Functors

Prof. Clarkson
Fall 2015

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

Previously in 3110:
• modules, structures, signatures, abstract types

Today:
• higher-order usage of modules: functors
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 type specifications

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

Signatures are the types of structures
Module and module types: bind structures and signatures to names

```ocaml
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
```
Encapsulation: hide parts of module from clients

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

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

module ListStack : Stack = struct
  type 'a t =
  let push x s =
  end
module is sealed: all definitions in it except those given in signature Stack are hidden from clients
```
Consider this code:

```ocaml
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:

```ocaml
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
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
end

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

Implementation inheritance: reuse code from other structures

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

module FloatField = struct
  include FloatRing
  let neg = (~~.)
  let div = (/.)
end
```
Timeout: Assignments

• Still individual
• Be especially careful in office hours
• If you take code from somewhere, cite it
• If you get an idea from someone, credit them

Rules of thumb:
• Never look at any other student's assignment code.
• Never have another student's code in your possession, in any portion or form whatsoever.
• Never share your assignment code with other students.
FUNCTORS
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)
module C = IncX(B)
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)
module C = IncX(B)
Alternative functor syntax

Instead of:

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

Could write:

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

Parallels syntax for anonymous functions
Examples of functors

A. Testing implementations of an interface
B. Parameterizing a data structure on another data structure
C. Functions for free
D. Standard library `Map` functor

Examples are going to get too big for slides

(compelling examples of programming-in-the-large require, well, larger code)

...see the accompanying code
Testing implementations

Problem: how to test multiple implementations of the same interface without having to rewrite test code for each implementation

Solution: a functor that generates test code

See queues.ml
Parameterized implementation

**Problem:** how to parameterize the implementation of one data structure on the implementation of another

**Solution:** a functor that generates the new data structure out of the old

See `matrices.ml`
Functions for free

Problem: how to derive a family of convenience functions out of one underlying function without having to rewrite that family every time

Solution: a functor that generates the functions

See iterable1.ml and iterable2.ml
Standard library Map

**Problem:** how to use the standard library Map module to get dictionaries

**Solution:** use the functorial interface it provides

See `maps.ml`

*Set* module is also functorial

*Hashtbl* is too, but is imperative: don't use for now
Recap

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

• [Thursday] A2 due (soft deadline)

This is higher-order fun.

THIS IS 3110