Announcements:

- PS3 back on Monday in section
- Quiz #3 in class Tue Oct 18
- Coverage includes Monday section
- PS4 due Thursday Oct 20, 11:59PM
- Partner up for PS5!

- Main reason for bugs: side effects
- So far we haven't had them
  - You'll wish we didn't…
- Need to talk about how a computer actually stores information
  - OCaml support for side effects

- We've been working with the purely functional fragment of OCaml.
  - That is, we've been working with the subset of the language that does not include computational effects (also known as side effects) other than printing.

- In particular, whenever we coded a function, we never changed variables or data.
  - Rather, we always computed new data.

- For instance, when we wrote code for an abstract data type such as a stack, queue, or dictionary, the operations to insert an item into the data structure didn't affect the old copy of the data structure.
- Instead, we always built a new data structure with the item appropriately inserted.
  - (Note that the new data structure might refer to the old data structure, so this isn't as inefficient as it first sounds.)

- For the most part, coding in a functional style (i.e., without side effects) is a "good thing" because it's easier to reason locally about the behavior of the code.
    - For instance, when we code purely functional queues or stacks, we don't have to worry about a non-local change to a queue or stack.
    - However, in some situations, it is more efficient or clearer to destructively modify a data structure than to build a new version.
    - In these situations, we need some form of **mutable** data structures.

- Like most imperative programming languages, OCaml provides support for mutable data structures,
    - Unlike languages such as C, C++, or Java, they are not the default.

- Thus, programmers encouraged to code purely functionally by default
    - only resort to mutable data structures when absolutely necessary.
    - In addition, unlike imperative languages, OCaml provides no support for mutable *variables.*

- In other words, the value of a variable cannot change in OCaml. Rather, all mutations must occur through data structures.
- There are only two built-in mutable data structures in OCaml: refs and arrays.

- OCaml supports imperative programming through the primitive parameterized `ref` type.
    - A value of type "int ref" is a pointer to a location in memory, where the location in memory contains an integer.
        - It's analogous to "int*" in C/C++ or "Integer" in Java (but not "int" in Java).
        - Like lists, refs are polymorphic, so in fact, we can have a ref (i.e., pointer) to a value of any type.
- A partial signature for refs is below:

```
module type REF =
    sig
      type 'a ref

      (* ref(x) creates a new ref containing x *)
      val ref : 'a -> 'a ref

      (* !x is the contents of the ref cell x *)
      val (!) : 'a ref -> 'a

      (* Effects: x := y updates the contents of x
       * so it contains y. *)
      val (:=) : 'a ref * 'a -> unit
    end
```

- A ref is like a box that can store a single value. By using the := operator, the value in the box can be changed as a side effect.
    - It is important to distinguish between the value that is stored in the box, and the box itself.
    - A ref is the simplest **mutable** data structure.
    - A mutable data structure is one that can be changed imperatively, or **mutated**.

- The following code shows an example where we use a ref:

```
let x : int ref = ref 3 in
    let y : int = !x in
        (x := (!x) + 1);
        y + (!x)
    end
```

- The code above evaluates to 7. Let's see why:
    - The first line "let x:int ref = ref 3" creates a new ref cell, initializes the contents to 3, and then returns a reference (i.e., pointer) to the cell and binds it to x.
    - The second line "let y:int = !x" reads the contents of the cell referenced by x, returns 3, and then binds it to y.
    - The third line "x := (!x) + 1;" evaluates "!x" to get 3, adds one to it to get 4, and then sets the contents of the cell referenced by x to this value.
    - The fourth line "y + (!x)" returns the sum of the values y (i.e., 3) and the contents of the cell referenced by x (4).
    - Thus, the whole expression evaluates to 7.

- Here's an example of a mutable stack build using refs:

```
module type MUTABLE_STACK =
      sig
       (* An 'a mstack is a mutable stack of 'a elements *)
         type 'a mstack
         (* new() is a new empty stack *)
         val new : unit -> 'a mstack
         (* Effects: push(m,x) pushes x onto m *)
         val push : 'a mstack * 'a -> unit
         (* pop(m) is the head of m.
          * Effects: pops the head off the stack. *)
         val pop : 'a mstack -> 'a option
      end

module Mutable_Stack : MUTABLE_STACK =
  struct
     (* A mutable stack is a reference
      * to the list of values, with the top
      * of the stack at the head. *)
     type 'a mstack = ('a list) ref
     let new():'a mstack = ref([])
     let push(s:'a mstack, x:'a):unit =
         s := x::(!s)
     let pop(s:'a stack):'a option =
         match (!s) with
           [] => NONE
         | hd::tl => (s := tl; SOME(hd))
   end
```

