Functional Programming with OCaml

Programming Abstractions
Department of Computer Science,
Stony Brook University
Dr. Ritwik Banerjee

OCaml

- The main implementation of the Caml programming language, formerly known as Objective Caml.
  - Relatively recent language, developed in 1996.
  - Extends the original Caml language with object-oriented features.
  - A **strong, statically typed** language.
  - A powerful language that has inspired other languages like Scala and F#.
  - A **functional language**.
  - Official website: https://ocaml.org/ has links to documentation, cheat sheets, tutorials, code examples, etc.

Installing and Running OCaml

- Install by following instructions available here:

  http://www.ocaml.org/docs/install.html ← may take a while!

- Online OCaml interactive shell: https://try.ocamlpro.com/
- File names have the **.ml** extension.
- Running the read-eval-print loop, or REPL:

```
$ ocaml
        OCaml version 4.02.3
# print_endline "Hello World!";; (* ;; ends an expression *)
Hello World!
- : unit = ()
# #use "hello.ml";; (* import a file, and explain in a comment *)
Hello World!
- : unit = ()
# (* #quit terminates the interactive shell, so does 'exit 0;;'
     or ctrl+D *)
  #quit;;
```

Installing and Running OCaml

From a **.ml** file*:

- Create “hello.ml”: **print_endline “Hello World!”**
- Run it with the interpreter:

```
$ ocaml hello.ml
Hello World!
```

- Compile a native executable and run:

```
$ ocamlopt –o hello hello.ml
$ ./hello
Hello World!
```

- Use ocamlbuild:

```
$ ocamlbuild hello.native
$ ./hello.native
Hello World!
```

You can also *feed* the file into the interactive shell (this is called a **batch interactive session**, where the input commands are taken from the file:

```
$ ocaml < hello.ml
    OCaml version 4.02.3

# hello world
- : unit = ()
```

* You may come across mentions of files with **.mli** extension. These are interface files, which we may cover later in the syllabus if time permits.

Code compilation

- OCaml offers two ways of compiling and running code:
  1) bytecode compilation using **`ocamlc`** and run by the interpreter **`ocamlrun`**
  2) native code compilation using the compiler **`ocamlopt`**
- In this course, we will use the native code compilation because it
  - is more 'standard'
  - provides standalone executables
  - is faster than bytecode compilation

- The '**`-c`**' option is used to compile individual modules. It produces compiled files (with the **`.cmx`** extension) but no executable.

```
ocamlopt -c myfile.ml
```

- The '**`-o`**' option is used to specify the name of the output file produced by the linking of the compiled codes.

```
ocamlopt -o program myfile.cmx
```

  which will produce the executable named **`program`** or **`program.exe`**.
- In this course, we will not go too far into OCaml, so you should not worry about the linking of multiple modules, interfaces, etc.

Basic types and expressions

```
# 42 + 17;;
- : int = 59
# 42.0 + 17.5;;
Error: This expression has type float but an expression was expected of type
         int
# 42.0 +. 17.5;;
- : float = 59.5
# 42 + 17.5;;
Error: This expression has type float but an expression was expected of type
         int
# 42 + int_of_float 17.5;;
- : int = 59
# true || (3 > 4) && not false;;
- : bool = true
# "hello " ^ "world";;
- : string = "hello world"
# String.contains "hello" 'o';;
- : bool = true
# ();;
- : unit = ()
# print_endline "hello world";;
hello world
- : unit = ()
```

- floating-point operators (+.) must be explicit
- type conversions must be explicit (int_of_float)
- the **`unit`** type is similar to **`void`** in Java.

