FUNCTIONAL PROGRAMMING

Modular Programming

Prof. Clarkson
Summer 2015

Today’s music: "Giorgio By Moroder" by Daft Punk
The Moog modular synthesizer
Review

Last class:
• Algebraic data types (recursive, polymorphic)
• Exceptions

So far:
• lots of language features
• how to build small programs

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

• OCaml: 200,000
• Unreal engine: 2,000,000
• Windows 7: 40,000,000

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
OCaml features for modularity

• modules
  – organize identifiers (functions, values, etc.) into namespaces

• signatures
  – describe related modules

• abstract types
  – control what is visible outside a namespace
Stack module

(* implement stacks as lists *)
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

ListStack.peek (ListStack.push 1 ListStack.empty) ==> 1
Might seem backwards...

• In Java, might write
  
  ```java
  s = new ListStack();
  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 = ListStack.empty in
  let s' = ListStack.push 1 s in
  let one = ListStack.peek s'
  ```
  
  • The stack is an argument to every function (common **idioms** are last argument or first argument)

• **Just a syntactic detail** (boring)
Stack signature

```ocaml
module type Stack = 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 : Stack = struct ... end
```
Other Stack implementations

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

module FastListStack : Stack = struct
  let empty = FastNil
  ...
end

Type of empty is now 'a fastlist -> bool

Won't compile:
  Stack requires empty : 'a list -> bool
Other Stack implementations

- The representation type changed
  - from 'a list
  - to 'a fastlist

- If type is abstract in signature, clients continue to compile

- If type is revealed in signature, clients who relied on a list fail to compile
  - For more complicated data structures, this problem can be very bad
  - e.g., suppose Microsoft wants to update the data structure representing a window or canvas or file
Other Stack implementations

General principle: information hiding aka encapsulation

• *Clients* of **Stack** don’t need to know it’s implemented 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
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

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

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

module type SignatureName = sig
  declarations
end

• 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

```ocaml
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

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

evaluate each definition in order
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.
PRAGMATICS
Modules and files

Compilation unit = myfile.ml + myfile.mli

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

[module type Myfile = sig
   DS
   end]

module Myfile [:: Myfile] = struct
   DM
end

Module types and modules are in separate namespaces, so the above really does work, though it might be confusing.
Modules and files

File `stack.mli`:

```ocaml
module 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
```

File `stack.ml`:

```ocaml
module type 'a t = 'a list
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
```

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

Must compile interface first:

```
ocamlc stack.mli
(* produces stack.cmi *)
ocamlc stack.ml
(* produces stack.cmo *)
```

Beware that `#use` does not check the `.ml` for conformance with the `.mli`
Namespace management

Recall this code:

```ml
let s = ListStack.empty in
let s' = ListStack.push 1 s in
let one = ListStack.peek s'
```

Code is less readable with "ListStack." everywhere (and annoying to write!)

Four solutions...
1. Module-scoped open

• Anywhere you could write a definition in module, write `open M`
• Throughout module, instead of writing `M.x` you can write just `x`

```ml
open ListStack

let s = empty in
let s' = push 1 s in
let one = peek s'
```

• Best not to do this too often: later opens shadow earlier, perhaps leaving to surprising type errors
2. Let-scoped open

- As part of let expression, write `open M`
- From that point forward in the let expression, instead of writing `M.x` you can write just `x`

```ml
let open ListStack in
let s = empty in
let s' = push 1 s in
let one = peek s'
```
3. Expression-scoped open

- As part of expression, write \( \text{M. (e)} \)
- In expression \( e \), instead of writing \( \text{M. x} \) you can write just \( x \)

```let one =
  ListStack.(peek (push 1 empty))
```
4. Module synonym

- As part of expression, write `let module M=ModName in e`
- In expression `e`, instead of writing `M.x` you can write just `x`

```
let module L = ListStack in
let s = L.empty in
let s' = L.push 1 s in
let one = L.peek s'
```

- Useful if you have a couple modules with overlapping names that you need to use in same scope