

OCaml 프로그래밍

오학주

고려대학교 정보대학 컴퓨터학과

January 10–11, 2019 @Tezos Blockchain Camp

소개

- ▶ 소속: 고려대학교 정보대학 컴퓨터학과
- ▶ 전공: 프로그래밍 언어, 소프트웨어 분석, 소프트웨어 보안
- ▶ 웹사이트: <http://pr1.korea.ac.kr>

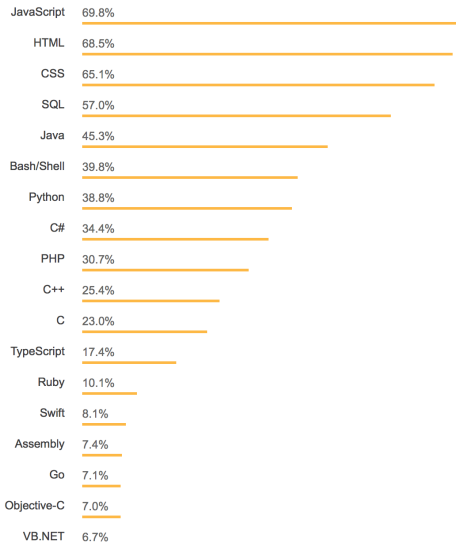
강의 내용

OCaml 프로그래밍

- ▶ Part 1: 기초 OCaml 프로그래밍 (5시간)
 1. OCaml 기본 구성
 2. 리스트와 재귀 함수
 3. 고차 함수
 4. 사용자 정의 타입
- ▶ Part 2: 고급 OCaml 프로그래밍 (5시간)

Why OCaml?

프로그래밍 언어 순위 (인기순)



프로그래밍 언어 랭킹 (연봉순)



OCaml 및 함수중심 언어의 활용 사례



Bloomberg



Jane Street



값중심, 함수중심 프로그래밍 언어

- ▶ 새롭고 중요한 생각의 틀을 제공
- ▶ 미래 프로그래밍 언어의 청사진
- ▶ 프로그래밍 언어의 근본 원리에 대한 이해
- ▶ 간결하고 아름다운 코드를 작성하는 즐거움
- ▶ ...

소프트웨어 전공자라면 갖추어야 하는 기본 소양

기본기

OCaml의 주요 특징

- ▶ 값중심, 계산형, 함수중심 프로그래밍 (Functional programming)
- ▶ 정적 타입 시스템 (Static type system)
- ▶ 자동 타입 추론 (Automatic type inference)
- ▶ 데이터 타입, 패턴 매칭 (Datatypes and pattern matching)
- ▶ 다형성 (Polymorphism)
- ▶ 모듈 (Modules)
- ▶ 메모리 재활용 (Garbage Collection)

OCaml 프로그램의 기본 단위는 식

- ▶ 프로그램을 구성하는 두 단위:
 - ▶ 명령문 (statement): 기계 상태를 변경
$$x = x + 1$$
 - ▶ 식 (expression): 상태 변경 없이 값을 계산
$$(x + y) * 2$$
- ▶ 프로그래밍 언어를 구분하는 한가지 기준:
 - ▶ 명령문을 중심으로 프로그램을 작성
 - ▶ C, C++, Java, Python, JavaScript, etc
 - ▶ often called “imperative languages”
 - ▶ 식을 중심으로 프로그램을 작성
 - ▶ ML, Haskell, Scala, Lisp, etc
 - ▶ often called “functional languages”

OCaml 프로그램의 기본 구조

값 정의들의 나열:

```
let  $x_1$  =  $e_1$   
let  $x_2$  =  $e_2$   
   $\vdots$   
let  $x_n$  =  $e_n$ 
```

- ▶ 식 e_1, e_2, \dots, e_n 을 순차적으로 계산
- ▶ 변수 x_i 는 식 e_i 의 값을 지칭

예제

- ▶ Hello World:

```
let hello = "Hello"
let world = "World"
let helloworld = hello ^ " " ^ world
let _ = print_endline helloworld
```

- ▶ 인터프리터를 이용한 실행:

```
$ ocaml helloworld.ml
Hello World
```

- ▶ REPL을 이용한 실행:

```
$ ocaml
      OCaml version 4.04.0
```

```
# let hello = "Hello";;
val hello : string = "Hello"
# let world = "World";;
val world : string = "World"
# let helloworld = hello ^ " " ^ world;;
```

산술식 (Arithmetic Expressions)

- ▶ 숫값(정수값, 실수값)을 계산하는 식

```
# 1 + 2 * 3;;  
- : int = 7  
# 1.1 +. 2.2 *. 3.3;;  
- : float = 8.36
```

- ▶ 정수값을 위한 산술 연산자:

