CMSC 330: Organization of Programming Languages

OCaml 4
Data Types & Modules
OCaml Data

So far, we’ve seen the following kinds of data
• Basic types (int, float, char, string)
• Lists
  ➢ One kind of data structure
  ➢ A list is either [ ] or h::t, deconstructed with pattern matching
• Tuples
  ➢ Let you collect data together in fixed-size pieces
• Functions

How can we build other data structures?
• Building everything from lists and tuples is awkward
User Defined Types

- **type** can be used to create new names for types
  - Useful for combinations of lists and tuples

**Examples**

- **type** my_type = int * (int list)
  
  
  ((3, [1; 2]) : my_type)

- **type** my_type2 = int * char * (int * float)
  
  
  ((3, ‘a’, (5, 3.0)) : my_type2)
public interface Shape {
    public double area();
}

class Rect implements Shape {
    private double width, length;
    Rect (double width, double length) {
        this.width = width;
        this.length = length;
    }
    double area() {
        return width * length;
    }
}

class Circle implements Shape {
    private double radius;
    Circle (double radius) {
        this.radius = radius;
    }
    double area() {
        return radius * radius * 3.14159;
    }
}

How to achieve this in Ocaml?
Data Types

- **type** can also be used to create **variant types**
  - Equivalent to C-style unions

```
type shape =
    Rect of float * float (* width*length *)
  | Circle of float (* radius *)
```

- **Rect** and **Circle** are **value constructors**
  - Here a **shape** is either a **Rect** or a **Circle**

- Constructors must begin with uppercase letter
Data Types (cont.)

Use pattern matching to **deconstruct** values

- s is a **shape**
- Do different things for s depending on its constructor

```ml
let area s =
  match s with
  | Rect (w, l) -> w *. l
  | Circle r -> r *. r *. 3.14

area (Rect (3.0, 4.0))
area (Circle 3.0)
```

Unlike classes in Java, shape functions are separate from shape data – will later examine tradeoffs
Data Types (cont.)

```ocaml
type shape =
  Rect of float * float (* width*length *)
| Circle of float (* radius *)

let lst = [Rect (3.0, 4.0) ; Circle 3.0]
```

- What's the type of lst?
  ```ocaml```
  ```ocaml```
- What's the type of lst's first element?
  ```ocaml```
Option Type

<table>
<thead>
<tr>
<th>Option Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>This option type can work with any kind of data</td>
</tr>
<tr>
<td>In fact, this option type is built into Ocaml</td>
</tr>
<tr>
<td>Specify as: int option, char option, etc...</td>
</tr>
</tbody>
</table>

```ocaml
type optional_int =
    None
  | Some of int
let add_with_default a x = match x with
    None -> a + 42
  | Some n -> a + n
add_with_default 3 None    (* 45 *)
add_with_default 3 (Some 4) (* 7  *)
```
Recursive Data Types

- We can build up lists with variant types

```ocaml
type 'a list =
    Nil
  | Cons of 'a * 'a list

let rec len = function
    Nil  ->  0
  | Cons (_, t)  ->  1 + (len t)

len (Cons (10, Cons (20, Cons (30, Nil)))))
```

- Won’t have nice \([1; 2; 3]\) syntax for this kind of list
Data Type Representations

- Values in a data type are stored
  1. Directly as integers
  2. As pointers to blocks in the heap

```
type t =
  A of int
| B
| C of int * int
| D
```
Exercise: A Binary Tree Data Type

- Write type `bin_tree` for binary trees over `int`
  - Trees should be ordered (binary search tree)
- Implement the following
  ```
  empty : bin_tree
  is_empty : bin_tree -> bool
  member : int -> bin_tree -> bool
  insert : int -> bin_tree -> bin_tree
  remove : int -> bin_tree -> bin_tree
  equal : bin_tree -> bin_tree -> bool
  fold : (int -> 'a -> 'a) -> bin_tree
                   -> 'a -> 'a
  ```
Modules

- So far, most everything we’ve defined has been at the “top-level” of OCaml
  - This is not good software engineering practice

- A better idea: Use modules to group associated types, functions, and data together
  - Avoid polluting the top-level with unnecessary stuff

- For lots of sample modules, see the OCaml standard library
Creating A Module In OCaml

```ocaml
module Shapes =

struct

  type shape =
  | Rect of float * float (* wid*len *)
  | Circle of float (* radius *)

  let area = function
    | Rect (w, l) -> w *. l
    | Circle r -> r *. r *. 3.14

  let unit_circle = Circle 1.0

end;;
```
Creating A Module In OCaml (cont.)

