OCaml Tutorial

Prof. Jaeseung Choi

Dept. of Computer Science and Engineering

Sogang University

OCaml for Course Project

- **As announced in the course orientation, we will use OCaml language for the course project**
  - Also, refer to the previous ***4. Intermission*** lecture slide
- **We will focus on learning the minimal features required for the project**
- **You will do some OCaml programming exercise in a programming assignment (HW #2)**
  - Warm-up before the course project

Why learn OCaml?

- **OCaml has many useful features for handling programming languages**
  - Good for writing a ***program that deals with programs***
  - Ex) Compiler, interpreter, program analyzer, ...
- **OCaml gives you a new viewpoint for programming**
  - A different paradigm from languages like C/C++ or Python
- **Of course, OCaml is not a perfect language**
  - Too slow and unpopular to be used in the industry
- **Still, learning OCaml will be a good experience**
  - Experiencing various paradigms will help you to learn a new state-of-the-art language in the future (e.g., ***Rust***)

Environment for OCaml

- **OCaml 4.02.3 is installed in cspro5.sogang.ac.kr**
  - Strongly recommend to use this environment
- **You can use either (1) `ocaml` to directly run a program, or (2) `ocamlc` to compile a program into binary**

```
jason@ubuntu:~/Example/OCaml$ ocaml hello.ml
Hello world
jason@ubuntu:~/Example/OCaml$ ocamlc hello.ml -o hello.bin
jason@ubuntu:~/Example/OCaml$ ./hello.bin
Hello world
```

- **You may setup OCaml in your computer, too (using `opam`)**
  - But be careful about the difference in versions
  - If your code does not run due to version issue, **you lose points**

Languages you learned so far

- ***Imperative*** **programming language**
  - Ex) C, C++, Python, ...
- **Programming = Ordering a computer what to do**
  - Ex) Allocate and deallocate memory
  - Ex) Store certain value in a memory space
  - Ex) Manipulate a variable or object allocated in the memory

```
int x = 10;
int y = x + 1;
Set *s = new Set(); // Object to store integer elements
s->add(y);
delete s;
...
```

OCaml: A Different Paradigm

- ***Declarative*** **programming language (a.k.a., functional programming language)**
  - Ex) OCaml, Scala, F#, Haskell, …
  - Personally, I like to describe it ***value-oriented*** programming
- **Programming = Declaring (defining) values to compute**
  - How are those values computed? We do not care
  - This allows us to focus on high-level logics

```
let x = 10
let y = x + 1
let s = Set.add y Set.empty (* s represents set { 11 } *)
...
```

Your First OCaml Program

- **Remember: program is a sequence of value definitions**
  - Integer value with name **`i`**
  - String value with name **`s`**

Must start in lowercase!

- **Sometimes, the value itself is not important**
  - For **`printf`**, it just returns a **`unit`** (= **`void`** in C) value
  - We are not interested in this returned value, so don't name it
  - We are using **`printf`** for its side-effect (print out something)

1-first.ml

```
let i = 1
let s = "OCaml"
let _ = Printf.printf "%d\n" i
let _ = Printf.printf "%s\n" s
(* '_' means we are not giving any name to the value *)
```

Add comments with (* *)

Type in OCaml

- **Note that you did not write the type of values**
- **OCaml can automatically infer the type of each value**
- **You may choose to explicitly annotate types like below**
- **OCaml also detects type errors in your program**
  - Similar to what we will do in the *type checking* project
  - But far more complex than what we learn in this course

2-type.ml

```
let b: bool = true
let i: int = 1
let s: string = "OCaml"
let x = i + s (* Type error! *)
```

Defining Function

- **You can also define functions like below**
  - Note there is **no `"return"`**; you just write the return value itself (similar to how you define a function in *mathematics*)
  - When defining a value inside a function, you must use **`in`**
  - If the function does not take in any argument, put **`()`** instead

3-function.ml

```
let add x y = x + y

let square_add x y =
  let s1 = x * x in
  let s2 = y * y in
  s1 + s2

let print_msg () = (* No argument *)
  Printf.printf "Hello\n"
```

Don’t forget this "in"

Using Defined Function

- **So far, you have just defined function**
  - Defined functions are not automatically called
  - Cf. Recall the function definition in Python
- **You must explicitly call the wanted functions at the end**
  - Note that it's not **`"add(2,3)"`**; write the arguments directly

3-function.ml

```
let add x y = ...

let square_add x y = ...

let print_msg () = ...

let a = add 2 3
let b = square_add 4 5
let _ = print_msg () (* Parentheses ared only used here *)
```

Recursive Function & If-then-else

- **When defining a recursive function, use `"let rec"`**
  - In OCaml, **recursion** replaces loops (don't use **`while`** or **`for`**)
- **`"if b then X else Y"` is a conditional expression**
  - If **`b`** is true, it evaluates to **`X`**
  - If **`b`** is false, it evaluates to **`Y`**
  - Only *Boolean* values can come in **`b`** (types like **`int`** not allowed)
    - "`true`", "`false`", "`x = y`", "`n > 1`", …

