Modules and Abstract Data Types

COS 326
David Walker
Princeton University

Reminder

The take-home midterm exam has been posted.

You can use any course materials but it is an individual exercise. It is "closed friends" "closed TAs" and "closed rest of the internet"

See Piazza for instructions on how to download it.

You have 24 hours to do the exam.

The latest start time is Tuesday 11:59PM

The latest end time is Wednesday 11:59PM.

(ie: like an assignment, hand it in before midnight on Wednesday)

The Reality of Development

We rarely know the *right* algorithms or the *right* data structures when we start a design project.

- When implementing a search engine, what data structures and algorithms should you use to build the index? To build the query evaluator?

Reality is that *we often have to go back and change our code*, once we've built a prototype.

- Often, we don't even know what the *user wants* (requirements) until they see a prototype.
- Often, we don't know where the *performance problems* are until we can run the software on realistic test cases.
- Sometimes we just want to change the design -- come up with *simpler* algorithms, architecture later in the design process

Engineering for Change

Given that we know the software will change, how can we write the code so that doing the changes will be easier?

Engineering for Change

Given that we know the software will change, how can we write the code so that doing the changes will be easier?

The primary trick: use *data and algorithm abstraction*.

Engineering for Change

Given that we know the software will change, how can we write the code so that doing the changes will be easier?

The primary trick: use *data and algorithm abstraction*.

- *Don't* code in terms of *concrete representations* that the language provides.
- *Do* code with *high-level abstractions* in mind that fit the problem domain.
- Implement the abstractions using a *well-defined interface*.
- Swap in *different implementations* for the abstractions.
- *Parallelize* the development process.

Example

Goal: Implement a query engine.

Requirements: Need a scalable *dictionary* (a.k.a. index)

- maps words to *set* of URLs for the pages on which words appear.
- want the index so that we can efficiently satisfy queries
  - e.g., all links to pages that contain “Dave” and “Jill”.

Wrong way to think about this:

- Aha! A *list* of pairs of a word and a *list* of URLs.
- We can look up “Dave” and “Jill” in the *list* to get back a *list* of URLs.

Example

```
type query =
  Word of string
| And of query * query
| Or of query * query

type index = (string * (url list)) list

let rec eval(q:query)(h:index) : url list =
  match q with
  | Word x ->
        let (_,urls) = List.find (fun (w,urls) -> w = x) h in
        urls
  | And (q1,q2) ->
        merge_lists (eval q1 h) (eval q2 h)
  | Or (q1,q2) ->
        (eval q1 h) @ (eval q2 h)
```

Example

```
type query =
  Word of string
| And of query * query
| Or of query * query

type index = (string * (url list)) list

let rec eval(q:query)(h:index) : u
  match q with
  | Word x ->
      let (_,urls) = List.find                    w = x) in
      urls
  | And (q1,q2) ->
      merge_lists (eval q1 h) (eval q2 h)
  | Or (q1,q2) ->
      (eval q1 h) @ (eval q2 h)
```

merge expects to be passed sorted lists.

Example

```
type query =
  Word of string
| And of query * query
| Or of query * query

type index = (string * (url list)) list

let rec eval(q:query)(h:index) : u
  match q with
   | Word x ->
       let (_,urls) = List.find                        ) in
       urls
   | And (q1,q2) ->
       merge_lists (eval q1 h) (e
   | Or (q1,q2) ->
       (eval q1 h) @ (eval q2 h)
```

merge expects to be passed sorted lists.

Oops!

Example

```
type query =
  Word of string
| And of query * query
| Or of query * query

type index = string (url list) hashtable

let rec eval(q:query)(h:index) : url list =
  match q with
  | Word x ->
      let i = hash_string x in
      let l = Array.get h [i] in
      let urls = assoc_list_find l x in
      urls
  | And (q1,q2) -> ...
  | Or (q1,q2) -> ...
```

I find out there's a better hash-table implementation

A Better Way

```
type query =
  Word of string
| And of query * query
| Or of query * query

type index = string url_set dictionary

let rec eval(q:query)(d:index) : url_set =
  match q with
   | Word x -> Dict.lookup d x
   | And (q1,q2) -> Set.intersect (eval q1 h) (eval q2 h)
   | Or (q1,q2) -> Set.union (eval q1 h) (eval q2 h)
```

A Better Way

```
type query =
  Word of string
| And of query * query
| Or of query * query

type index = string url_set dictionary

let rec eval(q:query)(d:index) : url_set =
  match q with
   | Word x -> Dict.lookup d x
   | And (q1,q2) -> Set.intersect (eval q1 h) (eval q2 h)
   | Or (q1,q2) -> Set.union (eval q1 h) (eval q2 h)
```

The problem domain talked about an abstract type of *dictionaries* and *sets of URLs*.

A Better Way

```
type query =
  Word of string
| And of query * query
| Or of query * query

type index = string url_set dictionar

let rec eval(q:query)(d:index) : url_s
  match q with
   | Word x -> Dict.lookup d x
   | And (q1,q2) -> Set.intersect (eval q1 h) (eval q2 h)
   | Or (q1,q2) -> Set.union (eval q1 h) (eval q2 h)
```

The problem domain talked about an abstract type of *dictionaries* and *sets of URLs.*

Once we've written the client, we know what operations we need on these abstract types.

A Better Way

```
type query =
  Word of string
| And of query * query
| Or of query * query

type index = string url_set dictionar

let rec eval(q:query)(d:index) : url_s
  match q with
  | Word x -> Dict.lookup d x
  | And (q1,q2) -> Set.intersect (eval q1 h) (eval q2 h)
  | Or (q1,q2) -> Set.union (eval q1 h) (eval q2 h)
```

The problem domain talked about an abstract type of *dictionaries* and *sets of URLs*.

Once we've written the client, we know what operations we need on these abstract types.

So we can define an interface, and send a pal off to implement the *abstract types* dictionary and set.

Later on, when we find out linked lists aren't so good for sets, we can replace them with balanced trees.

Abstract Data Types

Barbara Liskov
Assistant Professor, MIT
1973

Invented CLU language
that enforced data abstraction

Barbara Liskov
Professor, MIT
Turing Award 2008

"For contributions to practical and theoretical foundations of programming language and system design, especially related to data abstraction, fault tolerance, and distributed computing."

Building Abstract Types in OCaml

OCaml has mechanisms for building new abstract data types:

- ***signature***: an interface.
  - specifies the abstract type(s) without specifying their implementation
  - specifies the set of operations on the abstract types
- ***structure***: an implementation.
  - a collection of type and value definitions
  - notion of an implementation matching or satisfying an interface
    - gives rise to a notion of sub-typing
- ***functor***: a parameterized module
  - really, a function from modules to modules
  - allows us to factor out and re-use modules

The Abstraction Barrier

Rule of thumb: Use the language to enforce the abstraction barrier.

- Reveal little information about *how* something is implemented
- Provide maximum flexibility for change moving forward.
- Murphy's Law: What is not enforced, will be broken

But rules are meant to broken: Exercise judgement.

- may want to reveal more information for debugging purposes
  - eg: conversion to string so you can print things out

ML gives you precise control over how much of the type is left abstract

- different amounts of information can be revealed in different contexts
- type checker helps you detect violations of the abstraction barrier

Simple Modules

Recall assigment #2:

query.ml

```
type movie = { ... }

let sort_by_studio  = ...
let sort_by_year = ...
```

main.ml

```
open Io
open Query

let main () = ... sort_by_studio ...

let _ = main ()
```

Simple Modules

Recall assigment #2:

query.ml

```
type movie = { ... }

let sort_by_studio  = ...
let sort_by_year = ...
```

main.ml

```
open Io
open Query

let main () = ... sort_by_studio ...

let _ = main ()
```

Each .ml file actually defines an ML module.

