함수형 프로그래밍 소개

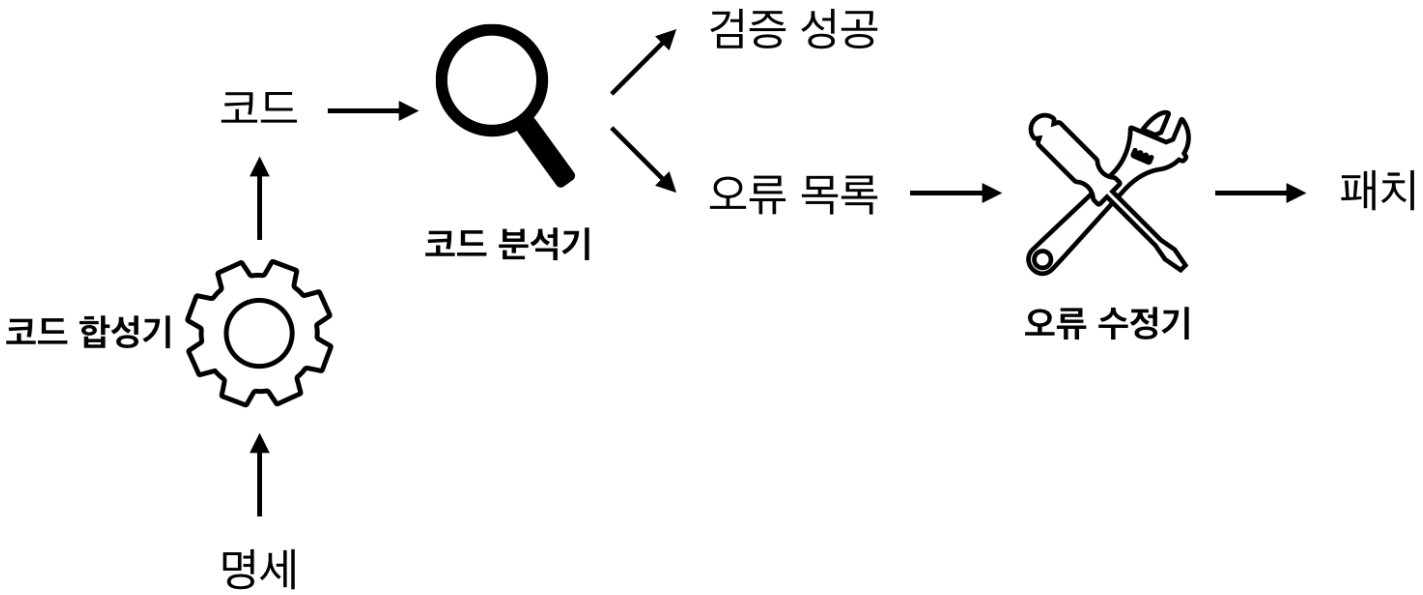
오학주

고려대학교 정보대학 컴퓨터학과

September 19, 2019

소개

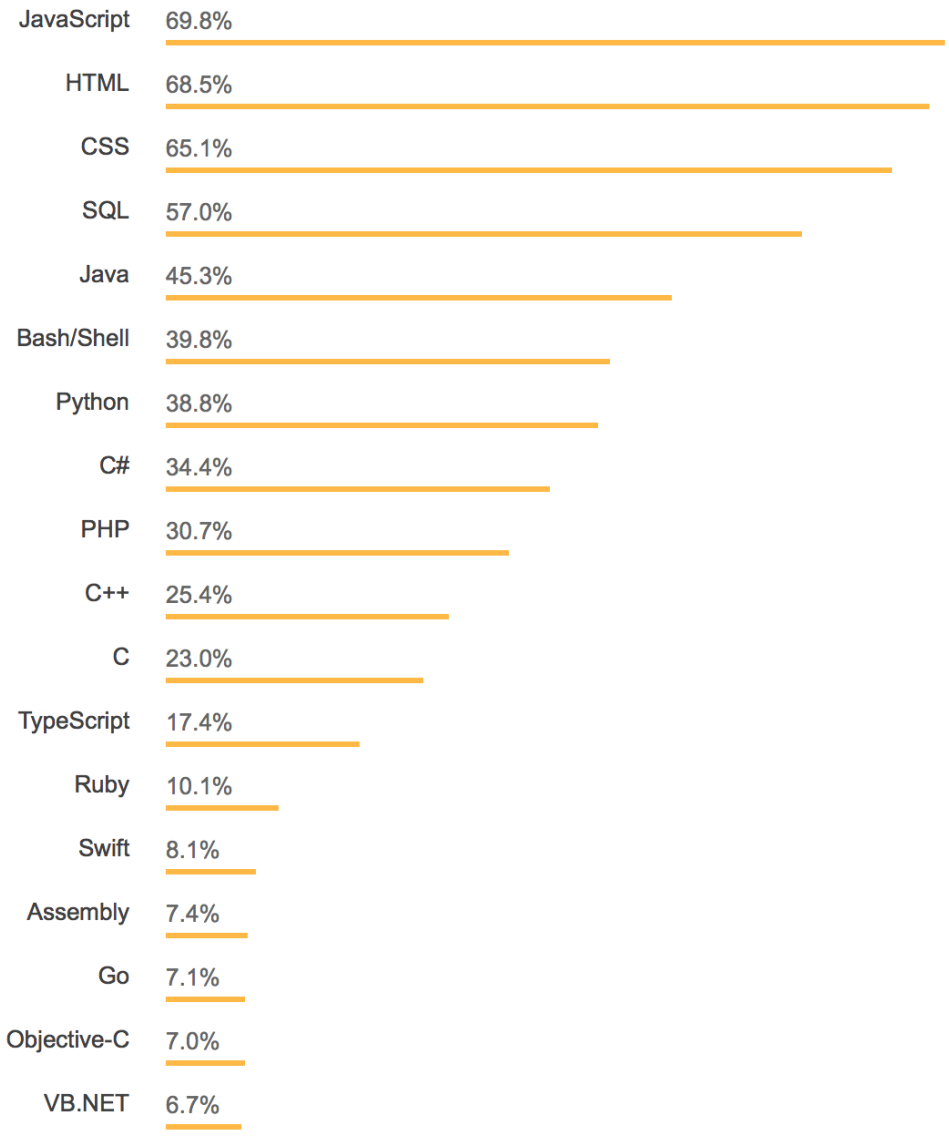
- 소속: 고려대학교 정보대학 컴퓨터학과
- 분야: 프로그래밍 언어, 소프트웨어 분석, 소프트웨어 보안



- 웹페이지: `http://prl.korea.ac.kr`

함수형 프로그래밍
(Functional Programming)

프로그래밍 언어 순위 (인기순)



https://insights.stackoverflow.com/survey/2018

프로그래밍 언어 순위 (연봉순)

https://insights.stackoverflow.com/survey/2018

함수형 언어의 활용 사례


facebook


docker


Bloomberg


FOURSQUARE


Jane Street


twitter


LinkedIn


Microsoft

함수형 프로그래밍 언어의 특징

- 함수를 특별하지 않게, 다른 값과 동일하게 취급 (functions are first-class value)
  - 예: 이름을 붙이지 않고 함수를 정의할 수 있고, 다른 함수의 인자로 넘길 수 있고, 다른 함수로부터 리턴할 수 있음
  - 언어의 표현력을 높여주어 간결한 프로그램 작성이 가능
- 한번 정의된 값은 변하지 않음 (values are immutable)
  - 예: `str.replace('a', 'b')`는 새로운 문자열을 생성할 뿐 다른 부수효과(side effect)를 가지지 않음
  - 프로그램의 복잡도를 낮추어 이해하기 쉬운 프로그램 작성이 가능
- 정적 타입 시스템 (static type system)
  - 예: `a+b`에서 타입 오류가 발생할 수 있는지 컴파일 단계에서 검증
  - 발생가능한 모든 타입 오류를 컴파일 단계에서 검출하여 테스팅 불필요

Programming in OCaml

- 함수형 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:

$$\begin{aligned}
&\texttt{let } x_1 = e_1 \\
&\texttt{let } x_2 = e_2 \\
&\qquad \vdots \\
&\texttt{let } x_n = e_n
\end{aligned}$$

- 식 $e_1, e_2, \ldots, e_n$을 순차적으로 계산
- 변수 x_i는 식e_i의 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
```

- REPL(대화형 실행 환경)을 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)

- 정수 또는 실수값을 계산

```
# 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 `mod` b	나눗셈 (나머지)
a `*` b	곱셈		

