

# Modules

## Concepts of Programming Languages Lecture 10

# Outline

Do some practice problems in preparation for the midterm

Discuss modules a way to make abstract and reusable code

# Practice Problem

$\emptyset \vdash \text{let } x = 2 + 2 \text{ in } (\text{fun } y \rightarrow x = y) \ x : \text{bool}$

$\text{let } x = 2 + 2 \text{ in } (\text{fun } y \rightarrow x = y) \ x \Downarrow \top$

*Give derivations of each of the above judgments*

# Another Practice Problem

Implement the function

```
val filter_op : ('a -> 'b -> (bool * 'c))  
               -> ('a * 'b) list -> 'c list
```

where `filter_op f l` is the output of `f` on those elements of `l` which satisfy `f`

# Modular Programming

# High Level

# High Level

Modules attempt to capture multiple programming patterns with a single construct:

# High Level

Modules attempt to capture multiple programming patterns with a single construct:

**1. Namespaces:** a way of separate coding into logical units

```
module Interpreter = struct
  let type_check = ...
  let eval = ...
end
```

(1)



# High Level

Modules attempt to capture multiple programming patterns with a single construct:

- 1. Namespaces:** a way of separate coding into logical units
- 2. Abstraction/Encapsulation:** a way of abstracting away implementation details and organizing core functionality (e.g., of a data structure)

```
module Interpreter = struct
  let type_check = ...
  let eval = ...
end
```

(1)

```
module Stack = struct
  type 'a t = 'a list
  let push x s = x :: s
  let pop s = match s with
    | [] -> None
    | x :: xs -> Some (x, xs)
end
```

(2)

# High Level

Modules attempt to capture multiple programming patterns with a single construct:

- 1. Namespaces:** a way of separate coding into logical units
- 2. Abstraction/Encapsulation:** a way of abstracting away implementation details and organizing core functionality (e.g., of a data structure)
- 3. Code Reuse:** a way to write general code that can be instantiated in different settings

```
module Interpreter = struct
  let type_check = ...
  let eval = ...
end
```

(1)

```
module Stack = struct
  type 'a t = 'a list
  let push x s = x :: s
  let pop s = match s with
    | [] -> None
    | x :: xs -> Some (x, xs)
end
```

(2)

```
module VarSet = Set.Make(String)
module Context = Map.Make(String)
```

(3)

# Structures

```
module Foo = struct
  let double (x : int) : int = x + x

  let is_whitespace (c : char) =
    List.mem c [' '; '\n'; '\t'; '\r']

  let version = 225

  exception MyException
end
```

# Structures

A **structure** is a collection of definitions used to define a **module**

```
module Foo = struct
  let double (x : int) : int = x + x

  let is_whitespace (c : char) =
    List.mem c [' '; '\n'; '\t'; '\r']

  let version = 225

  exception MyException
end
```

# Structures

A **structure** is a collection of definitions used to define a **module**

Structures are *not* first-class values, we *must* use the **module** keyword when defining a structure

```
module Foo = struct
  let double (x : int) : int = x + x

  let is_whitespace (c : char) =
    List.mem c [' '; '\n'; '\t'; '\r']

  let version = 225

  exception MyException
end
```

# Structures

A **structure** is a collection of definitions used to define a **module**

Structures are *not* first-class values, we *must* use the **module** keyword when defining a structure

We can put anything in a structure that we can put in a standalone .ml file (and vice versa, more on this later)

```
module Foo = struct
  let double (x : int) : int = x + x

  let is_whitespace (c : char) =
    List.mem c [' '; '\n'; '\t'; '\r']

  let version = 225

  exception MyException
end
```

# Signatures

```
module type F00 =  
  sig  
    val double : int -> int  
    val is_whitespace : char -> bool  
    val version : int  
    exception MyException  
  end
```

# Signatures

A **signature** is a collection of *specifications*

```
module type F00 =  
  sig  
    val double : int -> int  
    val is_whitespace : char -> bool  
    val version : int  
    exception MyException  
  end
```



# Signatures

A **signature** is a collection of *specifications*

A specification is a name together with a type

```
module type F00 =  
  sig  
    val double : int -> int  
    val is_whitespace : char -> bool  
    val version : int  
    exception MyException  
  end
```

# Signatures

A **signature** is a collection of *specifications*

A specification is a name together with a type

A signature defines a **module type**

```
module type F00 =  
  sig  
    val double : int -> int  
    val is_whitespace : char -> bool  
    val version : int  
    exception MyException  
  end
```

# Signatures

A **signature** is a collection of *specifications*

A specification is a name together with a type

A signature defines a **module type**

A module **implements** a signature if it's defined as a structure which has the values required by the signature

```
module type F00 =  
  sig  
    val double : int -> int  
    val is_whitespace : char -> bool  
    val version : int  
    exception MyException  
  end
```

# General Syntax

```
module ModuleName : SIG_NAME = struct
  val val_name1 : ty
  val val_name2 : ty
  ...
end

module L = List
module S = String
```

# General Syntax

```
module ModuleName : SIG_NAME = struct      module L = List
  val val_name1 : ty                        module S = String
  val val_name2 : ty
  ...
end
```

