Ocaml Tutorial

2011. 9. 14
Programming Languages, 2011 FALL

Jinyung Kim & Yongho Yoon
{jykim,yhyoon}@ropas.snu.ac.kr
ROPAS

Introduction

Programming Experience..?

- C/C++, Java, Python
- Scheme, Haskell, ML

OCaml

- Objective Caml
- Popular, expressive, high-performance
- (Mostly) Functional
- Strongly typed
- Value-oriented

Functional?

```
// in C
int gcd (int a, int b) {
    while (b!=0) {
        int m = a % b;
        a = b;
        b = m;
    }
    return a;
}
```

```
(* in OCaml *)
let rec gcd a b =
    if b = 0 then a
    else gcd b (a mod b)
```

Additional Sources

- http://www.ocaml.org
- Style guide
  - http://www.cs.caltech.edu/~cs20/a/style.html
- Introduction to Objective Caml, J. Hickey
  - http://files.metaprl.org/doc/ocaml-book.pdf

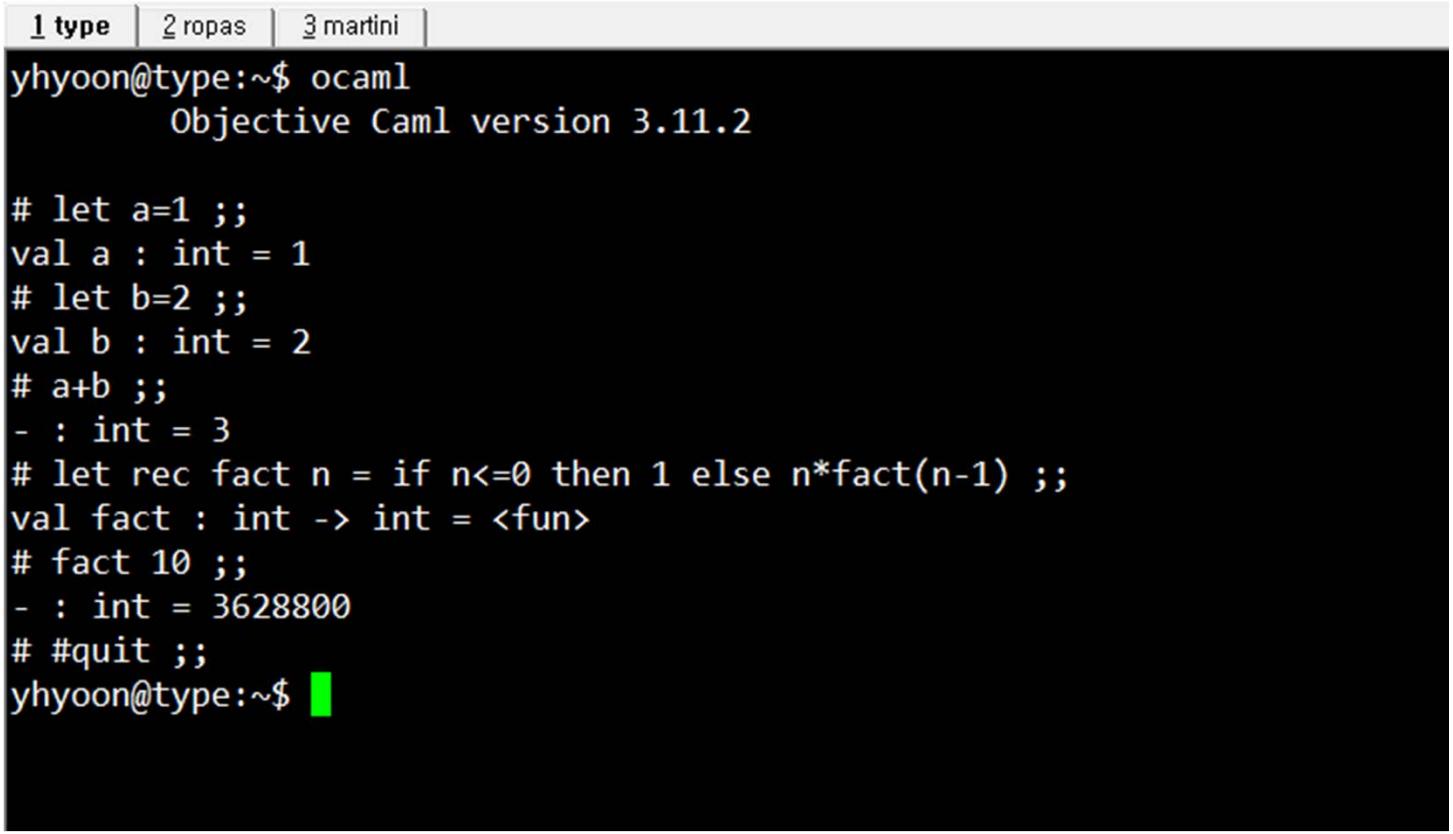
