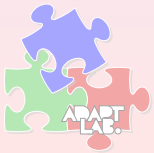
Datatypes in ML

lists, tuples, arrays records, variants ...

Walter Cazzola

Dipartimento di Informatica
Università degli Studi di Milano
e-mail: cazzola@di.unimi.it
twitter: @w_cazzola



OCaML's Primitive Datatypes
Introduction

Even if not explicitly said

- ML is a strongly and statically typed programming language;
- the type of each expression is inferred from the use.

```
[10:46]cazzola@surtur:~/lp/ml>ocaml
# 1+2*3;;
- : int = 7
# let pi = 4.0 * atan 1.0;;
Error: This expression has type float but an expression was expected of type
         int
# let pi = 4.0 *. atan 1.0;;
val pi : float = 3.14159265358979312
# let square x = x *. x;;
val square : float -> float = <fun>
# square 5;;
Error: This expression has type int but an expression was expected of type
         float
# square 5. ;;
- : float = 25.
```

OCaML's Primitive Datatypes
Booleans

OCaML provides two constants

- `true` and `false`

Operations on Booleans

logic operators
&& || not logical and, or and negation respectively
relational operators
== <> equal and not equal to operators
< > <= >= less than, greater than, less than or equal to and greater than or equal to operators

```
[12:01]cazzola@surtur:~/lp/ml>ocaml
# true;;
- : bool = true
# true || false;;
- : bool = true
# 1<2;;
- : bool = true
# 2.5<>2.5;;
- : bool = false
```

OCaML's Primitive Datatypes
Strings

Strings

- they are native in OCaML
- several operations come from the **`String`** module
- since OCaML 4, strings are immutable and <- deprecated
  - **`Bytes`**/**`Bytes`**`.set` must be used instead.

Operations on Strings

`^` string concatenation

`.[]` positional access to chars

```
let s1 = "walter" and s2 = "cazzola" ;;
val s1 : string = "walter"
val s2 : string = "cazzola"
# let s=s1^" "^s2;;
val s : string = "walter cazzola"
# s.[0];;
- : char = 'z'
# String.length(s);;
- : int = 14
# let b = Bytes.of_string s ;;
val b : bytes = Bytes.of_string "walter cazzola"
# Bytes.set b 0 'W'; Bytes.set b 7 'C';;
- : unit = ()
# let s = Bytes.to_string b;;
val s : string = "Walter Cazzola"
```

OCaML's Collections
Lists

Lists

- homogeneous
- cons operator ::
- concatenation operator @ (inefficient).

```
[14:54]cazzola@surtur:~/lp/ml>ocaml
# [1; 1+1; 3];;
- : int list = [1; 2; 3]
# [1; 'a';3.14];;
Error: This expression has type char but an expression was expected of type
         int
# let l1 = [1;2;3] and l2 = [4;5;6];;
val l1 : int list = [1; 2; 3]
val l2 : int list = [4; 5; 6]
# l1@l2;;
- : int list = [1; 2; 3; 4; 5; 6]
# let l1 = 0::l;;
val l1 : int list = [0; 1; 2; 3]
# List.nth l1 2;;
- : int = 2
```

More operations come from the List module.

OCaML's Collections
Lists: Introspecting on the List

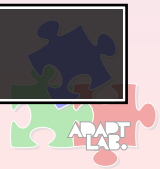
You can check if an element is in the list

```
# let a_list = [2; 7; 25; 3; 11; -1; 0; 7; 25; 25; 999; -25; 7];;
val a_list : int list = [2; 7; 25; 3; 11; -1; 0; 7; 25; 25; 999; -25; 7]
# let rec is_in l x = if l==[] then false else x==List.hd(l) || is_in (List.tl l) x;;
val is_in : 'a list -> 'a -> bool = <fun>
# is_in a_list 11;;
- : bool = true
# is_in a_list 12;;
- : bool = false
```

Count the number of occurrences

```
let count x l =
  let rec count tot x = function
    []    -> tot
  | h::tl -> if (h==x) then count (tot+1) x tl else count tot x tl
  in count 0 x l
```

```
# #use "count2.ml" ;;
val count : 'a -> 'a list -> int = <fun>
# count 7 a_list ;;
- : int = 3
```



OCaML's Collections
Lists: Introspecting on the List (Cont'd)

Look for the position of an item

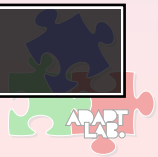
```
let idx l x =
    let rec idx2 l x acc=
      if (List.hd l) == x then acc else idx2 (List.tl l) x (acc+1)
    in idx2 l x 0;;
```

```
# #use "idx.ml" ;;
# idx a_list 999;;
- : int = 10
```

Slice the list from an index to another

```
let slice i j l =
   let rec slice count res = function
      hd::tl when count < i  -> slice (count+1) res tl
    | _      when count == j -> (List.rev res)
    | hd::tl                 -> slice (count+1) (hd::res) tl
in slice 0 [] l ;;
```

```
# #use "slice.ml" ;;
val slice : int -> int -> 'a list -> 'a list = <fun>
# slice 2 5 a_list ;;
- : int list = [25; 3; 11]
```



