Programming Languages and Techniques (CIS120)

Lecture 12

Partiality, Sequencing

Chapters 11, 12

Midterm 1

- Friday, September 27th
- Coverage: up to Monday, Sept. 23 (Chs. 1-10)
- Time: During lecture (001 @ 11am, 002 @ noon)
  Last names: A – L Leidy Labs 10
  Last names: M – Z Stitler (STIT) B6
- Review Session: TONIGHT 6:00-8:00pm Towne 100
- Review Material:
  - old exams on the web site lecture schedule
- Makeup exam
  - Monday, Sept. 30th
  - sign up form on the web site

Announcements

- Dr. Sheth will have extra office hours Thursday 4:00-6:00PM in Levine 264
- Homework 4
  - Available soon after exam
  - Due: Tuesday, Oct. 8th

Signature: Finite Map

```
module type MAP = sig

  type ('k,'v) map

  val empty   : ('k,'v) map
  val add     : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
  val remove  : 'k       -> ('k,'v) map -> ('k,'v) map
  val mem     : 'k -> ('k,'v) map -> bool
  val get     : 'k -> ('k,'v) map -> 'v
  val entries : ('k,'v) map -> ('k * 'v) list
  val equals  : ('k,'v) map -> ('k,'v) map -> bool

end
```

Properties of Finite Maps

For any finite map m, key k, and value v:

1. `get k (add k v m) = v`
2. If `k1 <> k2` and `get k1 m = v1` then
   `get k1 (add k2 v2 m) = v1`
3. if `mem k m = true` then
   there is a v such that `get k m = v`
4. If `mem k m = false` then
   `get k m = v` fails
5. `mem k (add k v m) = true`
6. `mem k (remove k m) = false`

And others…

Completing module implementation

finiteMap.ml

Implementation: Ordered Lists

```
module Assoc : MAP = struct

  (* Represent a finite map as a list of pairs. *)
  (* Representation invariant:                     *)
  (*   - no duplicate keys (helps get, remove)     *)
  (*   - keys are sorted (helps equals, helps get) *)

type ('k,'v) map = ('k * 'v) list

  let empty : ('k,'v) map = []

  let rec mem (key:'k) (m : ('k,'v) map) : bool =
    begin match m with
    | [] -> false
    | (k,v)::rest ->
      (key >= k) &&
         ((key = k) || (mem key rest))
    end

;; run_test "mem test" (fun () -> mem "b" [("a",3); ("b",4)])
```

Implementation: Ordered Lists

```
let rec get (key:'k) (m : ('k,'v) map) : 'v =
  begin match m with
  | [] -> failwith "key not found"
  | (k,v)::rest ->
    if key < k then failwith "key not found"
    else if key = k then v
    else get key rest
  end

let rec remove (key:'k) (m : ('k,'v) map) : ('k,'v) map =
  begin match m with
  | [] -> []
  | (k,v)::rest ->
    if key < k then m
    else if key = k then rest
    else (k,v)::remove key rest
  end
```

Abstract types

BIG IDEA: Hide the *concrete representation* of a type behind an *abstract interface* to preserve invariants

- The interface **restricts** how other parts of the program can interact with the data
  - Type checking ensures that the **only** way to create a set is with the operations in the interface
  - If all operations preserve invariants, then all sets in the program must satisfy invariants
  - Example: all BST-implemented sets must satisfy the BST invariant, therefore the lookup function can assume that its input satisfies the invariant
- Benefits:
  - **Safety**: The other parts of the program can't cause bugs in the set implementation
  - **Modularity**: It is possible to change the implementation without changing the rest of the program

Summary: Abstract Types

- Different programming languages have different ways of letting you define abstract types
- At a minimum, this means providing:
  - A way to specify (write down) an interface
  - A means of hiding implementation details (*encapsulation*)
- In OCaml:
  - Interfaces are specified using a *signature* or *interface*
  - Encapsulation is achieved because the interface can *omit* information
    - type definitions
    - names and types of auxiliary functions
  - Clients *cannot* mention values or types not named in the interface

Dealing with Partiality*

*A function is said to be *partial* if it is not defined for all inputs.

