

# INTRODUCTION TO OCAML

Baishakhi Ray

Fall 2018

---

*These slides are motivated from Prof. Stephen Edwards*



## Useful Link

---

- <https://ocaml.org/learn/tutorials/>
- <http://www.cs.columbia.edu/~%20sedwards/classes/2017/4115-fall/ocaml.pdf>

# Basics: Running OCaml Code

---

```
$ocaml  
OCaml version <version>  
  
# 1+1;;  
- : int = 2
```

## Command Line

### Hello.ml

```
let hello = "Hello World!"  
let _ = print_endline hello
```

**Run it with the interpreter:**  
\$ocaml hello.ml  
Hello World!

**Compile into bytecode and run:**  
\$ocamlc -o hello hello.ml  
\$./hello Hello World!

**Compile an executable and run:**  
\$ ocamlopt -o hello hello.ml  
\$ ./hello Hello World!

## Comments: (\* .... \*)

---

---

```
(* This is a single-line comment. *)
```

```
(* This is a  
* multi-line  
* comment.  
*)
```

(\* a + b

Is a broken comment

You have to close it \*)

# Calling Functions

---

---

```
Repeated ("hello", 3) /* C code */
```

```
Repeated "hello" 3 (* OCaml code *)
```

```
Repeated ( print_string ("hello"), 3) /* C code */
```

```
Repeated (print_string "hello") 3 (* OCaml code *)
```

# Defining a Function

---

---

```
#let average a b = ( a + .b) /. 2.0;;
val average : float -> float -> float = <fun>
```

```
/*c code */
double average (double a, double b) {
    return (a+b)/2;
}
```

# Defining a Function

---

---

```
#let average a b = ( a + .b) /. 2.0;;
val average : float -> float -> float = <fun>
```

Auto inferring type

Strongly Statically Typed Language  
No implicit casting!!

```
/*c code */
double average (double a, double b) {
    return (a+b)/2;
}
```

# Basic Type

---

- **int**: 31-bit signed int (roughly +/- 1 billion) on 32-bit processors, or 63-bit signed int on 64-bit processors
  - nativeint
  - Big\_int
- **float**: IEEE double-precision floating point, equivalent to C's double
- **bool**: A boolean, written either true or false
- **char**: An 8-bit character string A string
- **unit**: Written as ()
  - similar to void

# Implicit vs. explicit casts

---

---

```
1 + 2.5 ;;
```



*Error: This expression has type float but an expression was expected of type int*

```
1 +. 2.5 ;;
```



*Error: This expression has type int but an expression was expected of type float*

```
(float_of_int i) +. f
```

# Basic Types

---

---

- let i = 42 + 17;; (\* int \*)  
print\_int i;;
- let f = 42.0 +. 18.3;; (\* float \*)  
print\_float f;;
- let g = i + f ;; (\* ERROR \*)
- let g = float\_of\_int i +. f;; (\* OK \*)
- let b = true or false;; (\* bool \*)  
print\_endline (if b then "true" else "false");;

- let c = 'a';; (\* char \*)  
print\_char c;;
- let s = "Hello " ^ "World!";; (\* string \*)  
print\_endline s;;
- let u = ();; (\* unit, like "void" in C \*)

# Standard Operators and Functions

---

---

+ - * / mod	Integer arithmetic
. + . - . * . / . **	Floating-point arithmetic
ceil floor sqrt exp log log10 cos sin tan acos asin atan	Floating-point functions
not &&	Boolean operators

# Recursive Function

---

By default the function name is not visible in its default expression

```
#let rec range a b =
  if a > b then []
  else a :: range (a+1) b;;
val range : int -> int -> int list = <fun>
```

```
#let rec fac n =
  if n < 2 then 1
  else n * fac n-1;;
```

```
#let rec fac n =
  if n < 2 then 1
  else n * fac1 n
and fac1 n = fac (n-1);;
```

Mutual Recursion

## Types derived from Function declaration

---

- $f : arg1 \rightarrow arg2 \rightarrow \dots \rightarrow argn \rightarrow rettype$
- int\_to\_char
- Repeated “hello” 3 ?
- average a b = ( a + .b) /. 2.0;;
- output\_char