Basic operations 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
`= <>`	structural comparison (polymorphic)
`== !=`	physical comparison (polymorphic)
`< > <= >=`	comparisons (polymorphic)

Equality and comparisons

Physical equality compares pointers

```
# 1 == 3;;
- : bool = false
# 1 == 1;;
- : bool = true
# 1.5 == 1.5;;
- : bool = false
# let d = 1.5 in d == d;;
- : bool = true
# "hello" == "hello";;
- : bool = false
# let s = "hello" in s == s;;
- : bool = true
```

Structural equality compares values

```
# 1 = 3;;
- : bool = false
# 1 = 1;;
- : bool = true
# 1.5 = 1.5;;
- : bool = true
# let d = 1.5 in d = d;;
- : bool = true
# "hello" = "hello";;
- : bool = true
```

TLDR: *do NOT use the physical equality* **==** *in your programs.*

Conditional expressions

`if expr₁ then expr₂ else expr₃`

- The "else" part is mandatory.
- The type of $\texttt{expr}_1$ must be boolean.
- The types of $\texttt{expr}_2$ and $\texttt{expr}_3$ must match.

```
# if 2 = 3 then "hello" else "bye";;
- : string = "bye"
# if 2 = 3 then 2 else "bye";;
Error: This expression has type string but an expression was expected of type
         int
```

Binding names using “let”

- We can say **let name = expr**$_1$ **in expr**$_2$, which binds the name to the 1st expression within the scope of the 2nd expression only.
- We can also say **let name = expr**, which binds the name to the expression for the rest of the execution.

```
# let x = 12 in x+4;;
- : int = 16
# let x = (let y=2 in 3*y)*10 in x;;
- : int = 60
# x + 3;;
Error: Unbound value x
# let x = (let y=2) in x;;
Error: Syntax error: operator expected.
# let x = 1 in let y=x;;
Error: Syntax error
```

Binding names using “let”

- The ‘**`let`**’ keyword can be used to bind in *succession*, since each binding is local.
- But always remember that this is **not an assignment**! The name disappears as soon as the scope ends.

```
# let x = 4 in
  let x = 2*x in
  let x = x+3 in
  x;;
- : int = 11
# x;;
Error: Unbound value x
```

Binding names using “let”

- Again, ‘**`let`’ is not an assignment**!
- OCaml picks up bindings that were in effect at the time of function definition. This is very different from using a name as a variable bound to a value.
- In this example, it sees `a = 2` here, and then again here.
- Defining the function again after **`let`**ting `a = 20` changes the behavior.

```
# let a = 2;;
val a : int = 2
# let add_a x = x + a;;
val add_a : int -> int = <fun>
# add_a 3;;
- : int = 5
# let a = 20;;
val a : int = 20
# add_a 3;;
- : int = 5
# let add_a x = x + a;;
val add_a : int -> int = <fun>
# add_a 3;;
- : int = 23
```

Functions

- Functions are *first-class* objects in OCaml.
- We define a function just like any other value can be defined.
- We can apply a function to its argument(s) immediately.
- Functions with multiple parameters are modeled as nested functions, each with a single parameter.

```
# fun x -> x * x;;
- : int -> int = <fun>
# (fun x -> x * x) 10;; (* apply the function to the argument 10 *)
- : int = 100
# fun x -> (fun y -> x * y);;
- : int -> int -> int = <fun>
# fun x y -> x * y;; (* shorthand for the previous definition *)
- : int -> int -> int = <fun>
# (fun x -> (fun y -> (x + 1) * y)) 3 7;;
- : int = 28
# let square = fun x -> x * x;;
val square : int -> int = <fun>
# square 5;;
- : int = 25
# let square x = x * x;; (* shorthand for the previous definition *)
val square : int -> int = <fun>
# square 11;;
- : int = 121
```

"let" is like a function application

- The expression **`let name = expr`**$_1$ **`in expr`**$_2$ is semantically equivalent to the expression **`(fun name -> expr`**$_2$**`) in expr`**$_1$.

- Both mean "in **`expr`**$_2$, replace *name* with the **`expr`**$_1$".

- But **`let`** is usually easier to read:

```
# let a = 3 in a + 2;;
- : int = 5
# (fun a -> a + 2) 3;;
- : int = 5
```

Currying

- Did you notice the type of the multiplication function?

```
fun x y -> x*y;;
```

- Ordinarily, we would expect the type to be **`(int, int) -> int`**
- But recall from lambda-calculus that a function can only take a single argument. So we have the following expansion:
  - **`fun`** takes the argument **`x`** and returns a function (let's call it **`fun_x`**).
  - **`fun_x`** then takes the argument **`y`** and multiplies **`y`** with **`x`**, yielding the final result.
- Thus, the type of fun is given by **`int -> int -> int = <fun>`**. We may read this as **`int -> (int -> int)`**. That is, a function that takes an **`int`**, and returns a function of the type **`int -> int`**.
- This type of evaluation of a function that takes multiple arguments as evaluating a sequence of functions, each with a single argument, is called **Currying** (named after the mathematician Haskell Curry).

