Programming Languages and Techniques (CIS120)

Lecture 13

Mutable State & The Abstract Stack Machine

Chapters 14 &15

Announcements

- Midterm grading in progress
  - Scores will be released after make-up exams are finished (by Weds.)
- Homework 4
  - now available now, due Tuesday next week
- Lecture Section 002:
  - Dr. Sheth will be away Weds. & Fri.
  - Dr. Zdancewic will give those lectures

Commands and Unit

unit: the first-class type

- Can define values of type unit

```
let x : unit = ()
```

```
val x : unit
```

- Can pattern match unit (even in function definitions)

```
let z = begin match x with
  | () -> 4
end
```

```
fun () -> 3
```

- Is the result of an implicit else branch:

```
;; if z <> 4 then
     failwith "oops"
```

=

```
;; if z <> 4 then
     failwith "oops"
   else ()
```

Sequencing Commands and Expressions

- Expressions of type unit are useful because of their *side effects* – they "*do*" stuff
  - e.g. printing, changing the value of mutable state

```
let f (x:int) : int =
  print_string "f called with ";
  print_string (string_of_int x);
  x + x
```

do *not* use ';' here!

note the use of ';' here

- We can think of ';' as an infix function of type:

```
unit -> 'a -> 'a
```

What is the type of f in the following program:

```
let f (x:int) =
   print_int (x + x)
```

1. unit -> int
2. unit -> unit
3. int -> unit
4. int -> int
5. f is ill typed

Answer: 3

What is the type of f in the following program:

```
let f (x:int) =
   (print_int x);
   (x + x)
```

1. `unit -> int`
2. `unit -> unit`
3. `int -> unit`
4. `int -> int`
5. `f is ill typed`

Answer: 4

Records

Immutable 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 after 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;}
```

Why Pure Functional Programming?

- Simplicity
  - small language: arithmetic, local variables, recursive functions, datatypes, pattern matching, generic types/functions and modules
  - simple *substitution* model of computation
- Persistent data structures
  - Nothing changes; retains all intermediate results
  - Good for version control, fault tolerance, etc.
- Typechecker can give more helpful errors
  - Once your program compiles, it needs less testing
  - Options vs. NullPointerException
- Easier to parallelize and distribute
  - No implicit interactions between parts of the program.
  - All of the behavior of a function is specified by its arguments

Mutable State

Mutable Record Fields

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

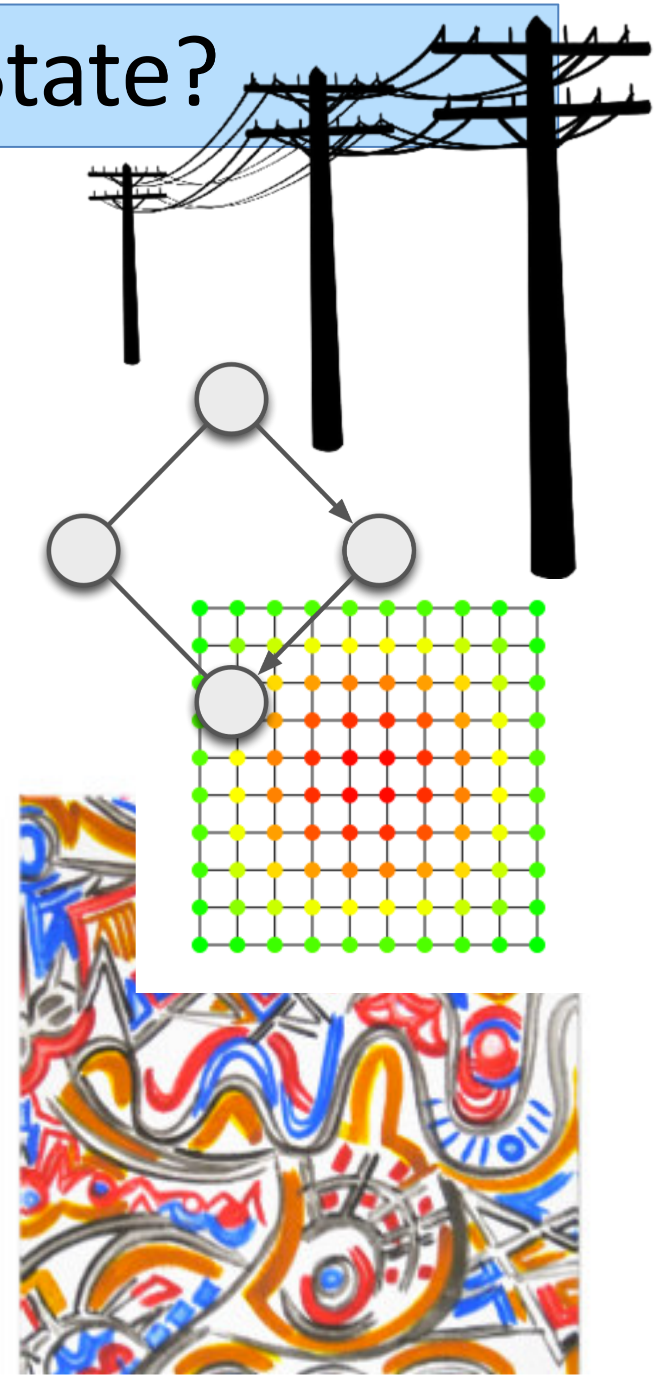
- Functions can assign to mutable record fields
- Note that the return type of ‘<-’ is unit
  - i.e., it is a *command*

```
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
```

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

Why Use Mutable State?



- Action at a distance
  - allow remote parts of a program to communicate / share information without threading the information through all the points in between
- Data structures with explicit sharing
  - e.g. graphs
  - without mutation, it is only possible to build trees – no cycles
- Efficiency/Performance
  - A few data structures have imperative implementations with better asymptotic efficiency than the best declarative version
- Re-using space (in-place update)
- Random-access data (arrays)
- Direct manipulation of hardware
  - device drivers, displays, etc.

Different views of imperative programming

Java (and C, C++, C#)

- Code is a sequence of **statements** (a.k.a. commands) that do something, sometimes using expressions to compute values.
- References are **mutable** by default, must be explicitly declared to be constant

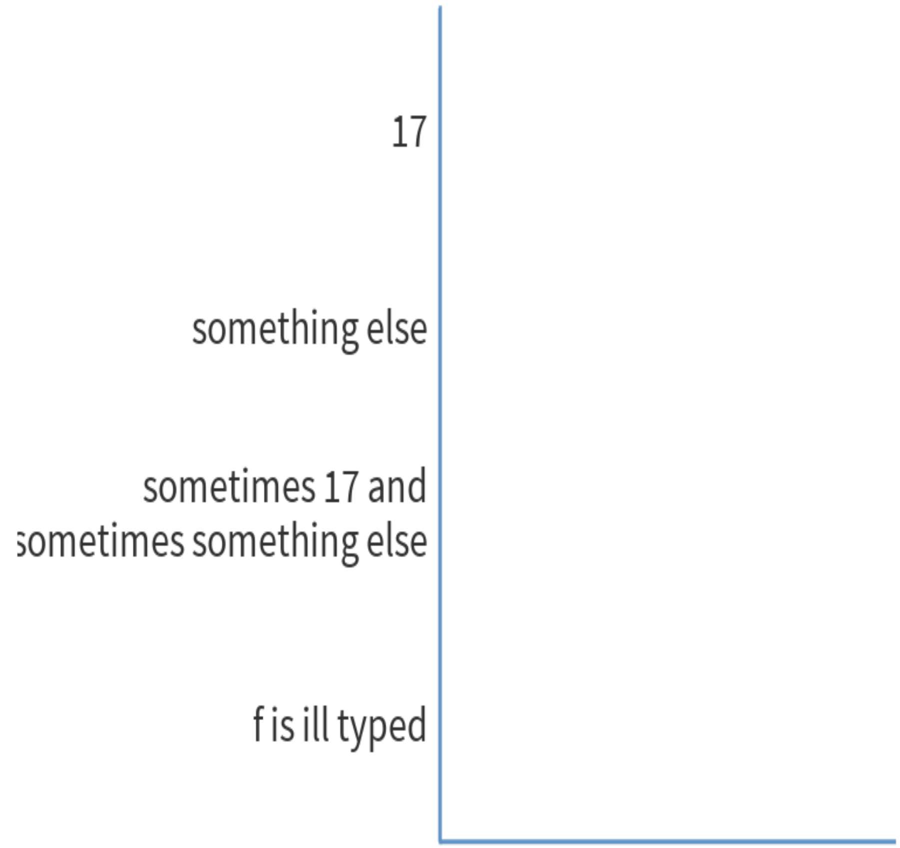
OCaml (and Haskell, etc.)

- Code is an **expression** that has a value. Sometimes computing that value has other effects.
- References are **immutable** by default, must be explicitly declared to be mutable

What answer does the following function produce when called?

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

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



What answer does the following function produce when called?

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

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

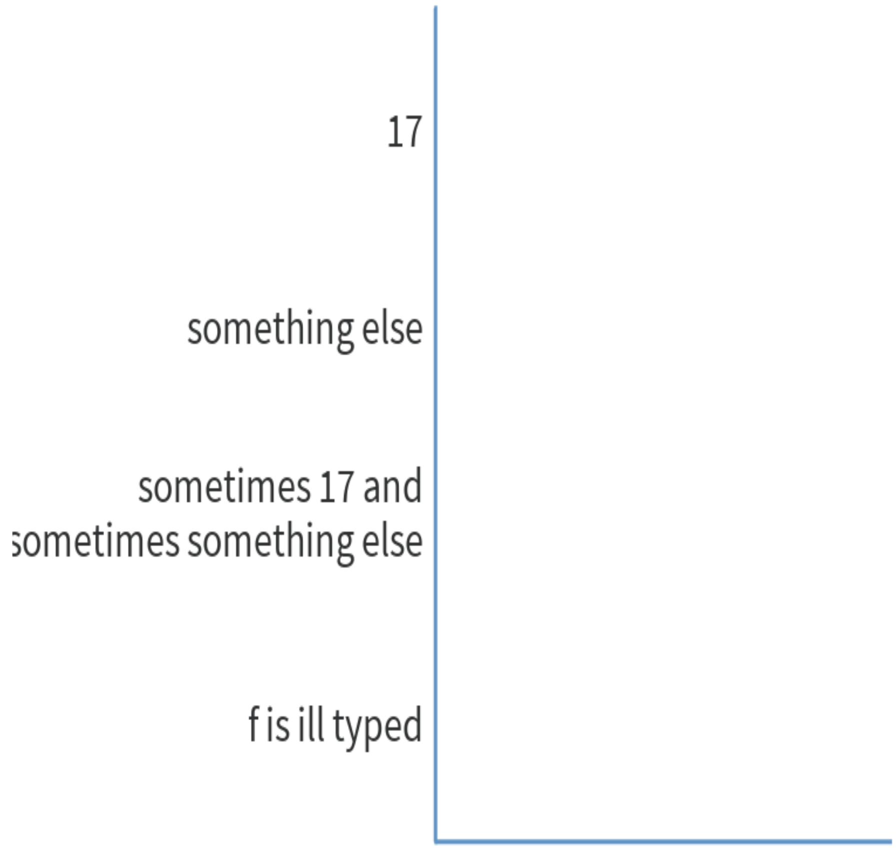
1. 17
2. something else
3. sometimes 17 and sometimes something else
4. f is ill typed

ANSWER: 1

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
```



