고급 OCaml 프로그래밍
Part 1

Jan 11-14, 2019 @ Tezos Blockchain Camp

이우석
한양대학교 소프트웨어학부

소개

- 소속: 한양대학교 ERICA 캠퍼스 소프트웨어학부
- 전공: 프로그래밍 언어, 소프트웨어 분석, 프로그램 자동 합성
- 웹페이지: http://ropas.snu.ac.kr/~wslee

강의 내용

- 고급 OCaml 프로그래밍 (5시간)
  - 모듈 프로그래밍
  - 에러 처리
  - OCaml 표준 라이브러리
  - 여러가지 자료구조
  - 언어 해석기 (interpreter) 구현

OcaIDE

- OCaml 통합 개발 환경 IDE
- Eclipse plugin
- 설치 방법:
  - http://www.algo-prog.info/ocaide/install.php

모듈 프로그래밍 1
(Modular Programming 1)

Cornell CS 3110 Fall 2017 Data Structures and Functional Programming
http://www.cs.cornell.edu/courses/cs3110/2017fa/index.php의 강의자료를 참고하여 작성되었음

프로그램 크기

OCaml: 20만줄

Unreal engine 3: 200만줄

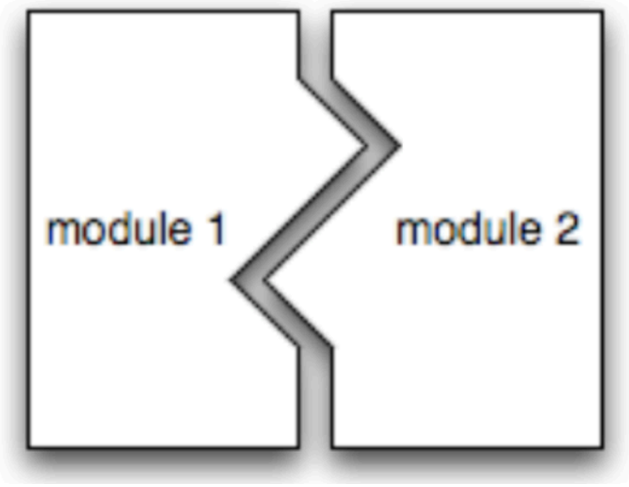
Windows Vista: 5천만줄†

- 한 개인에 의해 개발 불가
- 어떤 프로그래머도 모든 부분에 대한 디테일 이해 불가
- 현재까지 배운 OCaml 지식으로 작성하기에 실제 프로그램은 너무 복잡

†http://www.informationisbeautiful.net/visualizations/million-lines-of- code/

모듈화 (Modularity)

- 프로그램을 서로 독립적인 여러개의 부품으로 구성하는 것
- 각 모듈은 따로 독립적으로 개발됨
- 개개 모듈의 행동을 이해하기 위해서 다른 모듈에 대한 이해가 필요 없도록 (지역적 추론 O, 전체적 추론 X)
- 속내용 감추기 (Abstraction)



모듈화를 위한 지원들

- 이름공간 (Namespaces): 동일한 이름들로 충돌이 없도록
- 속내용 감추기 (Abstraction): 모듈 사용자가 구현 디테일을 알고 오용/악용하여 코드의 올바름이 깨지지 않도록
- 코드 재사용 (Code reuse): 반복되는 코드 부분이 없도록

Java vs. OCaml 모듈화

Java	OCaml	목표
class, package	Structures	함수, 변수 등의 이름을 이름공간(namespace)에 조직
Interface	Signatures	부품의 필요 조건 명세
public, protected, private	Abstract types	이름공간 밖에서 접근가능한 것들이 무엇인지 설정
subtyping, inheritance	functors, includes	코드 재사용

불변형 자료구조 (immutable datastructure)

- 함수형 (functional) 혹은 영속적 (persistent) 자료구조라고도 불림
- 자료구조를 변형시키지 않음. 업데이트 -> 새로운 버전 및 구 버전 함께 존재
- 쓰레기 수집기: 메모리에서 재사용이 최대한 가능하도록
- 예: 스택 (stacks), 큐 (queues), 사전식 자료구조 (dictionaries)

MyStack 모듈

```
module MyStack = struct
  type 'a stack =
    | Empty
    | Entry of 'a * 'a stack

  let empty = Empty
  let is_empty s = s = Empty
  let push x s = Entry (x, s)
  let peek = function
    | Empty -> failwith "Empty"
    | Entry(x,_) -> x
  let pop = function
    | Empty -> failwith "Empty"
    | Entry(_,s) -> s
end
```

