OCaml Functions & Pattern Matching

Jibesh Patra, Soumyajit Dey
Week 2 – Software Engineering – CS20202 – Spring 2026 – IIT Kharagpur

OCaml Functions

Functions

- In OCaml, everything is a value, functions are values too.
- Functions are first-class values.
- No *return* keyword in OCaml.

The identifier `square` receives a functional type `int → int`

```
let square = fun x -> x * x;;
let square x = x * x;;
square 10;;
```

```
val square : int -> int = <fun>
- : int = 100
```

value can't be displayed

Takes a single parameter `x`. the function body is the expression `x * x`.

Functions with Multiple Parameters

OCaml functions can have multiple parameters.

generic type

```
let max x y = if x <= y then y else x;;
```

```
val max : 'a -> 'a -> 'a = <fun>
```

<= is polymorphic. Works for integers, floats, strings, characters, lists

- Take an argument of any type.
- Take a second argument of the same type.
- Return a value of that same type.

```
max [1;2] [3;1];;
```

```
- : int list = [3; 1]
```

Works for only `int`

```
let max x y : int = if x <= y then y else x;;
```

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

Functions with Multiple Parameters

```
let lst1 = [1; 2; 3; 4];;
let lst2 = [11; 22; 33; 440];;
let combine x1 x2= x1 @ x2;;
combine lst1 lst2;;
```

list concatenation operator

```
val lst1 : int list = [1; 2; 3; 4]
val lst2 : int list = [11; 22; 33; 44]
val combine : 'a list -> 'a list -> 'a list = <fun>
- : int list = [1; 2; 3; 4; 11; 22; 33; 44]
```

Functions with Multiple Parameters

```
let max x y : int = if x <= y then y else x;;
let max_of_three x y z = max (max x y) z;;
max_of_three 1 2 3;;
```

no need to specify return type

```
val max : int -> int -> int = <fun>
val max_of_three : int -> int -> int -> int = <fun>
- : int = 3
```

```
let max (x : int) (y : int) : int = if x <= y then y else x;;
```

Can annotate argument types

Labelled and Optional Arguments

```
let say_hello  ~(name:string)  ~(salutation:string) ?(course:string="SE") ()= "Hello, " ^
salutation ^" "^ name ^ ". Welcome to " ^ course;;
say_hello ~name:"Spiderman" ~salutation:"Mr." ();;
say_hello ~salutation:"Mr." ~name:"Batman" ();;
say_hello ~name:"Superman" ~course:"FLAT" () ~salutation:"Mr.";;
say_hello "Shaktiman" ~course:"FLAT" () ~salutation:"Mr.";;
```

Optional argument

```
val say_hello :
  name:string -> salutation:string -> ?course:string -> unit -> string = <fun>
- : string = "Hello, Mr. Spiderman. Welcome to SE"
- : string = "Hello, Mr. Batman. Welcome to SE"
- : string = "Hello, Mr. Superman. Welcome to FLAT"

Error: The function applied to this argument has type name:string -> string This argument
cannot be applied without label
```

Anonymous Functions

- Do not have a name.
- Can write anonymous function and immediately apply to a value.
- Multiple parameters possible
- Use case: Higher-Order Function → A *function* may expect a *function* as a parameter or return functions as values.

```
let nums = [1; 2; 3; 4];;
let squares = List.map (fun x -> x * x);;
squares nums;;
```

part of the predefined library of functions for `lists`

```
val nums : int list = [1; 2; 3; 4]
val squares : int list -> int list = <fun>
- : int list = [1; 4; 9; 16]
```

Recursive Functions

```
let rec nsum n = if n = 0 then 0 else n + nsum (n - 1);;
nsum 4;;
```

```
val nsum : int -> int = <fun>
- : int = 10
```

explicit `rec` keyword.

```
let rec nsum n = if n = 0 then 0 else n + nsum (n - 1);;
nsum 100000000;;
```

```
val nsum : int -> int = <fun>
Stack overflow during evaluation (looping recursion?).
```

Tail Recursion

Functional programming languages are designed such that tail recursive functions are preferable over loops.

```
let rec gcd x y = if y < 1 then x else gcd y (x mod y);;
gcd 21 18;;
```

```
val gcd : int -> int -> int = <fun>
- : int = 3
```

Tail Recursion

```
let rec summing n total = if n = 0 then total else summing (n - 1) (n + total);;
```

Often recursive functions that are not tail recursive can be reformulated as tail recursive functions by introducing an extra argument serving as accumulator argument.