Recursive functions

OCaml

```
(* let allows for recursion *)
let rec gcd a b =
  if a = b then
    a
  else if a > b then
    gcd (a-b) b
  else
    gcd a (b-a)
```

Java

```
int gcd(int a, int b) {
    while (a != b) {
        if (a > b)
            a -= b;
        else
            b -= a;
        }
    return a;
}
```

Recursive functions

```
# let fac n = if n < 2 then 1 else n * fac (n-1);;
Error: Unbound value fac
# let rec fac n = if n < 2 then 1 else n * fachelp n
  and fachelp n = fac (n-1);;
val fac : int -> int = <fun>
val fachelp : int -> int = <fun>
# fac 7;;
- : int = 5040
```

- By default, a name is not visible in the expression it is getting bound to (i.e., its own definition).
- The **`rec`** keyword allows for such self-referential visibility.
- The **`and`** keyword allows for mutual recursion.

```
# let app_add = fun f -> (f 42) + 17;;
val app_add : (int -> int) -> int = <fun>
# let plus_five x = x + 5;;
val plus_five : int -> int = <fun>
# app_add plus_five;;
- : int = 64
```

First-class functions

Name them and pass them around as arguments.

```
# let make_incr i = fun x -> x + i;;
val make_incr : int -> int -> int = <fun>
# let incr_five = make_incr 5;;
val incr_five : int -> int = <fun>
# incr_five 12;;
- : int = 17
```

Higher-order functions

Functions can return functions.

Data Types

<table>
<tr><th>Type</th><th>Constants</th><th>Operations</th></tr>
<tr><td>unit</td><td><code>()</code></td><td>no operation!</td></tr>
<tr><td>bool</td><td><code>true false</code></td><td><code>&& || not</code></td></tr>
<tr><td>char</td><td><code>'a' '\n' '\097'</code></td><td><code>Char.code Char.chr</code></td></tr>
<tr><td>int</td><td>1 2 3</td><td><code>+ - * / max_int</code></td></tr>
<tr><td>float</td><td>1.0 2. 3.14 6e23</td><td><code>+. -. *. /. cos</code></td></tr>
<tr><td>string</td><td><code>"a\tb\010c\n"</code></td><td><code>^ s.[i] s.[i] <- c</code></td></tr>
<tr><td colspan="3"></td></tr>
<tr><td colspan="3">Polymorphic types and operations</td></tr>
<tr><td>arrays</td><td><code>[| 0; 1; 2; 3 |]</code></td><td><code>t.(i) t.(i) <- v</code></td></tr>
<tr><td>pairs</td><td>(1, 2)</td><td><code>fst snd</code></td></tr>
<tr><td>tuples</td><td>(1, 2, 3, 4)</td><td>Use pattern matching!</td></tr>
</table>

The `unit` type and its `()` value

- The `unit` value `()` of type `unit` conveys no information: it is the unique value of its type.
- It serves the purpose of `void` (e.g., in in Java and C).
- No operations are possible on this type or its only value.

Tuples

```
# (42, “John”);;
- : int * string = (42, “John”)
# (42, “John”, “Doe”);;
- : int * string * string = (42, “John”, “Doe”)
# let p = (42, “John”);;
val p : int * string = (42, “John”)
# fst p;;
- : int = 42
# snd p;;
- : string = “John”
# let profile = (42, “John”, “Doe”, 177, “male”);;
val profile : int * string * string * int * string =
  (42, “John”, “Doe”, 177, “male”)
# let (_, _, lastname, _, sex) = profile in
(lastname, sex);;
- : string * string = (“Doe”, “male”)
```

