

# **Beginning OCaml II: Functions**

**CAS CS 320: Principles of Programming Languages**

Thursday, January 25, 2024

## Administrivia

- Homework 0 (not graded) is due today by 11:59 pm.
- Homework 1 (graded) is posted today and due on Thursday, Feb 1, by 11:59 pm.
- The office hours calendar has been updated with locations.

## Administrivia

- Homework 0 (not graded) is due today by 11:59 pm.
- Homework 1 (graded) is posted today and due on Thursday, Feb 1, by 11:59 pm.
- The office hours calendar has been updated with locations.

## Reading Assignment

- OCP Section 2.4: **Functions**
- OCP Section 2.6: **Printing**

## function definitions (OCP 2.4.1)

```
(* increment function "inc" *)  
let inc x = x + 1 ;;  
  
(* increment function "inc1" *)  
let inc1 x = x +. 1.0 ;;
```

**Question 1:** What is the type of "inc"?

**Question 2:** What is the type of "inc1"?

## function definitions (OCP 2.4.1, 2.4.2)

```
(* increment function "inc" *)  
let inc x = x + 1 ;;  
  
(* increment function "inc1" *)  
let inc1 x = x +. 1.0 ;;
```

**Question 1:** What is the type of "inc"?

**Question 2:** What is the type of "inc1"?

```
(* anonymous increment "(fun x -> x + 1)" applied to 3 *)  
(fun x -> x + 1) 3 ;;
```

```
(* different way of defining named "inc" *)  
let inc = fun x -> x + 1 ;;
```

## function definitions (OCP 2.4.1)

```
(* squaring function "square" *)  
let square x = x * x ;;  
  
(* squaring function "square1" *)  
let square1 x = x *. x ;;
```

**Question 1:** What is the type of "square"?

**Question 2:** What is the type of "square1"?

## function definitions (OCP 2.4.1, 2.4.2)

```
(* squaring function "square" *)  
let square x = x * x ;;  
  
(* squaring function "square1" *)  
let square1 x = x *. x ;;
```

**Question 1:** What is the type of "square"?

**Question 2:** What is the type of "square1"?

```
(* anonymous squaring "(fun x -> x * x)" applied to 3 *)  
(fun x -> x * x) 3 ;;
```

```
(* different way of defining "square" *)  
let square = fun x -> x * x ;;
```

## recursive function definitions (OCP 2.4.1)

```
(* factorial function with if-then-else *)  
(* requires: [n >= 0] *)  
let rec fact n =  
    if n = 0 then 1 else n * fact (n - 1)
```

```
(* factorial function with types inserted *)  
(* requires: [n >= 0] *)  
let rec fact1 (n : int) : int =  
    if n = 0 then 1 else n * fact1 (n - 1);;
```

```
(* print first 15 factorials 0! 1! ... 15! *)  
let () =  
    for n = 0 to 15 do Printf.printf "%d! = %d\n" n (fact n)  
    done ;;
```



## recursive function definitions (OCP 2.4.1)

```
(* factorial function with pattern matching *)  
(* requires: [n >= 0] *)  
let rec fact2 n =  
    match n with  
    | 0 -> 1  
    | n -> n * fact2 (n - 1) ;;
```

## recursive function definitions (OCP 2.4.1)

```
(* power function with if-then-else *)  
(* requires: [y >= 0]. *)  
let rec pow1 x y =  
    if y = 0 then 1 else x * pow1 x (y - 1);;
```

```
(* power function with pattern matching *)  
(* requires: [y >= 0]. *)  
let rec pow2 x y =  
    match y with  
    | 0 -> 1  
    | y -> x * pow2 x (y - 1) ;;
```

## recursive function definitions (OCP 2.4.1)

```
(* GCD function with if-then-else *)  
(* requires: [n >= 0 && m >= 0] ??? *)  
let rec gcd1 n m =  
  if m = 0 then n else gcd1 m (n mod m) ;;
```

```
(* GCD function with pattern matching *)  
(* requires: [n >= 0 && m >= 0] ??? *)  
let rec gcd2 n m =  
  match m with  
    | 0 -> n  
    | m -> gcd2 m (n mod m) ;;
```

## mutually recursive function definitions – using the “and” keyword (OCP 2.4.1)

```
(* [even n] is whether [n] is even.  
   Requires: [n >= 0]. *)  
let rec even (n : int) : bool =  
    n = 0 || odd (n - 1)  
  
(* [odd n] is whether [n] is odd.  
   Requires: [n >= 0]. *)  
and odd (n : int) : bool =  
    n <> 0 && even (n - 1);;
```

## mutually recursive function definitions – using the “and” keyword (OCP 2.4.1)

```
let rec first x = match x with  
  | 1 -> 1  
  | x -> second (x mod 10)  
and second x = first (x + 1);;
```

- **Question 1:**  
What are the types of “first” and “second”?
- **Question 2:**  
What do “first” and “second” compute?