Convention: the file foo.ml or Foo.ml defines the module named Foo.

Simple Modules

Recall assigment #2:

open gives direct access to module components

query.ml

```
type movie = { ... }

let sort_by_studio  = ...
let sort_by_year = ...
```

main.ml

```
open Io
open Query

let main () = ... sort_by_studio ...

let _ = main ()
```

Simple Modules

Recall assigment #2:

query.ml

```
type movie = { ... }

let sort_by_studio  = ...
let sort_by_year = ...
```

main.ml

```
open Io
open Query

let main () =
  ... Query.sort_by_studio ...
```

redacted

Can refer to module components using dot notation

Simple Modules

query.ml

```
type movie = { ... }

let sort_by_studio  = ...
let sort_by_year = ...
```

main.ml

```
open Io
~~open Query~~

let main () =
  ... Query.sort_by_studio ...
```

query.mli

```
type movie

val sort_by_studio : movie list -> movie list
val sort_by_year : movie list -> movie list
```

You can add interface files (.mli) (also called *signatures* in ML)

These interfaces can hide module components or render types abstract.

Simple Modules

query.ml

```
type movie = { ... }

let sort_by_studio  = ...
let sort_by_year = ...
```

main.ml

```
open Io
open Query

let main () =
  ... Query.sort_by_studio ...
```

query.mli

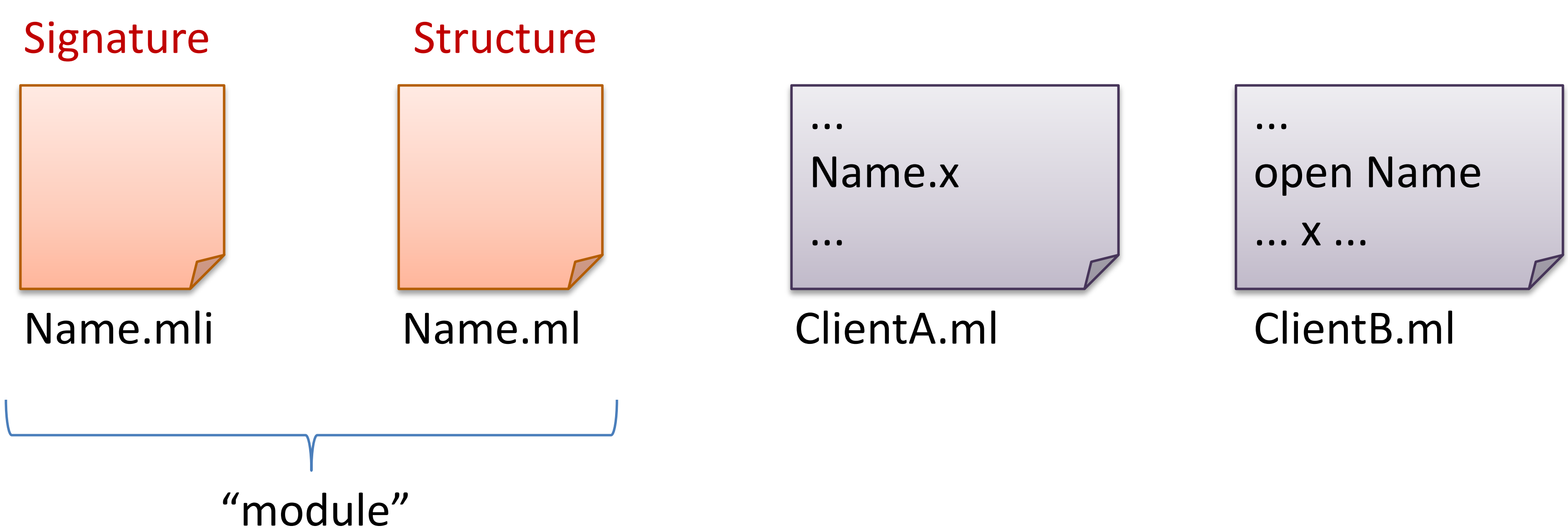
```
type movie

val sort_by_studio : movie list -> movie list
val sort_by_year : movie list -> movie list
```

If you have no signature file, then the default signature is used: all components are fully visible to clients.

Simple Modules

Simple summary:

- file Name.ml is a *structure* implementing a module named Name
- file Name.mli is a *signature* for the module named Name
  - if there is no file Name.mli, OCaml infers the default signature



At first glance: OCaml modules = C modules?

C has:

- .h files (signatures) similar to .mli files?
- .c files (structures) similar to .ml files?

But ML also has:

- tighter control over type abstraction
  - define abstract, transparent or translucent types in signatures
    - i.e.: give none, all or some of the type information to clients
- more structure
  - modules can be defined within modules
  - i.e.: signatures and structures can be defined inside files
- more reuse
  - multiple modules can satisfy the same interface
  - the same module can satisfy multiple interfaces
  - modules take other modules as arguments (functors)
- fancy features: dynamic, first class modules

Signature Definitions Inside Files

```
module type INT_STACK =
  sig
    type stack
    val empty : unit -> stack
    val push  : int -> stack -> stack
    val is_empty : stack -> bool
    val pop : stack -> stack
    val top : stack -> int option
  end
```

Signature Definitions Inside Files

```
module type INT_STACK =
  sig
    type stack
    val empty : unit -> stack
    val push  : int -> stack -> stack
    val is_empty : stack -> bool
    val pop : stack -> stack option
    val top : stack -> int option
  end
```

empty and push are abstract ***constructors***: functions that build our abstract type.

Signature Definitions Inside Files

```
module type INT_STACK =
  sig
    type stack
    val empty : unit -> stack
    val push  : int -> stack -> stack
    val is_empty : stack -> bool
    val pop : stack -> stack
    val top : stack ->
  end
```

is_empty is an ***observer*** – useful for determining properties of the ADT.

Signature Definitions Inside Files

```
module type INT_STACK =
  sig
    type stack
    val empty : unit -> stack
    val push  : int -> stack -> stack
    val is_empty : stack -> bool
    val pop : stack -> stack
    val top : stack -> int option
  end
```

pop is sometimes called a ***mutator*** (though it doesn't really change the input)

Signature Definitions Inside Files

```
module type INT_STACK =
  sig
    type stack
    val empty : unit -> stack
    val push  : int -> stack -> stack
    val is_empty : stack -> bool
    val pop : stack -> stack
    val top : stack -> int option
  end
```

top is also an *observer*, in this functional setting since it doesn't change the stack.

Put comments in your signature!

```
module type INT_STACK =
  sig
    type stack

    (* create an empty stack *)
    val empty : unit -> stack

    (* push an element on the top of the stack *)
    val push  : int -> stack -> stack

    (* returns true iff the stack is empty *)
    val is_empty : stack -> bool

    (* pops top element off the stack;
       returns empty stack if the stack is empty *)
    val pop : stack -> stack

    (* returns the top element of the stack; returns
        None if the stack is empty *)
    val top : stack -> int option
  end
```

Signature Comments

Signature comments are for clients of the module

- explain what each function should do
  - how it manipulates abstract values (stacks)
- ***not*** how it manipulates concrete values
- don't reveal implementation details that should be hidden behind the abstraction

Don't copy signature comments into your structures

- your comments will get out of date in one place or the other
- an extension of the general rule: don't copy code

Place implementation comments inside your structure

- comments about implementation invariants hidden from client
- comments about helper functions

https://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html

Previous Up Next

Module List

`module List: sig .. end`

List operations.

