CISS445 Lecture 12: OCAML Part 2

Yihsiang Liow

January 6, 2020

Table of contents I

Input/output I

- You have already seen
  - `print_string`, `print_int`, ...
  - `Printf.printf`
- What about output to file?

```
let f = open_out "test.txt";; (* output channel *)
output f "hello world" 2 8;;  (* What's 2 8 for? *)
output_string f "tada!";;
close_out f;;                 (* close file *)
```

- stdout is the (obvious) output channel:

```
output stdout "hello world" 2 8;;
```

Input/output I

- Try this:

```
let s = read_line ();;
let i = read_int ();;
```

- Exercise. What about input for float, bool, char?
- To read from file:

```
let f = open_in test.txt;;
let s = input_line f;;
close_in f;;
```

Evaluation of functions I

- A value is called a **first-class value** if it can be passed as an argument to a function and can be returned from a function call.
- In all functional languages, functions are first-class values.

Environment I

- In order for function to be first-class values, need to extend the simple concept of environments to include function.
- Suppose you have the following sequence of expressions:

  ```
  ...
  let x = 1;;           (* ρ1 = {x->1, ...} *)
  let f x = x * x;;     (* ρ2 = {f->fv, x->1, ...} *)
  ```

  where

  ```
  vf = <x -> x * x, ρ1>
  ```

- The above declaration of `twice` is syntactic sugar for

  ```
  let twice = fun f -> (fun x -> f(f x))
  ```

Environment II

Note that `fun f -> (fun x -> f(f x))` is considered a value but not "`twice f x = f(f x)`".

Evaluation of functions I

- STEP 1:

twice is a name. Need to substitute.

```
twice (fun x -> x * x) 42
(fun f -> (fun x -> f(f x))) (fun x -> x * x) 42
```

Evaluation of functions II

- STEP 2:

Already a value. No need to substitute and evaluate.

```
twice (fun x -> x * x) 42

(fun f -> (fun x -> f(f x))) (fun x -> x * x) 42
```

Evaluation of functions III

- STEP 3:

```
twice (fun x -> x * x) 42

(fun f -> (fun x -> f(f x))) (fun x -> x * x) 42
```

Evaluation of functions IV

- STEP 4:

```
twice (fun x -> x * x) 42
(fun f -> (fun x -> f(f x))) (fun x -> x * x) 42
(fun x -> (fun x -> x * x) ((fun x -> x * x) x)) 42
```

Evaluation of functions V

- STEP 5:

```
twice (fun x -> x * x) 42

(fun f -> (fun x -> f(f x))) (fun x -> x * x) 42

(fun x -> (fun x -> x * x) ((fun x -> x * x) x)) 42
```

Make sure you know which x!

Evaluation of functions VI

- STEP 6:

```
twice (fun x -> x * x) 42

(fun f -> (fun x -> f(f x))) (fun x -> x * x) 42

(fun x -> (fun x -> x * x) ((fun x -> x * x) x)) 42

(fun x -> x * x) ((fun x -> x * x) 42)
```

Evaluation of functions VII

- STEP 7:

```
twice (fun x -> x * x) 42

(fun f -> (fun x -> f(f x))) (fun x -> x * x) 42

(fun x -> (fun x -> x * x) ((fun x -> x * x) x)) 42

(fun x -> x * x) ((fun x -> x * x) 42)
```

Evaluation of functions VIII

- STEP 8:

```
twice (fun x -> x * x) 42

(fun f -> (fun x -> f(f x))) (fun x -> x * x) 42

(fun x -> (fun x -> x * x) ((fun x -> x * x) x)) 42

(fun x -> x * x) ((fun x -> x * x) 42)

(fun x -> x * x) (42 * 42)
```

Evaluation of functions IX

- STEP 9:

Evaluation of functions X

```
twice (fun x -> x * x) 42

(fun f -> (fun x -> f(f x))) (fun x -> x * x) 42

(fun x -> (fun x -> x * x) ((fun x -> x * x) x)) 42

(fun x -> x * x) ((fun x -> x * x) 42)

(fun x -> x * x) (42 * 42)

(fun x -> x * x) 1764
```

Evaluation of functions XI

The last steps should be easy.