Module names are usually CamelCase and module types in SCREAMING\_SNAKE

# General Syntax

```
module ModuleName : SIG_NAME = struct      module L = List
  val val_name1 : ty                        module S = String
  val val_name2 : ty
  ...
end
```

Module names are usually CamelCase and module types in SCREAMING\_SNAKE

The inner part of the **struct** is anything we could write in a `.ml` file

# General Syntax

```
module ModuleName : SIG_NAME = struct      module L = List
  val val_name1 : ty                        module S = String
  val val_name2 : ty
  ...
end
```

Module names are usually CamelCase and module types in SCREAMING\_SNAKE

The inner part of the **struct** is anything we could write in a `.ml` file

The **module** keyword is like the **let** keyword except that the RHS of the "`=`" must be a structure *or another module*

# General Syntax

```
module ModuleName : SIG_NAME = struct      module L = List
  val val_name1 : ty                        module S = String
  val val_name2 : ty
  ...
end
```

Module names are usually CamelCase and module types in SCREAMING\_SNAKE

The inner part of the **struct** is anything we could write in a `.ml` file

The **module** keyword is like the **let** keyword except that the RHS of the `"="` must be a structure *or another module*

**Trick:** We can write shorthand names for module names we use frequently



# Signature Inference and Interface Files

```
let double (x : int) : int = x + x

let is_whitespace (c : char) =
  List.mem c [' '; '\n'; '\t'; '\r']

let version = 225

exception MyException
```

**foo.ml**

```
val double : int -> int

val is_whitespace : char -> bool

val version : int

exception MyException
```

**foo.mli**

# Signature Inference and Interface Files

```
let double (x : int) : int = x + x

let is_whitespace (c : char) =
  List.mem c [' '; '\n'; '\t'; '\r']

let version = 225

exception MyException
```

**foo.ml**

```
val double : int -> int

val is_whitespace : char -> bool

val version : int

exception MyException
```

**foo.mli**

In most cases, OCaml infers the signature of a given module (the annotation is optional)

# Signature Inference and Interface Files

```
let double (x : int) : int = x + x

let is_whitespace (c : char) =
  List.mem c [' '; '\n'; '\t'; '\r']

let version = 225

exception MyException
```

**foo.ml**

```
val double : int -> int

val is_whitespace : char -> bool

val version : int

exception MyException
```

**foo.mli**

In most cases, OCaml infers the signature of a given module (the annotation is optional)

In fact, we've been defining modules the entire time: *every file defines a module, whose name is the same as the filename (capitalized)*

# Signature Inference and Interface Files

```
let double (x : int) : int = x + x

let is_whitespace (c : char) =
  List.mem c [' '; '\n'; '\t'; '\r']

let version = 225

exception MyException
```

**foo.ml**

```
val double : int -> int

val is_whitespace : char -> bool

val version : int

exception MyException
```

**foo.mli**

In most cases, OCaml infers the signature of a given module (the annotation is optional)

In fact, we've been defining modules the entire time: *every file defines a module, whose name is the same as the filename (capitalized)*

We can make signatures of files explicit with **.mli** files

# Working with Modules

```
module type F00 =  
  sig  
    val double : int -> int  
    val is_whitespace : char -> bool  
    val version : int  
    exception MyException  
  end
```

```
let check c =  
  if Foo.version > 300 && Foo.is_whitespace c  
  then "okay"  
  else "not okay"
```

Once a module is defined, we can use values defined therein by **dot notation**

(This should feel somewhat familiar, again, we've been working with modules this whole time)

# Opening Modules

```
open Foo
```

```
let check c =  
  if version > 300 && is_whitespace c  
  then "okay"  
  else "not okay"
```

# Opening Modules

```
open Foo
```

```
let check c =  
  if version > 300 && is_whitespace c  
  then "okay"  
  else "not okay"
```

We can bring all definitions in a module into scope with the **open** keyword

# Opening Modules

```
open Foo
```

```
let check c =  
  if version > 300 && is_whitespace c  
  then "okay"  
  else "not okay"
```

We can bring all definitions in a module into scope with the **open** keyword

**Caution:** Do this sparingly, it's like **import \*** except worse because there's no overloading in OCaml



# Opening Modules

```
open Foo
```

```
let check c =  
  if version > 300 && is_whitespace c  
  then "okay"  
  else "not okay"
```

We can bring all definitions in a module into scope with the **open** keyword

**Caution:** Do this sparingly, it's like **import \*** except worse because there's no overloading in OCaml

*If there are multiple definition of the function, the most recent open prevails*

# .(...) Syntax

```
let check c =  
  Foo.(if version > 300 && is_whitespace c  
    then "okay"  
    else "not okay")
```

It's possible to parenthesize expressions after the dot notation!