Lists

```
# [];; (* empty list, also called 'nil' *)
- : 'a list = []
# [1];; (* singleton list *)
- : int list = [1]
# [ [[]]; [[1;2;3]]];; (* nested list *)
- : int list list list = [[[]]; [[1; 2; 3]]]

# 7 :: [6; 5];; (* cons, ::, puts an element at the
start *)
- : int list [7; 6; 5]
# [1; 2] :: [3; 4];;
Error: This expression has type int but an expression
       was expected of type int list

# [1; 2] @ [3; 4];; (* @ concatenated two lists *)
- : int list = [1; 2; 3; 4]

(* get the first item of a list *)
# List.hd [1; 2; 3];;
- : int 1

(* get everything after the first item of a list *)
# List.tl [1; 2; 3];;
- : int list [2; 3]
```

A note on the syntax for list & tuple

- The square brackets are convenient but not required for lists.
  The list `[1; 2; 3]` is equivalent to `1::2::3::[]`.

- A list cannot contain a mixture of elements of different types:

```
# [[1; 2]; [[1; 2]]];;
Error: This expression has type 'a list
           but an expression was expected of type int
# [[1; 2]; [2; 3]];;
- : int list list = [[1; 2]; [2; 3]]
```

- Separating by the semi-colon creates a list. Separating by comma creates a tuple. The parentheses are optional, but convenient.

```
# ["1", 1, [1; 2]];;
- : (string * int * int list) list = [("1", 1, [1; 2])]
```

(Some) List functions

In Ocaml, lists are immutable. We cannot change an element of a list from one value to another. So, OCaml programmers create new lists out of old lists.

Using **map** and **reduce** operations, we can largely replace traditional loops:

- Apply a function `f` to each element of a list to transform it:
  ```
  List.map f [a1; ...; an] = [f a1; ...; f an]
  ```
- Apply a function `f` recursively to the *current* result together with an element of the list, to finally produce a single element:
  ```
  List.fold_left f a [b1; ...; bn] = f(...(f(f a b1) b2)...) bn)
  ```
- Apply a function `f` to each element of a list, to produce **`unit`** as the result:
  ```
  List.iter f [a1; ...; an] = begin f a1; ... ; f an; () end
  ```
- Reverse a list:
  ```
  List.rev [a1; ...; an] = [an; ...; a1]
  ```

```
# List.map (fun a -> a * a) [1;2;3;4;5];;
- : int list = [1; 4; 9; 16; 25]
# List.map string_of_int [1;2;3;4;5];;
- : string list = ["1"; "2"; "3"; "4"; "5"]
# List.fold_left (fun a b -> a + b) 0 [1;2;3;4;5];;
- : int = 15
# List.iter print_int [1;2;3];;
123- : unit = ()
# List.iter (fun i -> print_int i; print_newline ()) [1;3;11];;
1
3
11
- : unit = ()
```

(Some) List functions

Enumerating list items

- To transform a list and pass information from one element to another, we can use **`fold_left`** with a tuple:

```
# let (l, _) = List.fold_left (fun (l, n) e -> ((e, n) :: l, n+1))
([], 0) [1; 2; 3; 4] in List.rev l;;
- : (int * int) list = [(1, 0); (2, 1); (3, 2); (4, 3)]
```

- Here, the function **`fun`**
  - takes a tuple and an element, and
  - returns a tuple,
  - where the first element is a list of pairs, and the second element is an integer.
- If you apply such a function to a list, and use an initial value of `0` in the **accumulator** (i.e., the starting value of **`n`**),
  - the list gets transformed into pairs where each list element is paired with its index.
  - but the accumulation using `cons` ends up inverting the order, so we use **`List.rev`**.

Pattern Matching

- To access the various components of a data structure, OCaml uses a powerful feature called **pattern matching**, which divides the process into various *branches* or *cases*.

- For example, to compute the sum of integers in a list:

```
# let rec sum lst =
    match lst with
    | [] -> 0
    | h::t -> h + sum t;;
val sum : int list -> int = <fun>
```

- We are using **h** and **t** for 'head' and 'tail', respectively.

- Another common idiom in OCaml is to use **x** and **xs** (to think of the first element as a variable 'x', and the rest of the list as multiple such 'xs'):

```
| x::xs -> x + sum xs;;
```

Semantics of pattern matching

- Pattern matching involves two tasks. These are to determine
  1. whether or not a pattern matches a value, and
  2. what parts of the value are to be bound to which variable names in the pattern.

