

EC3204: Programming Languages and Compilers

Functional Programming in OCaml

Sunbeom So

Fall 2024

Functional Programming?

- **Functions are “First-class”:** functions are ordinary values.
 - ▶ can be stored in a variable.
 - ▶ can be passed as arguments of a function.
 - ▶ can be return values of other functions.
- **Expression-oriented:** computational steps are described by “expressions” without changing states (i.e., no side effects).
 - ▶ In imperative programming, computational steps are described by commands that change memory states.

```
int factorial (int n) {  
    int i; int r = 1;  
    for (i=0; i<n; i++)  
        r = r * i;  
    return r;  
}
```

```
vs.      let rec fact n =  
          if n = 0 then 1  
          else n * fact (n-1)
```

Why Functional Programming?

- Using functional languages, you can write programs more concisely.
 - ▶ Recursions, pattern matching, etc.
- Functional programming, as opposed to imperative programming, provides a new and important frame of thinking.
 - ▶ Basic knowledge that every SW major should have!
- Functional languages are increasingly widely used in industry and academia.



Jane Street



Microsoft

cf) Isn't Recursion Expensive?

- In C and Java, we are encouraged to avoid recursion because function calls consume additional memory.

```
/* segmentation fault (due to stack overflow) */  
void f() { f(); }
```

- This is not true in functional languages. The same program in ML iterates forever.

```
let rec f () = f ()
```

- Interpreters and compilers of functional languages like ML, Scheme, Scala, and Haskell perform **tail-call optimization**.¹

¹<https://peerdh.com/uk/blogs/programming-insights/implementing-tail-call-optimization-in-ocaml>

cf) Tail-Call Optimization

- Non-tail-recursive factorial:

```
let rec fact a =  
  if a = 1 then 1 else a * fact (a - 1)
```

- Tail-recursive version that requires no additional computation and memory:

```
/* initialize 'acc' with 1 when invoking 'fact' */  
let rec fact n acc =  
  if n = 0 then acc else fact (n - 1) (n * acc)
```

Why Functional Programming in OCaml?

OCaml, a French Dialect of ML, is a good programming language.

- functional programming: scala, java8, haskell, python, JavaScript, etc.
- static and sound type system: scala, haskell, etc.
 - ▶ “static”: Type checking is done at compile-time.
 - ▶ “sound”: Your program will never produce type errors.
- automatic type inference: scala, haskell, etc.
 - ▶ You do not need to manually annotate types.
- pattern matching: scala, etc.
 - ▶ You can do case-analysis concisely.
- algebraic data types, module system, etc

²<https://github.com/facebook/infer>

³<https://github.com/kupl/VeriSmart-public>

Why Functional Programming in OCaml?

OCaml, a French Dialect of ML, is a good programming language.

- functional programming: scala, java8, haskell, python, JavaScript, etc.
- static and sound type system: scala, haskell, etc.
 - ▶ “static”: Type checking is done at compile-time.
 - ▶ “sound”: Your program will never produce type errors.
- automatic type inference: scala, haskell, etc.
 - ▶ You do not need to manually annotate types.
- pattern matching: scala, etc.
 - ▶ You can do case-analysis concisely.
- algebraic data types, module system, etc

Examples of projects implemented in OCaml:

- INFER (Facebook): a tool for detecting memory leaks in Java, C++, Object-C, and C.²
- VERISmart: a tool for verifying safety of smart contracts.³

²<https://github.com/facebook/infer>

³<https://github.com/kupl/VeriSmart-public>

Topics to Cover

- Expressions
- Names
- Functions
- Pattern matching
- Type inference
- Tuples and lists
- Variants (union data types)
- Exceptions
- Modules

Try all examples by yourself!

Basic Structure of OCaml Programs

An OCaml program is a sequence of definitions of expressions:

```
let  $x_1 = e_1$   
let  $x_2 = e_2$   
   $\vdots$   
let  $x_n = e_n$ 
```

- e_1, e_2, \dots, e_n are evaluated in order.
- Variable x_i refers to the value of e_i .

