Programming Languages and Techniques (CIS120)

Lecture 13

September 28th , 2015

Unit; Sequencing; Mutable State

Chapters 12, 13, 14

Announcements

Midterm 1

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

- My office hours: TODAY 3:30 – 5:00

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

- Commands (i.e. expressions of type unit) are useful because of their *side effects*: interactions with the external environment
  - e.g. printing output, reading user input, 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`

What is the type of f in the following program:

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

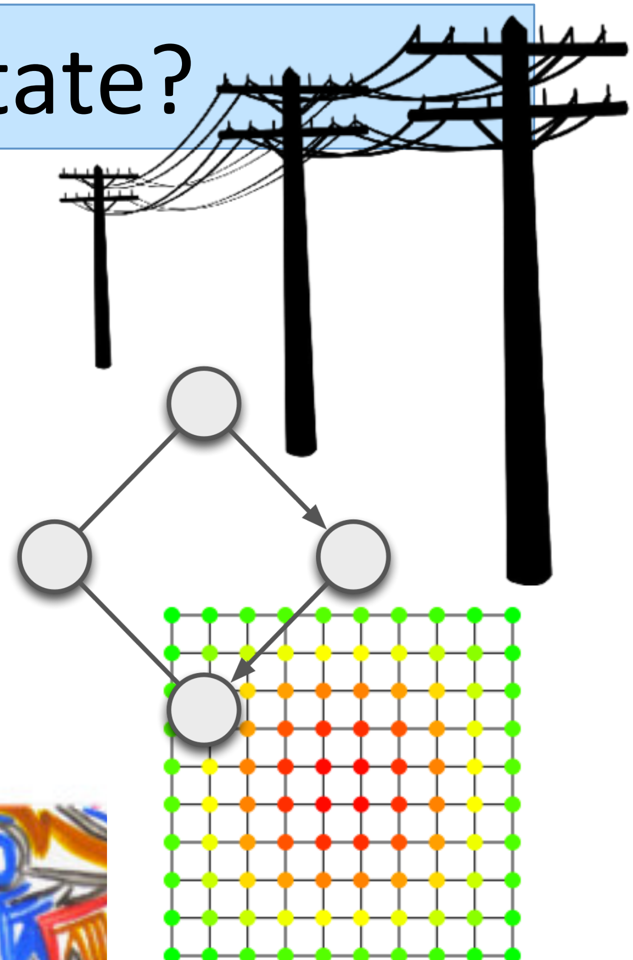
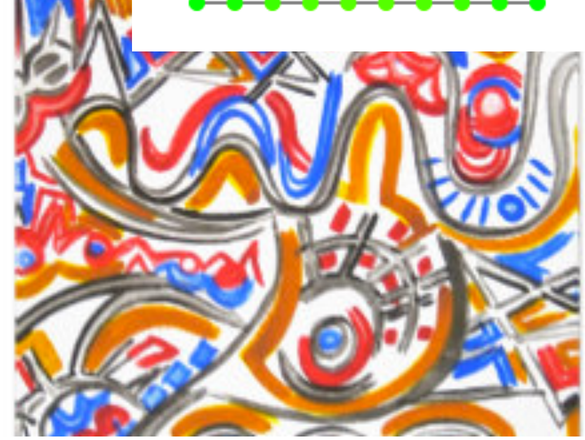
1. unit -> int
2. unit -> unit
3. int -> unit
4. int -> int
5. f is ill typed

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.

Mutable state

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))
```

Command that performs "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}

(* 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
```

What answer does the following function produce when called?

```
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

```
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
```

What answer does the following function produce when called?