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

What does this function return?

```
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

Opening a Whole New Can of Worms*

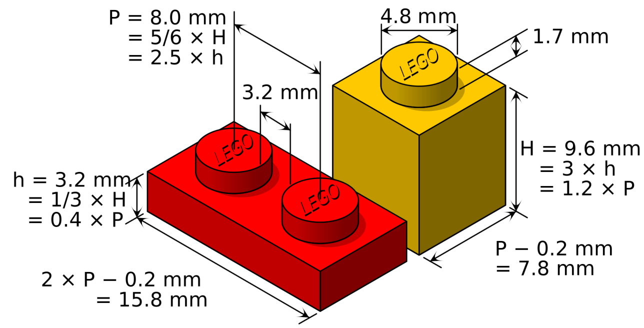
*t-shirt courtesy of ahrefs.com

Modeling State

Location, Location, Location!

Need for a New Computation Model

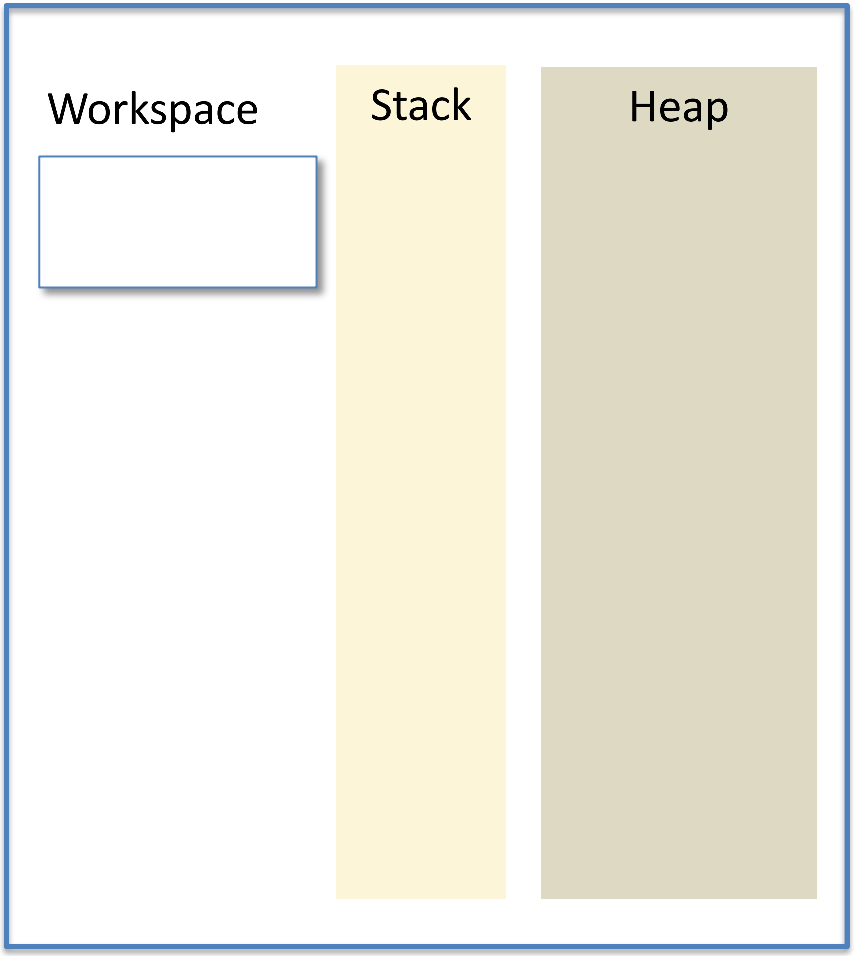
- A simple substitution model works well for pure value oriented programming
  - "Observable" behavior of a value is *completely* determined by its structure
  - Pure functions are *referentially transparent*: two different calls to the same function with the same arguments always yield the same results
  - These properties justify the the "replace equals by equals" model



- 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 working on simplifying
  - 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
  - (Sometimes this will also involve changes to the stack and/or heap)
  - Stop when there are no more simplifications to be done



Abstract stack machine

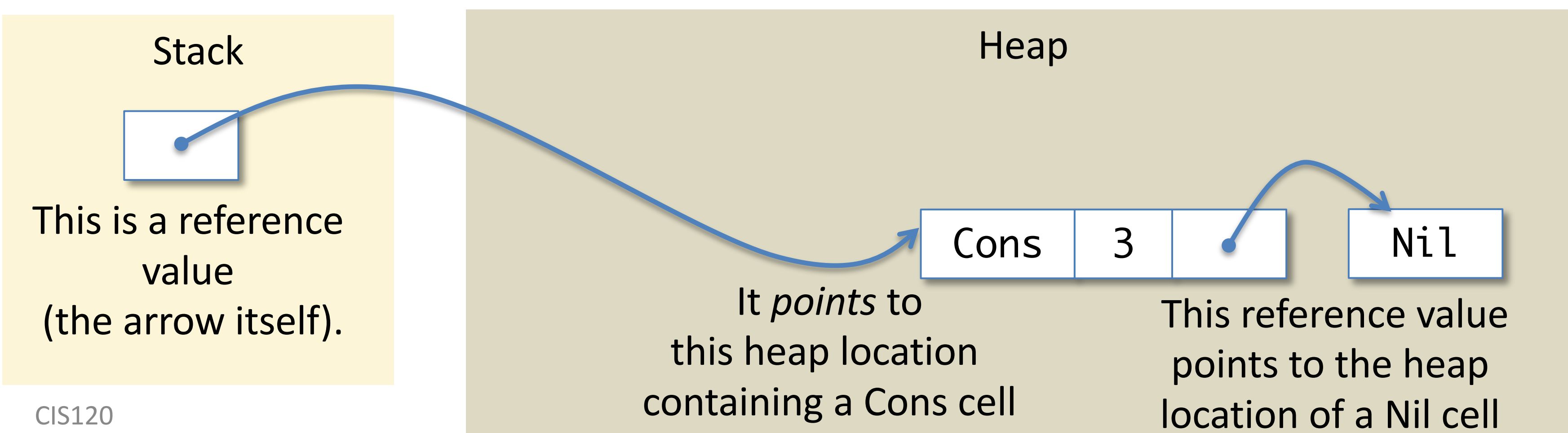
Values and References

A *value* is either:

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

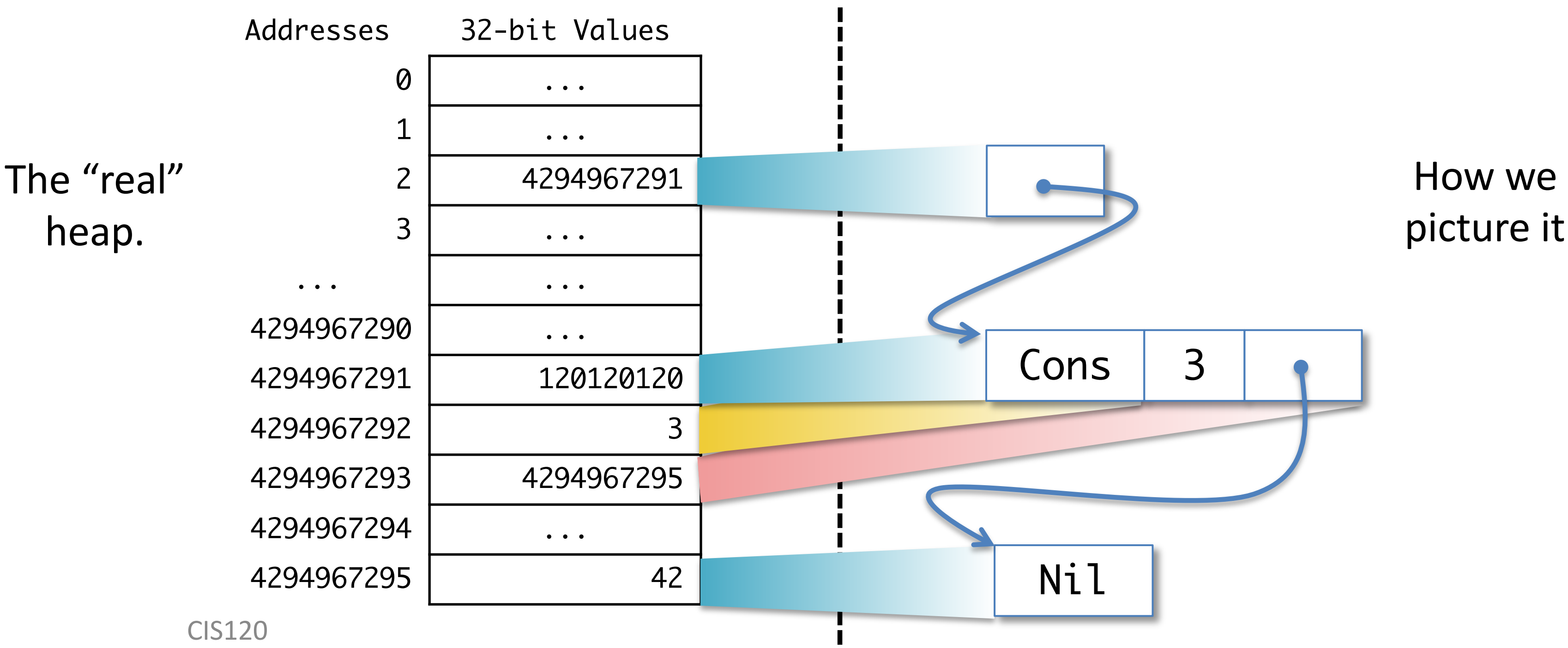
A reference is the *address (location)* of a piece of data in the heap. We draw a reference value as an "arrow":

- The start of the arrow is the reference itself (i.e. the address)
- The arrow "points" to the value located at this 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: Simplifying variables, operators, `let` expressions, and `if` expressions

Simplification

Workspace

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

Stack

Heap

What to simplify next?

- At each step, the ASM finds the left-most *ready subexpression* in the workspace
