# CS 3110

# Functors

Prof. Clarkson
Fall 2016

Today's music: "Uptown Funk"
by Mark Ronson feat. Bruno Mars

# Review

**Previously in 3110:**

- modules, structures, signatures, abstract types
- aspects of modularity:  namespaces, abstraction

**Today:**

- higher-order usage of modules: functors
- another aspect:  code reuse

# Review

**Structure:** a group of related *definitions*

```
struct
  type 'a t = 'a list
  let push x s = x::s
end
```

**Signature:** a group of related *declarations* aka *type specifications*

```
sig
  type 'a t
  val push : 'a -> 'a t -> 'a t
end
```

**Signatures are the types of structures**

# Review

**Module and module types:** bind structures and signatures to names

```
module type Stack = sig
  type 'a t
  val push : 'a -> 'a t -> 'a t
end

module ListStack : Stack = struct
  type 'a t = 'a list
  let push x s = x::s
end
```

# Review

**Encapsulation:** hide parts of module from clients

```
module type Stack = sig
  type 'a t
  val push : 'a -> 'a t -> 'a t
end

module ListStack : Stack = struct
  type 'a t = 'a list
  let push x s = x::s
end
```

# Review

**Encapsulation:** hide parts of module from clients

```
module type Stack = sig
  type 'a t
  val push : 'a -> 'a t -> 'a t
end

module ListStack : Stack = struct
  type 'a t = 'a list
  let push x s = x::s
end
```

type constructor **t** is *abstract*: clients of this signature know the type exists but not what it is

# Review

**Encapsulation:** hide parts of module from clients

```
module type Stack = sig
  type 'a t
  val push : 'a -> 'a t -> 'a t
end

module ListStack : Stack = struct
  type 'a t = 'a list
  let push x s = x::s
end
```

module is *sealed*: all definitions in it except those given in signature **`Stack`** are hidden from clients

# Question

Consider this code:

```
module type Stack =
sig
  type 'a t
  val empty : 'a t
  val push  : 'a -> 'a t -> 'a t
end

module ListStack : Stack =
struct
  type 'a t     = 'a list
  let empty     = []
  let push x s = x::s
end
```

Which of the following expressions will type check?

A. `Stack.empty`

B. `ListStack.push 1 []`

C. `fun (s:ListStack) -> ListStack.push 1 s`

D. All of the above

E. None of the above

# Question

Consider this code:

```
module type Stack =
sig
  type 'a t
  val empty : 'a t
  val push  : 'a -> 'a t -> 'a t
end

module ListStack : Stack =
struct
  type 'a t     = 'a list
  let empty     = []
  let push x s = x::s
end
```

Which of the following expressions will type check?

A. `Stack.empty`

B. `ListStack.push 1 []`

C. `fun (s:ListStack) -> ListStack.push 1 s`

D. All of the above

E. None of the above

# INCLUDES

# Include a signature

**Interface inheritance:** reuse code from other signatures

```
module type Ring = sig
  type t
  val zero : t
  val one  : t
  val add  : t -> t -> t
  val mult : t -> t -> t
  val neg  : t -> t
end

module type Field = sig
  include Ring
  val div : t -> t -> t
end
```

# Include a module

**Implementation inheritance:** reuse code from other structures

```
module FloatRing = struct
  type t = float
  let zero = 0.
  let one  = 1.
  let add  = (+.)
  let mult = ( *. )
  let neg  = (~-.)
end

module FloatField = struct
  include FloatRing
  let div = (/.)
end
```

# Code reuse from includes

- Implementer of one module can rely on code from another module:  no need to copy code
- Solves a similar problem as class inheritance in Java
  - but without creating subtype relationships
  - decouples inheritance from subtyping

# FUNCTORS

(funk you up?)

https://www.youtube.com/watch?v=Au56Ah92Ulk

# Structures are higher order

- You can write "functions" that manipulate structures
  - take structures as input, return structure as output
  - syntax is a bit different than functions we've seen so far
- These "functions" are called *functors*
  - One of the most advanced features in OCaml
  - A *higher-order module system*
  - Time for some funky higher-order fun...

