code on the ASM

1	0%
2	0%
3	0%
4	0%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

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

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

Stack

z	22
w	2 + z

1.

Stack

z	20
w	22

2.

Stack

w	22

3.

Stack

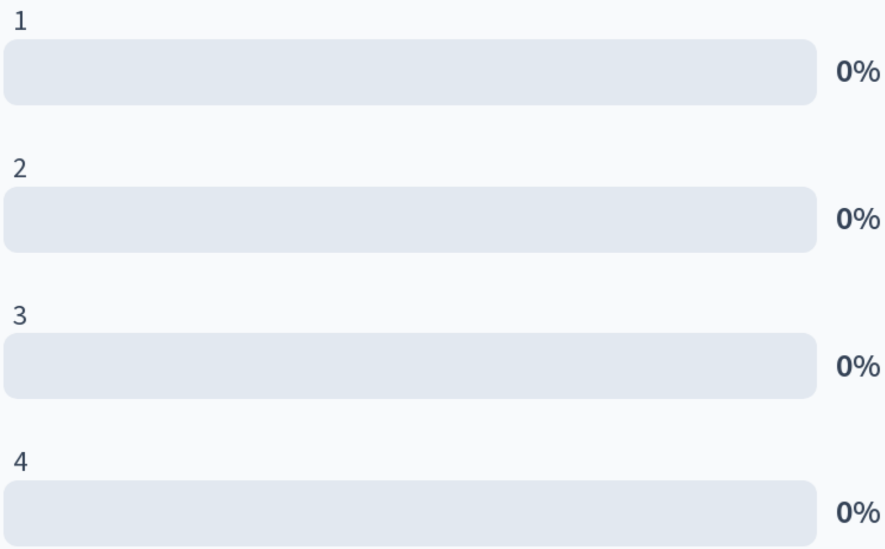
w	22
z	20

4.

ANSWER: 2

14: Simplifying code on the ASM

✓ 0



Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

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

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

Stack

z	22
z	20

1.

Stack

z	20
z	22

2.

Stack

z	22

3.

Stack

z	22
z	22

4.

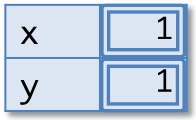
ANSWER: 2

Mutable Records

- The reason for introducing the ASM model is to make 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**



A point record in the heap.

Allocate a Record

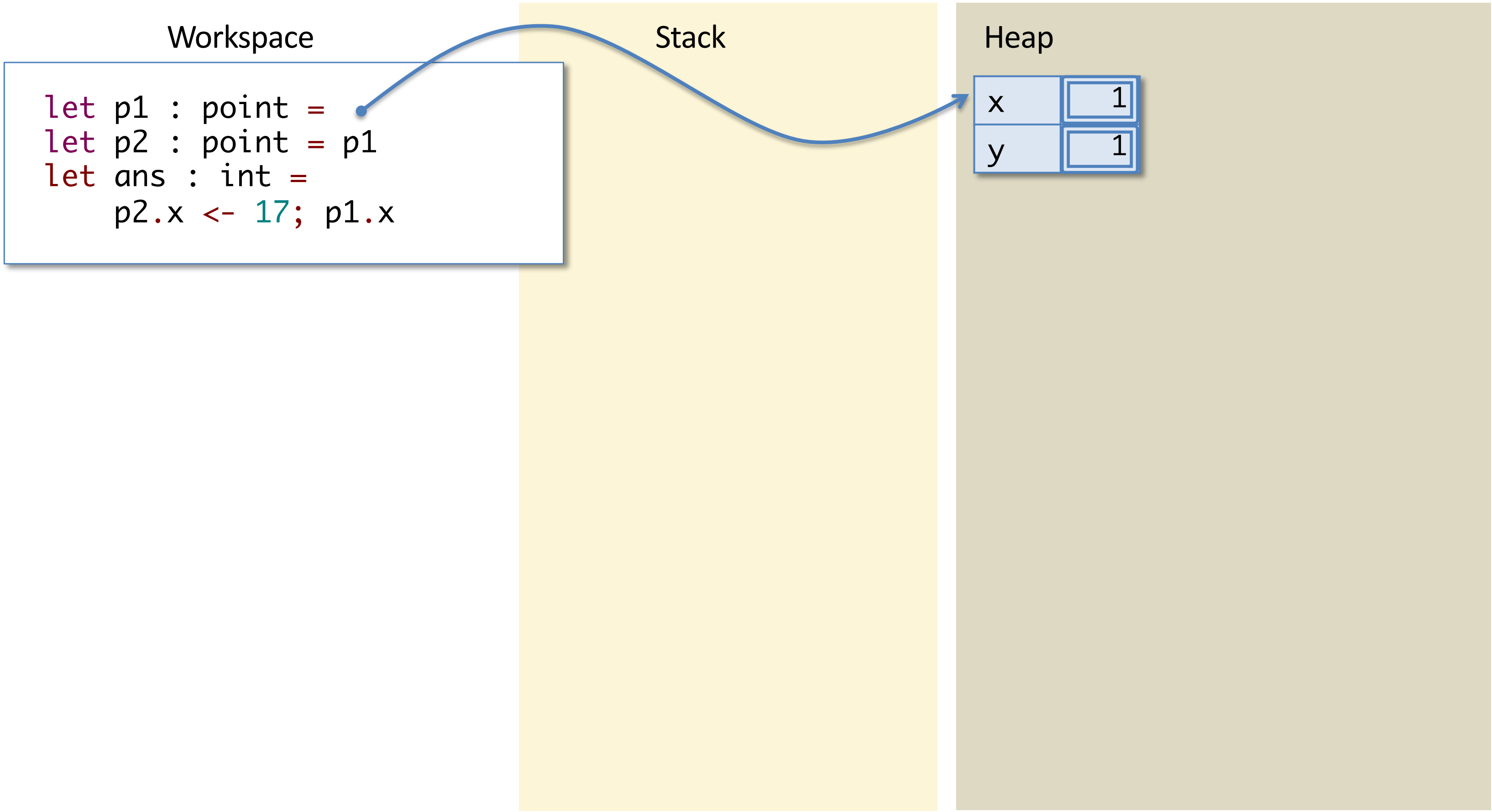
Workspace

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

Stack

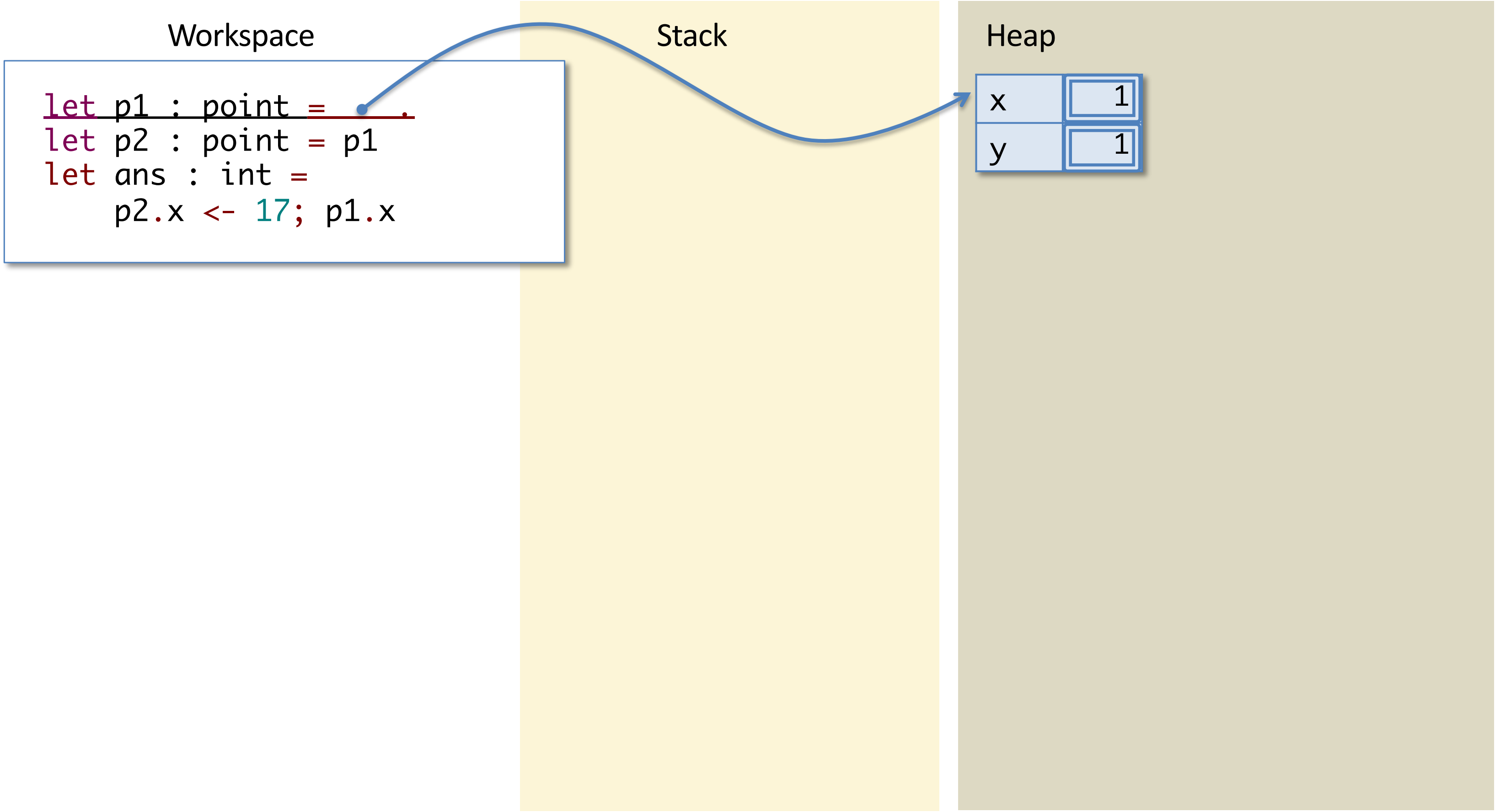
Heap

Allocate a Record



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

Let Expression



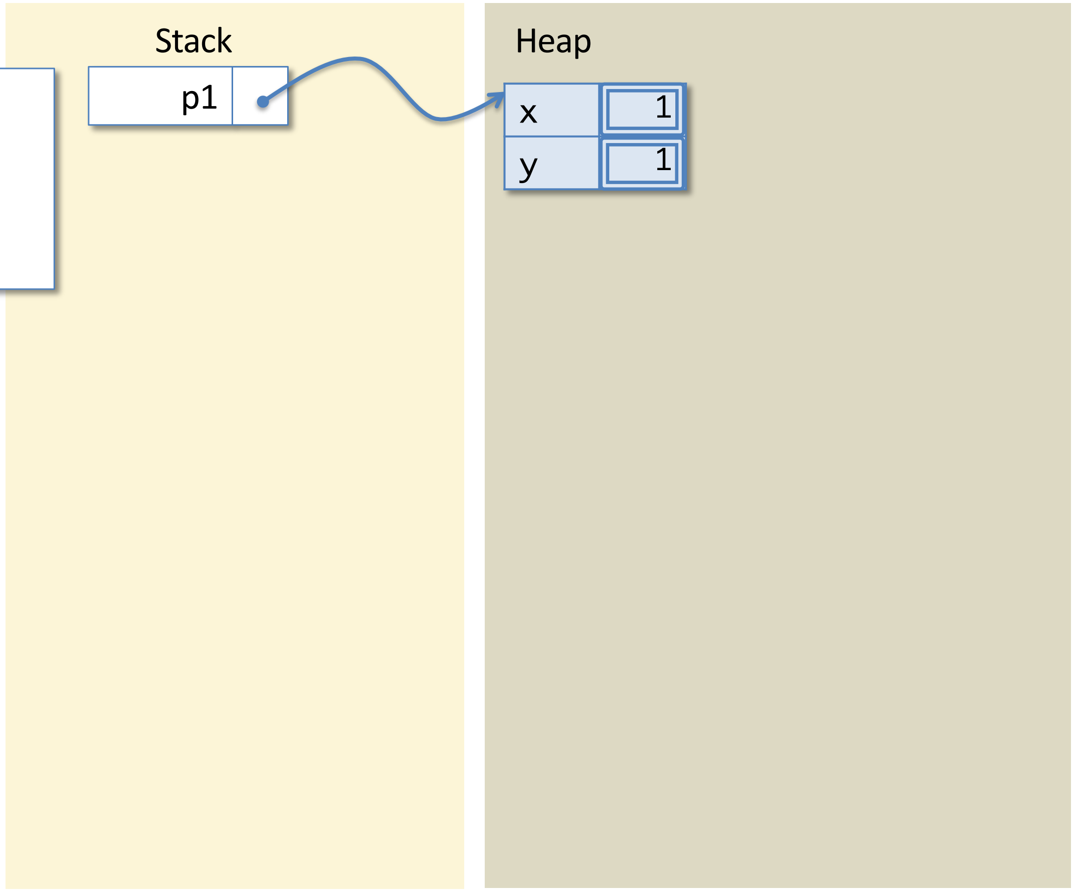
Workspace

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

Push p1

Workspace

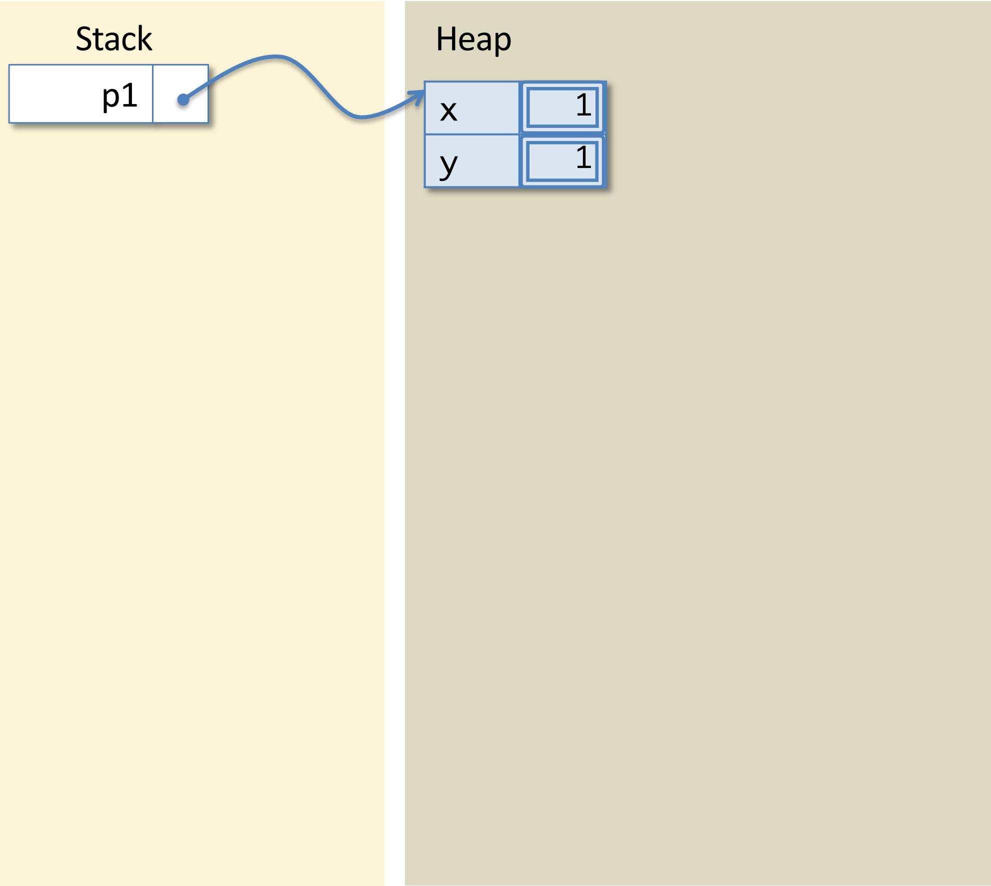
```
let p2 : point = p1
let ans : int =
    p2.x <- 17; p1.x
```