- An expression involving a ***primitive operator*** (eg "`+`") is *ready* if all its arguments are values
  - Expression is replaced with its result
- A ***let expression*** `let x : t = e in body` is *ready* if `e` is a value
  - A new binding for `x` to `e` is added at the end of the stack
  - let expression is replaced with `body` in the workspace
- A ***variable*** is always *ready*
  - The variable is replaced by its binding in the stack, searching from the most recent bindings
- A ***conditional expression*** `if e then e1 else e2` is *ready* if `e` is either `true` or `false`
  - The workspace is replaced with either `e1` (if `e` is `True`) or `e2` (if e is `False`)

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 proceeds from most recent entries to the least recent entries. Note that the "top" (most recent part) of the stack is drawn 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

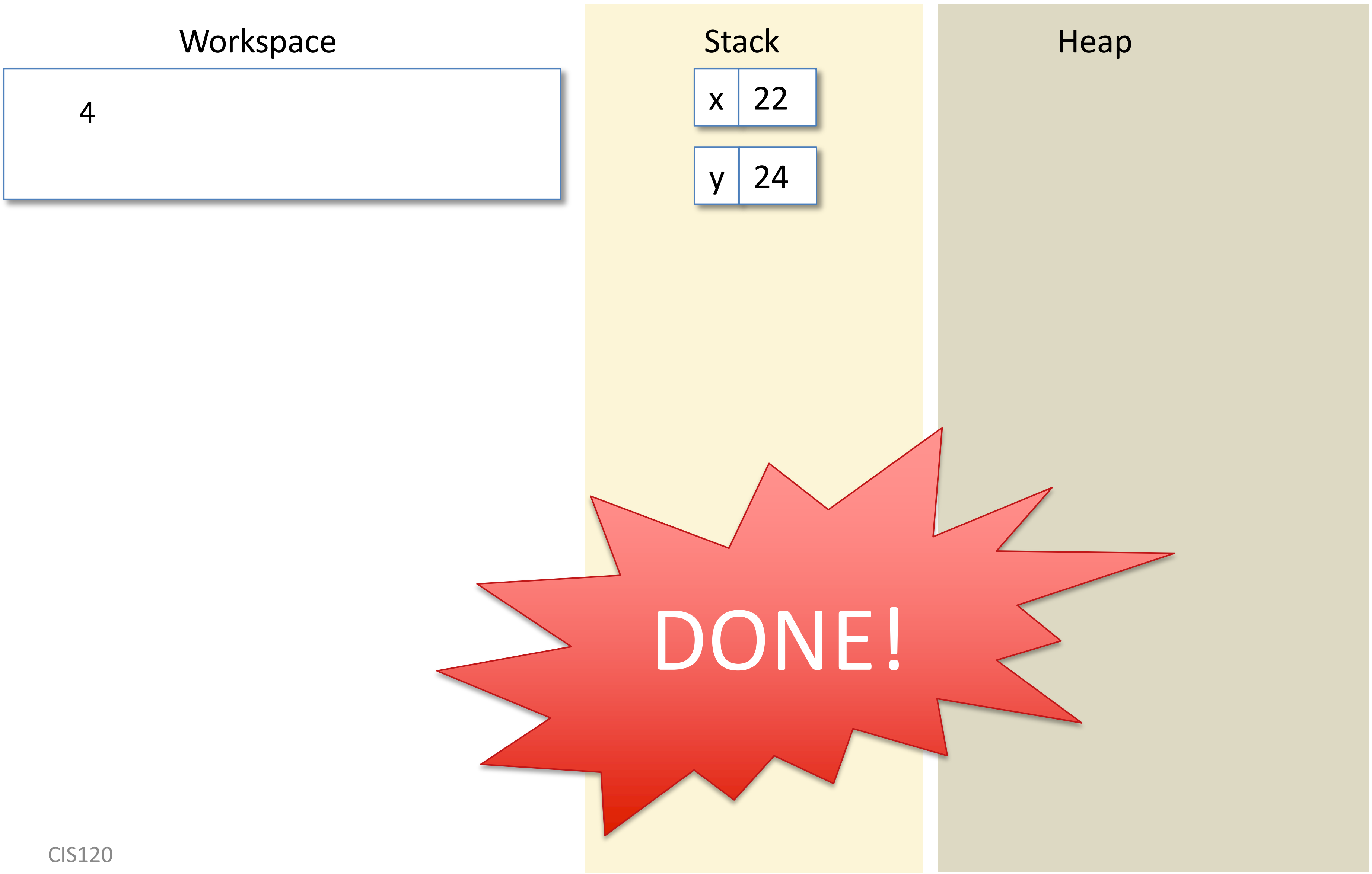
```
if false then 3 else 4
```

Stack

x	22
y	24

Heap

Simplification



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

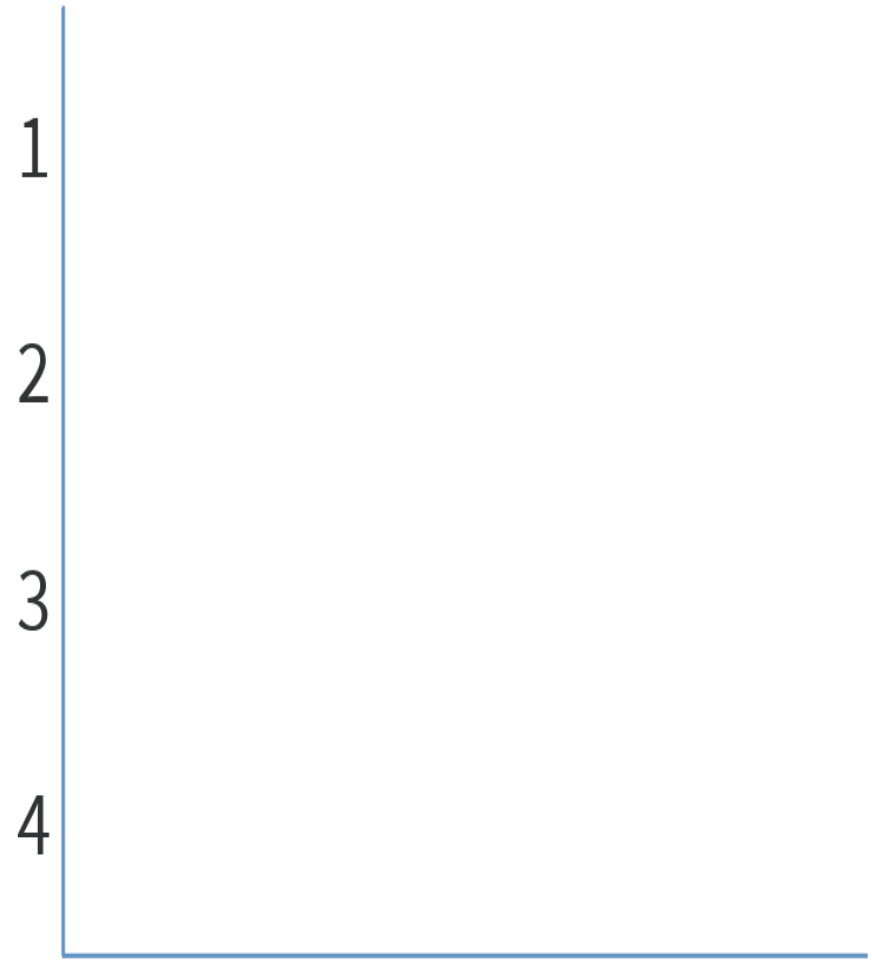
ANSWER: 2

Simplifying code on the ASM

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	2. Stack	3. Stack	4. Stack
z 22	z 20	w 22	w 22
w 2 + z	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		2. Stack		3. Stack		4. Stack	
z	22	z	20	z	22	z	22
z	20	z	22			z	22

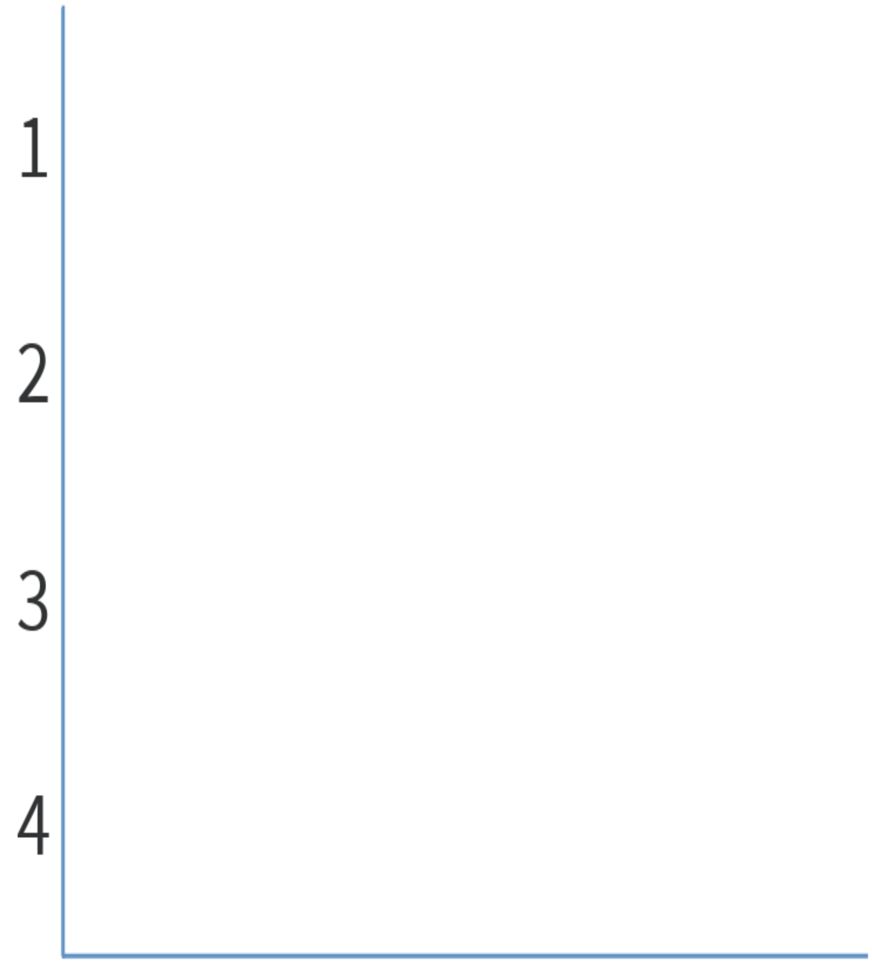
ANSWER: 2

Simplifying code on the ASM

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	Stack	Stack	Stack
z | 22	z | 20	z | 22	z | 22
z | 20	z | 22		z | 22
1.	2.	3.	4.



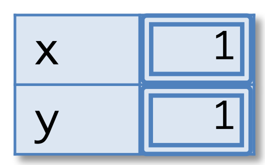
Mutable Records

- 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
```