# Simple functor

```
module type X = sig val x : int end

module IncX (M : X) = struct
  let x = M.x + 1
end

module A = struct let x = 0 end
module B = IncX(A) (* B.x is 1 *)
module C = IncX(B) (* C.x is 2 *)
```

functor: takes structure of type **X** as input, uses **M** as the name of that structure in its own body, returns a structure

# Alternative functor syntax

Instead of:

```
module IncX (M : X) = struct
  let x = M.x + 1
end
```

Could write:

```
module IncX = functor (M : X) -> struct
  let x = M.x + 1
end
```

Parallels syntax for anonymous functions

# STANDARD LIBRARY: MAP

# Map

```
(* maps over totally ordered keys *)
module Map : sig
  (* the input type of Make *)
  module type OrderedType = sig type t ... end

  (* the output type of Make *)
  module type S = sig type key ... end

  (* functor that makes a module *)
  module Make (Ord : OrderedType)
    : S with type key = Ord.t
end
```

# Map

```
module type S =
sig
  type key
  type 'a t
  val empty : 'a t
  val mem : key -> 'a t -> bool
  val add : key -> 'a -> 'a t -> 'a t
  ...
end
```

# Map

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

Must return 0 if equal,
-1 if first argument is lesser,
1 if second argument is lesser

# Map

```
(* maps over totally ordered keys *)
module Map : sig
  (* the input type of Make *)
  module type OrderedType = sig type t ... end

  (* the output type of Make *)
  module type S = sig type key ... end

  (* functor that makes a module *)
  module Make (Ord : OrderedType)
    : S with type key = Ord.t
end
```

sharing constraint: the output of Make additionally knows that the **`key`** type and the **`OrderedType`** are the same

# Map

```
module type S with type key = Ord.t =
sig
  type key = Ord.t
  type 'a t
  val empty : 'a t
  val mem : key -> 'a t -> bool
  val add : key -> 'a -> 'a t -> 'a t
  ...
end
```

# Map

Why does this work? The String module already provides a type **`t`** and a function **`compare`**.

```
# module StringMap = Map.Make(String) ;;
module StringMap : sig
  type key = string
  ...
end

# let sm = StringMap.(
    empty |> add "Alice" 4.0 |> add "Bob" 3.7)

# StringMap.find "Bob" sm
- : float =  3.7
```

# Map

- What if we wanted a map with keys that are int's?
- There's no standard library module that gives us a type **`t`** and function **`compare`** for ints.
- So we build our own...

```
module Int = struct
  type t = int
  let compare = Pervasives.compare
end

module IntMap = Map.Make(Int)
let im = IntMap.(
  empty |> add 1 "one" |> add 2 "two")
```

# Question

Why didn't we write:

```
module Int : Map.OrderedType = struct
  type t = int
  let compare = Pervasives.compare
end
module IntMap = Map.Make(Int)
```

A. Actually we should have; without it the code before doesn't compile
B. We didn't need to; the compiler infers that module type for us
C. It's incorrect; that's not a valid module type for Int
D. None of the above; I'm funked up

# Question

Why didn't we write:

```
module Int : Map.OrderedType = struct
  type t = int
  let compare = Pervasives.compare
end
module IntMap = Map.Make(Int)
```

A. Actually we should have; without it the code before doesn't compile
B. We didn't need to; the compiler infers that module type for us
C. It's incorrect; that's not a valid module type for Int
**D. None of the above;** *I'm funked up*

# Answer

```
# module IntMap  = Map.Make(Int);;
module IntMap : sig
  type key = Int.t  (* t is abstract! *)
  ...
end

# IntMap.(empty |> add 1 "one");;
Error: This expression has type int but
an expression was expected of type key
```

# Map

- What if we wanted a map over records that sorts in a custom order?
- Again, build our own module...

```
type name = {first:string; last:string}

module Name = struct
  type t = name
  let compare {first=first1;last=last1}
              {first=first2;last=last2} =
    match Pervasives.compare last1 last2 with
    | 0 -> Pervasives.compare first1 first2
    | c -> c
end

module NameMap = Map.Make(Name)
```

Sort by last name then by first name

# Map

```
let k1 =
  {last="Kardashian"; first="Kourtney"}
let k2 =
  {last="Kardashian"; first="Kimberly"}
let k3 =
  {last="Kardashian"; first="Khloe"}
let k4 =
  {last="West"; first="Kanye"}

let nm = NameMap.(
  empty |> add k1 1979 |> add k2 1980
        |> add k3 1984 |> add k4 1977)
```

# Map

```
let print_entry {first;last} v
  = print_string (first ^ " " ^ last ^ ": ");
    print_int v;
    print_newline ()

let () = NameMap.iter print_entry nm
```

Khloe Kardashian: 1984
Kimberly Kardashian: 1980
Kourtney Kardashian: 1979
Kanye West: 1977

# Code reuse with Map

- The Map implementer built all the tricky parts of maps:  adding keys and values, iterating over them, etc.
- As clients, all we have to provide is a description of our keys and how to sort them; then we get to reuse everything the implementer already built
- Solves a similar problem as Java does with interfaces +subtyping:  see Java's TreeMap constructor that takes a Comparator
- OCaml's Set module is quite similar to Map in its *functorial interface*

# Recap

- Functors are "functions" from structures to structures
- Functors make the OCaml module system higher-order
- Functors enable code reuse

# Upcoming events

- [Wednesday] A2 due

*This is higher-order funk.*

**THIS IS 3110**