(1,2) = 1 과 2를 직접적으로 나타내어 튜플을 만드는거고
(type * type) = 튜플이지만 타입을명시하는 것

function 이란 익명함수인데 패턴매칭 지원 하는 함수

ListStack 모듈

```
module ListStack = struct
  let empty = []
  let is_empty s = s = []
  let push x s = x :: s
  let peek = function
    | []    -> failwith "Empty"
    | x::_  -> x
  let pop = function
    | []    -> failwith "Empty"
    | _::xs -> xs
end
```

Java와 OCaml 의 구문적 차이

- Java: `s = new Stack(); s.push(1); s.pop();`
  - 스택은 dot(.) 왼쪽, 메소드는 오른쪽
- OCaml:

```
let s = MyStack.empty in
let s' = Mystack.push 1 s in
MyStack.peek s'
```

  - 스택은 항상 함수의 인자로 사용됨

질문

- 다음 코드에서

```
let s = ListStack.push 1 ListStack.empty in
let t = ListStack.pop s in
s, t
```

결과 값은?

1. [], []
2. [], 1
3. [1], [] 정답
4. [1], 1
5. 정답 없음

질문

- 다음 코드에서

```
let s = ListStack.push 1 ListStack.empty in
let t = ListStack.pop s in
s, t
```

결과 값은?

1. [], []
2. [], 1
3. **[1], []**
4. [1], 1
5. 정답 없음

모듈의 구문

- `module ModuleName = struct`
  `definitions`
  `end`

  메인 함수역할은 마지막꺼

- ModuleName 은 항상 대문자로 시작

- *`definitions`* 은 `let, type, exception,` 혹은 다른 `module` 을 포함할 수 있음 (nested module)

- 모듈은 한 이름공간을 만들어냄:

```
module M = struct let x = 42 end
let y = M.x
```

- *`definitions`* 안에서 정의된 것들은 순서대로 실행됨

속내용 감추기

- 스택들이 각기 자신만의 고유한 데이터 표현방식을 사용할 수 있음
  - `MyStack` 은 `'a stack, ListStack`은 `'a list`
- 스택의 구체적 구현과 관계없이 스택이 갖춰야할 최소조건만 명세하고 싶다면?

스택 모듈을 구현한 최대한의 자유도를 주기 위해서 최소 조건만 명세하는 것이다.

```
module type StackSig = sig
  type 'a stack
  val is_empty : 'a stack -> bool
  val push :'a->'a stack -> 'a stack
  val peek : 'a stack -> 'a
  val pop : 'a stack -> 'a stack
end
```

추상 타입 (Abstract types)

- `'a stack` 은 추상 타입: 타입의 존재만 선언하고, 구체적인 타입 정의는 생략

- 모든 `Stack` 타입 모듈은 추상 타입을 정의해야함.

```
module MyStack : Stack = struct
type 'a stack = Empty | Entry of 'a * 'a stack
...

module ListStack : Stack = struct
type 'a stack = 'a list
...
module FastListStack : Stack = struct
type 'a stack = 'a fastlist
...
```

강제 조건을 만드는 법

속내용 감추기의 이점

- 모듈을 사용하는 다른 부분은 그 모듈의 구조를 안다는 가정하에 작성되어서는 안됨
  - 예: ListStack 이 리스트로 구현되어 있다는 것을 알고 외부에서 ListStack.push x s 대신 x::s 를 사용했다고 가정.

    이 후에 ListStack을 더 빠른 다른 가상의 자료구조를 사용하도록 변경하는 경우, x::s 를 사용하는 부분들이 더 이상 작동되지 않음

- 모듈 외부에서는 해당 모듈의 표현형(representation type)을 모르도록 작성해야.

모듈 타입 구문

- `module type SignatureName = sig`
  `type specifications`
  `end`

  - `SignatureName` 은 대문자로 시작할 필요는 없지만 보통 그렇게 씀

  - *`type specifications`* 은 `val, type, exception, module type` 을 포함할 수 있음

모듈 정의에 모듈 타입 강제하기

- module ModuleName : t = struct
  definitions
  end

- module ModuleName = (struct
  definitions
  end : t)

  - 타입 t는 모듈 타입

모듈 타입 의미

```
module Mod : Sig = struct … end
```

OCaml 타입 검사기가 다음을 확인

- Mod 정의 시 (요구조건 만족여부): Sig 안에 정의되어 있는 것들이 모두 Mod 에 정의되어 있음
- Mod 사용 시 (속내용 감추기): Sig 안에 정의되어 있는 것 외에 다른 것은 Mod 밖에서 접근될 수 없음

요구조건 만족 여부

```
module type S1 = sig
   val x:int
   val y:int
end
module M1 : S1 = struct
 let x = 42 end

(* type error:
   Signature mismatch:
   The value `y' is required but not provided
*)
```

둘 다 있어야 되는데 하나만 있어서 뭐라고한다,

속내용 감추기

```
module type S2 = sig
  val x:int
end
module M2 : S2 = struct
  let x = 42
  let y = 7
end
M2.y
(* type error: Unbound value M2.y *)
```

이러한 변수 y는 자바의 프라이빗 변수 로컬로만 사용 가능

질문

- 다음 중 타입 검사기를 통과하는 코드는?