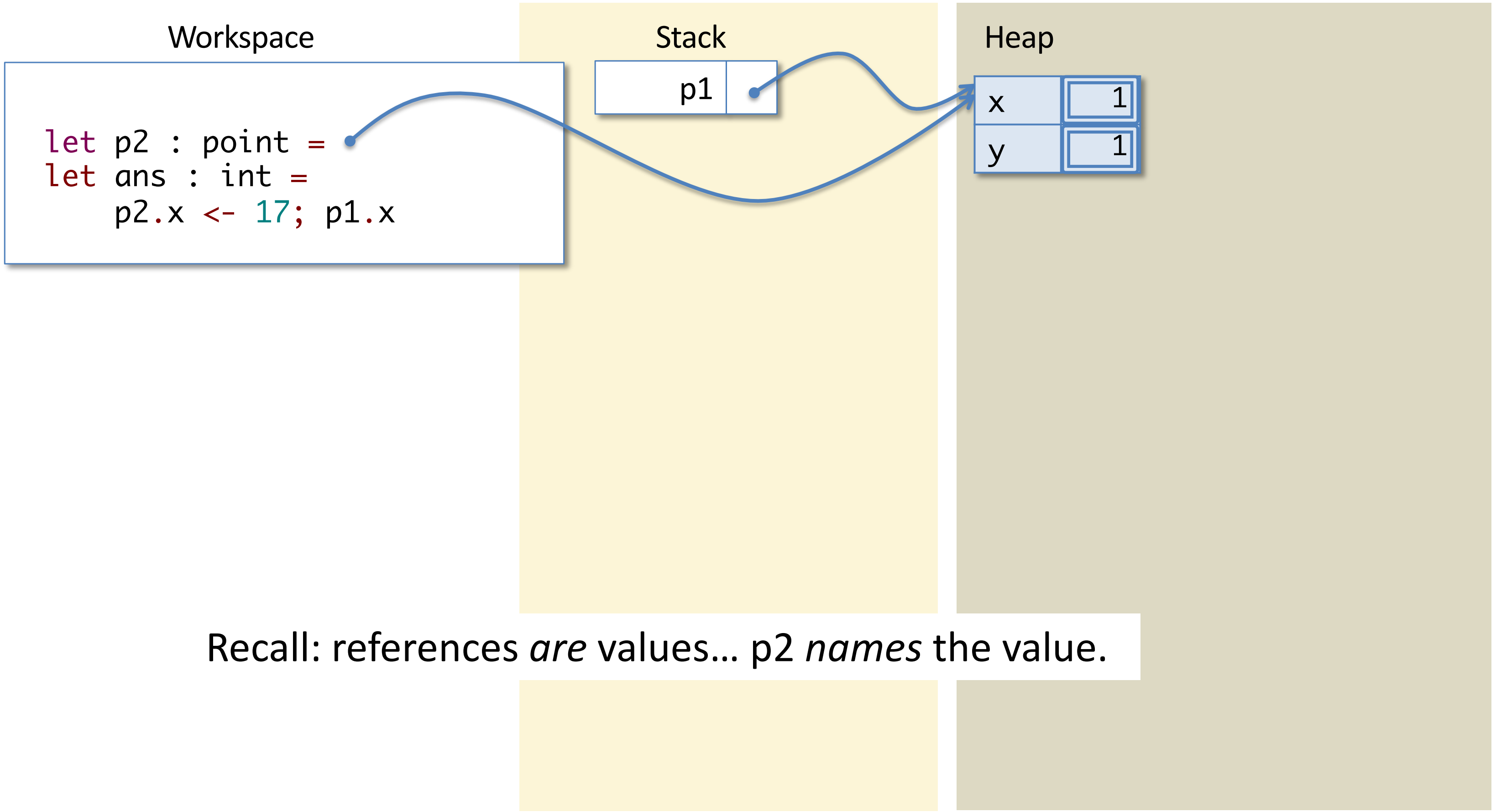
Look Up ‘p1’

Workspace

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



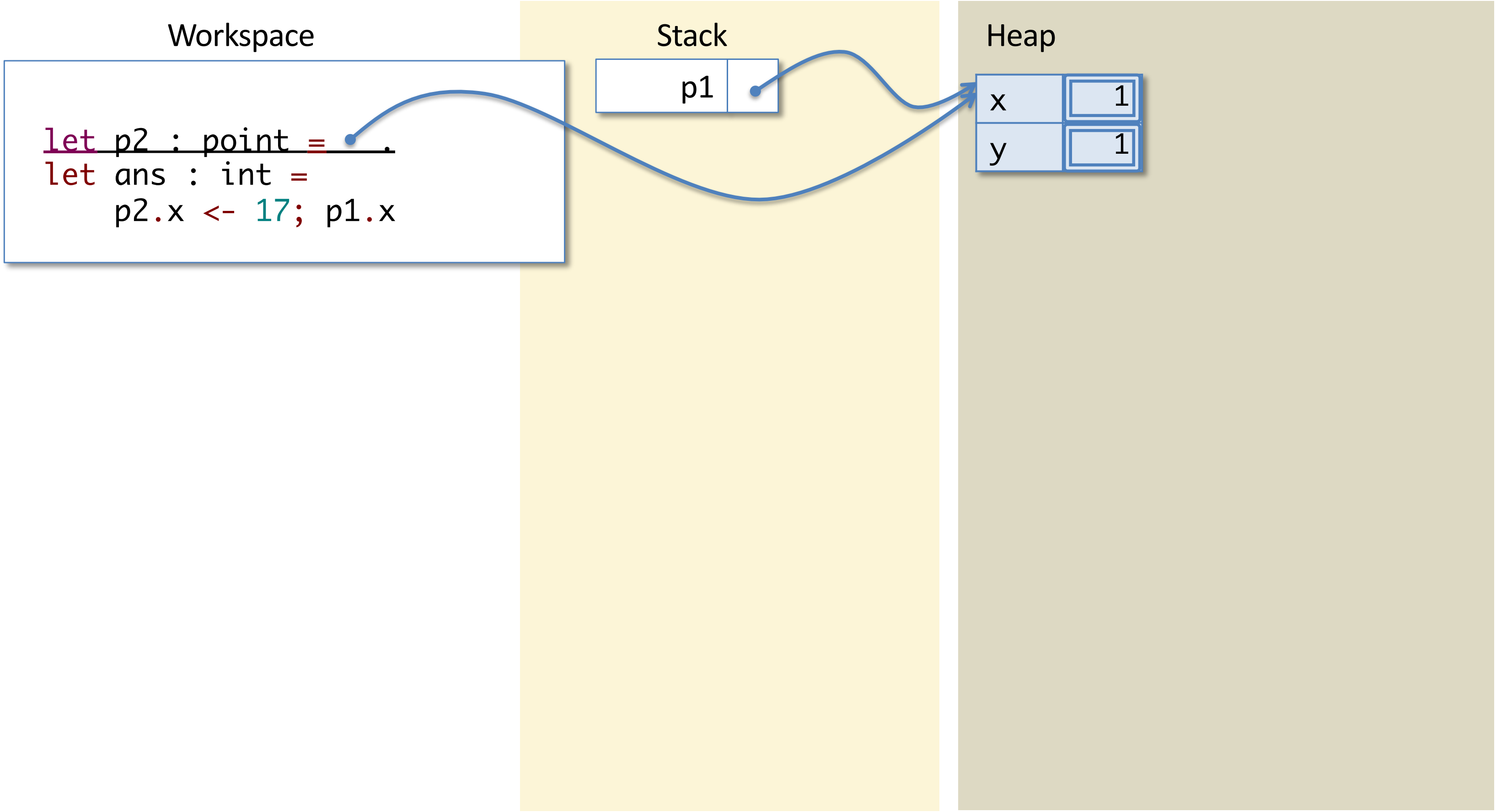
Look Up ‘p1’



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

Recall: references *are* values… p2 *names* the value.

Let Expression

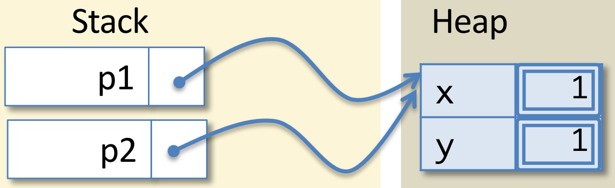


```
let p2 : point =   .
let ans : int =
    p2.x <- 17; p1.x
```

Push p2

Workspace

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

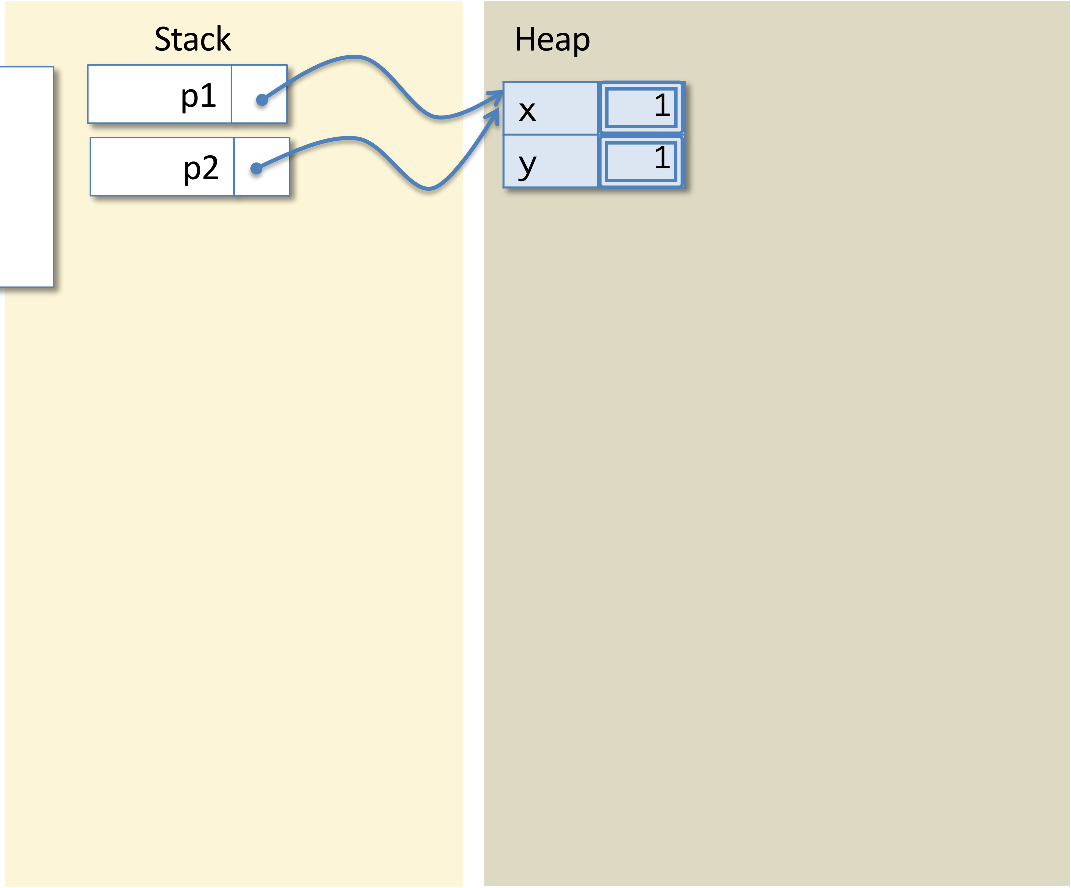


Now p1 and p2 are references to the *same* heap record.
They are *aliases* – two different names for the *same location*.

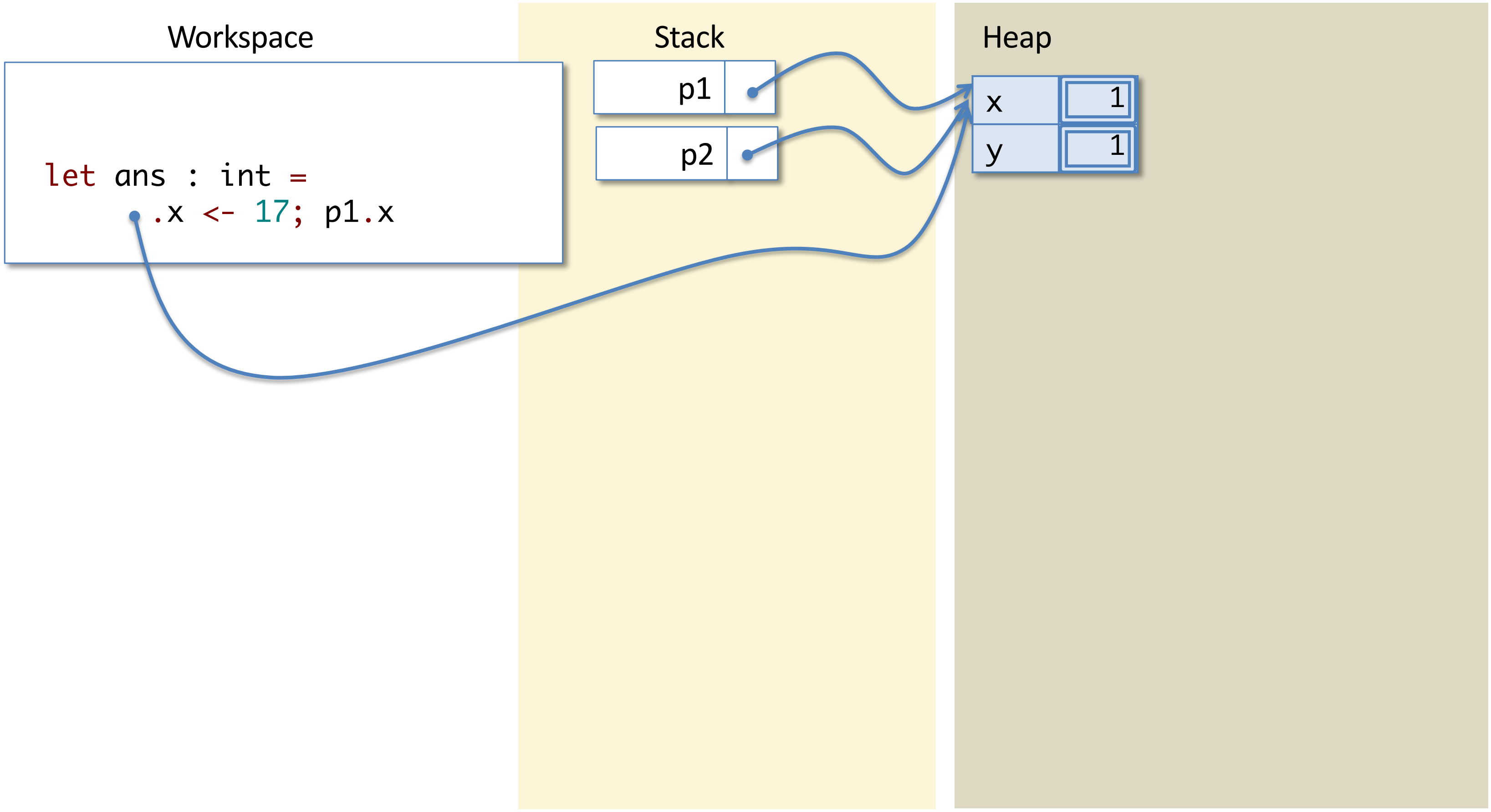
Look Up ‘p2’