$a + b$	덧셈	a / b	나눗셈 (몫)
$a - b$	뺄셈	$a \bmod b$	나눗셈 (나머지)
$a * b$	곱셈		

- ▶ 실수값을 위한 산술 연산자: $+. , -. , *. , /. , \dots$
- ▶ 정수 타입과 실수 타입을 명확히 구분

```
# 3 + 2.0;;  
Error: This expression has type float but an expression  
was expected of type int  
# 3 + int_of_float 2.0;;  
- : int = 5
```

논리식 (Boolean Expressions)

- ▶ 논리값 (참, 거짓)을 계산하는 식

```
# true;;  
- : bool = true  
# false;;  
- : bool = false
```

- ▶ 비교 연산자 (산술식들로부터 논리식을 구성):

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

- ▶ 논리 연산자 (논리식들로부터 새로운 논리식을 구성):

```
# true && (false || not false);;  
- : bool = true  
# (2 > 1) && (3 > 2);;  
- : bool = true
```

기본값 (Primitive Values)

- ▶ 프로그래밍 언어에서 기본적으로 제공하는 값
- ▶ OCaml은 정수(integer), 실수(float), 논리(boolean), 문자(character), 문자열(string), 유닛(unit)값을 제공

```
# 'c';;
```

```
- : char = 'c'
```

```
# "Objective " ^ "Caml";;
```

```
- : string = "Objective Caml"
```

```
# ();;
```

```
- : unit = ()
```


정적 타입 시스템 (Static Type System)

- ▶ 타입 오류가 있는 프로그램은 컴파일러를 통과하지 못함:

```
# 1 + true;;
```

Error: This expression has type bool but an expression was expected of type int

- ▶ 발생 가능한 모든 타입 오류를 실행전에 찾아냄
- ▶ OCaml로 작성된 프로그램의 안정성이 높은 이유

정적/동적 타입 시스템

타입 시스템을 기준으로 한 프로그래밍 언어의 구분:

- ▶ 정적 타입 언어 (Statically typed languages)
 - ▶ 타입 체킹을 컴파일 시간에 수행
 - ▶ 타입 오류를 프로그램 실행전에 검출
 - ▶ C, C++, Java, ML, Scala, etc
- ▶ 동적 타입 언어 (Dynamically typed languages):
 - ▶ 타입 체킹을 실행중에 수행
 - ▶ 타입 오류를 프로그램 실행중에 검출
 - ▶ Python, JavaScript, Ruby, Lisp, etc

정적 타입 언어들은 다시 두가지로 구분:

- ▶ 안전한(Type-safe) 언어:
 - ▶ 타입 체킹을 통과한 프로그램은 실행중에 타입 오류가 없음.
 - ▶ 모든 타입 오류가 실행전에 검출됨.
 - ▶ 프로그램이 실행중에 비정상적으로 종료되지 않음.
 - ▶ ML, Haskell, Scala
- ▶ 안전하지 않은 (Unsafe) 언어:
 - ▶ 타입 체킹을 통과해도 실행중에 여전히 타입 오류가 발생 가능
 - ▶ C, C++

장단점

정적 타입 언어:

- ▶ (+) 타입 오류가 프로그램 개발중에 검출됨
- ▶ (+) 실행중에 타입 체크를 하지 않으므로 실행이 효율적
- ▶ (-) 동적 언어보다 경직됨

동적 타입 언어:

- ▶ (-) 타입 오류가 실행중에 예상치 못하게 나타남
- ▶ (+) 다양한 언어 특징을 유연하게 제공하기 쉬움
- ▶ (+) 쉽고 빠른 프로토타이핑

조건식 (Conditional Expression)

`if e_1 then e_2 else e_3`

- ▶ e_1 은 반드시 논리식이여야 함. 즉 e_1 의 값은 참 또는 거짓

```
# if 1 then 2 else 3;;
```

Error: This expression has type int but an expression was expected of type bool

- ▶ 조건식의 값은 e_1 의 값에 따라서 결정

```
# if 2 > 1 then 0 else 1;;
```

```
- : int = 0
```

```
# if 2 < 1 then 0 else 1;;
```

```
- : int = 1
```

- ▶ e_2 와 e_3 는 타입이 같아야 함

```
# if true then 1 else true;;
```

Error: This expression has type bool but an expression was expected of type int

함수식 (Function Expression)

`fun x -> e`

- ▶ 형식 인자 (formal parameter)가 x 이고 몸통 (body)이 e 인 함수값 (function value)을 생성
- ▶ 함수의 예:

- ▶ `fun x -> x + 1`
- ▶ `fun y -> y * y`
- ▶ `fun x -> if x > 0 then x + 1 else x * x`
- ▶ `fun x -> fun y -> x + y`
- ▶ `fun x -> fun y -> fun z -> x + y + z`