Evaluation of functions XII

- Note that the above function declaration does not depend on names declared earlier. Otherwise the function would need to refer to the environment in its closure.
- Example:

  let x = 5;;   (* $\rho_1 = \{$x->2$\}$ *)
  let f y = x * y;;   (* $\rho_2 = \{$f-><y->x*y,x->2>, x->2$\}$ *)
  f 5;;

- Evaluation of an application:
  - First evaluate the function name (i.e. replace with function value which looks like fun x)
  - For each argument, evaluate and substitute

Evaluation of functions XIII

- Technically, I should have written

```
twice (fun x -> x * x) 42
<f -> x -> f(f x), rho> (fun x -> x * x) 42
```

  etc. in the earlier computation. In other words, the informal notation

```
(fun f -> (fun x -> f(f x)))
```

  is not a value. It should

```
<f -> x -> f(f x), rho>.
```

  But I ignored rho since `twice` depends only on its parameters.

Evaluation of functions XIV

- Exercise.
  - Write a function `max_at` such when when you call `(max_at x g h)`, the function `g` is returned if `(g x)` is >= than `(h x)`.
  - What is the type of `max_at`
  - What is the type of `(max_at 0)`?
  - What is the type of `(max_at 1 (fun x -> 2 * x))`?
  - What is the type of `(max_at 1 (fun x -> 2 * x) (fun x -> x + 3))`?

Recursive functions I

- Factorial function in mathematical notation:

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times \text{factorial}(n-1) & \text{if } n > 0 \end{cases}$$

- Factorial in C/C++:

```
int factorial(int n)
{
  if (n == 0) return 1;
  else return n * factorial(n - 1);
}
```

Recursive functions II

- Factorial in OCAML

```
let rec factorial n =
    if n = 0 then
        1
    else
        n * factorial (n - 1)
;;
```

  You must use "rec". Test it.

- It's OK to use `rec` even if the function is not recursive:

```
let rec f x = x + 1;; (* ... but why do that? *)
```

Recursive functions III

- Exercise.
  - Write a function `fib` for the Fibonacci sequence: 1, 1, 2, 3, 5, 8, 13, where `(fib 0)` is 1.
  - The Fibonacci sequence starts with 1, 1. You can also have the following sequence that is like the Fibonacci sequence except that it starts with 2, 1: 2, 1, 3, 4, 7, 11, 18, ... Write a function `f` such that `(f a b)` returns a function that gives you the "Fibonacci sequence" except that it starts with `a` and `b`.

Mutual recursion I

- Mutual recursion: collection of functions calling each other.

```
let rec
    f x =
        if x = 0 then
            0
        else
            let _ = print_string "f\n" in
            g (x - 1)
and
    g x =
        if x = 0 then
            0
        else
            let _ = print_string "g\n" in
            f (x - 1)
;;
(f 10);;
```

No parameter, no return value I

- Recall that all expressions must have a value. If you have a function that does not return a value, just return (). (Like `print_int`).
- What about a function with no parameters?
- Recall that a function is

```
fun x -> [some expression]
```

- So you must have parameter `x` - no choice.
- If you do not intend to pass a value to a function, just pass in `()`.

No parameter, no return value II

- Try this:

```
let f () = 42;
```

- Make sure you look at the type of `f`.
- Now try to call it:

```
f ();;
```

- Of course you can also do this:

```
let f = fun () -> 42;
```

- Try this too:

```
let g = fun () -> ();
```

No parameter, no return value III

- Exercise. Compare

```
let f () = 42;
```

with

```
let f x = 42;
```

with

```
let f 1 = 42;
```

No parameter, no return value IV

- Here's a program that keeps asking for an `int` until you give it 42:

```
let rec main () =
  let _ = print_string
          "Give me 42 please ... " in
  let x = read_int () in
  if x = 42 then
    print_string "Thanks!!!\n"
  else
    main ()
;;
```

Make sure you check the type of `main`

No parameter, no return value V

- Here's how to get random integers 0..9:

```
Random.self_init ();;
print_int (Random.int 10);;
print_int (Random.int 10);;
print_int (Random.int 10);;
```

- Exercise. How do you get a random integer in the range `10..25`?
- Exercise. Write a program that simulates die rolls until it gets two consecutive sixes.

