

Programming Languages and Techniques (CIS1200)

Lecture 13

Partiality and Options

Records

Unit, Sequencing, and Commands

Mutable State and Aliasing

Chapters 11, 12, 13

Announcements (1)

- Midterm 1
 - Grades and solutions will be posted in a few days
- HW04 released tomorrow
 - Due in 2 weeks (October 15)
- Fall Break this week
 - No class on Friday
 - Recitations are cancelled for this week
 - No Office Hours from Thu-Sun
- No office hours for Benjamin next Monday (Oct 7)

Plan for Today

- Two more useful features
 - partiality via "options"
 - records with named components
- Then, a paradigm shift:
 - *mutable state*

Dealing with Partiality*

*A function is said to be *partial* if it is not defined for all inputs.

13: Which of these is a function that calculates the maximum value in a (generic) list?

0

1

0%

2

0%

3

0%

4

0%

Which of these is a function that calculates the maximum value in a (generic) list:

1.

```
let rec list_max (l:'a list) : 'a =  
  begin match l with  
  | [] -> []  
  | h :: t -> max h (list_max t)  
  end
```

2.

```
let rec list_max (l:'a list) : 'a =  
  fold max 0 l
```

3.

```
let rec list_max (l:'a list) : 'a =  
  begin match l with  
  | h :: t -> max h (list_max t)  
  end
```

4. None of the above

Answer: 4

Oops!

Not clear what to do when `list_max` is called with an empty list!

```
let rec list_max (l:'a list) : 'a =  
  begin match l with  
    | [] -> failwith "empty list"  
    | [h] -> h  
    | h::t -> max h (list_max t)  
  end
```

Client of list_max

```
(* string_of_max calls list_max *)  
let string_of_max (x:int list) : string =  
  string_of_int (list_max x)
```

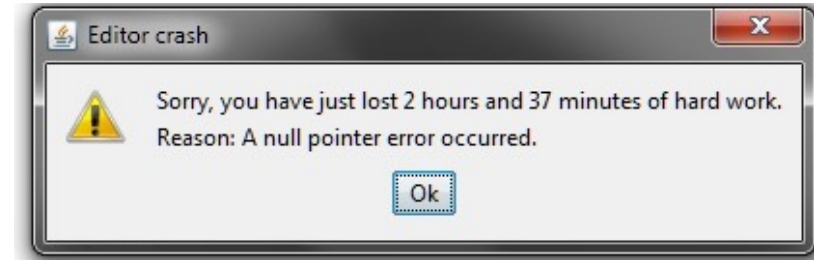
- Oops! `string_of_max` will fail if given `[]`
- Not so easy to debug if `string_of_max` is written by one person and `list_max` is written by another.
- Interface of `list_max` is not very informative
`val list_max : int list -> int`

Solutions to Partiality: Option 1

- *Abort the program:*
 - failwith “an error message”
 - Whenever it is called, failwith halts the program and reports the error message it is given.
- This solution is appropriate when:
 - *You know* that a certain case is impossible...
 - ...but the compiler isn't smart enough to figure out that the case is impossible
 - E.g., perhaps because there is an invariant on a data structure that the compiler doesn't understand

Solutions to Partiality: Option 2

- Return a *default or error value*
 - e.g. define `list_max []` to be `-1`
 - “Error codes” used often in C programs
 - `null` used often in Java



- But...
 1. What if `-1` (or whatever default you choose) really *is* the maximum value?
 - Can lead to hideous bugs if the default isn't handled properly by the callers.
 2. *Impossible* to implement generically!
 - No way to generically create a sensible default value for every possible type

Sir Tony Hoare, Turing Award winner and inventor of `null`, calls it his “*billion dollar mistake*”!

Default return values should be avoided if possible!

Solutions to Partiality: Option 3

Return something that *cannot*
be misinterpreted as a
legitimate, non-exceptional
result ...

Optional values

Solutions to Partiality: Option 3

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 = ...
```

- Compared with “null” (a legal value of *any type* in Java):
 - Caller can forget to check whether null was returned; results in NullPointerException or crash
- Modern language designs (e.g. Apple's Swift, Mozilla's Rust) distinguish between the types String (definitely not null) and String? (optional string)

Example: list_max

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

Revised Client of list_max

```
(* string_of_max calls list_max *)  
let string_of_max (l:int list) : string =  
  begin match (list_max l) with  
  | None -> "no maximum"  
  | Some m -> string_of_int m  
  end
```

- string_of_max will never fail
- The type of list_max makes it explicit that a *client* must check for partiality.