Example

- Hello World

```
let hello = "Hello"  
let world = "World"  
let helloworld = hello ^ " " ^ world  
let _ = print_endline helloworld
```

- Interpreter

```
$ ocaml helloworld.ml  
Hello World
```

- REPL (Read-Eval-Print-Loop)

```
$ ocaml  
OCaml version 5.1.1  
  
# let hello = "Hello";;  
val hello : string = "Hello"  
# let world = "World";;  
val world : string = "World"  
# let helloworld = hello ^ " " ^ world;;  
val helloworld : string = "Hello World"
```

Arithmetic Expressions

- Arithmetic expressions evaluate to numbers: e.g., $1+2*3$, $1+5$, 7
- Evaluate expressions in the REPL:

```
# 1+2*3;;  
- : int = 7
```

- Arithmetic operators on integers:

$a + b$	addition
$a - b$	subtraction
$a * b$	multiplication
a / b	divide a by b , returning the whole part
$a \bmod b$	divide a by b , returning the remaining part

Boolean Expressions

- Boolean expressions evaluate to boolean values (true, false).
- Evaluate boolean expressions.

```
# true;;
```

```
- : bool = true
```

```
# false;;
```

```
- : bool = false
```

```
# 1 > 2;;
```

```
- : bool = false
```

- Comparison operators produces boolean values:

$a = b$ true if a and b are equal

$a \neq b$ true if a and b are not equal

$a < b$ true if a is less than b

$a \leq b$ true if a is less than or equal to b

$a > b$ true if a is greater than b

$a \geq b$ true if a is greater than or equal to b

Boolean Operators

- Boolean expressions are combined by boolean operators:

```
# true && false;;
```

```
- : bool = false
```

```
# true || false;;
```

```
- : bool = true
```

```
# (2 > 1) && (3 > 2);;
```

```
- : bool = true
```

ML is a Statically Typed Language

If you try to evaluate an expression that does not make sense, OCaml rejects and does not evaluate the program. For example,

```
# 1 + true;;
```

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

Static Types and Dynamic Types

Programming languages are classified into:

- **Statically typed languages:** type checking is done at compile-time.
 - ▶ type errors are detected before program executions
 - ▶ C, C++, Java, ML, Scala, etc
- **Dynamically typed languages:** type checking is done at run-time.
 - ▶ type errors are detected during program executions
 - ▶ Python, JavaScript, Ruby, Lisp, etc

Static Types and Dynamic Types

Programming languages are classified into:

- **Statically typed languages:** type checking is done at compile-time.
 - ▶ type errors are detected before program executions
 - ▶ C, C++, Java, ML, Scala, etc
- **Dynamically typed languages:** type checking is done at run-time.
 - ▶ type errors are detected during program executions
 - ▶ Python, JavaScript, Ruby, Lisp, etc

Statically typed languages are further classified into:

- **Type-safe languages** guarantee that compiled programs do not have type errors at run-time.
 - ▶ All type errors are detected at compile time.
 - ▶ Compiled programs do not stuck.
 - ▶ ML, Haskell, Scala
- **Unsafe languages** do not provide such a guarantee.
 - ▶ Some type errors remain at run-time.
 - ▶ C, C++

cf) Which one is Better?

Statically typed languages:

- (+) Type errors are caught early in the development cycle.
- (+) Program execution is efficient by omitting runtime checks.
- (−) Less flexible than dynamic languages.

Dynamically typed languages:

- (−) Type errors appear at run-time, often unexpectedly.
- (+) Easy and fast prototyping.
- (+) Provide more flexible language features.

Conversion between Different Types

- In OCaml, different types of values are distinguished.

```
# 3 + 2.0;;
```

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

- Types must be explicitly converted.

```
# 3 + int_of_float 2.0;;
```

```
- : int = 5
```

- Operators for floating point numbers.

```
# 1.2 +. 2.3;;
```

```
- : float = 3.5
```

```
# 1.5 *. 2.0;;
```

```
- : float = 3.
```

```
# float_of_int 1 +. 2.2;;
```

```
- : float = 3.2
```

Other Primitive Values

- OCaml provides six primitive values: integers, booleans, floating point numbers, characters, strings, and unit.

```
# 'c';;  
- : char = 'c'  
# "GIST";;  
- : string = "GIST"  
# ();;  
- : unit = ()
```

Conditional Expressions

if be then e_1 else e_2

- If be is true, the value of the conditional expression is the value of e_1 .
- If be is false, the value of the expression is the value of e_2 .

```
# if 2 > 1 then 0 else 1;;  
- : int = 0  
# if 2 < 1 then 0 else 1;;  
- : int = 1
```

- (Note 1) be must be a boolean expression.
- (Note 2) types of e_1 and e_2 must be equivalent.

```
# if 1 then 1 else 2;;  
Error: ...  
# if true then 1 else true;;  
Error: ...  
# if true then true else false;;  
- : bool = true
```

Names and Functions

- Create a global variable with the `let` keyword:

```
# let x = 3 + 4;;  
val x : int = 7
```

We say a variable x is bound to the value 7.