Workspace

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



Look Up ‘p2’



Workspace

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

Stack

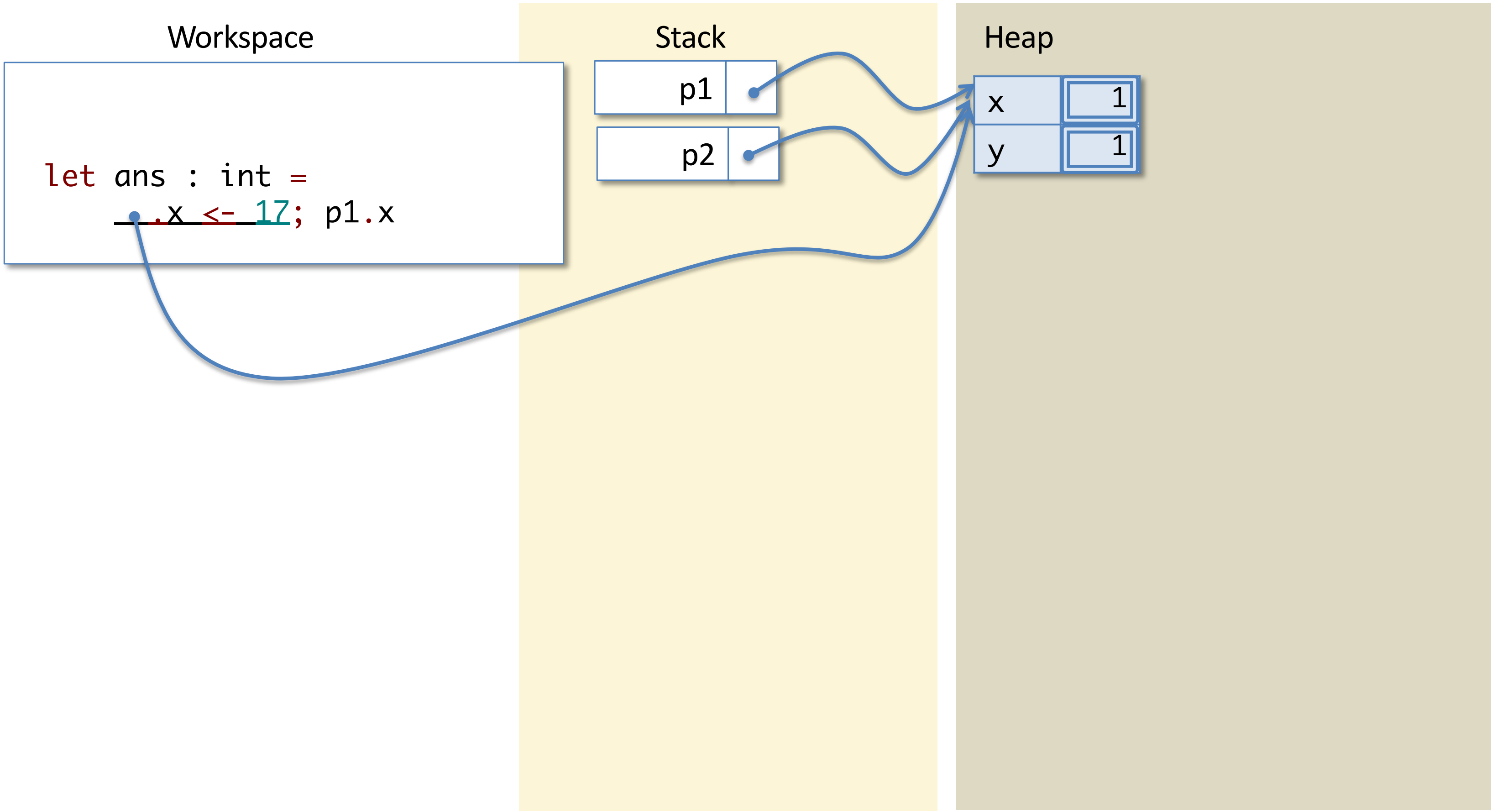
p1

p2

Heap

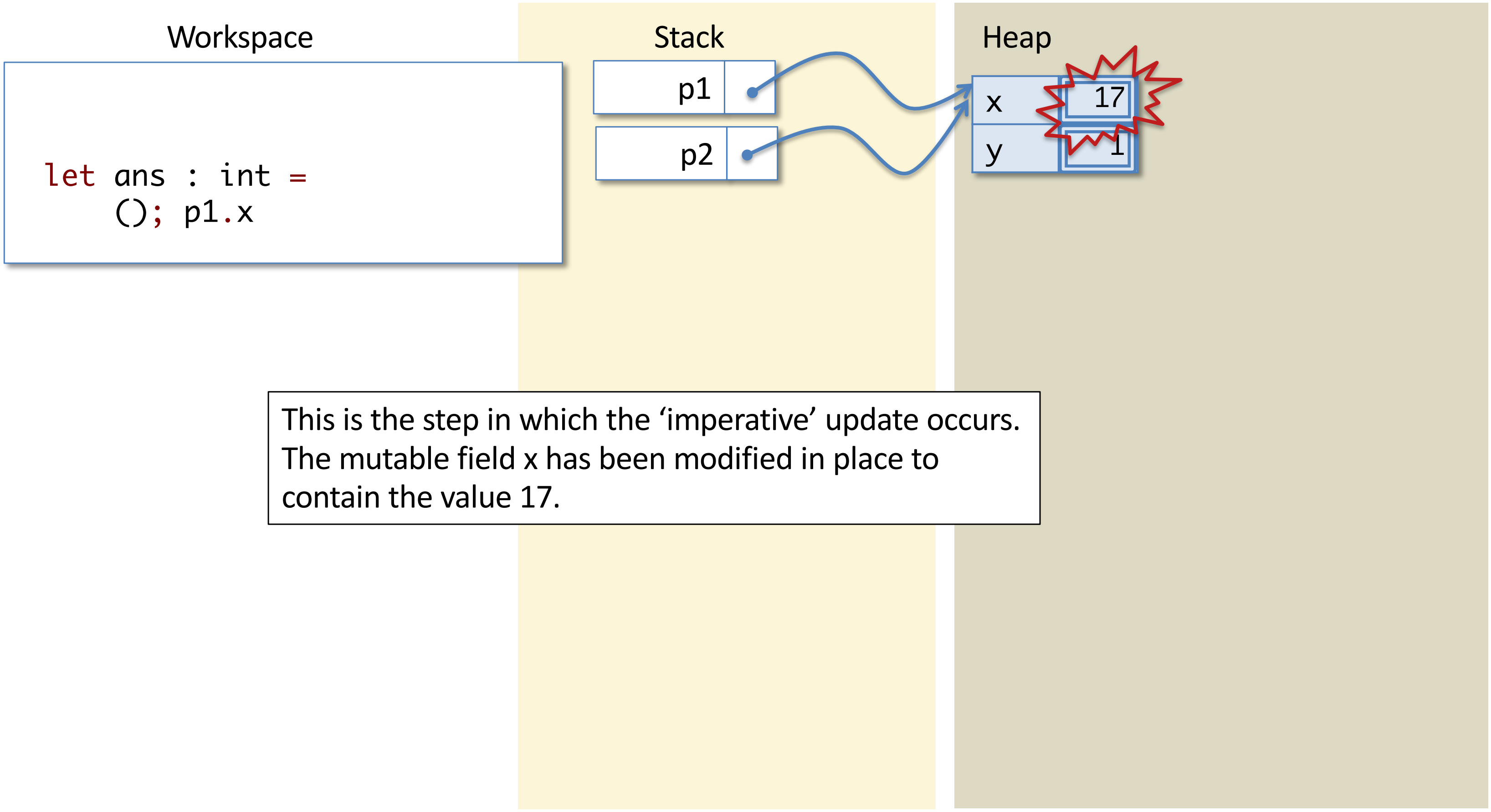
x	1
y	1

Assign to x field



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

Assign to x field



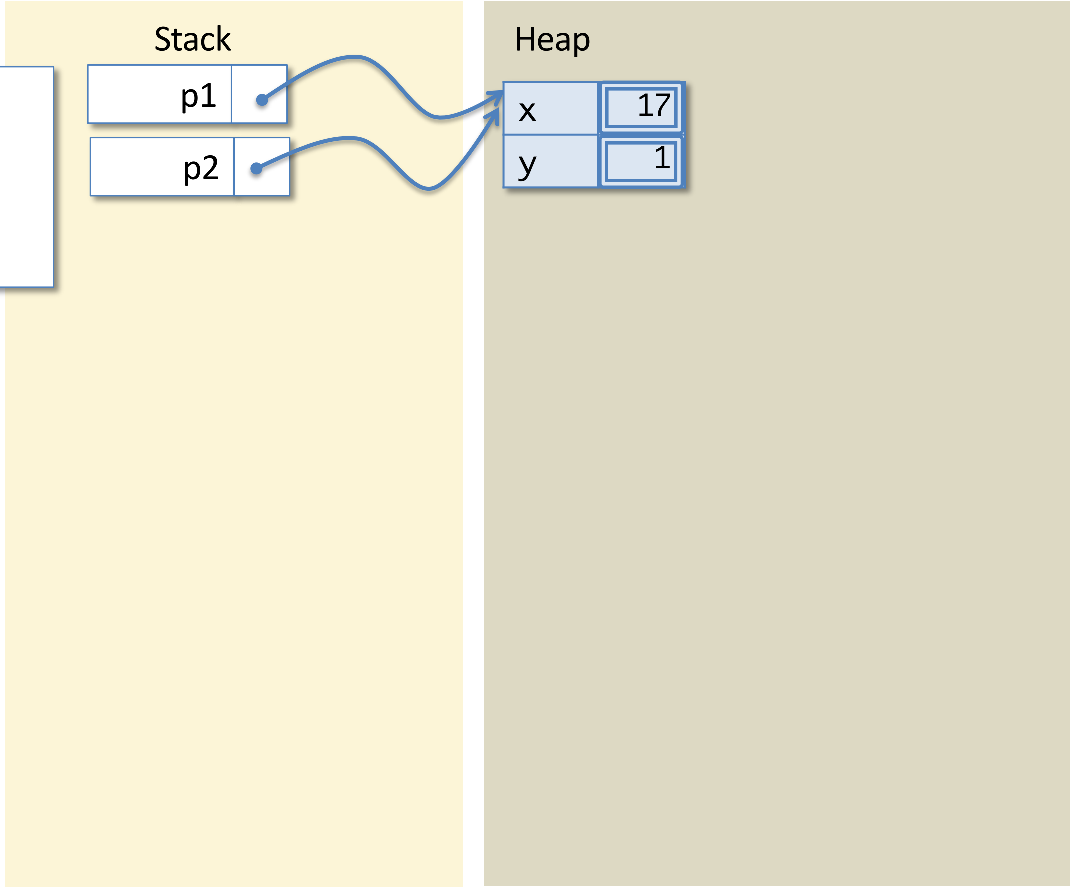
```
let ans : int =
    (); p1.x
```

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

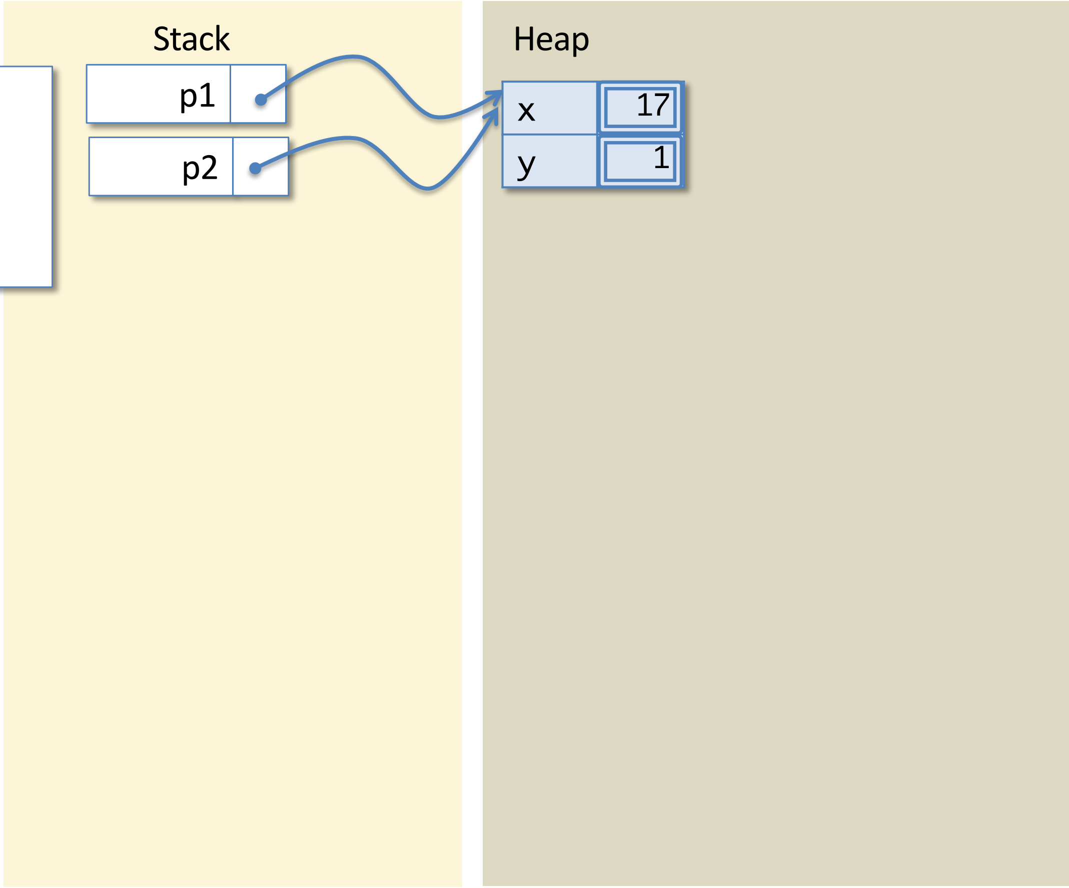
```
let ans : int =
    ~~();~~ p1.x
```



