4471028: 프로그래밍언어

Lecture 4 — OCaml 기초
Basics of the OCaml Language

임현승

2020 봄학기

Why learn ML?

Learning ML is a good way of experiencing modern language features:

- functional programming: scala, java8, haskell, python, JavaScript, etc
- value-oriented programming: scala, haskell, scheme, etc
- type inference: scala, haskell, etc 타입추론
- pattern matching: scala, haskell, etc
- algebraic data types, module system, etc 손쉬운 데이터 타입 관리.

Basics of the OCaml Language

- Expressions and values
- Names and functions
- Pattern matching
- Type inference
- Tuples and lists
- Data types
- Exceptions
- Modules

Write and run all examples in the slides by yourself!

An OCaml Program is an Expression

Statements and expressions:

- A statement *does something*. OCaml는 표현식으로 되어있음.
- An expression *evaluates to a value*. 표현식은 값으로 되어있음.

Programming languages can be classified into

- statement-oriented: C/C++, Java, Python, JavaScript, etc
  - often called "imperative languages"
- value-oriented: ML, Haskell, Scala, Lisp, etc 값 중심언어 (함수중심).
  - often called "functional languages"

Arithmetic Expressions

- Arithmetic expressions evaluate to numbers: e.g., `1+2*3`, `1+5`, `7`
- Try to evaluate expressions in the REPL:

```
# 1+2*3;;
- : int = 7
```

→ 표현식이 끝났다.

- Arithmetic operators on integers:

→ 정수연산이 됨. → 실수는 1.=1.0

$a + b$	addition
$a - b$	subtraction
$a * b$	multiplication
a / b	divide a by b, returning the quotient
a `mod` b	divide a by b, returning the remaining part

5/3=1
5.0/3.0= 1.XXXX
정수결과.

나머지연산 ←

Boolean Expressions

- Boolean expressions evaluate to boolean values (i.e., true, false).
- Try to evaluate boolean expressions:

```
# true;;
- : bool = true
# true;;
- : bool = true
# 1 > 2;;
- : bool = false
```

- Comparison operations produce boolean values:

$a = b$	true if a and b are equal
a <> b	true if a and b are not equal
$a < b$	true if a is less than b
$a <= b$	true if a is less than or equal to b
$a > b$	true if a is greater than b
$a >= b$	true if a is greater than or equal to b

Boolean Operators 논리연산자

- Boolean expressions may be combined by boolean operators:

```
# true && false;;
- : bool = false
# true || false;;
- : bool = true
# (2 > 1) && (3 > 2);; → true && true
- : bool = true
```

ML is a Statically Typed Language ML은 정적타입언어.

If you try to evaluate an expression that does not make sense, OCaml rejects and does not evaluate the program: e.g.,

```
# 1 + true;;
Error: This expression has type bool but an expression
was expected of type int
```

→ 실행하기도 전에 에러 뜸.

정적타입언어: 잘못된 프로그램을 받았을 때 거부하고 실행하지 않음.

cf) Static Types and Dynamic Types

Programming languages can be classified into: 타입검사. 컴파일 때

- *Statically typed languages*: type checking is done at compile time.
  - type errors are detected before program executions
  - C/C++, Java, ML, Scala, etc

실행중에

- *Dynamically typed languages*: type checking is done at run-time.
  - type errors are detected during program executions
  - Python, JavaScript, Ruby, Lisp, etc

스크립트 언어

Statically typed languages can be further classified into:

- *Type-safe languages*: guarantee that compiled programs do not have type errors at runtime. 컴파일된 프로그램은 문제가 없다
  - All type errors are detected at compile time.
  - Compiled programs do not stuck.
  - ML, Haskell, Scala

⇓ 컴파일되면 실행중에 멈추지 않음.

- *Unsafe languages*: do not provide such a guarantee.
  - Some type errors remain at runtime.
  - C/C++

위에건 보장 X.

Which one is better?

Statically typed languages:

- (+) Type errors are caught early in the development cycle.
- (+) Program execution is efficient by omitting runtime checks.
- (−) Less flexible than dynamic languages.

Dynamically typed languages:

- (−) Type errors appear at runtime, often unexpectedly.
- (+) Provide more flexible language features.
- (+) Easy and fast prototyping.