- You can refer to predefined global variables.

```
# let y = x + x;;  
val y : int = 14
```

- Create a local variable with `let ... in ...` construct:

$$\text{let } x = e_1 \text{ in } e_2$$

- ▶ x is bound to the value of e_1 .
- ▶ the scope of x is e_2 .
- ▶ the value of e_2 becomes the final value of the entire expression.

Examples

- ```
let a = 1 in a;;
- : int = 1
let a = 1 in a * 2;;
- : int = 2
```
- ```
# let a = 1 in  
  let b = a + a in  
  let c = b + b in  
    c + c;;  
- : int = 8
```
- ```
let d =
 let a = 1 in
 let b = a + a in
 let c = b + b in
 c + c;;
val d : int = 8
```

# Functions

- Define a function with `let`:

```
let square x = x * x;;
 val square : int -> int = <fun>
```

- Apply the function:

```
square 2;;
- : int = 4
square (2 + 5);;
- : int = 49
square (square 2);;
- : int = 16
```

- The body can be any expression:

```
let neg x = if x < 0 then true else false;;
val neg : int -> bool = <fun>
neg 1;;
- : bool = false
neg (-1);;
- : bool = true
```

# Functions

- Functions with multiple arguments:

```
let sum_of_squares x y = (square x) + (square y);;
val sum_of_squares : int -> int -> int = <fun>
sum_of_squares 3 4;;
- : int = 25
```

- Recursive functions are defined with `let rec` construct:

```
let rec factorial a =
 if a = 1 then 1 else a * factorial (a - 1);;
val factorial : int -> int = <fun>
factorial 5;;
- : int = 120
```



# Nameless Functions

- Many modern programming languages support nameless functions.
  - ▶ ML, Scala, Java8, JavaScript, Python, etc.
- In OCaml, a function can be defined without names:

```
fun x -> x * x;;
- : int -> int = <fun>
```

Called **nameless** or **anonymous** functions.

- Apply nameless function as usual:

```
(fun x -> x * x) 2;;
- : int = 4
```

- A variable can be bound to functions:

```
let square = fun x -> x * x;;
val square : int -> int = <fun>
```

- The following are equivalent:

```
let square = fun x -> x * x
let square x = x * x
```

# Functions are First-Class in OCaml

In programming languages, a value is **first-class**, if the value can be

- stored in a variable,
- passed as an argument of a function, and
- returned from other functions.

A language is often called **functional**, if functions are first class values, e.g., ML, Scala, Java8, JavaScript, Python, Lisp, etc.

# Functions are First-Class in OCaml

- Functions can be stored in variables:

```
let square = fun x -> x * x;;
square 2;;
- : int = 4
```

- Functions can be passed to other functions:

```
let sum_if_true test first second =
 (if test first then first else 0)
 + (if test second then second else 0);;
val sum_if_true : (int -> bool) -> int -> int -> int = <fun>

let even x = x mod 2 = 0;;
val even : int -> bool = <fun>

sum_if_true even 3 4;;
- : int = 4

sum_if_true even 2 4;;
- : int = 6
```

# Functions are First-Class in OCaml

- Functions can be also returned from a procedure:

```
let plus_a a = fun b -> a + b;;
val plus_a : int -> int -> int = <fun>

let f = plus_a 3;;
val f : int -> int = <fun>

f 1;;
- : int = 4

f 2;;
- : int = 5
```

Functions that manipulate functions are called **higher-order functions**.

- That is, functions that take as argument functions or return functions.
- This feature greatly increases the expressiveness of the language.

# Pattern Matching

- An elegant way of doing case analysis; you will better understand its advantages when manipulating variant types.
- E.g., using pattern-matching, the factorial function

```
let rec factorial a =
 if a = 1 then 1 else a * factorial (a - 1)
```

can be written as follows:

```
let rec factorial a =
 match a with
 | 1 -> 1
 | _ -> a * factorial (a - 1)
```

# Pattern Matching

The nested if-then-else expression

```
let isabc c = if c = 'a' then true
 else if c = 'b' then true
 else if c = 'c' then true
 else false
```

can be written using pattern matching:

```
let isabc c =
 match c with
 | 'a' -> true
 | 'b' -> true
 | 'c' -> true
 | _ -> false
```

or simply,

```
let isabc c =
 match c with
 | 'a' | 'b' | 'c' -> true
 | _ -> false
```