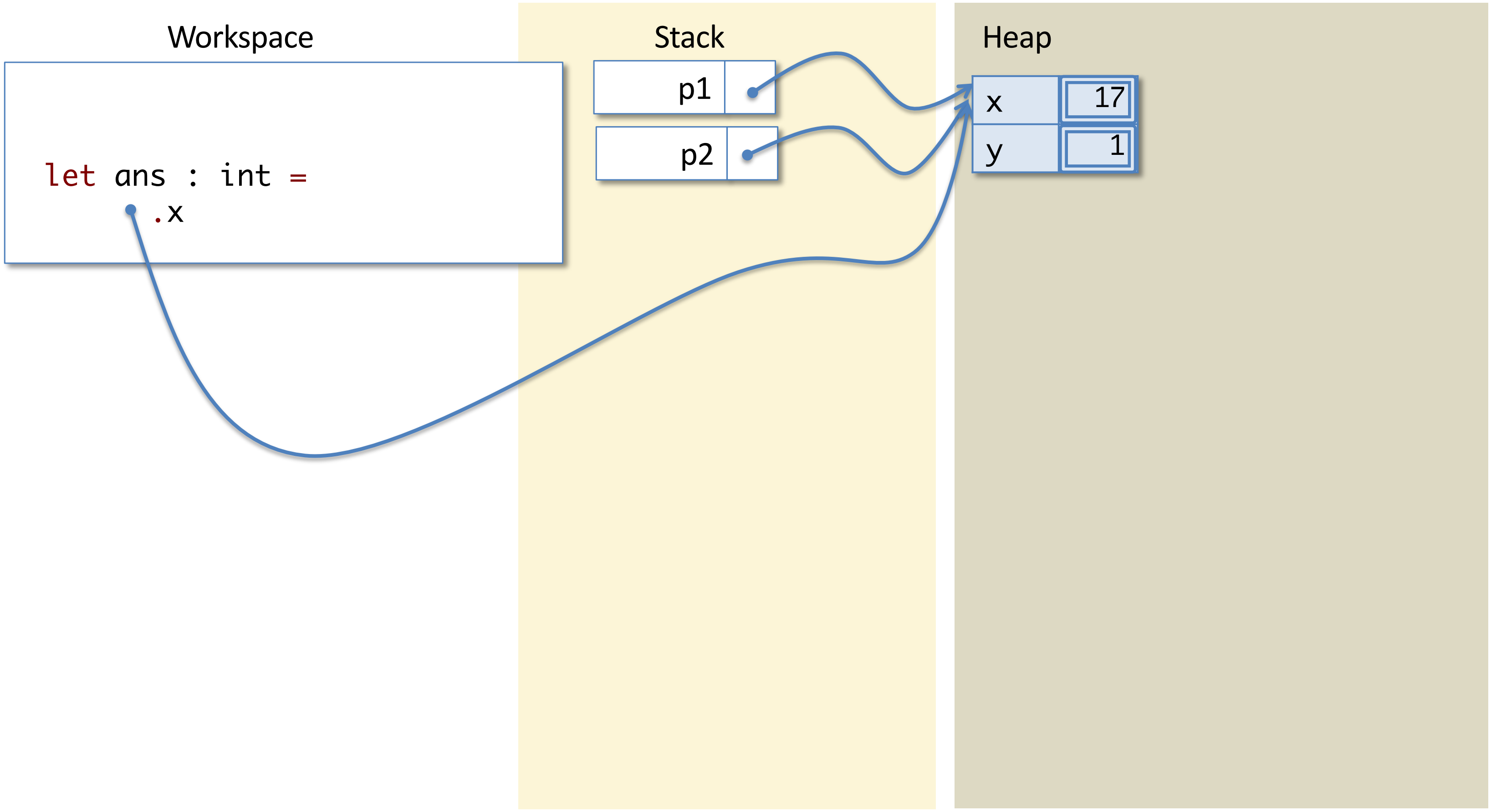
Look Up ‘p1’

Workspace

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



Look Up ‘p1’