The recursive calls do not nest; the additions are done immediately.

```
let nsum n =
 let rec summing n total = if n = 0 then total else summing (n - 1) (n + total)
 in
 summing n 0;;
nsum 100000000;;
```

Works fine

All OCaml Functions take Exactly One Argument

$f(1, 2, 3) \rightarrow f$ takes only one argument (the argument 1), returning a function that takes the second argument 2, again returning a function that takes the third and final argument 3, returning the final value.

equivalent

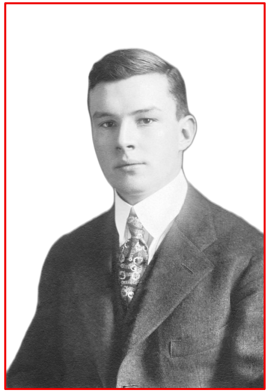
```
let add x y = x + y;;
```

```
let add = fun x -> (fun y -> x + y);;
```

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

Currying

- Haskell Curry (1900 - 1982) was a logician and computer scientist.
- Name of two programming languages after him.
- Use of higher-order functions to simulate functions of multiple arguments.



```
let lst1 = [1; 2; 3; 4];;
let lst2 = [11; 22; 33; 440];;
let combine x1 x2 = x1 @ x2;;
combine lst1 lst2;;
let cmb = combine lst1;;
cmb [100; 200];;
```

```
val lst1 : int list = [1; 2; 3; 4]
val lst2 : int list = [11; 22; 33; 440]
val combine : 'a list -> 'a list -> 'a list = <fun>
- : int list = [1; 2; 3; 4; 11; 22; 33; 440]
val cmb : int list -> int list = <fun>
- : int list = [1; 2; 3; 4; 100; 200]
```

Partial Application

- Functions don't have to be called with all the arguments they expect.
- True for curried functions.

```
let lst1 = [1; 2; 3; 4];;
let lst2 = [11; 22; 33; 440];;
let combine x1 x2 = x1 @ x2;;
combine lst1 lst2;;
let cmb = combine lst1;;
cmb [100; 200];;
```

```
val lst1 : int list = [1; 2; 3; 4]
val lst2 : int list = [11; 22; 33; 440]
val combine : 'a list -> 'a list -> 'a list = <fun>
- : int list = [1; 2; 3; 4; 11; 22; 33; 440]
val cmb : int list -> int list = <fun>
- : int list = [1; 2; 3; 4; 100; 200]
```

Introduction to Pattern Matching

Value Constructors

- Value constructor of a type creates data of that type.
- For the type `tuple`, the value is formed using the value constructor for tuples, an infix comma.
  - The type of a tuple is determined by the type of its parts. Example, if it is a pair `(true, 100)`, the type is `bool * int.` Read it as “bool cross int”
  - `(100, true)` and `(true, 100)` are completely different types. One is “int cross bool” while the other is “bool cross int”.
- Another example: For list, we can create value using the `::` operator.
  - The reason why it is also called the *cons* operator.

Pattern Matching

- Value constructor can construct composite value from parts, how to extract the parts from composite structure?
  - Use the same value constructor.
- Pattern matching generalizes case analysis.
- Pattern matching allows us to inspect data of any kind, except functions.
- The following is the general form of pattern matching:

$$
\begin{aligned}
\langle \text{expr} \rangle ::= \texttt{match}\ \langle \text{expr}_{\text{value}} \rangle\ \texttt{with} \\
&\mid \langle \text{pattern}_1 \rangle\ \texttt{->}\ \langle \text{expr}_1 \rangle \\
&\mid \langle \text{pattern}_2 \rangle\ \texttt{->}\ \langle \text{expr}_2 \rangle \\
&\ldots
\end{aligned}
$$

Pattern Matching Example

```
let add_two_numbers (nums : int * int) : int =
match nums with
| x, y -> x + y ;;

add_two_numbers (2, 3);;
```

```
val add_two_numbers : int * int -> int = <fun>
- : int = 5
```

Using the value constructor , to destructure.

Pattern Matching Example

```
let rec sum lst =
 match lst with
 | [] -> 0
 | head :: tail -> head + sum tail;;
```

Using the value constructor :: to destructure.

```
sum [1; 2; 3];;
```

```
val sum : int list -> int = <fun>
- : int = 6
```

head :: tail pattern in the second matching expression is used to destructure the list into head and tail where head is the first element and tail is the rest of the list.

Pattern Matching Example

- Pattern matching also works for atomic types.

```
let int_of_bool (cond : bool) : int =
match cond with
| true -> 1
| false -> 0 ;;

int_of_bool true;;
```

```
val int_of_bool : bool -> int = <fun>
- : int = 1
```

👑

```
let int_of_bool (cond : bool) : int =
if cond then 1 else 0 ;;
```

Pattern Matching Example

```
let check_int (x : int) : bool =
match abs x with
| 0 -> true
| 1 -> true
| 2 -> true
| _ -> false ;;

check_int ~-1;;
check_int 2;;
check_int 7;;
```

wild-card pattern

```
val check_int : int -> bool = <fun>
- : bool = true
- : bool = true
- : bool = false
```

unary negation operator

This is the end