OCaml Tutorial

2013년 가을 프로그래밍 언어

최준원, 강동옥
{jwchoi, dokang}@ropas.snu.ac.kr

서울대학교 프로그래밍연구실

차례

- 시작하기 전 : OCaml?
  - 어떤 언어인가
  - 설치, 실행 및 컴파일
- 시작하기 : 이름 붙이기 / 함수 / 값 / 타입
- 더 나아가서
  - currying
  - pair / tuple / list / record
  - match - with / try - with, raise
  - inductive type / polymorphic type
  - module system
  - reference
- 마치며 - 참고자료

OCaml

- 다중 패러다임을 가지는 상위(high-level) 언어
  - 함수형, 절차형, 객체지향 등의 프로그래밍을 모두 지원 가능.
- 안전성(safety)과 편의성에 중점을 둔 언어
  - 타입 시스템
  - 사용자가 정의한 타입 / 패턴 매칭
  - 자동으로 메모리 관리

컴파일러 설치

- 방법 1 : ocaml.org - Use - Releases
  - 소스를 받아서 직접 컴파일 할 수도 있고,
  - 운영체제에 맞는 binary distribution 을 받아 설치할 수도 있음
- 방법 2 : apt-get (Ubuntu) / port(MAC) 에서 ocaml 패키지 찾기
- 방법 1,2 사용시 3.10.2 버전으로 설치 (반드시)
- 방법 3 : martini 등의 snucse 서버 사용

실행: ocaml

- 정상적으로 설치했다면, ocaml 로 실행 가능

```
jwchoi@type:~$ ocaml
        Objective Caml version 3.11.2

# let a = 1;;
val a : int = 1
# let b = a;;
val b : int = 1
# a + b;;
- : int = 2
# let incr n = n + 1;;
val incr : int -> int = <fun>
# incr 3;;
- : int = 4
# #quit;;
jwchoi@type:~$
```

실행: ocaml

- 편집한 파일을 ocaml 로 불러오고,
실행할 수 있음

test.ml

```
1 let _ =
2   let msg = "Hello World!" in
3   print_endline msg
4
```

console

```
jwchoi@type:~/tmp$ vi test.ml
jwchoi@type:~/tmp$ ocaml test.ml
Hello World!
jwchoi@type:~/tmp$
```

컴파일: **ocamlc**

- 작성한 파일을 ocamlc 로 컴파일 할 수 있음

```
jwchoi@type:~/tmp$ vi test.ml
jwchoi@type:~/tmp$ ocaml test.ml
Hello World!
jwchoi@type:~/tmp$ ocamlc test.ml
jwchoi@type:~/tmp$ ./a.out
Hello World!
jwchoi@type:~/tmp$
```

실행기와 컴파일러

- 실행기
  - 식을 입력하면 바로 결과를 볼 수 있다.
  - 끝에 ;; 를 붙여주어야 함.
- 컴파일러
  - 끝에 ;; 를 붙이지 않음.
  - 컴파일의 대상이 되는 파일은 let 의 연속으로 구성됨 (잠시 뒤에 let 설명)
  - 과제로는 컴파일이 되는(꼭 확인!) .ml 파일을 제출

OCaml 시작

- 다시 강조: 왜 OCaml 로 실습할까?
  - 상위 언어이기 때문에 편리하다.
  - 다중 패러다임을 지원하기 때문에 우리가 하고 싶은 일들을 원하는 방법으로 해낼 수 있음.
  - 안전성을 자동으로 검사해 주기 때문에 걱정이 없다
    - 컴파일만 잘 된다면.
  - 값 중심의 언어이기 때문에 복잡하지 않다
    - 한 번 이름붙여진 값은 변하지 않음.

이름붙이기: let

- 값에 이름을 붙이고 사용하고 싶다.

```
let a = 3 in ...
```

  - 앞으로 3이라는 값의 이름은 a이다.
  - 변수가 아님 : a 이름에 해당하는 값은 변하지 않음.

```
let incr = fun x -> x + 1 in ...
```

  - 앞으로 x를 받아 x+1을 돌려주는 함수의 이름은 incr 이다.

함수와 이름

- 함수에 이름을 붙이고 자유롭게 사용할 수 있다.

```
# let incr n = n + 1;;
val incr : int -> int = <fun>
# incr 3;;
- : int = 4
#
```

- 함수를 이름 없이 정의하고 사용할 수도 있음.

```
# (fun n -> n + 1) 3;;
- : int = 4
#
```

함수와 이름

- 재귀 함수에는 반드시 이름이 필요 : 어떤 함수를 불러야 하는지는 알아야...
  - 추가로 재귀 함수임을 알리는 rec 키워드를 붙임.