## function application (OCP 2.4.3)

**Question:** Which expressions are correctly parenthesized?

1. gcd inc 5 square 4
2. (((gcd inc) 5) square) 4)
3. (gcd (inc (5 (square 4))))
4. (gcd (inc 5) (square 4))
5. gcd (inc 5) square 4
6. gcd inc 5 (square 4)
7. gcd (inc 5) (square 4)

## function application (OCP 2.4.3)

**Question:** Which expressions are correctly parenthesized?

- |                               |     |
|-------------------------------|-----|
| 1. gcd inc 5 square 4         | NO  |
| 2. (((gcd inc) 5) square) 4)  | NO  |
| 3. (gcd (inc (5 (square 4)))) | NO  |
| 4. (gcd (inc 5) (square 4))   | YES |
| 5. gcd (inc 5) square 4       | NO  |
| 6. gcd inc 5 (square 4)       | NO  |
| 7. gcd (inc 5) (square 4)     | YES |

## function application (OCP 2.4.3)

**Question:** Which expressions are correctly parenthesized?

- |                               |     |
|-------------------------------|-----|
| 1. gcd inc 5 square 4         | NO  |
| 2. (((gcd inc) 5) square) 4)  | NO  |
| 3. (gcd (inc (5 (square 4)))) | NO  |
| 4. (gcd (inc 5) (square 4))   | YES |
| 5. gcd (inc 5) square 4       | NO  |
| 6. gcd inc 5 (square 4)       | NO  |
| 7. gcd (inc 5) (square 4)     | YES |

**more generally:**

$e_0 e_1 \dots e_n$  means  $(\dots (e_0 e_1) \dots e_n)$   
provided  $(e_i : t \rightarrow u)$  and  $(e_{\{i+1\}} : t)$



## type-checking function application

```
(gcd : int -> (int -> int))  
  (inc : int -> int)  
    (5 : int)  
      (square : int -> int)  
        (4 : int)
```

## type-checking function application

```
(gcd : int -> (int -> int))  
  (inc : int -> int)  
    (5 : int)  
      (square : int -> int)  
        (4 : int)
```

```
(gcd : int -> (int -> int))  
  (inc : int -> int)  
    (5 : int) : int  
      (square : int -> int)  
        (4 : int) : int
```

## type-checking function application

```
(gcd : int -> (int -> int))  
  (inc : int -> int)  
    (5 : int)  
      (square : int -> int)  
        (4 : int)
```

```
(gcd : int -> (int -> int))  
  (inc : int -> int)  
    (5 : int) : int  
      (square : int -> int)  
        (4 : int) : int
```

```
(gcd : int -> (int -> int))  
  (inc : int -> int)  
    (5 : int) : int : (int -> int)  
      (square : int -> int)  
        (4 : int) : int
```

## pipelining (OCP 2.4.4)

instead of `(inc (square (inc 4)));;` we can **pipeline** the function applications:

```
4 |> inc |> square |> inc ;;
```

**more generally:** `e_1 |> e_2` means `(e_2 e_1)`  
provided that `(e_1 : t)` and `(e_2 : t -> u)`

## pipelining (OCP 2.4.4)

instead of `(inc (square (inc 4)));;` we can **pipeline** the function applications:

```
4 |> inc |> square |> inc ;;
```

**more generally:** `e_1 |> e_2` means `(e_2 e_1)`  
provided that `(e_1 : t)` and `(e_2 : t -> u)`

pipelining binary (or non-unary) functions, such as `gcd`, is a little cumbersome ...

**Question:** how to pipeline `gcd (inc 5) (square 4) ;;`?

## pipelining (OCP 2.4.4)

instead of `(inc (square (inc 4)))`; we can **pipeline** the function applications:

```
4 |> inc |> square |> inc ;;
```

**more generally:** `e_1 |> e_2` means `(e_2 e_1)`  
provided that `(e_1 : t)` and `(e_2 : t -> u)`

pipelining binary (or non-unary) functions, such as `gcd`, is a little cumbersome ...

