CS:314 Fall 2024

Section **04**

Recitation **10**

chris.tu@rutgers.edu

Office hours: 2-3pm @ Thursday CoRE 335

Topics Covered

- OCaml Fundamentals
- OCaml Experimenting

Three ways to run OCaml code

- **`ocaml`** -- The interactive interpreter
- **`ocamlc`** -- The OCaml bytecode compiler
- `ocamlopt` -- The OCaml native code compiler (not used in this course)

`ocaml` or **`ocamlc`** are put directly in your terminal or command prompt.
You can use it in the iLab environment.

`ocaml` is used for interactive sessions where you evaluate quick expressions,
test code snippets, or experiment with Ocaml.
Typing it in the terminal will let you enter the OCaml Read Eval Print Loop.

Example OCaml REPL Session

```
$ ocaml
          OCaml Version [version#]

# let x = 10;;
val x : int = 10
```

$ - at the terminal prompt, we invoke ocaml
if OCaml is installed, the version displays.
# - the pound sign shows that the OCaml Read Eval Print Loop (REPL) is ready to accept input.
Reads your expression/code: `let x = 10;;`
Evaluates the expression.
`val x : int = 10` is outputted by the REPL to show that x has been defined as an integer with a value 10.

You can continue writing expressions without needing compile or a separate file.

Example OCaml REPL Session

```
$ ocaml
          OCaml Version [version#]

# let x = 10;;
val x : int = 10
# x + 1;;
- : int = 11
```

# - the pound sign shows that the OCaml Read Eval Print Loop (REPL) is once again ready to accept input.
variable x is bound to the integer 10. Typing an expression that adds one outputs:

```
- : int = 11
```

The minus sign is a placeholder that indicates an evaluated value that is not bound to a name. The result of the expression is 11 of type int.

Three ways to run OCaml code

1. Invoke **`ocaml`** and type in your code.

2. Write your OCaml code in a **.ml** file, then invoke **`ocaml`** and type:

   # use filename.ml

3. Invoke **`ocamlc`** on your **.ml** file.

```
$ ocamlc [filename].ml -o [executable name]
$ ./[executable name]
```

Types in OCaml

OCaml is **strongly typed**.

- Every expression has a type that is checked during compilation.

- There is no implicit casting or coercion. All type conversions are explicit. OCaml provides functions to convert between types explicitly. These are part of the standard library (no imports necessary)

- The basic types in OCaml are: **int, float, char, string, bool**

Tuples

- If a and b are **two types**, then we can build an ordered pair consisting of an expression of type a, and an expression of type b.
- Such an ordered pair is called a ***tuple***.
- The tuple itself is written as (x, y).
- The type of the tuple is written as a * b.

Tuples

OCaml REPL:

```
# let pair = (“hawk”, 21);;
val pair : string * int = (“hawk”, 21)
```

`pair` is a tuple consisting of two components:

- "hawk" of type string
- “21” of type int
- The tuple type is <u>int * string.</u>

Lists

- In OCaml you can only build **homogeneous** lists,
  - i.e. all elements in the list must have the same type.
- If a is a type, then a list consisting of elements of type a has type a list.
- The list itself is written as [x; y; z; …].
- list in OCaml is an example of **parameterized variant**.
- You will occasionally see notations like 'a list.
  - Here 'a is a placeholder representing an arbitrary type.
  - this is similar to generic types in Java like <T> for type in generic interfaces like List<T>

List Constructor

- If e1 has type a, and e2 has type a list, then e1::e2 is a list with type a list.
  - :: prepends element e1 to list e2.
  - :: is pronounced cons
- This is analogous to `(cons e1 e2)` in Scheme.
- [] (the empty list) has type 'a list, hence x::[] is always valid.

If expression

- If p has type bool, both x, y have type a, then

  if p then x else y

  is an expression with type a.

- **In OCaml, you cannot omit the else part!**
- **All branches of an if expression must have the same type, because OCaml is a statically typed language and every expression must evaluate to a single, well-defined type.**

Pattern Match

- Functional languages typically use **algebraic data types**.
- The identity of a value is solely determined by the way it is constructed.
- Pattern match gives us information about how a value is constructed.

Pattern Match

- There are only two ways to construct a list, namely the empty list [] and the list constructor ::.
  - `# let empty_list = [];;`
  - `# let my_list = 1 :: 2 :: 3 :: [];;`
    `(* creates a list [1; 2; 3] *)`
- Pattern match allows us to deconstruct a list:

```
match my_list with
 | e1::e2 -> [do something with e1, e2]
 | [] -> [return something]
```

- OCaml will issue a warning if your pattern match does not exhaust all cases (e1::e2) or ([])

Pattern Match Simple Integer Example

```
let describe_number n =
  match n with
  | 0 -> "Zero"
  | 1 -> "One"
  | _ -> "Other";;
```

n is the expression being matched.
0, 1, and _ are patterns.
_ (underscore) is a wildcard pattern that matches anything not explicitly covered by other cases.

