OCaml 둘러보기

Tour of OCaml

이재호 jhlee@ropas.snu.ac.kr

2024년 3월 8일

ROPAS@SNU

목차

들어가기 전에

설문 조사

나는…

- OCaml 설치를 하였다.

설문 조사

나는…

- OCaml 설치를 하였다.
- OCaml을 써본 적이 있다.

설문 조사

나는…

- OCaml 설치를 하였다.
- OCaml을 써본 적이 있다.
- OCaml은 안 써봤지만 다른 함수형 언어를 써본 적이 있다.
  - Haskell, Scala, F#, Clojure, Scheme, Racket 등

설문 조사

나는…

- OCaml 설치를 하였다.
- OCaml을 써본 적이 있다.
- OCaml은 안 써봤지만 다른 함수형 언어를 써본 적이 있다.
  - Haskell, Scala, F#, Clojure, Scheme, Racket 등
- 함수형 프로그래밍에 대해 알고 있다.

OCaml 설치하기

1. OPAM *OCaml Package Manager* 설치하기
2. OCaml 및 관련 도구 설치하기
   - OCaml 4.14 이상
   - UTop, Dune OCamlFormat, ocaml-lsp-server 등
3. 프로그래밍 환경 설정하기
   - (Neo)vim, Emacs, Visual Studio Code 등

OPAM 설치하기 (macOS)

1. macOS 유저는 Homebrew 패키지 관리자로 설치하는 것을 추천합니다.

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/
↪  Homebrew/install/HEAD/install.sh)"
```

   Homebrew 설치 이후 나오는 메시지를 보고 `PATH`에 `brew`를 추가합니다.

2. Homebrew로 OPAM을 설치합니다.

```
brew install opam
```

OPAM 설치하기 (Linux)

예시로 Ubuntu 18.04 이상을 기준으로 설명합니다. 이외는 OPAM 설치 문서를 참고하세요.

1. ppa를 추가하여 설치합니다.

```
add-apt-repository ppa:avsm/ppa
apt update
apt install opam
```

OPAM 설치하기 (Windows)

1. WSL2를 사용해서 설치해야 합니다. 공식 설명서를 참고하세요.
2. 이후는 Linux 설치 방법과 동일합니다.

OCaml 설치하기

1. 쉘에서 다음 명령어를 실행합니다.

```
opam init
```

   초기화 후 나오는 안내에 따라 y를 입력하여 쉘 설정(.bashrc, .zshrc 등)을 변경합니다.

2. 이후

```
eval $(opam env --switch=default)
```

   를 실행합니다.

3. OCaml이 설치되었는지 확인합니다.

```
ocaml -version
```

프로그래밍 환경 설정하기

취향에 따라 (Neo)vim, Emacs, Visual Studio Code 등을 사용할 수 있습니다.

1. 미리 OPAM을 통해 Dune, ocaml-lsp-server, ocamlformat을 설치하세요.

```
opam install dune ocaml-lsp-server ocamlformat
eval $(opam env)
```

2.
   - Vim 및 Emacs 사용자는 https://dev.realworldocaml.org/install.html을 참고하세요.
   - VSCode 사용자는 OCaml Platform 플러리긴을 설치하세요.
   - 이외의 편집기 사용자는 어떻게 사용하는지 본인이 더 잘 알고 있을 것입니다…

실행기 UTop

- Python의 REPL과 같이, OCaml에도 UTop이라는 훌륭한 실행기가 있습니다.

```
opam install utop
eval $(opam env)
```

- OCaml 코드 조각을 손쉽게 실행할 수 있습니다.

OCaml 첫 걸음

OCaml이 뭐지?

- OCaml은 여러 종류의 프로그래밍 방식을 지원합니다.
  - 명령형 *Imperative*, 물건 중심 *object-oriented*, 값 중심 *value-oriented, functional* 등
  - 실용성과 안전함의 조화
- Java, Python, Go 등과 같이 자동으로 메모리를 재활용 *garbage collected*합니다.
- 강력한 (정적) 타입 시스템을 가지고 있습니다.
  - 타입으로 많은 성질을 보장
- OCaml 5부터는 병렬 프로그래밍을 지원합니다.

명령형과 값 중심 프로그래밍의 차이

- 기계가 순서대로 명령을 실행
- "변수"는 변할 수 있는 값을 담는 그릇