This will evaluate the expression *as if* the module was opened

# demo

(smart/short constructors)

# Encapsulation

# Modules and Structural Subtyping

A structure needs to implement *every* value in a signature, but it can also implement *extra* values

If we use a module constraint on a definition, we *cannot* access those extra values

```
module type BAR = sig
  val bar : int
end
```

```
module Foo : BAR = struct
  let foo = "twenty two"
  let bar = 22
end
```

```
let _ = assert (Foo.bar = 22)
(* let _ = assert (Foo.foo =
"twenty two") *)
```

# Modules and Structural Subtyping

A module type **S** is a **subtype** of **T**  
if **S** is a *superset* of **T**

*Said another way:* anything that  
implements **S** also implements **T**

*Note:* We can write **(Mod : MOD\_TY)**  
to "type-check" the module **Mod**

```
module type S = sig
  val a : int
  val b : int
end
```

```
module type T = sig
  val b : int
end
```

```
module ImplS : S = struct
  let a = 0
  let b = 1
end
```

```
module ImplT : T = struct
  let b = 2
end
```

```
module _ = (ImplS : T)
(* module _ = (ImplT : S) *)
```

# Private vs. Public Definitions

This gives us a simple way to distinguish between *private* and *public* definitions of a module:

- » Write a signature with gives an *interface* for the given module ("*interface*" is the "*i*" in *.mli*)
- » Use module constraints to force only those functions to be "visible" to the user

# demo

(private definitions)



# Functional Data Structures

# Abstract Types

```
module type S = sig
  type t
  type 'a t_param
  val op : t -> t -> t
  val op_param : 'a t_param -> 'a t_param -> 'a t_param
end
```

We can also define **abstract types** in modules

This is an extension of "private definitions" to include types

It allows us to define structures which are *type agnostic* to the "outside world"

# Interfaces for Functional Data Structures

```
module type LIST_STACK = sig
  type 'a stack
  val empty : 'a stack
  val push : 'a -> 'a stack -> 'a stack
  val pop : 'a stack -> 'a stack
end
```

# Interfaces for Functional Data Structures

```
module type LIST_STACK = sig
  type 'a stack
  val empty : 'a stack
  val push : 'a -> 'a stack -> 'a stack
  val pop : 'a stack -> 'a stack
end
```

So we can define modules which expose an abstract interface, *without* exposing the data representation

# Interfaces for Functional Data Structures

```
module type LIST_STACK = sig
  type 'a stack
  val empty : 'a stack
  val push : 'a -> 'a stack -> 'a stack
  val pop : 'a stack -> 'a stack
end
```

So we can define modules which expose an abstract interface, *without* exposing the data representation

This allows us to "swap out" our stack type without affecting any code which depends on the module

# Interfaces for Functional Data Structures

```
module type LIST_STACK = sig
  type 'a stack
  val empty : 'a stack
  val push : 'a -> 'a stack -> 'a stack
  val pop : 'a stack -> 'a stack
end
```

So we can define modules which expose an abstract interface, *without* exposing the data representation

This allows us to "swap out" our stack type without affecting any code which depends on the module

***This is just good abstraction: don't expose the low-level details unless it's necessary***

# Abstract Types are Opaque

```
module ListStack : LIST_STACK = struct
  type 'a stack = 'a list
  let empty = []
  let push x xs = x :: xs
  let pop = List.tl
end
```

```
let x = ListStack.(empty |> push 1 |> push 2)
(* let x = 3 :: x *)
```

We can't make *any* assumptions about an abstract type if we don't expose it

*Our code must still work if the abstract type changes*

# Important: This is not OOP

```
module ListStack : LIST_STACK = struct
  type 'a stack = 'a list
  let empty = []
  let push x xs = x :: xs
  let pop = List.tl
end
```

```
let x = ListStack.(empty |> push 1 |> push 2)
(* let x = 3 :: x *)
```

A module is not the same thing as a class, from which objects are instantiated (i.e., there is no **new** constructor)

Functions in structures are not *methods* of a given type of object

*(and there's still no mutability)*



demo  
(integer sets)

# **Advanced: Module Functors**

# High Level

```
module type A = sig  
  val a : int  
end
```

```
module B (ImplA : A) = struct  
  let b = ImplA.a  
end
```

We can parameterize modules by *other* modules

So the definitions in one module can depend on the implementation of another module

# A Common Pattern

```
module type Set = sig
  type 'a t
  val empty : 'a t
  val single : 'a -> 'a t
  val union : 'a t -> 'a t -> 'a t
end
```

A **set** data structure can be made more efficient if we can assume that its elements are *orderable* (so that we can use something like a binary tree)

**But how do we require that the keys are orderable?**

*(without (<) for reasons I won't get into)*

# A Common Pattern

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

```
module type Set = functor (E: Orderable) -> sig
  type t
  val empty : t
  val single : E.t -> t
  val union : t -> t -> t
end
```

We parameterize our **Set** module by an **Orderable** module which ensures that the underlying elements are *orderable*

Because of structural subtyping, we can parametrize by any type that *at least* implements **compare**

# Why do we care?

```
module VarSet = Set.Make(String)  
module Context = Map.Make(String)
```

Besides being interesting, we'll use sets and maps in our interpreters

Maps are natural data structures for representing contexts (collections of variable-type mappings)

*I mostly wanted to make sure you saw this before we got there*

# Summary

We can encapsulate data and define interfaces for types or data structures all with the same construct

When we write code in a file, we're building a module