

Programming Languages and Techniques (CIS120)

Lecture 11

Feb 12, 2014

Options and Unit

Announcements

- Homework 4 available on the web today
 - due Tuesday, Feb 17th
 - n-body physics simulation
 - start early; see Piazza for discussions
- Read Chapters 11, 12 & 13 (they're short)
- Midterm 1
 - Scheduled *in class* on **Friday, Feb 21st**
 - Review session Wednesday, Feb 19th , 7-9PM in Levine 101

List processing

The fold design pattern

Functions as Data

- We've seen a number of ways in which functions can be treated as data in OCaml
- Present-day programming practice offers many more examples at the “small scale”:
 - objects bundle “functions” (a.k.a. methods) with data
 - iterators (“cursors” for walking over data structures)
 - event listeners (in GUIs)
 - etc.
- The idiom is useful at the “large scale”: Google's MapReduce
 - Framework for mapping across sets of key-value pairs
 - Then “reducing” the results per key of the map
 - Easily distributed to 10,000 machines to execute in parallel!

Refactoring code, again

- Is there a pattern in the definition of these two functions?

```
let rec exists (l : bool list) : bool =  
  begin match l with  
  | [] -> false  
  | h :: t -> h || exists t  
  end
```

base case:
Simple answer when
the list is empty

```
let rec acid_length (l : acid list) : int =  
  begin match l with  
  | [] -> 0  
  | x :: t -> 1 + acid_length t  
  end
```

combine step:
Do something with
the head of the list
and the recursive call

- Can we factor out that pattern using first-class functions?

List Fold

```
let rec fold (combine: 'a -> 'b -> 'b)
             (base:'b) (l : 'a list) : 'b =
  begin match l with
  | [] -> base
  | x :: t -> combine x (fold combine base t)
  end

let acid_length (l : acid list) : int =
  fold (fun (x:acid) (acc:int) -> 1 + acc) 0 l
let exists (l : bool list) : bool =
  fold (fun (x:bool) (acc:bool) -> x || acc) false l
```

- Fold (aka Reduce)
 - Another foundational function for programming with lists
 - Captures the pattern of recursion over lists
 - Also part of OCaml standard library (List.fold_right)
 - Similar operations for other recursive datatypes (fold_tree)

How would you rewrite this function

```
let rec sum (x : int list) : int =  
  begin match x with  
  | [] -> 0  
  | h :: t -> h + sum t  
  end
```

using fold? What should be the arguments for base and combine?

1. combine is: (fun (h:int) (acc:int) -> acc + 1)
base is: 0
2. combine is: (fun (h:int) (acc:int) -> h + acc)
base is: 0
3. combine is: (fun (h:int) (acc:int) -> h + acc)
base is: 1
4. sum can't be written by with fold.

How would you rewrite this function

```
let rec reverse (x : int list) : int list =  
  begin match x with  
    | [] -> []  
    | h :: t -> reverse t @ [h]  
  end
```

using fold? What should be the arguments for base and combine?

1. combine is: `(fun (h:int) (acc:int list) -> h :: acc)`
base is: `0`
2. combine is: `(fun (h:int) (acc:int list) -> acc @ [h])`
base is: `0`
3. combine is: `(fun (h:int) (acc:int list) -> acc @ [h])`
base is: `[]`
4. reverse can't be written by with `fold`.

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

1.

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

2.

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

3.

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

4. None of the above

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
```

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

Dealing with Partiality

Option Types

Partial Functions

- Sometimes functions aren't defined for all inputs:
 - `tree_max` from the BST implementation isn't defined for empty trees
 - integer division by 0
 - `Map.find k m` when the key `k` isn't in the finite map `m`
- We have seen how to deal with partiality using `failwith`, but `failwith` aborts the program
- Can we do better?
- Hint: we already have all the technology we need.

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 value, a “legal” return value of any type
 - caller can accidentally forget to check whether null was used; results in NullPointerExceptions or crashes
 - Sir Tony Hoare, Turing Award winner and inventor of “null” calls it his “*billion dollar mistake*”!

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
```

1. 'a list -> 'a
2. 'a list -> 'a list
3. 'a list -> 'b option
4. 'a list -> 'a option
5. None of the above

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]]
```

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

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

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.*
- 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 = ()
```

```
val x : unit
```

- Can pattern match unit (even in function definitions)

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

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

- Is the implicit else branch:

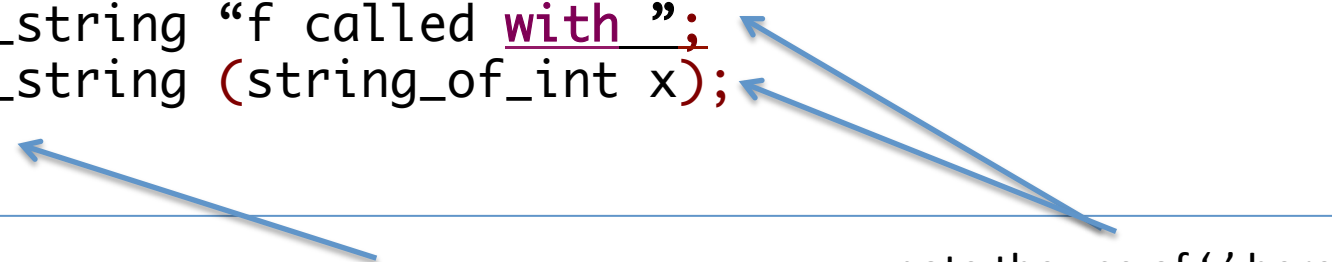
```
;; if z <> 4 then  
  failwith "oops"
```

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

Sequencing Commands and Expressions

- Expressions of type `unit` are useful because of their *side effects* (e.g. printing)
- We can *sequence* those effects 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

- 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
```

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

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

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

Imperative Programming

Course Overview

- Declarative programming
 - *persistent* data structures
 - *recursion* is main control structure
 - heavy use of functions as data
 - Imperative programming
 - *mutable* data structures (that can be modified “in place”)
 - *iteration* is main control structure
 - Object-oriented programming
 - pervasive “abstraction by default”
 - mutable data structures / iteration
 - heavy use of functions (objects) as data
- We are here.
Midterm 1 covers material up to this point.
-