Environment

Setup

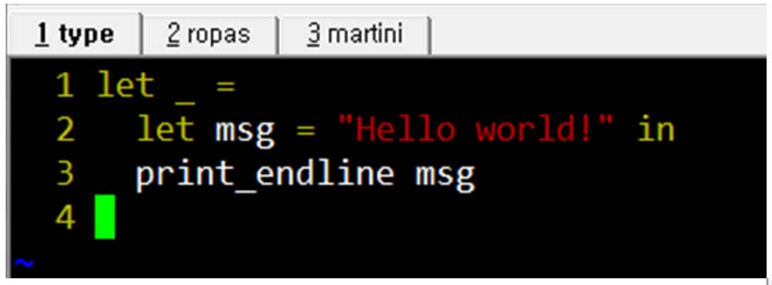
- ocaml.org > Download
- or snucse servers (martini, etc)

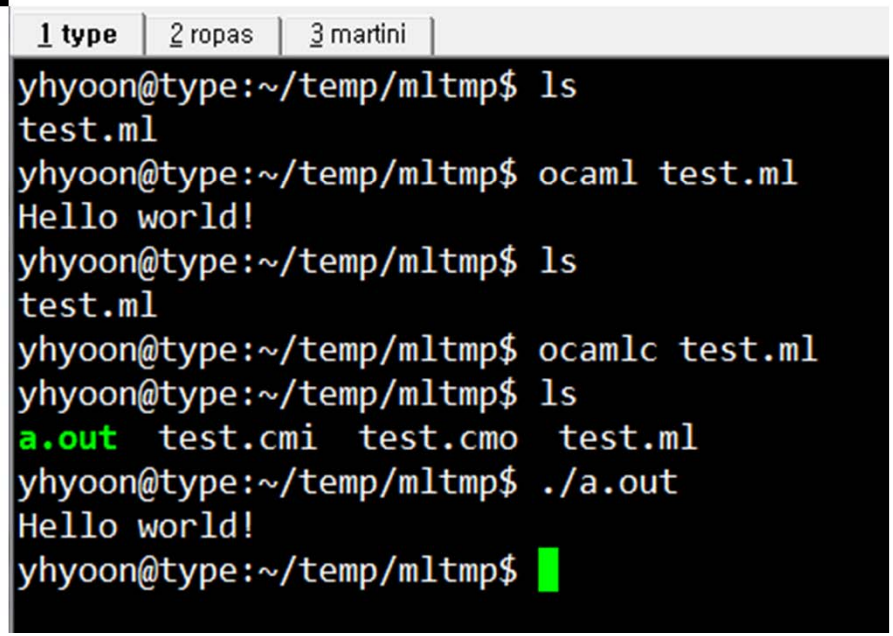
- martini, 3.10.2
- most recent, 3.12.1
  - (not allowed in our class)
- we (TAs) use 3.11.2

Interactive!



You can compile it, too





Interpreter vs Compiler

- Interpreter
  - Can see the result right away
  - (expression) ;;
  - #quit;; to exit

- Compiler (recommended)
  - Create a ???.ml file then compile
  - Don't use ;; here
  - (mostly) a sequence of definitions
  - Please submit files that **COMPILES SUCCESSFULLY..**

Editor

- vi, emacs, …
  - Whatever suits you best
  - Configurations are available
- eclipse plugin (OcaIDE)
  - +Cygwin
- Notepad?
- Should look good to you
  - To avoid mistakes

Let’s Begin

Let It Be (for real..)