Which of these is a function that calculates the maximum value in a (generic) list?

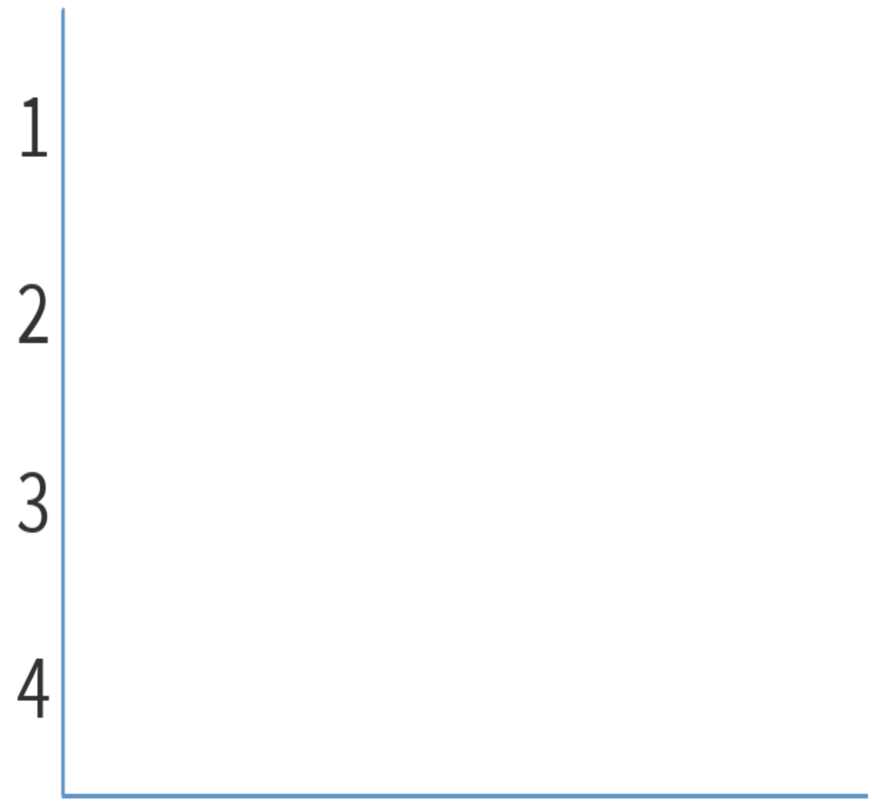
Which of these is a function that calculates the maximum value in a (generic) list:

1.
```
let rec list_max (l:'a list) : ’a =
  begin match l with
  | [] -> []
  | h :: t -> max h (list_max t)
  end
```

2.
```
let rec list_max (l:'a list) : ’a =
  fold max 0 l
```

3.
```
let rec list_max (l:’a list) : ‘a =
  begin match l with
  | h :: t -> max h (list_max t)
  end
```

4. None of the above



Which of these is a function that calculates the maximum value in a (generic) list:

1.

```
let rec list_max (l:'a list) : ’a =
  begin match l with
  | [] -> []
  | h :: t -> max h (list_max t)
  end
```

2.

```
let rec list_max (l:'a list) : ’a =
  fold max 0 l
```

3.

```
let rec list_max (l:’a list) : ‘a =
  begin match l with
  | h :: t -> max h (list_max t)
  end
```

4. None of the above

Answer: 4

Quiz answer

- list_max isn't defined for the empty list!

```
let rec list_max (l:'a list) : 'a =
  begin match l with
    | [] -> failwith "empty list"
    | [h] -> h
    | h::t -> max h (list_max t)
  end
```

Client of list_max

```
(* string_of_max calls list_max *)
let string_of_max (x:int list) : string =
  string_of_int (list_max x)
```

- Oops! `string_of_max` will fail if given `[]`
- Not so easy to debug if `string_of_max` is written by one person and `list_max` is written by another.
- Interface of list_max is not very informative

```
val list_max : int list -> int
```