- The 2nd task (i.e., bindings) needs a more careful analysis. The list is:
  1) The pattern **`x`** matches a value **`v`**, and binds **`x -> v`**.
  2) The pattern _ matches a value **`v`**, and does not bind to anything.
  3) The **nil** pattern **`[]`** only matches the **nil** value **`[]`**, and does not bind to anything.
  4) If a pattern $\mathbf{p_1}$ matches a value $\mathbf{v_1}$, and produces a set of bindings $\mathbf{b_1}$, and a pattern $\mathbf{p_2}$ matches a value $\mathbf{v_2}$, and produces a set of bindings $\mathbf{b_2}$, then $\mathbf{p_1::p_2}$ matches $\mathbf{v_1::v_2}$. The set of bindings is the union $\mathbf{b_1} \cup \mathbf{b_2}$.
  5) Similarly for lists, generalizing the above notation, the pattern **`[p1; ...; pk]`** matches **`[v1; ...; vk]`**, and produces the bindings $\bigcup_{i=1}^{k} \mathbf{b}_i$.

- Based on the above list, we can evaluate general pattern matching expressions of the type **`match e with`** $\mathbf{p_1}$ **`->`** $\mathbf{e_1}$ **`| ... |`** $\mathbf{p_n}$ **`->`** $\mathbf{e_n}$**`;;`**

Example of bindings in a function definition that uses pattern matching

```
# let xor p =
  match p with
   | (false, false) -> false
   | (false,  true) -> true
   | ( true, false) -> true
   | ( true,  true) -> false;;
val xor : bool * bool -> bool = <fun>

# xor (true, true);;
- : bool = false
```

- Note that
  - in the first two cases, the output is the same as the second item in the pair, and
  - in the last cases, the output is the negation of the second item in the pair.
- We can use variable binding here:

```
# let xor p =
  match p with
   | (false, x) -> x
   | ( true, x) -> not x;;
```

Wildcards

- The underscore symbol is a **wildcard** that matches anything.
- This is often used when there is a part of the pattern we must account for, but we don't care about that part. For example:

```
# let xor p =
    match p with
    | (true, false) -> true
    | (false, true) -> true
    | _ -> false;;
val xor : bool * bool -> bool = <fun>
```

Wildcards

- The underscore symbol is a **wildcard** that matches anything.
- This is often used when there is a part of the pattern we must account for, but we don't care about that part. For example:

```
let length = function
     []          -> "empty"
   | [_]         -> "singleton"
   | [_; _]      -> "pair"
   | [_; _; _]   -> "triple"
   | hd :: tl    -> "many";;
val length : 'a list -> string = <fun>
```

A note on function syntax

- By now, you have probably noticed that there are two different ways we can define a function in OCaml:
  1. **`let {`*`function_name`*`} {`*`args`*`} = match ... with {`*`cases`*`}`**
  2. **`let {`*`function_name`*`} = function {`*`cases`*`}`**
- The two syntaxes are semantically identical (they produce the same assembly code after compilation).
  - For example, the two definitions shown here are identical.

```
let rec fib = function
  | 0 -> 0
  | 1 -> 1
  | i -> fib (i-1) + fib (i-2);;
```

```
let rec fib n = match n with
  | 0 -> 0
  | 1 -> 1
  | i -> fib (i-1) + fib (i-2);;
```

Semantics of pattern matching: static and dynamic

```
# let xor = function
    | (false, x) -> x
    | (x,  true) -> not x;;
Warning: this pattern-matching is not
exhaustive.
Here is an example of a value that is
not matched: (true, false)
val xor : bool * bool -> bool = <fun>
```

```
# let xor = function
  | (false, x) -> x
  | (true, x) -> not x
  | (false, false) -> false;;
Warning: this match case is unused.
val xor : bool * bool -> bool = <fun>
```

- The semantics we have described so far are **dynamic semantics**. That is, every name is associated with a value at runtime.
- But OCaml also performs some **static semantic** checks:
  1. **Type inference**: the type of pattern matching expression must be valid.
  2. **Exhaustiveness**: the compiler checks whether or not the cases guarantee that at least one of the cases will match any valid input expression.
  3. **Unused cases**: the compiler checks if there is any redundancy (i.e., a case is not needed due to the previous cases already being exhaustive).

