Functional Programming

{percywtc}

2021-05-01

Outline

- What is Functional Programming?
- Introduction to OCaml
- OCaml mini-minicomp

What is Functional Programming?

Pure Functions

- Identical return value for identical arguments
- No side-effects
  - I/O
  - Changing global variables

```
int f(int x) {
  return x / 2;
}

int g(int x, int y) {
  return x > y ? x : y;
}

int global_x;
int h(int y) {
  return global_x + y;
}

int global_y;
int j(int x) {
  global_y++;
  return x + 1;
}

void k(int x) {
  cout << x;
}
```

Pure Functions

- Identical return value for identical arguments
- No side-effects
  - I/O
  - Changing global variables

```
int f(int x) { // pure
  return x / 2;
}

int g(int x, int y) { // pure
  return x > y ? x : y;
}

int global_x;
int h(int y) { // impure
  return global_x + y;
}

int global_y;
int j(int x) { // impure
  global_y++;
  return x + 1;
}

void k(int x) { // impure
  cout << x;
}
```

Functional Programming

In a pure functional programming language, all functions should be pure

Restricting yourself only writing pure functions

- Fewer bugs
- Easier to debug

Introduction to OCaml

OCaml

- (not purely) functional
- object oriented
- type inferred
  - the compile can detect the type of variables and functions even if it is not provided
- static typed
  - the type of variables are determined at compile time

ML → Caml → OCaml

OCaml: Tools

- utop - a universal toplevel (i.e., REPL) for OCaml
  - *REPL: read–eval–print loop, also termed an interactive toplevel or language shell*
  - Recommended if you're using Unix systems now (MacOS, Ubuntu, WSL, etc.)
  - https://github.com/ocaml-community/utop
  - Preferred way: use `opam` to install
    - Install `opam` first - https://opam.ocaml.org/doc/Install.html
    - Then install `utop` via `opam` - `opam install utop`
    - Pay attention to "*# To setup the new switch in the current shell, you need to run: ...*"
  - Alternative way: `sudo apt instal utop` (version might be older)
- https://try.ocamlpro.com/ - an online version REPL

OCaml: Expressions

Expressions are ended with ;;

It will come with an evaluation result

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

# let x = 42;;
val x : int = 42

# cos 1.0;;
- : float = 0.540302305868139765

# let x = 1.0 in cos x;;
- : float = 0.540302305868139765
```

OCaml: Expressions

Expressions are ended with ;;

It will come with an evaluation result

```
# let x = 3 in let y = 4 in x + y;;
- : int = 7

# let z = let x = 3 in let y = 4 in x + y;;
val z : int = 7

#
let z =
  let x = 3 in
  let y = 4 in
  x + y
;;
val z : int = 7

# y;;
Line 1, characters 0-1:
Error: Unbound value y
```

OCaml: Basic Types

- int
- float
- bool
- char
- string

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

# 42.0;;
- : float = 42.

# true;;
- : bool = true

# 'f';;
- : char = 'f'

# "hkoi";;
- : string = "hkoi"
```

OCaml: Basic Operators

- `int` `+ - * / mod`
- `float` `+. -. *. /.`
- `bool` `&& ||`
- `char`
- `string` `^`

Comparisons `= <>`

- don't use == and !=

No implicit type casting

- use `float_of_int`, `int_of_float` etc.

```
# 3 + 5;;
- : int 8

# 1.414 + 3.141;;
Error: This expression has type float but an expression
was expected of type int

# 1.414 +. 3.141;;
- : float = 4.555

# "hk" ^ "oi";;
- : string = "hkoi"

# "hkoi" = "hk" ^ "oi";;
- : bool = true

# 2 + 3 <> 5;;
- : bool = false

# int_of_float 3.14;;
- int = 3
```

OCaml: Range of Int

In usual 64-bit systems:

- -2^{62} to $2^{62}-1$

Why not "-2^{63} to $2^{63}-1$" or "-2^{31} to $2^{31}-1$"?
https://dev.realworldocaml.org/runtime-memory-layout.html
https://stackoverflow.com/questions/3773985/why-is-an-int-in-ocaml-only-31-bits

(also implementation-dependent)
https://try.ocamlpro.com/ might gives $2^{31}-1$ to 2^{31}

```
# Int.min_int;;
- : int = -4611686018427387904

# Int.max_int;;
- : int = 4611686018427387903

# Int.max_int + 1;;
- : int = -4611686018427387904
```

OCaml: First Class Functions

Functions are treated like

- first-class citizens
- just like other basic data types

```
# sqrt;;
- : float -> float = <fun>

# let f = sqrt;;
val f : float -> float = <fun>

# f 2.0;;
- : float = 1.41421356237309515

# let g = fun x -> x + 3;;
val g : int -> int = <fun>

# g 10;;
- : int = 13
```

OCaml: First Class Functions

Functions are treated like

- first-class citizens
- just like other basic data types

```
# (+);;
- : int -> int -> int = <fun>

# (+) 3 4;;
- : int = 7

# let add_three = (+) 3;;
val add_three : int -> int = <fun>

# add_three 4;;
- : int = 7

# let twice f x = f (f x);;
val twice : ('a -> 'a) -> 'a -> 'a = <fun>

# twice sqrt 2.0;;
- : float = 1.18920711500272103

# twice add_three 4;;
- : int = 10
```

OCaml: Immutable

Variables in OCaml are immutable (in normal cases)

```
# let foo = 42;;
val foo : int = 42

# let foo_plus_x x = foo + x;;
val foo_plus_x : int -> int = <fun>

# let foo = 3;;
val foo : int = 3

# foo_plus_x 1000;;
- : int = 1042

# foo;;
- : int = 3
```

OCaml: if .. then .. else ..