```
val list_max : int list -> int option
```

Revising the MAP interface

```
module type MAP = sig  
  
  type ('k, 'v) map  
  
  val empty      : ('k, 'v) map  
  val add        : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map  
  val remove     : 'k          -> ('k, 'v) map -> ('k, 'v) map  
  val mem        : 'k -> ('k, 'v) map -> bool  
  val get        : 'k -> ('k, 'v) map -> 'v option  
  val entries    : ('k, 'v) map -> ('k * 'v) list  
  val equals     : ('k, 'v) map -> ('k, 'v) map -> bool  
  
end
```

get returns an optional 'v.
Now its type isn't a lie!

Records

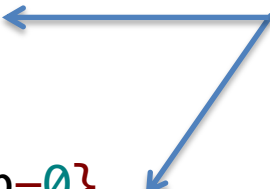
Records

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;}
```

Imperative Programming

And now for something completely different...



Being vs Doing

Imperative programming

- Most of the code we have written so far is focused on *being*.
 - An *expression* is just a complicated way of describing a *value*
 - *Computation* is just simplifying an expression until it can't be simplified any more
- But sometimes it is useful to *make things happen* outside the computer
 - E.g., `print_string`
- And sometimes it is useful to make things happen *inside* the computer as well
 - E.g., “mutating” a data structure in memory

Different views of imperative programming

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

- Code is a sequence of **statements** (a.k.a. commands) that **produce effects**
- Data structures 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 also produces effects along the way**
- Data structures are **immutable** by default; **must be explicitly declared to be mutable**

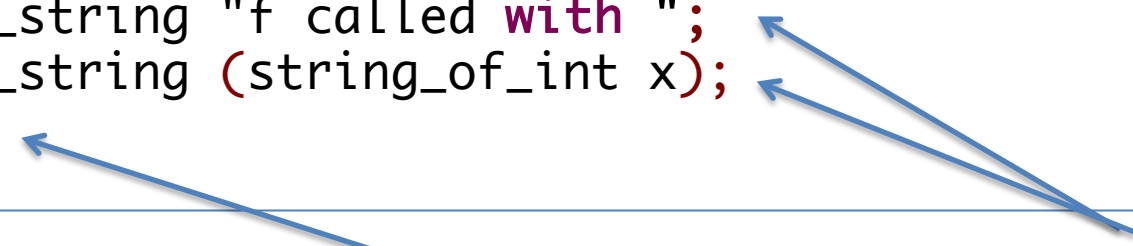
Commands, Sequencing, and Unit

What is the type of `print_string`?

Sequencing Commands and Expressions

We can *sequence* commands inside expressions using ‘;’

```
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

Unlike in C, Java, etc., ‘;’ doesn’t terminate a statement---it *separates* a command from an expression

The distinction between commands & expressions is artificial

- print_string is a **function** of type string -> unit
- Commands are just expressions of type unit

Sequencing Commands and Expressions

- Expressions of type **unit** are useful because of their *side effects* – they "*do*" stuff

```
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

Something to be Careful Of

What does this function do?

```
let f (x:int) : int =  
  if x < 0 then  
    print_string "f called with negative argument ";  
    print_string (string_of_int x)  
  else  
    print_string "f called with non-negative argument ";  
    print_string (string_of_int x);  
  x + x
```

File "gotcha.ml", line 5, characters 2-6:

```
5 |   else  
   ^^^^
```

Error: Syntax error

Something to be Careful Of

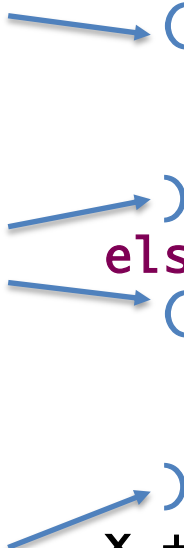
Compound commands inside then and else branches of if statements should be enclosed in `begin/end` or parens `()`

```
let f (x:int) : int =  
  if x < 0 then  
    → begin  
      print_string "f called with negative argument ";  
      print_string (string_of_int x)  
    → end  
  else  
    → begin  
      print_string "f called with non-negative argument ";  
      print_string (string_of_int x)  
    → end;  
  x + x
```

Something to be Careful Of

Compound commands inside then and else branches of if statements should be enclosed in `begin/end` or parens `()`

```
let f (x:int) : int =  
  if x < 0 then  
    (  
      print_string "f called with negative argument ";  
      print_string (string_of_int x)  
    )  
  else  
    (  
      print_string "f called with non-negative argument ";  
      print_string (string_of_int x)  
    );  
  x + x
```



In OCaml, `begin` and `end` are just syntactic sugar for `(` and `)`

unit: the trivial type

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

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

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

- ... and 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 (a 0-ary tuple)*
- Used for functions that don't take any **interesting** arguments

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

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

- ...And 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 (not so useful)

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

```
val x : unit
```

- Can pattern match against unit (useful in function definitions!)