Workspace

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

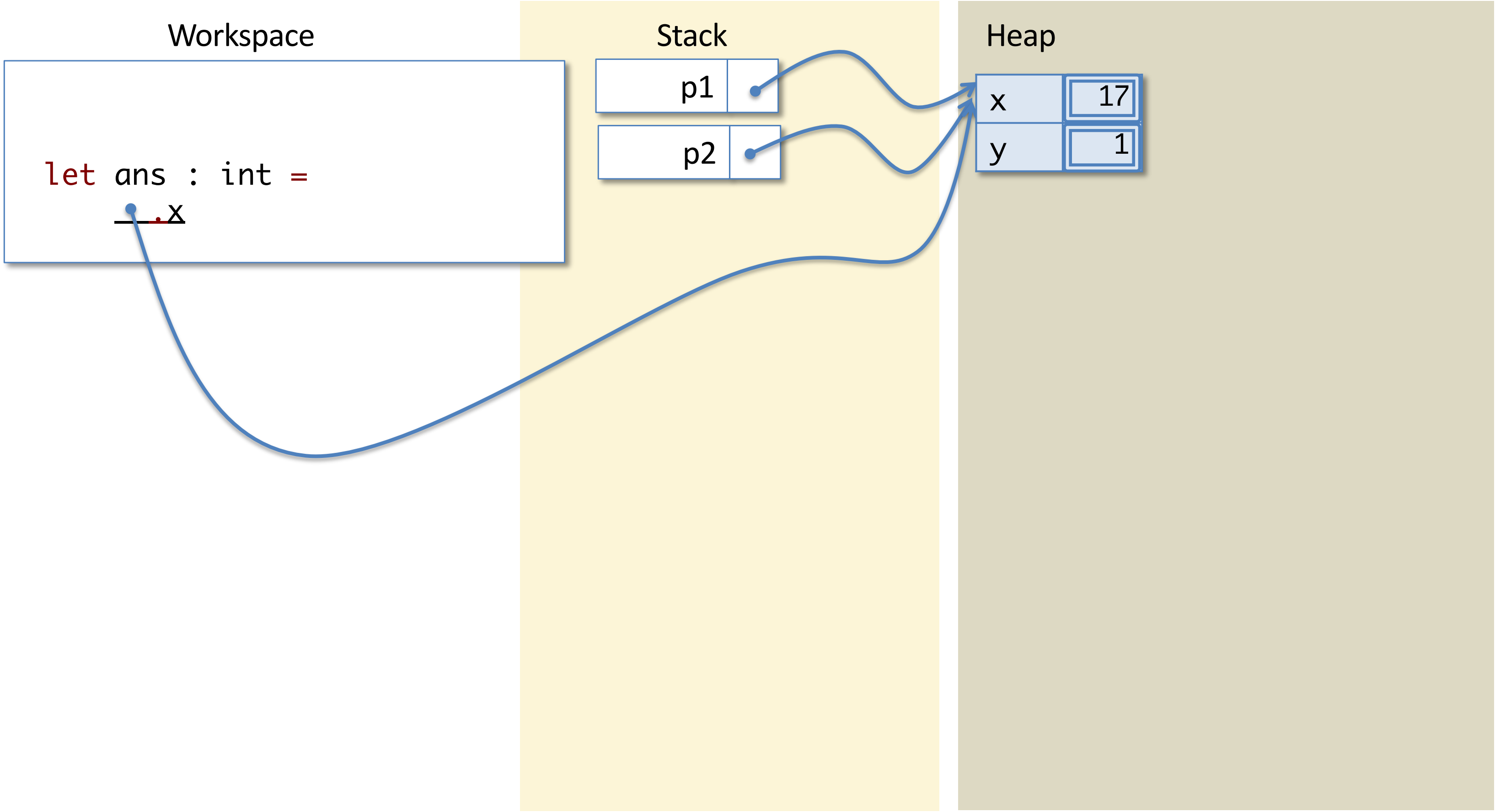
Stack

p1	●
p2	●

Heap

x	17
y	1

Project the ‘x’ field

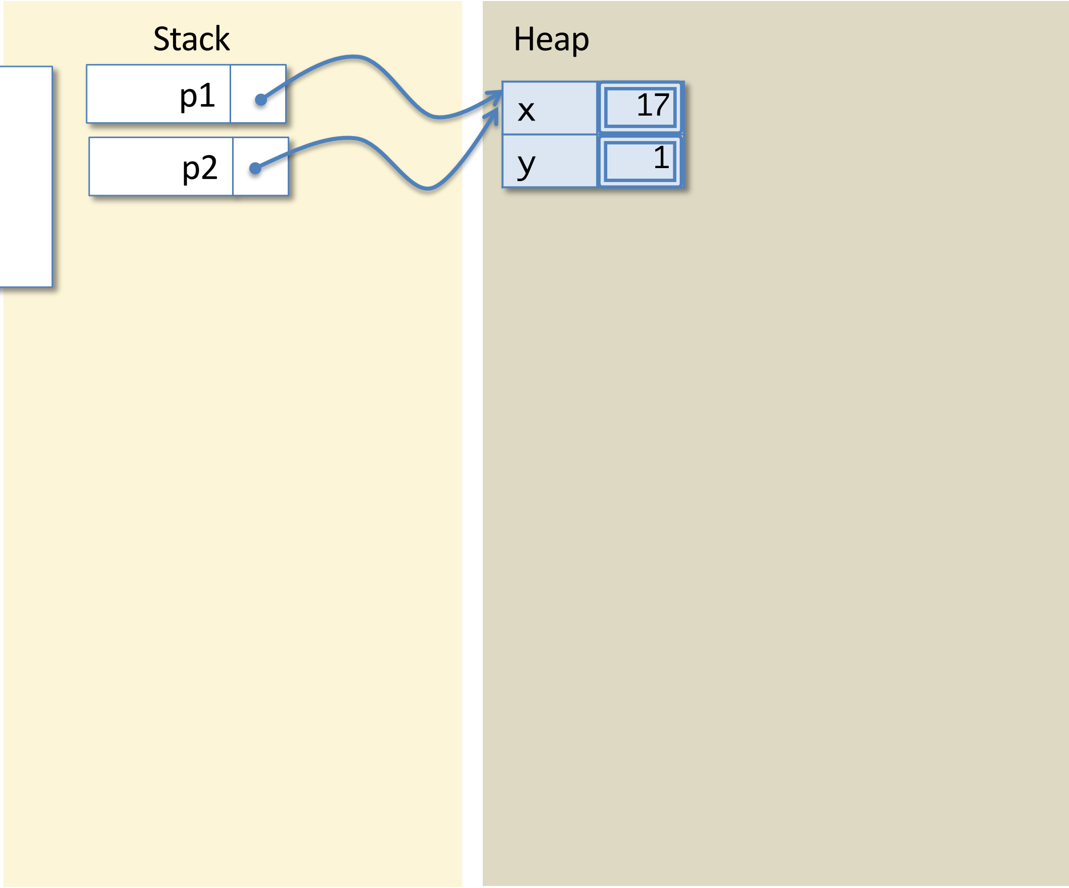


```
let ans : int =
    _.x
```

Project the ‘x’ field

Workspace

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



Stack

p1

p2

Heap

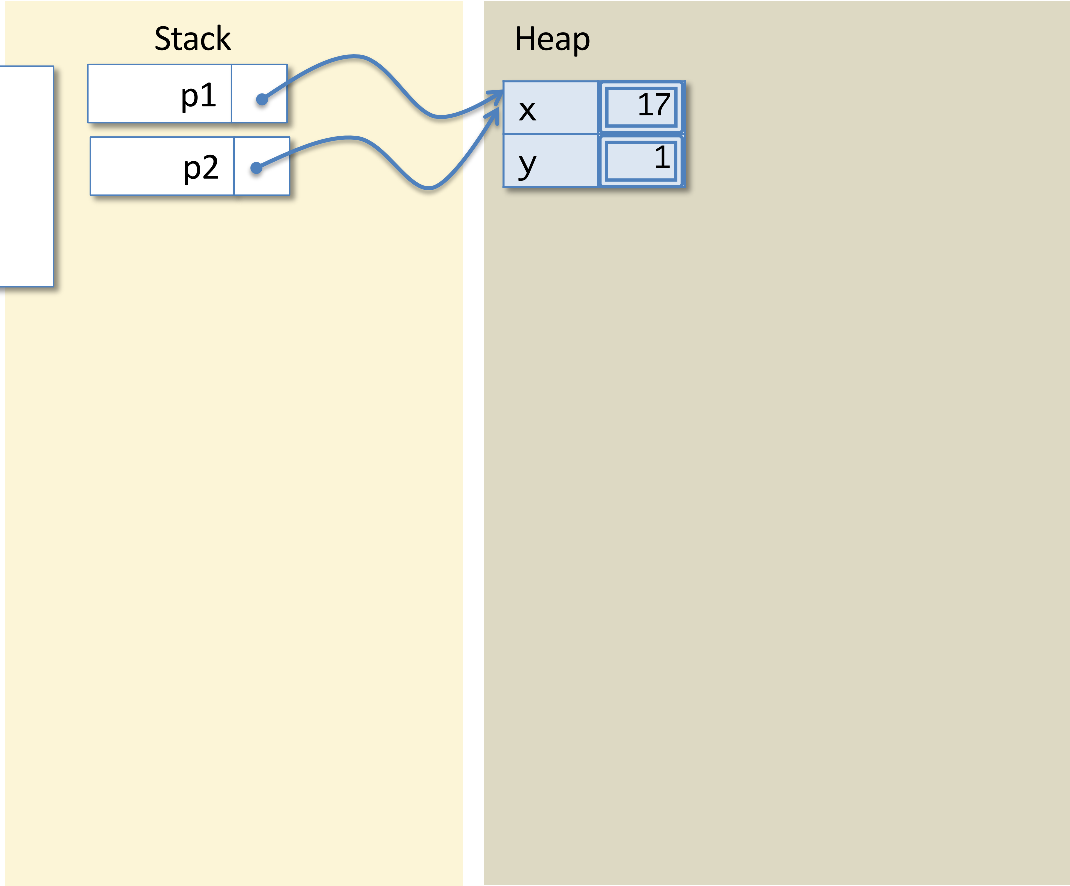
x 17

y 1

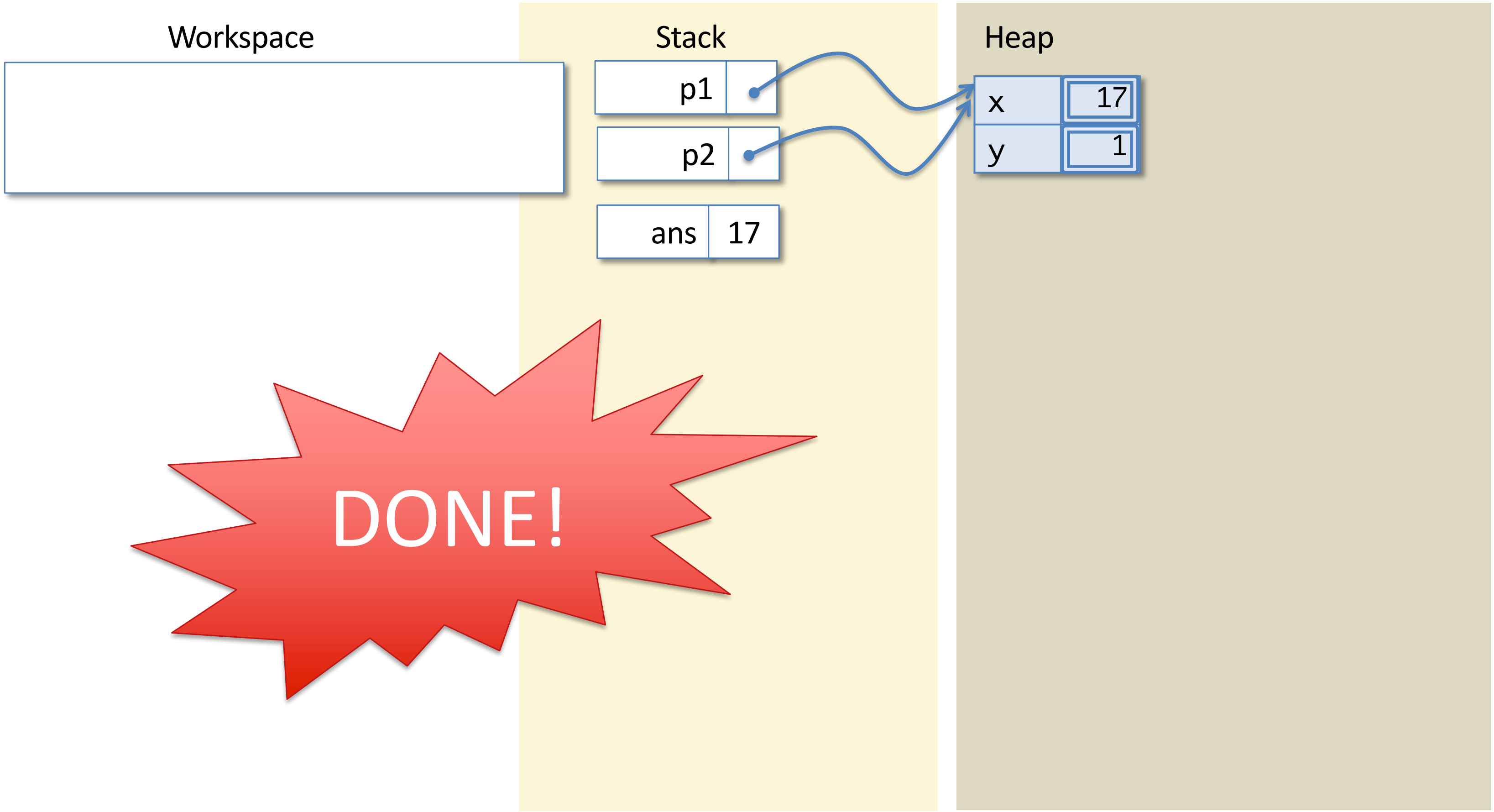
Let Expression

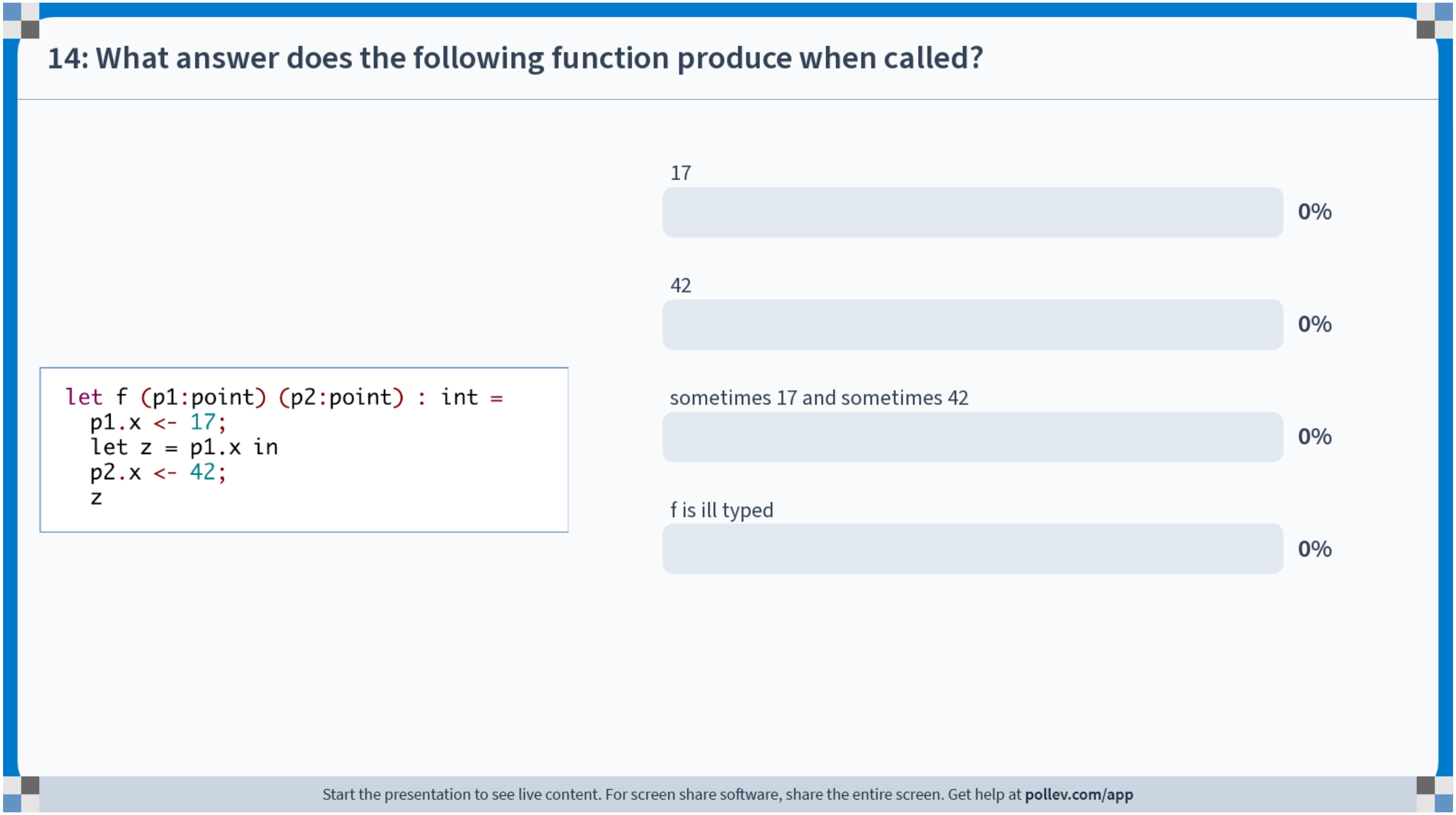
Workspace

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



Push ans





14: What answer does the following function produce when called?

```
let f (p1:point) (p2:point) : int =
  p1.x <- 17;
  let z = p1.x in
  p2.x <- 42;
  z
```

17

0%

42

0%

sometimes 17 and sometimes 42

0%

f is ill typed

0%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

What answer does the following function produce when called?

```
let f (p1:point) (p2:point) : int =
  p1.x <- 17;
  let z = p1.x in
  p2.x <- 42;
  z
```

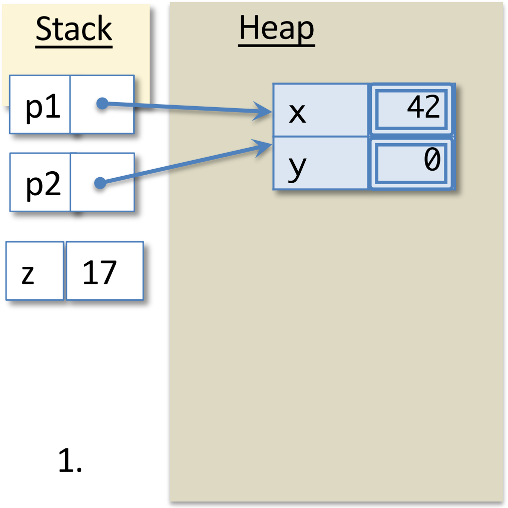
1. 17
2. 42
3. sometimes 17 and sometimes 42
4. f is ill typed

Answer: 1

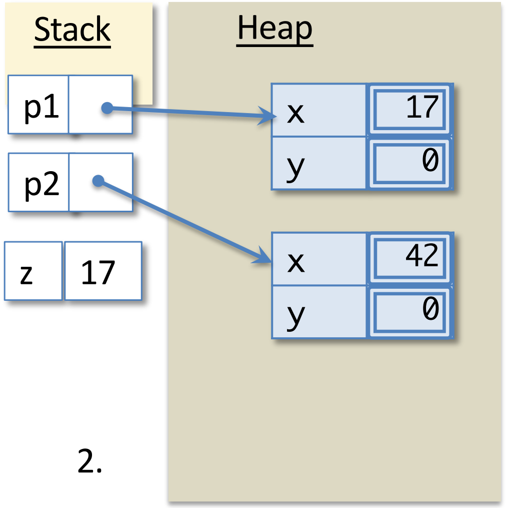
What do the Stack and Heap look like after simplifying the following code on the workspace?

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

1.



2.



Answer: 1

References and Equality

= vs. ==

Reference Equality

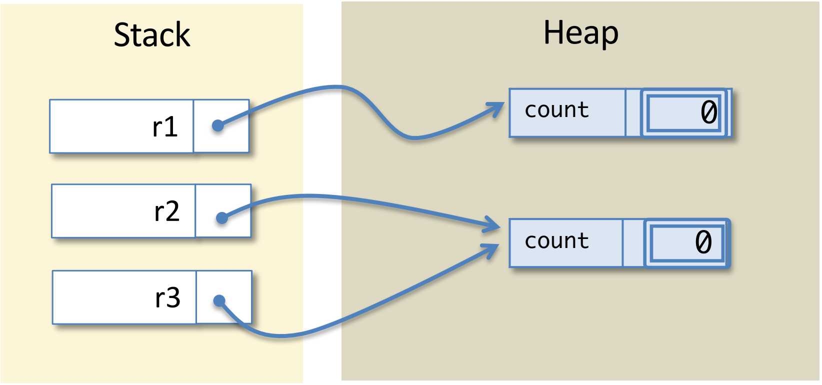
- Suppose we have two counters. Are they at the same location?

```
type counter = { mutable count : int }
let c1 : counter = …
let c2 : counter = …
```

  - We could increment one and see whether the other’s value changes.
  - But we could also just test whether the references are **aliases**.

- OCaml uses ‘==‘ to mean *reference* equality:
  - two reference values are ‘==‘ if they point to the same location in the heap; so:

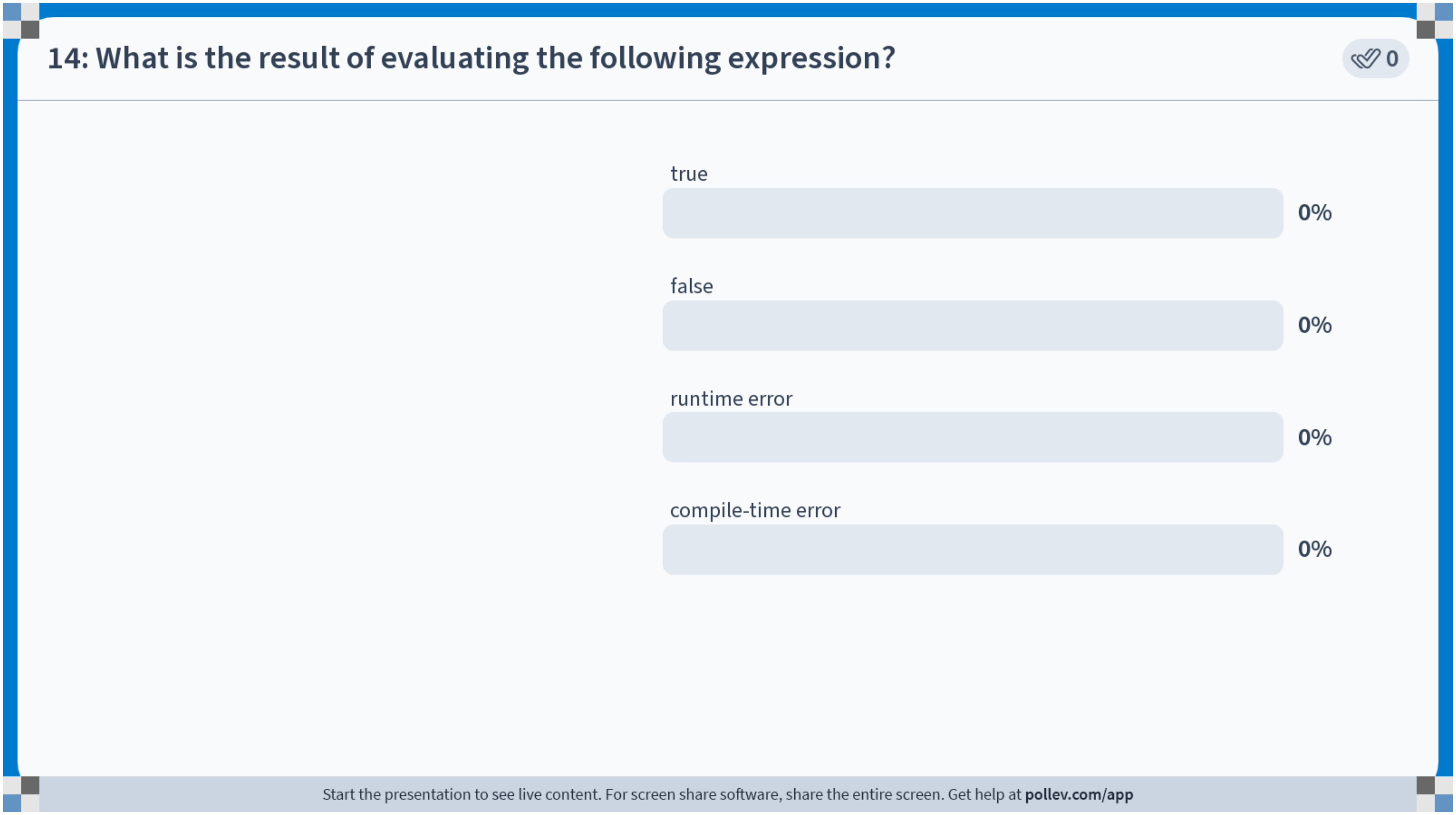
```
r2 == r3

not (r1 == r2)

r1 = r2
```



Structural vs. Reference Equality

- *Structural (in)equality*: `v1 = v2` `v1 <> v2`
  - recursively traverses over the *structure* of the data, comparing the two values' components for structural equality
  - function values cannot be compared structurally
  - structural equality can go into an infinite loop on cyclic structures
  - appropriate for comparing *immutable* datatypes

- *Reference (in)equality*: `v1 == v2` `v1 != v2`
  - Only looks at where the two references point in the heap
  - function values are only equal to themselves
  - even if v1 = v2, we may not have v1 == v2
  - appropriate for comparing *mutable* datatypes



14: What is the result of evaluating the following expression?