No parameter, no return value VI

- Exercise: Write a number guessing game. Call (guess 42) and the user is prompted to enter a guess. If the number entered is 42, the program will stop. If the number entered is $<$ 42, the program will ask the user to try a larger number. If the number entered is $>$ 42, the program will ask the user to try a smaller number. Solution on next slide ...

No parameter, no return value VII

- ```
  let rec guess n =
      let _ = print_string "guess my int: " in
      let i = read_int () in
      if i = n then
          print_string "correct!!!\n"
      else
          let _ =
              if i < n then
                  print_string "try higher ...\n"
              else
                  print_string "try lower ...\n"
              in
      guess n
  ;;
  guess 42;;
  ```
  ```

No parameter, no return value VIII

- Exercise. Write a program that assigns random integers `90..99` to `x` and `y` and that prompts the user for the product of `x * y`.
- Exercise. Write a program to compute the average gain of playing the following game 100000 times: To play the game, you have to pay me $1. You roll a die. If it's 1, you get $2, if it's even you give me $0.75, if it's 3, I give you $0.25, if it's 5, you give me $0.40.
- Exercise. Write a program that plays a game of tic-tac-toe against the user.

No parameter, no return value I

- Try this:

```
# let x = (1,2.2,"3.3.3");;
val x : int * float * string = (1, 2.2, "3.3.3")
```

- This is a **tuple**.
- Try the above without parentheses.
- Now try this:

```
let a,b,c = x;;
```

- Now try to pick up the first two values of `x`:

```
let x = (1,2.2,"3.3.3");;
let a,b = x;;
```

No parameter, no return value II

- Too bad. You need to have the right number of names. But we don't care about the third value. So ...

  ```
  let a,b,_ = x;;
  ```

  _ is like an anonymous variable. Can _ be used at any position?

- Tuples can be nested.

  ```
  let x = (1,2,3),(4,5,(6,7,8));;
  ```

- You can now use tuples to declare functions:

  ```
  let sum1 (x,y) = x + y;;
  let sum2 x y = x + y;;
  ```

  Make sure you compare the types!

No parameter, no return value III

- Of course

```
let sum1 (x,y) = x + y;;
let sum2 x y = x + y;;
let inc1 = sum1 1;; (* BAD *)
let inc2 = sum2 1;; (* OK *)
```

  We say that
  - `sum1` is uncurried
  - `sum2` is curried

  Compare the types of `sum1`, `sum2`.

No parameter, no return value IV

- Exercise. What is the type of `f`?

```
let a = 2;;
let b = fun x -> x + 1;;
let f = (a, b);
```

- Exercise. Write a function `f` such that `(f 2 5)` returns `(2, 5, 7)` (the third value is the sum of the first two). What is the type of `f`?

No parameter, no return value V

- Exercise. Write a function `evaluate` that behaves as follows:

```
let SUM = 0;;
let PROD = 1;;
(evaluate (SUM, 1, 2));;     (* 3 *)
(evaluate (SUM, 10, 5));;    (* 15 *)
(evaluate (PROD, 2, 3));;    (* 6 *)
(evaluate (PROD, 5, 3));;    (* 15 *)
```

Expression matching I

- Exercise. Complete the following function which returns 0 if x is 0, y if x is 1 and 42 otherwise.

```
let f (x, y) =
```

Test it.

Expression matching II

- Now for something new:

```
let f (x, y) = match (x, y) with
  (0, y) -> 0
| (1, y) -> y
| (x, y) -> 42
;;
```

  Test the function.

Expression matching III

- You can of course write the above like this:

```
let f = fun (x, y) ->  match (x, y) with
  (0, y) -> 0
| (1, y) -> y
| (x, y) -> 42
;;
```

Expression matching IV

- Actually we can do better:

```
let f (x, y) = match (x, y) with
  (0, _) -> 0
| (1, y) -> y
| (_, _) -> 42
;;
```

  Test the function.

- _ is an anonymous variable: like a variable but you can't access the value. Try

```
let _ = 42;;
print_int _;;
```

Expression matching V

- Exercise. Rewrite the factorial function using expression matching. [Solution on next slide.]
- Exercise. Write the fibonacci function using expression matching.
- Exercise. Write a function `f` that accepts `(x, y)` and returns the first nonzero value if any; otherwise 0 is returned.

Expression matching VI

- Solutions:

```
let rec factorial = fun n ->
    match n with
      0 -> 1
    | n -> n * (factorial (n - 1))
;;
let rec fib n = match n with
  0 -> 1
| 1 -> 1
| n -> (fib (n - 1)) + (fib (n - 2))
;;
let rec f = fun (x, y) -> match (x, y) with
  (0, y) -> y
| (x, _) -> x
;;
```

Lists I

- Try these:

```
[];;
[1];;
[(1,2)];;
[1;2];;
[1; 2.2];;
[1;];;
[[1;2];[3;4]];;
["abc"; "def"];;
```

- List: Homogeneous and ; separated
- Pay attention to the type.

Lists II

- Try this. `::` is called the **cons**. Watch the ones with errors.

```
1::[];;
1::2::[];;
1::2;;
1::2::3::[];;
1::[2;3];;
[1;2]::3::[];;
[1;2]::[3];;
```

- `[1;2;3]` is really `1::2::3::[]`

Lists III

- Try these:

```
let x = [1;2;3];;
let y = [3;2;1];;
x = y;;
x <> y;
[1;2;3] @ [4;5;6];;
```

Lists IV

- Note that `[1;2;3;4]` is really `1::[2;3;4]`
- 1 is the **car** of `[1;2;3;4]` and `[2;3;4]` is the **cdr** of `[1;2;3;4]`.
  - Very common LISP concepts.
  - Google "car cdr" for more information.
- Sometimes `1` is called the **head** of `[1;2;3;4]` and `[2;3;4]` is called the **tail** (or the rest) of `[1;2;3;4]`

Lists V

- OCAML lists are implemented using singly linked lists
- The following is a very important recursive definition of list
- A **type `t` list** is
  - Either: the empty list of the form `[]`
  - Or: a nonempty list of the form `x::xs` where `x` is a type `t` value and `xs` is a type `t` list

Recursive functions on lists I

- Example: Write a function to increment every value in an int list.

```
let rec inclist list = match list with
  [] -> []
| x::xs -> x+1 :: (inclist xs)
;;
```

Test it. Trace:

```
inclist [4;5;6] = (4+1)::(inclist [5;6])
                = (4+1)::(5+1)::(inclist [6])
                = (4+1)::(5+1)::(6+1)::(inclist [])
                = (4+1)::(5+1)::(6+1)::[]
```

Recursive functions on lists II

- Note the use of the recursive definition of list and definition of a function on list:

```
let rec f list = match list with
  [] -> ...
| x::xs -> ... x ... (f xs) ...
;;
```

- Of course the base case and recursive case must have the same type

Recursive functions on lists III

- Example: Write a function to add all values in an int list.

```
let rec sum list = match list with
  [] -> 0
| x::xs -> x + (sum xs)
;;
```

Trace

```
sum [2;4;3] = 2 + (sum [4;3])
            = 2 + (4 + (sum [3]))
            = 2 + (4 + (3 + (sum [])))
            = 2 + (4 + (3 + (0)))
```

Recursive functions on lists IV

- Suppose given the list `[3;4;5]`, I want to compute `3 * 4 * 5`.
- In general I want to do this for a list of any length. Call function `prodlist`.
- Think recursively:

```
prodlist [3;4;5] = 3 * (prodlist [4;5])
```

- Now think about the base case:

```
prodlist [3;4;5] = ...
                 = 3 * 4 * (prodlist [5])
                 = 3 * 4 * 5 * (prodlist [])
```

Therefore, I want `prodlist []` to be 1.

Recursive functions on lists V

- In general

$$\texttt{prodlist list} = \begin{cases} 1 & \text{if } \texttt{list = []} \\ \texttt{x * (prodlist xs)} & \text{if } \texttt{list = x::xs} \end{cases}$$

Now were ready to code ... your turn ...

Recursive functions on lists VI

- Exercise. Note that the above `inclist` applies the increment function to each element of a list. This can be generalized. Complete the following:

```
let rec map f list =







;;
let inclist = map (fun x->x+1);;
let declist = map (fun x->x-1);;
```

Recursive functions on lists VII

- Exercise. Consider this `f`:

```
let rec f list a = match list with
  [] -> []
| x::xs -> x::a::(f xs a)
;;
```

What is the type of `f`? What does `f` do?

Recursive functions on lists VIII

- Exercise. What does this do?