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

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

- Unit is the result of an implicit else branch:

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

=

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


13: What is the type of f in the following program?



unit -> int

0%

unit -> unit

0%

int -> unit

0%

int -> int

0%

f is ill typed

0%

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

Mutable State

Opening a Whole New Can of Worms*



*t-shirt courtesy of ahrefs.com

Mutable Record Fields

- By default, records in OCaml are *immutable*: once created, 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 ' \leftarrow ' 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*

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

0

17

0%

something else

0%

sometimes 17 and sometimes someth...

0%

f is ill typed

0%

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

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

0

17

0%

something else

0%

sometimes 17 and sometimes someth...

0%

f is ill typed

0%

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:

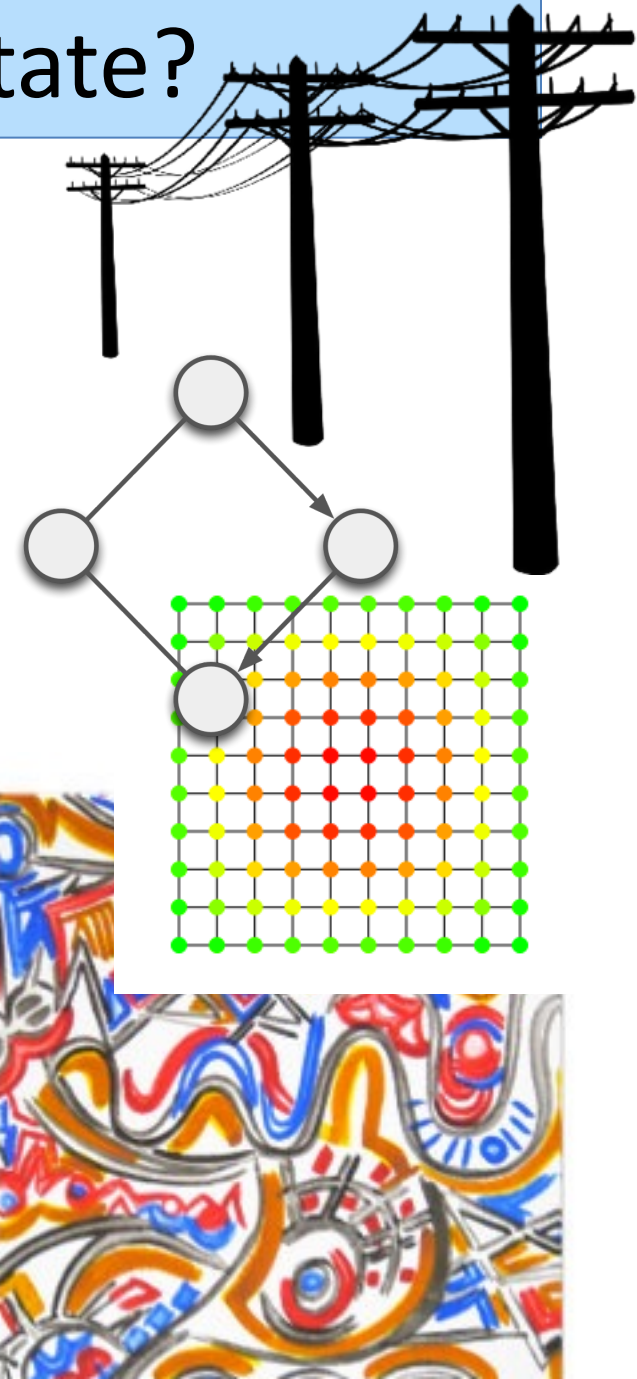
```
let p0 = {x=0; y=0} in  
  f p0 p0
```

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

Why Use Mutable State?