```
let f (p1:point) (p2:point) : int =
  p1.x <- 17;
  p2.x <- 34;
  p1.x
```

1. 17
2. 34
3. sometimes 17 and sometimes 34
4. f is ill typed

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
```

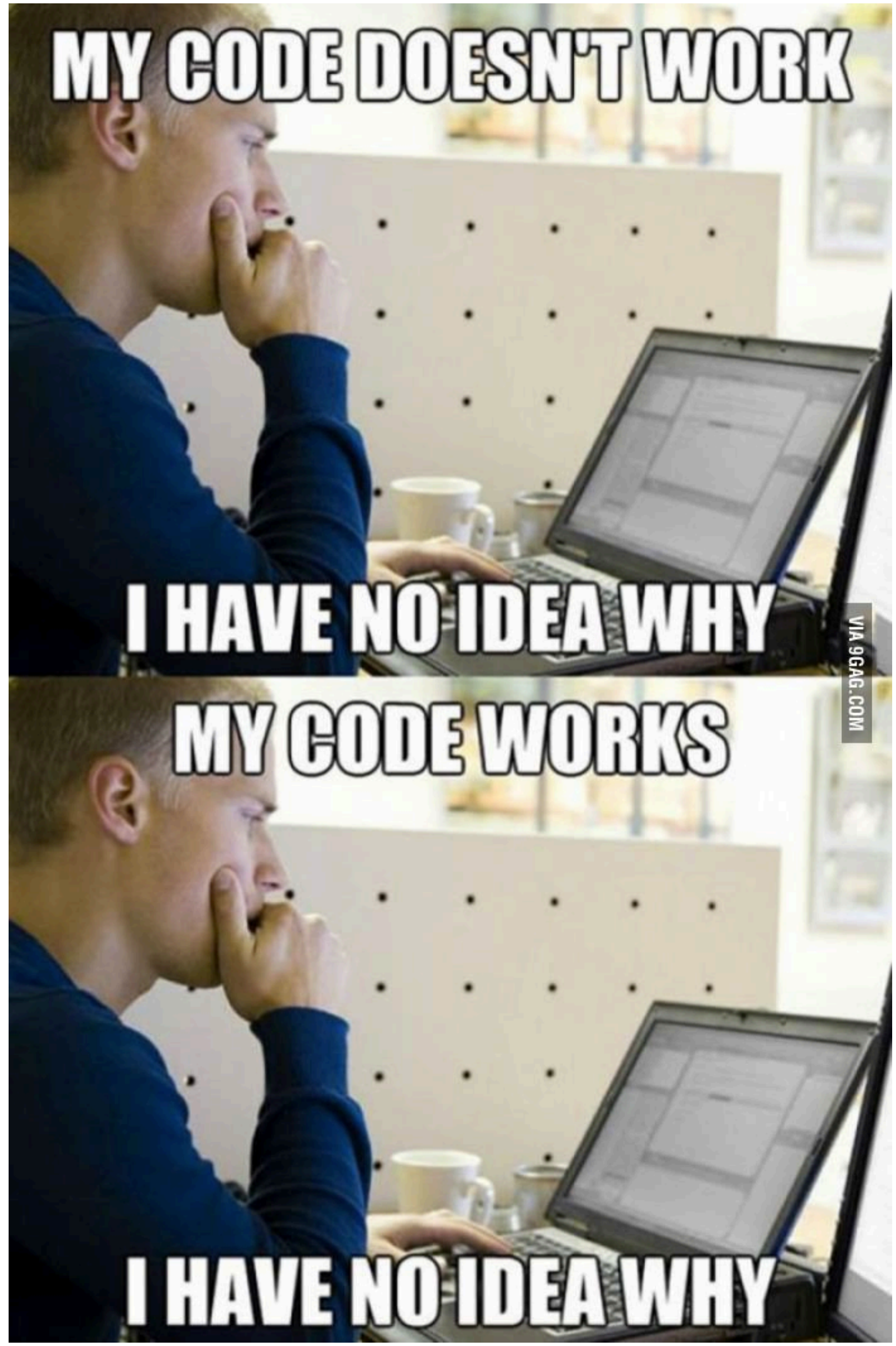
Two identifiers are said to be *aliases* if they both name the *same* mutable record. Inside `f`, `p1`, and `p2` might be aliased, depending on which arguments are passed to `f`.