- ▶ 설탕 구조 (syntactic sugar):

`fun x1 ... xn -> e`

- ▶ `fun x y -> x + y`
- ▶ `fun x y z -> x + y + z`

함수식 (Function Expression)

```
# fun x -> x + 1;;  
- : int -> int = <fun>  
  
# fun y -> y * y;;  
- : int -> int = <fun>  
  
# fun x -> if x > 0 then x + 1 else x * x;;  
- : int -> int = <fun>  
  
# fun x -> fun y -> x + y;;  
- : int -> int -> int = <fun>  
  
# fun x -> fun y -> fun z -> x + y + z;;  
- : int -> int -> int -> int = <fun>  
  
# fun x y z -> x + y + z;;  
- : int -> int -> int -> int = <fun>
```

함수 호출식(Function Call)

$e_1 \ e_2$

- ▶ e_1 : 함수값을 만들어내는 식이어야 함
- ▶ e_2 : 함수 전달 인자 (actual parameter)

```
# (fun x -> x * x) 3;;
```

```
- : int = 9
```

```
# (fun x -> if x > 0 then x + 1 else x * x) 1;;
```

```
- : int = 2
```

```
# (fun x -> if x > 0 then x + 1 else x * x) (-2);;
```

```
- : int = 4
```

```
# (fun x -> fun y -> x + y) 1 2;;
```

```
- : int = 3
```

```
# (fun x -> fun y -> fun z -> x + y + z) 1 2 3;;
```

```
- : int = 6
```

- ▶ e_2 는 임의의 식이 가능:

```
# (fun f -> f 1) (fun x -> x * x);;
```

```
- : int = 1
```

```
# (fun x -> x * x) ((fun x -> if x > 0 then 1 else 2) 3);;
```

```
- : int = 1
```

Let Expression

값에 이름 붙이기:

`let x = e1 in e2`

- ▶ e_1 의 값을 x 라고 하고 e_2 를 계산
 - ▶ x : 값의 이름 (변수)
 - ▶ e_1 : 정의식 (binding expression)
 - ▶ e_2 : 몸통식 (body expression)
- ▶ x 의 유효범위(scope)는 e_2

```
# let x = 1 in x + x;;  
- : int = 2  
# let x = 1 in x + 1;;  
- : int = 2  
# (let x = 1 in x) + x;;  
Error: Unbound value x  
# (let x = 1 in x) + (let x = 2 in x);;  
- : int = 3
```


Let Expression

- ▶ e_1 과 e_2 는 임의의 식이 될 수 있음

```
# let x = (let y = 1 in y + 1) in x + 1;;  
- : int = 3  
# let x = 1 in  
    let y = 2 in  
        x + y;;  
- : int = 3
```

- ▶ 함수 정의:

```
# let square = fun x -> x * x in square 2;;  
- : int = 4  
# let add x y = x + y in add 1 2;;  
- : int = 3
```

- ▶ 재귀 함수 정의:

```
# let rec fact a = if a = 1 then 1 else a * fact (a - 1);;  
val fact : int -> int = <fun>  
# fact 5;;  
- : int = 120
```

연습 문제

- ▶ `let a = 1 in
 let b = a + a in
 let c = b + b in
 c + c`
- ▶ `let x = 1 in
 let f y = x + y in
 let x = 2 in
 f x`
- ▶ `let x = 1 in ((let x = 2) + x)`

함수값을 자유롭게 사용 가능

프로그램에서 함수도 최대의 자유도를 가짐 (First-class values):

- ▶ 함수를 지칭하는 이름을 만들 수 있음:

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

- ▶ 함수를 다른 함수의 인자로 전달 가능:

```
# 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>  
  
# 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
```

함수값을 자유롭게 사용 가능

- ▶ 함수를 다른 함수의 반환 값으로 전달 가능

```
# let plus_a a = fun b -> a + b;;  
val plus_a : int -> int -> int = <fun>  
  
# let f = plus_a 3;;  
val f : int -> int = <fun>  
# f 1;;  
- : int = 4  
# f 2;;  
- : int = 5
```

- ▶ 고차함수(Higher-order function): 다른 함수를 인자로 받거나 반환하는 함수. 언어의 표현력을 높이는 주된 특징.

패턴 매칭 (Pattern Matching)

- ▶ 패턴 매칭을 이용한 값의 구조 분석
- ▶ 팩토리얼 예제:

```
let rec factorial a =  
  if a = 1 then 1 else a * factorial (a - 1)
```

```
let factorial a =  
  match a with  
  1 -> 1  
  | _ -> a * factorial (a - 1)
```

패턴 매칭 (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
```

or simply,

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

자동 타입 추론 (Automatic Type Inference)

- ▶ C나 Java 에서는 타입을 생략할 수 없음:

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

- ▶ OCaml 에서는 타입을 생략 가능. 컴파일러가 자동으로 유추:

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

타입 유추 알고리즘

```
# 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 컴파일러가 타입을 유추하는 과정:

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 $\alpha \rightarrow \beta$, because `test` is used as a function.
3. α must be of `int`, because `test` is applied to `first`, a value of `int`.
4. β must be of `bool`, because conditions must be boolean expressions.
5. The return value of the function has type `int`, because the two conditional expressions are of `int` and their addition gives `int`.

