Programming Languages and Techniques (CIS120)

Lecture 14

February 15th, 2016

Sequencing, Mutable State

Chapters 12, 13, 14

Announcements

Midterm 1

- Tomorrow evening, 6:15 PM
  - Last names A – Schwartz MEYH B1
  - Last names Shah – Z DRLB A8
- Covers lecture material through last Wednesday
  - Pure, value-oriented programming up to option Types
- Review materials (old exams) on course website
- Should have received email confirmation about make-up exam

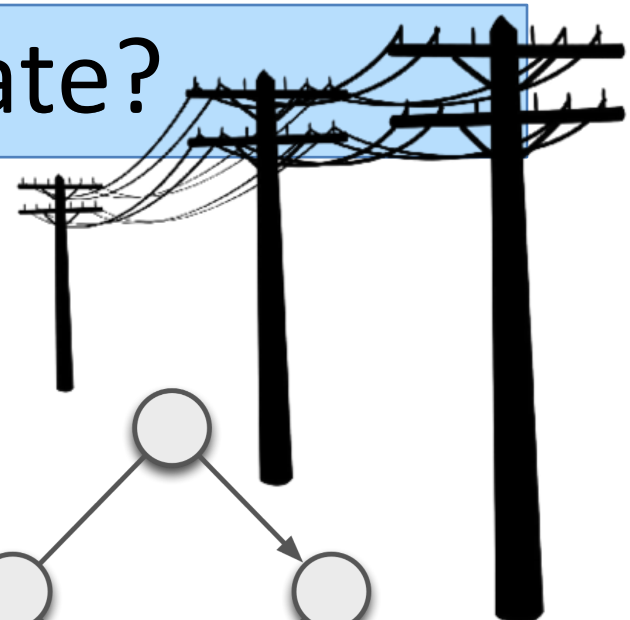
- My office hours: TODAY 3:30 – 5:00

Mutable state & effectful programming

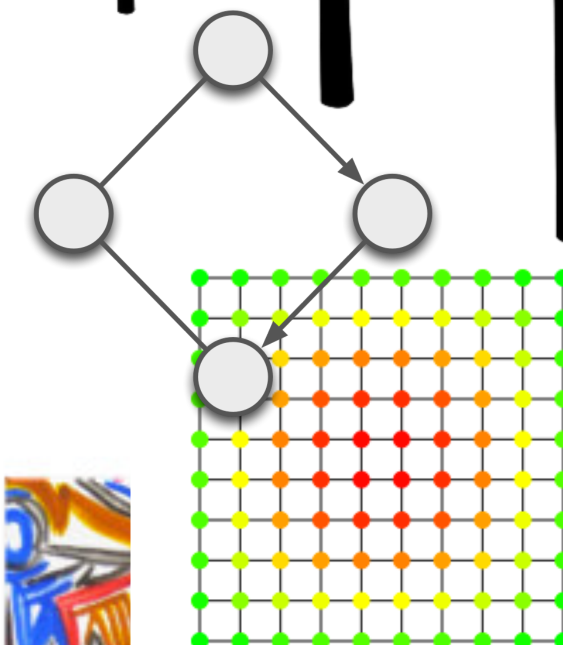
Why Pure Functional Programming?

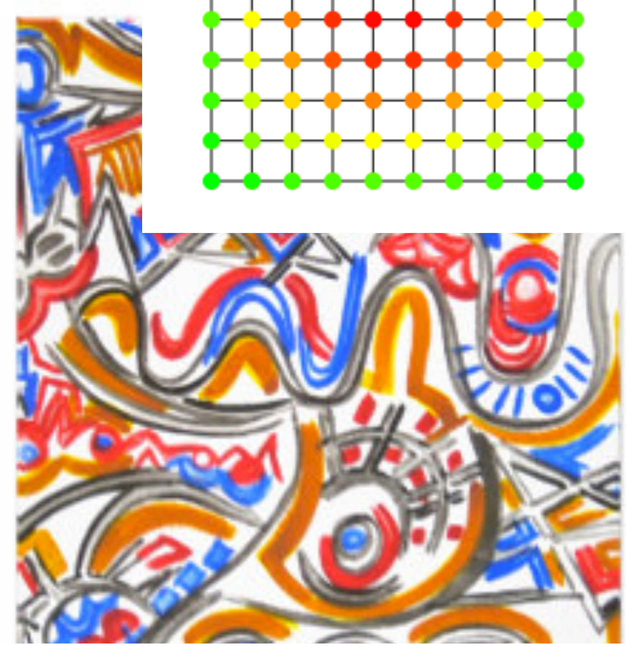
- Simplicity
  - small language: arithmetic, local variables, recursive functions, datatypes, pattern matching, generic types/functions and modules
  - simple substitution model of computation
- Persistent data structures
  - Nothing changes; retains all intermediate results
  - Good for version control, fault tolerance, etc.
- Typecheckers 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

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
  - some data structures have imperative versions 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, etc.





A new view of imperative programming

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

- Null is contained in (almost) every type. Partial functions can return **null**.
- Code is a sequence of **statements** 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.)

- No null. Partiality must be made explicit with **options**.
- 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

Commands, Sequencing and Unit

What is the type of print_string?

Sequencing Commands and Expressions

We can *sequence* commands inside expressions using ';'

– unlike in C, Java, etc., ';' doesn't terminate a statement it *separates* a command from an expression

```
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

The distinction between commands & expressions is artificial.

- `print_string` is a function of type: `string -> unit`
- Commands are actually just expressions of type: `unit`

unit: the trivial type

- Similar to "void" in Java or C
- For functions that don't take any arguments

```
let f () : int = 3
let y : int =  f ()
```

```
val f : unit -> int
val y : int
```

- Also for functions that don't return anything, such as testing and printing functions a.k.a *commands*:

```
(* run_test : string -> (unit -> bool) -> unit *)
;; run_test “TestName” test

(* print_string : string -> unit *)
;; print_string “Hello, world!”
```

unit: the boring type

- *Actually, ( ) is a value just like any other value.*
- For functions that don't take any interesting arguments

```
let f () : int = 3
let y : int =  f ()
```

```
val f : unit -> int
val y : int
```

- Also for functions that don't return anything interesting, such as testing and printing functions a.k.a *commands*:

```
(* run_test : string -> (unit -> bool) -> unit *)
;; run_test “TestName” test

(* print_string : string -> unit *)
;; print_string “Hello, world!”
```

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*
  - 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
```

note the use of ';' here

do *not* use ';' 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`

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

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;}
```

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))
```

"in-place" update of p0.x

Defining new Commands

- Functions can assign to mutable record fields
- Note that the return type of ‘<-’ is unit

```
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
```

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

What answer does the following expression produce?

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

1. 17
2. 42
3. 0
4. runtime error

Answer: 17

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

What answer does the following expression produce?

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

1. 17
2. 42
3. 0
4. runtime error

Answer: 42

What answer does the following function produce when called?

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

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

Answer: sometimes 17 and sometimes 42

Issue with 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 ans = f p0 p0
```

Two identifiers are said to be *aliases* if they both name the *same* mutable record. Inside `f`, `p1`, and `p2` might be aliased, depending on which arguments are passed to `f`.

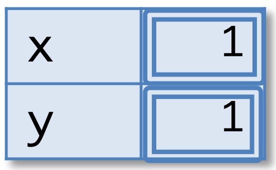
Mutable Records

- 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
```