# Polymorphic Functions

---

```
#let give_three x = 3;;
val give_three: 'a -> int = <fun>
```

'a : "any type you fancy"

# Automatic Type Inference: How?

---

---

```
#let average a b = ( a +. b) /. 2.0;;
val average : float -> float -> float = <fun>
```

1. checks where a and b are used? →  $(a +. b)$
2.  $+$ . function always takes two floats → a and b have to be floats
3.  $/.$  function return float → return type has to be float

# Structural vs. Physical Equality

---

- Structural equality (`=`) compares values;
- physical equality (`==`) compares pointers.
- Compare strings and floating-point numbers structurally.

<code>= &lt;&gt;</code>	Structural comparison (polymorphic)
<code>== !=</code>	Physical comparison (polymorphic)
<code>&lt; &gt; &lt;= &gt;=</code>	Comparisons (polymorphic)

## If then else

---

---

- if expr1 then expr2 else expr
- It is an expression in ocaml
- Else part is compulsory
- if 3 = 4 then 42 else 17;;
- if "a" = "a" then 42 else 17;;
- if true then 42 else "17";;
  - This expression has type string but is here used with type int

## Local “variable”

---

---

```
/*c code */  
double average (double a, double b) {  
    double sum = a+b;  
    return sum/2;  
}
```

```
#let average a b =  
  let sum = ( a +. b) in  
    sum /. 2.0;;  
val average : float -> float -> float = <fun>
```

The standard phrase  
`let name = expr1 in expr2`  
is used to define a named local expression, and name can then be used later on in the function instead of expression, till a `;;` which ends the block of code.

# Global variables

---

---

```
#let avg =
  let average a b =
    let sum = (a +. b) in
    sum /. 2.0;;
val average : float -> float -> float = <fun>
```

bind name to expression in everything that follows

**let name = expression**

Any use of **let** ..., whether at the top level (globally) or within a function, is often called a **let-binding**.

# References: real variables

---

- Create a reference : `ref 0;;`
  - A reference is created, but as we did not name it a garbage collector will immediately collect it.
- Name it: `let my_ref = ref 0;;`
  - This reference is currently storing a zero integer.
- `my_ref := 100;` ← updating the value
- printing reference: `!my_ref`

The `:=` operator is used to assign to references, and the `!` operator dereferences to get out the contents.

## References: real variables

---

---

```
# let my_ref = ref 0;;
val my_ref : int ref = {contents = 0}
# my_ref := 100;;
- : unit = ()
# !my_ref;;
- : int = 100
```

```
/*c code */
int a = 0;
int *my_ptr = &a;
*my_ptr = 100;
*my_ptr;
```

# Nested Function

---

---

```
# let read_whole_channel chan =
  let buf = Buffer.create 4096 in
  let rec loop () =
    let newline = input_line chan in
    Buffer.add_string buf newline;
    Buffer.add_char buf '\n';
    loop ()
  in try
    loop ()
  with
    End_of_file -> Buffer.contents buf;;
val read_whole_channel : in_channel -> string = <fun>
```

# Modules

---

- Ocaml has modules in /usr/lib/ocaml/
- For example, to use the functions in Graphics module, simply do:
  - **open Graphics;** at top and call its functions, say **open\_graph** later
  - or **Graphics.open\_graph " 640x480";**
  - (\* *To compile this example: ocamlc graphics.cma grtest.ml -o grtest* \*)
- *Rename module: module Gr = Graphics;;*
  - *Gr.open\_graph*

# Filenames

---

---

Purpose	C	Bytecode	Natice Code
Source Code	*.c	*.ml	*.ml
Header files	*.h	*.mli	*.mli
Object files	*.o	*.cmo	*.cmx <sup>2</sup>
Library files	*.a	*.cma	*.cmxa <sup>3</sup>

\*.cmi files are intermediate files which are compiled forms of the .mli (interface or "header file").