- 실수값을 위한 산술 연산자: `+.`, `-.`, `*.`, `/.`, ...
- 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)

$$\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$$

- e_1은 반드시 논리식이어야 함. 즉 e_1의 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_1의 값에 따라서 결정

```
# if 2 > 1 then 0 else 1;;
- : int = 0
# if 2 < 1 then 0 else 1;;
- : int = 1
```

- e_2와 e_3는 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)

$$\texttt{fun}\ x\ \texttt{->}\ e$$

- 형식 인자 (formal parameter)가 x이고 몸통 (body)이 e인 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):

$$\texttt{fun}\ x_1\ \cdots\ x_n\ \texttt{->}\ e$$

  - `fun x y -> x + y` = `fun x -> fun y -> x + y`
  - `fun x y z -> x + y + z` = `fun x -> fun y -> fun 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_2는 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:

$$\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2$$

- e_1의 값을 x라고 하고 e_2를 계산
  - x: 값의 이름 (변수)
  - e_1: 정의식 (binding expression)
  - e_2: 몸통식 (body expression)
- x의 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_1과 e_2는 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
```

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>

# sum_if_true (fun x -> x mod 2 = 0) 3 4;;
- : int = 4
```

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
```

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, "thre
  ```
- ```
  # [[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:

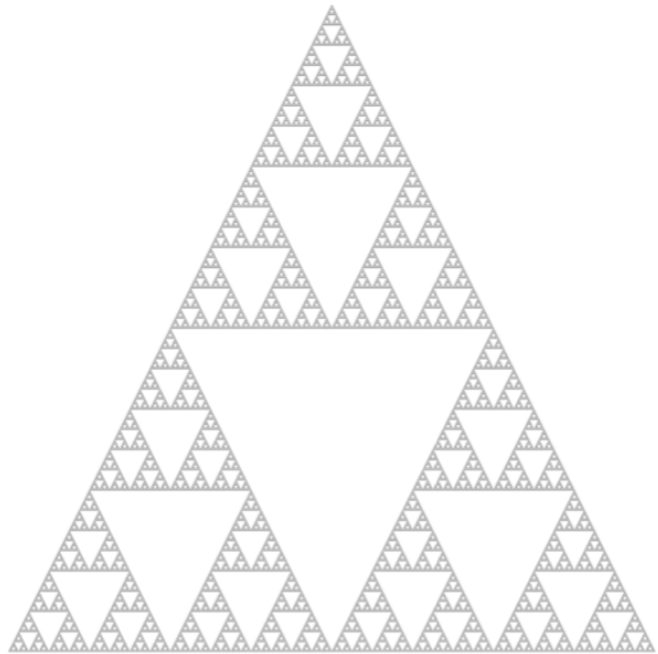
```
# let rec length l =
    match l with
     [] -> 0
    |h::t -> 1 + length t;;
val length : 'a list -> int = <fun>
# length [1;2;3];;
- : int = 3
```

쓰이지 않는 구성요소는 _ 로 대체 가능:

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

재귀적 사고의 간결함

- 함수형 언어에서 반복은 주로 재귀함수로 표현
- Ex) 아래 도형을 그리는 프로그램?



예: 삽입 정렬 (insertion sort)

- 정렬된 리스트 `l`에 원소 `a` 추가하기:

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

예: 삽입 정렬 (insertion sort)

- 정렬된 리스트 l에 원소 a 추가하기:

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

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

예: 삽입 정렬 (insertion sort)

- 정렬된 리스트 l에 원소 a 추가하기:

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

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

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

예: 삽입 정렬 (insertion sort)

- 정렬된 리스트 l에 원소 a 추가하기:

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

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

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

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

cf) C-style 언어와 비교

```
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--;
 }
}
```

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.

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

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.

꼬리 재귀 함수 (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.

고차 함수 (Higher-Order Functions)

- 다른 함수를 인자로 받거나 리턴하는 함수
- 높은 추상화(abstraction) 수준에서 프로그램을 작성하게 함
- 간결하고 재활용 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
```

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-oder 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
```

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)

요약

함수형 프로그래밍 언어의 특징

- 함수를 특별하게 다루지 않는다 $\Rightarrow$ 표현력 증가
- 한번 정의된 값은 변하지 않는다 $\Rightarrow$ 복잡도 감소
- 안전한 정적 타입 시스템 $\Rightarrow$ 안정성 증가
- 자동 타입 추론, 데이터 타입, 모듈 함수, ...

감사합니다!