- `let a=1 in …`
  - Definition
  - NOT a variable
- `let incr = fun x -> x+1`
  - Let incr be the function increments an integer
  - `incr 10 = 11`
- `let incr x = x+1`

Named Function

- You name it

```
1 let incr n = n+1
2
3 let rec fact n =
4     if n<0 then raise (Invalid_argument "factorial")
5     else if n=0 then 1
6     else n * fact(n-1)
```

- REC for recursion

Nameless Function

- Named
  - `let incr n = n+1 in incr 1`
  - `2`
- Nameless
  - `(fun x->x+1) 1`
  - `2`
- Function as a value

Value

- Integer
  - `let i = 1`
- String
  - `let s = “hello world!”`
- List
  - `let l = [1;2;3;4;5]`
  - `let l2 = 1::2::3::4::5::[]`
- Function
  - `let incr = fun x -> x+1`
  - `let cons a b = a::b`
- And so on

Type (1/2)

- Integer
  - `let i = 1` `(* int *)`
- String
  - `let s = "hello world!"` `(* string *)`
- List
  - `let l = [1;2;3;4;5]` `(* int list *)`
  - `let l2 = 1::2::3::4::5::[]` `(* int list *)`
- Function
  - `let incr = fun x -> x+1` `(* int -> int *)`
  - `let cons a b = a::b` `(* 'a -> 'a list -> 'a list *)`
- And so on

Type (2/2)

- Integer
  - `let i : int = 1 (* int *)`
- String
  - `let s : string = “hello world!” (* string *)`
- List
  - `let l : int list = [1;2;3;4;5] (* int list *)`
  - `let l2 : int list = 1::2::3::4::5::[] (* int list *)`
- Function
  - `let incr : int -> int = fun x -> x+1 (* int -> int *)`
  - `let cons a b = a::b (* ’a -> ’a list -> ’a list *)`
- An so on

Type Inference

- Figure out the types for you
  - C/C++ : `int a=1; string s="abc";`
  - Ocaml : `let a=1  let s="abc"`
- Type inference
  - automatically
  - safe
  - Strongly typed (at compile time)

Types

- int
- float
- string
- 'a list
  - int list, string list, float list, int list list, …

- 'a * 'b : pair
  - (1, 2.0) : int * float
  - ("yhyoon", 20889) : string * int
- 'a -> 'b : function with an argument
  - incr : int -> int
  - fst : ('a * 'b) -> 'a snd : ('a * 'b) -> 'b
  - List.length : 'a list -> int

No main

- Runs line-by-line

```
 1 let a=1
 2
 3 let b=2
 4
 5 let add x y = x+y
 6
 7 let sum = add a b
 8
 9 let _ =
10     print_endline ("Hello world!");
11     print_int sum;
12     print_newline()
13
```

No main

- Runs line-by-line

```
1 let a=1
2
3 let b=2
4
5 let add x y = x+y
6
7 let sum = add a b
8
```

Some more..

Currying (1/2)

- `gcd a b`?
- `gcd(a,b)`?

Currying (1/2)

- `gcd a b`?
- `gcd(a,b)`?

- Different
  - `gcd(a,b) : (int * int) -> int`
    - Pair of int
  - `gcd a b : int -> int -> int`
    - Two ints
  - `let add a b = a + b`
  - `let incr = add 1`

Currying (2/2)

- **BE CAREFUL**
  - `int -> int -> int`
    - `let rec gcd a b = …`
  - `(int * int) -> int`
    - `let rec gcd (a,b) = …`
  - If the type is different from what is given your program won't be graded

match – with

- Similar to switch – case
- Comes very handy

- ```
  match x with
    A -> a
  | B -> b
  | C -> c
  | _ -> default
  ```
  ```

match - with example

- Gcd

```
let rec gcd a b =
    if a=1 || b=1 then 1
    else if a=b then a
    else if a<b then gcd b a
    else gcd (a-b) b
```

```
let rec gcd a b =
  match (a,b) with
  | (1, _) | (_, 1) -> 1
  | _ ->
    if a=b then a
    else if a<b then gcd b a
    else gcd (a-b) b
```

- Handling list

```
let rec length l =
  match l with
  | [] -> 0
  | _::t -> 1 + length t
```

```
let rec sum_of_list l =
  match l with
  | [] -> 0
  | h::t -> h + sum_of_list t
```

Be careful

- Nested?