Some functions are flagged as not tail-recursive. A tail-recursive function uses constant stack space, while a non-tail-recursive function uses stack space proportional to the length of its list argument, which can be a problem with very long lists. When the function takes several list arguments, an approximate formula giving stack usage (in some unspecified constant unit) is shown in parentheses.

The above considerations can usually be ignored if your lists are not longer than about 10000 elements.

`val length : 'a list -> int`

Return the length (number of elements) of the given list.

`val compare_lengths : 'a list -> 'b list -> int`

Compare the lengths of two lists. `compare_lengths l1 l2` is equivalent to `compare (length l1) (length l2)`, except that the computation stops after itering on the shortest list.
Since 4.05.0

`val compare_length_with : 'a list -> int -> int`

Compare the length of a list to an integer. `compare_length_with l n` is equivalent to `compare (length l) n`, except that the computation stops after at most `n` iterations on the list.
Since 4.05.0

`val cons : 'a -> 'a list -> 'a list`

`cons x xs` is `x :: xs`
Since 4.03.0

`val hd : 'a list -> 'a`

Return the first element of the given list. Raise `Failure "hd"` if the list is empty.

`val tl : 'a list -> 'a list`

Return the given list without its first element. Raise `Failure "tl"` if the list is empty.

`val nth : 'a list -> int -> 'a`

Return the `n`-th element of the given list. The first element (head of the list) is at position 0. Raise `Failure "nth"` if the list is too short. Raise `Invalid_argument "List.nth"` if `n` is negative.

`val nth_opt : 'a list -> int -> 'a option`

Return the `n`-th element of the given list. The first element (head of the list) is at position 0. Return `None` if the list is too short. Raise `Invalid_argument "List.nth"` if `n` is negative.
Since 4.05

`val rev : 'a list -> 'a list`

Example Structure Inside a File

```
module ListIntStack : INT_STACK =
  struct
    type stack = int list
    let empty () : stack = []
    let push (i:int) (s:stack) : stack = i::s
    let is_empty (s:stack) =
      match s with
       | [] -> true
       | _::_ -> false
    let pop (s:stack) : stack =
      match s with
       | [] -> []
       | _::t -> t
    let top (s:stack) : int option =
      match s with
       | [] -> None
       | h::_ -> Some h
  end
```

Example Structure Inside a File

```
module ListIntStack : INT_STACK =
  struct
    type stack = int list
    let empty () : stack = []
    let push (i:int) (s:stack) = i:
    let is_empty (s:stack) =
      match s with
       | [] -> true
       | _::_ -> false
    let pop (s:stack) : stack =
      match s with
       | [] -> []
       | _::t -> t
    let top (s:stack) : int option =
      match s with
       | [] -> None
       | h::_ -> Some h
  end
```

Inside the module, we know the ***concrete type*** used to implement the abstract type.

Example Structure Inside a File

```
module ListIntStack : INT_STACK =
  struct
    type stack = int list
    let empty () : stack = []
    let push (i:int) (s:stack) = i
    let is_empty (s:stack) =
      match s with
       | [] -> true
       | _::_ -> false
    let pop (s:stack) : stack =
      match s with
       | [] -> []
       | _::t -> t
    let top (s:stack) : int option =
      match s with
       | [] -> None
       | h::_ -> Some h
  end
```

But by giving the module the INT_STACK interface, which does not reveal how stacks are being represented, we prevent code outside the module from knowing stacks are lists.

An Example Client

```
module ListIntStack : INT_STACK =
  struct
    …
  end

let s0 = ListIntStack.empty ()
let s1 = ListIntStack.push 3 s0
let s2 = ListIntStack.push 4 s1
let x = ListIntStack.top s2
```

An Example Client

```
module ListIntStack : INT_STACK =
  struct
    …
  end

let s0 = ListIntStack.empty ()
let s1 = ListIntStack.push 3 s0
let s2 = ListIntStack.push 4 s1
let x = ListIntStack.top s2
```

```
s0 : ListIntStack.stack
s1 : ListIntStack.stack
s2 : ListIntStack.stack
```

An Example Client

```
module ListIntStack : INT_STACK =
  struct
    …
  end

let s0 = ListIntStack.empty ()
let s1 = ListIntStack.push 3 s0
let s2 = ListIntStack.push 4 s1
let x = ListIntStack.top s2
x : option int = Some 4
```

An Example Client

```
module ListIntStack : INT_STACK =
  struct
    …
  end

let s0 = ListIntStack.empty ()
let s1 = ListIntStack.push 3 s0
let s2 = ListIntStack.push 4 s1
let x = ListIntStack.top s2
x : option int = Some 4
let x = ListIntStack.top (ListIntStack.pop s2)
x : option int = Some 3
```

An Example Client

```
module ListIntStack : INT_STACK =
  struct
    …
  end

let s0 = ListIntStack.empty ()
let s1 = ListIntStack.push 3 s0
let s2 = ListIntStack.push 4 s1
let x = ListIntStack.top s2
x : option int = Some 4
let x = ListIntStack.top (ListIntStack.pop s2)
x : option int = Some 3
open ListIntStack
```

An Example Client

```
module ListIntStack : INT_STACK =
  struct
    …
  end

let s0 = ListIntStack.empty ()
let s1 = ListIntStack.push 3 s0
let s2 = ListIntStack.push 4 s1
let x = ListIntStack.top s2
x : option int = Some 4
let x = ListIntStack.top (ListIntStack.pop s2)
x : option int = Some 3
open ListIntStack
let x = top (pop (pop s2))
x : option int = None
```

An Example Client

```
module type INT_STACK =
  sig
    type stack
    val push  : int -> stack -> stack
    …
  end
module ListIntStack : INT_STACK

let s2 = ListIntStack.push 4 …
…
List.rev s2
Error: This expression has type stack but an
expression was expected of type 'a list.
```

Notice that the client is not allowed to know that the stack is a list.

Example Structure

```
module ListIntStack (* : INT_STACK *) =
  struct
    type stack = int list
    let empty () : stack = []
    let push (i:int) (s:stack) = i::s
    let is_empty (s:stack) =
      match s with
       | [ ] -> true
       | _::_ -> false
    exception EmptyStack
    let pop (s:stack) =
      match s with
       | [] -> []
       | _::t -> t
    let top (s:stack) =
      match s with
       | [] -> None
       | h::_ -> Some h
  end
```

Note that when you are debugging, you may want to comment out the signature ascription so that you can access the contents of the module.

The Client without the Signature

```
module ListIntStack (* : INT_STACK *) =
  struct
    …
  end

let s = ListIntStack.empty()
let s1 = ListIntStack.push 3 s
let s2 = ListIntStack.push 4 s1

…
let x = List.rev s2
x : int list = [3; 4]
```

If we don't seal the module with a signature, the client can know that stacks are lists.

Example Structure

```
module ListIntStack : INT_STACK =
  struct
    type stack = int list
    let empty () : stack = []
    let push (i:int) (s:stack) =
    let is_empty (s:stack) =
      match s with
       | [ ] -> true
       | _::_ -> false
    exception EmptyStack
    let pop (s:stack) =
      match s with
       | [] -> []
       | _::t -> t
    let top (s:stack) =
      match s with
       | [] -> None
       | h::_ -> Some h
  end
```

When you put the signature on here, you are restricting client access to the information in the signature (which does *not* reveal that stack = int list.) So clients can *only* use the stack operations on a stack value (not list operations.)