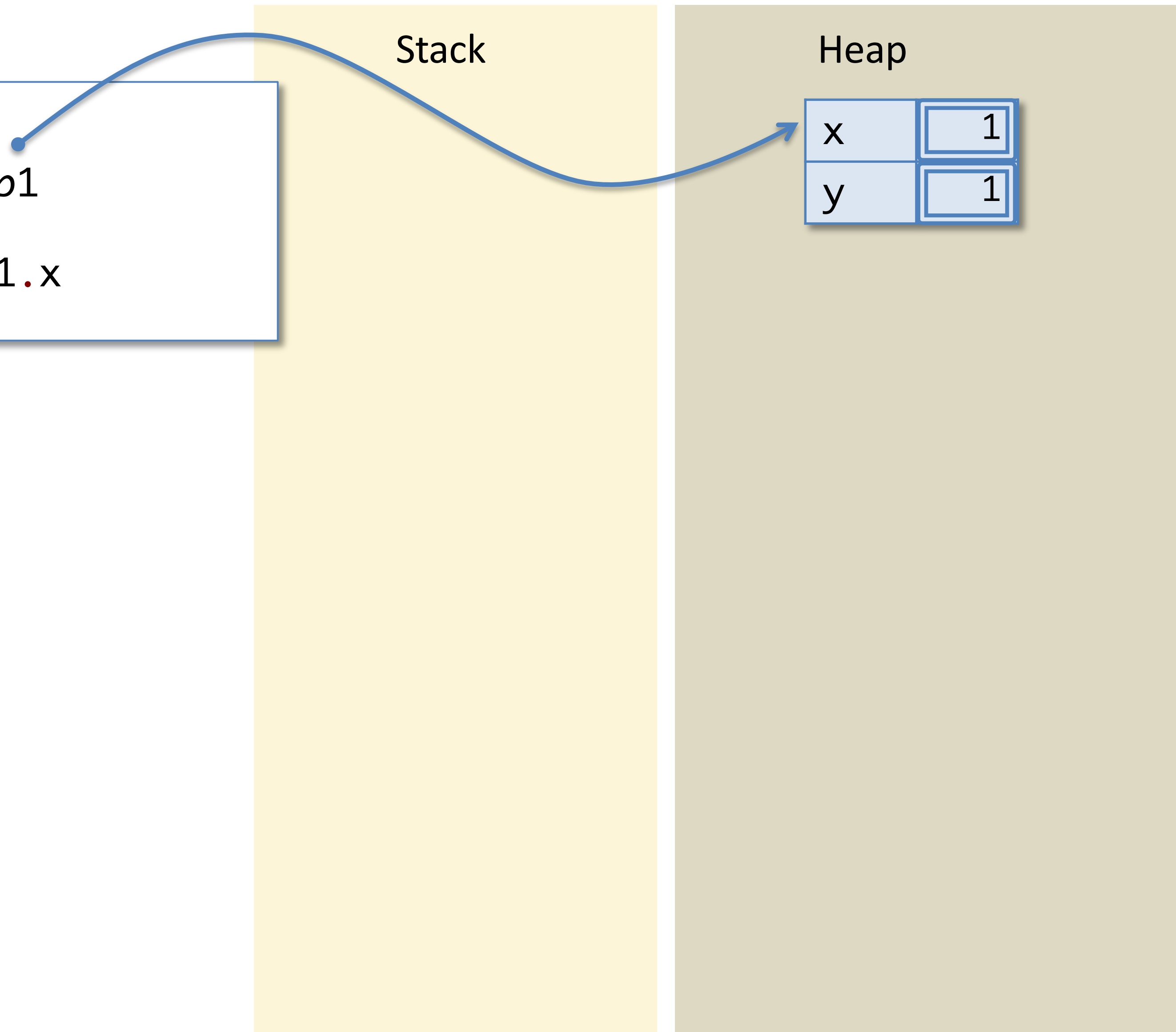
Stack

Heap

Allocate a Record

Workspace

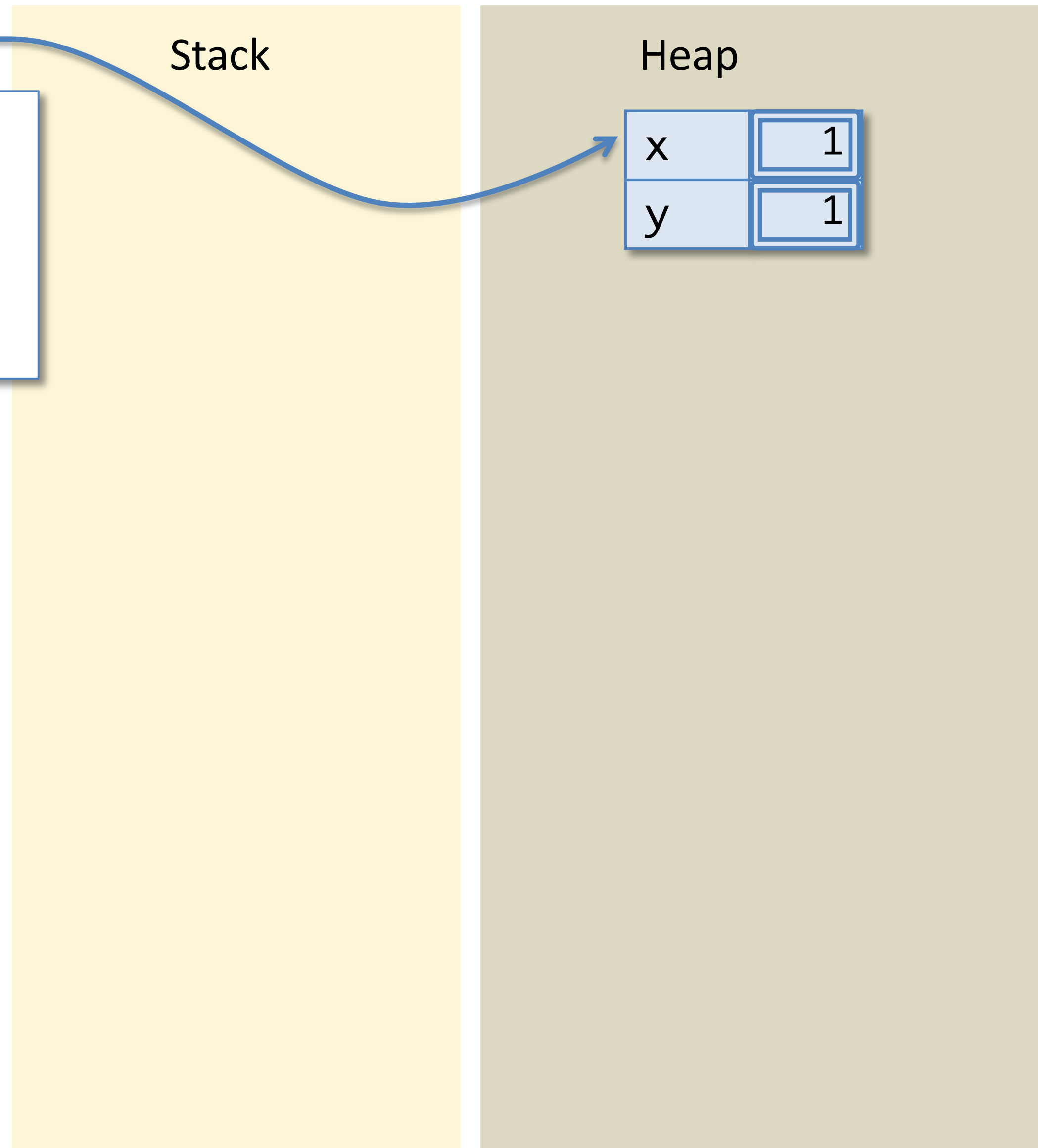
```
let p1 : point =
let p2 : point = p1
let ans : int =
    p2.x <- 17; p1.x
```



Let Expression

Workspace

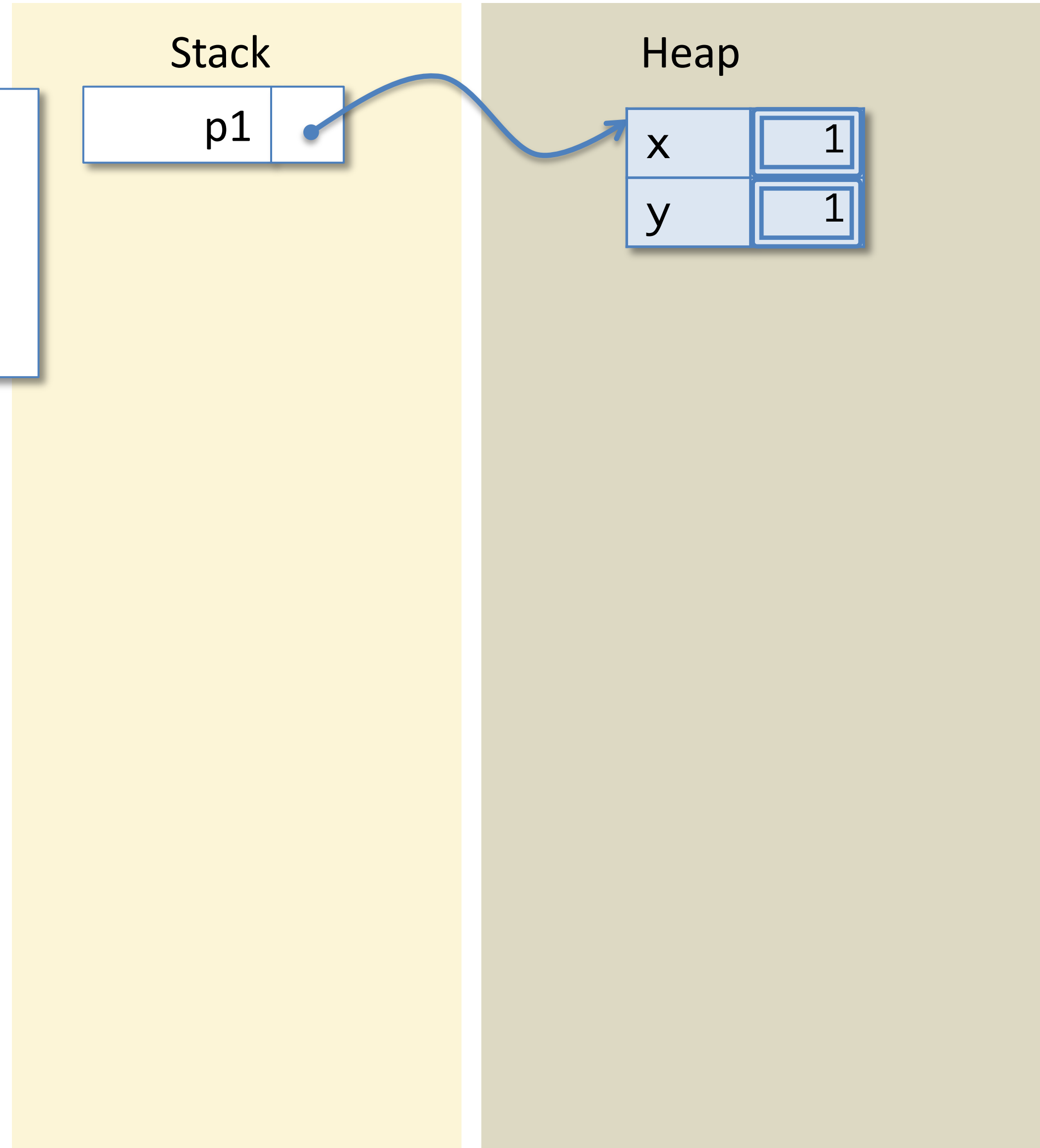
```
let p1 : point =   .
let p2 : point = p1
let ans : int =
    p2.x <- 17; p1.x
```



Push p1

Workspace

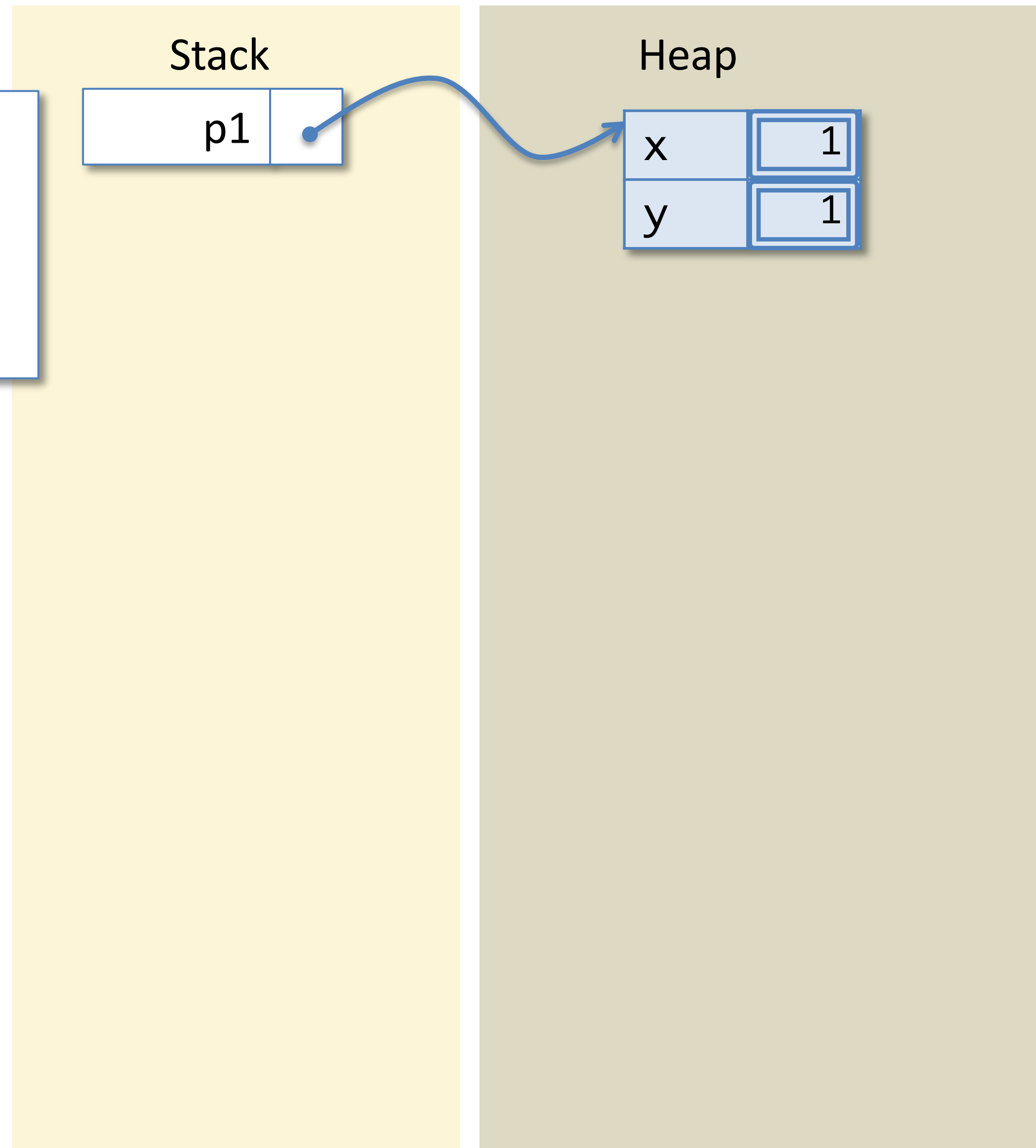
```
let p2 : point = p1
let ans : int =
    p2.x <- 17; p1.x
```



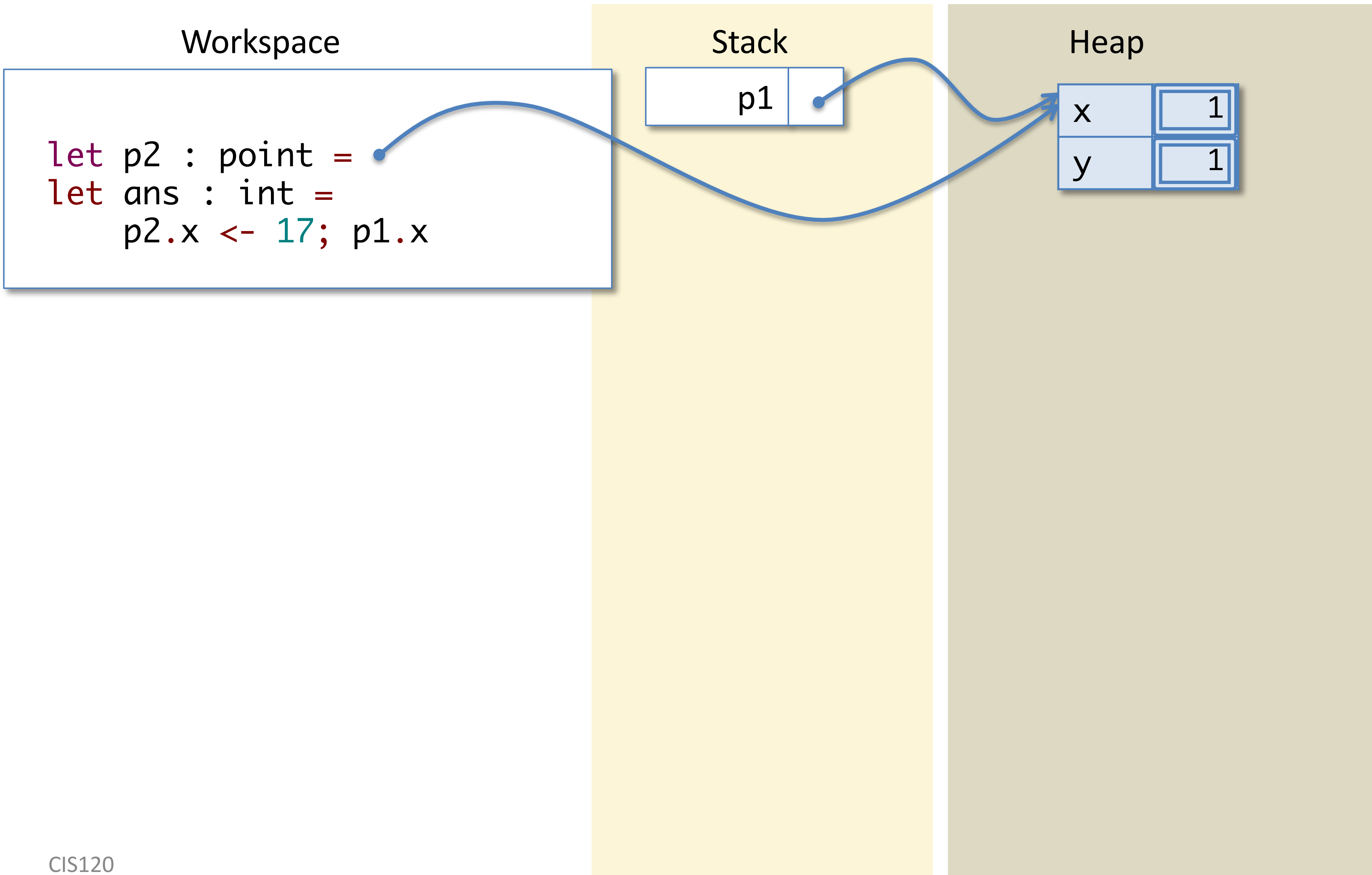
Look Up ‘p1’