✓ 0

true
0%

false
0%

runtime error
0%

compile-time error
0%

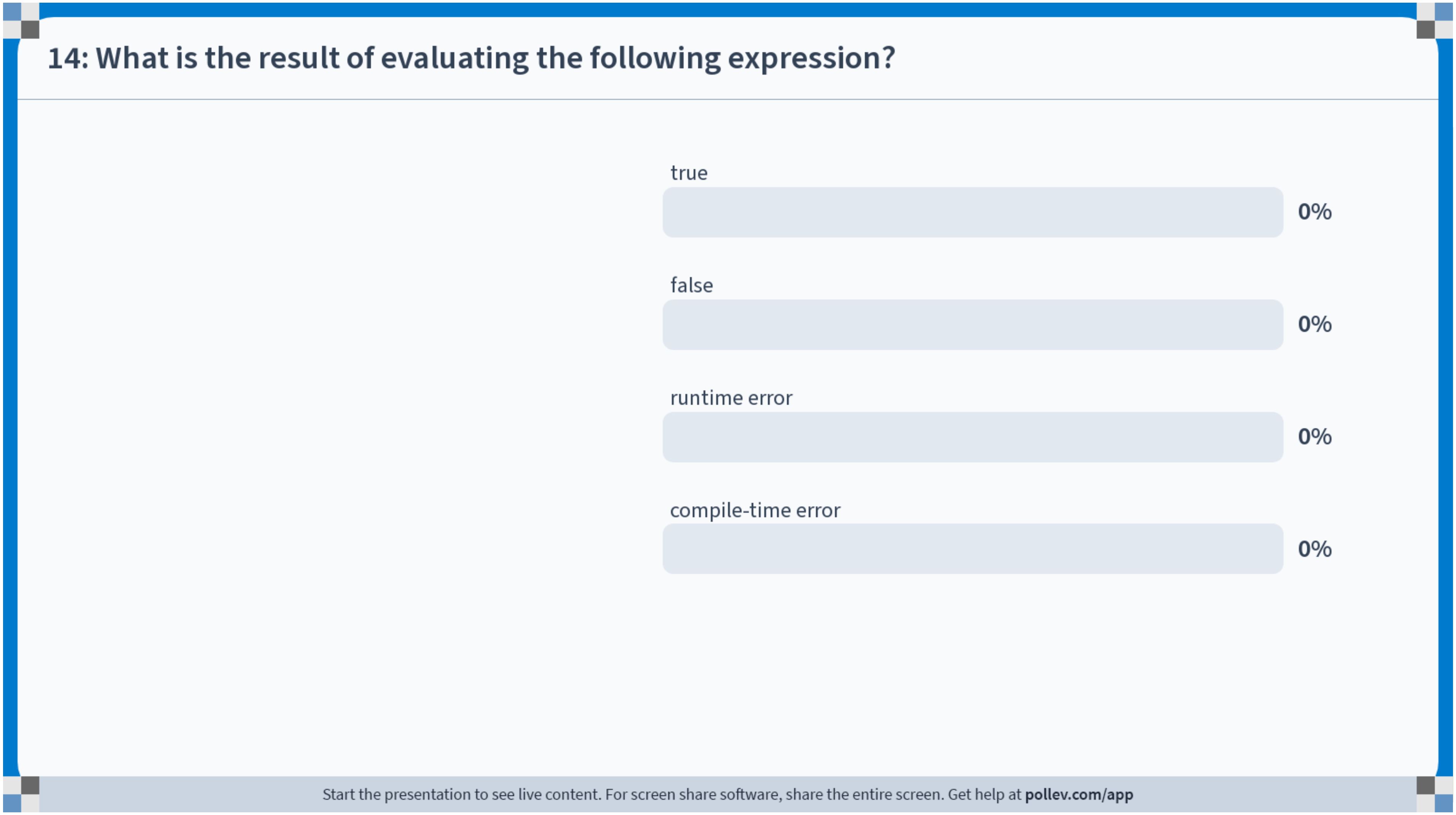
Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0 } in
let p2 : point = p1 in

p1 = p2
```

1. `true`
2. `false`
3. runtime error
4. compile-time error

Answer: true



14: What is the result of evaluating the following expression?

true

0%

false

0%

runtime error

0%

compile-time error

0%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0 } in
let p2 : point = p1 in

p1 == p2
```

1. `true`
2. `false`
3. runtime error
4. compile-time error

Answer: true

What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0 } in
let p2 : point = { x = 0; y = 0 } in

p1 == p2
```

1. true
2. false

Answer: false

What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0 } in
let p2 : point = { x = 0; y = 0 } in
let l1 : point list = [p1] in
let l2 : point list = [p2] in

l1 = l2
```

1. true
2. false

Answer: true

What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0 } in
let p2 : point = p1 in
let l1 : point list = [p1] in
let l2 : point list = [p2] in

l1 == l2
```

1. true
2. false

Answer: false

ASM: Lists and datatypes

Tracking the space usage of *immutable* data structures

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

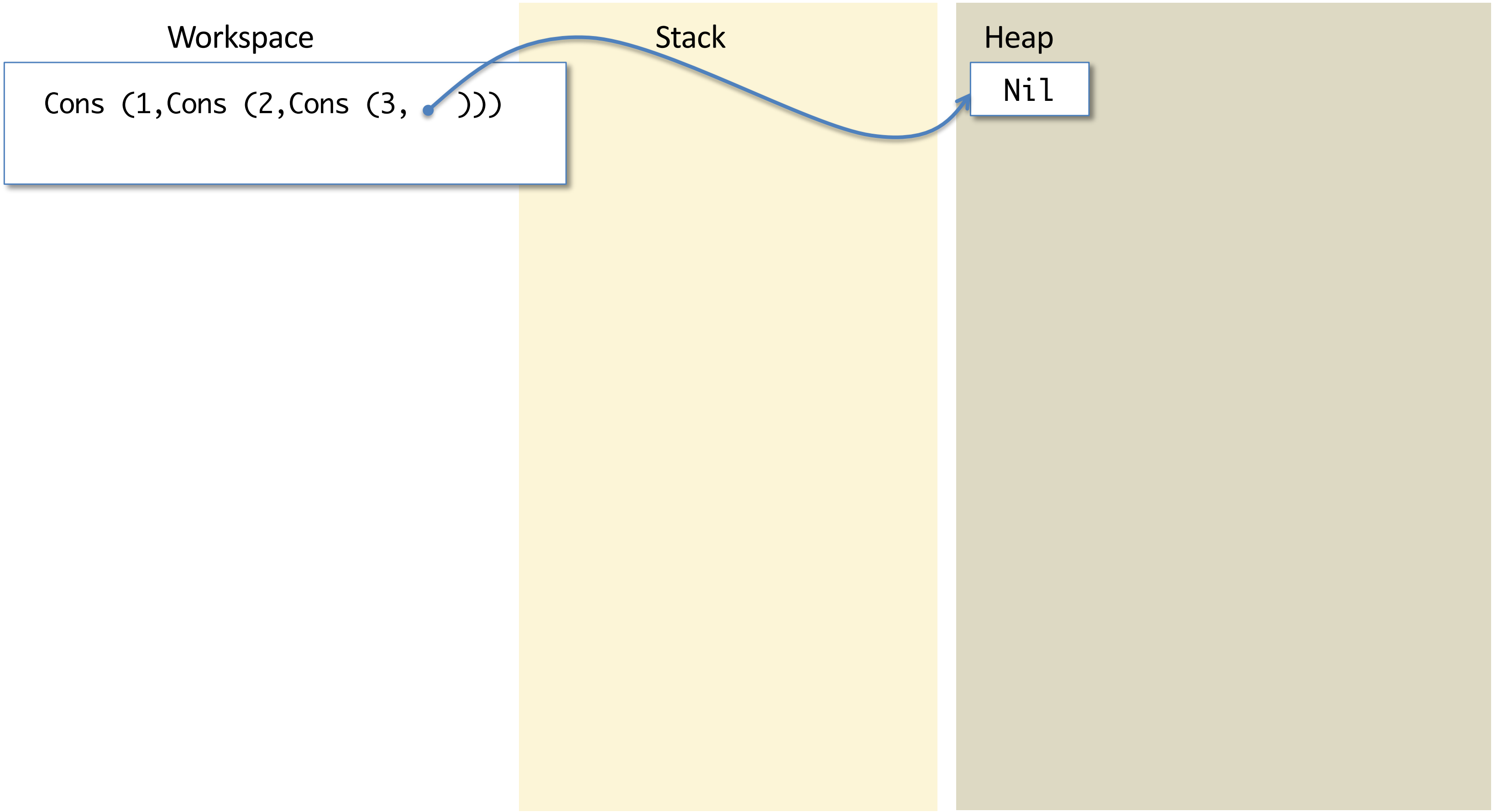
Workspace

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

Stack

Heap

Simplification



Workspace

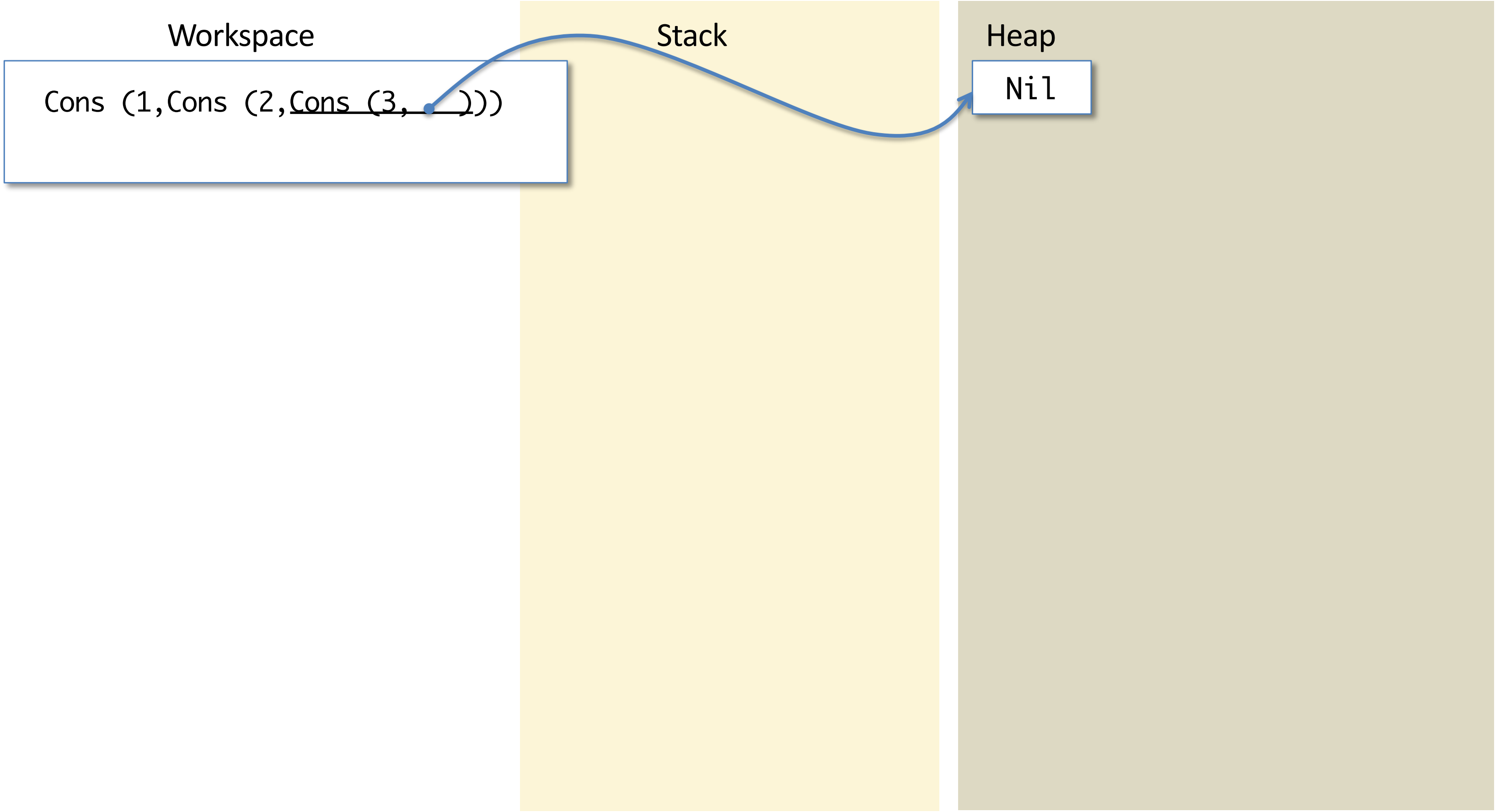
```
Cons (1,Cons (2,Cons (3,  )))
```

Stack

Heap