Conversion between Different Types

- In OCaml, `int` and `float` are distinguished:

```
# 3 + 2.0;;
Error: This expression has type float but an expression
was expected of type int
```

- Values of one type must be explicitly converted into values of another type:

f→i

```
# 3 + int_of_float 2.0;;
- : int = 5
```

- Operators for floating point numbers:

i→f

```
# 1.2 +. 2.3;;
- : float = 3.5
# 1.5 *. 2.0;;
- : float = 3.
# float_of_int 1 +. 2.2;;
- : float = 3.2
```

Other Primitive Values 기본적인 값.

- OCaml provides six primitive values: integers, booleans, floating point numbers, characters, strings, and unit.

```
# 'c';;
- : char = 'c'
# "hello";;
- : string = "hello"
# ();;
- : unit = ()
```

아무것도 아닌 값

※ 6 개본 값.

- integer
- boolean
- floating point N
- characters
- strings
- unit

Conditional Expressions

bool식. 참 거짓. 같은 type.

if be then e_1 else e_2

- If be is true, the value of the conditional expression is the value of e_1.
- If be is false, the value of the expression is the value of e_2.
- T F
  ```
  # if 2 > 1 then 0 else 1;;
  - : int = 0
  # if 2 < 1 then 0 else 1;;
  - : int = 1
  ```
- be must be a boolean expression.
- types of e_1 and e_2 must be equivalent.
-
  ```
  # if 1 then 1 else 2;;
  Error: ...
  # if true then 1 else true;;
  Error: ...
  # if true then false else true;;
  - : bool = false
  ```

Names and Functions 변수 이름과 함수

- Create a global variable with the `let` keyword:
  전역변수

```
# let x = 3 + 4;;
val x : int = 7
```

변수 x는 int 7로 묶여있다.

We say a variable x is *bound* to value 7. 묶여있다.

```
# let y = x + x;;
val y : int = 14
```

변수 y는 int 14로 묶여있다.

- Create a local variable with `let ... in` ... construct:
  지역변수

  `let` x = e_1 `in` e_2 → 결과.

  - x is bound to the value of e_1 → in 안에서만 씀.
  - the scope of x is e_2
  - the value of e_2 becomes the value of the entire expression

Examples

계산

```
# let a = 1 in a;;
- : int = 1
```

계산

```
# let a = 1 in a * 2;;
- : int = 2
```

```
# let a = 1 in
  let b = a + a in
  let c = b + b in
    c + c;;
- : int = 8
```

a, b, c는 모두 지역변수.

```
# let d =
    let a = 1 in
    let b = a + a in
    let c = b + b in
      c + c;;
 val d : int = 8
```

d는 전역변수.

a, b, c는 지역변수

Functions

- Define a function with `let`:

함수이름

```
# let square x = x * x;;
 val square : int -> int = <fun>
```

- Apply the function: 함수 적용 (사용)

```
# square 2;;
- : int = 4
# square (2 + 5);;
- : int = 49
# square (square 2);;
- : int = 16
```

square 2= 2*2

square 7= 7x7

square (square 2) = (2x2)x(2x2)

- The body can be any expression:

```
# let neg x = if x < 0 then true else false;;
val neg : int -> bool = <fun>
# neg 1;;
- : bool = false
# neg (-1);;
- : bool = true
```

'int를 인수로 받아 bool을 반환하는 함수다.'

Functions

- Functions with multiple arguments:

```
# let sum_of_squares x y = (square x) + (square y);;
                            square
val sum_of_squares : int -> int -> int = <fun>
# sum_of_squares 3 4;;              sum_of_s
- : int = 25
```

OcamL은 재귀함수 구분해야 함.

- Recursive functions are defined with `let rec` construct:

```
           재귀        이름
# let rec factorial n =
    if n = 1 then 1 else n * factorial (n - 1);;
val factorial : int -> int = <fun>
# factorial 5;;
- : int = 120
```

Nameless Functions 이름 없는 함수

- Many modern programming languages provide nameless functions, e.g., ML, Scala, Java8, JavaScript, Python, etc.
- In OCaml, a function can be defined without names:

이름없는 함수 키워드

```
# fun x -> x * x;;
- : int -> int = <fun>
```

  Called *nameless* or *anonymous* functions.
- Apply nameless function as usual:

```
# (fun x -> x * x) 2;;
- : int = 4
```

- A variable can be bound to functions:

```
# let square = (fun x -> x * x);;
val square : int -> int = <fun>
```

미리 정의된 nameless를 나중에 함수 정의해도 됨.