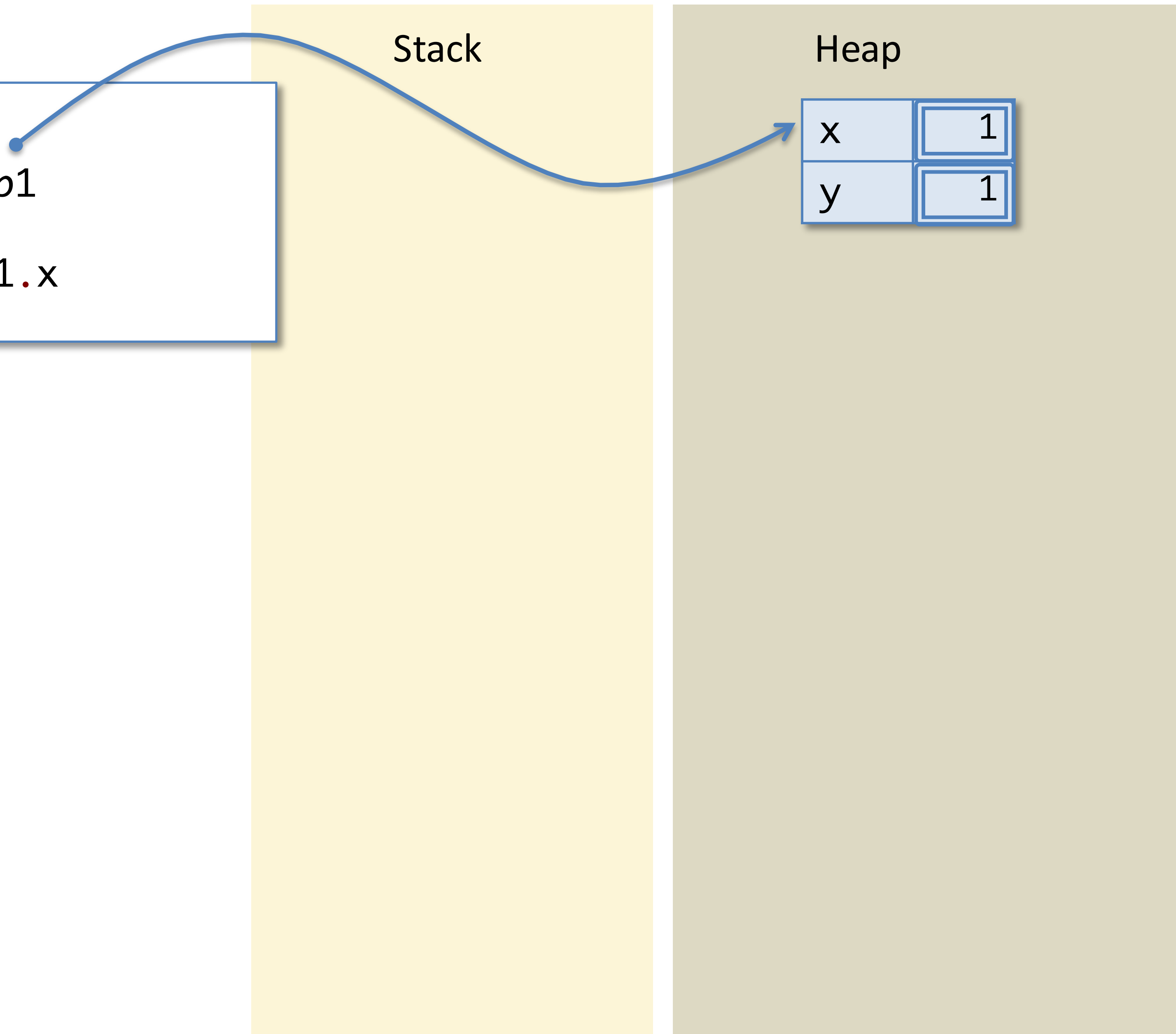
Stack

Heap

Allocate a Record

Workspace

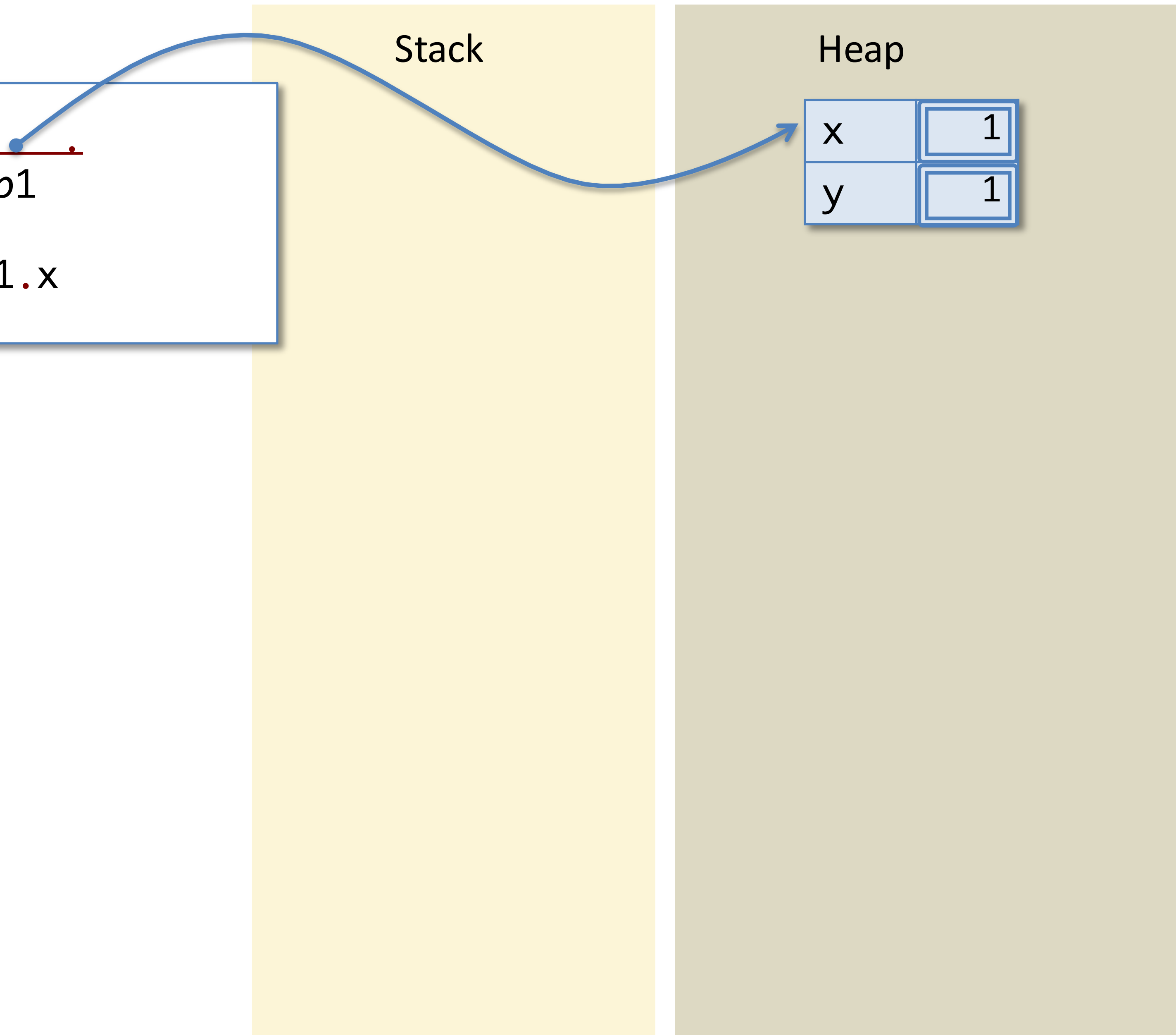
```
let p1 : point =
let p2 : point = p1
let ans : int =
    p2.x <- 17; p1.x
```



Let Expression

Workspace

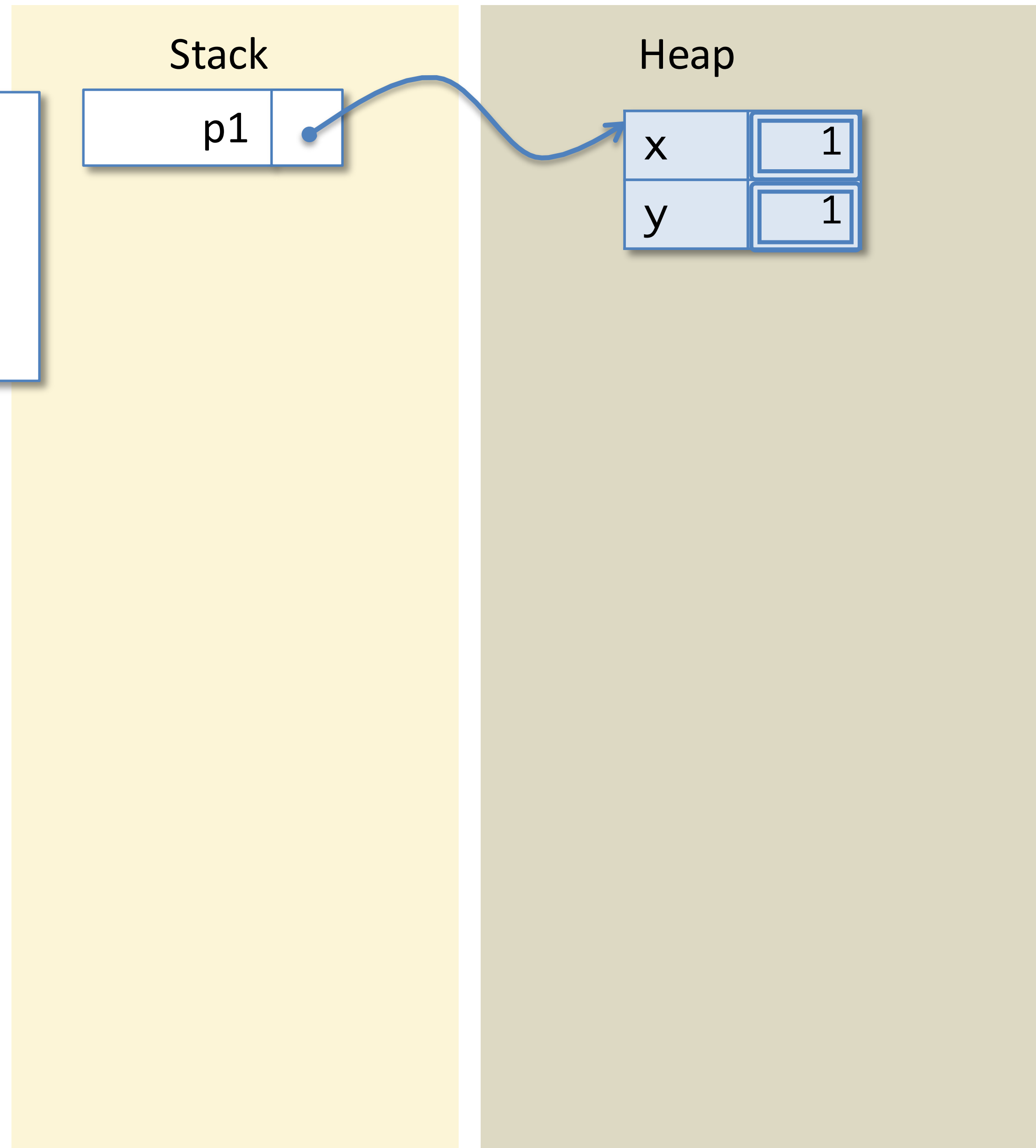
```
let p1 : point =  .
let p2 : point = p1
let ans : int =
    p2.x <- 17; p1.x
```