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

함수에 이름붙이기 정리

- 결국, 아래 세 가지는 모두 같은 의미
  - ```
    let incr = (fun n -> n + 1) in
      incr 3
    ```
  - ```
    let incr n = n + 1 in
      incr 3
    ```
  - ```
    (fun n -> n + 1) 3
    ```

- 상황에 따라 적절한 방법으로 사용하기
    ```

값

- Boolean
  - `let b = true in ...`
- 정수
  - `let a = 1 in ...`
- 문자열
  - `let s = “hello world!” in ...`
- 함수도 값
  - `let incr = (fun n -> n + 1) in ...`
- 그 밖에 : pair / tuple / list / record - 잠시 뒤에

타입

- 모든 값은 타입을 가진다.

```
# let a = 1;;
val a : int = 1
# let f = 3.0;;
val f : float = 3.
# let s = "hello world!";;
val s : string = "hello world!"
# let incr = (fun n -> n + 1);;
val incr : int -> int = <fun>
#
```

타입은 자동으로 추론

- 내가 쓰지도 않은 타입이 표기되어 있다?
  - OCaml 의 타입 추론 시스템 : 자동으로 정확하게 값의 타입을 추론해 준다.
  - 잘못된 타입으로 프로그래밍 했다면 에러를 내 줌.

```
# let incr n = n + 1;;
val incr : int -> int = <fun>
# incr "This is a string";;
Error: This expression has type string but an expression was expected of type
         int
# 
```

  - 과제를 제출할 때는 과제에서 요구한 타입 명세가 맞는지 반드시 확인한 후 제출.

타입 명세와 과제, 조금 더

- 과제의 예
  - Exercise. 두 정수를 받아 최대공약수를 반환하는 함수 gcd 를 정의하세요:

    gcd 의 타입은 `gcd : int -> int -> int`.

- 제출 파일의 예

```
1 let rec gcd a b =
2     if a = 1 || b = 1 then 1
3     else if a = b then a
4     else if a < b then gcd a (b-a)
5     else gcd (a-b) b
```

타입 명세와 과제, 중요

- 다시 강조 : 타입 명세에 맞는 프로그램을 구현.
- 제출할 때 테스트 케이스, 디버그용 문자열 등을 제거했는지 확인.

```
1 let rec gcd a b =
2   if a = 1 || b = 1 then 1
3   else if a = b then a
4   else if a < b then gcd a (b-a)
5   else gcd (a-b) b
6
7 let test = gcd 3 5
8
9 let _ = print_endline "Debug string"
```

더 나아가서

- 쉽고 효율적인 프로그래밍을 위해 알아야 하는 여러가지 내용들
  - currying
  - pair / tuple / list / record
  - match - with / try - with, raise
  - inductive type / polymorphic type
  - module system
  - reference

Currying

- 함수가 여러 개의 인자를 받는 방법
  - 한꺼번에 넘겨주기
  - 하나 넘겨주고, 다음 하나 넘겨주고, ...
- gcd 함수로 비교해보면,
  - gcd (a, b) 의 함수 인자는 한 개 : (a, b) // pair
  - gcd a b 의 함수 인자는 두 개 : a, 그리고 b
- Currying 은 함수 인자를 하나씩 넘겨주는 방법

Why Currying

- 함수 인자를 하나씩 넘겨주면 일부 인자가 넘겨진 함수를 사용할 수 있음.

```
# #use "exercise.ml";;
val gcd : int -> int -> int = <fun>
# let gcd3 = gcd 3;;
val gcd3 : int -> int = <fun>
# gcd3 6;;
- : int = 3
#
```

Currying: 주의

- 세 번째 강조 : 과제의 타입 명세에 맞는 프로그램을 제출하세요.
  - 절대 봐 드리지 않음
  - 타입 명세 틀려서 0점 처리되는 일이 없도록 주의

- 예
  - `gcd : int -> int -> int` 인데,
  - `gcd : (int * int) -> int` 로 작성해서 제출하면 0점.

Pair

- 너무나도 사용하기 직관적이고 편리함.
  - 정의는 (a, b) / 풀고 싶을 때는 fst & snd 사용.