```
a = 0
a = a + 1
print(a)
```

- 기계가 주어진 값을 계산
- "값"은 수학의 세계에서처럼 변하지 않음

```
let a = 0 in
let a = a + 1 in (* 동명이인 *)
print a
```

Hello, World!

UTop에서…

```
utop # print_endline "Hello, World!";;
Hello, World!
- : unit = ()
```

Hello, World!

UTop에서…

```
utop # print_endline "Hello, World!";;
Hello, World!
- : unit = ()
```

- 왜 괄호가 없지?

```
utop # print_endline("Hello, World!");;
Hello, World!
- : unit = ()
```

Hello, World!

UTop에서…

```
utop # print_endline "Hello, World!";;
Hello, World!
- : unit = ()
```

- 왜 괄호가 없지?

```
utop # print_endline("Hello, World!");;
Hello, World!
- : unit = ()
```

- ;;는 뭐지?

Hello, World!

UTop에서…

```
utop # print_endline "Hello, World!";;
Hello, World!
- : unit = ()
```

- 왜 괄호가 없지?

```
utop # print_endline("Hello, World!");;
Hello, World!
- : unit = ()
```

- ;;는 뭐지?
- unit이 뭐지?

Hello, World!

UTop에서…

```
utop # print_endline "Hello, World!";;
Hello, World!
- : unit = ()
```

- 왜 괄호가 없지?

```
utop # print_endline("Hello, World!");;
Hello, World!
- : unit = ()
```

- ;;는 뭐지?
- unit이 뭐지?
- ()는 뭐지?

계산기

```
utop # "hel" ^ "lo"  (* 주석: 문자열 *);;
- : string = "hello"
utop # true && false  (* 불리언 *);;
- : bool = false
utop # 1 + 3  (* 정수 *);;
- : int = 4
utop # 1 / 3;;
- : int = 0
utop # 1. /. 3.  (* 부동소수점 *);;
- : float = 0.33333333333333315
utop # ( && );;
- : bool -> bool -> bool = <fun>
utop # ( + );;
- : int -> int -> int = <fun>
utop # ( / );;
- : int -> int -> int = <fun>
utop # ( /. );;
- : float -> float -> float = <fun>
```

이름 붙이기

```
utop # let x = 2;;
val x : int = 2
utop # let y = x * 2;;
val y : int = 4
utop # let x' = succ x;;
val x' : int = 3
utop # succ;;
- : int -> int = <fun>
utop # let _ = x;;
- : int = 2
utop # let X = x;;
Error: Unbound constructor X
utop # let x-prime = x;;
Error: Syntax error
utop # let 1x = x;;
Error: Unknown modifier 'x' for literal 1x
```

이름 안에 이름 붙이기

`let` x `=` e `in` e'에서 x는 e' 안에서만 이름이 유효합니다.

```
utop # let echo () =
  let i = read_int () in
  let str = string_of_int i in
  print_endline ("> " ^ str);;
val echo : unit -> unit = <fun>
utop # echo ();;
42
> 42
- : unit = ()
```

이름 안에 이름 붙이기

`let` x `=` e `in` e'에서 x는 e' 안에서만 이름이 유효합니다.

```
utop # let echo () =
  let i = read_int () in
  let str = string_of_int i in
  print_endline ("> " ^ str);;
val echo : unit -> unit = <fun>
utop # echo ();;
42
> 42
- : unit = ()
```

- x를 재귀적으로 정의하려면?

함수도 값

```
utop # (fun x -> x * x) 2;;
- : int = 4
utop # let square x = x * x;;
val square : int -> int = <fun>
utop # let square = fun x -> x * x;;
val square : int -> int = <fun>
utop # let square = function x -> x * x;;
val square : int -> int = <fun>
utop # square 2;;
- : int = 4
utop # let cube x = x * square x;;
val cube : int -> int = <fun>
utop # cube 2;;
- : int = 8
```

조건문과 재귀 함수

- 함수가 자신의 정의에 사용되려면 `rec` 지정어를 사용해야 합니다.
  - `let rec` x `=` e `in` e'에서 x는 e와 e' 안에서 이름이 유효합니다.
- `(=)` 비교 연산자를 통해 값이 같은지 비교합니다.
- `if` p `then` e `else` e'와 같이 조건문을 사용할 수 있습니다.
  - `else` 부분은 `then` 부분이 `unit` 타입일 경우 생략할 수 있습니다.