Push p1

Workspace

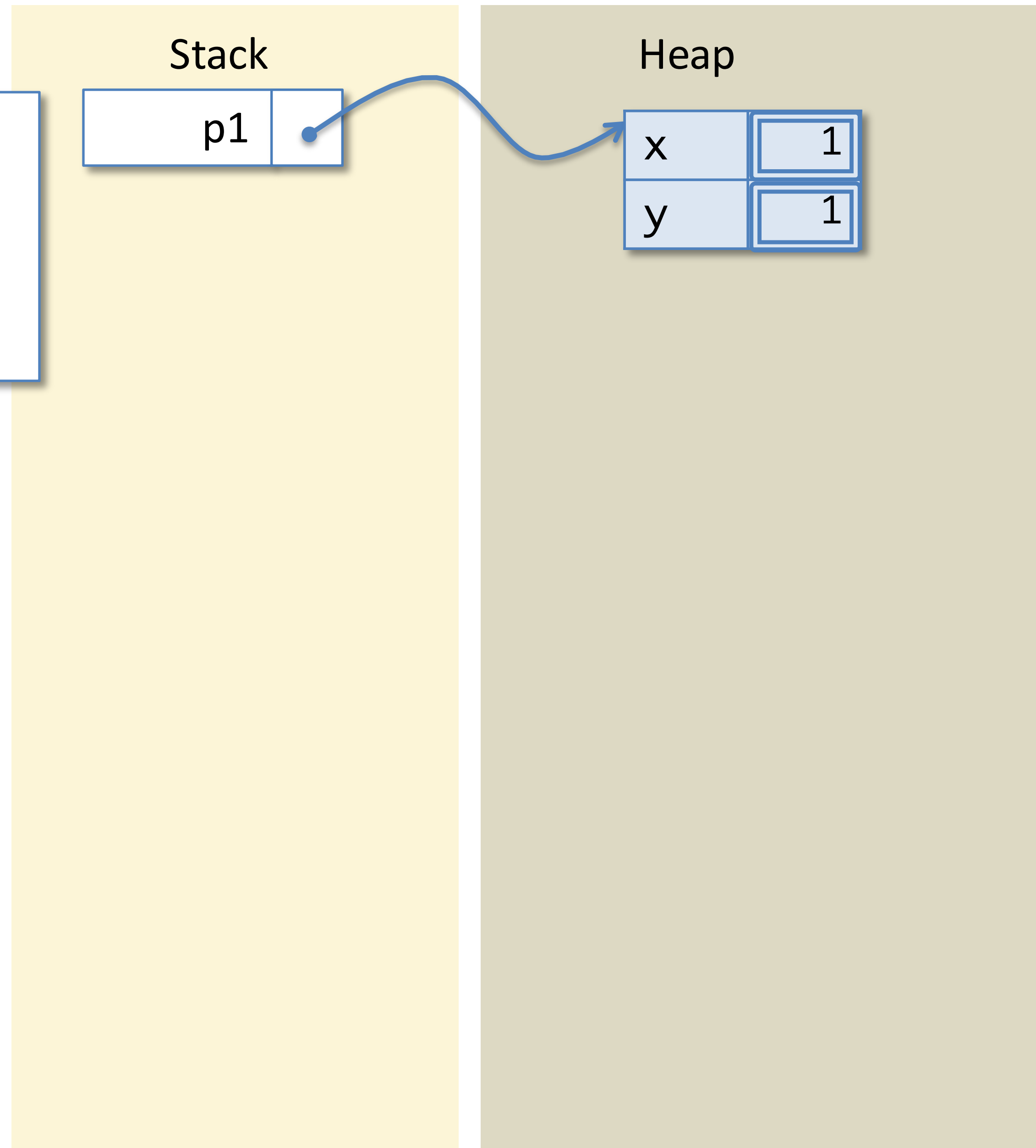
```
let p2 : point = p1
let ans : int =
    p2.x <- 17; p1.x
```



Look Up ‘p1’

Workspace

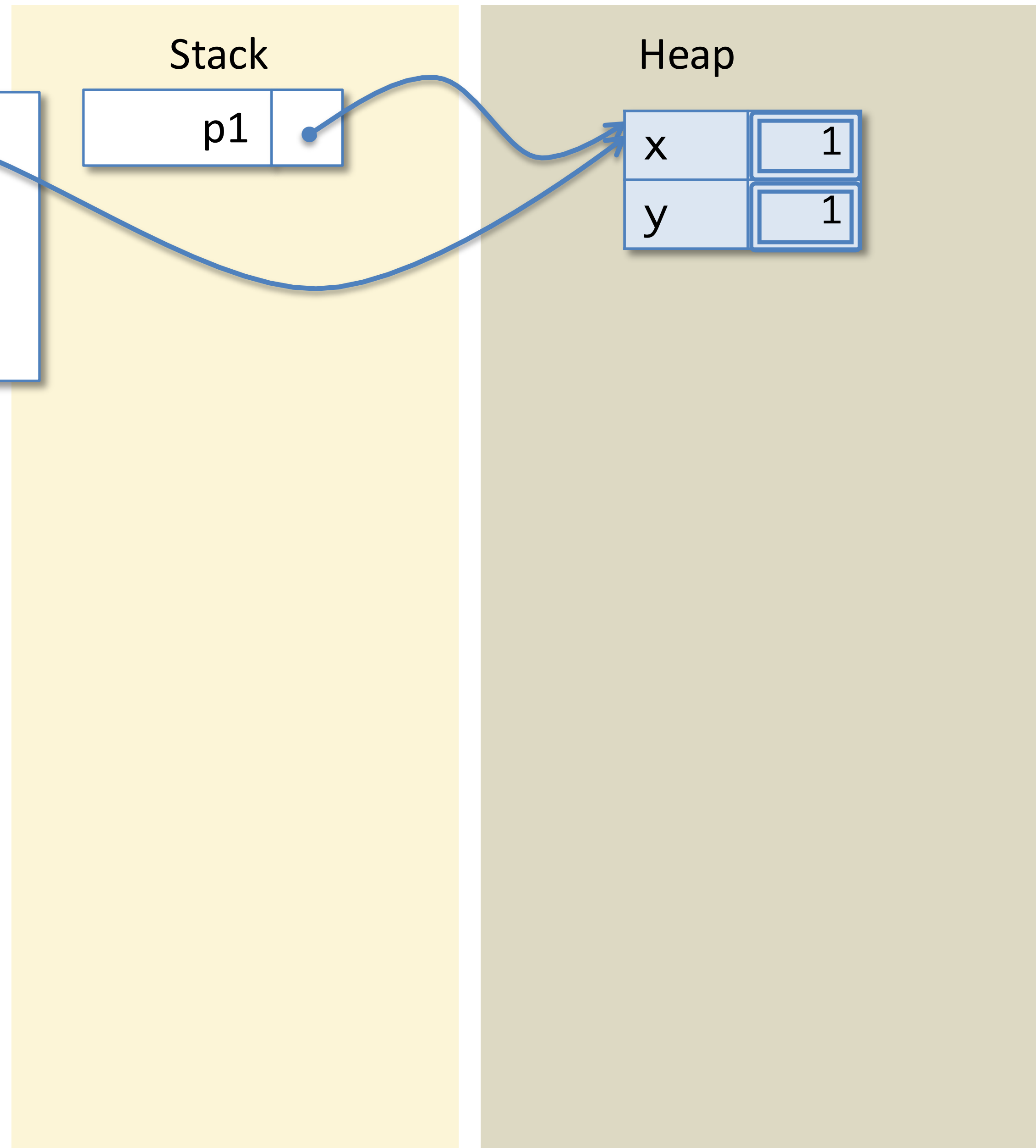
```
let p2 : point = p1
let ans : int =
    p2.x <- 17; p1.x
```



Look Up ‘p1’