```
# let p = (1, 2);;
val p : int * int = (1, 2)
# fst p;;
- : int = 1
# snd p;;
- : int = 2
#
```

Tuple

- Pair 의 연장선. 여러 종류의 타입을 묶음.
  - 정의는 (a1, a2, ..., an)
  - 풀고 싶다면 아쉽게도 직접 함수를 정의해야...
    (뒤의 match - with 참조)

```
# let tr = (true, 1, "string");;
val tr : bool * int * string = (true, 1, "string")
# let get_second t =
    match t with
    | (_, s, _) -> s
  ;;
val get_second : 'a * 'b * 'c -> 'b = <fun>
# get_second tr;;
- : int = 1
#
```

List

- 정말 많이 사용되는 자료구조
- 리스트는
  - 빈 리스트이거나 - `[]`
  - 원소와 리스트의 결합 - `3::[]`
- 두 가지 생성 방법을 사용하여 자유롭게 리스트 만들기
  - `1::2::3::[]`
  - 편리한 표기법 (같은 의미) - `[1; 2; 3]`

List

- 리스트의 원소는 모두 같은 타입이어야 함.

```
# let wrong_list = 1::2::true::[];;
Error: This expression has type bool but an expression was expected of type
         int
# 
```

- Tuple 과 다른 점이 무엇인가요 : Tuple 은 생성되는 각각의 타입이 다름. List는 모두 같은 타입.

```
# let tuple1 = (1, 2, 3);;
val tuple1 : int * int * int = (1, 2, 3)
# let tuple2 = (1, 2, 3, 4);;
val tuple2 : int * int * int * int = (1, 2, 3, 4)
# let list1 = [1; 2; 3];;
val list1 : int list = [1; 2; 3]
# let list2 = [1; 2; 3; 4];;
val list2 : int list = [1; 2; 3; 4]
# 
```

List

- OCaml List library
  - http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html
  - 알아두면 한 학기동안 큰 도움이 됨
  - 리스트 관련해서 필요한 것이 생기면 구현하기 전에 여기서 먼저 검색

Record

- 여러 개의 값을 묶어 하나의 값으로 사용하기.

```
type student_info = {
  id : int;
  name : string
};;
```

```
let jwchoi =
  { id = 100; name = “Joonwon Choi” };;
```

- Tuple 과 다른 점이 무엇인가요 : Tuple 에는 각각의 원소에 이름이 없음. Record 는 이름을 주면서 이름으로 원소 값에 접근할 수 있음.

Record

- 정의하고, 원소 값에 접근하기.

```
# type student_info = { id:int; name:string };;
type student_info = { id : int; name : string; }
# let jwchoi = { id = 100; name = "Joonwon Choi" };;
val jwchoi : student_info = {id = 100; name = "Joonwon Choi"}
# jwchoi.id;;
- : int = 100
# jwchoi.name;;
- : string = "Joonwon Choi"
#
```

match - with

- 패턴 매칭 : 값을 경우를 나누어 계산하고 싶을 때 사용.

```
match x with
| A -> a
| B -> b
| _ -> default
```

match - with

- 예) 정수 리스트의 합 구하기

```
# let rec sum_of_list l =
    match l with
     | [] -> 0
     | head::tail -> head + (sum_of_list tail)
  ;;
val sum_of_list : int list -> int = <fun>
#
```

- 자동으로 타입 추론이 됨을 확인!

try - with, raise

- 예외 처리
  - Java / C++ 의 그것과 비슷
  - try 로 예외가 생길만한 부분을 묶고,
    with 으로 예외를 처리하고,
    raise 는 예외를 발생시킨다.

```
try ... with Exception -> ...
```

try - with, raise

- 예) Division_by_zero
(OCaml 에서 제공하는 예외)

```
# let div_by_zero =
    try
      let a = 1 in
      let b = 0 in
      a / b
    with Division_by_zero -> 1000
  ;;
val div_by_zero : int = 1000
#
```

- 역시 자동으로 타입 추론이 됨을 확인

Inductive Type

- 사용자가 직접 타입을 정의할 수 있다.
- 예) 정수를 원소로 가지는 이진 트리 타입 정의

```
type tree = Leaf of int | Node of tree * tree
let t = Node (Leaf 2, Leaf 4);;
```

Inductive Type

- 함수 예) 이진 트리의 모든 원소의 합 구하기
  - match - with 을 사용하면 쉽다.

