The Moog modular synthesizer
Review

Previously in 3110:
• Functions, data
• lots of language features
• how to build small programs

Today:
• how to build large programs
• language features: structure, signatures, modules
Question

What’s the largest program you’ve ever worked on, by yourself or as part of a team?

A. 10-100 LoC
B. 100-1,000 LoC
C. 1,000-10,000 LoC
D. 10,000-100,000 LoC
E. 100,000 LoC or bigger
Scale

- Staff solution to A1: 100 LoC
- Staff solution to A2: 200 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

...together, these features comprise the OCaml module system
MODULES
Running examples

• Stacks
• Queues

• *Functional* aka *persistent* data structures:
  – never mutate the data structure
  – old versions of the data structure *persist* and are still usable
Stack module

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
Another Stack module

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
  
  ```java
  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's seemingly backward:
  ```ocaml
  let s = MyStack.empty in
  let s' = MyStack.push 1 s in
  let one = MyStack.peek s'
  ```

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

- Just a syntactic detail (boring)
Yet another Stack module

Assume a type 'a fastlist with constructor FastNil and FastCons that have a more efficient implementation than 'a list...

module FastStack = struct
  let empty = FastNil
  ...
end
A multitude of implementations

• Each has its own representation type
  – **MyStack** uses 'a stack
  – **ListStack** uses 'a list
  – **FastStack** uses (hypothetical) 'a fastlist

• Which causes each module to have a different interface...
**Defining signatures**

```ocaml
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
```
Stack signatures

module ListStack : 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 MyStack : 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
Stack signatures

module ListStack : 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 MyStack : 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
A multitude of implementations

- Client code shouldn't need to know what the representation type is
- Client code shouldn't get 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
- So how can we unify these representations?
Abstract types

```ocaml
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
```

'a stack is abstract: signature declares only that type exists, but does not define what the type is
Abstract types

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

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

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

- Every module of type `Stack` must define the abstract type
- Inside the module, types are synonyms
- Outside the module, types are not synonyms
  ```ocaml
  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 $t$:

```ocaml
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
  ...
```
Queues

• Two implementations of functional queues in code accompanying lecture:
  – Queues as lists (poor performance)
  – Queues as two lists (good performance)

• Study these!
SYNTAX AND SEMANTICS
Module syntax

module ModuleName [:t] = struct
  definitions
end

• the ModuleName must be capitalized
• type t (which must be a module type) is optional
• 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
**Signature syntax**

```ocaml
code
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`
  - `val name : type`
  - `type t [= definition]`
- declarations can even include nested `module type`
Type checking

If you give a module a type...

```plaintext
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 *)
Evaluation

To evaluate a module

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

evaluate each definition in order
Modules and files

Compilation unit = \texttt{myfile.ml + myfile.mli}

If \texttt{myfile.ml} has contents $DM$
[and \texttt{myfile.mli} has contents $DS$]
then OCaml compiler behaves essentially as though:

\begin{verbatim}
module Myfile [: sig DS end] =
struct
  DM
end
\end{verbatim}
**Modules and files**

**File `stack.mli`:**

(* The type of a stack whose elements are type 'a *)

```ocaml
type 'a t
(* The empty stack *)
val empty : 'a t
(* Whether the stack is empty*)
val is_empty : 'a t -> bool
(* [push x s] is the stack [s] with [x] pushed on the top *)
val push : 'a -> 'a t -> 'a t
(* [peek s] is the top element of [s]. raises Failure if [s] is empty *)
val peek : 'a t -> 'a
(* [pop s] pops and discards the top element of [s]. raises Failure if [s] is empty *)
val pop : 'a t -> 'a t
```

Note: no `struct` or `sig` keywords, no naming of module or module type

**File `stack.ml`:**

(* Represent a stack as a list. [x::xs] is the stack with top element [x] and remaining elements [xs]. *)

```ocaml
type 'a t = 'a list
let empty = []
let is_empty s = s = []
let push x s = x :: s
(* Consider: using options instead of exceptions. *)
let peek = function
  | [] -> failwith "Empty"
  | x::_ -> x
let pop = function
  | [] -> failwith "Empty"
  | _::xs -> xs
```

Note: comments to client in `.mli`, comments to implementers in `.ml`
What about `main()`?

- There is no specific entry point into a module
- Common **idiom** is to make the last definition in a module be a function call that starts computation, e.g.

  ```
  let _ = go_do_stuff()
  ```

- And you might call that function `main` instead of `go_do_stuff`, but you don't need to
Upcoming events

• [next Thursday] A2 due

This is modular.

THIS IS 3110