Modeling Computation with Mutable State

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?

Mutable Records

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

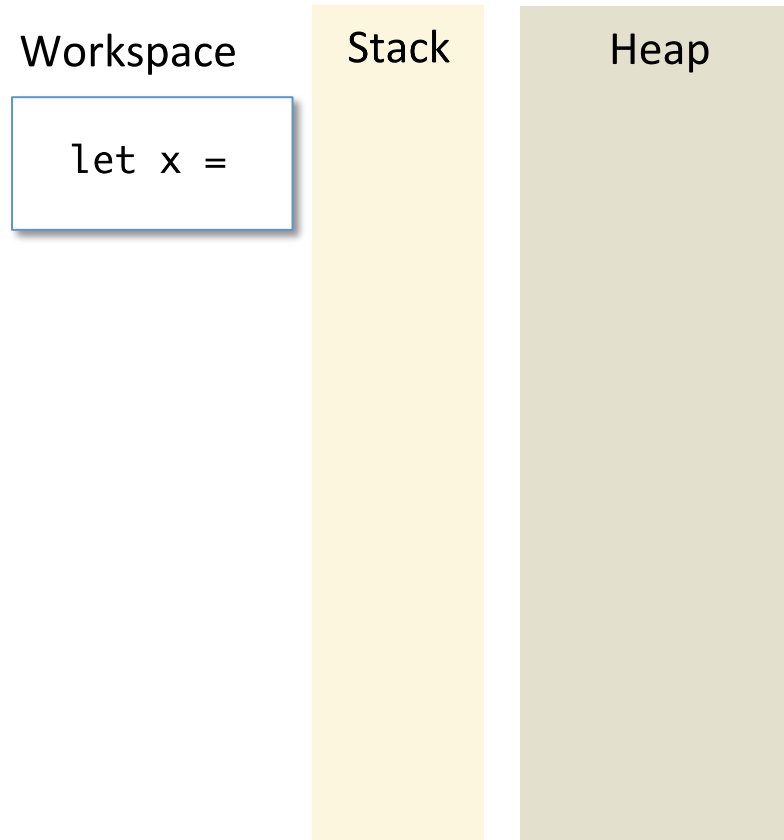
```
type point = {mutable x:int; mutable y:int}

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.

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