Tail Recursion

- A recursive function is called **tail recursive** if does not perform any computation after the recursive call returns, and immediately returns to its caller the value of its recursive call.
- Let us look at two implementations of the function to compute the sum of all the integers in a list:

```
(* not tail recursive *)
let rec sum lst = match lst with
  | [ ] -> 0
  | x :: xs -> x + sum xs;;
val sum : int list -> int = <fun>
```

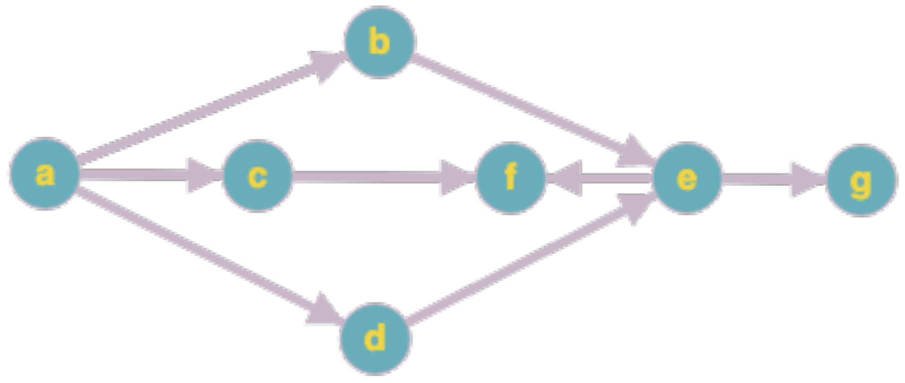
```
(* tail recursive *)
let rec acc s lst = match lst with
  | [ ] -> s
  | x :: xs -> acc (s+x) xs;;
val acc : int -> int list -> int = <fun>

let sum lst = acc 0 lst;;
val sum : int list -> int = <fun>
```

Tail Call Optimization

- Recursion leads to a call stack, where each element corresponds to a function call that has started but not yet returned.
- If the recursion is a tail recursion, the caller does absolutely nothing more than just "pass along" the result, once the caller receives it.
  - We can perhaps imagine it as follows: once the recursion stops, the entire stack just 'collapses' quickly, since the final value is just being passed down from the top of the stack, until the bottom stack frame is popped and the same value that was passed down all the way, is returned.
  - But if the value doesn't change upon return from a recursive call, is there a point in maintaining the full stack?
- Many languages thus optimize for tail recursion – instead of growing the call stack, the caller's stack frame is simply replaced with that of the called function.
- This is called **tail call optimization**, and it reduces the space complexity of tail-recursive algorithms from $O(n)$ to $O(1)$.

```
let edges = [("a", "b"); ("a","c"); ("a","d"); ("b","e");
             ("c","f"); ("d","e"); ("e","f"); ("e","g")];;
let rec successors v = function
  | []              -> []
  | (s,t)::edges -> if s = v
                    then t::successors v edges
                    else successors v edges
val successors : 'a -> ('a * 'b) list -> 'b list = <fun>
# successors "a" edges;;
- : string list = ["b"; "c"; "d"]
# successors "b" edges;;
- : string list = ["e"]
```

Example: depth-first search



1) In the pattern matching code, we are *searching* through a list of tuples, and checking if the first item is equal to the input vertex **v**.

- In other words, we are *filtering* the list based on this condition.

2) From those filtered edges, we are collecting the second item of each tuple. So, a functional way of implementing this would be:

```
let successors v edges =
    let matching (s,_) = (s = v) in
    List.map snd (List.filter matching edges);;
```

Example: depth-first search

```
let rec successors v = function
  | []            -> []
  | (s,t)::edges -> if s = v
                     then t::successors v edges
                     else successors v edges
```

```
let rec dfs edges visited = function
  | []          -> List.rev visited
  | v::vertices -> if   List.mem v visited
                   then dfs edges visited vertices
                   else dfs edges (v::visited)
                              ((successors v edges) @ vertices);;
```

Example: depth-first search

```
# dfs edges [] ["d"];;
- : string list = ["d";"e";"f";"g"]
# dfs edges [] ["a"];;
- : string list = ["a";"b";"e";"f";"g";"c";"d"]
```