```
utop # let rec fact n =
  if n = 0 then 1 else n * fact (n - 1);;
val fact : int -> int = <fun>
utop # fact 10;;
- : int = 3628800
```

타입 추론

사실 앞에서 모두 타입을 직접 명시하지 않았고, OCaml이 스스로 타입을 추론한 것입니다!

```
utop # let square (x : int) : int = x * x;;
val square : int -> int = <fun>
utop # let square : int -> int = fun x -> x * x;;
val square : int -> int = <fun>
utop # let try_first pred value fallback =
  if pred value then value else fallback;;
val try_first : ('a -> bool) -> 'a -> 'a -> 'a = <fun>
```

타입 추론

사실 앞에서 모두 타입을 직접 명시하지 않았고, OCaml이 스스로 타입을 추론한 것입니다!

```
utop # let square (x : int) : int = x * x;;
val square : int -> int = <fun>
utop # let square : int -> int = fun x -> x * x;;
val square : int -> int = <fun>
utop # let try_first pred value fallback =
  if pred value then value else fallback;;
val try_first : ('a -> bool) -> 'a -> 'a -> 'a = <fun>
```

마지막은 타입 변수 'a에 어떠한 타입이든 들어갈 수 있는 (다형 *polymorphic*) 함수입니다.

타입 변환 함수들

```
utop # int_of_float;;
- : float -> int = <fun>
utop # float_of_int;;
- : int -> float = <fun>
utop # int_of_float 3.14;;
- : int = 3
utop # float_of_int 1 /. 2.;;
- : float = 0.5
```

데이터 조립과 분해

데이터 조립과 분해: 순서쌍 *Pair, Tuple*

조립은 ,로, 분해는 ,를 사용한 **패턴 매칭** *pattern matching* 혹은 두 원소의 순서쌍(pair)이라면 `fst`와 snd 함수로 할 수 있습니다.

```
utop # let a_tuple = (3, "one")  (* 조립 *);;
val a_tuple : int * string = (3, "one")
utop # let (x, y) = a_tuple  (* 분해 *);;
val x : int = 3
val y : string = "one"
utop # fst a_tuple;;
- : int = 3
utop # snd a_tuple;;
- : string = "one"
utop # let dist (x1, y1) (x2, y2) =
  sqrt ((x1 -. x2) ** 2. +. (y1 -. y2) ** 2.);;
utop # dist (1., 2.) (4., 6.);;
- : float = 5.
utop # 1, 2, 3;;
- : int * int * int = (1, 2, 3)
```

데이터 조립과 분해: 리스트 *List*

조립은 `::`와 `[]`로, 분해는 `::`와 `[]`를 사용한 패턴 매칭 혹은 `List.hd`와 `List.tl` 함수로 할 수 있습니다. 나아가 리스트의 모든 원소는 같은 타입이어야 합니다.

```
utop # 0 :: 1 :: 2 :: 3 :: [];;
- : int list = [0; 1; 2; 3]
utop # let lst = [3; 1; 4; 1; 5; 9]  (* 문법 설탕 *);;
val lst : int list = [3; 1; 4; 1; 5; 9]
utop # let head = List.hd lst;;
val head : int = 3
utop # let tail = List.tl lst;;
val tail : int list = [1; 4; 1; 5; 9]
utop # List.nth lst 3;;
- : int = 1
utop # List.mem 0 lst;;
- : bool = false
utop # lst @ [2; 6; 5; 3; 5];;
- : int list = [3; 1; 4; 1; 5; 9; 2; 6; 5; 3; 5]
utop # ( @ );;
- : 'a list -> 'a list -> 'a list = <fun>
```

데이터 조립과 분해: 리스트와 패턴 매칭