Solutions to Partiality: Option 1

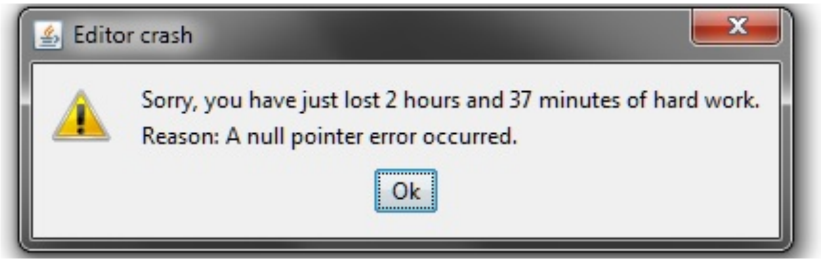
- Abort the program:

```
failwith "an error message"
```

  - Whenever it is called, `failwith` halts the program and reports the error message it is given.
- This solution is appropriate whenever you *know* that a certain case is impossible
  - The compiler isn't smart enough to figure out that the case is impossible…
  - Often happens when there is an invariant on a data structure
  - `failwith` is also useful to "stub out" unimplemented parts of your program.
- Languages (e.g. OCaml, Java) support *exception handling facilities* to let programs recover from such failures.
  - We'll talk about these when we get to Java

Solutions to Partiality: Option 2

- Return a *default* or *error value*
  - e.g. define **`list_max []`** to be **`-1`**
  - Error codes used often in C programs
  - `null` used often in Java



- But…
  - What if -1 (or whatever default you choose) really *is* the maximum value?
  - Can lead to many bugs if the default isn't handled properly by the callers.
  - *IMPOSSIBLE* to implement generically!
    - No way to generically create a sensible default value for every possible type
  - Sir Tony Hoare, Turing Award winner and inventor of `null` calls it his "*billion dollar mistake*"!
- *Defaults should be avoided if possible*

Optional values

Solutions to Partiality: Option 3

Option Types

- Define a generic datatype of *optional values*:

```
type 'a option =
   | None
   | Some of 'a
```

- A “partial” function returns an option

```
let list_max (l:list) : int option = …
```

- Contrast this with “null”, a “legal” return value of any type
  - caller can accidentally forget to check whether null was used; results in NullPointerExceptions or crashes
- Modern language designs (e.g. Apple's Swift, Mozilla's Rust) distinguish between the type String (definitely not null) and String? (optional string)

Example: list_max

- A function that returns the maximum value of a list as an option (None if the list is empty)

```
let list_max (l:'a list) : 'a option =
  begin match l with
    | [] -> None
    | x::tl -> Some (fold max x tl)
  end
```

Revised client of list_max

```
(* string_of_max calls list_max *)
let string_of_max (l:int list) : string =
  begin match (list_max l) with
  | None -> “no maximum”
  | Some m -> string_of_int m
  end
```

- `string_of_max` will never fail

- The type of list_max makes it explicit that a *client* must check for partiality.

```
val list_max : int list -> int option
```

What is the type of this function?

```
let head (x: ______) : ______ =
  begin match x with
  | [] -> None
  | h :: t -> Some h
  end
```

- 'a list -> 'a
- 'a list -> 'a list
- 'a list -> 'b option
- 'a list -> 'a option
- None of the above

Start the presentation to see live content. Still no live content? Install the app or get help at **PollEv.com/app**

Total Results

Which of these is a function that calculates the maximum value in a (generic) list?