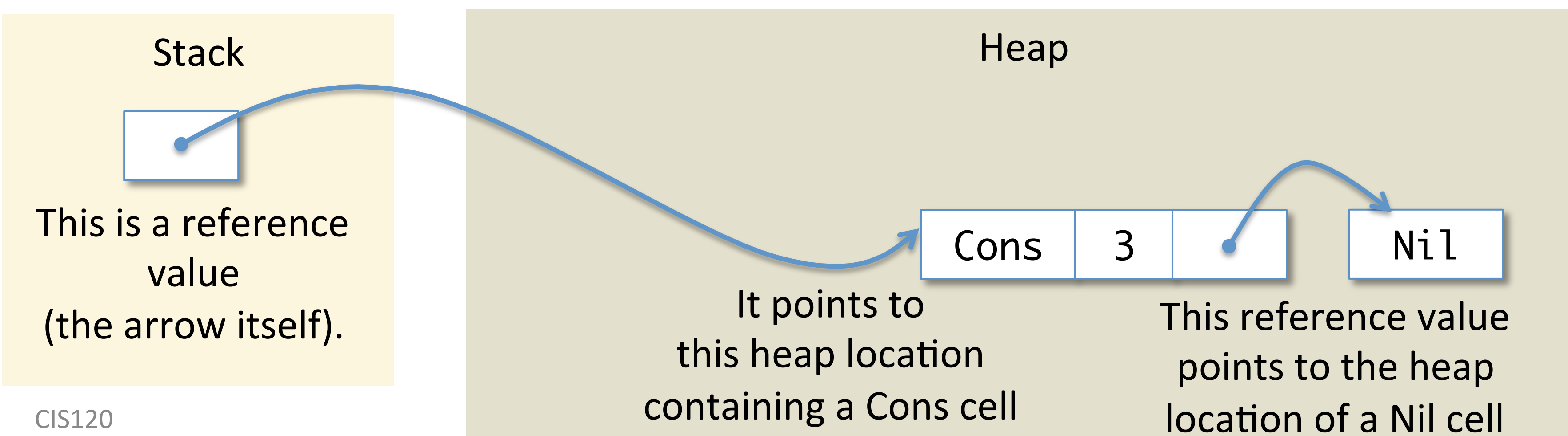
Values and References

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":

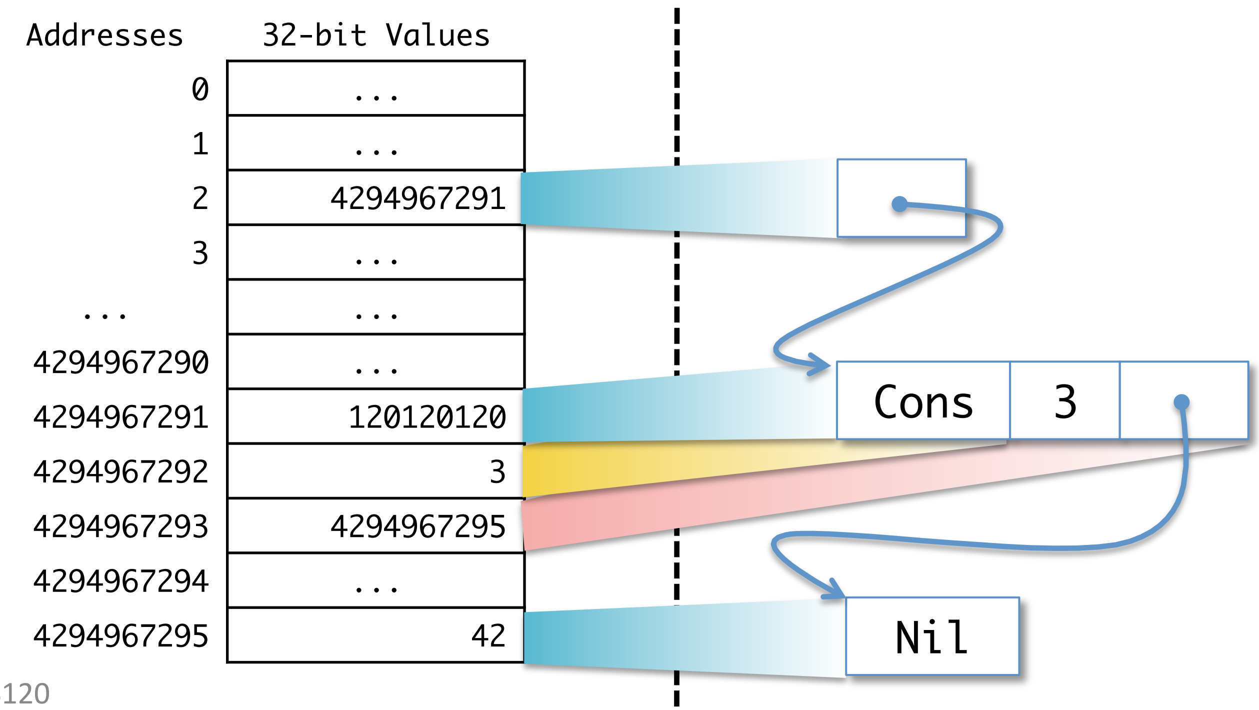
- The start of the arrow is the reference itself (i.e. the address).
- The arrow "points" to the value located at the reference's address.



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 "real" heap.



How we picture it.