Workspace

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



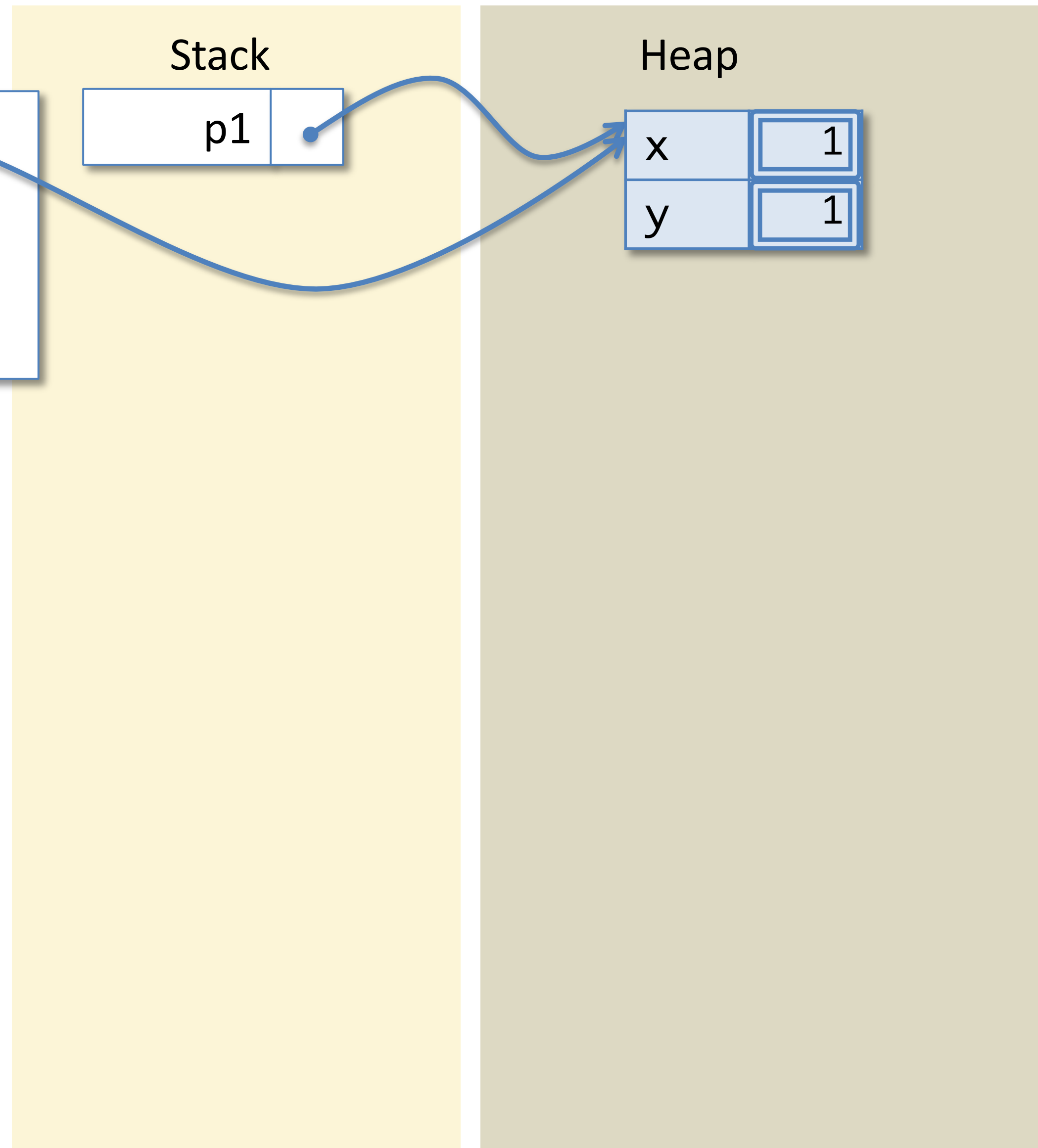
Look Up ‘p1’


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

Let Expression

Workspace

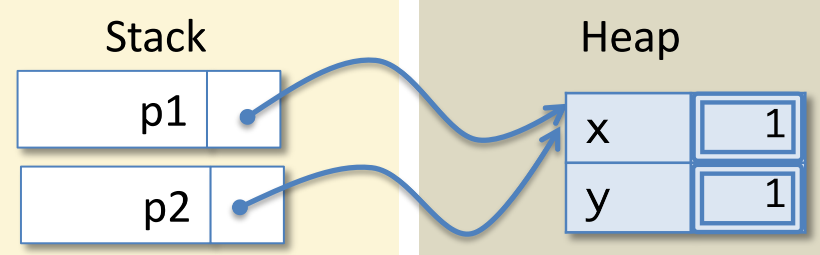
```
let p2 : point = ● .
let ans : int =
    p2.x <- 17; p1.x
```



Push p2

Workspace

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

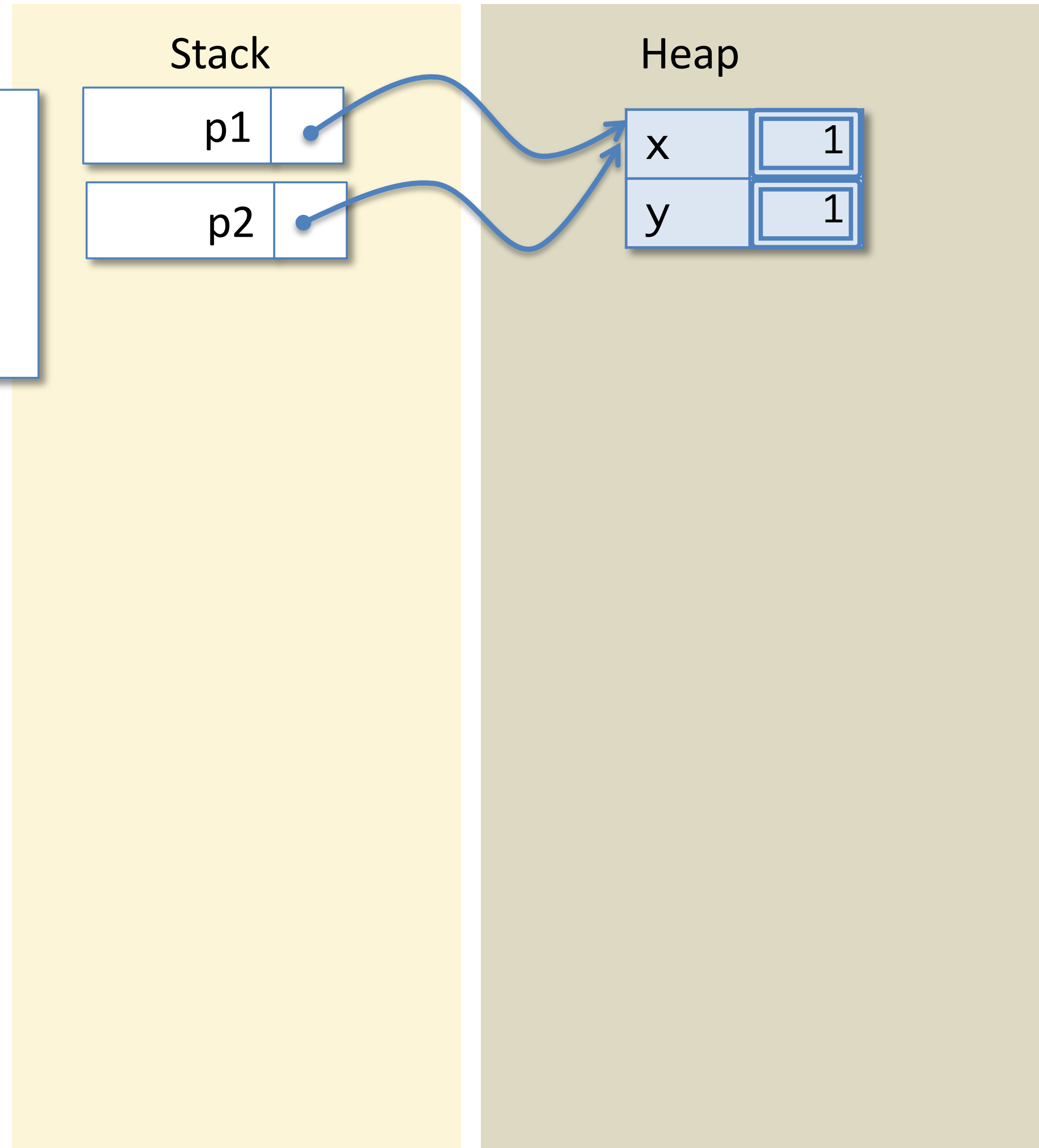


Note: p1 and p2 are references to the *same* heap record.
They are *aliases* – two different names for the *same thing*.