Example Structure

```
module type INT_STACK =
  sig
    type stack
    ...

    val inspect : stack -> int list
    val run_unit_tests : unit -> unit

  end


module ListIntStack : INT_STACK =
  struct
    type stack = int list
    ...

    let inspect (s:stack) : int list = s
    let run_unit_tests () : unit = ...
  end
```

Another technique:

Add testing components to your signature.

Or have 2 signatures, one for testing and one for the rest of the code)

DESIGN CHOICES FOR CORNER CASES

Interface design

```
module type INT_STACK =
  sig
    type stack
    (* create an empty stack *)
    val empty : unit -> stack

    (* push an element on the top of the stack *)
    val push  : int -> s

    (* returns true ifl                              y *)
    val is_empty : stack -> boo

    (* pops top element off the stack;
       returns empty stack if the stack is empty *)
    val pop : stack -> stack

    (* returns the top element of the stack; returns
       None if the stack is empty *)
    val top : stack -> int option
  end
```

Is this a good idea?

Design choices

```
sig
 type stack
 (* pops top element;
    returns empty if empty
  *)
  val pop : stack -> stack
end
```

```
sig
 type stack
 (* pops top element;
    returns arbitrary stack
    if empty *)
  val pop : stack -> stack
end
```

```
sig
 type stack
 (* pops top element;
    returns option *)
  val pop:
       stack -> stack option
end
```

```
sig
 type stack
 exception EmptyStack
 (* pops top element;
    raises EmptyStack if empty
  *)
  val pop : stack -> stack
end
```

Design choices

For some functions,

there are some input values

outside the *domain*

of the function & the domain

is not easily described by

a simple type.

```
sig
 type stack
 (* pops top element;
    returns arbitrary stack
    if empty *)
  val pop : stack -> stack
end
```

Say the function returns an arbitrary result on those inputs.

When proving things about the program, there's an extra proof obligation: Prove that the input is in the domain of the function.

Design choices

For some functions,
there are some input values
outside the *domain*
of the function & the domain
is not easily described by
a simple type.

```
sig
 type stack
 (* pops top element;
    returns arbitrary stack
    if empty *)
  val pop : stack -> stack
end
```

Say the function returns an arbitrary result on those inputs.

When proving things about the program, there's an extra proof obligation: Prove that the input is in the domain of the function.

But when a programmer forgets to do this proof (or makes a mistake), such silent errors can be hard to track down.

Design choices

For some functions, there are some input values outside the *domain* of the function.

```
sig
 type stack
 (* pops top element;
    crashes the program
    if empty *)
  val pop : stack -> stack
end
```

This is not *completely* crazy. One might still be able to guarantee that the input is always in the domain of the function.

It's what the C language does, for example.

Design choices

For some functions, there are some input values outside the *domain* of the function.

```
sig
 type stack
 (* pops top element;
    crashes the program
    if empty *)
  val pop : stack -> stack
end
```

This is not *completely* crazy. One might still be able to guarantee that the input is always in the domain of the function.

It's what the C language does, for example.

But it's ***almost completely*** crazy. This is the biggest source of security vulnerabilities ever. It's why the hackers can drive your car, steal your money, read your e-mails, …

Design choices

```
sig
 type stack
 (* pops top element;
    returns empty if empty
  *)
  val pop : stack -> stack
end
```

```
sig
 type stack
 (* pops top element;
    returns arbitrary stack
   if empty *)
  val pop : stack -> stack
end
```

It's also reasonable to say the function returns a *specified, convenient,* result on those inputs.
This is pretty much the same thing, in practice.

Consider: If supplying an empty stack to pop is probably a mistake on the part of the caller, it is better to stop the program right away (by raising an exception) than to let the error silently slip by. In the long run, finding the real error is tougher.

Design choices

For some functions,

there are some

input values

outside the *domain*

of the function.

That's what exceptions are for!

Raise an exception for values

not in the domain.

```
sig
 type stack
 exception EmptyStack
 (* pops top element;
    raises EmptyStack if empty
   *)
  val pop : stack -> stack
end
```

Design choices

Finally, you can just use option types in the obvious way.

Using an option has the advantage of forcing the caller to consider what to do on the “error” condition every time the function is called. They can’t *forget* to handle this situation.

```
sig
 type stack
 (* pops top element;
    returns option *)
  val pop:
       stack -> stack option
end
```

Design choices

```
sig
  type stack
  (* pops top element;
       returns empty if empty
    *)
   val pop : stack -> stack
end
```

```
sig
  type stack
  (* pops top element;
       returns arbitrary stack
     if empty *)
   val pop : stack -> stack
```

All of these are reasonable design choices!

```
sig
  type stack
  (* pops top element;
       returns option *)
   val pop:
          stack -> stack option
end
```

```
  type stack
  exception EmptyStack
  (* pops top element;
       raises EmptyStack if empty
     *)
   val pop : stack -> stack
end
```

Design choices

```
sig
 type stack
 (* pops top element;
    returns empty if empty
  *)
  val pop : stack -> stack
end
```

But use these two with extreme care

```
 type stack
 (* pops top element;
    returns arbitrary stack
  if empty *)
  val pop : stack -> stack
```

All of these are reasonable design choices!

```
sig
 type stack
 (* pops top element;
    returns option *)
  val pop:
       stack -> stack option
end
```

```
 type stack
 exception EmptyStack
 (* pops top element;
    raises EmptyStack if empty
  *)
  val pop : stack -> stack
end
```

But use the bottom two are more common. Options are the “safest.” They force consideration of the error condition every time.

ANOTHER EXAMPLE

Polymorphic Queues

```
module type QUEUE =
  sig
    type 'a queue
    val empty : unit -> 'a queue
    val enqueue : 'a -> 'a queue -> 'a queue
    val is_empty : 'a queue -> bool
    exception EmptyQueue
    val dequeue : 'a queue -> 'a queue
    val front : 'a queue -> 'a
  end
```

Polymorphic Queues

```
module type QUEUE =
  sig
    type ‘a queue
    val empty : unit -> ‘a queue
    val enqueue : ‘a -> ‘a queue -> ‘a queue
    val is_empty : ‘a queue -> bool
    exception EmptyQueue
    val dequeue : ‘a queue -> ‘a queue
    val front : ‘a queue -> ‘a
  end
```

These queues are re-usable for different element types.

Here's an exception that client code might want to catch

One Implementation

```
module AppendListQueue : QUEUE =
  struct
    type ‘a queue = ‘a list
    let empty() = []
    let enqueue(x:’a)(q:’a queue) : ‘a queue = q @ [x]
    let is_empty(q:’a queue) =
     match q with
     | [] -> true
     | _::_ -> false

    ...



end
```

One Implementation

```
module AppendListQueue : QUEUE =
  struct
    type ‘a queue = ‘a list
    let empty() = []
    let enqueue(x:’a)(q:’a queue) : ‘a queue = q @ [x]
    let is_empty(q:’a queue) = ...

    exception EmptyQueue
    let deq(q:’a queue) : (‘a * ‘a queue) =
      match q with
      | [] -> raise EmptyQueue
      | h::t -> (h,t)
    let dequeue(q:’a queue) : ‘a queue = snd (deq q)
    let front(q:’a queue) : ‘a = fst (deq q)
end
```

One Implementation

```
module AppendListQueue : QUEUE =
  struct
    type 'a queue = 'a list
    let empty() = []
    let enqueue(x:'a)(q:'a queue) : 'a
    let is_empty(q:'a queue) = ...

    exception EmptyQueue
    let deq(q:'a queue) : ('a * 'a queue) =
      match q with
      | [] -> raise EmptyQueue
      | h::t -> (h,t)
    let dequeue(q:'a queue) : 'a que
    let front(q:'a queue) : 'a = fst (de
end
```

Notice deq is a helper function that doesn't show up in the signature.