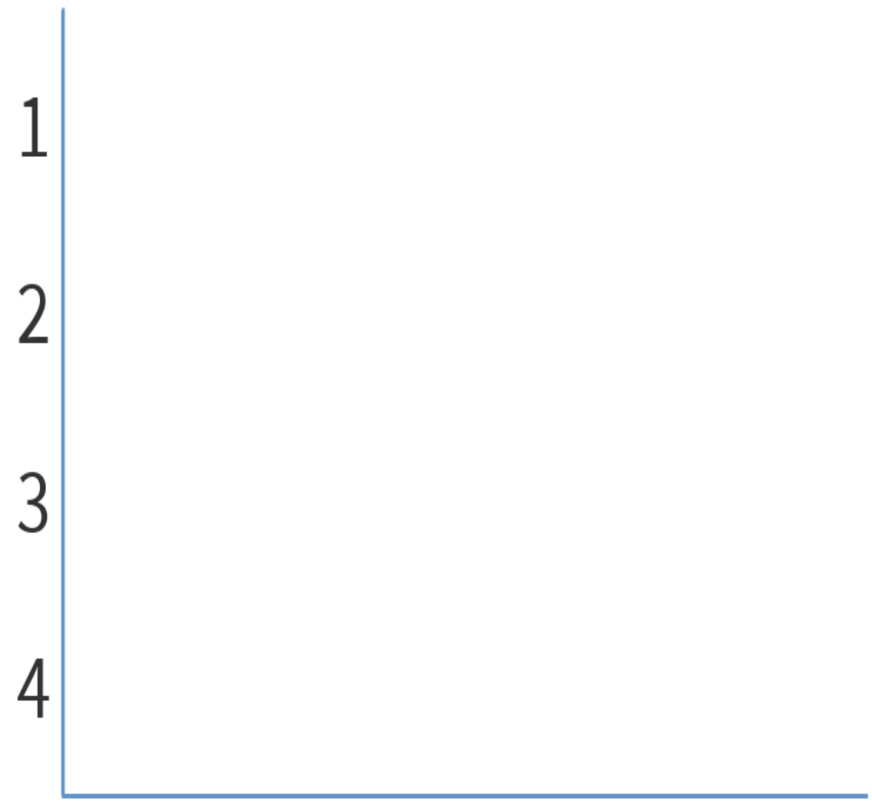
Which of these is a function that calculates the maximum value in a (generic) list:

1.
```
let rec list_max (l:'a list) : ’a =
  begin match l with
  | [] -> []
  | h :: t -> max h (list_max t)
  end
```

2.
```
let rec list_max (l:'a list) : ’a =
  fold max 0 l
```

3.
```
let rec list_max (l:’a list) : ‘a =
  begin match l with
  | h :: t -> max h (list_max t)
  end
```

4. None of the above



Start the presentation to see live content. Still no live content? Install the app or get help at **PollEv.com/app**

Total Results

What is the type of this function?

```
let head (x: ______) : ______  =
  begin match x with
  | [] -> None
  | h :: t -> Some h
  end
```

1. ‘a list -> ‘a

2. ‘a list -> ‘a list

3. ‘a list -> ‘b option

4. ‘a list -> ‘a option

5. None of the above

Answer: 4

What is the value of this expression?

```
let head (x: 'a list) : 'a option  =
   begin match x with
   | [] -> None
   | h :: t -> Some h
   end in

[ head [1];  head [] ]
```

- [1; 0]
- 1
- [Some 1; None]
- [None; None]
- None of the above

Start the presentation to see live content. Still no live content? Install the app or get help at **PollEv.com/app**

Total Results

What is the value of this expression?

```
let head (x: 'a list) : 'a option  =
   begin match x with
   | [] -> None
   | h :: t -> Some h
   end in

[ head [1];  head [] ]
```

1. [1 ; 0]
2. 1
3. [Some 1; None]
4. [None; None]
5. None of the above

Answer: 3

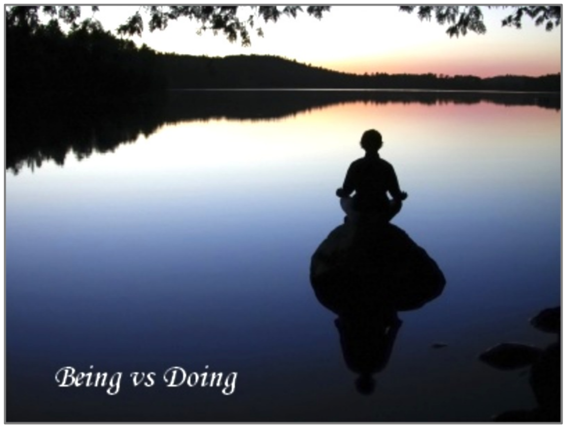
Revising the MAP interface

```
module type MAP = sig

  type ('k,'v) map

  val empty   : ('k,'v) map
  val add     : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
  val remove  : 'k       -> ('k,'v) map -> ('k,'v) map
  val mem     : 'k -> ('k,'v) map -> bool
  val get     : 'k -> ('k,'v) map -> 'v option
  val entries : ('k,'v) map -> ('k * 'v) list
  val equals  : ('k,'v) map -> ('k,'v) map -> bool

end
```