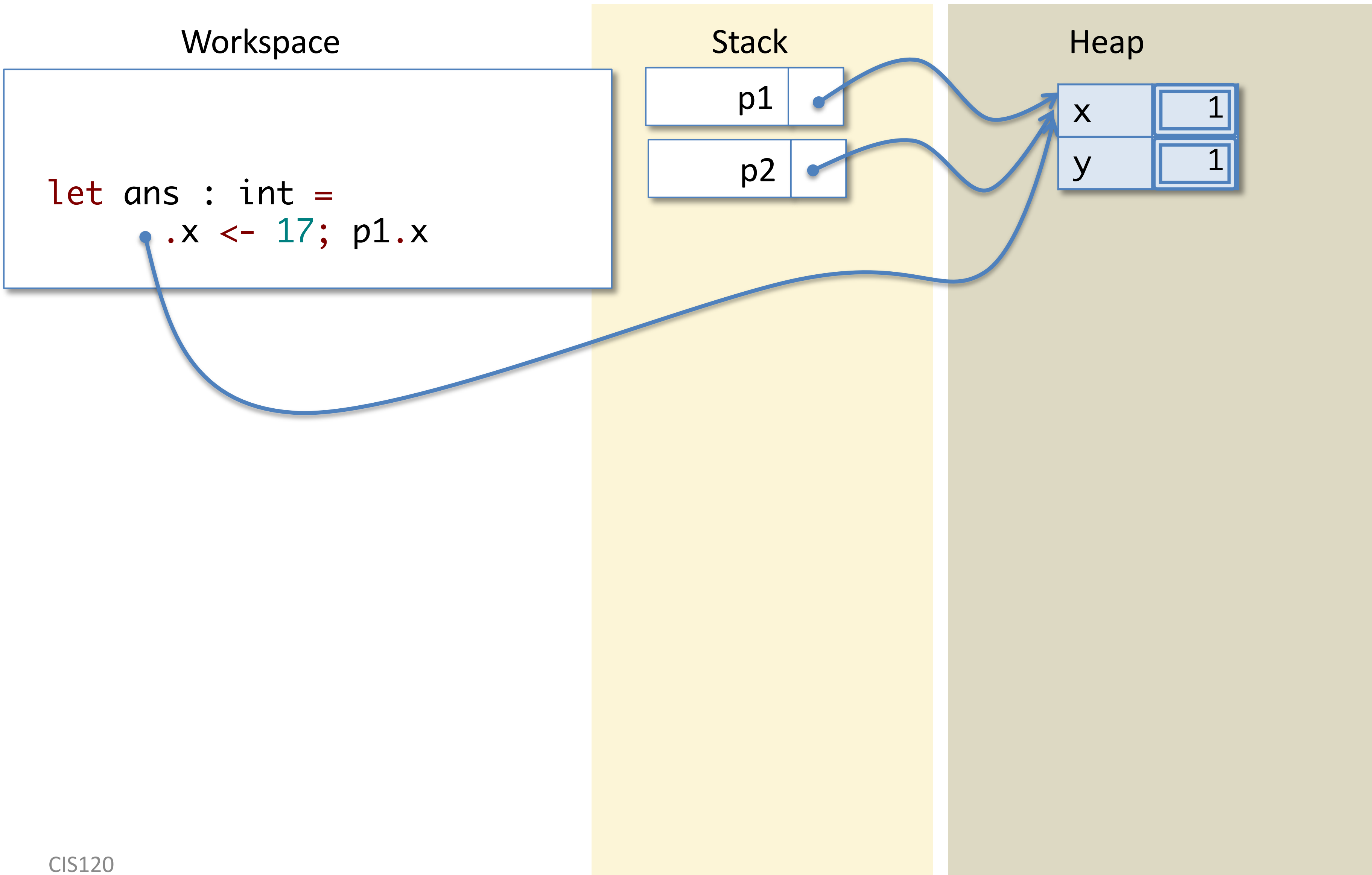
Look Up ‘p2’

Workspace

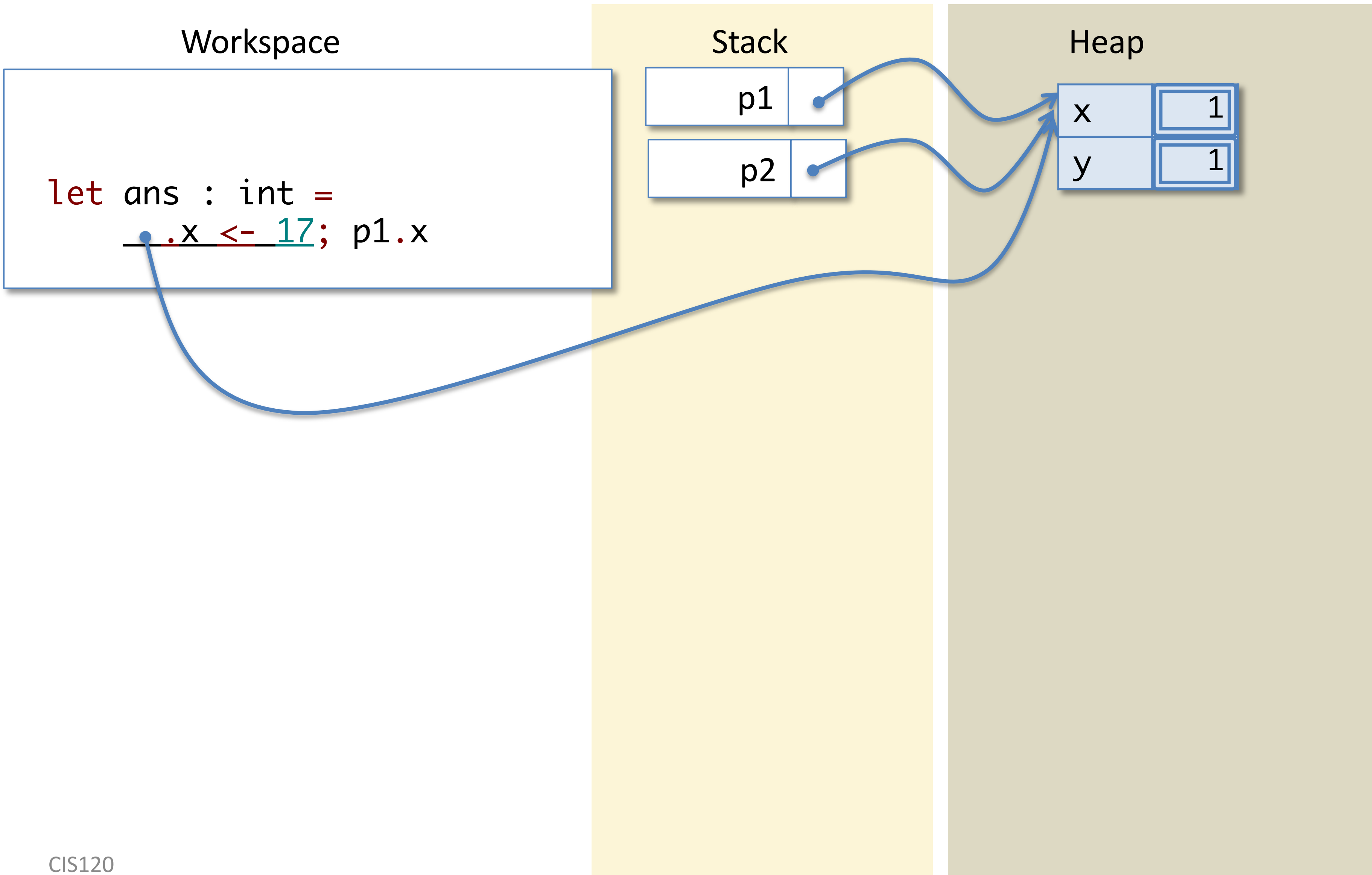
```
let ans : int =
    p2.x <- 17; p1.x
```



Look Up ‘p2’



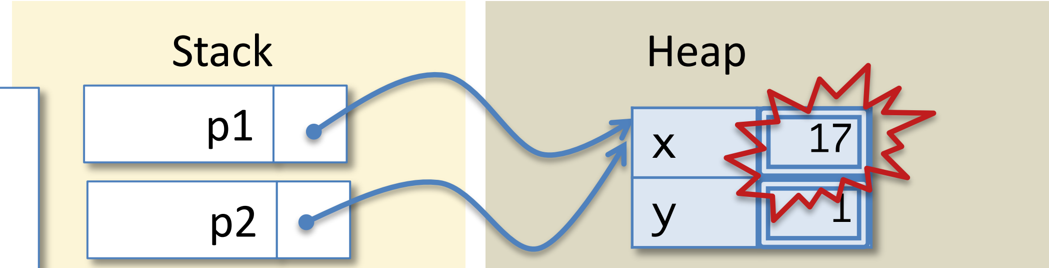
Assign to x field



Assign to x field

Workspace

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

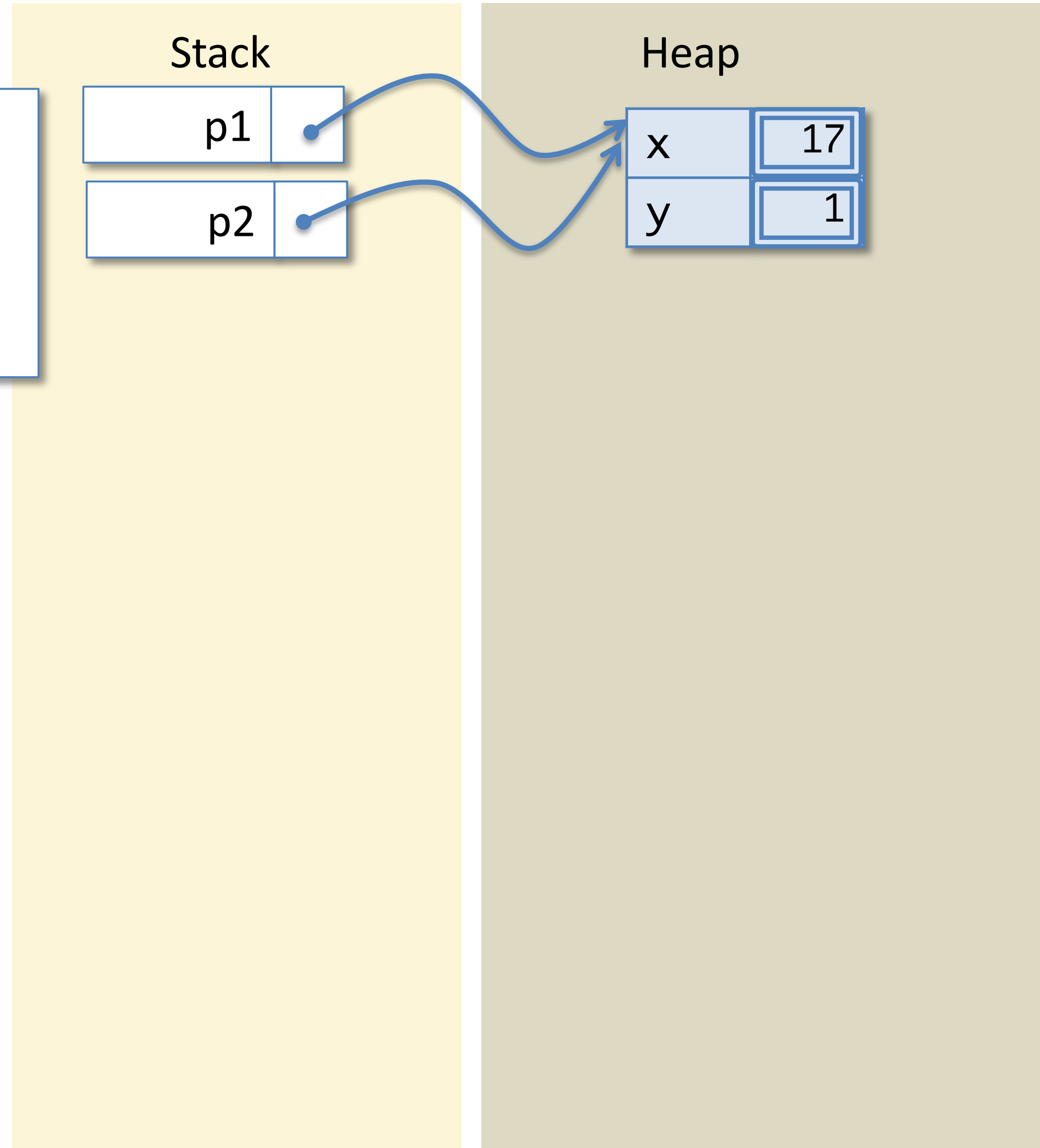


This is the step in which the ‘imperative’ update occurs. The mutable field x has been modified in place to contain the value 17.