- The followings are equivalent:

```
let square = (fun x -> x * x)
let square (x = x * x)
```

Functions are First-Class in OCaml ⇒ functional language.

In programming languages, a value is *first-class*, if the value can be

- stored in a variable, 변수에 저장 가능.
- passed as an argument of a function, and 함수 선언은 함수 사용가능
- returned from a function. 함수 인자, 반환값으로 함수 가능.

A language is often called *functional*, if functions are first-class values, e.g., ML, Scala, Java8, JavaScript, Python, Lisp, etc.

Functions are First-Class in OCaml

- Functions can be stored in variables:

```
# let square = fun x -> x * x;;
# square 2;;
- : int = 4
```

- Functions can be passed to other functions:

int int

```
# let sum_if_true test first second =
    (if test first then first else 0)
  + (if test second then second else 0);;
```

→ 3개의 인자.

```
val sum_if_true : (int -> bool) -> int -> int -> int = <fun>
```

test. if결과 if결과 함수결과

```
# let even x = (x mod 2 = 0);;
val even : int -> bool = <fun>
# sum_if_true even 3 4;;
- : int = 4
# sum_if_true even 2 4;;
- : int = 6
```

even 3 → 3%2=1 → F → 0

+ = 4.

even 4 → 4%2=0 → T → 4

Functions are First-Class in OCaml

- Functions can be also returned from a procedure:

```
# let add a = fun b -> a + b;;
val add : int -> (int -> int) = <fun>

# let add_3 = add 3;;  = fun b → 3+b;;
val add_3 : int -> int = <fun>
# add_3 1;;
- : int = 4
# add_3 2;;
- : int = 5
```

add_3 = add 3 = fun b → 3+b

함수를 인자로 사용할수 있음

Functions that manipulate functions are called *higher-order functions*.

- i.e., functions that take functions as argument or return functions
- greatly increase the expressiveness of the language

Pattern Matching

- An elegant way of doing case analysis.
- E.g., using pattern-matching, the factorial function

```
let rec fact n =
  if n = 1 then 1 else n * fact (n - 1)
```

Patter Matching을 이용해 Recursive를 구현할 수 있다.

can be written as follows:

```
let fact n =
  match n with
   1 -> 1
  |_ -> n * fact (n - 1)
```

n이 1이면 1 반환.

n이 1이 아니면. n * fact (n - 1) 반환

Pattern Matching

The nested if-then-else expression

```
let isabc c = if c = 'a' then true
              else if c = 'b' then true
              else if c = 'c' then true
              else false
```

can be written using pattern matching:

```
let isabc c =
   match c with
   'a' -> true
  |'b' -> true
  |'c' -> true
  | _  -> false
```

isabc 'a' ;; true

isabc 'd' ;; False.

or simply,

```
 let isabc c =
   match c with
   'a' | 'b' | 'c' -> true
   | _ -> false
```

Type Inference

In C or Java, types must be annotated:

```
public static int f(int n)
{
  int a = 2;
  return a * n;
}
```

In OCaml, type annotations are not mandatory:

```
# let f n =
    let a = 2 in
      a * n;;
val f : int -> int = <fun>
```

Type Inference

OCaml can infer types, no matter how complex the program is:

```
# let sum_if_true test first second =
    (if test first then first else 0)
  + (if test second then second else 0);;
val sum_if_true : (int -> bool) -> int -> int -> int = <fun>
```

OCaml compiler infers the type through the following reasoning steps:

1. the types of `first` and `second` must be `int`, because both branches of a conditional expression must have the same type,
2. the type of `test` is a function type $\alpha \to \beta$, because `test` is used as a function,
3. α must be `int`, because `test` is applied to `first`, a value of `int`,
4. β must be `bool`, because conditions must be boolean expressions, and
5. the function's return value has type `int`, because the two conditionals are of type `int` and their addition gives `int`.