```
Nil
```

Simplification



Workspace

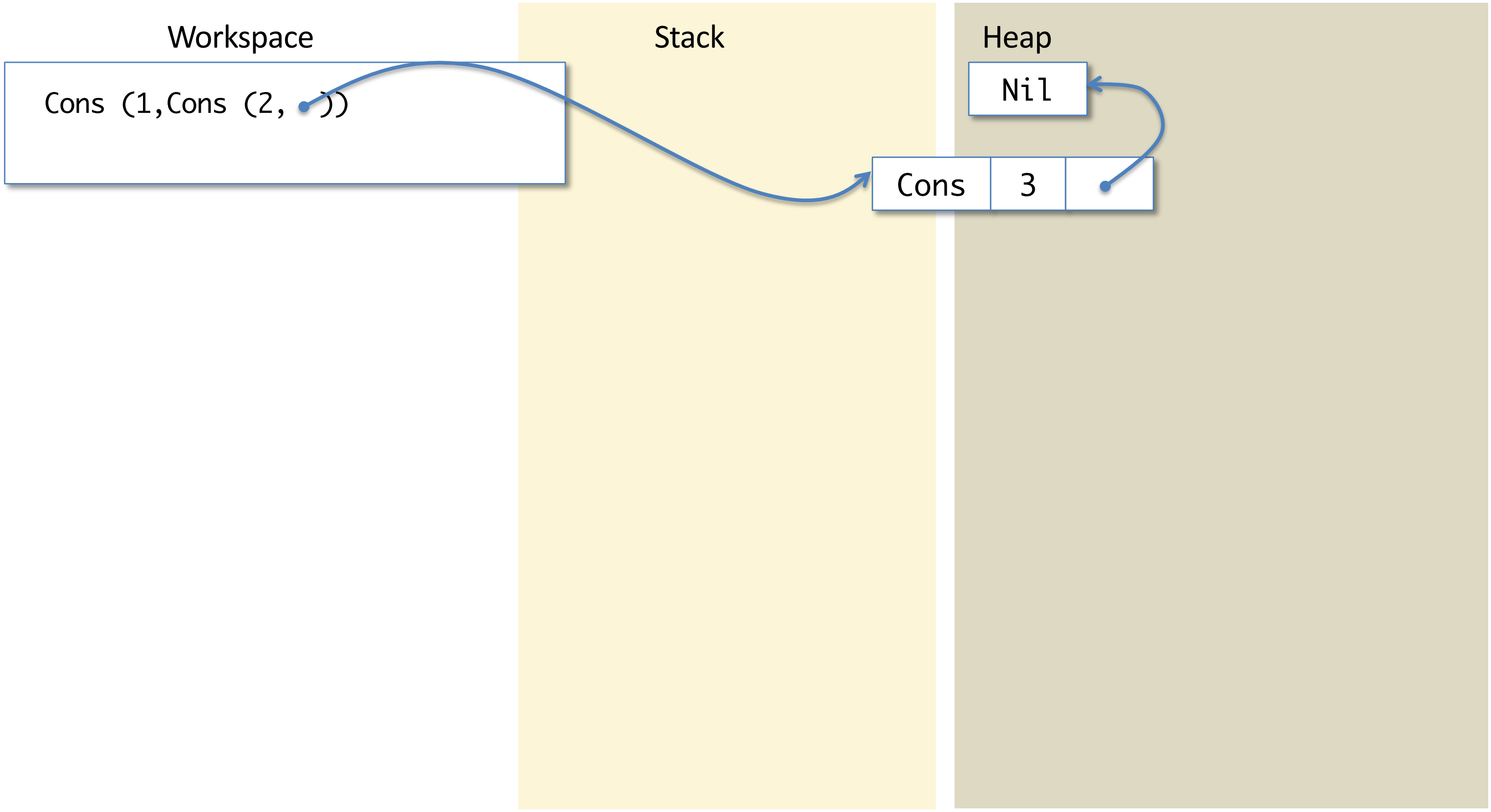
Stack

Heap

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

Nil

Simplification



Workspace

Stack

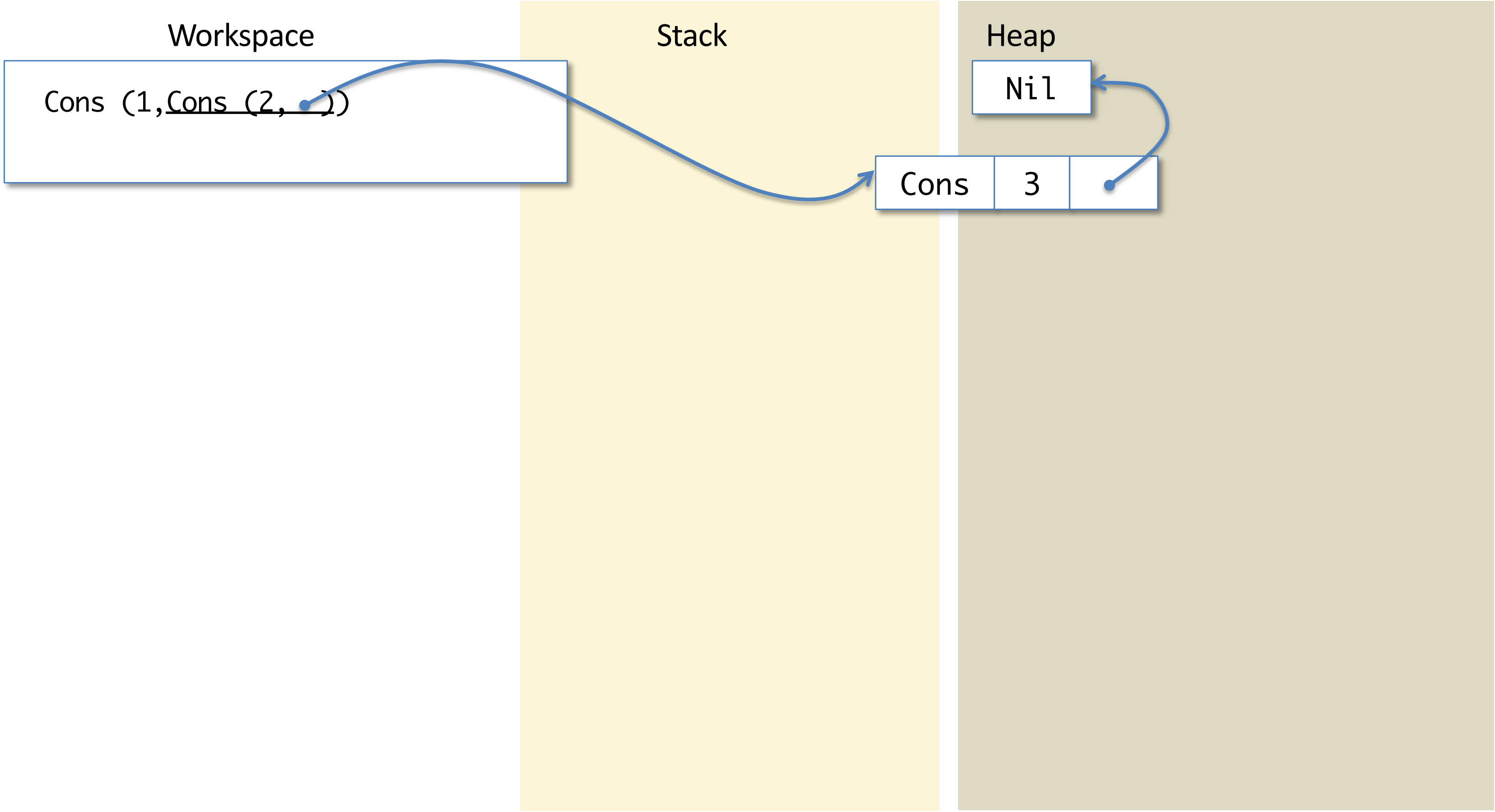
Heap

```
Cons (1,Cons (2, ))
```

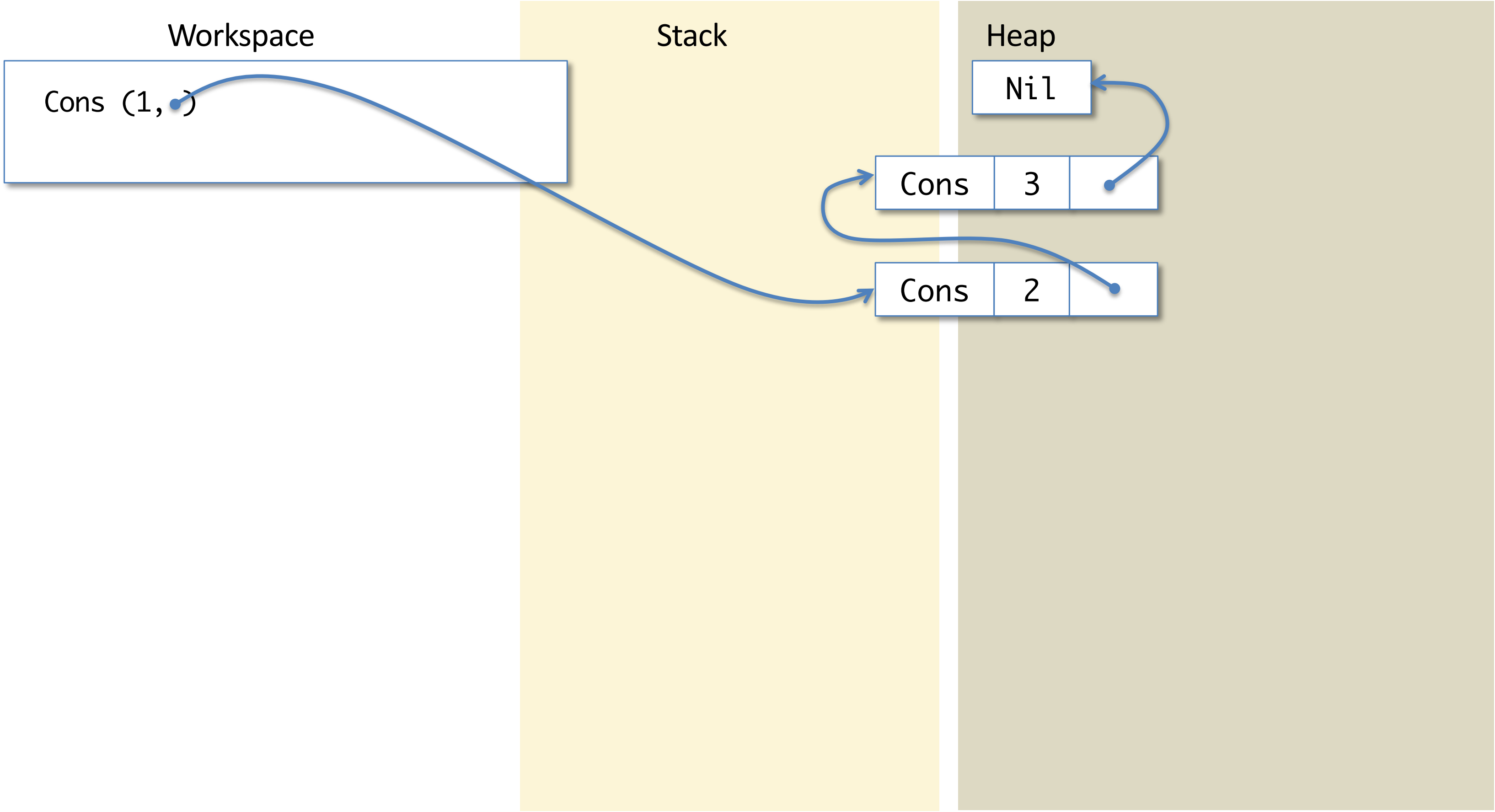
Nil

Cons 3

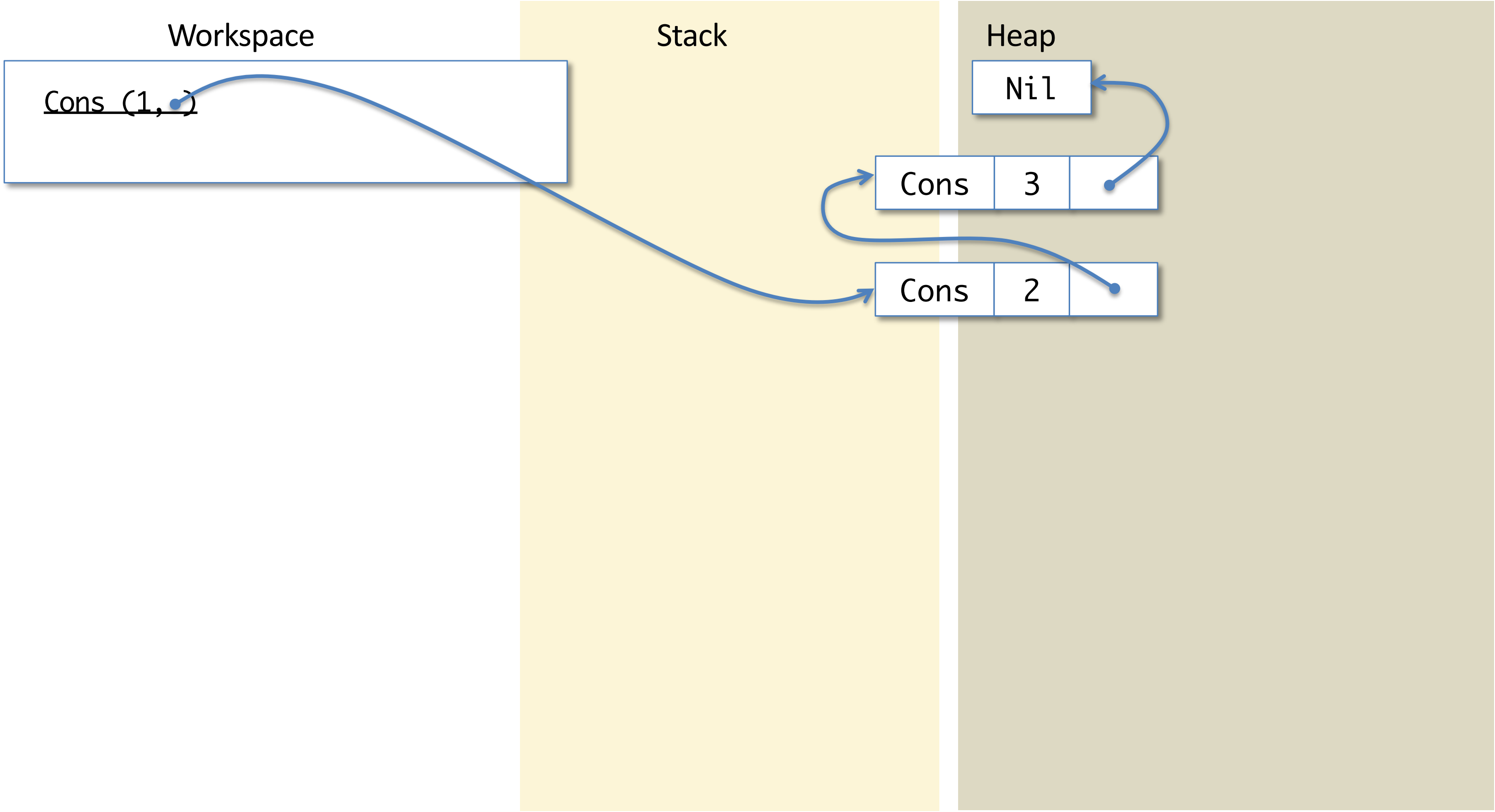
Simplification



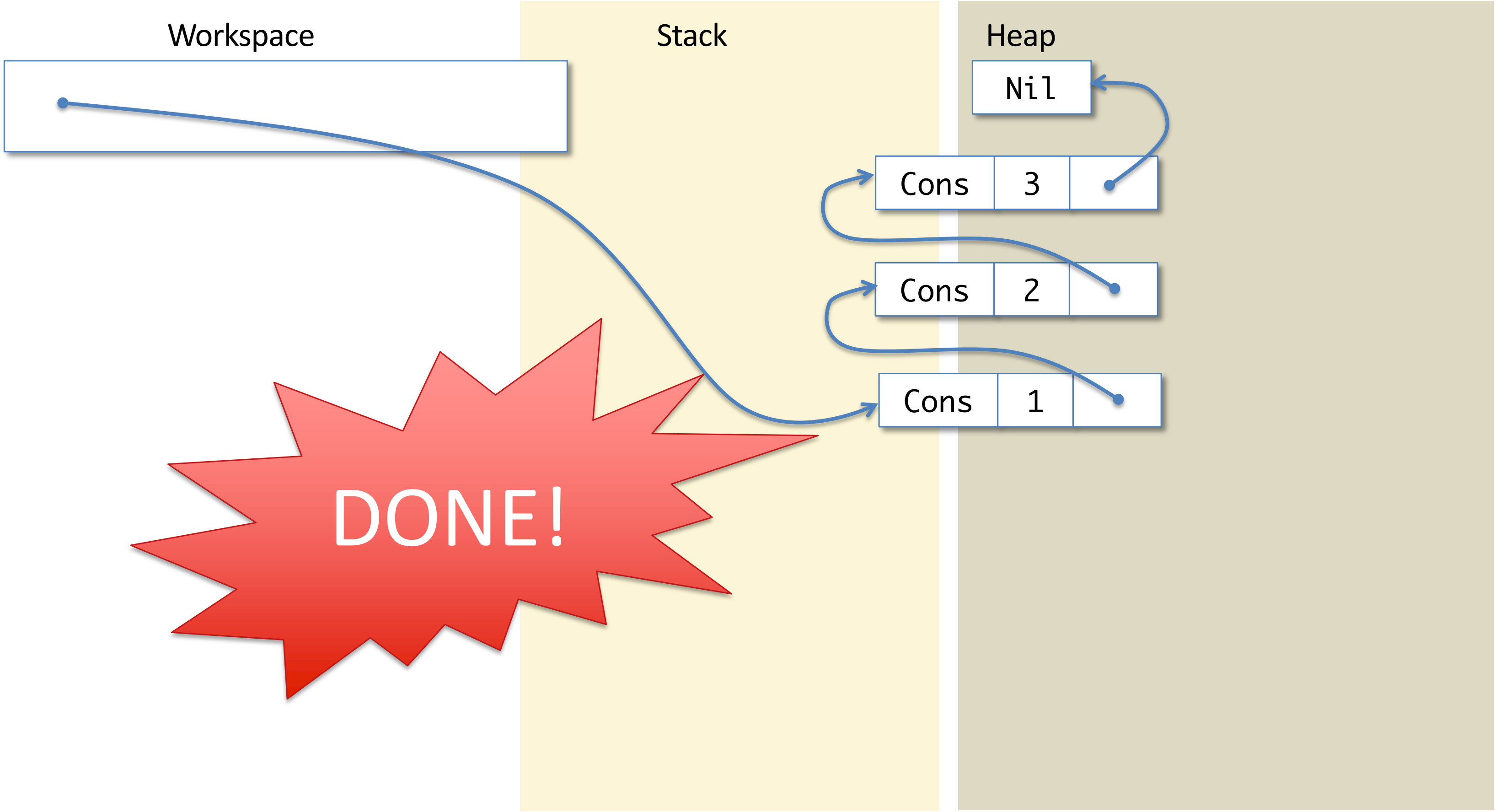
Simplification



Simplification

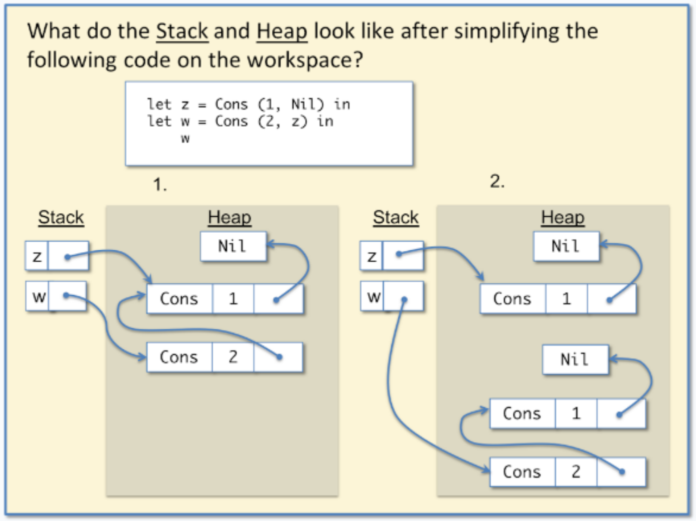


Simplification



15: Simplifying code on the ASM

0



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

0%

2

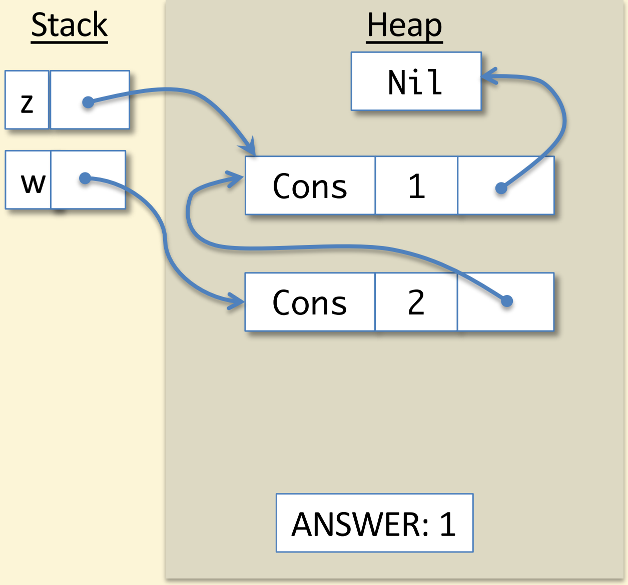
0%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

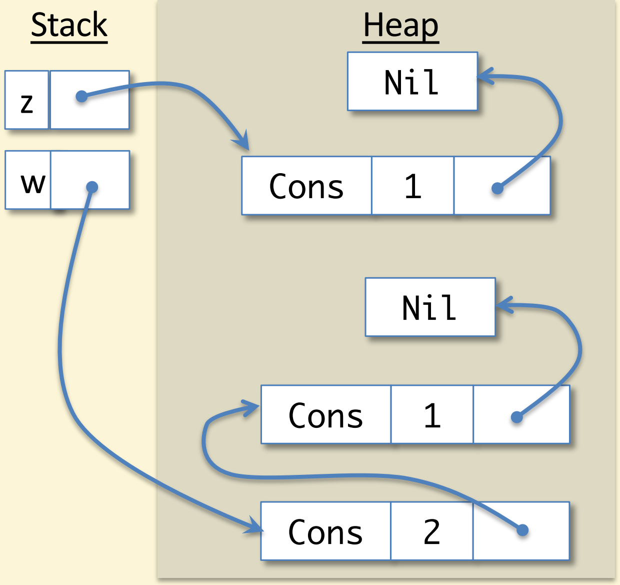
What do the Stack and Heap look like after simplifying the following code on the workspace?

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

1.



2.



An Optimization

- Datatype constructors that carry no extra information can be treated as “small” values.
- Examples:

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