Workspace

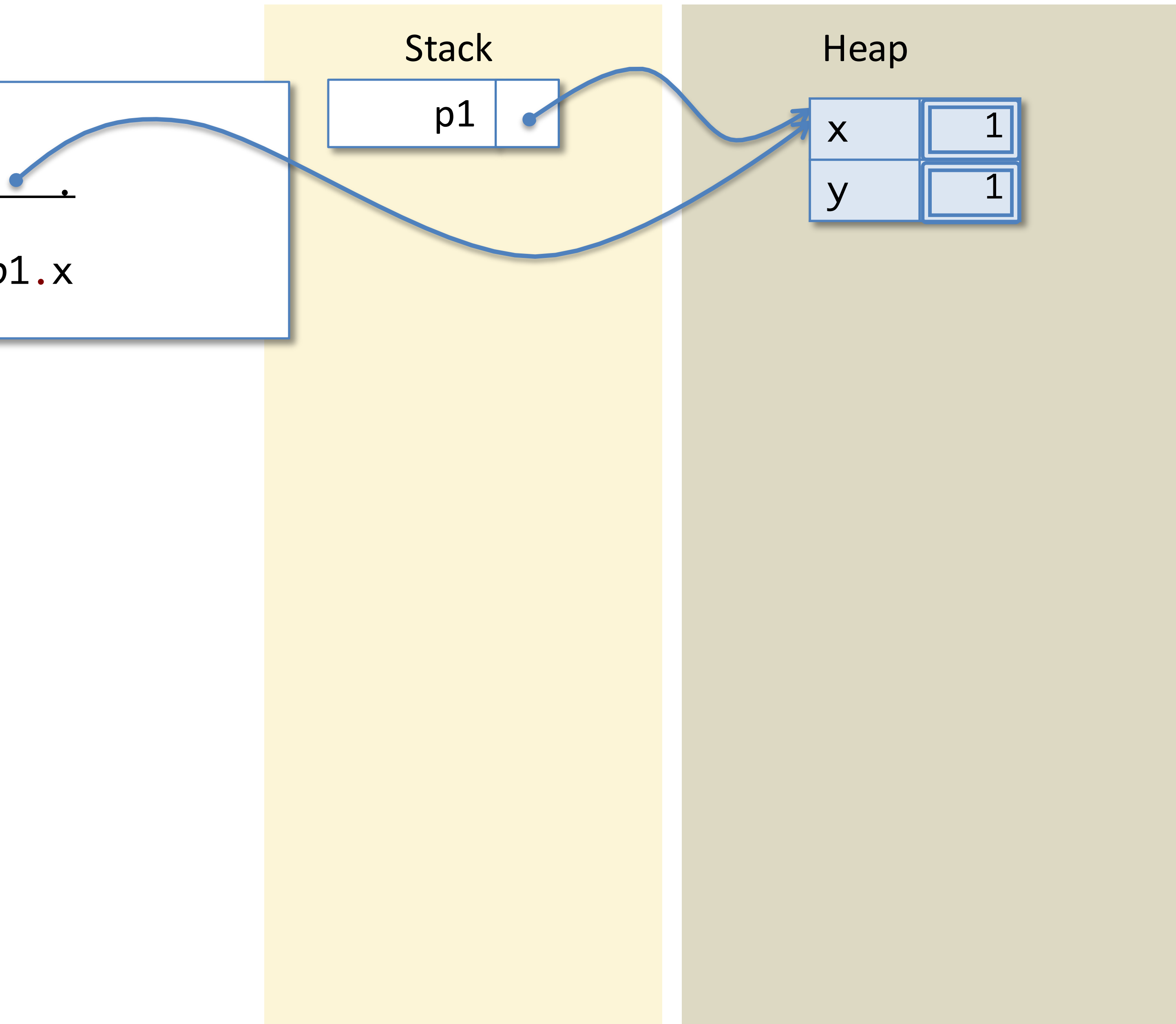
```
let p2 : point =
let ans : int =
    p2.x <- 17; p1.x
```



Let Expression

Workspace

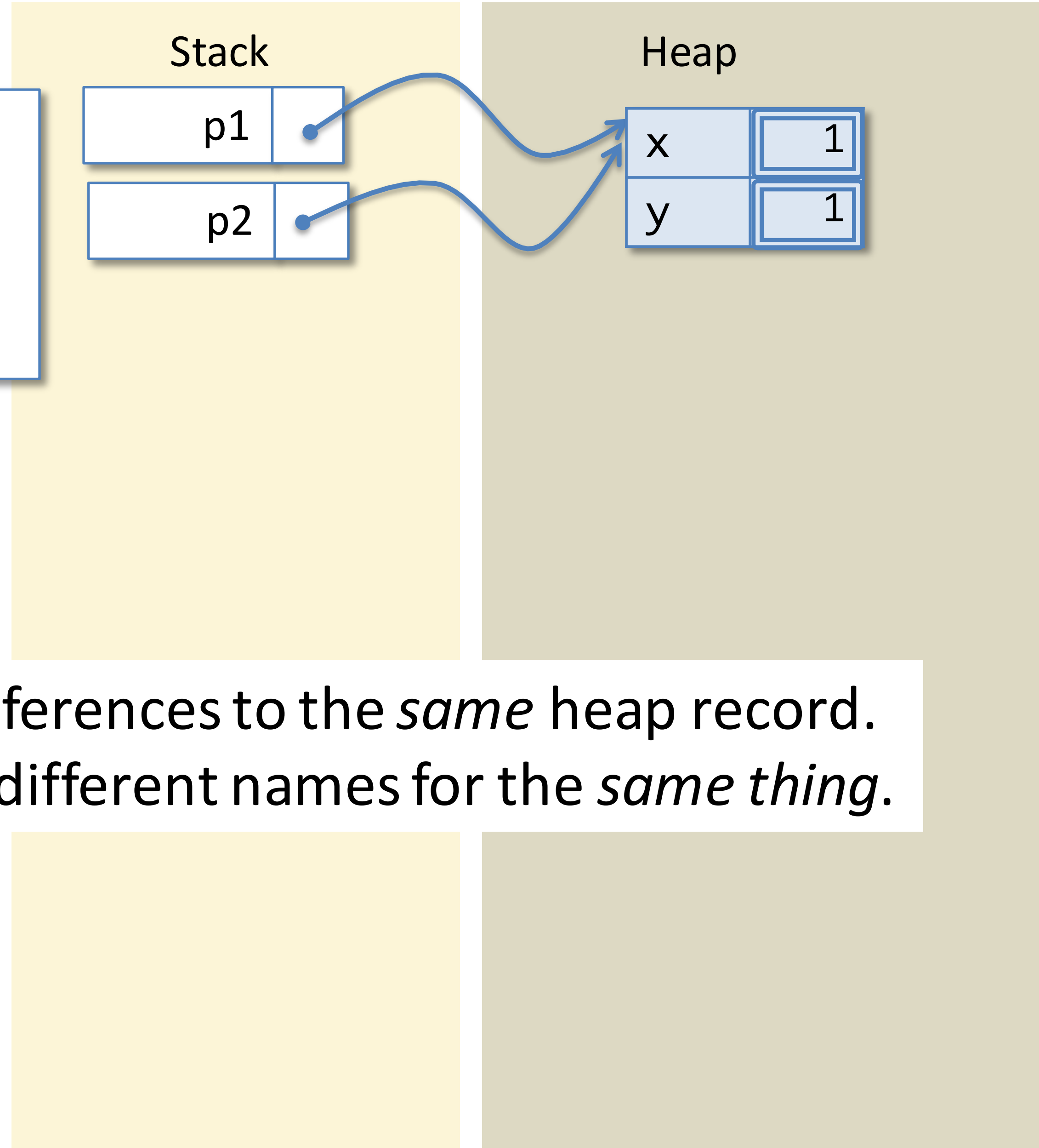
```
let p2 : point =  .
let ans : int =
    p2.x <- 17; p1.x
```



Push p2

Workspace

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

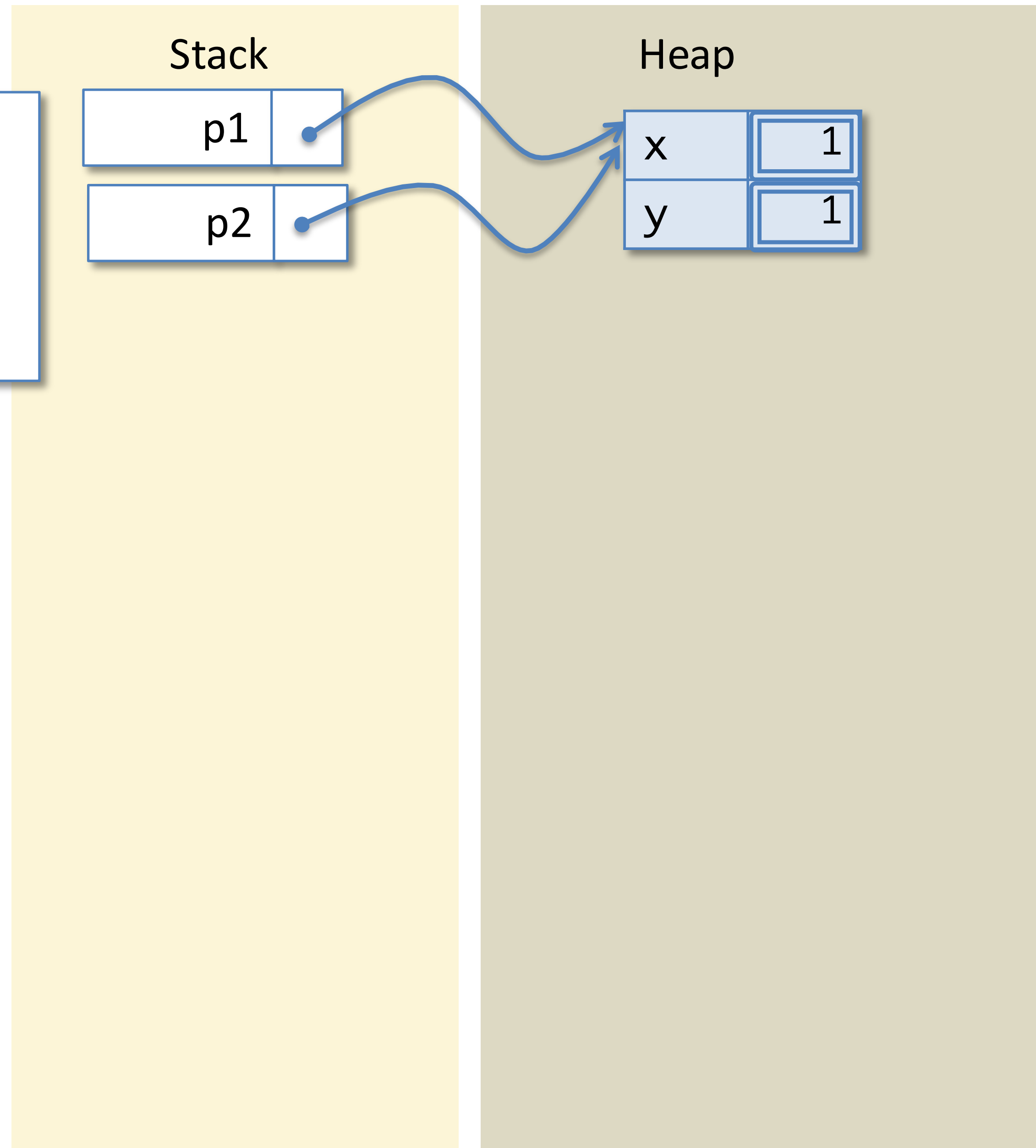


Note: p1 and p2 are references to the *same* heap record.
They are *aliases* – two different names for the *same thing*.