A.
```
module M =
  (struct let inc x = x+1 end : sig end)
```

B.
```
module M =
  (struct let inc x = x+1 end : sig val inc end)
```

C.
```
module M =
  (struct let inc x = x+1 end
  : sig val inc : int -> int end)
```

D. 전부

질문

- 다음 중 타입 검사기를 통과하는 코드는?

A.
```
module M =
  (struct let inc x = x+1 end : sig end)
```

B.
```
module M =
  (struct let inc x = x+1 end : sig val inc end)
```

타입 명세 필요

C.
```
module M =
  (struct let inc x = x+1 end
  : sig val inc : int -> int end)
```

D. 전부

기타: 모듈 열기

자바의 import 와 같은 부분

- open 연산자를 쓰면 dot (.) 연산자를 쓸 필요 없이 모듈 구성원을 접근 가능

```
# module M = struct let x = 42 end;;
module M : sig val x : int end

# M.x;;
- : int = 42
```

```
# x;;
Error: Unbound value x

# open M;;

# x;;
- : int = 42
```

- 만약 동일한 이름을 정의하는 두 모듈을 open 할 경우, 나중에 open 된 것이 앞의 것을 덮어씀

```
module M = struct let x = 42 end
module N = struct let x = "bigred" end
open M
open N
(* what is [x]?  an [int] or a [string]? *)
```

기타: 모듈 열기

- 그러므로 범위(scope)를 정하여 사용하는 것이 권장됨:

```
(* without [open] *)
let f x =
  let y = List.filter ((>) 0) x in
      (* many more lines of code that use [List.] a lot *)
  ...

(* with [open] *)
let f x =
  let open List in (* [filter] is now bound to [List.filter] *)
  let y = filter ((>) 0) x in
  ...  (* many more lines of code that now can omit [List.] *)
```

- ModuleName.(…) 과 같이도 사용 가능

예: 산술연산 모듈

```
module type Arith = sig
  type t
  val zero  : t
  val one   : t
  val (+)   : t -> t -> t
  val ( * ) : t -> t -> t
  val (~-)  : t -> t
end
```

```
module Ints : Arith = struct
  type t     = int
  let zero   = 0
  let one    = 1
  let (+)    = Pervasives.(+)
  let ( * )  = Pervasives.( * )
  let (~-)   = Pervasives.(~-)
end
```

standard librarary

- 속내용 감추기

(* 는 주석이라 띄어쓰기해야된다

```
# Ints.(one + one);;
- : Ints.t = <abstr>
```

속을 못보게 하려고 이렇게 출력된다 ..

예: 산술연산 모듈

- 바깥세상에서 Ints의 내용물 파악을 위해선?

```
module type Arith = sig
  (* everything else as before, and... *)
  val to_string : t -> string
end
```

```
module Ints : Arith = struct
  (* everything else as before, and... *)
  let to_string = string_of_int
end
```

```
# Ints.(to_string (one + one));;
- : string = "2"
```

예: 산술연산 모듈

- 바깥세상에서 Ints의 t 가 int 임을 알게하고 싶은 경우

```
module Ints = struct
  type t    = int
  let zero  = 0
  let one   = 1
  let (+)   = Pervasives.(+)
  let ( * ) = Pervasives.( * )
  let (~-)  = Pervasives.(~-)
end

module IntsAbstracted : Arith = Ints
(* IntsAbstracted.(1 + 1) is illegal *)

module IntsExposed : (Arith with type t = int) = Ints
(* IntsExposed.(1 + 1) is legal *)
```

시스템

실습 1: 두 개의 큐

```
module ListQueue = struct
  type 'a queue = 'a list

  let empty = []

  let is_empty q = (q = [])

  let enqueue x q = q @ [x]

  let peek = function
    | []   -> failwith "Empty"
    | x::_ -> x

  let dequeue = function
    | []   -> failwith "Empty"
    | _::q -> q
end
```

실습 1: 두 개의 큐

```
module TwoListQueue = struct
  type 'a queue = {front:'a list; back:'a list}

  let empty = {front=[]; back=[]}

  let is_empty = function
    | {front=[]; back=[]} -> true
    | _ -> false

  let norm = function
    | {front=[]; back} -> {front=List.rev back; back=[]}
    | q -> q

  let enqueue x q = {q with back=x::q.back}

  let peek = function
    | {front=[]; _} -> None
    | {front=x::_; _} -> Some x

  let dequeue = function
    | {front=[]; _} -> None
    | {front=_::xs; back} -> Some (norm {front=xs; back})
end
```