Type Annotation

Explicit type annotations are possible:

```
# let sum_if_true (test : int -> bool) (x : int) (y : int) : int =
    (if test x then x else 0) + (if test y then y else 0);;
val sum_if_true : (int -> bool) -> int -> int -> int = <fun>
```

If the annotation is wrong, OCaml finds the error and report it:

```
# let sum_if_true (test : int -> int) (x : int) (y : int) : int =
    (if test x then x else 0) + (if test y then y else 0);;
Error: The expression (test x) has type int but an expression
was expected of type bool
```

Polymorphic Types

- What is the type of the program?

  ```
  let id x = x
  ```

  See how OCaml infers its type:

  ```
  # let id x = x;;
  val id : 'a -> 'a = <fun>
  ```

  The function works for values of any type:

  ```
  # id 1;;
  - : int = 1
  # id "abc";;
  - : string = "abc"
  # id true;;
  - : bool = true
  ```

- Such a function is called *polymorphic* and `'a` is a *type variable*.

Polymorphic Types

Quiz) What is the type of the function?

```
let first_if_true test x y =
  if test x then x else y
```

Tuples

- An ordered collection of values, each of which may have a different type, e.g.,

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

```
# let y = (2, "two", true);;
val y : int * string * bool = (2, "two", true)
```

- Extract each component using pattern-matching:

```
# let fst p = match p with (x,_) -> x;;
val fst : 'a * 'b -> 'a = <fun>
# let snd p = match p with (_,x) -> x;;
val snd : 'a * 'b -> 'b = <fun>
```

  or equivalently,

```
# let fst (x,_) = x;;
val fst : 'a * 'b -> 'a = <fun>
# let snd (_,x) = x;;
val snd : 'a * 'b -> 'b = <fun>
```

Tuples

- Tuple patterns can be used in `let`:

```
# let p = (1, true);;
val p : int * bool = (1, true)
# let (x,y) = p;;
val x : int = 1
val y : bool = true
```

Lists

- A finite sequence of elements, all of which have the same type, e.g.,

  `[1; 2; 3]`

  is a list of integers:

  ```
  # [1; 2; 3];;
  - : int list = [1; 2; 3]
  ```

  Note that
  - all elements must have the same type, e.g., `[1; true; 2]` is not a list,
  - the elements are ordered, e.g., `[1; 2; 3]` $\neq$ `[2; 3; 1]`, and
  - the first element is called *head*, the rest *tail*.
- `[]`: the empty list, i.e., nil. What are head and tail of `[]`?
- `[5]`: a list with a single element. What are head and tail of `[5]`?

List Examples

```
# [1;2;3;4;5];;
- : int list = [1; 2; 3; 4; 5]

# ["OCaml"; "Java"; "C"];;
- : string list = ["OCaml"; "Java"; "C"]

# [(1,"one"); (2,"two"); (3,"three")];;
- : (int * string) list = [(1, "one"); (2, "two"); (3, "three")]

# [[1;2;3];[2;3;4];[4;5;6]];;
- : int list list = [[1; 2; 3]; [2; 3; 4]; [4; 5; 6]]

# [1;"OCaml";3] ;;
Error: This expression has type string but an expression was
 expected of type int
```

List Operators

- `::` (cons): adds a single element to the front of a list, e.g.,

  ```
  # 1::[2;3];;
  - : int list = [1; 2; 3]
  # 1::2::3::[];;
  - : int list = [1; 2; 3]
  ```

  (`[1; 2; 3]` is a shorthand for `1::2::3::[]`)
- `@` (append): combines two lists, e.g.,

  ```
  # [1; 2] @ [3; 4; 5];;
  - : int list = [1; 2; 3; 4; 5]
  ```

Patterns for Lists

Pattern matching is useful for manipulating lists.

- A function to check if a list is empty:

```
# let isnil l =
    match l with
    | [] -> true
    | _  -> false;;
val isnil : 'a list -> bool = <fun>
# isnil [1];;
- : bool = false
# isnil [];;
- : bool = true
```

Patterns for Lists

- A function that computes the length of lists:

```
# let rec length l =
    match l with
      [] -> 0
    | h::t -> 1 + length t;;
val length : 'a list -> int = <fun>
# length [1;2;3];;
- : int = 3
```

We can replace pattern `h` by `_`:

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

Datatypes

- If data elements are finite, just enumerate them, e.g., “days”:

```
# type days = Mon | Tue | Wed | Thu | Fri | Sat | Sun;;
type days = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

  Construct values of type `days`:

```
# Mon;;
- : days = Mon
# Tue;;
- : days = Tue
```

  A function that manipulates the defined data:

```
# let nextday d =
  match d with
  | Mon -> Tue | Tue -> Wed | Wed -> Thu | Thu -> Fri
  | Fri -> Sa  | Sat -> Sun | Sun -> Mon ;;
val nextday : days -> days = <fun>
# nextday Mon;;
- : days = Tue
```

Datatypes

- Constructors may have arguments, e.g.,

```
# type shape = Rect of int * int | Circle of int;;
type shape = Rect of int * int | Circle of int
```

Construct values of type `shape`:

```
# Rect (2,3);;
- : shape = Rect (2, 3)
# Circle 5;;
- : shape = Circle 5
```

A function that manipulates the values of type `shape`:

```
# let area s =
    match s with
      Rect (w,h) -> w * h
    | Circle r -> r * r * 3;;
val area : shape -> int = <fun>
# area (Rect (2,3));;
- : int = 6
# area (Circle 5);;
- : int = 75
```

Datatypes

- Inductive data types, e.g.,

```
# type mylist = Nil | List of int * mylist;;
type mylist = Nil | List of int * mylist
```

  Construct values of type `mylist`:

```
# Nil;;
- : mylist = Nil
# List (1, Nil);;
- : mylist = List (1, Nil)
# List (1, List (2, Nil));;
- : mylist = List (1, List (2, Nil))
```

  A function that manipulates the data:

```
# let rec mylength l =
    match l with
      Nil -> 0
    | List (_,l') -> 1 + mylength l';;
val mylength : mylist -> int = <fun>
# mylength (List (1, List (2, Nil)));;
- : int = 2
```

Exceptions

- An exception means a runtime error: e.g.,

```
# let div a b = a / b;;
val div : int -> int -> int = <fun>
# div 10 5;;
- : int = 2
# div 10 0;;
Exception: Division_by_zero.
```

- The exception can be handled with `try ... with` constructs.

```
# let div a b =
    try
      a / b
    with Division_by_zero -> 0;;
val div : int -> int -> int = <fun>
# div 10 5;;
- : int = 2
# div 10 0;;
- : int = 0
```

Exceptions

- User-defined exceptions: e.g.,

```
# exception Problem;;
exception Problem
# let div a b =
    if b = 0 then raise Problem
    else a / b;;
val div : int -> int -> int = <fun>
# div 10 5;;
- : int = 2
# div 10 0;;
Exception: Problem.
# try
    div 10 0
  with Problem -> 0;;
- : int = 0
```

Module System

ML provides an elegant module system:

- *Structure* is a collection of types, exceptions, values, and functions, i.e., implementation details.
- *Signature* is the *interface* of the structure.

Example

The interface of a queue data structure:

- `empty`: an empty queue
- `is_empty`: a predicate testing whether `q` is empty or not
- `enq(q,x)`: the queue obtained by inserting `x` at the end of `q`
- `deq(q)`: returns a pair of the front element of `q` and the queue obtained by removing the front element of `q`
- `print(q)`: shows the contents of `q`
- `E`: the exception raised by `deq` if the queue is empty

Example

The signature of the queue data structure:

```
module type IntQueue =
sig
  type t
  exception E
  val empty : t
  val is_empty : t -> bool
  val enq : t -> int -> t
  val deq : t -> int * t
  val print : t -> unit
end
```

Example

An implementation:

```
module IntQueue : IntQueue =
struct
  type t = int list
  exception E
  let empty = []
  let enq q x = q @ [x]
  let is_empty q = q = []
  let deq q = match q with [] -> raise E | h::t -> (h, t)
  let rec print q =
    match q with
      [] -> print_string "\n"
    | h::t -> print_int h; print_string " "; print t
end
```

Example

The module can be used as follows:

```
let q0 = IntQueue.empty
let q1 = IntQueue.enq q0 1
let q2 = IntQueue.enq q1 2
let (_,q3) = IntQueue.deq q2
let _  = IntQueue.print q1
let _  = IntQueue.print q2
let _  = IntQueue.print q3
```

The program prints:

```
1
1 2
2
```

Example

The OCaml module system ensures the abstraction layer of the program:

```
let q4 = q1 @ [2]
```

produces a compile error:

```
Error: This expression has type IntQueue.t
       but an expression was expected of type 'a list
```