

# CMSC 330: Organization of Programming Languages

## Modules

# Quiz 1

- What's the largest program you have ever worked on, by yourself or as part of a team?
  - A. 100-1,000 LoC
  - B. 1,000-10,000 LoC
  - C. 10,000-100,000 LoC
  - D. 100,000 LoC or bigger

# Scale

- Windows Vista: 50 million LOC
- Mac OS X Tiger: 86 million LOC
- Google: 2 billion LOC

# Modular Design

- When a program is small enough, we can keep all the details of the program in one file.
- Real application programs are simply too large and complex to hold all their details in our heads.



# Modular Programming

- The code is composed of many different code modules that are developed separately.
- **Modules** group associated types, functions, and data together.
- For lots of sample modules, see the OCaml standard library, e.g., **List**, **Str**, etc.

# Creating A Module In OCaml

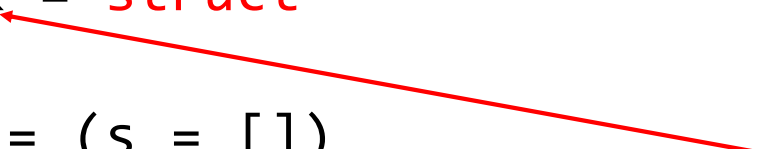
Modules in OCaml are implemented by module definitions that have the following syntax:

```
module ModuleName = struct  
    (* definitions *)  
end
```

# Module Example: ListStack

```
module ListStack = struct
  let empty = []
  let is_empty s = (s = [])
  let push x s = x :: s
  let peek = function
    | [] -> failwith "Empty"
    | x::_ -> x
  let pop = function
    | [] -> failwith "Empty"
    | _::xs -> xs
end
```

Module names  
must begin with an  
uppercase letter.



# Scope

After a module has been defined, you can access the names within it using the `.` operator. For example:

```
# module M = struct
  let x = 42
end;;
```

```
# M.x;;
- : int = 42
```



# Scope

Opening a module brings all the definitions of a module into the current scope. For example

```
# module M = struct
  let x = 42
end;;
```

```
# x;;
Error: Unbound value x
# open M;;
# x;;
- : int = 42
```

# Opening Multiple Modules

If two modules both define the same name, and you open both of them, any names defined later shadow names defined earlier.

```
module M = struct
  let x = 42
end
module N = struct
  let x = "CMSC330"
end
open M
open N
```

What is x?      **CMSC330**

# Opening Multiple Modules

To not to pollute the current scope, you can locally open a module

## Without opening List

```
let f x =  
  let y = List.filter ((>) 0) x in ...
```

## Opening List locally

```
let f x = let open List in  
  let y = filter ((>) 0) x in
```

[filter] is now bound to [List.filter]

# Module Signatures

- Signatures are **interfaces** for structures.
- A signature specifies which components of a structure are accessible from the outside, and with which **type**.
- It can be used to **hide** some components of a structure (e.g. local function definitions)

```
module type F00 =  
  sig  
    val add : int -> int -> int  
  end;;  
module Foo : F00 =  
  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**
  - This is what OCaml gives you if you just type in a module with no signature at the top-level

# Implementing a Signature

```
module type Sig =  
  sig  
    val f : int -> int  
  end
```

```
module M1 : Sig = struct  
  let f x = x+1  
end
```

← Exact the type specified by Sig: `int->int`

```
module M2 : Sig = struct  
  let f x = x  
end
```

← Type: `'a -> 'a`, safe to use  
it for `int->int`

```
# M2.f;;  
- : int -> int
```

# ListStack

```
module type Stack = sig
  type 'a stack = 'a list
  val empty      : 'a stack
  val is_empty   : 'a stack -> bool
  val push       : 'a -> 'a stack -> 'a stack
  val peek       : 'a stack -> 'a
  val pop        : 'a stack -> 'a stack
end
```

# ListStack