```
utop # let rec sum l =
  match l with
  | [] -> 0                    (* base case *)
  | hd :: tl -> hd + sum tl    (* inductive case *);;
val sum : int list -> int = <fun>
utop # let rec sum = function
  | [] -> 0
  | hd :: tl -> hd + sum tl;;
val sum : int list -> int = <fun>
utop # sum [1; 2; 3];;
- : int = 6
```

데이터 조립과 분해: 리스트와 패턴 매칭

```
utop # let rec sum l =
  match l with
  | [] -> 0                    (* base case *)
  | hd :: tl -> hd + sum tl    (* inductive case *);;
val sum : int list -> int = <fun>
utop # let rec sum = function
  | [] -> 0
  | hd :: tl -> hd + sum tl;;
val sum : int list -> int = <fun>
utop # sum [1; 2; 3];;
- : int = 6
```

match를 중첩하여 사용할 때는 (match ... with ...)와 같이 괄호로 감싸서 사용합니다.

데이터 조립과 분해: 새로운 타입 만들기

새로운 타입을 만들어 조립할 수 있는 구성자 *constructor*를 직접 정의할 수 있습니다. 분해는 정의한 구성자를 통한 패턴 매칭으로 할 수 있습니다.

아래 color처럼 여러 값을 가질 수 있는 타입을 갈래 *variant* 타입[1]이라고 부릅니다.

```
utop # type color = Red | Green | Blue;;
type color = Red | Green | Blue
utop # let color_to_string = function
  | Red -> "red"
  | Green -> "green"
  | Blue -> "blue";;
val color_to_string : color -> string = <fun>
utop # color_to_string Red;;
- : string = "red"
```

[1]새로운 번역이 있으면 쉬운 전문용어/variant type에 의견을 제시해주세요!

데이터 조립과 분해: 귀납적 타입 *Inductive Type*

tree처럼 정의가 귀납적인 구조를 만들 수 있습니다.

```
utop # type tree =
  | Leaf of int
  | Node of int * tree * tree;;
type tree = Leaf of int | Node of int * tree * tree
utop # let t1 = Leaf 0;;
val t1 : tree = Leaf 0
utop # let t2 = Leaf 1;;
val t2 : tree = Leaf 1
utop # let t3 = Node (2, t1, t2);;
val t3 : tree = Node (2, Leaf 0, Leaf 1)
utop # let rec sum_of_tree tree =
  match tree with
  | Leaf n -> n
  | Node (n, l, r) -> n + sum_of_tree l + sum_of_tree r;;
val sum_of_tree : tree -> int = <fun>
utop # sum_of_tree t3;;
- : int = 3
```

데이터 조립과 분해: 레코드 타입 *Record Type*

C 계열의 언어에서 구조체 *struct*와 비슷한 레코드 타입을 정의할 수 있습니다.

```
utop # type rgb = { red : int; green : int; blue : int };;
type rgb = { red : int; green : int; blue : int; }
utop # let cyan = { red = 0; green = 255; blue = 255; };;
val cyan : rgb = {red = 0; green = 255; blue = 255}
utop # cyan.red;;
- : int = 0
utop # cyan.blue;;
- : int = 255
utop # let { red = r; green; blue } = cyan;;
val r : int = 0
val green : int = 255
val blue : int = 255
```

커링 *Currying*

$$f : A \times B \to C \quad \xrightarrow{\text{curry}} \quad f' : A \to B \to C$$

```
utop # let addp (a, b) = a + b;;
val addp : int * int -> int = <fun>
utop # let add a b = a + b;;
val add : int -> int -> int = <fun>
utop # let add42 = add 42;;
val add42 : int -> int = <fun>
utop # add42 1337;;
- : int = 1379
```

커링 *Currying*

$$f : A \times B \to C \quad \xrightarrow{\text{curry}} \quad f' : A \to B \to C$$

```
utop # let addp (a, b) = a + b;;
val addp : int * int -> int = <fun>
utop # let add a b = a + b;;
val add : int -> int -> int = <fun>
utop # let add42 = add 42;;
val add42 : int -> int = <fun>
utop # add42 1337;;
- : int = 1379
```

꼭 과제에서 요구하는 타입에 맞추어 프로그램을 작성하세요!

다형성 *Polymorphism*

타입을 가리지 않고 받는 다형 함수를 만들어 보다 유연한 프로그램을 작성할 수 있습니다.

