

CMSC 330: Organization of Programming Languages

OCaml – Data Types, Exceptions, Modules,

Data Types

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

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

CMSC 330

3

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

CMSC 330

2

Data Types (cont.)

```
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)
```

- ▶ Use pattern matching to **deconstruct** values
 - **s** is a **shape**
 - Do different things for **s** depending on its constructor

CMSC 330

4

Data Types (cont.)

```
type shape =  
  Rect of float * float (* width * length *)  
  | Circle of float      (* radius *)  
  
let l = [Rect (3.0, 4.0) ; Circle 3.0]
```

- ▶ What's the type of l?
shape list
- ▶ What's the type of l's first element?
shape

CMSC 330

5

Data Types Constructor

- ▶ Constructors must begin with uppercase letter
- ▶ 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
```

- ▶ Example
 - Arity of **None** = 0
 - Arity of **Some** = 1

CMSC 330

6

Polymorphic Data Types

```

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 *)

```

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

CMSC 330

7

Recursive Data Types

- We can build up lists this way

```

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

CMSC 330

8

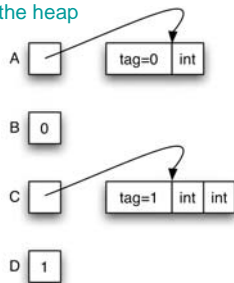
Data Type Representations

- 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

```



CMSC 330

9

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

```

CMSC 330

10

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.printf "Caught %d\n" n
  | Failure s ->
    Printf.printf "Caught %s\n" s

```

CMSC 330

11

OCaml 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 be nullary
- 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 reaches the top level

CMSC 330

12

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

```
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 ""? *)
```

CMSC 330

13

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 together associated
 - > Types, functions, and data
 - Avoid polluting the top-level with unnecessary stuff
- ▶ For lots of sample modules
 - See the OCaml standard library

CMSC 330

14

Modularity and Abstraction

- ▶ Another reason for creating a module
 - So we can **hide** details
 - Example
 - > Build a binary tree module
 - > Hide 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)

CMSC 330

15

Modularity

- ▶ Definition
 - Extent to which a computer program is composed of **separate parts**
 - Higher degree of modularity is better
- ▶ Modular programming
 - Programming techniques that increase modularity
 - > Interface vs. implementation
- ▶ Modular programming languages
 - Explicit support for modules
 - Ada, Fortran, ML, Modula-2, Python, Ruby, **OCaml**

CMSC 330

16

Creating a Module in 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;;
```

CMSC 330

17

Creating a Module in OCaml (cont.)

```
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 *)
```

CMSC 330

18

Module Signatures

Entry in signature Supply function types

```
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 *)
```

CMSC 330

19

Module Signatures (cont.)

- ▶ Convention
 - 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
 - You'll notice this is what OCaml gives you if you just type in a module with no signature at the top-level

CMSC 330

20

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

CMSC 330

21

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?

CMSC 330

22

.ml and .mli files

- ▶ 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

CMSC 330

23

Example – OCaml Module Signatures

```
shapes.mli
type shape
val area : shape -> float
val unit_circle : shape
val make_circle : float -> shape
val make_rect : float -> float -> shape
```

```
shapes.ml
type shape =
  Rect of ...
...
let make_circle r = Circle r
let make_rect x y = Rect (x, y)
```

```
% ocamlc shapes.mli # produces shapes.cmi
% ocamlc shapes.ml # produces shapes.cmo
ocaml
# #load "shapes.cmo" (* load Shapes module *)
```

CMSC 330

24

Functors

- ▶ Modules can take other modules as arguments
 - Such a module is called a **functor**
 - You're mostly on your own if you want to use these
- ▶ Example: **Set** in standard library

```
module type OrderedType = sig
  type t
  val compare : t -> t -> int
end

module Make(Ord: OrderedType) =
  struct ... end

module StringSet = Set.Make(String);;
(* works because String has type t,
  implements compare *)
```

CMSC 330

25

Module 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

CMSC 330

26

Module 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



CMSC 330

27

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
```

CMSC 330

28

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** !
 - Don't care whether data is shared in memory
 - > Aliasing is irrelevant
 - Provides strong support for compositional reasoning and abstraction
 - > Example: Calling a function f with argument x always produces the same result
 - But could take (much) more memory & time to execute

CMSC 330

29

Imperative OCaml

- ▶ There are three basic operations on memory
 - 1) **ref** : 'a -> 'a ref
 - > Allocate an updatable reference
 - 2) **!** : 'a ref -> 'a
 - > Read the value stored in reference
 - 3) **:=** : 'a ref -> 'a -> unit
 - > Write to a reference

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

CMSC 330

30

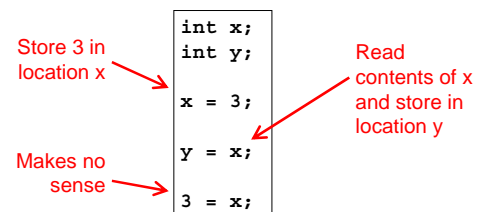
Comparison to L- and R-values

- 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

CMSC 330

31

L-Values and R-Values in C (cont.)



- Notice that x, y, 3 all have the **same** type: `int`

CMSC 330

32

Comparison to OCaml

<pre>int x; C int y; x = 3; y = x; 3 = x;</pre>	<pre>let x = ref 0;; OCaml let y = ref 0;; x := 3;; (* x : int ref *) y := (!x);; 3 := x;; (* 3 : int; error *)</pre>
--	---

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

CMSC 330

33

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
```

CMSC 330

34

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

CMSC 330

35

Grouping with begin...end

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

```
let x = ref 0

let f () =
  begin
    print_string "hello";
    x := (!x) + 1
  end
```

CMSC 330

36

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
- ▶ But...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 `r1` and `r2`, it might do strange things if `r1` and `r2` are aliased

CMSC 330

37

OCaml Language Choices

- ▶ Implicit or explicit declarations?
 - Explicit – variables must be introduced with `let` before use
 - But you don't need to specify type of variable
- ▶ Static or dynamic types?
 - Static – but without type declarations
 - OCaml does **type inference** to figure out types for you
 - Advantage – less work to write programs
 - Disadvantages – easier to make mistakes, harder to find errors

CMSC 330

38