```
type ‘a option =
| None
| Some of ‘a
```

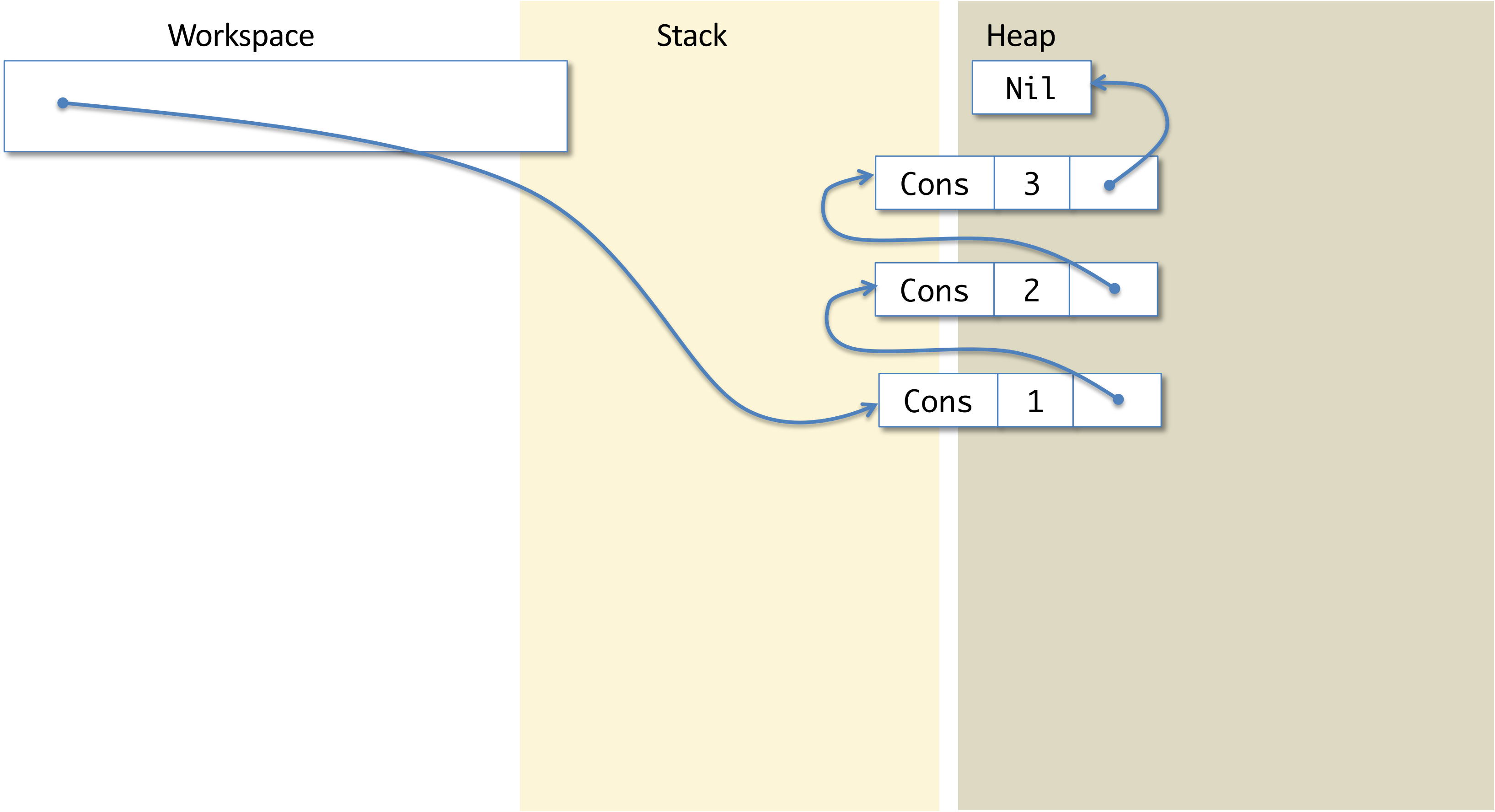
```
type ‘a tree =
| Empty
| Node of ‘a tree * ‘a * ‘a tree
```

- They can be placed directly in the stack.
- They don’t require a reference in the heap.

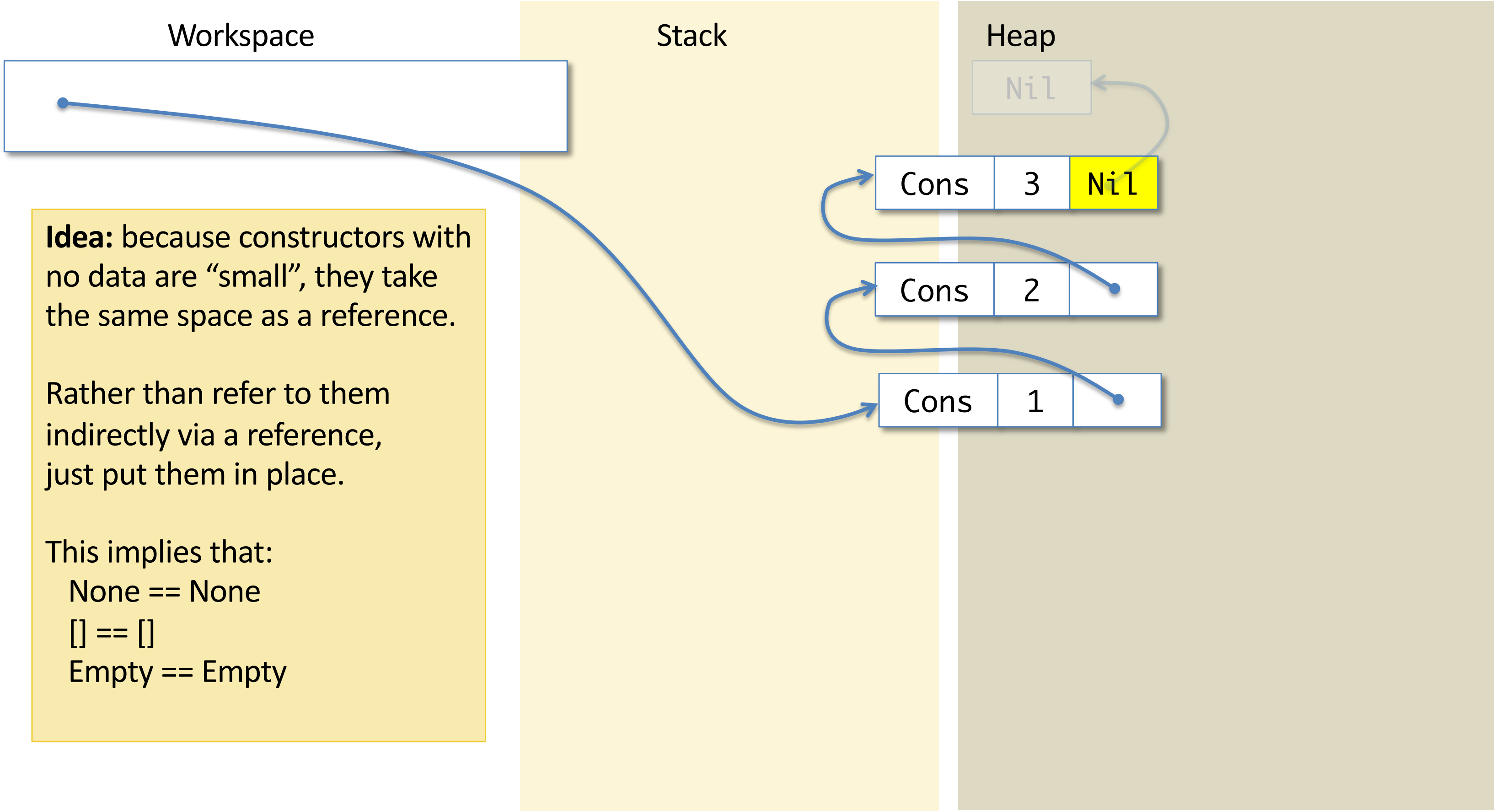
Saves space!

- N.b.: This optimization affects reference equality.

Example Optimization



Example Optimization



Idea: because constructors with no data are "small", they take the same space as a reference.

Rather than refer to them indirectly via a reference, just put them in place.

This implies that:
None == None
[] == []
Empty == Empty