CS 3110

# Modular Programming

Steffen Smolka

Spring 2018

# Moog modular synthesizer


Based in Trumansburg, NY, 1953-1971
Game changing!  picked up by the Beatles, the Rolling Stones…

# Review

**Previously in 3110:**

- Functions, data
- lots of language features
- how to build *small* programs

**Today:**

- language features for building *large* programs: structures, signatures, modules

# Scale

- My solution to A1: 100 LoC
- OCaml: 200,000 LoC
- Unreal engine 3: 2,000,000 LoC
- Windows Vista: 50,000,000 LoC

http://www.informationisbeautiful.net/visualizations/million-lines-of-code/

...can't be done by one person

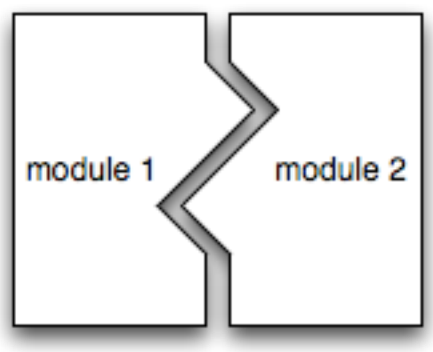
...no individual programmer can understand all the details

...too complex to build with subset of OCaml we've seen so far

# Modularity

**Modular programming:** code comprises independent *modules*

- developed separately
- understand behavior of module in isolation
- reason locally, not globally



# Java features for modularity

- **classes, packages:** organize identifiers (classes, methods, fields, etc.) into namespaces
- **interfaces:** describe related classes
- **public, protected, private:** control what is visible outside a namespace
- **subtyping, inheritance:** enables code reuse

# OCaml features for modularity

- **structures:** organize identifiers (functions, values, etc.) into namespaces
- **signatures:** describe related modules
- **abstract types:** control what is visible outside a namespace
- **functors, includes:** enable code reuse

...the OCaml *module system*

# Functional data structures

- aka *persistent* data structures
- Never mutate the data structure
- Old versions of the data structure *persist* and are still usable
- Language implementation ensures as much *sharing* as possible in memory
- In lecture: stacks
- In lab: queues and dictionaries

# STRUCTURES

```ocaml
module MyStack = 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
```

```ocaml
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
```

# Might seem backwards...

- In Java, might write

```
s = new Stack();
s.push(1);
s.pop();
```

- The stack is to the left of the dot, the method name is to the right
- In OCaml, it might feel backwards for awhile:

```
let s = MyStack.empty in
let s' = MyStack.push 1 s in
MyStack.peek s'
```

The stack is an argument to every function (common **idioms** are last argument or first argument)

- Just a syntactic detail (boring)

# Module syntax

```
module ModuleName = struct
  definitions
end
```

- the **`ModuleName`** must be capitalized
- definitions can include **`let`**, **`type`**, **`exception`**
- definitions can even include nested **`module`**

A module creates a new **namespace:**

```
module M = struct let x = 42 end
let y = M.x
```

# Module semantics

To evaluate a structure

```
struct
  def1
  def2
  ...
  defn
end
```

evaluate each definition in order

# SIGNATURES

# A multitude of implementations

- Each has its own *representation type*
  - **`MyStack`** uses **`'a stack`**
  - **`ListStack`** uses **`'a list`**
- Which causes each module to have a different *signature*...

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

module ListStack : ListStackSig = struct
  ...
end
```

```
module type MyStackSig = sig
  type 'a stack
    = Empty | Entry of 'a * '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 : MyStackSig = struct
  ...
end
```

# Module type syntax

```
module type SignatureName = sig
  type specifications
end
```

- type specifications aka *declarations*
- the **`SignatureName`** does not have to be capitalized but usually is
- declarations can include **`val`**, **`type`**, **`exception`**
- declarations can even include nested **`module type`**

# Module syntax revisited

```
module ModuleName : t = struct
  definitions
end

module ModuleName = (struct
  definitions
end : t)
```

type **t** must be a module type; including it has consequences…

# Module type semantics

If you give a module a type...

```
module Mod : Sig = struct ... end
```

Then type checker ensures...

1. **Signature matching:** everything declared in `Sig` must be defined in `Mod`
2. **Encapsulation:** nothing other than what's declared in `Sig` can be accessed from outside `Mod`

# 1. Signature matching

```
module type S1 = sig
  val x:int
  val y:int
end
module M1 : S1 = struct
  let x = 42
end
(* type error:
   Signature mismatch:
   The value `y' is required but not provided
*)
```

# 2. Encapsulation

```
module type S2 = sig
  val x:int
end
module M2 : S2 = struct
  let x = 42
  let y = 7
end
M2.y
(* type error: Unbound value M2.y *)
```

For Recitation

# ABSTRACT TYPES

# Imagine: Fast lists

Assume a hypothetical type **`'a fastlist`** with constructors **`FastNil`** and **`FastCons`** that have a more efficient implementation than **`'a list`**...

```
module FastStack = struct
  type 'a stack = 'a fastlist
  let empty = FastNil
  ...
end
```

Suppose you want to upgrade stacks from lists to fast lists...

# Exposure is bad

- Client code shouldn't **need to know** what the representation type is
- Rule of thumb: clients will exploit knowledge of representation if you let them
  - One day a client of **`ListStack`** will write **`x::s`** instead of **`push x s`**
  - And the day you upgrade to fast lists, you will break their code
- Client code shouldn't **get to know** what the representation type is

# Abstract types

```
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
```

# Abstract types

```
module type Stack = sig
  type 'a stack
```

- `'a stack` is **abstract**: signature *declares* only that type exists, but does not *define* what the type is
- Every module of type **`Stack`** must define the abstract type with some concrete type **`t`**
- Inside the module, `'a stack` and **`t`** are synonyms
- Outside the module, are not synonyms

# Abstract types

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

module ListStack : Stack = struct
  type 'a stack = 'a list
  ...

module FastListStack : Stack = struct
  type 'a stack = 'a fastlist
  ...
```

# Abstract types

```
module ListStack : Stack = struct
  type 'a stack = 'a list
  let empty = []
  ...
```

Recall: outside the module, types are not synonyms

So **`List.hd ListStack.empty`** will not compile

# Abstract types

**General principle:** information hiding *aka* encapsulation

- *Clients* of **`Stack`** don't need to know it's implemented (e.g.) with a list
- *Implementers* of **`Stack`** might one day want to change the implementation
  - If list implementation is exposed, they can't without breaking all their clients' code
  - If list implementation is hidden, they can freely change
  - e.g., suppose Microsoft wants to update the data structure representing a window or canvas or file

# Abstract types

Common **idiom** is to call the abstract type **t**:

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

module ListStack : Stack = struct
  type 'a t  = 'a list
  ...
```