Sequence ‘;’ Discards Unit

Workspace

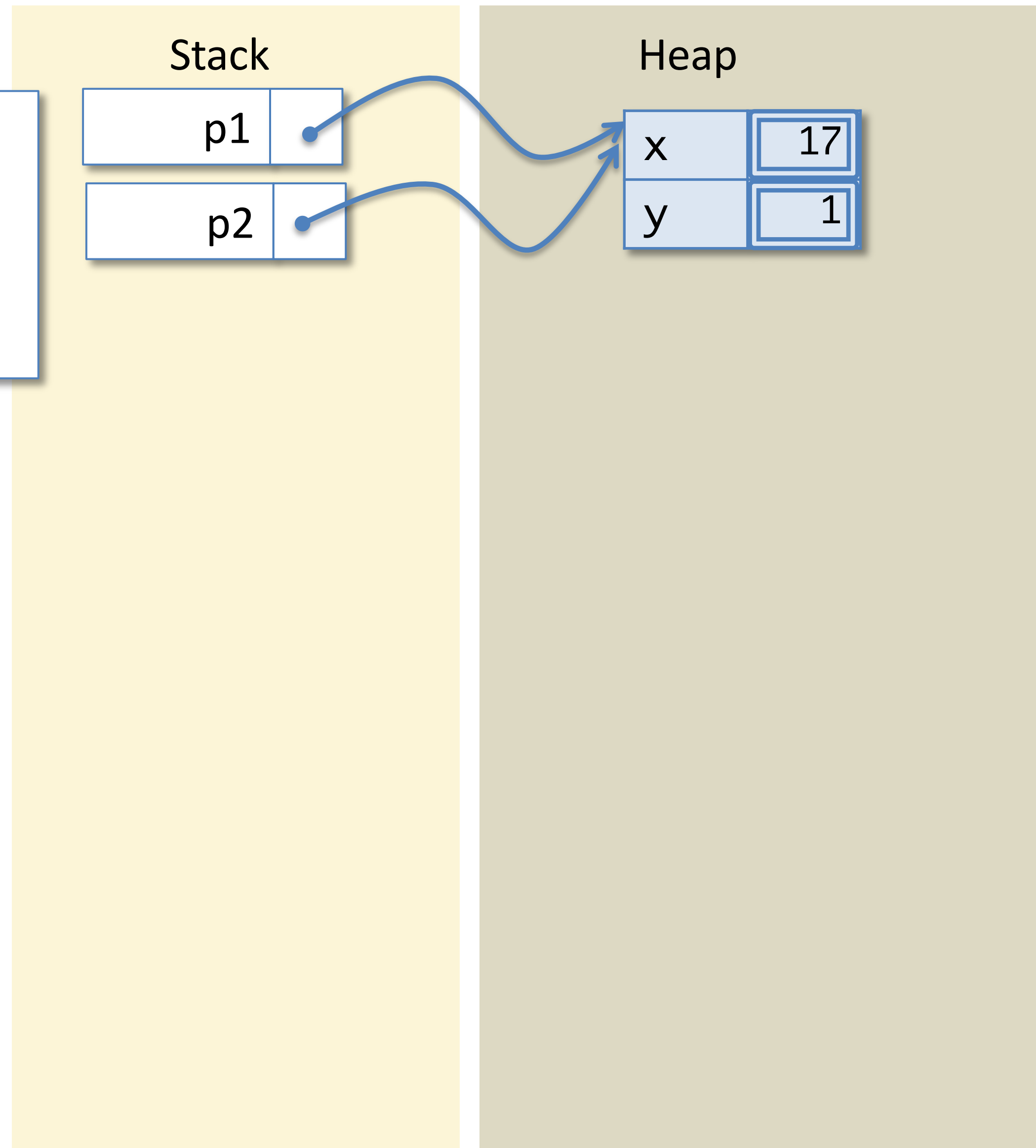
```
let ans : int =
    (); p1.x
```



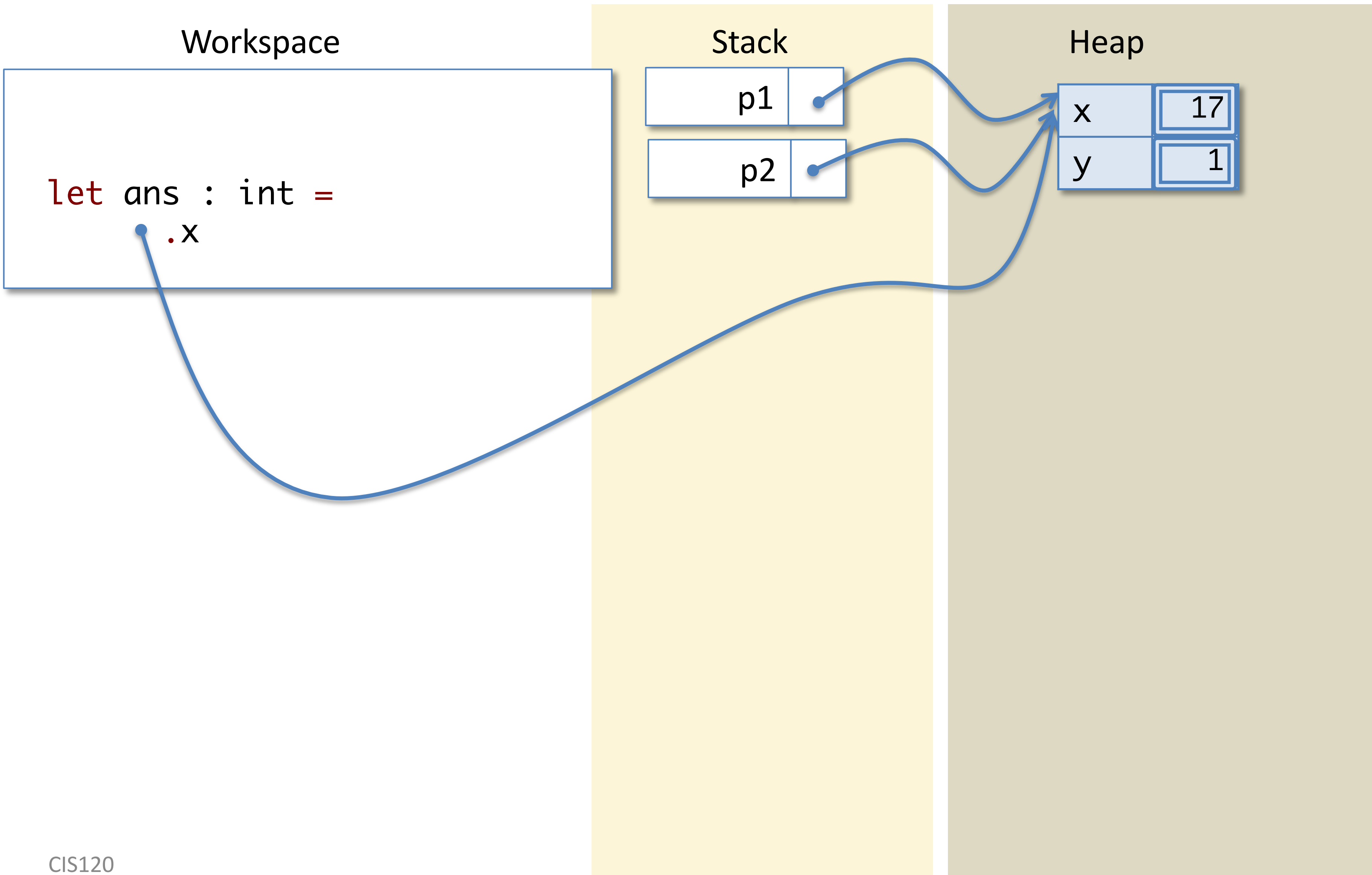
Look Up 'p1'

Workspace

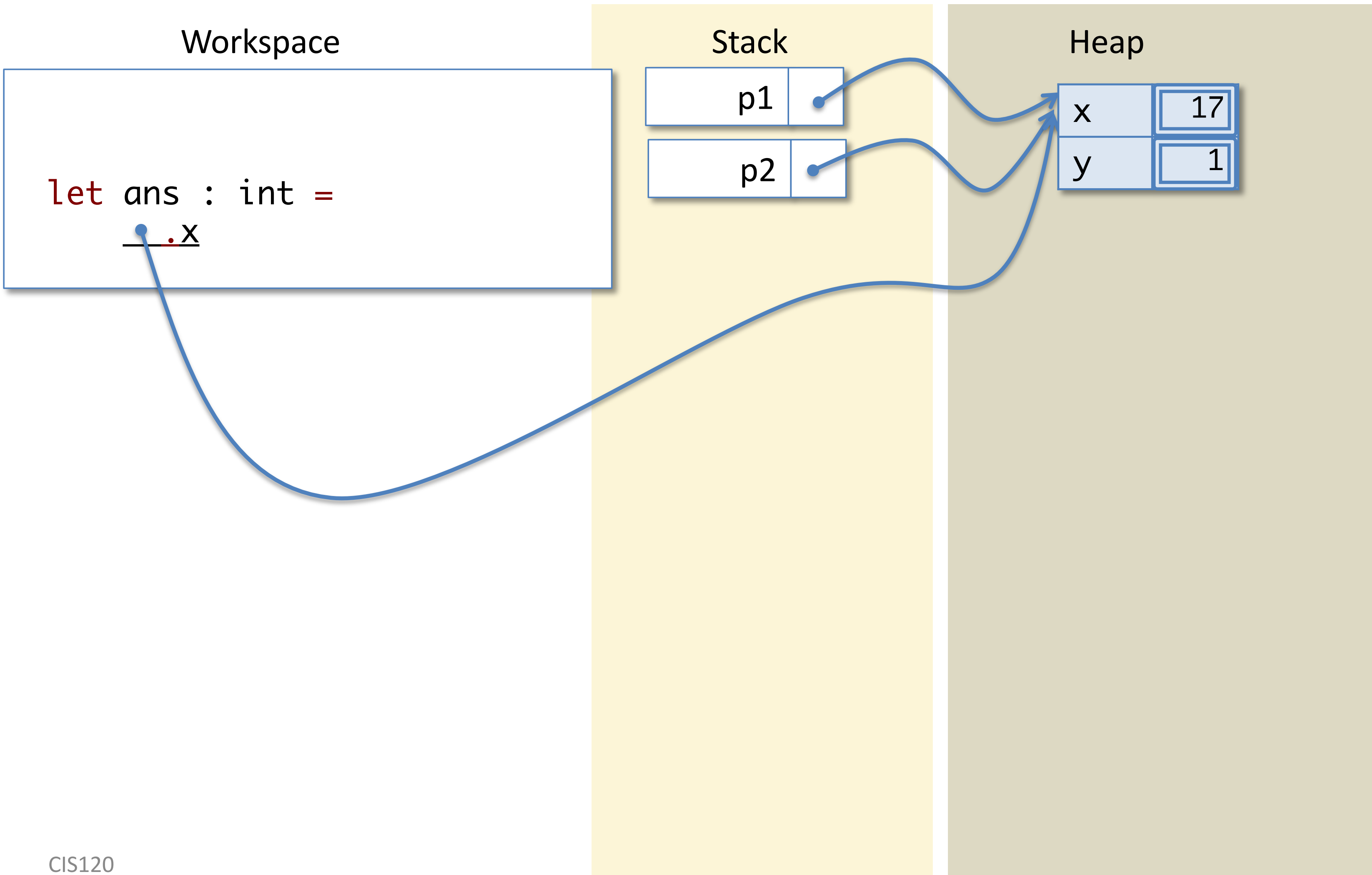
```
let ans : int =
    p1.x
```



Look Up ‘p1’