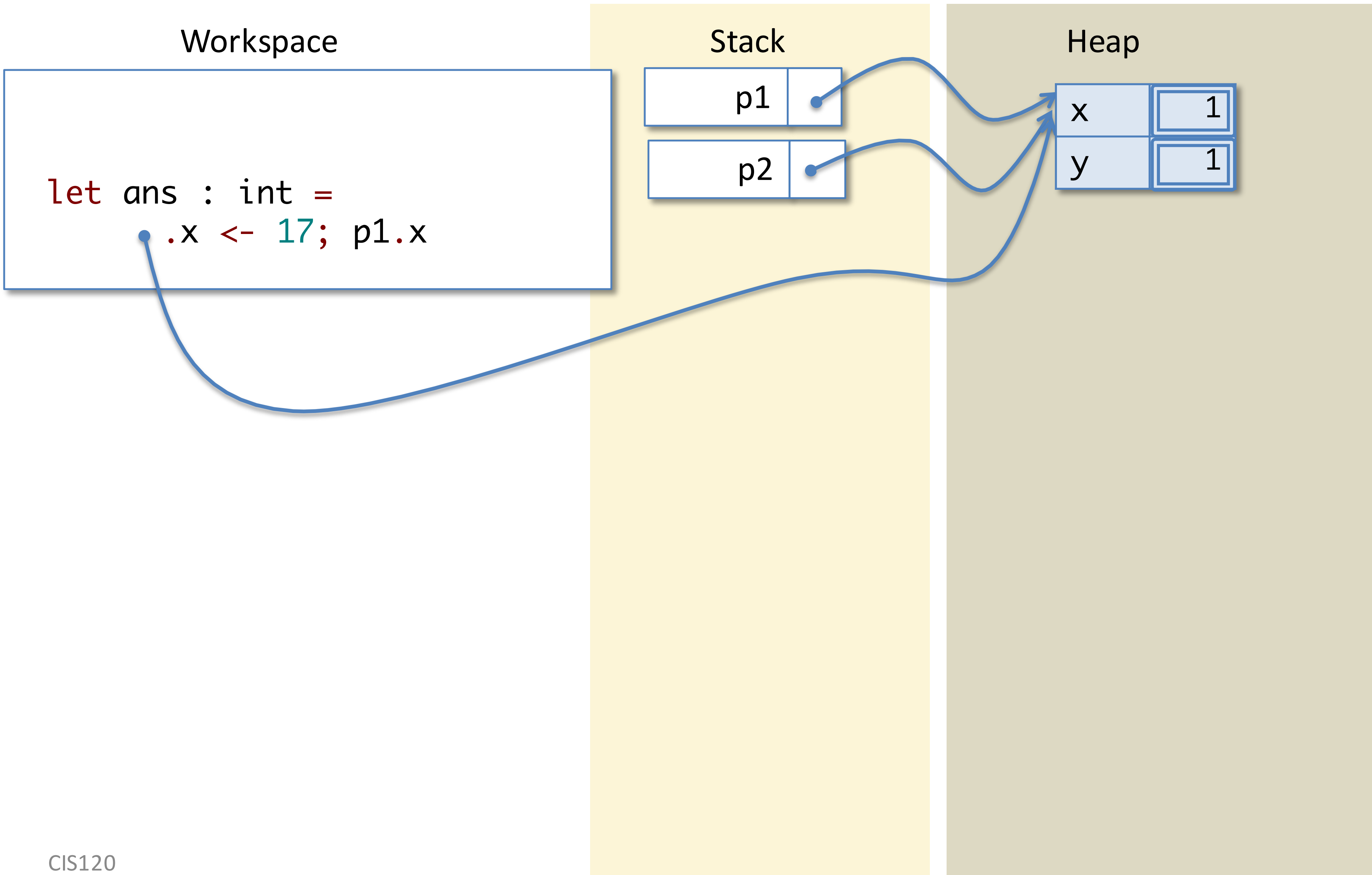
Look Up 'p2'

Workspace

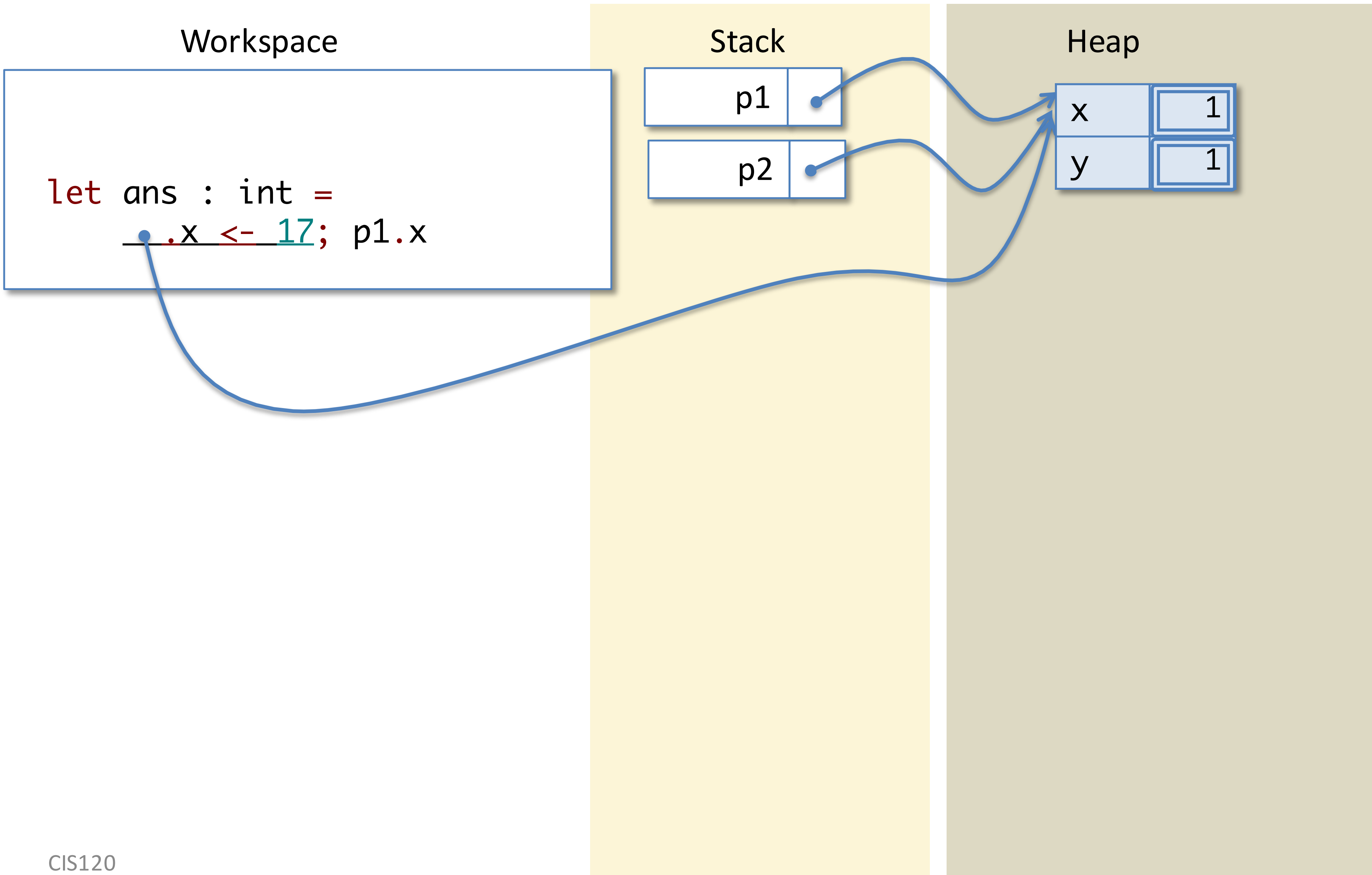
```
let ans : int =
    p2.x <- 17; p1.x
```



Look Up 'p2'