6

16

**Question:** how to pipeline `gcd (inc 5) (square 4)` ; ; ?

**Answer:** `4 |> square |> (5 |> inc |> gcd)` ; ;

## polymorphic functions (OCP 2.4.5)

the identity function `id` is **polymorphic**:

```
let id1 x = x ;;  
let id2 = fun x -> x ;;
```

in both cases, the type is `val id : 'a -> 'a = <fun>` .

## polymorphic functions (OCP 2.4.5)

the identity function `id` is **polymorphic**:

```
let id1 x = x ;;  
let id2 = fun x -> x ;;
```

in both cases, the type is `val id : 'a -> 'a = <fun>` .

We can enforce a specific type for the identity, e.g.

```
let id3 (x : int) = x ;;  
let id4 (x : bool) = x ;;  
let id5 (x : (int -> bool) -> (int -> bool)) = x ;;  
let id6 (x : ('a -> 'b) -> ('a -> 'b)) = x ;;
```



## labeled and optional arguments (OCP 2.4.6)

```
(* inserting labels ~lbl1 and ~lbl2, and types *)  
let foo1 ~lbl1:(x : int) ~lbl2:(y : int) = x / y ;;  
(* same as preceding after omitting the type *)  
let foo2 ~lbl1: x ~lbl2: y = x / y ;;
```

the type of foo1 and foo2 is `lbl1:int -> lbl2:int -> int`

## labeled and optional arguments (OCP 2.4.6)



```
(* inserting labels ~lbl1 and ~lbl2, and types *)  
let foo1 ~lbl1:(x : int) ~lbl2:(y : int) = x / y ;;  
(* same as preceding after omitting the type *)  
let foo2 ~lbl1: x ~lbl2: y = x / y ;;
```

the type of foo1 and foo2 is `lbl1:int -> lbl2:int -> int`

a little more confusing ...

```
(* labels ~x and ~y have same names as args x and y!!! *)  
let foo3 ~x:(x : int) ~y:(y : int) = x / y ;;
```

the type of foo3 is `x:int -> y:int -> int`

## labeled and optional arguments (OCP 2.4.6)

```
(* inserting labels ~lbl1 and ~lbl2, and types *)  
let foo1 ~lbl1:(x : int) ~lbl2:(y : int) = x / y ;;  
(* same as preceding after omitting the type *)  
let foo2 ~lbl1: x ~lbl2: y = x / y ;;
```

the type of foo1 and foo2 is `lbl1:int -> lbl2:int -> int`

a little more confusing ...

```
(* labels ~x and ~y have same names as args x and y!!! *)  
let foo3 ~x:(x : int) ~y:(y : int) = x / y ;;
```

the type of foo3 is `x:int -> y:int -> int`

**Question:** what is the point of labels?

**Answer:** we can permute the argument, e.g. we can write  
`foo3 ~y:6 ~x:40` instead of `foo3 40 6` .

## partial application (OCP 2.4.7)

Once more, the power function:

```
(* power function with if-then-else *)  
(* requires: [y >= 0]. *)  
let rec pow x y =  
    if y = 0 then 1 else x * pow x (y - 1);;
```

the interpreter returns the following type :

```
val pow : int -> int -> int = <fun>
```

implicitly "->" associates to the right, i.e. the type is:

```
int -> (int -> int)
```

## partial application (OCP 2.4.7)

Once more, the power function:

```
(* power function with if-then-else *)  
(* requires: [y >= 0]. *)  
let rec pow x y =  
    if y = 0 then 1 else x * pow x (y - 1);;
```

the interpreter returns the following type :

```
val pow : int -> int -> int = <fun>
```

implicitly “->” associates to the right, i.e. the type is:

```
int -> (int -> int)
```

we can apply “pow” to one argument at a time as in:

```
let power_of_2 = pow 2 ;;  
val power_of_2 : int -> int = <fun>
```

## function associativity (OCP 2.4.8)

a function with more than one argument, e.g.:

```
let f x_1 x_2 x_3 x_4 = e
```

is semantically equivalent to

```
let f =  
  fun x_1 ->  
    (fun x_2 ->  
      (fun x_3 ->  
        (fun x_4 -> e))))
```

the type of function `f` is of the form:

```
t_1 -> (t_2 -> (t_3 -> t_4))           t_1 -> (t_2 -> (t_3 -> (t_4 -> e)))
```

since function types are *right associative*, the type of `f` is:

```
t_1 -> t_2 -> t_3 -> t_4
```

( THIS PAGE INTENTIONALLY LEFT BLANK )