# Creating Modules

---

- In OCaml, every piece of code is wrapped into a module.
- A module itself can be a submodule of another module, pretty much like directories in a file system-but we don't do this very often.

```
/*amodule.ml*/  
let hello( ) = print_endline "Hello"
```

```
/*bmodule.ml*/  
Amodule.hello ()
```

# Interfaces and Signature

---

```
/*amodule.ml*/  
let message = "Hello"  
let hello( ) = print_endline message;;
```

Interface

```
val message : string  
val hello : unit -> unit
```

Interface: amodule.mli

```
val hello : unit -> unit
```

# Types

---

- modules often define new types
- type name = typedef

```
type date = { day : int; month : int; year : int }
```

- There are four options when it comes to writing the corr. .mli file:
  1. The type is completely omitted from the signature.
  2. The type definition is copy-pasted into the signature.
  3. The type is made abstract: only its name is given. (eg. type date)
  4. The record fields are made read-only: type date = private { ... }

# Abstract Types

---

- type date;
- users can manipulate objects of type date, but they can't access the record fields directly.

```
type date
val create : ?days:int -> ?months:int -> ?years:int -> unit -> date
val sub : date -> date -> date
val years : date -> float
```

- only create and sub can be used to create date records. It is not possible for the user of the module to create ill-formed records..



---

# LANGUAGE FEATURES

Data Types and Matching

---

# Linked Lists

---

- OCaml has support for lists built into the language.
- All elements of a list in OCaml must be the same type.
  - [1;2;3];
  - Empty list = []
- Head: the first element 1
- Tail: the **rest of the list** [2,3]
- Alternate ways to write list:
  - [1;2;3]
  - 1::[2;3]
  - 1::2::[3]
  - 1::2::3

# Type of Linked List

---

- The type of a linked list of integers is int list
- the type of a linked list of foos is foo list.
- ‘a list?
  - Type of anything
  - Does not mean that each individual element has a different type
- A linked list must have the same type. But the type can be polymorphic
- List.length : 'a list -> int

# Structures

---

---

```
/*c code */  
struct pair_of_ints {  
    int a, b;  
};
```

**Tuples:** Pairs or tuples of different types separated by commas.

```
(3,4)  
- : int * int = (3, 4)  
(3, "hello", 'x')  
- : int * string * char = (3, "hello", 'x')
```

**Records:** allow name of the elements

```
# type pair_of_ints = { a : int; b : int };;  
(*Use*)  
# {a =3; b=5}  
- : pair_of_ints = {a = 3; b = 5}
```

Unlike list, different types of elements are allowed

# Variants (qualified unions and enums)

---

```
/*c code */
struct foo {
    int type;
#define TYPE_INT 1
#define TYPE_PAIR_OF_INTS 2
#define TYPE_STRING 3
    union {
        int i; // If type == TYPE_INT.
        int pair[2]; // If type == TYPE_PAIR_OF_INTS.
        char *str; // If type == TYPE_STRING.
    } u;
};
```

# Variants (qualified unions)

---

```
/*c code */
struct foo {
    int type;
#define TYPE_INT 1
#define TYPE_PAIR_OF_INTS 2
#define TYPE_STRING 3
    union {
        int i; // If type == TYPE_INT.
        int pair[2]; // If type == TYPE_PAIR_OF_INTS.
        char *str; // If type == TYPE_STRING.
    } u;
};
```

```
# type foo =
| Nothing
| Int of int
| Pair of int * int
| String of string;;
```

Use:  
**Nothing**  
-: foo Nothing  
**Int 3**  
-: foo Int 3  
**Pair (4, 5)**  
-: foo Pair (4, 5)  
**String "hello"**  
-: foo String "hello"

## Variants (enums)

---

---

```
/*c code */  
enum sign { positive, zero, negative };
```

```
/*OCaml code */  
# type sign = Positive | Zero | Negative
```

## Recursive variants (used for trees)

---

```
/*OCaml code */
# type binary_tree =
    | Leaf of int
    | Tree of binary_tree * binary_tree;;
type binary_tree = Leaf of int | Tree of binary_tree * binary_tree
```