실습 1: 두 개의 큐

- 다음 두 함수를 이용하여 두 타입의 큐에 2만개의 원소를 삽입할 때 걸리는 시간을 측정하세요

```
(* Creates a ListQueue filled with [n] elements. *)
let fill_listqueue n =
  let rec loop n q =
    if n=0 then q
    else loop (n-1) (ListQueue.enqueue n q) in
  loop n ListQueue.empty

let fill_twolistqueue n =
  let rec loop n q =
    if n=0 then q
    else loop (n-1) (TwoListQueue.enqueue n q) in
  loop n TwoListQueue.empty
```

- 다음은 시간 측정 후 출력 방법

```
let start = Sys.time () in
(* do something *)
let elapsed_time = Sys.time () -. Start in
Printf.printf “%d seconds\n” elapsed_time
```

실습 1: 두 개의 큐

코드를 적절히 수정하여 두 타입의 큐 모두 fill_queue 함수를 이용하여 원소를 삽입할 수 있도록

- 모듈 타입 Queue 를 정의하고 사용하여 두 모듈 정의를 수정하고
- 아래 fill_queue 함수를 정의하세요.

```
(* Creates a Queue filled with [n] elements. *)
let fill_queue n (enqueue, empty_q) = …
```

실습 2: 사전

- 사전 (dictionary): 키에서 값으로의 매핑

```
module type Dictionary = sig
  type ('k, 'v) t

  (* The empty dictionary *)
  val empty  : ('k, 'v) t

  (* [insert k v d] produces a new dictionary [d'] with the same mappings
   * as [d] and also a mapping from [k] to [v], even if [k] was already
   * mapped in [d]. *)
  val insert : 'k -> 'v -> ('k,'v) t -> ('k,'v) t

  (* [lookup k d] returns the value associated with [k] in [d].
   * raises:  [Not_found] if [k] is not mapped to any value in [d]. *)
  val lookup  : 'k -> ('k,'v) t -> 'v
end
```

- 모듈 타입 Dictionary 를 따르는 AssocListDict 를 정의하세요. lookup 함수는 List.assoc 을 이용하여 정의할 수 있습니다.
(참조: https://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html)

```
module AssocListDict : Dictionary = struct …
```

실습 3: 유리수

- 유리수 (분자 / 분모로 표현) 를 위한 모듈을 작성하세요. 단, 항상 기약분수 (분자 분모가 서로소) 형태를 띄고 있어야 합니다.

```
module type Fraction = sig
  (* A fraction is a rational number p/q, where q != 0.*)
  type t

  (* [make n d] is n/d. Requires d != 0. *)
  val make : int -> int -> t

  val numerator   : t -> int
  val denominator : t -> int
  val toString    : t -> string
  val toReal      : t -> float

  val add : t -> t -> t
  val mul : t -> t -> t
end
```

실습 3: 유리수

- 다음 유클리드 호제법 함수를 이용하세요 (gcd x y 는 x 와 y의 최대 공약수를 반환).

```
(* [gcd x y] is the greatest common divisor of [x] and [y].
 * requires: [x] and [y] are positive.
 *)
let rec gcd (x:int) (y:int) : int =
  if x = 0 then y
  else if (x < y) then gcd (y - x) x
  else gcd y (x - y)
```

모듈 프로그래밍 2
(Modular Programming 2)

Cornell CS 3110 Fall 2017 Data Structures and Functional Programming
http://www.cs.cornell.edu/courses/cs3110/2017fa/index.php의 강의자료를 참고하여 작성되었음

Java vs. OCaml 모듈화

Java	OCaml	목표
class, package	Structures	함수, 변수 등의 이름을 이름공간(namespace)에 조직
Interface	Signatures	부품의 필요 조건 명세
public, protected, private	Abstract types	이름공간 밖에서 접근가능한 것들이 무엇인지 설정
subtyping, inheritance	functors, includes	코드 재사용

집합의 구현

```
module type Set = sig
  type 'a t

  (* [empty] is the empty set *)
  val empty : 'a t

  (* [mem x s] holds iff [x] is an element of [s] *)
  val mem   : 'a -> 'a t -> bool

  (* [add x s] is the set [s] unioned with the set containing exactly [x] *)
  val add   : 'a -> 'a t -> 'a t

  (* [elts s] is a list containing the elements of [s].  No guarantee
   * is made about the ordering of that list. *)
  val elts  : 'a t -> 'a list
end
```

집합의 구현

```
module ListSetNoDups : Set = struct
  type 'a t   = 'a list
  let empty   = []
  let mem     = List.mem
  let add x s = if mem x s then s else x::s
  let elts s  = s
end

module ListSetDups : Set = struct
  type 'a t   = 'a list
  let empty   = []
  let mem     = List.mem
  let add x s = x::s
  let elts s  = List.sort_uniq Pervasives.compare s
end
```

집합의 구현

- Include 를 이용하여 코드 재사용

ListSetNoDups 상속

```
module ListSetDupsExtended : Set = struct
  include ListSetDups
  let of_list lst = List.fold_right add lst empty
end
```

새 함수 추가

  - `let of_list lst = lst` 로 정의하지 않는 이유:

    `ListSetDups` 는 모듈 타입 Set 으로 속내용이 감춰졌기 때문에 타입 `'a t` 가 추상타입임. `'a t = 'a list` 임을 `ListSetDupsExtended` 에서도 알 수 없음.