get returns an optional 'v.
Now its type isn't a lie!

Commands, Sequencing and Unit

What is the type of print_string?



Sequencing Commands and Expressions

We can *sequence* commands inside expressions using ';'

- unlike in C, Java, etc., ';' doesn't terminate a statement it *separates* a command from an expression

```
let f (x:int) : int =
  print_string "f called with ";
  print_string (string_of_int x);
  x + x
```

do *not* use ';' here!

note the use of ';' here

The distinction between commands & expressions is artificial.

- `print_string` is a function of type: `string -> unit`
- Commands are actually just expressions of type: `unit`

unit: the trivial type

- Similar to "void" in Java or C
- For functions that don't take any arguments

```
let f () : int = 3
let y : int =  f ()
```

```
val f : unit -> int
val y : int
```

- Also for functions that don't return anything, such as testing and printing functions a.k.a *commands*:

```
(* run_test : string -> (unit -> bool) -> unit *)
;; run_test “TestName” test

(* print_string : string -> unit *)
;; print_string “Hello, world!”
```

unit: the boring type

- *Actually, () is a value just like any other value (a 0-ary tuple)*
- For functions that don't take any interesting arguments

```
let f () : int = 3
let y : int =  f ()
```

```
val f : unit -> int
val y : int
```

- Also for functions that don't return anything interesting, such as testing and printing functions a.k.a *commands*:

```
(* run_test : string -> (unit -> bool) -> unit *)
;; run_test "TestName" test

(* print_string : string -> unit *)
;; print_string "Hello, world!"
```

unit: the first-class type

- Can define values of type unit

```
let x : unit = ()
```

```
val x : unit
```

- Can pattern match unit (even in function definitions)

```
let z = begin match x with
  | () -> 4
end
```

```
fun () -> 3
```

- Is the result of an implicit else branch:

```
;; if z <> 4 then
     failwith "oops"
```

=

```
;; if z <> 4 then
     failwith "oops"
   else ()
```

Sequencing Commands and Expressions

- Expressions of type unit are useful because of their *side effects* – they "*do*" stuff
  – e.g. printing, changing the value of mutable state

```
let f (x:int) : int =
  print_string "f called with ";
  print_string (string_of_int x);
  x + x
```

do *not* use ';' here!

note the use of ';' here

- We can think of ';' as an infix function of type:

```
unit -> 'a -> 'a
```

What is the type of f in the following program?

```
let f (x:int) =
  print_int (x + x)
```

- unit -> int
- unit -> unit
- int -> unit
- int -> int
- f is ill typed

Start the presentation to see live content. Still no live content? Install the app or get help at **PollEv.com/app**

Total Results

What is the type of `f` in the following program:

```
let f (x:int) =
   print_int (x + x)
```

1. `unit -> int`
2. `unit -> unit`
3. `int -> unit`
4. `int -> int`
5. `f is ill typed`

Answer: 3

What is the type of f in the following program?

```
let f (x:int) =
  (print_int x);
  (x + x)
```

unit -> int

unit -> unit

int -> unit

int -> int

f is ill typed

Start the presentation to see live content. Still no live content? Install the app or get help at **PollEv.com/app**

Total Results

What is the type of `f` in the following program:

```
let f (x:int) =
   (print_int x);
   (x + x)
```

1. unit -> int
2. unit -> unit
3. int -> unit
4. int -> int
5. f is ill typed

Answer: 4