4-factorial.ml

```
let rec factorial n =
  if n <= 1 then 1
  else n * factorial (n - 1)   (* ( ) is for priority *)

let x = factorial 5
let _ = Printf.printf "%d\n" x
```

Pair & Tuple

- **You can group two (or more) values into one value**
  - You must have seen similar features in python
- **This explains why `add(2,3)` did not work previously**
  - OCaml will think that you are trying to pass one argument (whose type is pair)

5-tuple.ml

```
let x = (1, "abc")  (* x is a tuple of int and string *)
let (a, b) = x  (* a = 1, b = "abc" *)
let _ = Printf.printf "%d %s\n" a b

(* You can also annotate tuple type like this *)
let y: (int * bool * string) = (1, false, "A")
```

Next: Advanced topics that are important for our course project

Defining Type

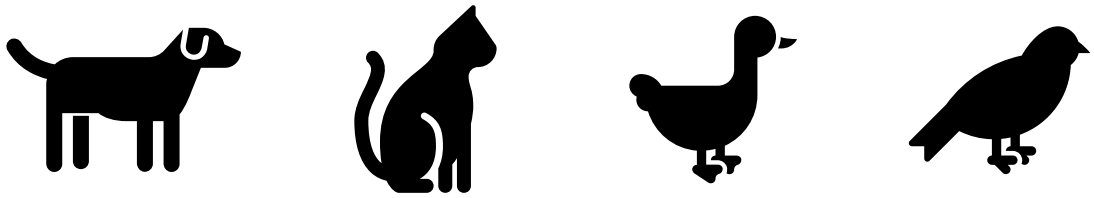
- **You can define your own type**
  - Union type is especially useful in OCaml
  - Ex) An animal can be one of: dog, cat, duck, or sparrow

Must start with uppercase!

6-animal.ml

```
type animal = Dog | Cat | Duck | Sparrow

let a1 = Dog (* Type of 'a1' is 'animal' *)
let a2 = Sparrow (* Type of 'a2' is also 'animal' *)
let _ = Printf.printf "Are they same? %b\n" (a1 = a2)
```



Pattern Matching

- We can use **match-with** to compute different value for each case of union type

6-animal.ml

```
type animal = Dog | Cat | Duck | Sparrow

let count_leg a =
  match a with
  | Dog -> 4
  | Cat -> 4
  | Duck -> 2
  | Sparrow -> 2

let a1 = Dog
let _ = Printf.printf "a1 has %d legs\n" (count_leg a1)
```

```
(* More compact version *)
let count_leg a =
  match a with
  | Dog | Cat -> 4
  | Duck | Sparrow -> 2
```

Recursive (Inductive) Type

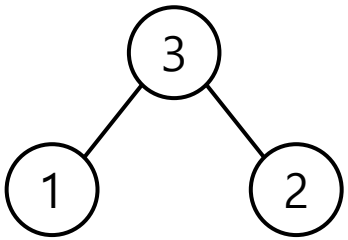
- **Let's define a (full) binary tree that store integers**
- **Inductive type definition**
  - Base case: a single leaf is a binary tree
  - Inductive case: a node with **two** subtrees is also a tree

7-tree.ml

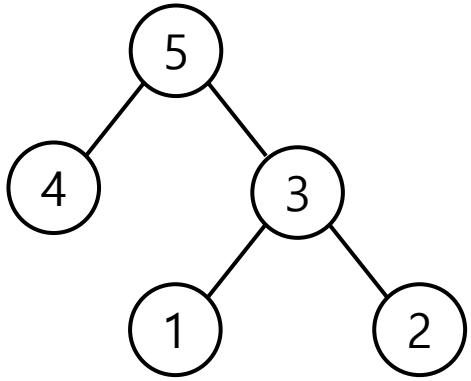
```
type tree = Leaf of int | Node of int * tree * tree

let t1: tree = Node (3, Leaf 1, Leaf 2)
let t2: tree = Node (5, Leaf 4, t1)
```

Tree **t1**:



Tree **t2**:



Example: AST

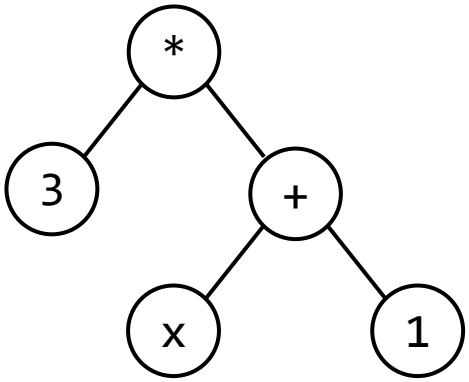
- **Let's define a simple language for numeric expression**
  - We have seen a similar CFG in the lecture and HW #1
  - We can define more concisely than the C code from HW #1

8-ast.ml

```
type exp =
    Num of int
  | Var of string
  | Add of exp * exp
  | Sub of exp * exp
  | Mul of exp * exp
  | Div of exp * exp

let e = Mul (Num 3, (Add (Var "x", Num 1)))
```

AST for
3 * (x + 1)