집합의 구현

- 아래의 ListSetDupsImpl 은 속내용이 감춰지지 않음

```
module ListSetDupsImpl = struct
  type 'a t   = 'a list
  let empty   = []
  let mem     = List.mem
  let add x s = x::s
  let elts s  = List.sort_uniq Pervasives.compare s
end

module ListSetDupsExtended = struct
  include ListSetDupsImpl
  let of_list lst = lst
end
```

집합의 구현

- 모듈 타입도 include 사용 가능

```
module type SetExtended = sig
  include Set
  val of_list : 'a list -> 'a t
end


module ListSetDupsExtended : SetExtended = struct
  include ListSetDupsImpl
  let of_list lst = lst
end
```

Include vs. Open

- Open: 모듈 내부에서의 소비만을 위해 외부 정의를 가져옴
  Include: 외부 정의를 가져와서 사용하고 복제본을 다시 외부에 제공

```
module M = struct
  let x = 0
end
```

```
module N = struct
  include M
  let y = x + 1
end

module O = struct
  open M
  let y = x + 1
end
```

```
module M : sig val x : int end
module N : sig val x : int val y : int end
module O : sig val y : int end
```

Include로 불충분

- 리스트와 집합을 받아서 리스트의 원소들을 모두 집합에 넣는 함수 (문제: 집합 구현에 따라 두 가지 버전 필요)

```
let rec add_all lst set = match lst with
    | [] -> set
    | h::t -> add_all t (ListSetNoDups.add h set)
```

```
let rec add_all lst set = match lst with
    | [] -> set
    | h::t -> add_all t (ListSetDups.add h set)
```

- 사용할 add 함수를 인자로 제공 (문제: 매번 함수 제공. 모듈 밖 정의):

```
let rec add_all' add lst set = match lst with
  | [] -> set
  | h::t -> add_all' add t (add h set)

let add_all_dups lst set = add_all' ListSetDups.add lst set
let add_all_nodups lst set = add_all' ListSetNoDups.add lst set
```

Include로 불충분

- 각 집합 구현 안에 포함되도록:
  (문제: 같은 코드 반복)

```
module AddAll = struct
  let rec add_all' add lst set = match lst with
    | [] -> set
    | h::t -> add_all' add t (add h set)
end

module ListSetNoDupsExtended : SetExtended = struct
  include ListSetNoDups
  include AddAll
  let add_all lst set = add_all' add lst set
end

module ListSetDupsExtended : SetExtended = struct
  include ListSetDups
  include AddAll
  let add_all lst set = add_all' add lst set
end
```

모듈 함수 (Functors)

- 모듈 함수 : 모듈 -> 모듈 타입의 함수
- 하지만 보통의 함수들과 다른 취급: 모듈은 1차원 (first-class) 객체가 아니므로 모듈 함수 또한 1차원 객체가 아님†
- 1차원 객체 (first-class values)
  - 변수에 저장 가능
  - 함수에 인자로 제공 가능
  - 다른 함수로 부터 반환가능

†최신 OCaml 버전에서는 first-class module 이 지원되나 본 강의에서는 다루지 않음.

모듈 함수 구문

- 함수와 유사

```
module F (M : S) = struct
  ...
end

module F = functor (M : S) -> struct
  ...
end
```

- 인자 여러 개 받을 때

```
module F (M1 : S1) ... (Mn : Sn) = struct
  ...
end
```

- 출력 모듈 타입 명세를 원할 때

```
module F (M : Si) : So = struct
  ...
end
```

```
module F (M : Si) = (struct
  ...
end : So)
```

모듈 함수 구문

```
module type X = sig
  val x : int
end
```

```
module IncX (M: X) = struct
  let x = M.x + 1
end
```

```
# module A = struct let x = 0 end
# A.x
- : int = 0
```

```
# module B = IncX(A)
# B.x
- : int = 1

# module C = IncX(B)
# C.x
- : int = 2
```

Include로 불충분 → Functor 로 충분

```
module ExtendSet(S:Set) = struct
  include S

  let add_all lst set =
    let add' s x = S.add x s in
    List.fold_left add' set lst
end
```

```
module ListSetNoDupsExtended = ExtendSet(ListSetNoDups);;
module ListSetDupsExtended = ExtendSet(ListSetDups);;
```

- 각 모듈에 add_all 함수 포함. 중복 코드 없음.