타입을 직접 명시하는 것도 가능

```
# 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>
```

컴파일러가 명시한 타입이 올바른지 자동으로 검증:

```
# 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
```

다형 타입 (Polymorphic Types)

- ▶ 아래 프로그램의 타입은?

```
let id x = x
```

OCaml 타입 추론 결과:

```
# let id x = x;;
```

```
val id : 'a -> 'a = <fun>
```

임의의 값에 대해 작동하는 다형 타입 함수:

```
# id 1;;
```

```
- : int = 1
```

```
# id "abc";;
```

```
- : string = "abc"
```

```
# id true;;
```

```
- : bool = true
```

- ▶ 아래 프로그램의 타입은?

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

예외 처리

- ▶ An exception means a run-time 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
```

예외 처리

- ▶ 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
```

리스트와 재귀 함수

튜플 (Tuples)

- ▶ 순서가 있는 값의 묶음. 각 구성 요소는 다른 값을 가질 수 있음:

```
# let x = (1, "one");;  
val x : int * string = (1, "one")  
  
# let y = (2, "two", true);;  
val y : int * string * bool = (2, "two", true)
```

- ▶ 패턴 매칭을 이용하여 각 구성 요소를 추출 가능:

```
# 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>
```

튜플 (Tuples)

- ▶ let에서 패턴 사용 가능:

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

리스트 (Lists)

- ▶ 유한한 원소들의 나열:

```
# [1; 2; 3];;
```

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

- ▶ 순서가 중요: e.g., [3;4], [4;3], [3;4;3], [3;3;4]

- ▶ 모든 원소가 같은 타입이어야 함

- ▶ [(1, "one"); (2, "two")] : (int * string) list

- ▶ [[]; [1]; [1;2]; [1;2;3]] : (int list) list

- ▶ 리스트의 원소는 변경이 불가능 (immutable)

- ▶ 리스트의 첫 원소를 *head*, 나머지를 *tail*이라고 부름

```
# List.hd [5];;
```

```
- : int = 5
```

```
# List.tl [5];;
```

```
- : int list = []
```


리스트 예제

- ▶ # [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

리스트를 만드는 방법

- ▶ []: 빈 리스트(nil)
- ▶ :: (cons): 리스트의 앞에 하나의 원소를 추가:

```
# 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): 두 리스트를 이어붙이기:

```
# [1; 2] @ [3; 4; 5];;
```

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

리스트 패턴

리스트를 다룰 때 패턴 매칭이 매우 유용하게 쓰임

- ▶ Ex1) 빈 리스트인지 검사하는 함수:

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

리스트 패턴

- ▶ Ex2) 리스트의 길이를 구하는 함수:

```
# 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
```

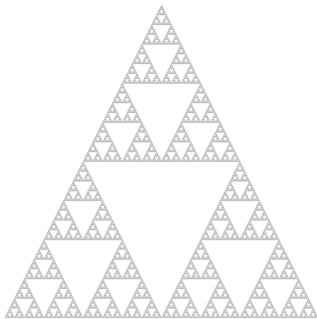
쓰이지 않는 구성요소는 _ 로 대체 가능:

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

- ▶ 리스트를 다루는 함수는 주로 재귀적으로 정의

재귀적 사고의 강력함

Ex) 아래 도형을 그리는 프로그램?



재귀적으로 문제 풀기

- ▶ 주어진 문제의 크기가 충분히 작다면 직접 푼다.
- ▶ 문제가 충분히 작지 않다면,
 1. 문제를 동일한 구조를 가지는 작은 문제들로 쪼갬다.
 2. 쪼개진 문제들을 재귀적으로 푼다.
 3. 결과를 합쳐서 원래 문제의 답을 구한다.

리스트 길이 구하기 (list length)

```
# length [];;  
- : int = 0  
# length [1;2;3];;  
- : int = 3  
  
let rec length l =  
  match l with  
  | [] -> 0  
  | hd::tl -> 1 + length tl
```

예제 1: 리스트 이어붙이기 (append)

```
# append [1; 2; 3] [4; 5; 6; 7];;  
- : int list = [1; 2; 3; 4; 5; 6; 7]  
# append [2; 4; 6] [8; 10];;  
- : int list = [2; 4; 6; 8; 10]  
  
let rec append l1 l2 =
```


예제 2: 리스트 뒤집기 (reverse)