You can't use it outside the module.

One Implementation

```
module AppendListQueue : QUEUE =
  struct
    type ‘a queue = ‘a list
    let empty() = []
    let enqueue(x:’a)(q:’a queue) : ‘a queue = q @ [x]
    let is_empty(q:’a queue) = ...

    exception EmptyQueue
    let deq(q:’a queue) : (‘a *
      match q with
      | [] -> raise EmptyQueue
      | h::t -> (h,t)
    let dequeue(q:’a queue) : ‘a queue = snd (deq q)
    let front(q:’a queue) : ‘a = fst (deq q)
end
```

enqueue takes time proportional to the length of the queue ☹

Dequeue runs in constant time ☺

An Alternative Implementation

```
module DoubleListQueue : QUEUE =
  struct
    type 'a queue = {front:'a list; rear:'a list}





    ...






end
```

In Pictures

abstraction → implementation

a, b, c, d, e → `{front=[a; b];rear=[e; d; c]}`

```
let q0 = empty           {front=[];rear=[]}
let q1 = enqueue 3 q0 {front=[];rear=[3]}
let q2 = enqueue 4 q1 {front=[];rear=[4;3]}
let q3 = enqueue 5 q2 {front=[];rear=[5;4;3]}
let q4 = dequeue q3    {front=[4;5];rear=[]}
let q5 = dequeue q4    {front=[5];rear=[]}
let q6 = enqueue 6 q5 {front=[5];rear=[6]}
let q7 = enqueue 7 q6 {front=[5];rear=[7;6]}
```

An Alternative Implementation

```
module DoubleListQueue : QUEUE =
  struct
    type 'a queue = {front:'a list;
                     rear:'a list}
    let empty() = {front=[]; rear=[]}

    let enqueue x q = {front=q.front; rear=x::q.rear}

    let is_empty q =
      match q.front, q.rear with
      | [], [] -> true
      | _, _ -> false

    ...
end
```

enqueue runs in constant time ☺

An Alternative Implementation

```
module DoubleListQueue : QUEUE =
  struct
    type 'a queue = {front:'a list;
                     rear:'a list}
    exception EmptyQueue

    let deq (q:'a queue) : 'a * 'a queue =
      match q.front with
      | h::t -> (h, {front=t; rear=q.rear})
      | [] -> match List.rev q.rear with
              | h::t -> (h, {front=t; rear=[]})
              | [] -> raise EmptyQueue

    let dequeue (q:'a queue) : 'a queue = snd(deq q)
    let front (q:'a queue) : 'a = fst(deq q)
  end
```

dequeue runs in *amortized* constant time ☺

How would we design an abstraction?

Think:

- what data do you want?
  - define some types for your data
- what operations on that data do you want?
  - define some types for your operations

Write some test cases:

- example data, operations

From this, we can derive a signature

- list the types
- list the operations with their types
- don't forget to provide enough operations that you can debug!

Then we can build an implementation

- when prototyping, build the simplest thing you can.
- later, we can swap in a more efficient implementation.
- (assuming we respect the abstraction barrier.)

Common Interfaces

The stack and queue interfaces are quite similar:

```
module type STACK =
  sig
    type 'a stack
    val empty : unit -> 'a stack
    val push  : int -> 'a stack -> 'a stack
    val is_empty : 'a stack -> bool
    exception EmptyStack
    val pop
    val top
  end
```

```
module type QUEUE =
  sig
    type 'a queue
    val empty : unit -> 'a queue
    val enqueue : 'a -> 'a queue -> 'a queue
    val is_empty : 'a queue -> bool
    exception EmptyQueue
    val dequeue : 'a queue -> 'a queue
    val front : 'a queue -> 'a
  end
```

It’s a good idea to factor out patterns

Stacks and Queues share common features.

Both can be considered “containers”

Create a reuseable container interface!

```
module type CONTAINER =
  sig
    type 'a t
    val empty : unit -> 'a t
    val insert : 'a -> 'a t -> 'a t
    val is_empty : 'a t -> bool
    exception Empty
    val remove : 'a t -> 'a t
    val first : 'a t -> 'a
  end
```

It's a good idea to factor out patterns

```
module type CONTAINER = sig ... end

module Queue : CONTAINER = struct ... end
module Stack : CONTAINER = struct ... end
```

```
module DepthFirstSearch : SEARCHER =
  struct
    type to_do : Graph.node Queue.t

  end
```

```
module BreadthFirstSearch : SEARCHER =
  struct
    type to_do : Graph.node Stack.t

  end
```

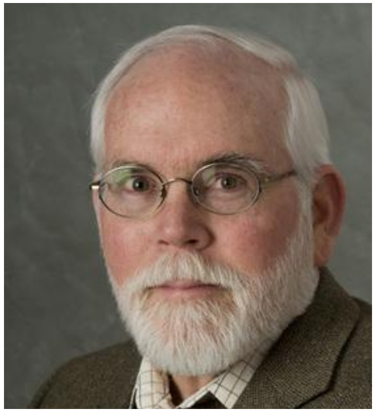
Still repeated code!

Breadth-first and depth-first search code is the same!

Just use different containers!

Need parameterized modules!

FUNCTORS



David MacQueen
Bell Laboratories 1983-2001
U. of Chicago 2001-2012

Designer of ML module system,
functors,
sharing constraints, etc.

Matrices

Suppose I ask you to write a generic package for matrices.

- e.g., matrix addition, matrix multiplication

The package should be *parameterized* by the element type.

- Matrix elements may be ints or floats or complex ...
- And the elements still have a collection of operations on them:
  - addition, multiplication, zero element, etc.

What we'll see:

- RING: a signature for matix elements
- MATRIX: a signature for operations on matrices
- DenseMatrix: a functor that will generate a MATRIX with a specific RING as an element type

Ring Signature

```
module type RING =
  sig
    type t
    val zero : t
    val one  : t
    val add  : t -> t -> t
    val mul  : t -> t -> t
  end
```

Some Rings

```
module IntRing =
  struct
    type t = int
    let zero = 0
    let one = 1
    let add x y= x + y
    let mul x y = x * y
  end
```

```
module FloatRing =
  struct
    type t = float
    let zero = 0.0
    let one = 1.0
    let add = (+.)
    let mul = ( *. )
  end
```

```
module BoolRing =
  struct
    type t = bool
    let zero = false
    let one = true
    let add x y= x || y
    let mul x y = x && y
  end
```

Matrix Signature

```
module type MATRIX =
  sig
    type elt
    type matrix
    val matrix_of_list : elt list list -> matrix
    val add : matrix -> matrix -> matrix
    val mul : matrix -> matrix -> matrix
   end
```

The DenseMatrix Functor

```
module DenseMatrix (R:RING) : (MATRIX with type elt = R.t) =
struct

  ...

end
```

The DenseMatrix Functor

```
module DenseMatrix (R:RING) : (MATRIX with type elt = R.t) =
struct

   ...

end
```

Argument R must be a RING

Result must be a MATRIX

Specify Result.elt = R.t

The DenseMatrix Functor

```
module DenseMatrix (R:RING) : (MATRIX with type elt = R.t) =
struct

  ...



end

module IntMatrix = DenseMatrix(IntRing)
module FloatMatrix = DenseMatrix(FloatRing)
module BoolMatrix = DenseMatrix(BoolRing)
```

Use DenseMatrix like it is a function from modules to modules

The Type of IntMatrix

```
module DenseMatrix (R:RING) : (MATRIX with type elt = R.t) =
struct ... end

module IntMatrix = DenseMatrix(IntRing)
```

What is the signature of IntMatrix?