모듈 함수의 필요성

- **부품 갈아치우기**: 시스템의 부품들을 교환가능하게 만듦.
- **새로운 기능 추가 용이**: 모듈 함수는 기존에 존재하는 모듈에 새로운 기능을 추가하는 표준화된 방법을 제공함. 이로 인해 모듈 기능 확장을 많은 부분 자동화할 수 있음
- **모듈의 여러 인스턴스가 필요할 때**: 모듈에 변환가능한 값들이 포함될 때 (reference, record 등) 각각 서로다른 mutable state를 갖는 여러 인스턴스를 만들 경우 사용

모듈 함수의 사용 예: 표준 라이브러리 Map

- Map: 균형잡힌 이진 트리 (balanced binary tree)를 이용한 사전 (dictionary) 자료구조 구현체
- Map 모듈은 Make 라는 모듈 함수 제공: 특정 타입 키를 갖는 사전 모듈 반환

```
(* maps over totally ordered keys *)
module Map : sig
  (* the input type of Make *)
module type OrderedType = sig type t ... end
  (* the output type of Make *)
module type S = sig type key ... end
  (* functor that makes a module *)
module Make (Ord : OrderedType) : S with type key = Ord.t
end
```

모듈 타입 S

```
module type S = sig
  type key
  type 'a t
  val empty : 'a t
  val mem : key -> 'a t -> bool
  val add : key -> 'a -> 'a t -> 'a t
  ...
end
```

모듈 타입 OrderedType

```
module type OrderedType = sig
  type t
  val compare : t -> t -> int
end
```

두 인자가 같으면 0,
앞의 것이 작으면 음수,
뒤의 것이 크면 양수 반환

Map

```
(* maps over totally ordered keys *)
module Map : sig

  (* the input type of Make *)
module type OrderedType = sig type t ... end
  (* the output type of Make *)

module type S = sig type key ... end

  (* functor that makes a module *)
module Make (Ord : OrderedType) : S with type key =
Ord.t

end
```

밖에서 Make의 반환값인 모듈의 key 타입과
Ord.t 가 같음을 알게끔. 아니면 Map 사용 불가.
(sharing constraint)

Map

```
module type S with type key = Ord.t =
sig
  type key = Ord.t
  type 'a t
  val empty : 'a t
  val mem : key -> 'a t -> bool
  val add : key -> 'a -> 'a t -> 'a t ...
end
```

Map 사용

```
# module StringMap = Map.Make(String) ;;
module StringMap : sig
type key = string
...
end

# let sm = StringMap.(
    empty |> add "Alice" 4.0 |> add "Bob" 3.7)

# StringMap.find "Bob" sm
- : float = 3.7
```

String 모듈에 이미 compare 함수가 정의되어 있음

```
let (|>)  x y = y x
```

Map 사용

- 순서가 고유하게 정의되는 타입을 키로 하고 싶을 때는?

```
type name = {first:string; last:string}

module Name = struct
  type t = name
  let compare {first=first1;last=last1}
  {first=first2;last=last2} =
  match Pervasives.compare last1 last2 with
  | 0 -> Pervasives.compare first1 first2
  | c -> c
  end

module NameMap = Map.Make(Name)
```

Map 사용

- 키가 정수인 맵

```
module Int = struct
  type t = int
  let compare = Pervasives.compare
end

module IntMap = Map.Make(Int)
let im = IntMap.(
   empty |> add 1 "one" |> add 2 “two") in
IntMap.find 3 im ;;

Exception: Not_found.
```

코드 재사용 사례: 대수 구조

- algebra.ml: 두 개의 모듈 타입과 네 개의 모듈 포함
- Ring: 대수 구조 Ring 정의. 덧셈 곱셈 연산자 포함
- Field: Ring + 나눗셈 연산
- IntRing, FloatRing, IntField, FloatField, IntRational, FloatRational
- 92줄 -> 68줄
- 강의자료 algebra.ml 과 algebra_refactor.ml 참조

코드 재사용 사례: 인터벌 연산

- 인터벌: 구간을 표현하는 객체. 다양한 타입 가능

  예: [1, 4] - 1이상 4이하 정수
  [2.2, 4.8] - 2.2 이상 4.8 이하 실수
  [10/31, 11/27] - 10월 31일부터 11월 27일까지
  …

- 가능한 연산들: 특정 구간의 특정 원소 포함 여부, 두 구간의 공통부분(intersection) 및 합한부분(union), …

- 강의자료 interval.ml 과 interval_refactor.ml 참조

시스템

실습 1: 날짜 Map

- Map.Make 모듈함수와 Date 모듈을 이용해서 DateMap 모듈을 만들고, 달력을 만드세요.

```
type date = { month:int; day:int }

module Date = struct
  type t = date
  let compare ...
end

type calendar = string DateMap.t
```

값의 타입이 문자열. 즉, 달력은 날짜에서 문자열로 가는 맵핑

- 다음 정보를 달력에 추가하고,
  - 1/7: “tezos camp starts”, 1/15: “tezos camp ends”
- DateMap.iter 를 이용, DateMap 객체 내용물을 출력하세요.