Standard Library: List

- **A popular data structure supported by the library**
- **Elements of a list must have the same type**
  - Note the type of **`il`** and **`sl`**
- **Note that functions like `List.length` can be used on both `int list` type and `string list` type**

Note: Not ","

9-list.ml

```
let il: int list = [1; 2; 3; 4]
let sl: string list = ["OCaml"; "F#"; "scala"]

let len1 = List.length il
let len2 = List.length sl
let b = List.mem 5 il       (* returns false *)
let head_elem = List.hd sl (* returns "OCaml" *)
```

Pattern Matching with List (★)

- **In fact, list type is also inductively defined**
  - **Base case:** an empty list (`[]`)
  - **Inductive case:** an element pushed on existing list (`elem::lst`)
  - So `[1;2;3]` is actually equal to `1 :: (2 :: (3 :: [ ]))`
- **We can perform useful computations using `match-with`**

9-list.ml

```
let rec my_length lst = (* Usable on any list *)
  match lst with
  | [] -> 0
  | head :: tail -> 1 + (my_length tail)

let rec double_list lst = (* Usable on int list only *)
  match lst with
  | [] -> []
  | head :: tail -> (2 * head) :: (double_list tail)
```

Standard Library: Map (★)

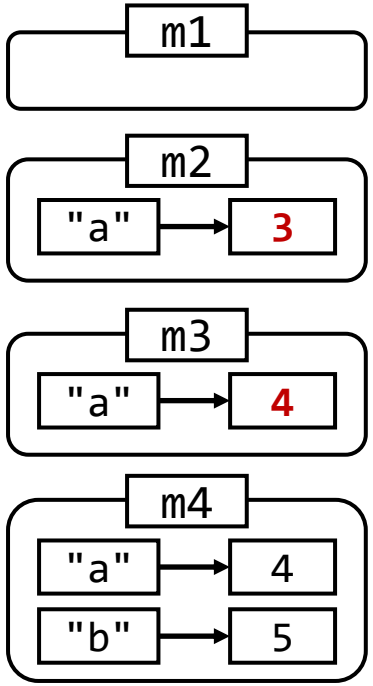
- **In the first line, we define a map that has `string` as key**
  - Cf. Template in C++ or generic in Java
  - Why `String` instead of `string`? I will not explain deeply
- **Then we can use values and functions in `IMap`**

10-map.ml

```
module IMap = Map.Make(String)

let m1 = IMap.empty
let m2 = IMap.add "a" 3 m1
let m3 = IMap.add "a" 4 m2
let m4 = IMap.add "b" 5 m3

let has_c = IMap.mem "c" m4 (* false *)
let has_b = IMap.mem "a" m4 (* true *)
let i = IMap.find "a" m4    (* 4 *)
```



Module

- **Collection of closely related types, values and functions**
  - We do similar things with a *class* in OOP, but the difference is that OCaml does **not** have a concept of *object*
- **For example, previous `List` and `IMap` were modules**
- **In the course project, you will be asked to complete a module in the skeleton code**
  - I will elaborate later if needed

```
module Queue = struct
  type t = int list * int list
  let empty: t = ([], [])
  let enqueue i queue = ...
  let dequeue queue = ...
end
```

Reference

- **There are useful materials (including a tutorial) in the official website: ocaml.org/docs**
  - These materials are good, but many of them are actually not necessary for our assignment and project
- **So I recommend you to focus on this tutorial slide**
  - I only included minimal materials needed for our course

Backup Slide:
Common Mistakes

Caution: No `if-then`

- **`if-then-else` in OCaml is an expression, not statement**
  - Recall that OCaml is a declarative language
- **So you cannot omit the `else` part**
  - The only exception is when the expression is **`unit`** type

```
let b = true

(* Error: how can this be computed? *)
let i = if b then 1

(* Here, () denotes a unit type value *)
let _ = if b then Printf.printf "Hello\n" else ()
(* In this case, the "else" part can be omitted *)
let _ = if b then Printf.printf "Hello\n"
```

Caution: Nested `match-with`

- **For nested `match-with` clauses, you need parentheses**
- **Otherwise, the parser cannot decide which `match-with` clause `Cat` belongs to**
  - You have learned that parsers are not that smart ☺

```
type animal = Dog | Cat
type color = Gray | Black

let is_black_dog a c =
  match a with
  | Dog ->
    (match c with
    | Gray -> false
    | Black -> true)
  | Cat -> false
```

Here, ( ) is required

Caution: Value does not change

- **From the previous example, what happens if we update `t1` into a new value? Does it affect `t2`?**
- **No: Once defined, a value does not change!**
  - In fact, you are not updating (changing) **`t1`**
  - You are defining a new value **`Leaf 7`** and giving it a name **`t1`**
  - Then, the previously defined **`Leaf (3, Leaf 1, Leaf 2)`** does not have a name and cannot be referred to anymore

```
type tree = Leaf of int | Node of int * tree * tree

let t1 = Node (3, Leaf 1, Leaf 2)
let t2 = Node (5, Leaf 4, t1)
let t1 = Leaf 7 (* What happens? Does t2 change? *)
```