The Type of IntMatrix

```
module DenseMatrix (R:RING) : (MATRIX with type elt = R.t) =
struct ... end

module IntMatrix = DenseMatrix(IntRing)
```

What is the signature of IntMatrix?

It depends on both the signatures of DenseMatrix and of it's argument IntRing

The Type of IntMatrix

```
module DenseMatrix (R:RING) : (MATRIX with type elt = R.t) =
struct ... end

module IntMatrix = DenseMatrix(IntRing)
```

```
module type MATRIX =
  sig
    type elt
    type matrix
    val matrix_of_list : elt list list -> matrix
    val add : matrix -> matrix -> matrix
    val mul : matrix -> matrix -> matrix
   end
```

\+

```
type elt = R.t
```

The Type of IntMatrix

```
module DenseMatrix (R:RING) : (MATRIX with type elt = R.t) =
struct ... end

module IntMatrix = DenseMatrix(IntRing)
```

```
module type MATRIX =
  sig
    type elt
    type matrix
    val matrix_of_list : elt list list -> matrix
    val add : matrix -> matrix -> matrix
    val mul : matrix -> matrix -> matrix
   end
```

\+

```
type elt = R.t
```

Recall:

```
module IntRing =
  struct
    type t = int
    let zero = 0
    ...
end
```

The Type of IntMatrix

```
module DenseMatrix (R:RING) : (MATRIX with type elt = R.t) =
struct ... end

module IntMatrix = DenseMatrix(IntRing)
```

```
module type MATRIX =
  sig
    type elt
    type matrix
    val matrix_of_list : elt list list -> matrix
    val add : matrix -> matrix -> matrix
    val mul : matrix -> matrix -> matrix
   end
```

\+

```
type elt = R.t
```

```
module IntRing =
  struct
    type t = int
    ...
  end
```

=

```
module type MATRIX =
  sig
    type elt = int
    type matrix
    ...
end
```

The DenseMatrix Functor

```
module DenseMatrix (R:RING) : (MATRIX with type elt = R.t) =
struct

  ...







end

module IntMatrix = DenseMatrix(IntRing)
module FloatMatrix = DenseMatrix(FloatRing)
module BoolMatrix = DenseMatrix(BoolRing)
```

redacted

The DenseMatrix Functor

```
module DenseMatrix (R:RING) : (MATRIX with type elt = R.t) =
struct

  ...

end
```

redacted

```
module type MATRIX =
  sig
    type elt
    type matrix

    val matrix_of_list :
          elt list list -> matrix

    val add : matrix -> matrix -> matrix
    val mul : matrix -> matrix -> matrix
  end
```

abstract = unknown!

nonexistent

```
module IntMatrix = DenseMatrix(IntRing)
module FloatMatrix = DenseMatrix(FloatRing)
module BoolMatrix = DenseMatrix(BoolRing)
```

The DenseMatrix Functor

```
module DenseMatrix (R:RING) : (MATRIX with type elt = R.t) =
struct

end
```

redacted

If the "with" clause is redacted then IntMatrix.elt is abstract -- we could never build a matrix because we could never generate an elt

```
module type MATRIX =
  sig
    type elt
    type matrix

    val matrix_of_list :
          elt list list -> matrix

    val add : matrix -> matrix -> matrix
    val mul : matrix -> matrix -> matrix
  end
```

abstract = unknown!

nonexistent

```
module IntMatrix = DenseMatrix(IntRing)
module FloatMatrix = DenseMatrix(FloatRing)
module BoolMatrix = DenseMatrix(BoolRing)
```

The DenseMatrix Functor

```
module DenseMatrix (R:RING) : (MATRIX with type elt = R.t) =
struct

  ...

end

module IntMatrix = DenseMatrix(IntRing)
module FloatMatrix = DenseMatrix(FloatRing)
module BoolMatrix = DenseMatrix(BoolRing)
```

sharing constraint

```
module type MATRIX =
  sig
    type elt = int
    type matrix

    val matrix_of_list :
          elt list list -> matrix

    val add : matrix -> matrix -> matrix
    val mul : matrix -> matrix -> matrix
  end
```

known to be int when R.t = int like when R = IntRing

list of list of ints

The DenseMatrix Functor

```
module DenseMatrix (R:RING) : (MATRIX with type elt = R.t) =
struct

end

module IntMatrix = DenseMatrix(IntRing)
module FloatMatrix = DenseMatrix(FloatRing)
module BoolMatrix = DenseMatrix(BoolRing)
```

sharing constraint

The "with" clause makes IntMatrix.elt equal to int -- we can build a matrix from any int list list

```
module type MATRIX =
  sig
    type elt = int
    type matrix

    val matrix_of_list :
          elt list list -> matrix

    val add : matrix -> matrix -> matrix
    val mul : matrix -> matrix -> matrix
  end
```

known to be int when R.t = int like when R = IntRing

list of list of ints

Matrix Functor

```
module DenseMatrix (R:RING) : (MATRIX with type elt = R.t) =
struct
  type elt = ...
  type matrix = ...
  let matrix_of_list = ...
  let add m1 m2 = ...
  let mul m1 m2 = ...
end
```

To define a functor, just write down a module as its body.

That module has to match the result signature (MATRIX).

This module may refer to the functor arguments, like R.

```
module IntMatrix = DenseMatrix(IntRing)
module FloatMatrix = DenseMatrix(FloatRing)
module BoolMatrix = DenseMatrix(BoolRing)
```

Matrix Functor

```
module DenseMatrix (R:RING) : (MATRIX with type elt = R.t) =
struct
  type elt = R.t
  type matrix = (elt list) list
  let matrix_of_list rows = rows
  let add m1 m2 =
    List.map (fun (r1,r2) ->
                List.map (fun (e1,e2) -> R.add e1 e2))
                  (List.combine r1 r2))
      (List.combine m1 m2)
  let mul m1 m2 = (* good exercise *)
end

module IntMatrix = DenseMatrix(IntRing)
module FloatMatrix = DenseMatrix(FloatRing)
module BoolMatrix = DenseMatrix(BoolRing)
```

Satisfies the sharing constraint

Can refer to functor argument *values* or argument *types*!

ANONYMOUS STRUCTURES

Another Example

```
module type UNSIGNED_BIGNUM =
sig
  type ubignum
  val fromInt : int -> ubignum
  val toInt : ubignum -> int
  val plus : ubignum -> ubignum -> ubignum
  val minus : ubignum -> ubignum -> ubignum
  val times : ubignum -> ubignum -> ubignum
  …
end
```

An Implementation

```
module My_UBignum_1000 : UNSIGNED_BIGNUM =
struct
  let base = 1000

  type ubignum = int list

  let toInt(b:ubignum):int = …

  let plus(b1:ubignum)(b2:ubignum):ubignum = …

  let minus(b1:ubignum)(b2:ubignum):ubignum = …

  let times(b1:ubignum)(b2:ubignum):ubignum = …
  …
end
```

What if we want to change the base? Binary? Hex? 2^32? 2^64?

Another Functor Example

```
module type BASE =
sig
  val base : int
end

module UbignumGenerator(Base:BASE) : UNSIGNED_BIGNUM =
struct
  type ubignum = int list
  let toInt(b:ubignum):int =
    List.fold_left (fun a c -> c*Base.base + a) 0 b   …
end

module Ubignum_10 =
  UbignumGenerator(struct let base = 10 end) ;;

module Ubignum_2 =
  UbignumGenerator(struct let base = 2 end) ;;
```

Anonymous structures

SIGNATURE SUBTYPING

Subtyping

A module matches any interface as long as it provides *at least* the definitions (of the right type) specified in the interface.