```ocaml
module Shapes =
  struct
    type shape = ...
    let area = ...
    let unit_circle = ...
  end;;
unit_circle;; (* not defined *)
Shapes.unit_circle;;
Shapes.area (Shapes.Rect (3.0, 4.0));;
open Shapes;; (* import names into curr scope *)
unit_circle;; (* now defined *)
```
Another reason for creating a module is so we can hide details

- Ex: Binary tree module
  - May not want to expose exact representation of binary trees
- This is also good software engineering practice
  - Prevents clients from relying on details that may change
  - Hides unimportant information
  - Promotes local understanding (clients can't inject arbitrary data structures, only ones our functions create)
Module Signatures

module type FOO =
  sig
    val add : int -> int -> int
  end;;
module Foo : FOO =
  struct
    let add x y = x + y
    let mult x y = x * y
  end;;
Foo.add 3 4;;    (* OK *)
Foo.mult 3 4;;   (* not accessible *)
Module Signatures (cont.)

- Convention: Signature names in all-caps
  - This isn't a strict requirement, though

- Items can be omitted from a module signature
  - This provides the ability to hide values

- The default signature for a module hides nothing
  - You’ll notice this is what OCaml gives you if you just type in a module with no signature at the top-level
Abstract Types In Signatures

module type SHAPES =
  sig
    type shape
    val area : shape -> float
    val unit_circle : shape
    val make_circle : float -> shape
    val make_rect : float -> float -> shape
  end;;

module Shapes : SHAPES =
  struct
    ...
    let make_circle r = Circle r
    let make_rect x y = Rect (x, y)
  end

- Now definition of shape is hidden
Abstract Types In Signatures

- `Shapes.unit_circle`
- `: Shapes.shape = <abstr>` (* OCaml won’t show impl *)
- `Shapes.Circle 1.0`
- Unbound Constructor `Shapes.Circle`
- `Shapes.area (Shapes.make_circle 3.0)`
- `: float = 29.5788`
- `open Shapes;;`
- `(* doesn’t make anything abstract accessible *)`

- How does this compare to modularity in...
  - C?
  - C++?
  - Java?
Modules In Java

- Java **classes** are like modules
  - Provides implementations for a group of functions
  - But classes can also
    - Instantiate objects
    - Inherit attributes from other classes

- Java **interfaces** are like module signatures
  - Defines a group of functions that may be used
  - Implementation is hidden
Modules In C

- .c files are like modules
  - Provides implementations for a group of functions

- .h files are like module signatures
  - Defines a group of functions that may be used
  - Implementation is hidden

- Usage is not enforced by C language
  - Can put C code in .h file
Module In Ruby

- Ruby explicitly supports modules
  - Modules defined by `module … end`
  - Modules cannot
    - Instantiate objects
    - Derive subclasses

```
puts Math.sqrt(4)  # 2
puts Math::PI      # 3.1416

include Math       # open Math
puts Sqrt(4)       # 2
puts PI            # 3.1416
```
OCaml Exceptions

```
exception My_exception of int
let f n =
  if n > 0 then
    raise (My_exception n)
  else
    raise (Failure "foo")
let bar n =
  try
    f n
  with My_exception n ->
    printf "Caught %d\n" n
  | Failure s ->
    printf "Caught %s\n" s
```
Exceptions (cont.)

- Exceptions are declared with `exception`
  - They may appear in the signature as well
- Exceptions may take arguments
  - Just like type constructors
  - May also have no arguments
- Catch exceptions with `try...with...`
  - Pattern-matching can be used in `with`
  - If an exception is uncaught
    - Current function exits immediately
    - Control transfers up the call chain
    - Until the exception is caught, or until it reaches the top level
OCaml Exceptions (cont.)

- Exceptions may be thrown by I/O statements
  - Common way to detect end of file
  - Need to decide how to handle exception

- Example

```ocaml
try
  (input_char stdin) (* reads 1 char *)
with End_of_file -> 0 (* return 0? *)
try
  read_line () (* reads 1 line *)
with End_of_file -> "" (* return ""?)
```
So Far, Only Functional Programming

- We haven’t given you **any** way so far to change something in memory
  - All you can do is create new values from old
- This actually makes programming easier in some ways
  - 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
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 In C

- Store 3 in location x
- Makes no sense
- Read contents of x and store in location y