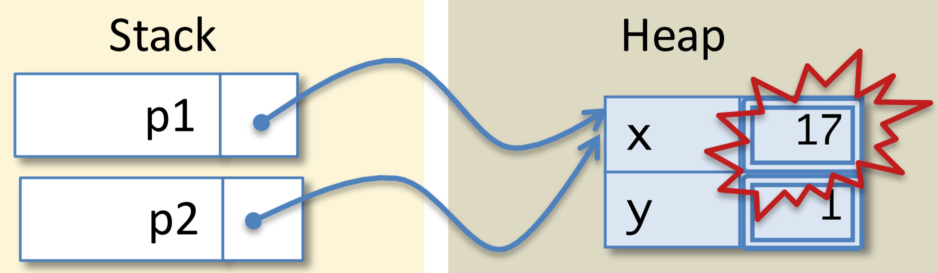
Assign to x field



Assign to x field

Workspace

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

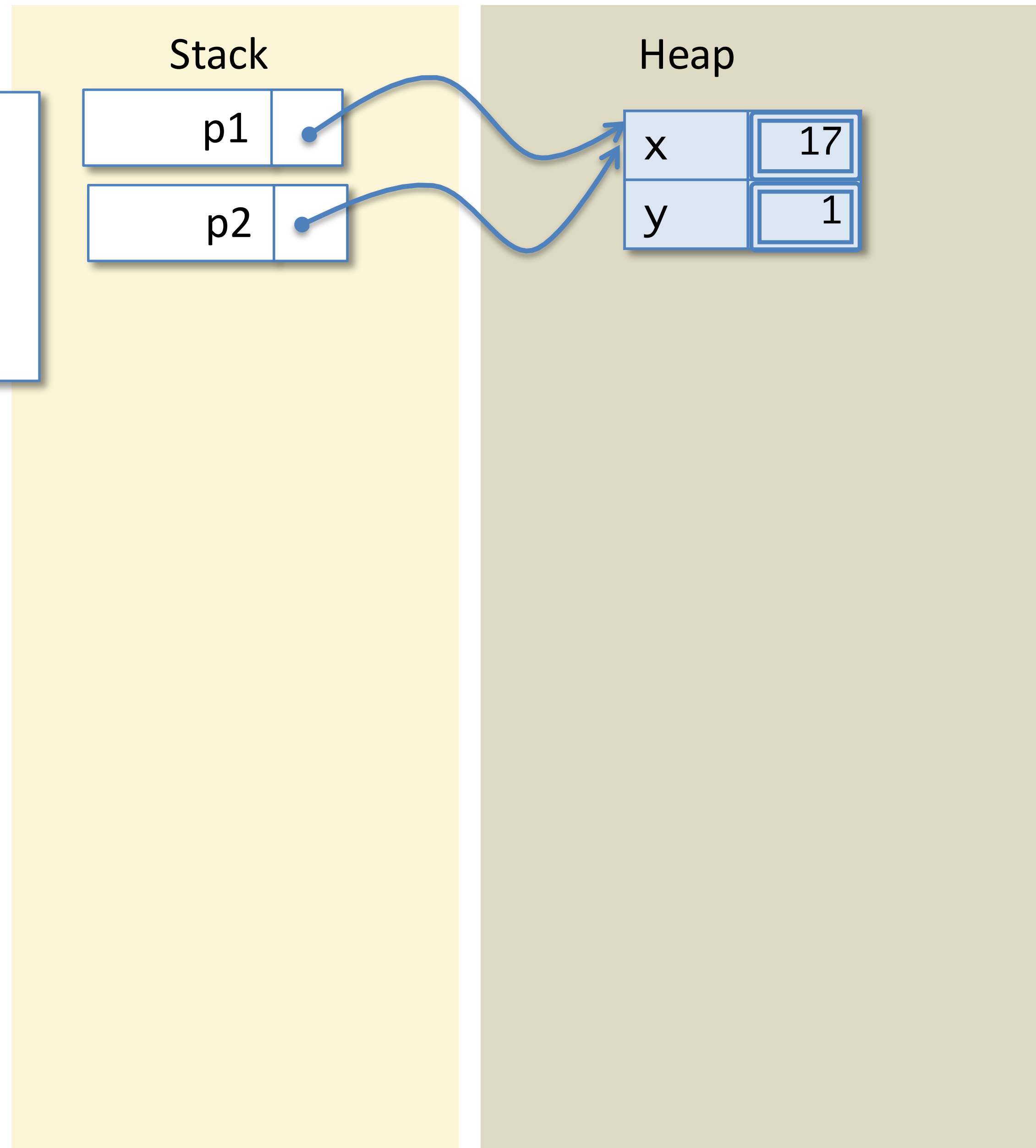


This is the step in which the 'imperative' update occurs. The mutable field x has been modified in place to contain the value 17.