실습 2: first after

- 함수 first_after : calendar -> Date.t -> string 를 작성하세요. 함수는 주어진 날짜를 받아, 그 날짜 이후에 발생하는 첫번째 이벤트를 반환합니다.

(힌트: DateMap.fold 를 사용)
https://caml.inria.fr/pub/docs/manual-ocaml/libref/Map.S.html

실습 3: 대/소문자 둔감 집합

- 표준 라이브러리의 집합 구현체 Set 은 Map과 매우 비슷합니다. https://caml.inria.fr/pub/docs/manual-ocaml/libref/Set.html

  표준라이브러리 Set을 사용하여, 대/소문자 구분 없는 단어 집합을 구현하세요. 예를들어

  {“aBc”, “Ab”}, {“Abc”, “ab”}

  는 동일한 집합이고, 두 집합에 “aB” 를 추가해도 변함이 없어야 합니다. 구현한 모듈이 CisSet 이라는 가정하에 다음의 결과를 출력해야 합니다.

```
# CisSet.(equal (of_list ["grr"; "argh"]) (of_list ["GRR"; "aRgh"]))
- : bool = true
```

에러 처리
(Error Handling)

리턴 값에 에러 포함하기 (Error-Aware Return)

- 에러를 리턴값에 포함 (리스트에서 인덱스번째 원소 반환)

```
# List.nth_opt;;
- : 'a list -> int -> 'a option = <fun>
```

- Option 리턴 타입이 특정 원소를 찾는데 실패할 가능성이 있다는 것 암시

```
# List.nth_opt [1;2;3] 0 ;;
- : int option = Some 1
```

```
# List.nth_opt [] 2 ;;
- : 'a option = None
```

- 에러를 리턴값에 포함하는 방식은 함수 호출부에서 명시적으로 에러 처리를 하도록 강제함

리턴 값에 에러 포함하기 (Error-Aware Return)

- 비교함수와 리스트를 받아서 가장 작은 원소와 가장 큰 원소의 쌍 반환

```
let compute_bounds cmp list =
    let sorted = List.sort cmp list in
   let last_ind = (List.length list) - 1 in
    match (List.nth_opt sorted 0), (List.nth_opt sorted last_ind) with
    | None,_ | _, None -> None
    | Some x, Some y -> Some (x,y)
```

리턴 값에 에러 포함하기 (Error-Aware Return)

- 두 해쉬테이블을 받아서 동일한 키에 대해 각기 다른 값이 두 해쉬테이블에 바인딩 된 경우들의 키를 반환
  https://caml.inria.fr/pub/docs/manual-ocaml/libref/Hashtbl.html

```
let find_mismatches table1 table2 =
      Hashtbl.fold (fun key data mismatches ->
        match Hashtbl.find table2 key with
        | Some data' when data' <> data -> key :: mismatches
        | _ -> mismatches
    ) table1 []
```

- 위의 경우: 리스트에서 원소를 못찾는게 오류. 아래의 경우: 어떤 키가 해쉬테이블에 없는게 오류가 아님.

- 함수를 사용하는 부분에서 알아서 처리하게끔 하는 것이 합리적

리턴 값에 에러 포함하기 (Error-Aware Return)

- Option 타입이 에러를 표현하는데 충분치 않을 수 있음 (None 은 에러 발생 여부만 알려줄 뿐 상세 정보는 전달하지 않으므로)
- 표준 라이브러리에서 다음의 타입 제공

```
type ('a, 'b) result =
 Ok of ‘a  |. Error of 'b
```

```
# [ Ok 3; Error "failure"; Ok 4 ];;
- : (int, string) result list = [Ok 3; Error "failure"; Ok 4]
```

예외 (Exceptions)

- Java, Python 등 여타 언어의 예외와 동일. 오류메세지와 함께 종료

```
# List.map (fun x -> Printf.printf "%d\n%!" x; 100 / x) [1;3;0;4];;
1
3
0
Exception: Division_by_zero.
```

- 예외처리기 (Exception handlers) 사용 가능

- 사용자 정의 예외

```
# exception Key_not_found of string;;
exception Key_not_found of string
# raise (Key_not_found "a");;
Exception: Key_not_found("a").
```

예외 (Exceptions)