But as we saw earlier, the module can have more stuff.

– e.g., the deq function in the Queue modules

Basic principle of subtyping for modules:

– wherever you are expecting a module with signature S, you can use a module with signature S', as long as all of the stuff in S appears in S'.
– That is, S' is a bigger interface.

Groups versus Rings

```
module type GROUP =
  sig
    type t
    val zero : t
    val add : t -> t -> t
  end
module type RING =
  sig
    type t
    val zero : t
    val one  : t
    val add  : t -> t -> t
    val mul  : t -> t -> t
  end
module IntGroup : GROUP = IntRing
module FloatGroup : GROUP = FloatRing
module BoolGroup : GROUP = BoolRing
```

Groups versus Rings

```
module type GROUP =
  sig
    type t
    val zero : t
    val add : t -> t -> t
  end
module type RING =
  sig
    type t
    val zero : t
    val one  : t
    val add  : t -> t -> t
    val mul  : t -> t -> t
  end
module IntGroup : GROUP = IntRing
module FloatGroup : GROUP = FloatRing
module BoolGroup : GROUP = BoolRing
```

RING is a sub-type of GROUP.

Groups versus Rings

```
module type GROUP =
  sig
    type t
    val zero : t
    val add : t -> t -> t
  end
module type RING =
  sig
    type t
    val zero : t
    val one  : t
    val add  : t -> t -> t
    val mul  : t -> t -> t
  end
module IntGroup : GROUP = IntRing
module FloatGroup : GROUP = FloatRing
module BoolGroup : GROUP = BoolRing
```

There are *more* modules matching the GROUP interface than the RING one.

Groups versus Rings

```
module type GROUP =
  sig
    type t
    val zero : t
    val add : t -> t -> t
  end
module type RING =
  sig
    type t
    val zero : t
    val one  : t
    val add  : t -> t -> t
    val mul  : t -> t -> t
  end
module IntGroup : GROUP = IntRing
module FloatGroup : GROUP = FloatRing
module BoolGroup : GROUP = BoolRing
```

Any module expecting a GROUP can be passed a RING.

Groups versus Rings

```
module type GROUP =
  sig
    type t
    val zero : t
    val add : t -> t -> t
  end
module type RING =
  sig
    include GROUP
    val one  : t
    val mul  : t -> t -> t
  end
module IntGroup : GROUP = IntRing
module FloatGroup : GROUP = FloatRing
module BoolGroup : GROUP = BoolRing
```

The **include** primitive is like cutting-and-pasting the signature's content here.

Groups versus Rings

```
module type GROUP =
  sig
    type t
    val zero : t
    val add : t -> t -> t
  end
module type RING =
  sig
    include GROUP
    val one  : t
    val mul  : t -> t -> t
  end
module IntGroup : GROUP = IntRing
module FloatGroup : GROUP = FloatRing
module BoolGroup : GROUP = BoolRing
```

That *ensures* we will be a sub-type of the included signature.

MODULE EVALUATION

Evaluating the contents of a module

A structure is a series of declarations

- How does one evaluate a type declaration? We'll ignore it.
- How does one evaluate a let declaration?

let x = e

evaluate the expression e
bind the value to x

How does one evaluate an entire structure?

- evaluate each declaration in order from first to last

Evaluating the contents of a module

main.ml

```
let x = 326

let main () =
   Printf.printf “Hello COS %d\n” x

let foo =
   Printf.printf “Byeee!\n”

let _ =
    main ()
```

Evaluating the contents of a module

main.ml

```
let x = 326

let main () =
  Printf.printf “Hello COS %d\n” x

let foo =
  Printf.printf “Byeee!\n”

let _ =
   main ()
```

Step 1:

evaluate the 1st declaration

but the RHS (326)
is already a value so there’s
nothing to do except
remember that x is bound
to the integer 326

Evaluating the contents of a module

main.ml

```
let x = 326

let main () =
  Printf.printf “Hello COS %d\n” x

let foo =
  Printf.printf “Byeee!\n”

let _ =
   main ()
```

Step 2:
evaluate the 2nd declaration
this is slightly trickier:

let main () = ...

really declares a function.
It’s equivalent to:

let main = fun () -> ...

“fun () -> ...” is already
a value, like 326.
So there’s nothing to do again.

Evaluating the contents of a module

main.ml

```
let x = 326

let main () =
  Printf.printf “Hello COS %d\n” x

let foo =
  Printf.printf “Byeee!\n”

let _ =
   main ()
```

Step 3:
evaluate the 3rd declaration

let foo = ...

evaluation of this expression has an effect – it prints out “Byeee!\n” to the terminal.

the resulting value is () which is bound to foo

Evaluating the contents of a module

main.ml

```
let x = 326

let main () =
  Printf.printf “Hello COS %d\n” x

let foo =
  Printf.printf “Byeee!\n”

let _ =
   main ()
```

Step 4:
evaluate the 4th declaration

let _ = ...

evaluation main ()
causes another effect.

“Hello ...” is printed

the resulting value is () again.
the “_” indicates we don’t
care to bind () to any variable

A Variation

main.ml

```
let x = 326

let main =
  (fun () ->
    Printf.printf “Hello COS %d\n” x)

let foo =
   Printf.printf “Byeee!\n”

let _ =
    main ()
```

This evaluates exactly the same way

We just replaced

`let main () = ...`

with the equivalent

`let main = fun () -> ...`

A Variation

main.ml

```
let x = 326

let main =
  Printf.printf "Hello COS %d\n" x;
  (fun () -> ())

let foo =
   Printf.printf "Byeee!\n"

let _ =
    main ()
```

This rewrite does something different.

On the 2nd step, it prints because that's what evaluating this expression does:

```
Printf.printf "Hello COS %d\n" x;
(fun () -> ())
```

The result of the expression is:

```
fun () -> ()
```

which is bound to main.
This is a pretty silly function.

A Variation

main.ml

```
module C326 =
struct
  let x = 326

  let main =
    Printf.printf “Hello COS %d\n” x;
    (fun () -> ())

  let foo = Printf.printf “Byeee!\n”

  let _ =  main ()
end

let _ =
  Printf.printf "Done\n"
```

Now what happens?

A Variation

main.ml

```
module C326 =
struct
  let x = 326

  let main =
    Printf.printf “Hello COS %d\n” x;
    (fun () -> ())

  let foo = Printf.printf “Byeee!\n”

  let _ =  main ()
end

let done =
  Printf.printf "Done\n"
```

Now what happens?

The entire file contains 2 decls:

- module C326 = ...
- let done = ...

We execute both of them in order.

A Variation

main.ml

```
module C326 =
struct
  let x = 326

  let main =
    Printf.printf “Hello COS %d\n” x;
    (fun () -> ())

  let foo = Printf.printf “Byeee!\n”

  let _ =  main ()
end

let done =
  Printf.printf "Done\n"
```

Now what happens?

The entire file contains 2 decls:

- module C326 = ...
- let done = ...

We execute both of them in order.

Executing the module declaration has the effect of executing every declaration within it in order.

Executing let done = ... is as before

A Variation

main.ml

```
module C326 =
struct
  exception Unimplemented
  let x = raise Unimplemented

  let main =
    Printf.printf “Hello COS %d\n” x;
    (fun () -> ())

  let foo = Printf.printf “Byeee!\n”

  let _ =  main ()
end

let done =
  Printf.printf "Done\n"
```

Now what happens?