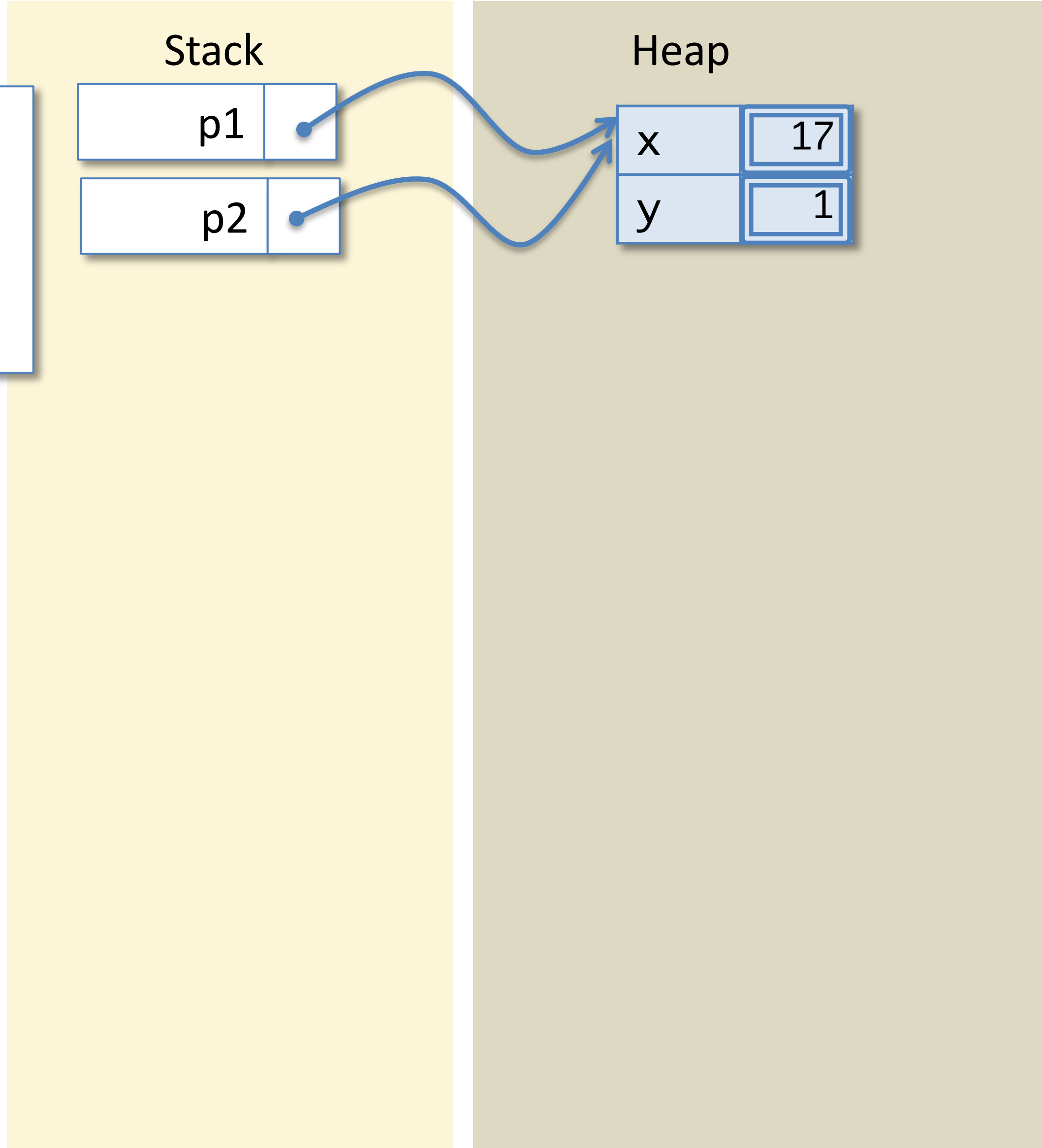
Project the 'x' field



Project the 'x' field

Workspace

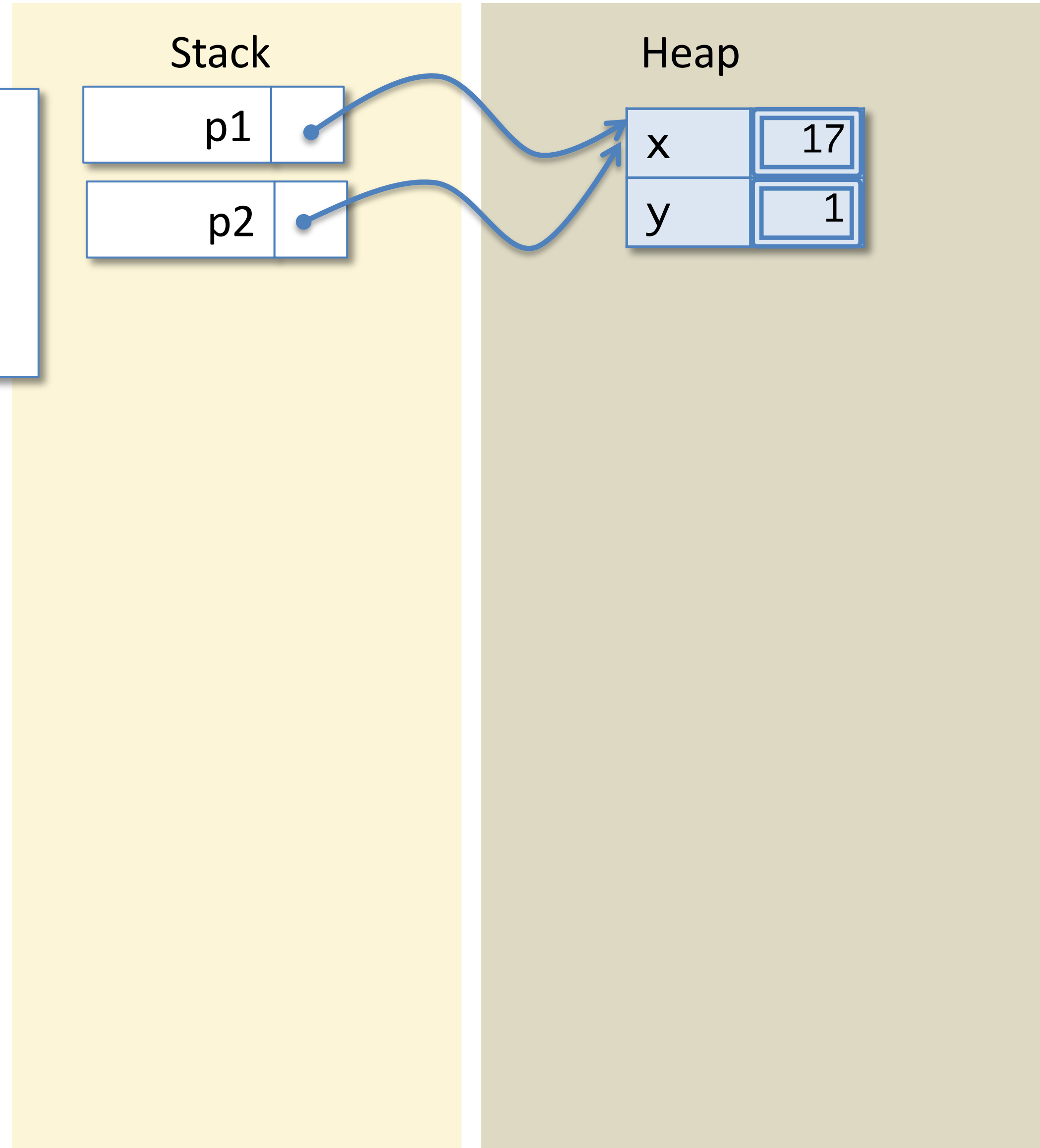
```
let ans : int =
    17
```



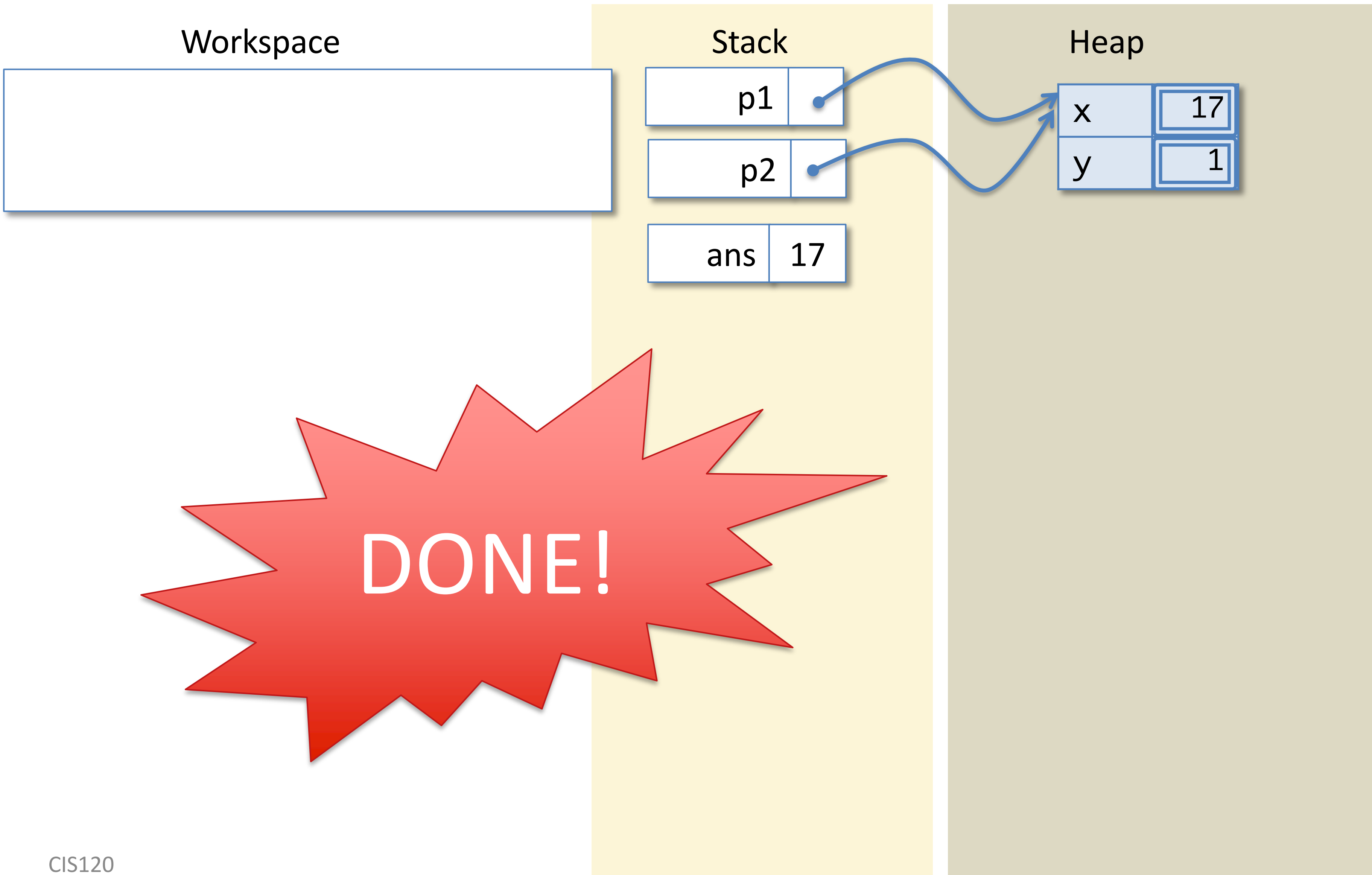
Let Expression

Workspace

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



Push ans



What answer does the following function produce when called?

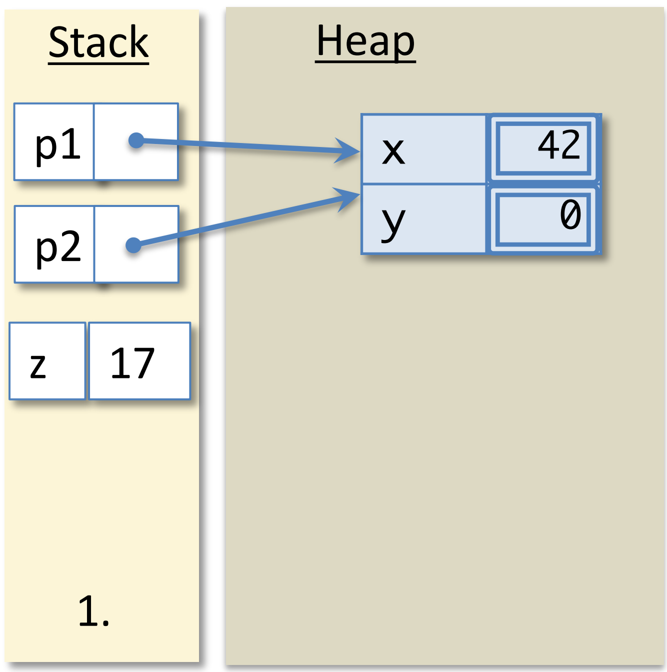
```
let f (p1:point) (p2:point) : int =
  p1.x <- 17;
  let z = p1.x in
  p2.x <- 42;
  z
```

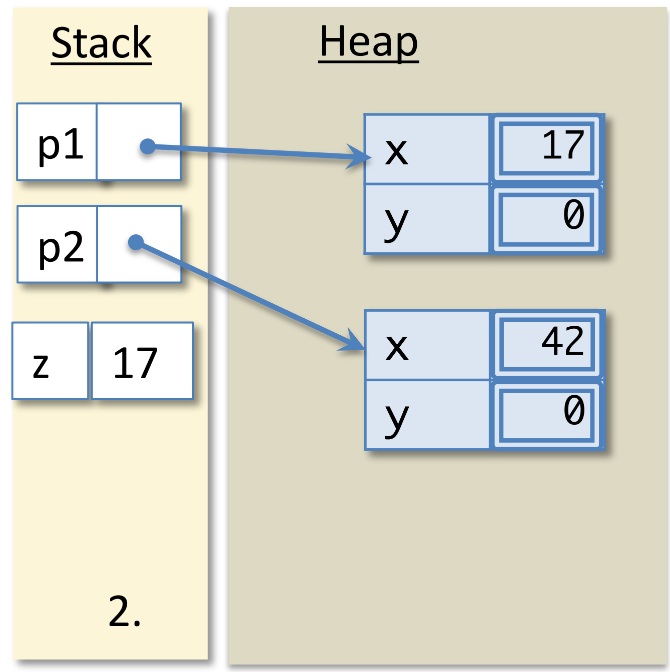
1. 17
2. 42
3. sometimes 17 and sometimes 42
4. f is ill typed

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
```





Answer: 1