Leaf 3

Tree (Leaf 3, Leaf 4)

Tree (Tree (Leaf 3, Leaf 4), Leaf 5)

Tree (Tree (Leaf 3, Leaf 4), Tree (Tree (Leaf 3, Leaf 4), Leaf 5))

## Parameterized variants

---

---

```
/*OCaml code */
# type 'a binary_tree =
  | Leaf of 'a
  | Tree of binary_tree * binary_tree;;
type 'a binary_tree = Leaf of 'a | Tree of 'a binary_tree * 'a binary_tree
```

Leaf 3

Tree (Leaf 3, Leaf 4)

Tree (Tree (Leaf 3, Leaf 4), Leaf 5)

Tree (Tree (Leaf 3, Leaf 4), Tree (Tree (Leaf 3, Leaf 4), Leaf 5))

# Parameterized variants

---

---

```
# type 'a equiv_list =
| Nil
| Cons of 'a * 'a equiv_list;;
type 'a equiv_list = Nil | Cons of 'a * 'a equiv_list

# Nil;;
- : 'a equiv_list = Nil
# Cons(1, Nil);;
- : int equiv_list = Cons (1, Nil)
# Cons(1, Cons(2, Nil));;
- : int equiv_list = Cons (1, Cons (2, Nil))
```

# List, Structure, Variant--summary

---

OCaml name	Example type definition	Example usage
list	int list	[1; 2; 3]
tuple	int * string	(3, "hello")
record	type pair = { a: int; b: string }	{ a = 3; b = "hello" }
variant	type foo =   Int of int   Pair of int * string	Int 3
variant	type sign =   Positive   Zero   Negative	Positive Zero Negative
parameterized variant	type 'a my_list =   Empty   Cons of 'a * 'a my_list	Cons (1, Cons (2, Empty))

# Pattern Matching

---

---

$n * (x + y)$

```
# type expr =
| Plus of expr * expr          (* means  $a + b$  *)
| Minus of expr * expr         (* means  $a - b$  *)
| Times of expr * expr        (* means  $a * b$  *)
| Divide of expr * expr       (* means  $a / b$  *)
| Value of string              (* "x", "y", "n", etc. *);;
```

```
# Times (Value "n", Plus (Value "x", Value "y"));
- : expr = Times (Value "n", Plus (Value "x", Value "y"))
```

# Pattern Matching

---

$n * (x + y)$

```
# let rec to_string e =
  match e with
  | Plus (left, right) -> "(" ^ to_string left ^ " + " ^ to_string right ^ ")"
  | Minus (left, right) -> "(" ^ to_string left ^ " - " ^ to_string right ^ ")"
  | Times (left, right) -> "(" ^ to_string left ^ " * " ^ to_string right ^ ")"
  | Divide (left, right) -> "(" ^ to_string left ^ " / " ^ to_string right ^ ")"
  | Value v -> v;;
val to_string : expr -> string = <fun> # let print_expr e = print_endline (to_string e);;
val print_expr : expr -> unit = <fun>
```

```
# print_expr (Times (Value "n", Plus (Value "x", Value "y")));
(n * (x + y))
- : unit = ()
```

# Pattern Matching

---

- match value with
  - | pattern -> result
  - | pattern -> result ...

# Pattern Matching

---

$n * (x + y) \implies (n*x + n*y)$

```
# let rec multiply_out e =
  match e with
  | Times (e1, Plus (e2, e3)) ->
    Plus (Times (multiply_out e1, multiply_out e2),
          Times (multiply_out e1, multiply_out e3))
  | Times (Plus (e1, e2), e3) ->
    Plus (Times (multiply_out e1, multiply_out e3),
          Times (multiply_out e2, multiply_out e3))
  | Plus (left, right) -> Plus (multiply_out left, multiply_out right)
  | Minus (left, right) -> Minus (multiply_out left, multiply_out right)
  | Times (left, right) -> Times (multiply_out left, multiply_out right)
  | Divide (left, right) -> Divide (multiply_out left, multiply_out right)
  | Value v -> Value v;;
val multiply_out : expr -> expr = <fun>
```