```
module type Stack = sig
  type 'a stack = 'a list
  val empty      : 'a stack
  val is_empty   : 'a stack -> bool
  val push       : 'a -> 'a stack -> 'a stack
  val peek       : 'a stack -> 'a
  val pop        : 'a stack -> 'a stack
end
```

```
module ListStack: Stack = struct
  type 'a stack = 'a list
  let empty = []
  let is_empty s = (s = [])

  let push x s = x :: s

  let peek = function
    | [] -> failwith "Empty"
    | x::_ -> x

  let pop = function
    | [] -> failwith "Empty"
    | _::xs -> xs
end
```



# ListStack Example

```
let t = ListStack.empty;;  
val t : 'a ListStack.stack = []  
  
# let t2 = ListStack.push 10 t;;  
val t2 : int ListStack.stack = [10]  
  
# let t3 = ListStack.push 20 t2;;  
val t3 : int ListStack.stack = [20; 10]  
  
# ListStack.peek t3;;  
- : int = 20  
  
# let t4 = ListStack.pop t3;;  
val t4 : int ListStack.stack = [10]
```

Implementation is visible to user

# Abstract Types

- The type 'a stack below is *abstract*
- A module that implements `Stack`
  - must specify concrete types for the abstract type 'a stack
  - define all the names declared in the signature.

```
module type Stack = sig
  type 'a stack
  val empty      : 'a stack
  val is_empty   : 'a stack -> bool
  val push       : 'a -> 'a stack -> 'a stack
  val peek       : 'a stack -> 'a
  val pop        : 'a stack -> 'a stack
end
```

Type is abstract

# Stack Signature: Abstract Stack

```
module type Stack = sig
  type 'a stack
  val empty      : 'a stack
  val is_empty   : 'a stack -> bool
  val push       : 'a -> 'a stack -> 'a stack
  val peek       : 'a stack -> 'a
  val pop        : 'a stack -> 'a stack
end
```

# ListStack

```
module type Stack = sig
  type 'a stack
  val empty      : 'a stack
  val is_empty   : 'a stack -> bool
  val push       : 'a -> 'a stack -> 'a stack
  val peek       : 'a stack -> 'a
  val pop        : 'a stack -> 'a stack
end
```

```
module ListStack: Stack = struct
  type 'a stack = 'a list
  let empty = []
  let is_empty s = (s = [])

  let push x s = x :: s

  let peek = function
    | [] -> failwith "Empty"
    | x::_ -> x

  let pop = function
    | [] -> failwith "Empty"
    | _::xs -> xs
end
```

# Abstract ListStack Example

```
let t = ListStack.empty;;  
val t : 'a ListStack.stack = <abstr>  
  
# let t2 = ListStack.push 10 t;;  
val t2 : int ListStack.stack = <abstr>  
  
# let t3 = ListStack.push 20 t2;;  
val t3 : int ListStack.stack = <abstr>  
  
# ListStack.peek t3;;  
- : int = 20  
  
# let t4 = ListStack.pop t3;;  
val t4 : int ListStack.stack = <abstr>
```

Implementation is NOT visible



# VariantStack

```
module type Stack = sig
  type 'a stack
  val empty      : 'a stack
  val is_empty   : 'a stack -> bool
  val push       : 'a -> 'a stack -> 'a stack
  val peek       : 'a stack -> 'a
  val pop        : 'a stack -> 'a stack
end
```

```
module MyStack : Stack = struct
  type 'a stack =
    | Empty
    | Entry of 'a * 'a stack

  let empty = Empty
  let is_empty s = s = Empty
  let push x s = Entry (x, s)
  let peek = function
    | Empty -> failwith "Empty"
    | Entry(x,_) -> x
  let pop = function
    | Empty -> failwith "Empty"
    | Entry(_,s) -> s
end
```

# Functional Data Structures

- Immutable, Persistent data structures
- Updating the data structure with one of its operations **does not change** the existing version of the data structure but instead produces a new version