```
let rec merge l1 l2 =
  match l1 with
  | [] -> l2
  | h1::t1 ->
    (match l2 with
    | [] -> l1
    | h2::t2 ->
      if h1<h2 then h1::merge t1 l2
      else h2::merge l1 t2)
```

- This looks better

```
let rec merge l1 l2 =
  match (l1,l2) with
  | ([], _) -> l2
  | (_, []) -> l1
  | (h1::t1, h2::t2) ->
    if h1<h2 then h1::merge t1 l2
    else h2::merge l1 t2
```

try – with, raise

- `raise` : exception

- `try – with`
  - Similar to `try – catch` in Java
  - `try … with Exception1 -> …`
  - Grammar is similar to `match – with`

try – with examples

- Division by zero

```
1 let s =
2   try
3     string_of_int (
4       let a=read_int() in
5       let b=read_int() in
6       a/b)
7   with Division_by_zero -> "error"
```

- Fail to open file

```
1 let _ =
2   try
3     let input = open_in "input.txt" in
4     let line = input_line input in
5     print_endline "the first line of the file is";
6     print_endline line
7   with Sys_error -> print_endline "fail to open file input.txt"
```

List (1/3)

- You'll gonna use it a lot

- Empty `[]`
- Not empty `[1;2;3;4;5]`
  - or `1::2::3::4::5::[]`
- Every list is either
  - `[]`
  - Or head::tail

List (2/3)

- Matching

```
1 match l with
2 | [] -> ...
3 | h::t -> ...
```

- Examples

```
1 let rec length l =
2   match l with
3   | [] -> 0
4   | _::t -> 1 + length t
```

```
1 let rec sum_of_list l =
2   match l with
3   | [] -> 0
4   | h::t -> h + sum_of_list t
```

List (3/3)

- Library
  - http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html
- `List.length`
  - `List.length [1;2;3] = 3`
- `List.nth`
  - `List.nth [1;2;3] 2 = 2`
- `List.rev`
  - `List.rev [1;2;3] = [3;2;1]`
- `List.mem`
  - `List.mem 1 [1;2;3] = true`

User-defined type

- You can make your own int list

```
 1 type intlist = Nil | List of int * intlist
 2
 3 let rec length l =
 4   match l with
 5   | Nil -> 0
 6   | List(_, t) -> 1 + length t
 7
 8 let rec sum_of_list l =
 9   match l with
10   | Nil -> 0
11   | List(h, t) -> h + sum_of_list t
```

Polymorphic type

- You can define like this too

```
type 'a mylist = Nil | List of 'a * 'a mylist
type intlist = int mylist

let rec length l =
  match l with
  | Nil -> 0
  | List(_, t) -> 1 + length t

let rec sum_of_list l =
  match l with
  | Nil -> 0
  | List(h, t) -> h + sum_of_list t
```

- ‘a can be any type
- Functions can also be polymorphic

Omitted

- Module, functor
- Reference (like variable)
  - `let id = ref 1`

- Mutual recursion

More Resources (1/2)

- Library document
  - http://caml.inria.fr/pub/docs/manual-ocaml/libref/index.html
  - Take advantage of library functions.
  - fold, map, iter, …
- Manual
  - http://caml.inria.fr/pub/docs/manual-ocaml/index.html

More Resources (2/2)

- Example codes by TAs
  - http://ropas.snu.ac.kr/~ta/4190.310/11f/
- Ask us
  - Email
  - Office hours

Reference

- Tutorial by Yongho
  - in Korean