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



The Abstract State Machine

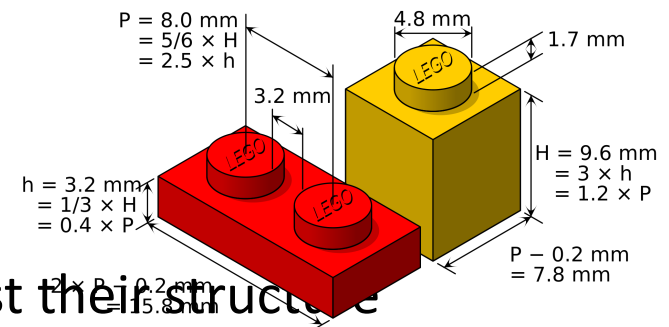
Location, Location, Location!

We need a new Computation Model

- The simple substitution model works well for *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 yield the same results
 - These properties justify "replace equals by equals" reasoning

- With mutable state...

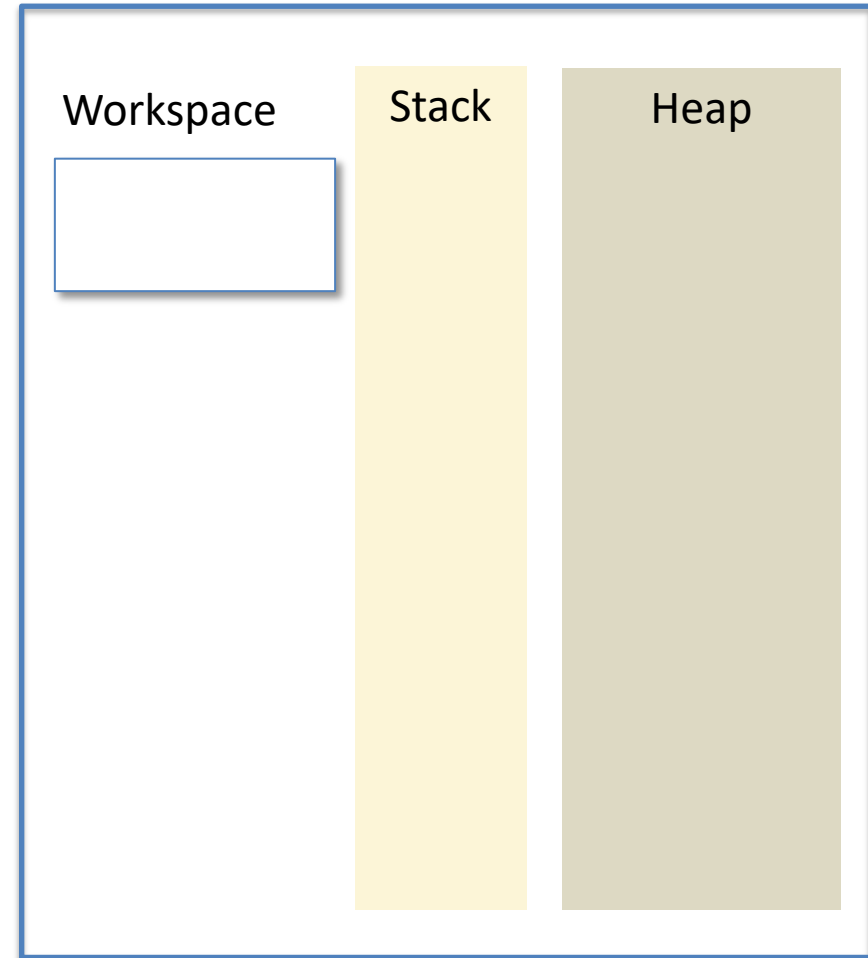
- The *location* of values matters, not just their structure
- Results returned by functions are not fully determined by their arguments (can also depend on "hidden" mutable state)



Abstract Stack Machine

Three “spaces”

- **workspace**
 - the expression the computer is currently simplifying
- **stack**
 - temporary storage for local variables and saved work
- **heap**
 - storage area for large data structures