```
let rec f list1 list2 =
    match list1,list2 with
      [],[] -> []
    | x::xs, y::ys -> x::y::(f xs ys)
;;
```

There are some unmatched cases. So redo the declaration of `f` and remove the warnings.

Recursive functions on lists IX

- Let's generalize the sum of int list and product of int list functions. They look very similar:

```
sumlist  [2;3;5] = 2 + 3 + 5
prodlist [2;3;5] = 2 * 3 * 5
```

- Call the function `reduce`. Obviously `reduce` must take a list as input. What else?

Recursive functions on lists X

- Recursively:

```
sumlist  [2;3;5] = 2 + (sumlist [3;5])
prodlist [2;3;5] = 2 * (prodlist [3;])
```

  The only difference is the + and ∗. Allowing these to be function inputs for `reduce`:

```
sumlist  [2;3;5] = f1 2 (sumlist [3;5])
prodlist [2;3;5] = f2 2 (prodlist [3;5])
```

  where `f1` is `fun x -> fun y -> x + y` and `f2` is `fun x -> fun y -> x * y`

- So `reduce` takes a list and a binary function `f` as input.

Recursive functions on lists XI

- The other difference is the base case:

```
sumlist [2;3;5] = 2 + (3 + (5 + (sumlist [])))
prodlist [2;3;5] = 2 * (3 * (5 * (prodlist [])))
```

- So `reduce` takes a binary function and a base case value and a list.

Recursive functions on lists XII

- Altogether

```
sumlist [2;3;5] = 2 + (3 + (5 + (sumlist [])))
prodlist [2;3;5] = 2 * (3 * (5 * (prodlist [])))
```

becomes

```
reduce f1 0 [2;3;5] = f1 2 (reduce f1 0 [3;5])
reduce f2 1 [2;3;5] = f2 2 (reduce f2 1 [3;5])
```

and

```
reduce f1 0 [] = 0
reduce f2 1 [] = 1
```

Recursive functions on lists XIII

- With the above `reduce` function we can define

```
let sumlist = reduce (fun x -> fun y -> x + y) 0;;
let prodlist = reduce (fun x -> fun y -> x * y) 1;;
```

- Exercise. Complete the `reduce` function and test it.
- Previously we had the `map` function. You now have the `map` and `reduce` functions. Google "mapreduce".

Recursive functions on lists XIV

- Exercise. Write a function that reverses a list. For instance `rev [1;2;3]` evaluates to `[3;2;1]`.

Recursive functions on lists XV

- Exercise. We want to define the length of a list. Think recursively:

  ```
  length [4;2;3] = 1 + length [2;3]
  ```

  Base case:

  ```
  length [4;2;3] =  = 1 + 1 + 1 + (length [])
  ```

  Complete the problem. (De jevu). Can you do it using map and reduce?

Recursive functions on lists XVI

- Exercise. We want to compute squares recursively. Think recursively:

```
n^2 = (n-1 + 1)^2
    = (n-1)^2 + 2(n-1) + 1
    = (n-1)^2 + 2n - 1
```

Mutual recursion and lists I

- Example. Write a function to sum up every other term of the list starting with first value. E.g., if the list is [1;3;2;4;5], want 1+2+5.

```
let rec
    add xs = match xs with
    [] -> 0 | y::ys -> y + (skip ys)
and
    skip xs = match xs with
    [] -> 0 | y::ys -> (add ys)
;;
add [1;2;3;4;5;6];;
```

Mutual recursion and lists II

- Example: Write a function `f` such that `(f [3;1;2;4;6;5])` returns `[3;2;6]`, i.e., list of every other value starting with first.

```
let rec
    take xs = match xs with
    [] -> [] | x::xs -> x::(skip xs)
and
    skip xs = match xs with
    [] -> [] | x::xs -> take xs
;;
take [1;3;2;4;5];;
```

Mutual recursion and lists III

- Exercise: Write a function `f` that computes the alternating sum of a list of values. For instance `(f [1;2;3;4;5])` returns `1 - 2 + 3 - 4 + 5`.
- Exercise. Write a function `f` that accepts a list and returns a list with `0`s inserted between list values. For instance `(f [1;2;3])` returns `[1;0;2;0;3]`.