Sequence ‘;’ Discards Unit

Workspace

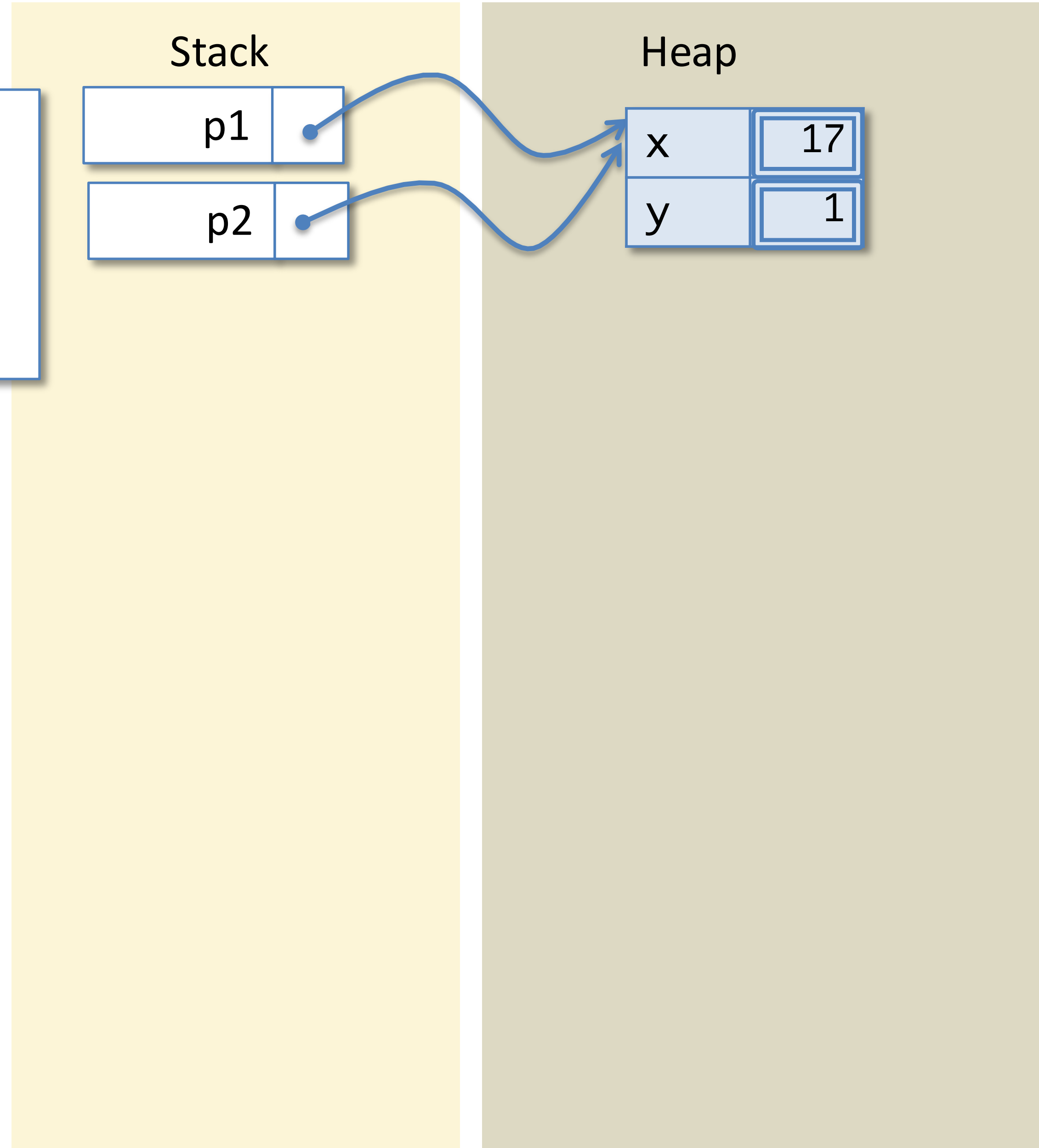
```
let ans : int =
    (); p1.x
```



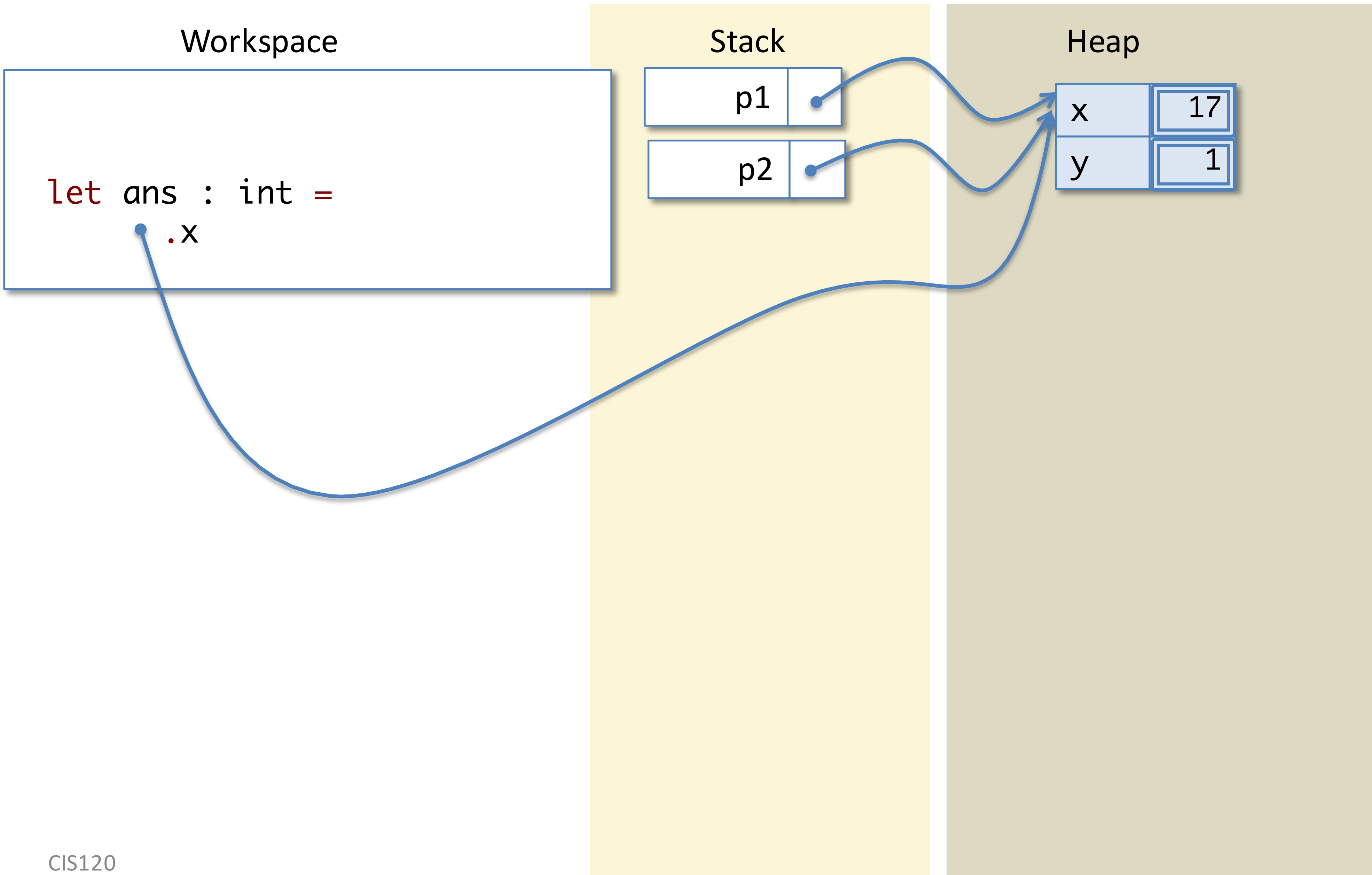
Look Up ‘p1’

Workspace

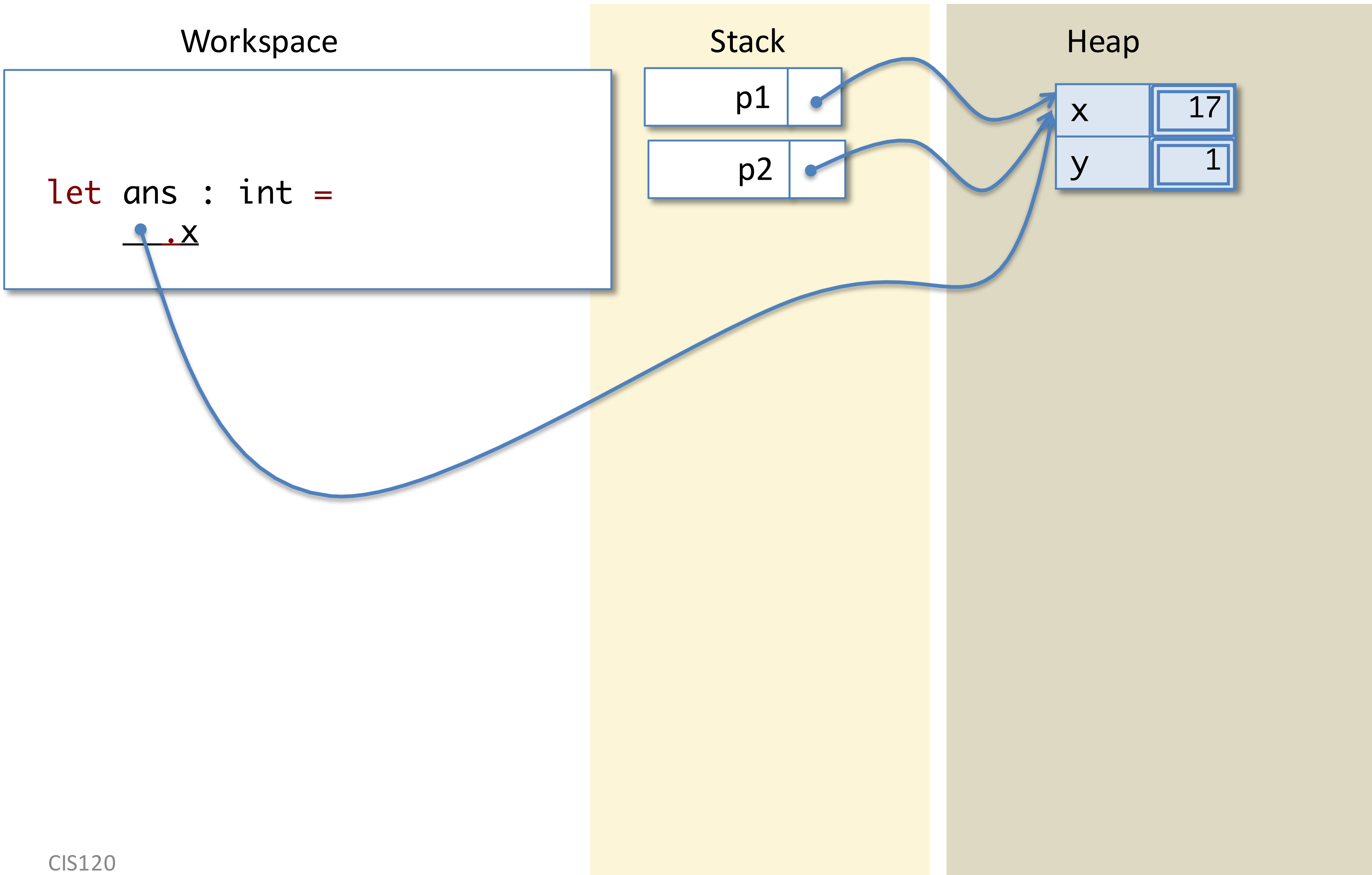
```
let ans : int =
    p1.x
```



Look Up ‘p1’