```
val reverse : 'a list -> 'a list = <fun>  
# reverse [1; 2; 3];;  
- : int list = [3; 2; 1]  
# reverse ["C"; "Java"; "OCaml"];;  
- : string list = ["OCaml"; "Java"; "C"]  
  
let rec reverse l =
```

예제 3: n번째 원소 찾기 (nth-element)

```
# nth [1;2;3] 0;;  
- : int = 1  
# nth [1;2;3] 1;;  
- : int = 2  
# nth [1;2;3] 2;;  
- : int = 3  
# nth [1;2;3] 3;;  
Exception: Failure "list is too short".
```

```
let rec nth l n =  
  match l with  
  | [] -> raise (Failure "list is too short")  
  | hd::tl -> (* ... *)
```

예제 4: 첫번째 원소 지우기 (remove-first)

```
# remove_first 2 [1; 2; 3];;  
- : int list = [1; 3]  
# remove_first 2 [1; 2; 3; 2];;  
- : int list = [1; 3; 2]  
# remove_first 4 [1;2;3];;  
- : int list = [1; 2; 3]  
# remove_first [1; 2] [[1; 2; 3]; [1; 2]; [2; 3]];;  
- : int list list = [[1; 2; 3]; [2; 3]]  
  
let rec remove_first a l =
```

예제 5: 정렬된 리스트에 원소 삽입 (insert)

```
# insert 2 [1;3];;  
- : int list = [1; 2; 3]  
# insert 1 [2;3];;  
- : int list = [1; 2; 3]  
# insert 3 [1;2];;  
- : int list = [1; 2; 3]  
# insert 4 [];;  
- : int list = [4]  
  
let rec insert a l =
```

예제 6: 삽입 정렬 (insertion sort)

```
let rec sort l =
```

예제 6: 삽입 정렬 (insertion sort)

```
let rec sort l =
```

cf) Compare with “C-style” non-recursive version:

```
for (c = 1 ; c <= n - 1; c++) {  
    d = c;  
    while ( d > 0 && array[d] < array[d-1]) {  
        t          = array[d];  
        array[d]   = array[d-1];  
        array[d-1] = t;  
        d--;  
    }  
}
```

cf) 명령형 vs. 함수형 프로그래밍

- ▶ Imperative programming focuses on describing **how** to accomplish the given task:

```
int factorial (int n) {  
    int i; int r = 1;  
    for (i = 0; i < n; i++)  
        r = r * i;  
    return r;  
}
```

Imperative languages encourage to use statements and loops.

- ▶ Functional programming focuses on describing **what** the program must accomplish:

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

Functional languages encourage to use expressions and recursion.

cf) 명령형 vs. 함수형 프로그래밍

함수형 프로그래밍

- ▶ 높은 추상화 수준에서 프로그래밍
- ▶ 견고한 소프트웨어를 작성하기 쉬움
- ▶ 소프트웨어의 실행 성질을 분석하기 쉬움

명령형 프로그래밍

- ▶ 낮은 추상화 수준에서 프로그래밍
- ▶ 견고한 소프트웨어를 작성하기 어려움
- ▶ 소프트웨어의 실행 성질을 분석하기 어려움

cf) 오해: 재귀 함수는 비싸다?

- ▶ In C and Java, we are encouraged to avoid recursion because function calls consume additional memory.

```
void f() { f(); }           /* stack overflow */
```

- ▶ This is not true in functional languages. The same program in ML iterates forever:

```
let rec f () = f ()
```

- ▶ 단순히 함수가 재귀적으로 정의되었다고 계산과정이 비싼것이 아님.

Tail-Recursive Functions

More precisely, *tail-recursive functions* are not expensive in ML. A recursive call is a tail call if there is nothing to do after the function returns.

- ▶

```
let rec last l =  
  match l with  
  | [a] -> a  
  | _::tl -> last tl
```
- ▶

```
let rec factorial a =  
  if a = 1 then 1  
  else a * factorial (a - 1)
```

Languages like ML, Scheme, Scala, and Haskell do *tail-call optimization*, so that tail-recursive calls do not consume additional amount of memory.

cf) Transforming to Tail-Recursive Functions

Non-tail-recursive factorial:

```
let rec factorial a =  
  if a = 1 then 1  
  else a * factorial (a - 1)
```

Tail-recursive version:

```
let rec fact product counter maxcounter =  
  if counter > maxcounter then product  
  else fact (product * counter) (counter + 1) maxcounter  
  
let factorial n = fact 1 1 n
```

연습 문제 1: range

두 수 n, m ($n \leq m$)을 받아서 n 이상 m 이하의 수로 구성된 리스트를 반환하는 함수 `range`를 작성하시오:

```
range : int -> int -> int list
```

예를 들어, `range 3 7` 는 `[3;4;5;6;7]`를 계산한다.

연습 문제 2: concat

리스트의 리스트를 받아서 모든 원소를 포함하는 하나의 리스트를 반환하는 함수 `concat`을 작성하시오:

```
concat: 'a list list -> 'a list
```

예를 들어,

```
concat [[1;2];[3;4;5]] = [1;2;3;4;5]
```

연습 문제 3: zipper

두 리스트 a 와 b 를 순차적으로 결합하는 함수 `zipper`를 작성하시오:

```
zipper: int list -> int list -> int list
```

순차적인 결합이란 리스트 a 의 i 번째 원소가 리스트 b 의 i 번째 원소 앞에 오는 것을 의미한다. 짝이 맞지 않는 원소들은 뒤에 순서대로 붙인다.

```
# zipper [1;3;5] [2;4;6];;  
- : int list = [1; 2; 3; 4; 5; 6]  
# zipper [1;3] [2;4;6;8];;  
- : int list = [1; 2; 3; 4; 6; 8]  
# zipper [1;3;5;7] [2;4];;  
- : int list = [1; 2; 3; 4; 5; 7]
```

연습 문제 4: unzip

두 원소를 가지는 튜플의 리스트를 두 리스트로 분해하는 함수 `unzip`을 작성하시오:

```
unzip: ('a * 'b) list -> 'a list * 'b list
```

예를 들어,

```
unzip [(1,"one");(2,"two");(3,"three")]
```

은 `([1;2;3],["one";"two";"three"])`을 계산한다.

연습 문제 5: drop

리스트 l 과 정수 n 을 받아서 l 의 첫 n 개 원소를 제외한 나머지 리스트를 구하는 함수 `drop`을 작성하시오:

```
drop : 'a list -> int -> 'a list
```

예를 들어,

```
drop [1;2;3;4;5] 2 = [3; 4; 5]
```

```
drop [1;2] 3 = []
```

```
drop ["C"; "Java"; "OCaml"] 2 = ["OCaml"]
```


고차 함수

고차 함수 (Higher-Order Functions)

- ▶ 다른 함수를 인자로 받거나 리턴하는 함수
- ▶ 높은 추상화(abstraction) 수준에서 프로그램을 작성하게 함
- ▶ 간결하고 재사용 가능한 코드를 작성하는데 있어서 필수

추상화 (Abstraction)

- ▶ 복잡한 개념에 이름을 붙여서 속내용을 모른채 사용할 수 있도록 하는 장치
- ▶ 좋은 프로그래밍 언어는 강력한 추상화 기법을 제공
- ▶ ex) $2^3 + 3^3 + 4^3$ 을 계산하는 프로그램을 작성하는 방법:
 - ▶ $2*2*2 + 3*3*3 + 4*4*4$
 - ▶

```
let cube n = n * n * n
in cube 2 + cube 3 + cube 4
```
- ▶ 모든 프로그래밍 언어는 추상화 도구로 변수와 함수를 제공
 - ▶ 변수: 반복적으로 사용하는 값에 붙인 이름
 - ▶ (일차)함수: 반복적으로 사용하는 연산에 붙인 이름
- ▶ 고차 함수: 반복되는 프로그래밍 패턴에 붙인 이름

List.map

Three similar functions:

```
let rec inc_all l =  
  match l with  
  | [] -> []  
  | hd::tl -> (hd+1)::(inc_all tl)
```

```
let rec square_all l =  
  match l with  
  | [] -> []  
  | hd::tl -> (hd*hd)::(square_all tl)
```

```
let rec cube_all l =  
  match l with  
  | [] -> []  
  | hd::tl -> (hd*hd*hd)::(cube_all tl)
```

List.map

The code pattern can be captured by the higher-order function `map`:

```
let rec map f l =  
  match l with  
  | [] -> []  
  | hd::tl -> (f hd)::(map f tl)
```

With `map`, the functions can be defined as follows:

```
let inc x = x + 1  
let inc_all l = map inc l  
  
let square x = x * x  
let square_all l = map square l  
  
let cube x = x * x * x  
let cube_all l = map cube l
```

Or, using nameless functions:

```
let inc_all l = map (fun x -> x + 1) l  
let square_all l = map (fun x -> x * x) l  
let cub_all l = map (fun x -> x * x * x) l
```

퀴즈

1. map의 타입은?
2. map (fun x mod 2 = 1) [1;2;3;4] 의 값은?

List.filter

```
let rec even l =  
  match l with  
  | [] -> []  
  | hd::tl ->  
    if hd mod 2 = 0 then hd::(even tl)  
    else even tl
```

```
let rec greater_than_five l =  
  match l with  
  | [] -> []  
  | hd::tl ->  
    if hd > 5 then hd::(greater_than_five tl)  
    else greater_than_five tl
```

List.filter