# Type Inference

In C or Java, types must be annotated:

```
public static int f(int n)
{
 int a = 2;
 return a * n;
}
```

In OCaml, type annotations are not mandatory:

```
let f n =
 let a = 2 in
 a * n;;
val f : int -> int = <fun>
```

# Type Inference

OCaml can infer types, no matter how complex the program is:

```
let sum_if_true test first second =
 (if test first then first else 0)
 + (if test second then second else 0);;
```



# Type Inference

OCaml can infer types, no matter how complex the program is:

```
let sum_if_true test first second =
 (if test first then first else 0)
 + (if test second then second else 0);;
```

```
val sum_if_true : (int -> bool) -> int -> int -> int = <fun>
```

OCaml compiler infers the type through the following reasoning steps:

- 1 the types of `first` and `second` must be `int`, because both branches of a conditional expression must have the same type,
- 2 the type of `test` is a function type  $\alpha \rightarrow \beta$ , because `test` is used as a function,
- 3  $\alpha$  must be of `int`, because `test` is applied to `first`, a value of `int`,
- 4  $\beta$  must be of `bool`, because conditions must be boolean expressions,
- 5 the return value of the function has type `int`, because the two conditional expressions are of `int` and their addition gives `int`.

# Type Annotation

Explicit type annotations are possible:

```
let sum_if_true (test : int -> bool) (x : int) (y : int) : int =
 (if test x then x else 0) + (if test y then y else 0);;
val sum_if_true : (int -> bool) -> int -> int -> int = <fun>
```

If the annotation is wrong, OCaml finds the error and report it:

```
let sum_if_true (test : int -> int) (x : int) (y : int) : int =
 (if test x then x else 0) + (if test y then y else 0);;
Error: The expression (test x) has type int but an expression
was expected of type bool
```

# Polymorphic Types

- What is the type of the program?

```
let id x = x
```

See how OCaml infers its type:

```
let id x = x;;
```

```
val id : 'a -> 'a = <fun>
```

The function works for values of any type:

```
id 1;;
```

```
- : int = 1
```

```
id "abc";;
```

```
- : string = "abc"
```

```
id true;;
```

```
- : bool = true
```

- Such a function is called **polymorphic**, and 'a is a **type variable**.

# Polymorphic Types

Quiz) What is the type of the function?

```
let first_if_true test x y =
 if test x then x else y
```

# Tuples

- An ordered collection of values, each of which can be a different types, e.g.,

```
let x = (1, "one");;
val x : int * string = (1, "one")
let y = (2, "two", true);;
val y : int * string * bool = (2, "two", true)
```

- Extract each component using pattern-matching:

```
let fst p = match p with (x,_) -> x;;
val fst : 'a * 'b -> 'a = <fun>
let snd p = match p with (_,x) -> x;;
val snd : 'a * 'b -> 'b = <fun>
```

or equivalently,

```
let fst (x,_) = x;;
val fst : 'a * 'b -> 'a = <fun>
let snd (_,x) = x;;
val snd : 'a * 'b -> 'b = <fun>
```

# Tuples

- Patterns can be used in let:

```
let p = (1, true);;
val p : int * bool = (1, true)
let (x,y) = p;;
val x : int = 1
val y : bool = true
```

# Lists

- A finite sequence of elements, each of which has the same type, e.g.,  
[1; 2; 3]

is a list of integers:

```
[1; 2; 3];;
```

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

Note that

- ▶ all elements must have the same type, e.g., [1; true; 2] is not a list,
  - ▶ the elements are ordered, e.g., [1; 2; 3]  $\neq$  [2; 3; 1], and
  - ▶ the first element is called **head**, the rest **tail**.
- []: the empty list, i.e., nil. What are head and tail of []?
  - [5]: a list with a single element. What are head and tail of [5]?

# List Examples

- `# [1;2;3;4;5];;`  
- : int list = [1; 2; 3; 4; 5]
- `# ["OCaml"; "Java"; "C"];;`  
- : string list = ["OCaml"; "Java"; "C"]
- `# [(1,"one"); (2,"two"); (3,"three")];;`  
- : (int \* string) list = [(1,"one"); (2,"two"); (3,"three")]
- `# [[1;2;3];[2;3;4];[4;5;6]];;`  
- : int list list = [[1; 2; 3]; [2; 3; 4]; [4; 5; 6]]
- `# [1;"OCaml";3] ;;`  
Error: This expression has type string but an expression was expected of type int



# List Operators

- `::` (cons): add a single element to the front of a list, e.g.,

```
1::[2;3];;
```

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

```
1::2::3::[];;
```

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

(`[1; 2; 3]` is a shorthand for `1::2::3::[]`)

- `@` (append): combine two lists, e.g.,

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

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

# Pattern Matching for Lists

Pattern matching is useful for manipulating lists.

