CMSC 330: Organization of Programming Languages

Relationships between Functional, Imperative, and Object-Oriented Programming

OCaml Language Choices

- Implicit or explicit declarations?
  - Explicit – variables must be introduced with let before use
  - But you don't need to specify types
- Static or dynamic types?
  - Static – but you don't need to state types
  - OCaml does *type inference* to figure out types for you
  - Good: less work to write programs
  - Bad: easier to make mistakes, harder to find errors

So Far, only Functional Programming

- We haven't given you any way so far to change something in memory in OCaml
  - All you can do is create new values from old
- This actually makes programming easier !
  - Don't care whether data is shared in memory
    - Aliasing is irrelevant
  - Provides strong support for compositional reasoning and abstraction
    - Ex: Calling a function f with argument x always produces the same result

What about Imperative Programming?

- In C or Java, we're used to doing things like:

```
int x = 3;
int y = 4;

void foo(void) {
  x = 42;
  y = x + 2;
}

int bar(void) {
  return x + y;
}
```

  - Can we model this without imperative constructs?
    - Imperative = able to *change* values in memory

Idea: "Thread" State through Fns

```
type state = (char * int) list
let read (s:state) (x:char):int = List.assoc x s
let write (s:state) (x:char) (i:int):state =
  let s' = List.remove_assoc s x in
  (x,i)::s'

let foo (s0:state):state =  (* could change to state*unit *)
  let s1 = write s0 'x' 42 in
  let s2 = write s1 'y' ((read s1 'x') + 2) in
  s2

let bar (s0:state):(state*int) =
  (s0, (read s0 'x') + (read s0 'y'))
```


This Can Actually be a Good Idea

- The Haskell language is *purely functional*
  - No way to write to memory, ever
- But, you can play the trick we just saw
  - In Haskell, something that behaves like the state type is a *monad*
  - Used for a bunch of different things
    - And there's some interesting theory to go with it
- OCaml is only *mostly functional*
  - It does actually have imperative constructs

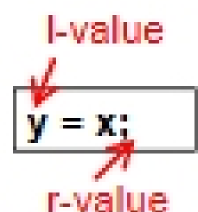
Imperative OCaml

- There are three basic operations on memory:
  - `ref : 'a -> 'a ref`
    - Allocate an updatable reference
  - `! : 'a ref -> 'a`
    - Read the value stored in reference
  - `:= : 'a ref -> 'a -> unit`
    - Write to a reference

```
let x = ref 3  (* x : int ref *)
let y = !x
x := 4
```

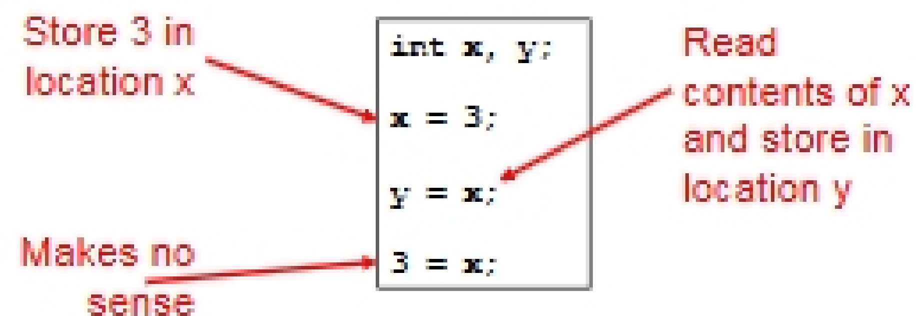

Comparison to L- and R-values

- Recall that in C/C++/Java, there's a strong distinction between l- and r-values
  - An *r-value* refers to just a value, like an integer
  - An *l-value* refers to a location that can be written



- A variable's meaning depends on where it appears
  - On the right-hand side, it's an r-value, and it refers to the contents of the variable
  - On the left-hand side of an assignment, it's an l-value, and it refers to the location the variable is stored in

L-Values and R-Values (cont'd)



- Notice that x, y, and 3 all have type `int`

Comparison to OCaml

```
int x, y;

x = 3;

y = x;

3 = x;
```

```
let x = ref 0;;
let y = ref 0;;

x := 3;;  (* x : int ref *)

y := (!x);;

3 := x;;  (* 3 : int; error *)
```

- In OCaml, an updatable location and the contents of the location have different types
  - The location has a `ref` type

Capturing a ref in a Closure

- We can use refs to make things like counters that produce a fresh number "everywhere"

```
let next =
  let count = ref 0 in
    function () ->
      let temp = !count in
        count := (!count) + 1;
        temp;;

# next ();;
- : int = 0
# next ();;
- : int = 1
```


Semicolon Revisited; Side Effects

- Now that we can update memory, we have a real use for ; and () : unit
  - e1; e2 means evaluate e1, throw away the result, and then evaluate e2, and return the value of e2
  - () means "no interesting result here"
  - It's only interesting to throw away values or use () if computation does something besides return a result
- A *side effect* is a visible state change
  - Modifying memory
  - Printing to output
  - Writing to disk