Pattern Match Simple Integer Example

```
# let describe_number n =
      match n with
      | 0 -> "Zero"
      | 1 -> "One"
      | _ -> "Other";;

val describe_number : int -> string = <fun>
```

```
# let result = describe_number 1;;
val result : string = "One"
# let result = describe_number 5;;
val result : string = "Other"
```

fun is a keyword for anonymous function

Exhaustive Cases

Consider the following variant type:

```
type color =
  | Red
  | Green
  | Blue
;;
```

Exhaustive Pattern Matching:

```
let describe_color c =
  match c with
  | Red -> "It's red"
  | Green -> "It's green"
  | Blue -> "It's blue"
;;
```

- **OCaml sees that all constructors (Red, Green, Blue) are covered, so the match is exhaustive.**
- **No warnings are issued.**

let expressions

- There are two kinds of let expressions: global and local let.
- Global let defines a symbol in the global scope:
  `let x = 1;;`
- Local let defines a symbol in the scope of a particular expression:
  `let x = 1 in (x + 3);;`
- Local let must have the in keyword, whereas global let never use the in keyword!

Functions

- let can be used to define functions:
- let func x = x + 3;;
- You must use let rec instead of let when defining recursive functions:
- let rec func x = if x <= 1 then 1 else x * func (x - 1);;

Lambda Functions

- Lambda functions are introduced by the `fun` keyword:

```
fun x -> x + 1
```

- Lambda functions can also have multiple arguments:

```
fun x y -> x + y
```

OCaml Resources

- **Online OCaml Interpreter** (Similar to Toplevel) Interactive Interpreter: https://try.ocamlpro.com/
- **Official manual:** https://caml.inria.fr/pub/docs/manual-ocaml/
- **Official tutorial:** https://ocaml.org/learn/tutorials/
- **99 Problems in OCaml:** https://ocaml.org/learn/tutorials/99problems.html
- **Rosetta Stone (lots of example code):** https://rosettacode.org/wiki/Category:OCaml

Experiment

Try Putting all of this sections code lines in the online interpreter or your iLab OCaml environment.

Test around and see what you can learn about how OCaml handles expressions and types.

Your next Project will most likely cover OCaml.

```
# print_endline "Hello, World!";;
```

Experiment

```
# print_endline "Hello";;
Hello
- : unit = ()
```

The **- : unit = () part** is the OCaml interpreter's way of showing the type and value of the expression.

Here, it indicates that the expression evaluates to the **unit type** (), which is similar to void in languages like C and Java.

The unit type is used in situations where an expression or function doesn't need to return a meaningful value.

Experiment

- # let sum = 1 + 2 in
- print_int sum;;
- 3- : unit = ()

- OCaml's interpreter shows that the last expression (print_int sum) returns a value of type unit, which is denoted by ().
    - This is typical for functions like print_int that are used for their side effects (in this case, output to the console) rather than their return values.

Experiment

- # let add x y = x + y in
- print_int (add 5 7);;
- 12- : unit = ()

Experiment

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

- 'a list means it's a list that could contain elements of any type, where 'a is a type variable representing an unspecified type.
- = [] shows the value of the expression, confirming that it's indeed an empty list.

Experiment

List

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

- [1+1;2+2;3+3;4+4]
- ["a";"b"; "c"^"d"]
- [[1]; [2;3]; [4;5;6]]

- # [1,'a',2];;
- - : (int * char * int) list = [(1, 'a', 2)]
- # [(1, 'a', 2)];;
- - : (int * char * int) list = [(1, 'a', 2)]

Experiment

List

- [1;2]@[3;4;5]

- hd [1;2] → 1
- hd(["a"]@["b"]) → "a"

- tl [1;2;3] → [2;3]
- tl (["a"]@["b"]) → ["b"]

Experiment

Tuple

- Tuples can be **heterogeneous**
    - Unlike lists, which must be **homogenous**

(1, 2)

(1, "string", 3.5)

(1, ["a"; "b"], 'c')

Remember, brackets are used for lists

[(1,2)]

[(1, 2); (3, 4)] - tuples of tuples

[(1,2); (1,2,3)] - try this and see what happens.

Experiment

Recursive Function

```
# let rec factorial n =
if n <= 1 then 1 else n * factorial (n - 1) in
print_int (factorial 5);;
```