- A good exercise for you is to implement mutable versions of queues, priority queues, dictionaries, or any other data structure that we've seen in class thus far using refs

- Substitution model and refs

- The substitution model that we've seen so far explains how computation works as long as no imperative features of OCaml are used.
    - This model describes computation as a sequence of rewrite steps in which a program subexpression is replaced by another until no further rewrites are possible.
    - However, imperative features introduce the possibility of **state** : an executing OCaml program is accompanied by a current memory state that also changes as computation proceeds.

- We don't want to get into the details of how memory heaps work yet, so we will use a simple abstract model of state.
    - A memory M is a collection of memory cells each with its own unique name.
    - We will call these names **locations**; a location is an abstract version of a memory address at the hardware level.
    - Given a location, we can look up in the memory what value is stored at that location.
    - As the program executes, the contents of some memory locations may change.

- One way to visualize the execution is the memory consists of a large (actually, infinite) number of boxes, each of which can contain a single value.

    - At any given point during execution, some boxes are in use and others are empty.

    - Each box has a unique name (its location) and this location can be used to find the single box with that name.

    - Given a memory, we can always find a box that is unused.

- Ref operations

- There are three principal operations on references: creation using the **`ref`** operator, deferencing using **`!`**, and update using **`:=`**.

    - Each of these operations has an associated reduction that is used when evaluating it.

    - In order to explain what these operations do, a new kind of expression is needed, representing a location.

- We will write the syntactic metavariable *loc* to represent a location.
    - For the purposes of explaining how to evaluate OCaml, we assume that there is an infinitely large set of locations (called **Loc**) available for use when evaluating programs, even though the actual memory is finite.
    - We don't care what the elements of **Loc** actually are. We can think of them as memory addresses, as integers, or even as strings. All that matters is that we can tell two different elements of **Loc** apart.

`ref`

- The ref operation creates a new location. It is reduced once its argument is a value, creating a new location.

  ref *v* --> *loc*
- The new location *loc* is one that is unused in the current memory.
  - This evaluation step also has a side effect: the memory cell named *loc* is made to contain the value *v*.
- This rule introduces a *loc* expression into the running program.
  - This is a bit different from all the evaluation rules that we have seen till this point, because a *loc* expression cannot occur in the original OCaml program.
- This isn't a problem; we have to remember that our models of evaluation are useful fictions.
  - As long as the model gives the right answer for what happens when the program runs, we are satisfied.
  - In OCaml, if a program evaluates to a location, it is printed as a ref expression (example below)

- Equality on references in OCaml is slightly odd. If we test two expressions of the type 'a ref using =, OCaml will actually check if their contents are equal.
- This is generally what you want, but you can also use == to check if the refs themselves are equal (i.e., if the two refs point to the same block of memory).

```
# let a = ref 2;;
val a : int ref = {contents = 2}
# let b = ref 2;;
val b : int ref = {contents = 2}
# a = b;;
- : bool = true
# a == b;;
- : bool = false
```

!

- The dereference (`!`) operation finds the value stored at a given location:

  `!` *loc* `-->` *v*

- Of course, the value v that replaces the subexpression `!`*loc* is the value found in the memory cell named *loc*.

:=

- The update (`:=`) operation updates the value stored at a given location:

  loc `:=` *v* `-->` `()`

- It evaluates to the unit value, but has the side effect of updating the memory location named *loc* to contain *v*.

Example

- Consider the following OCaml example:

```
let x = ref 0 in
let y = x in
    x := 1;
    !y
end
```

- What does this evaluate to? We can use the model about to figure it out:

```
let x = ref 0 in
let y = x in
    x := 1; !y
end
Memory: (empty)
-->
let x = loc1 in
let y = x in
    x := 1; !y
end
Memory: (loc1 = 0)
-->   (substitute loc1 for x)
let y = loc1 in
    loc1 := 1; !y
end
Memory: (loc1 = 0)
-->   (substitute loc1 for y)
loc1 := 1; !loc1
Memory: (loc1 = 0)
--> !loc1
Memory: (loc1 = 1)
--> 1
Memory: (loc1 = 1)
```