```
# open ListStack;;  
let s = empty;;  
let s2 = push 10 s  
let s3 = push 20 s2
```

Push to **s2** does not destroy **s2**, but creates new **s3**

# Loading Compiled Modules into Toplevel

- Compile: `ocamlbuild main.byte`
- Produced file are inside: `_build`

```
#directory "_build";;  
#load "liststack.cmo";;  
# open Liststack;;  
  
#show ListStack;;  
module ListStack : Stack.Stack  
  
# ListStack.empty;;  
- : 'a Liststack.ListStack.stack = <abstr>
```



# Includes

Suppose we wanted to add a function `max` to the `List` module that returns the max item from the list

```
module MyList = struct include List
  let max lst = ... (* implementation *)
end;;
```

```
#show MyList;;
```

- All the list function and max

# Functor


- Functors are “functions” from modules to modules.
- Functors let you create parameterized modules and then provide other modules as parameters to get a specific implementation.

```
module Set :  
  sig  
    module type OrderedType = sig type t val compare : t -> t -> int end  
    module type S =  
      sig  
        type elt  
        type t  
        ....  
      end  
    module Make : functor (Ord : OrderedType) -> sig ... end  
  end
```

Items in Set must  
be comparable



Make takes a module as an  
argument



# Functor: Set

```
module Set :
  sig
    module type OrderedType = sig type t val compare : t -> t -> int end
    module type S =
      sig
        type elt
        type t
        ....
      end
    module Make : functor (Ord : OrderedType) -> sig ... end
  end
```

Items in Set must  
be comparable



```
module StrSet = Set.Make(String);;
# let s = StrSet.empty;;
val s : StrSet.t = <abstr>
# let s = StrSet.add "hello" s;;
val s : StrSet.t = <abstr>
# let s = StrSet.add "world" s;;
val s : StrSet.t = <abstr>
```

# Functor: Example

```
module type Printable =
  sig
    type t
    val to_str: t-> string
  end

module Printer(E:Printable) =
  struct
    let print x =
      Printf.printf "%s\n" (E.to_str x)
    end
  end

module IntPrinter = Printer(Int);;
module FloatPrinter = Printer(Float);;
let main () =
  IntPrinter.print 88;
  FloatPrinter.print 2.5

let () = main ()
```

```
module Float =
  struct
    type t = float
    let to_str = string_of_float
  end

module Int =
  struct
    type t = int
    let to_str = string_of_int
  end
```

# Functor: Example

```
module type Printable =
  sig
    type t
    val to_str: t-> string
  end

module Printer(E:Printable) =
  struct
    let print x =
      Printf.printf "%s\n" (E.to_str x)
    end
  end

module IntPrinter = Printer(Int);;
module FloatPrinter = Printer(Float);;
let main () =
  IntPrinter.print 88;
  FloatPrinter.print 2.5

let () = main ()
```


```
module Float =
  struct
    type t = float
    let to_str = string_of_float
  end

module Int =
  struct
    type t = int
    let to_str = string_of_int
  end
```

# Map Module

**A map for integer keys.** To create a map, we have to pass a structure into `Map.Make`, and that structure has to define a type `t` and compare function.

```
# module IntMap = Map.Make(  
  struct  
    type t = int  
    let compare = Pervasives.compare  
  end  
);;  
# open IntMap;;  
# let m1 = add 100 "Alice" empty;;  
-val m1 : string t = <abstr>  
# find 100 m1;;  
- : string = "Alice"
```



pass an anonymous structure into the functor

# Summary

- abstract type
  - abstraction
  - code reuse
  - encapsulation
  - functional data structure
  - functor
  - implementation
  - include
  - information hiding
  - interface
  - local reasoning
  - Maintainability
- modular programming
  - modularity
  - module
  - module type
  - namespace
  - open
  - parameterized structure
  - persistant data structure
  - scope
  - signature
  - signature matching
  - structure