```
filter : ('a -> bool) -> 'a list -> 'a list
```

- ▶ `even = filter (fun x -> x mod 2 = 0)`
- ▶ `greater_than_five = filter (fun x -> x > 5)`

List.fold_right

Two similar functions:

```
let rec sum l =  
  match l with  
  | [] -> 0  
  | hd::tl -> hd + (sum tl)
```

```
let rec prod l =  
  match l with  
  | [] -> 1  
  | hd::tl -> hd * (prod tl)
```

```
# sum [1; 2; 3; 4];;  
- : int = 10  
# prod [1; 2; 3; 4];;  
- : int = 24
```

List.fold_right

The code pattern can be captured by the higher-order function fold:

```
let rec fold_right f l a =  
  match l with  
  | [] -> a  
  | hd::tl -> f hd (fold_right f tl a)
```

```
let sum lst = fold_right (fun x y -> x + y) lst 0  
let prod lst = fold_right (fun x y -> x * y) lst 1
```

ML 프로그램에서 loop의 역할은 곧 fold 의 역할이라고 한다.

fold_right vs. fold_left

```
let rec fold_right f l a =  
  match l with  
  | [] -> a  
  | hd::tl -> f hd (fold_right f tl a)
```

```
let rec fold_left f a l =  
  match l with  
  | [] -> a  
  | hd::tl -> fold_left f (f a hd) tl
```

차이점

▶ 순서

- ▶ `fold_right`은 리스트를 오른쪽에서 왼쪽으로:

```
fold_right f [x;y;z] init = f x (f y (f z init))
```

- ▶ `fold_left`는 리스트를 왼쪽에서 오른쪽으로:

```
fold_left f init [x;y;z] = f (f (f init x) y) z
```

- ▶ 결합법칙이 성립하지 않는 `f`에 대해서 결과가 다를 수 있음

▶ 타입

```
fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

```
fold_left  : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

- ▶ `fold_left`는 끝재귀호출(tail-recursion)

성능상으로는 `fold_left`가 훨씬 유용

예제

- ▶ `let rec length l =
 match l with
 | [] -> 0
 | hd::tl -> 1 + length tl`
- ▶ `let rec reverse l =
 match l with
 | [] -> []
 | hd::tl -> (reverse tl) @ [hd]`
- ▶ `let rec is_all_pos l =
 match l with
 | [] -> true
 | hd::tl -> (hd > 0) && (is_all_pos tl)`
- ▶ `map f l =`
- ▶ `filter f l =`

연습 문제 1

고차 함수 `sigma`를 작성하시오:

```
sigma : (int -> int) -> int -> int -> int
```

`sigma f a b`는 다음을 계산한다.

$$\sum_{i=a}^b f(i).$$

예를 들어,

```
sigma (fun x -> x) 1 10
```

의 값은 55이고,

```
sigma (fun x -> x*x) 1 7
```

는 140을 계산한다.

연습 문제 2

`fold_right`을 이용하여 고차 함수 `all`을 작성하시오:

```
all : ('a -> bool) -> 'a list -> bool
```

`all p l`은 리스트 `l`의 모든 원소들이 함수 `p`의 값을 참으로 만드는지 여부를 나타낸다. 예를 들어,

```
all (fun x -> x > 5) [7;8;9]
```

는 `true`를 계산한다.

연습 문제 3

`fold_left`를 이용하여 정수 리스트를 숫자로 변환하는 함수를 작성하시오:

```
lst2int : int list -> int
```

예를 들어, `lst2int [1;2;3]`는 123을 계산한다. 리스트의 원소들은 0이상 9이하의 수라고 가정한다.

함수를 반환하는 함수의 예

- ▶ 함수 합성 (composition): Let f and g be two one-argument functions. The composition of f after g is defined to be the function $x \mapsto f(g(x))$.

- ▶ In OCaml:

```
let compose f g = fun x -> f(g(x))
```

What is the value of the expression?

```
((compose square inc) 6)
```

연습 문제 4

함수 f 와 인자 a 를 받아서 f 를 a 에 두 번 적용한 결과를 반환하는 함수 `double`을 작성하시오:

```
double: ('a -> 'a) -> 'a -> 'a
```

예를 들어,

```
# let inc x = x + 1;;
val inc : int -> int = <fun>
# let mul x = x * 2;;
val mul : int -> int = <fun>
# (double inc) 1;;
- : int = 3
# (double mul) 1;;
- : int = 4
```

더블은 2번 하는거 더블 더블= 4번하는거
더블더블 더블 은 4번하는걸 2번하니깐 16번해0

아래 식의 값은?