You can treat it like `?:` of C++

Usually, you should always write `else`

```
# let x = if 3 + 2 = 5 then "cool";;
Error: This expression has type string but an expression
was expected of type unit because it is in the result of
a conditional with no else branch

# let x = if 3 + 2 = 5 then "cool" else "sad";;
val x : string = "cool"
```

OCaml: Lists

Lists in OCaml are linked lists

```
# #show list;;
type nonrec 'a list = [] | (::) of 'a * 'a list

# [];;
- : 'a list = []

# 1 :: 2 :: [];;
- : int list = [1; 2]

# [1;2;3;4];;
- : int list = [1; 2; 3; 4]

# 1 :: [2;3;4];;
- : int list = [1; 2; 3; 4]

# [1;2] :: [3;4;5];;
Error: This expression has type int but an expression
was expected of type int list

# [1;2] @ [3;4;5];;
- : int list = [1; 2; 3; 4; 5]
```

OCaml: Pattern Matchings

Pattern matchings with `let`

```
# let hd :: tl = [1;2;3;4];;
val hd : int = 1
val tl : int list = [2; 3; 4]

# let hd :: tl = [1];;
val hd : int = 1
val tl : int list = []

# let hd :: tl = [];;
Exception: Match_failure

# let a, b = 3, 4;; (* tuple *)
val a : int = 3
val b : int = 4

# let s, hd :: tl = "hkoi", [3.0; 4.0; 5.0];;
val s : string = "hkoi"
val hd : float = 3.
val tl : float list = [4.; 5.]
```

OCaml: Pattern Matchings

Pattern matchings with `match`

```
#
let lengthy l =
  match l with
  | hd :: hd2 :: tl -> "so long"
  | hd :: tl -> "1 element"
  | [] -> "empty"
;;
val lengthy : 'a list -> string = <fun>



#
let lengthy_bad l =
  match l with
  | hd :: tl -> "1 element"
  | hd :: hd2 :: tl -> "so long"
  | [] -> "empty"
;;
Line 5, characters 4-19:
Warning 11: this match case is unused.
val lengthy_bad : 'a list -> string = <fun>
```

OCaml: Options

C++ equivalent is `std::optional`

- https://en.cppreference.com/w/cpp/utility/optional
- Only since C++17

```
# #show option;;
type nonrec 'a option = None | Some of 'a

#
let get_second_element l =
  match l with
  | hd :: hd2 :: tl -> Some hd2
  | _ -> None
;;
val get_second_element : 'a list -> 'a option = <fun>

# get_second_element [1;2;3;4];;
- : int option = Some 2

# get_second_element [1];;
- : int option = None

# get_second_elemnt [];;
- : 'a option = None
```

OCaml: Options

Pattern match with Option

```
#
let get_or_default x default =
  match x with
  | Some v -> v
  | None -> default
;;
val get_or_default : 'a option -> 'a -> 'a = <fun>
```

OCaml: Recursive Functions

For recursive functions

- add keyword `rec` to indicate

```
#
let rec sum l =
  match l with
  | [] -> 0
  | hd :: tl -> hd + sum tl
;;
val sum : int list -> int = <fun>



#
let rec add_1_to_all l =
  match l with
  | [] -> []
  | hd :: tl -> (hd + 1) :: add_1_to_all tl
;;
val add_1_to_all : int list -> int list = <fun>
```

OCaml: Recursive Functions

For mutual recursion

- add keyword `and` to indicate

```
#
let rec even x =
  if x = 0
  then true
  else odd (x-1)
and odd x =
  if x = 0
  then false
  else even (x-1)
;;
val even : int -> bool = <fun>
val odd : int -> bool = <fun>
```

OCaml: Tail Recursion

```
# (* non tail recursion *)
let rec sum l =
  match l with
  | [] -> 0
  | hd :: tl -> hd + sum tl
;;
val sum : int list -> int = <fun>

(*
sum [1;2;3;4] =
1 + (sum [2;3;4]) =
1 + (2 + sum [3;4]) =
1 + (2 + (3 + sum [4])) =
1 + (2 + (3 + (4 + sum []))) =
1 + (2 + (3 + (4 + 0))) =
... =
10
*)
```

```
# (* tail recursion *)
let rec sum_tail l acc =
  match l with
  | [] -> acc
  | hd :: tl -> sum_tail tl (acc + hd)
;;
val sum_tail : int list -> int -> int = <fun>

(*
sum_tail [1;2;3;4] 0 =
sum_tail [2;3;4] 1 =
sum_tail [3;4] 3 =
sum_tail [4] 6 =
sum_tail [] 10 =
10
*)
```