Abstract stack machine

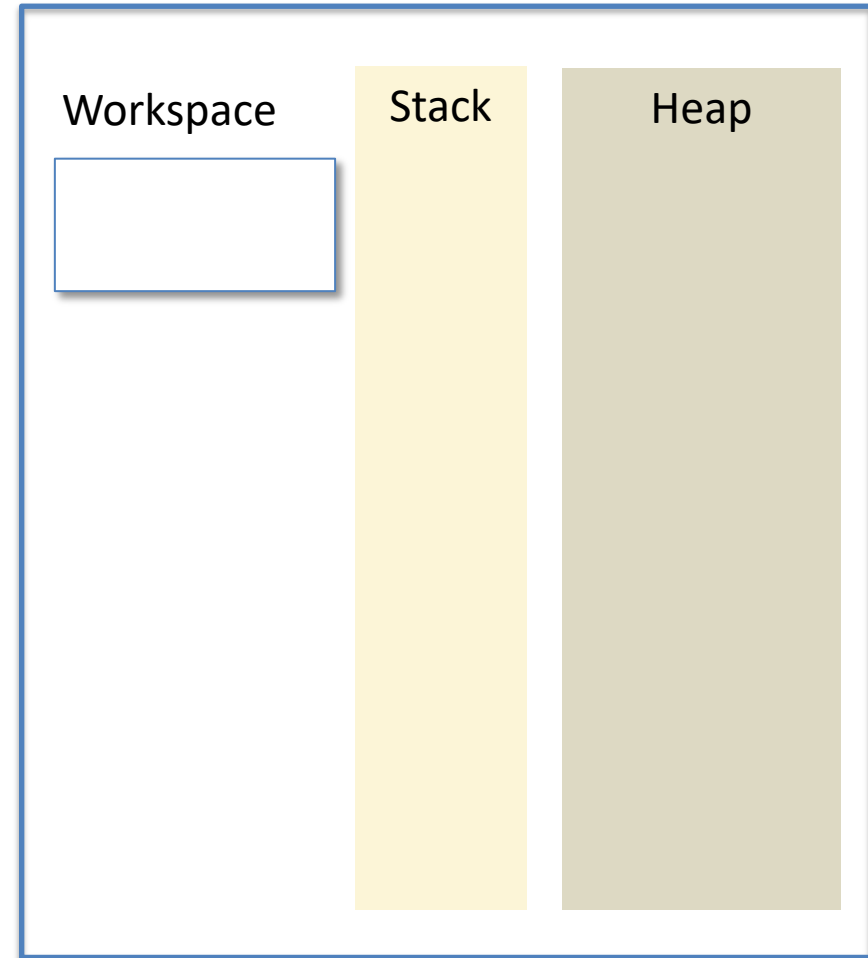
Abstract Stack Machine

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

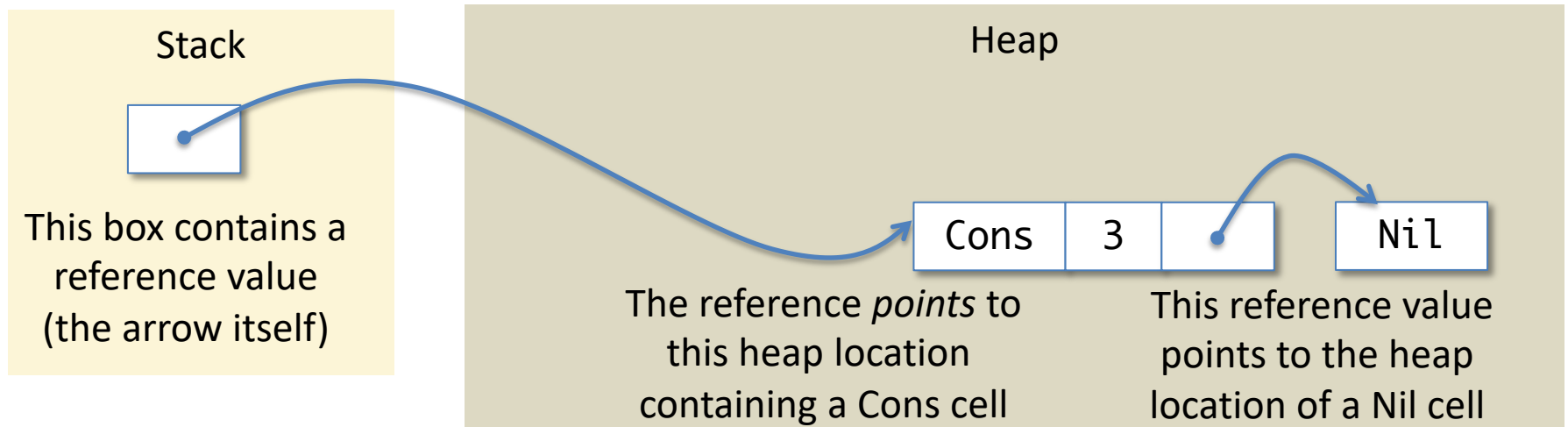
A *value* is either:

- 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”

- The arrow “points” to a box or cell located at this address
- Where we are storing this value also matters:



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