```
utop # let rec len lst =
  match lst with
  | x :: xs -> 1 + len xs
  | [] -> 0;;
val len : 'a list -> int = <fun>
utop # len [1; 2; 3];;
- : int = 3
utop # len ["hello"; "world"];;
- : int = 2
```

다형성 *Polymorphism*

타입을 가리지 않고 받는 다형 함수를 만들어 보다 유연한 프로그램을 작성할 수 있습니다.

```
utop # let rec len lst =
  match lst with
  | x :: xs -> 1 + len xs
  | [] -> 0;;
val len : 'a list -> int = <fun>
utop # len [1; 2; 3];;
- : int = 3
utop # len ["hello"; "world"];;
- : int = 2
```

`len` 함수는 `'a` 타입에 무엇이 오든 상관없이 작동합니다.

- `'a` = `int`일 때나
- `'a` = `string`일 때나…

더 둘러보기

참조 *Reference*

명령형 프로그래밍 언어처럼 값을 변경할 수 있는 참조 *reference* 타입이 있습니다.

```
utop # let x = ref 0;;
val x : int ref = {contents = 0}
utop # x;;
- : int ref = {contents = 0}
utop # !x;;
- : int = 0
utop # x := !x + 1;;
- : unit = ()
utop # !x;;
- : int = 1
utop # incr x;;
- : unit = ()
utop # !x;;
- : int = 2
utop # incr;;
- : int ref -> unit = <fun>
```

예외 *Exception* 처리 i

- Java, Python, C++ 등과 같이 예외를 처리할 수 있습니다.
- match와 동일한 방식으로 try로 예외를 처리할 수 있습니다.

```
utop # 1 / 0;;
Exception: Division_by_zero.
utop # (* type 'a option = None | Some of 'a *)
let try_div a b =
  try Some (a / b) with
  | Division_by_zero -> None;;
val try_div : int -> int -> int option = <fun>
utop # try_div 63 9;;
- : int option = Some 7
utop # try_div 3 0;;
- : int option = None
```

예외 *Exception* 처리 ii

- 새로운 타입을 정의하듯이 `exception`을 통해 새로운 예외를 정의할 수 있습니다.
- `raise`로 예외를 발생시킬 수 있습니다.

```
utop # exception Explosion;;
exception Explosion
utop # let bomb = fun n ->
  let secret_code = 1337 in
  if n <> secret_code  (* 값이 다른지 비교 *)
  then raise Explosion
  else ();;
val bomb : int -> unit = <fun>
utop # bomb 12345;;
Exception: Explosion.
utop # bomb 1337;;
- : unit = ()
```

모듈 *Module* 시스템

모듈은 연관된 코드의 집합입니다.

- 모든 파일은 모듈이고, 명시적으로 모듈을 정의할 수 있습니다.
- Signature는 모듈의 타입을 정의하고, functor를 써서 "모듈 함수"를 만들 수 있습니다.
- 대부분의 경우 조교 팀에서 모듈을 설계해서 뼈대를 제공합니다.

```
utop # module type Color_intf = sig
  type t
  val red : t
  val blue : t
  val print : t -> string
end;;
module type Color_intf =
  sig type t val red : t val blue : t val print : t -> string end
utop # module Color_bool : Color_intf = struct
  type t = bool
  let red = true
  let blue = false
  let print c = if c then "red" else "blue"
end;;
module Color_bool : Color_intf
utop # Color_bool.print Color_bool.red;;
- : string = "red"
utop # Color_bool.(print blue);;
- : string = "blue"
utop # let open Color_bool in print blue;;
- : string = "blue"
```

컴파일하기 i

실제로 파일을 만들어서 컴파일해봅시다!