```
▶ ((double double) inc) 0
```

```
▶ (double double) mul 2
```

```
▶ ((double (double double)) inc) 5
```

연습 문제 5

고차 함수 `iter`를 작성하시오:

```
iter : int * (int -> int) -> (int -> int)
```

정의는 다음과 같다:

$$\text{iter}(n, f) = \underbrace{f \circ \dots \circ f}_n.$$

$n = 0$ 이면, $\text{iter}(n, f)$ 은 항등함수(identity function)으로 정의한다. $n > 0$ 이면, $\text{iter}(n, f)$ 은 f 를 n 번 반복 적용하는 함수이다. 예를 들어,

```
iter(n, fun x -> 2+x) 0
```

는 $2 \times n$ 를 계산한다.

사용자 정의 타입

이미 있는 타입에 새로운 이름 붙이기

```
type var = string  
type vector = float list  
type matrix = float list list  
  
let rec transpose : matrix -> matrix  
=fun m -> ...
```

새로운 타입 만들기 (Variants)

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 the type:

```
# 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
```

새로운 타입 만들기 (Variants)

Constructors can be associated with values, e.g.,

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

Construct values of the type:

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

A function that manipulates the data:

```
# 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
```

새로운 타입 만들기 (Variants)

Inductive data types, e.g.,

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

Construct values of the type:

```
# 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
```


새로운 타입 만들기 (Parameterized Variants)

- ▶ 임의의 타입을 담을 수 있는 리스트를 만드려면?
 - ▶ e.g., lists of ints, lists of strings, lists of lists of ints, etc
- ▶ 다른 타입을 인자로 가지는 타입을 정의 가능:

```
# type 'a mylist = Nil | Cons of 'a * 'a mylist;;
type 'a mylist = Nil | Cons of 'a * 'a mylist
# let l1 = Cons (3, Nil) ;;
val l1 : int mylist = Cons (3, Nil)
# let l2 = Cons ("three", Nil);;
val l2 : string mylist = Cons ("three", Nil)
# let rec length l = match l with
  | Nil -> 0
  | Cons (_,t) -> 1 + length t;;
val length : 'a mylist -> int = <fun>
```

- ▶ mylist: 다른 타입을 인자로 받아서 다른 타입을 만들어내는 타입 생성자 (type constructor)
- ▶ int mylist, float mylist, (int * float) mylist, etc

연습 문제 1: 이진 나무 (ver. 1)

이진 나무 (binary tree)는 다음과 같이 정의할 수 있다:

```
type btree =  
  Empty  
  | Node of int * btree * btree
```

예를 들어,

```
let t1 = Node (1, Empty, Empty)  
let t2 = Node (1, Node (2, Empty, Empty), Node (3, Empty, Empty))
```

이진 나무에서 주어진 원소가 존재하는지 여부를 반환하는 함수 `mem`을 작성하시오:

```
mem: int -> btree -> bool
```

예를 들어,

```
mem 1 t1
```

는 `true` 이고,

```
mem 4 t2
```

는 `false`이다.

연습 문제 2: 이진 나무 (ver. 2)

이진 나무를 다음과 같이 정의하자.

```
type btree =  
  | Leaf of int  
  | Left of btree  
  | Right of btree  
  | LeftRight of btree * btree
```

예를 들어,

```
Left (LeftRight (Leaf 1, Leaf 2))
```

이진 나무의 왼쪽, 오른쪽 자식을 재귀적으로 모두 교환하는 함수 `mirror`를 작성하시오:

```
mirror : btree -> btree
```

예를 들어,

```
mirror (Left (LeftRight (Leaf 1, Leaf 2)))
```

는 `Right (LeftRight (Leaf 2, Leaf 1))`를 계산한다.

연습 문제 3: 계산기 (ver. 1)

```
type exp =  
  Const of int  
|Minus of exp  
|Plus of exp * exp  
|Mult of exp * exp
```

위 산술식의 값을 계산하는 함수

```
calc: exp -> int
```

를 작성하시오. 예를 들어,

```
calc (Plus (Const 1, Const 2))
```

는 3을 계산한다.

연습 문제 4: 계산기 (ver. 2)

```
type exp = X
  | INT of int
  | ADD of exp * exp
  | SUB of exp * exp
  | MUL of exp * exp
  | DIV of exp * exp
  | SIGMA of exp * exp * exp
```

위 식에 대한 계산기를 작성하시오:

```
calculator : exp -> int
```

예를 들어,

$$\sum_{x=1}^{10} (x * x - 1)$$

는 다음과 같이 표현되며

```
SIGMA(INT 1, INT 10, SUB(MUL(X, X), INT 1))
```

그 계산 결과는 375이다.

리뷰

- ▶ OCaml 기본 구성: 식, 변수, 함수, 유효범위
- ▶ 리스트와 재귀 함수
- ▶ 고차 함수
- ▶ 사용자 정의 타입

감사합니다!