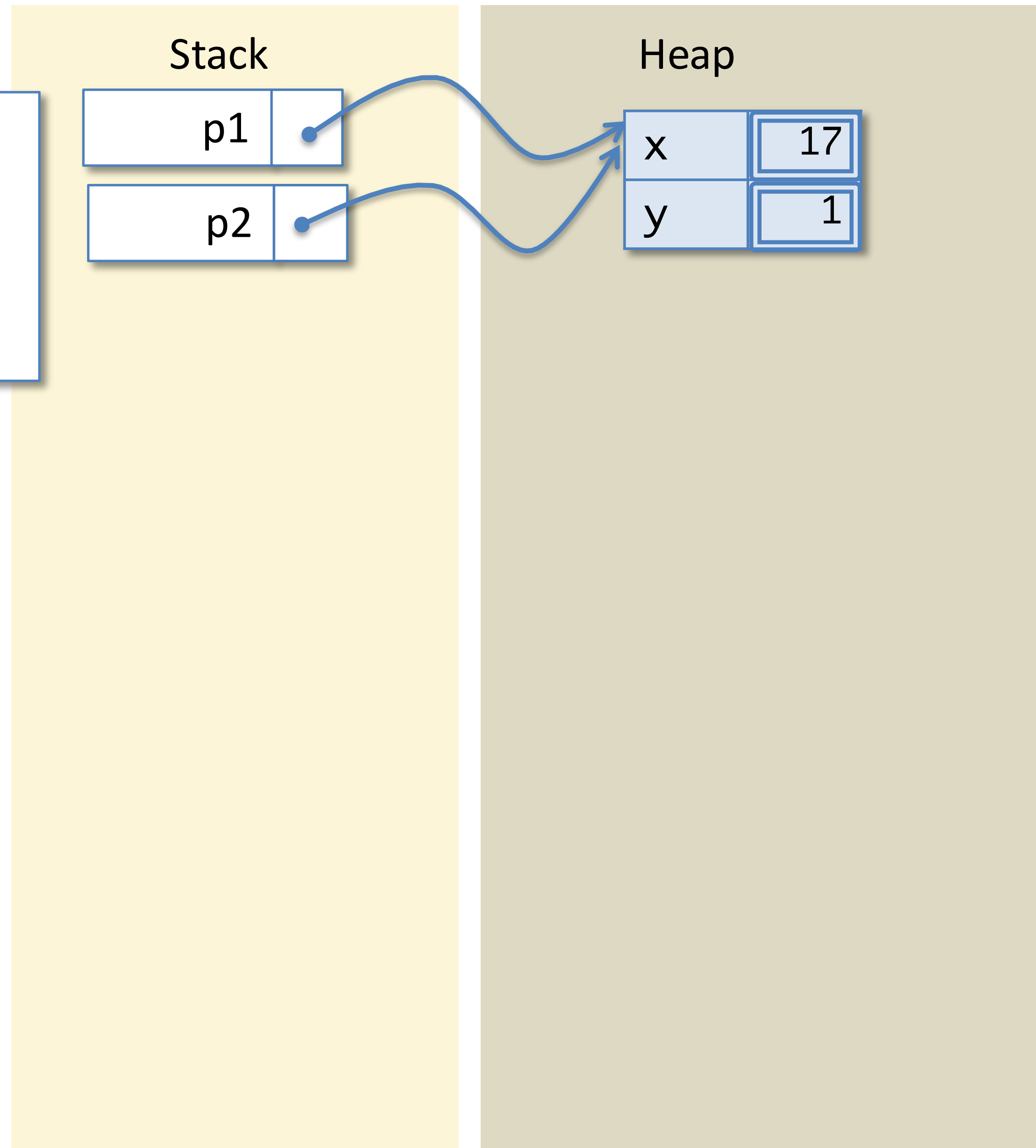
Project the ‘x’ field



Project the 'x' field

Workspace

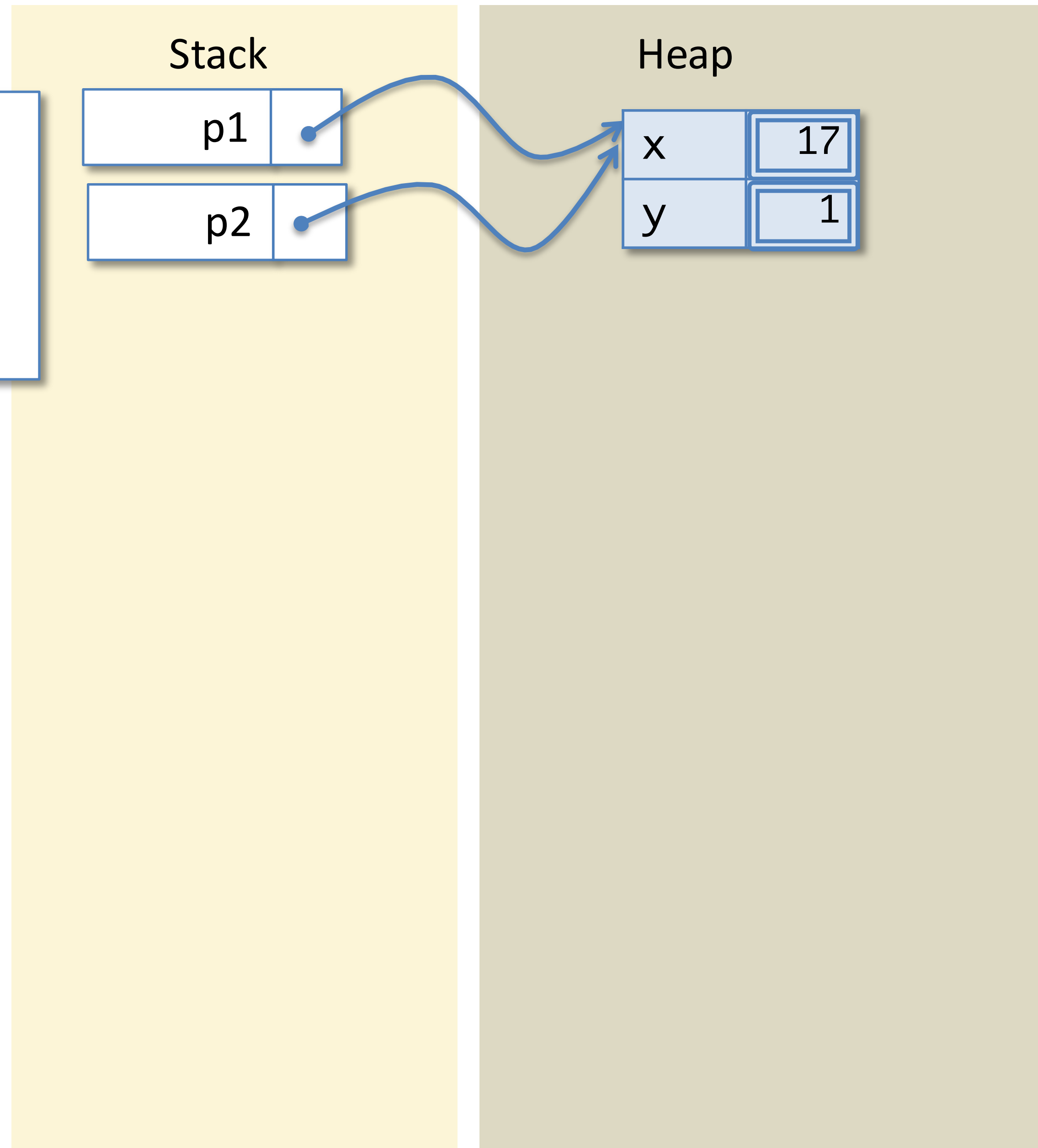
```
let ans : int =
    17
```



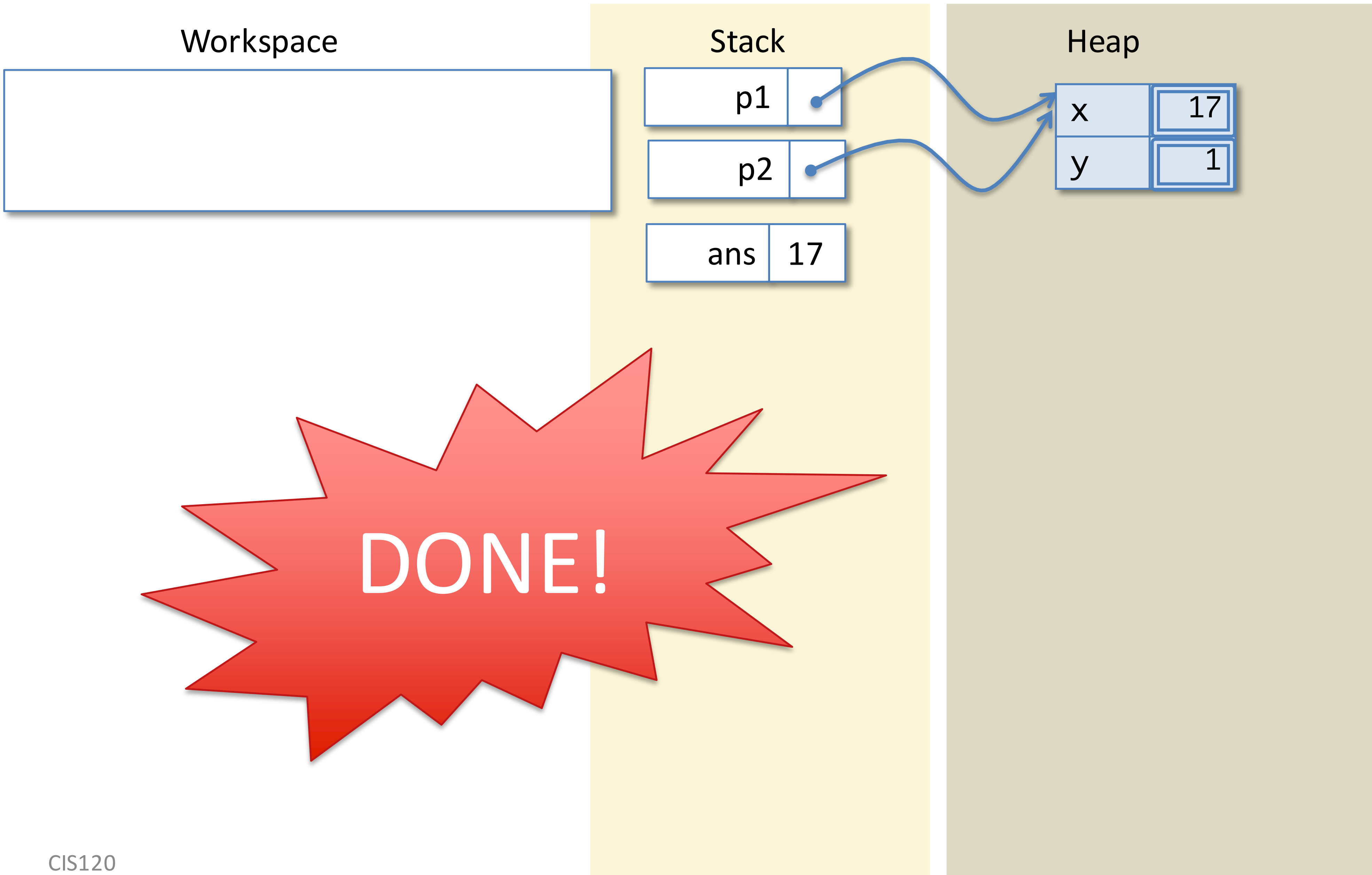
Let Expression

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: 17