```
# let rec sum_of_tree t =
    match t with
    | Leaf n -> n
    | Node (ltree, rtree) ->
      (sum_of_tree ltree) + (sum_of_tree rtree)
  ;;
val sum_of_tree : tree -> int = <fun>
# let t = Node (Node (Leaf 1, Leaf 2),
                Node (Leaf 3, Leaf 4));;
val t : tree = Node (Node (Leaf 1, Leaf 2), Node (Leaf 3, Leaf 4))
# sum_of_tree t;;
- : int = 10
#
```

Polymorphic Type

- 서로 다른 타입의 원소를 가지는 리스트이지만 비슷한 일을 하고 싶을 때가 있다.
  - 예) 리스트의 길이 구하기.
    - `length [1; 2; 3] = 3`
    - `length [true; false; true; false] = 4`
- 해결 : 다형 타입(polymorphic type) 시스템이 일반적인 타입에 대한 함수 정의를 가능하게 해 준다.

Polymorphic Type

- 예) 리스트의 길이 구하는 다형 타입 함수 구현

```
# let rec list_length l =
    match l with
      | [] -> 0
      | head::tail -> 1 + (list_length tail)
  ;;
val list_length : 'a list -> int = <fun>
#
```

- ‘α 는 임의의 타입. 함수가 계산될 때마다 결정된다.

Polymorphic Type

- 다형 타입으로 함수를 구현하면,
  - 비슷한 일을 한꺼번에 해 주니까 효율적으로 코드를 작성할 수 있겠다!
- OCaml List library 또한 다형 타입으로 구현되어 있음.
  - 리스트의 head 가져오기
  - `List.hd : 'a list -> 'a`
  - 'a 타입의 리스트를 'b 타입의 리스트로 변환하기
  - `List.map : ('a -> 'b) -> 'a list -> 'b list`

Reference

- let ... in ... 과는 달리 변수를 선언하고 사용할 수 있음.
  - 0이라는 값을 가리키는 참조자(reference) 선언

```
let r = ref 0;;
```

- 값을 가져오려면 ! 연산자를 이용한다.

```
let value_of_r = !r;;  // value_of_r = 0
```

- 값을 대입하려면 := 연산자를 이용한다.

```
r := !r + 1;;  // !r = 1
```

Reference

- 예) 참조자를 사용하여 1부터 100까지의 합 계산하기.

```
# let sum = ref 0;;
val sum : int ref = {contents = 0}
# let rec calculate n =
    if n = 0 then ()
    else (sum := !sum + n; calculate (n-1))
  ;;
val calculate : int -> unit = <fun>
# calculate 100;;
- : unit = ()
# !sum;;
- : int = 5050
#
```

Module System

- 모듈 : 관련 있는 코드들의 묶음.
- 예) 이전에 만든 정수 이진 트리 타입과 원소의 합을 구하는 함수를 같은 모듈에 넣어보기.

```
# module IntBinaryTree =
  struct
    type tree = Leaf of int | Node of tree * tree

    let rec sum_of_tree t =
      match t with
      | Leaf n -> n
      | Node (ltree, rtree) ->
        (sum_of_tree ltree) + (sum_of_tree rtree)
  end;;
```

Module System

- 모듈 안의 값을 사용하고 싶을 때는
  - [모듈 이름].[값 이름]
  - 타입도 모듈 이름이 같이 적용되어 추론됨.

```
# let t = IntBinaryTree.Node (IntBinaryTree.Leaf 1,
                              IntBinaryTree.Leaf 2);;
val t : IntBinaryTree.tree =
  IntBinaryTree.Node (IntBinaryTree.Leaf 1, IntBinaryTree.Leaf 2)
#
```

Module System

- 그냥 묶는 기능이 끝인가요? 아니요.
  - Signature : 묶은 뒤에 보여주고 싶은 것만 보여주기 (abstraction)
    - http://caml.inria.fr/pub/docs/manual-ocaml/manual004.html#toc14
  - Functor : 모듈을 받아서 새로운 모듈을 만들어주기
    - http://caml.inria.fr/pub/docs/manual-ocaml/manual004.html#toc15

참고 자료

- OCaml Library
  - http://caml.inria.fr/pub/docs/manual-ocaml/libref/index.html
  - 이미 언급한 List 뿐만 아니라 모든 자료구조에 대해서 유용하게 검색, 사용할 수 있음 : 먼저 찾아보고 없으면 구현하기.
- 좀 더 친절한 OCaml 설치, 실행, 컴파일 메뉴얼
  - http://ropas.snu.ac.kr/~ta/4190.210/12/practice/ocaml_tutorial.pdf

참고 자료

- 조교팀 홈페이지에 있는 모든 자료들
  - http://ropas.snu.ac.kr/~ta/4190.310/13/
    - OCaml Reference
    - Introduction to Objective Caml
    - 각종 예제들
    - 이전 OCaml 튜토리얼들
      (이 튜토리얼에서도 많이 참고함

마치며

- 조교팀 홈페이지에서 이 튜토리얼을 다운받아
  - 강조한 부분을 다시 한 번 읽어보기
  - 억울한 경우가 발생하지 않도록

- 수강생 모두가 즐겁게 배우고 능력을 모두 발휘하기를 기대합니다.