CMSC 330: Organization of Programming Languages

Object Oriented Programming with OCaml
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
Data Types

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

let area s =
 match s with
  Rect (wd, ln) -> wd *. ln
  Circle rad -> rad  *. rad *. 3.14

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

- **Rect** and **Circle** are *type constructors*- here a *shape* is either a **Rect** or a **Circle**
- Use pattern matching to *deconstruct* values, and do different things depending on constructor
Data Types, con't.

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

let x = [Rect (3.0, 4.0) ; Circle 3.0; Rect (10.0, 22.5)]

• What's the type of \( x \)?

\( x : \text{shape list} \)

• What's the type of \( x \)'s first element?

\( \text{shape} \)
Data Types (cont'd)

- The *arity* of a constructor is the number of arguments it takes – A constructor with no arguments is *nullary*

```
type optional_int = None | Some of int

let add_with_default a = function
  None -> a + 42
  | Some n -> a + n

add_with_default 3 None      (* 45 *)
add_with_default 3 (Some 4)  (* 7 *)
```

NOTES

```
# type int_option = None | Some of int;;
The OCaml compiler will warn of a function matching only Some ... values and neglecting the None value:
# let extract = function Some i -> i;;
Warning: this pattern-matching is not exhaustive. Here is an example of a value that is not matched:
None
val extract : int_option -> int = <fun>
This extract function then works as expected on Some ... values:
# extract (Some 3);;
- : int = 3
but causes a Match_failure exception to be raised at run-time if a None value is given, as none of the patterns in the pattern match of the extract function match this value:
# extract None;;
Exception: Match Failure ("", 5, -40).
```

- Constructors matching only Some...
Polymorphic Data Types

- This option type can work with any kind of data
  - In fact, this option type is built-in to OCaml

```ocaml
type 'a option =
  None
| Some of 'a

let add_with_default a = function
  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 this way:

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

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

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

– Note: Don’t have nice [1; 2; 3] syntax for this kind of list
Exercise: A Binary Tree Data Type

• Write type `bin_tree` for ordered binary trees over `int`

• Implement the following

  ```
  empty : bin_tree
  is_empty : bin_tree -> bool
  contains : int -> bin_tree -> bool
  insert : int -> bin_tree -> bin_tree
  equals : bin_tree -> bin_tree -> bool
  map : (int -> int) -> bin_tree -> bin_tree
  fold : ('a -> int -> 'a) -> 'a -> bin_tree -> 'a
  ```

• What about remove?
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

• For lots of sample modules, see the OCaml standard library
Creating a Module

```ocaml
module Shapes = 
  struct
    type shape =
      | Rect of float * float (* width * length *)
      | 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;;

unit_circle;; (* not defined *)
Shapes.unit_circle;;
Shapes.area (Shapes.Rect (3.0, 4.0));;
open Shapes;; (* import all names into current scope *)
unit_circle;; (* now defined *)
```
Modularity and Abstraction

• Another reason for creating a module is so we can *hide* details
  – For example, we can build a binary tree module, but we may not want to expose our exact representation of binary trees
  – This is also good software engineering practice
Module Signatures

Entry in signature

```
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 *)
```

Supply function types

Give type to module
Module Signatures (cont’d)

• The convention is for signatures to be all capital letters
  – 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
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 *)
**Note for Project 4**

- You may wish to put the “nfa” type definition in the signature to get better debugging information from Ocaml

- **REMOVE IT BEFORE YOU SUBMIT**
.ml and .mli files

• General structure: put the signature in a foo.mli file, the struct in a foo.ml file
  – Use the same names
  – Omit the sig...end and struct...end parts
  – The OCaml compiler will make a Foo module from these

• Similar to C file structure (.c/.h)

• Not implemented in project 4 for simplicity
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*!
  – You 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 – and there are no side effects!
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

```ocaml
define x = ref 3  (* x : int ref *)
define 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

• A variable's meaning depends on where it appears in an assignment
  – On the right-hand side, it’s an r-value, and it refers to the contents of the variable
  – On the left-hand side, it’s an l-value, and it refers to the location the variable is stored in
L-Values and R-Values (cont’d) (in C)

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

- In OCaml, an updatable location and the contents of the location have different types
  - The location has a \texttt{ref} type

```ocaml
let x = ref 0;;
let y = ref 0;;
x := 3;; (* x : int ref *)
y := (!x);;
3 := x;; (* 3 : int; error *)
```
Application: A ref in a Closure

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

```ocaml
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
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 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
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
```
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.printf "Caught %d\n" n
    | Failure s ->
        Printf.printf "Caught %s\n" s
Exceptions (cont’d)

- Exceptions are declared with `exception`
  - They may appear in the signature as well

- Exceptions may take arguments
  - Just like type constructors
  - May also be nullary

- Catch exceptions with `try...with...`
  - Pattern-matching can be used in `with`
  - If an exception is uncaught, the current function exits immediately and control transfers up the call chain until the exception is caught, or until it reaches the top level
OCaml

- Functions
- Types & polymorphism
- Lists & tuples
- Pattern matching
- Recursion & inductive definitions
- Map and fold
- Currying
- Object-oriented style
- Modules
- Side effects
- Exceptions
OCaml Language Choices

• Implicit or explicit variable 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