```
(* sample.ml *)

exception List_empty

let rec take (n : int) (xs : 'a list) : 'a list =
  if n = 0 then []
  else
    match xs with [] -> raise List_empty | x :: xs' -> x :: take (n - 1) xs'

let rec drop (n : int) (xs : 'a list) : 'a list =
  if n = 0 then xs
  else match xs with [] -> raise List_empty | _ :: xs' -> drop (n - 1) xs'

let rec print_list : int list -> unit = function
  | [] -> print_newline ()
  | x :: xs ->
      print_int x;
      print_string ";";
      print_list xs
```

컴파일하기 ii

```
let rec merge (cmp : 'a -> 'a -> bool) (xs : 'a list) (ys : 'a list) : 'a list =
  match (xs, ys) with
  | [], l | l, [] -> l
  | x :: xs', y :: ys' ->
      if cmp x y then x :: merge cmp xs' ys else y :: merge cmp xs ys'
let rec merge_sort (cmp : 'a -> 'a -> bool) (xs : 'a list) : 'a list =
  match xs with
  | [] | [ _ ] -> xs
  | _ ->
      let len = List.length xs in
      let left = take (len / 2) xs in
      let right = drop (len / 2) xs in
      merge cmp (merge_sort cmp left) (merge_sort cmp right)
let () =
  let lst = [ 3; 2; 1; 10; 5; 4; 7; 6; 9; 8 ] in
  let sorted = merge_sort ( < ) lst in
  print_list lst;
  print_string " -> ";
  print_list sorted
```

컴파일하기 iii

```
$ ocaml sample.ml
3;2;1;10;5;4;7;6;9;8;
 -> 1;2;3;4;5;6;7;8;9;10;
$ utop -init sample.ml
                ┌──────────────────────────────────────────────────────────────┐
                │ Welcome to utop version 2.13.1 (using OCaml version 5.1.0)! │
                └──────────────────────────────────────────────────────────────┘
3;2;1;10;5;4;7;6;9;8;
 -> 1;2;3;4;5;6;7;8;9;10;
Type #utop_help for help about using utop.
─( 00:43:09 )─< command 0 >──────────────────────────────────────{ counter: 0 }─
utop # merge;;
- : ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list = <fun>
─( 00:43:09 )─< command 1 >──────────────────────────────────────{ counter: 0 }─
utop # merge_sort;;
- : ('a -> 'a -> bool) -> 'a list -> 'a list = <fun>
─( 00:43:13 )─< command 2 >──────────────────────────────────────{ counter: 0 }─
utop # merge_sort ( > ) [4; 6; 1; 9];;
- : int list = [9; 6; 4; 1]
─( 00:43:16 )─< command 3 >──────────────────────────────────────{ counter: 0 }─
utop #
```

컴파일하기 iv

```
$ ocamlc sample.ml -o sample
$ ./sample
3;2;1;10;5;4;7;6;9;8;
 -> 1;2;3;4;5;6;7;8;9;10;
```

빌드 시스템: Dune

- Dune을 사용해 OCaml 프로젝트를 쉽게 만들고 빌드할 수 있습니다.
- 뼈대 코드가 Dune을 사용하여 제공되는 경우, 자세한 사용 방법을 안내해드리도록 하겠습니다.

마치며

주의할 점

- 반드시 실행되는 코드를 제출해주세요.
- 반드시 디버깅용 코드를 지우고 제출해주세요.
- `match`를 중첩하여 사용할 때나, 문법 오류가 있을 때 괄호로 적절히 감쌌는지 확인하세요.
- 과제에서 요구하는 타입에 반드시 맞추어 프로그램을 작성하세요.
  - `let f x y`와 `let f (x, y)`는 타입이 다릅니다.

더 알아보기

- 반드시 설명서를 읽어보세요: **The OCaml Manual**
- `ropas.snu.ac.kr/~ta/4190.310/24`
- OCaml Programming: Correct + Efficient + Beautiful
- Real World OCaml, 2nd Edition
- OCaml Discussion

참고 자료

- 역대 OCaml 튜토리얼
  - **2011** 이원찬, 윤용호, 김진영
  - **2013** 최준원, 강동옥
  - **2015** 최재승
  - **2017** 이동권
  - **2018** 이동권, 배요한
  - **2019** 고현수
  - **2021** 김세훈
  - **2022** 박규연
  - **2023** 김도형
- Real World OCaml, 2nd Edition
- OCaml from the Very Beginning