OCaML's Collections
Tuples

Tuples are

- fixed-length heterogeneous lists.

```
# let a_tuple = (5, 'a', "a string", [1; 2; 3], 3.14);;
val a_tuple : int * char * string * int list * float =
  (5, 'a', "a string", [1; 2; 3], 3.14)
# let a_pair = (1,"w");;
val a_pair : int * string = (1, "w")
# fst a_pair ;;  (* works only on a pair *)
- : int = 1
# snd a_pair;;   (* works only on a pair *)
- : string = "w"
# let a_triplet = ( "a", 0, true);;
val a_triplet : string * int * bool = ("a", 0, true)
# fst a_triplet;;
Error: This expression has type string * int * bool
       but an expression was expected of type 'a * 'b
```

OCaML's Collections
Arrays

Arrays are
- direct-accessible, homogeneous, and mutable lists.

```
# let an_array = [|1;2;3|];;
val an_array : int array = [|1; 2; 3|]
# an_array.(2);;
- : int = 3
# an_array.(1) <- 5;;
- : unit = ()
# an_array ;;
- : int array = [|1; 5; 3|]
```

More operations come from the Array module.

```
# let a = Array.make 5 0;;
val a : int array = [|0; 0; 0; 0; 0|]
# Array.concat [a; an_array] ;;
- : int array = [|0; 0; 0; 0; 0; 1; 5; 3|]
# let a_matrix = Array.make_matrix 2 3 'a' ;;
val a_matrix : char array array = [|[|'a'; 'a'; 'a'|]; [|'a'; 'a'; 'a'|]|]
# a_matrix.(1).(2) <- 'z';; (* .(1).(2) is equivalent to (1,2) *)
- : unit = ()
# a_matrix ;;
- : char array array = [|[|'a'; 'a'; 'a'|]; [|'a'; 'a'; 'z'|]|]
```


OCaML's Collections
Records

Records are
- name accessible (through field names),
- heterogeneous, and
- mutable (through the mutable keyword) tuples.

```
# type person = {name: string; mutable age: int};;
type person = { name : string; mutable age : int; }
# let p = {name = "Walter"; age = 35} ;;
val p : person = {name = "Walter"; age = 35}
# p.name;;
- : string = "Walter"
# p.age <- p.age+1;;
- : unit = ()
# p ;;
- : person = {name = "Walter"; age = 36}
# p.name <- "Walter Cazzola";;
Error: The record field label name is not mutable
```


User Defined Datatype in OCaML
Aliasing & Variants

Aliasing.
The easiest way to define a new type is to give a new name to an existing type.

```
# type int_pair = int*int;;
type int_pair = int * int
# let a : int_pair = (1,3);;
val a : int_pair = (1, 3)
# fst a;;
- : int = 1
```

Any type can be aliased.

Variants.
A variant type lists all possible shapes for values of that type.
- Each case is identified by a capitalized name, called a constructor.

```
# type int_option = Nothing | AnInteger of int ;;
type int_option = Nothing | AnInteger of int
# Nothing;;
- : int_option = Nothing
# AnInteger 7;;
- : int_option = AnInteger 7
```


User Defined Datatype in OCaML
Variants

Mutually recursive type must be declared via the **and** keyword.

```
type card = Card of regular | Joker
  and regular = { suit : card_suit; name : card_name; }
  and card_suit = Heart | Club | Spade | Diamond
  and card_name = Ace | King | Queen | Jack | Simple of int;;
let value = function
  Joker                  ->  0
| Card {name = Ace}       -> 11
| Card {name = King}      -> 10
| Card {name = Queen}     ->  9
| Card {name = Jack}      ->  8
| Card {name = Simple n} ->  n ;;
```

This code defines 4 types.
- the value function gives a value to each card.

```
# #use "cards.ml";;
type card = Card of regular | Joker
and regular = { suit : card_suit; name : card_name; }
and card_suit = Heart | Club | Spade | Diamond
and card_name = Ace | King | Queen | Jack | Simple of int
val value : card -> int = <fun>
# value ( Card { suit = Heart; name = Jack } ) ;;
- : int = 8
```

User Defined Datatype in OCaML
Variants.

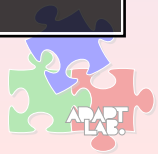
Compared to OO programming,

- a variant type is equivalent to a class hierarchy composed of an abstract base class or interface representing the type and derived classes representing each of the variant type constructors.

Moreover, it is possible to manipulate them by pattern matching.

```
type state = On | Off;;
let turn = function
   On -> Off
| Off -> On ;;
```

```
# #use "state.ml";;
val turn : state -> state = <fun>
# let s = Off;;
val s : state = Off
# turn s;;
- : state = On
```



References

- Davide Ancona, Giovanni Lagorio, and Elena Zucca.
  Linguaggi di Programmazione.
  Città Studi Edizioni, 2007.
- Greg Michaelson.
  An Introduction to Functional Programming through λ-Calculus.
  Addison-Wesley, 1989.
- Larry c. Paulson.
  ML for the Working Programmer.
  Cambridge University Press, 1996.