```c
int x, y;
x = 3;
y = x;
3 = x;
```

Notice that x, y, and 3 all have type `int`
### Comparison To OCaml

<table>
<thead>
<tr>
<th>C</th>
<th>OCaml</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x;</td>
<td>let x = ref 0;;</td>
</tr>
<tr>
<td>Int y;</td>
<td>let y = ref 0;;</td>
</tr>
<tr>
<td>x = 3;</td>
<td>x := 3;; (* x : int ref *)</td>
</tr>
<tr>
<td>y = x;</td>
<td>y := (!x);;</td>
</tr>
<tr>
<td>3 = x;</td>
<td>3 := x;; (* 3 : int; error *)</td>
</tr>
</tbody>
</table>

- 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 Needed For Side Effects

- Now that we can update memory, we have a 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
Examples – Semicolon

- **Definition**
  - e1 ; e2 (* evaluate e1, evaluate e2, return e2*)

- 1 ; 2 ;;
  - (* 2 – value of 2\(^{nd}\) expression is returned *)

- (1 + 2) ; 4 ;;
  - (* 4 – value of 2\(^{nd}\) expression is returned *)

- 1 + (2 ; 4) ;;
  - (* 5 – value of 2\(^{nd}\) expression is returned to 1 + *)

- 1 + 2 ; 4 ;;
  - (* 4 – because + has higher precedence than ; *)
;; versus ;

- ;; ends an expression in the top-level of OCaml
  - Use it to say: “Give me the value of this expression”
  - Not used in the body of a function
  - Not needed after each function definition
    - Though for now it won’t hurt if used there

- e1; e2 evaluates e1 and then e2, and returns e2

```
let print_both (s, t) = print_string s; print_string t;
    "Printed s and t"

let print_both = ("Colorless green ", "ideas sleep")
Prints "Colorless green ideas sleep", and returns "Printed s and t"
```
Grouping With Begin...End

- If you’re not sure about the scoping rules, use `begin...end` to group together statements with semicolons

```ocaml
let x = ref 0

let f () =
    begin
        print_string "hello";
        x := (!x) + 1
    end
```
The Trade-Off Of Side Effects

- Side effects are absolutely necessary
  - That’s usually why we run software! We want something to happen that we can observe

- They also make reasoning harder
  - Order of evaluation now matters
  - Calling the same function in different places may produce different results
  - **Aliasing** (two references to same object) is an issue
    - If we call a function with refs \( r_1 \) and \( r_2 \), it might do strange things if \( r_1 \) and \( r_2 \) are aliased
### Structural Vs. Physical Equality

- In OCaml, the `=` operator compares objects structurally
  - `[1;2;3] = [1;2;3]` (* true *)
  - `(1,2) = (1,2)` (* true *)
  - The `=` operator is used for pattern matching

- The `==` operator compares objects physically
  - `[1;2;3] == [1;2;3]` (* false *)

- Mostly you want to use the first one
  - But it’s a problem with cyclic data structures
Cyclic Data Structures Possible With Ref

- type 'a reflist = Nil | Cons of 'a * ('a reflist ref)
- let newcell x y = Cons(x,ref y);
- let updnext (Cons (_,r)) y = r := y;;
- let x = newcell 1 Nil;;
- updnext x x;; (* makes cycle *)
- x == x;; (* true *)
- x = x;; (* hangs *)
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
OCaml Programming Tips

- Compile your program often, after small changes
  - The OCaml parser often produces inscrutable error messages
  - It’s easier to figure out what’s wrong if you’ve only changed a few things since the last compile

- If you’re getting strange type error messages, add in type declarations
  - Try writing down types of arguments
  - For any expression e, can write (e:t) to assert e has type t
Watch out for precedence and function application

```ocaml
let mult x y = x*y

mult 2 2+3   (* returns 7 *)
(* parsed as (mult 2 2)+3 *)

mult 2 (2+3) (* returns 10 *)
```
All branches of a pattern match must return the same type

```ocaml
match x with
... -> -1 (* branch returns int *)
| ... -> () (* uh-oh, branch returns unit *)
| ... -> print_string "foo"
    (* also returns unit *)
```
You cannot assign to ordinary variables!

```ocaml
# let x = 42;;
val x : int = 42
# x = x + 1;;       (* this is a comparison *)
-: bool = false
# x := 3;;
Error: This expression has type int but is here used with type 'a ref
```
OCaml Programming Tips (cont.)

Again: You cannot assign to ordinary variables!

```ocaml
# let x = 42;;
val x : int = 42
# let f y = y + x;;    (* captures x = 42*)
val f : int -> int = <fun>
# let x = 0;;        (* shadows binding of x *)
val x : int = 0
# f 10;;             (* but f still refers to x=42 *)
- : int = 52
```