A Variation

main.ml

```
module C326 =
struct
  exception Unimplemented
  let x = raise Unimplemented

  let main =
    Printf.printf “Hello COS %d\n” x;
    (fun () -> ())

  let foo = Printf.printf “Byeee!\n”

  let _ =  main ()
end

let done =
  Printf.printf "Done\n"
```

Now what happens?

The entire file contains 2 decls:

- module C326 = ...
- let done = ...

We execute both of them in order.

Executing the module declaration has the effect of executing every declaration within it in order.

The first declaration within it raises an exception which is not caught! That is the only result.

A Variation

main.ml

```
module type S =
sig
  type t = int
  val x : t
end

module F (M:S) : S =
struct
  let wow = Printf.printf “%d\n” M.x
  let t = M.t
  let x = M.x
end

let done = Printf.printf "Done\n"
```

Now what happens?

The entire file contains 2 decls:

- module type = ...
- module F (M:S) : S = ...
- let done = ...

A Variation

main.ml

```
module type S =
sig
  type t = int
  val x : t
end

module F (M:S) : S =
struct
  let wow = Printf.printf “%d\n” M.x
  let t = M.t
  let x = M.x
end

let done = Printf.printf "Done\n"
```

The signature declaration has no (run-time) effect.

The functor declaration is like declaring a function value.

The body of the functor is not executed until it is applied.

The functor is not applied here so M.x is not printed.

Only “Done\n” is printed.

A Variation

main.ml

```
module type S = sig ... end

module F (M:S) : S =
struct
  let wow = Printf.printf “%d\n” M.x
  let t = M.t
  let x = M.x
end

let module M1 = F (
      struct
        type t = int
        val x = 3
      end)

let done = Printf.printf "Done\n"
```

What happens now?

A Variation

main.ml

```
module type S = sig ... end

module F (M:S) : S =
struct
  let wow = Printf.printf “%d\n” M.x
  let t = M.t
  let x = M.x
end

let module M1 = F (
      struct
        type t = int
        val x = 3
      end)

let done = Printf.printf "Done\n"
```

What happens now?

When M1 is declared,
F is applied to an argument.

This creates a new structure and its components are executed.

This has the effect of printing 3.

SUMMARY

Key Points

OCaml's linguistic mechanisms include:

- *signatures* (interfaces)
- *structures* (implementations)
- *functors* (functions from modules to modules)

We can use the module system

- provides support for *name-spaces*
- *hiding information* (types, local value definitions)
- *code reuse* (via functors, reuseable interfaces, reuseable modules)

Information hiding allows design in terms of *abstract* types and algorithms.

- think "sets" not "lists" or "arrays" or "trees"
- think "document" not "strings"
- the less you reveal, the easier it is to replace an implementation
- use linguistic mechanisms to implement information hiding
    - invariants written down as comments are easy to violate
    - use the type checker to guarantee you have strong protections in place

Wrap up and Summary

It is often tempting to break the abstraction barrier.

- e.g., during development, you want to print out a set, so you just call a convenient function you have lying around for iterating over lists and printing them out.

But the barrier supports future change of implementations.

- e.g., moving from unsorted invariant to sorted invariant.
- or from lists to balanced trees.

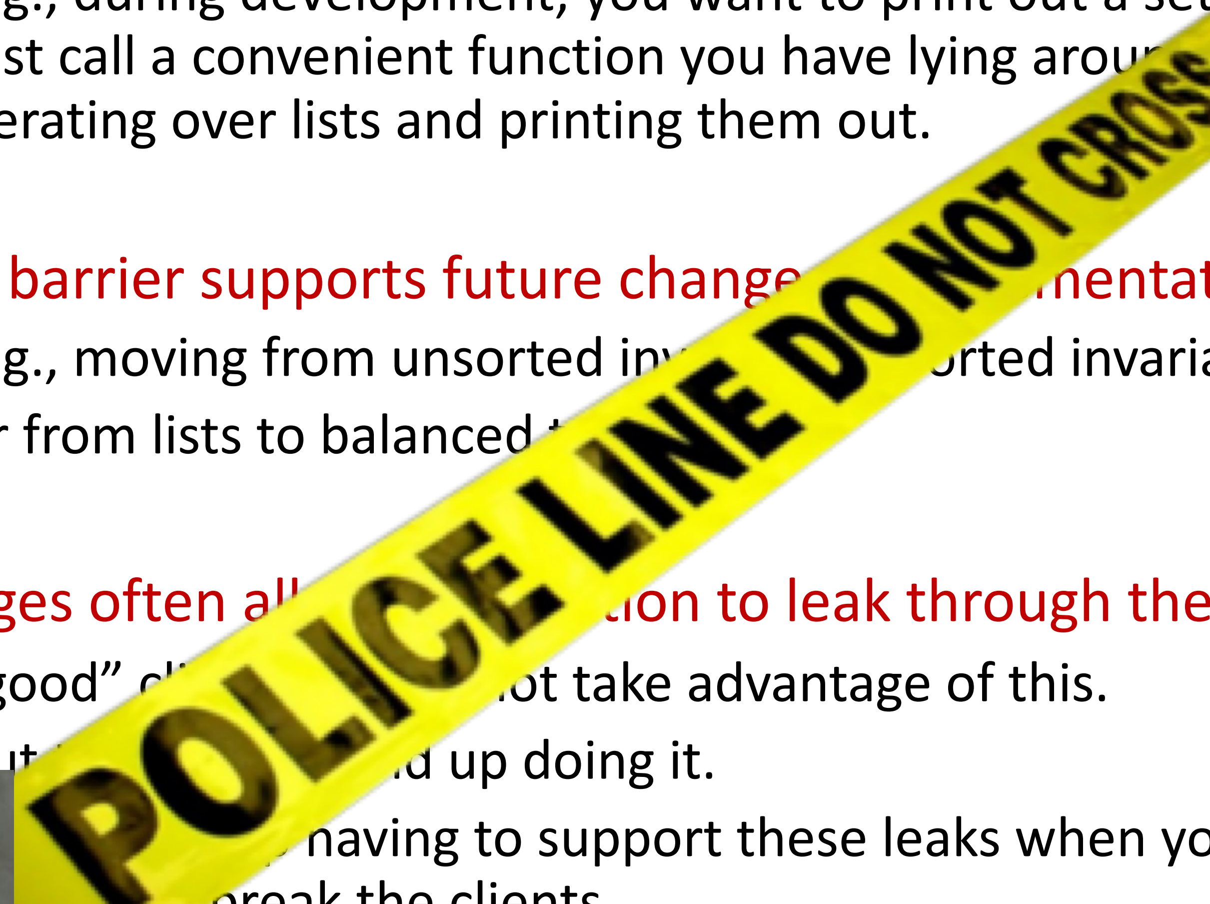
Languages often allow information to leak through the barrier.

- "good" clients should not take advantage of this.
- but they always end up doing it.
- so you end up having to support these leaks when you upgrade, else you'll break the clients.

Wrap up and Summary

It is often tempting to break the abstraction barrier.

– e.g., during development, you want to print out a set, just call a convenient function you have lying arou… iterating over lists and printing them out.



But the barrier supports future change… …nentat…

– e.g., moving from unsorted in… …orted invariant.

– or from lists to balanced …

Languages often al… …tion to leak through the barrier.

– "good" c… …ot take advantage of this.

– but … …d up doing it.

– … having to support these leaks when you upgrade, … break the clients.