OCaml: Tail Recursion

- Call itself recursively
- No computation after return
- immediately return callee's value

Compiler optimization:

- Caller's stack frame will pop;
  Then push callee's stack frame
- Avoid call stack overflow
- C++ also supports

https://en.wikipedia.org/wiki/Tail_call

```
# (* non tail recursion *)
let rec sum l =
  match l with
  | [] -> 0
  | hd :: tl -> hd + sum tl
;;
val sum : int list -> int = <fun>


# (* tail recursion *)
let rec sum_tail l acc =
  match l with
  | [] -> acc
  | hd :: tl -> sum_tail tl (acc + hd)
;;
val sum_tail : int list -> int -> int = <fun>
(* sum_tail [1;2;3;4] 0 =
   sum_tail [2;3;4] 1 =
   sum_tail [3;4] 3 =
   sum_tail [4] 6 =
   sum_tail [] 10 =
   10 *)
```

OCaml: Better Sum Function

hide the implementation of `sum_tail`

only expose `int list -> int`

```
# (* tail recursion *)
let rec sum_tail l acc =
  match l with
  | [] -> acc
  | hd :: tl -> sum_tail tl (acc + hd)
;;
val sum_tail : int list -> int -> int = <fun>

#
let sum l =
  let rec aux l acc =
    match l with
    | [] -> acc
    | hd :: tl -> aux tl (hd + acc)
  in
  aux l 0
;;
```

OCaml: Example - Manhattan Distance

Given two lists `l` and `l2`

Write a function `dist l l2` :

- `int list -> int list -> int`

Examples:

- `dist [1;2] [2;1]` : 2
- `dist [1;3;5] [2;4;6]` : 3
- `dist [] []` : 0

```
#
let rec dist l l2 =
  match l, l2 with
  | hd :: tl, hd2 :: tl2 -> (abs (hd - hd2)) + dist tl tl2
  | [], [] -> 0
;;
Line 3, characters 2-93:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
(([], _::_)|(_::_, []))
val dist : int list -> int list -> int = <fun>



#
let rec dist l l2 =
  match l, l2 with
  | hd :: tl, hd2 :: tl2 -> (abs (hd - hd2)) + dist tl tl2
  | [], [] -> 0
  | _, _ -> raise (Invalid_argument "dist")
;;
```

OCaml: List Module

https://ocaml.org/api/List.html

```
# #show List;;
module List = List
module List :
  sig
    type 'a t = 'a list = [] | (::) of 'a * 'a list
    val length : 'a t -> int
    val compare_lengths : 'a t -> 'b t -> int
    val compare_length_with : 'a t -> int -> int
    val cons : 'a -> 'a t -> 'a t
    val hd : 'a t -> 'a
    val tl : 'a t -> 'a t
    val nth : 'a t -> int -> 'a
    val nth_opt : 'a t -> int -> 'a option
    val rev : 'a t -> 'a t
    val init : int -> (int -> 'a) -> 'a t
    (* ... *)
  end
```

OCaml: Example - Manhattan Distance

```
val map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
```

`map2 f [a1; ...; an] [b1; ...; bn]` is `[f a1 b1; ...; f an bn]`.

Raises `Invalid_argument` if the two lists are determined to have different lengths. Not tail-recursive.

```
val rev_map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
```

`rev_map2 f l1 l2` gives the same result as `List.rev ( List.map2 f l1 l2)`, but is tail-recursive and more efficient.

"|>" is a pipe operator

You can also check out:
https://ocaml.org/api/List.html#VALfold_left2

```
val fold_left2 : ('a -> 'b -> 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a
```

`fold_left2 f init [a1; ...; an] [b1; ...; bn]` is
`f (... (f (f init a1 b1) a2 b2) ...) an bn`.

Raises `Invalid_argument` if the two lists are determined to have different lengths.

```
# let abs_diff x y = abs (x - y);;
val abs_diff : int -> int -> int = <fun>

# List.map2 abs_diff [1;2;4] [8;16;0];;
- : int list = [7; 14; 4]

# List.rev_map2 abs_diff [1;2;4] [8;16;0];;
- : int list = [4; 14; 7]



# sum;;
- : int list -> int = <fun>

# let dist l l2 =
  List.rev_map2 (fun x y -> x - y |> abs) l l2
  |> sum
;;
```

OCaml: Writing loops “functionally”

Example: sum of x to y

```
#
let range_sum x y =
  let rec loop i sum =
    if i > y
    then sum
    else loop (i+1) (sum+i)
  in
  loop x 0
;;
val range_sum : int -> int -> int = <fun>
```

Minicomp :)

Don’t worry about I/O, only focus on implementing the functions

Recommend using https://judge.hkoi.org/code for coding and testing

If you really want to compile code on your machine, try:

- `ocamlc -o program.exe program.ml`

More to Learn

We had only gone through a very small portion of the FP world and OCaml world...

https://ocaml.org/learn/tutorials/

https://www.cs.cornell.edu/courses/cs3110/2021sp/textbook/

https://dev.realworldocaml.org/