- 예외는 보통의 값과 같이 처리

```
# let exceptions = [ Not_found; Division_by_zero; Key_not_found
"b" ];;
val exceptions : exn list = [Not_found; Division_by_zero;
Key_not_found("b")]
```

```
# List.filter (function
    Key_not_found _ | Not_found -> true
    | _ -> false) exceptions;;
- : exn list = [Not_found; Key_not_found("b")]
```

- 앞선 예외 정의는 아래 구문의 설탕 구문 (syntactic sugar)
  https://caml.inria.fr/pub/docs/manual-ocaml/extn.html#sec266

```
# type exn += Key_not_found of string ;;
```

예외 (Exceptions)

- 예외를 이용한 에러 발생

```
# let rec find_exn alist key = match alist with
    | [] -> raise (Key_not_found key)
    | (key',data) :: tl -> if key = key' then data else find_exn tl key
  ;;
val find_exn : (string * 'a) list -> string -> 'a = <fun>
# let alist = [("a",1); ("b",2)];;
val alist : (string * int) list = [("a", 1); ("b", 2)]
# find_exn alist "a";;
- : int = 1
# find_exn alist "c";;
Exception: Key_not_found("c").
```

- 예외를 발생하는 raise (함수)의 타입: ('a 절대 반환안되는 값의 타입을 표현)

```
# raise;;
- : exn -> 'a = <fun>
```

```
# let rec forever () = forever ();;
val forever : unit -> 'a = <fun>
```

예외를 발생시키는 다른 방법들

- assert (조건식)

```
# let merge_lists xs ys ~f =
    if List.length xs <> List.length ys then None
    else
      let rec loop xs ys =
        match xs,ys with
        | [],[] -> []
        | x::xs, y::ys -> f x y :: loop xs ys
        | _ -> assert false
      in
      Some (loop xs ys)
  ;;
val merge_lists : 'a list -> 'b list -> f:('a -> 'b -> 'c) -> 'c list
option =
  <fun>
# merge_lists [1;2;3] [-1;1;2] ~f:(+);;
- : int list option = Some [0; 3; 5]
# merge_lists [1;2;3] [-1;1] ~f:(+);;
- : int list option = None
```

예외를 발생시키는 다른 방법들

- assert (조건식)

```
# let merge_lists xs ys ~f =
    let rec loop xs ys =
      match xs,ys with
      | [],[] -> []
      | x::xs, y::ys -> f x y :: loop xs ys
      | _ -> assert false
    in
    loop xs ys
  ;;
val merge_lists : 'a list -> 'b list -> f:('a -> 'b -> 'c) -> 'c list =
<fun>
# merge_lists [1;2;3] [-1] ~f:(+);;
Exception: (Assert_failure //toplevel// 5 13).
```

- 기타:

```
# let failwith msg = raise (Failure msg);;
val failwith : string -> 'a = <fun>
```

예외 처리기 (Exception Handler)

- 예외처리 문법

```
try <expr> with
| <pat1> -> <expr1>
| <pat2> -> <expr2>
...
```

- 특정 예외 잡기

```
# let lookup_weight ~compute_weight alist key =
    try
      let data = find_exn alist key in
      compute_weight data
    with
      Not_found -> 0. ;;

# lookup_weight ~compute_weight:(fun _ -> raise Not_found)
    ["a",3; "b",4] "a" ;;
- : float = 0.
```

예외 발생 경위 (Backtrace)

```
open Printf
exception Empty_list

let list_max = function
  | [] -> raise Empty_list
  | hd :: tl -> List.fold_left max hd tl
```

```
let () =
  printf "%d\n" (list_max [1;2;3]);
  printf "%d\n" (list_max [])
```

```
$ ocamlbuild exn_debug.d.byte
$ OCAMLRUNPARAM=b ./exn_debug.d.byte
3
Fatal error: exception Exn_debug.Empty_list
Raised at file "exn_debug.ml", line 5, characters 16-26
Called from file "exn_debug.ml", line 10, characters 16-29
```

좋은 에러 처리 방안 선택

- 예외 발생 및 처리
  - 장점: 넓은 범위에서 에러처리 용이. 타입정의를 더 추가할 필요 없음
  - 예외처리에서 실수가 발생할 수 있음 (with _ → …)
- 리턴 값에 에러 포함
  - 장점: 에러처리를 더 명시적으로 하게끔 강제 (예: Option 타입으로 값이 반환될 경우, 그 값을 받아서 처리할 때 마다 에러처리를 하게됨)
  - 단점: 코드가 장황해 질 수 있음
- 빠른 프로토타입 제작을 원하거나, 예외적 상황이 드물다면 → 예외 발생/처리
  올바르게 작동하는 것이 중요한 프로그램 제작을 원하면 → 리턴 값에 에러 포함

참고문헌

- Real World OCaml

  https://v1.realworldocaml.org/v1/en/html/error-handling.html