- A function to check if a list is empty:

```
let isnil l =
 match l with
 [] -> true
 |_ -> false;;
val isnil : 'a list -> bool = <fun>
isnil [1];;
- : bool = false
isnil [];;
- : bool = true
```

# Pattern Matching for Lists

- Implement a function that computes the length of lists:

```
let rec length l = (* TODO *)
```

# Pattern Matching for Lists

- Implement a function that computes the length of lists:

```
let rec length l =
 match l with
 | [] -> 0
 | h::t -> 1 + length t;;
val length : 'a list -> int = <fun>
length [1;2;3];;
- : int = 3
```

- We can replace pattern `h` by `_`:

```
let rec length l =
 match l with
 | [] -> 0
 | _::t -> 1 + length t;;
```

# Variants (Union Data Types)

- Constructors can be associated with values, e.g.,  

```
type shape = Rect of int * int | Circle of int;;
type shape = Rect of int * int | Circle of int
```
- Construct values of the type:  

```
Rect (2,3);;
- : shape = Rect (2, 3)
Circle 5;;
- : shape = Circle 5
```
- A function that manipulates the data:  

```
let area s =
 match s with
 | Rect (w,h) -> w * h
 | Circle r -> r * r * 3;;
val area : shape -> int = <fun>
area (Rect (2,3));;
- : int = 6
area (Circle 5);;
- : int = 75
```

# Variants (Union Data Types)

- Inductive data types, e.g.,  

```
type mylist = Nil | List of int * mylist;;
type mylist = Nil | List of int * mylist
```
- Construct values of the type:  

```
Nil;;
- : mylist = Nil
List (1, Nil);;
- : mylist = List (1, Nil)
List (1, List (2, Nil));;
- : mylist = List (1, List (2, Nil))
```
- A function that manipulates the data:  

```
let rec mylength l =
 match l with
 | Nil -> 0
 | List (_,l') -> 1 + mylength l';;
val mylength : mylist -> int = <fun>
mylength (List (1, List (2, Nil)));;
- : int = 2
```

# Exceptions

- An exception means a run-time error: e.g.,

```
let div a b = a / b;;
val div : int -> int -> int = <fun>
div 10 5;;
- : int = 2
div 10 0;;
Exception: Division_by_zero.
```

- The exception can be handled with `try ... with` constructs.

```
let div a b =
 try
 a / b
 with Division_by_zero -> 0;;
val div : int -> int -> int = <fun>
div 10 5;;
- : int = 2
div 10 0;;
- : int = 0
```

# Exceptions

- User-defined exceptions: e.g.,

```
exception Problem;;
exception Problem
let div a b =
 if b = 0 then raise Problem
 else a / b;;
val div : int -> int -> int = <fun>
div 10 5;;
- : int = 2
div 10 0;;
Exception: Problem.
try
 div 10 0
 with Problem -> 0;;
- : int = 0
```



ML provides an elegant module system:

- Structure is a collection of types, exceptions, values, and functions, i.e., implementation details.
- Signature is the interface of the structure.

# Example

The interface of a queue data structure:

- `empty`: the empty queue
- `isempty`: the boolean-valued test of whether `q` is empty
- `enq(q,x)`: the queue obtained by inserting `x` on the end of `q`
- `deq(q)`: the queue obtained by removing the front element of `q` (also returns the front element)
- `print(q)`: show the contents of `q`
- `E`: the exception raised by `deq` if the queue is empty

# Example

The signature of the queue data structure:

```
module type IntQueue =
sig
 type t
 exception E
 val empty : t
 val is_empty : t -> bool
 val enq : t -> int -> t
 val deq : t -> int * t
 val print : t -> unit
end
```

# Example

An implementation:

```
module IntQueue : IntQueue =
 struct
 type t = int list
 exception E
 let empty = []
 let enq q x = q @ [x]
 let is_empty q = q = []
 let deq q = match q with [] -> raise E | h::t -> (h, t)
 let rec print q =
 match q with
 [] -> print_string "\n"
 |h::t -> print_int h; print_string " "; print t
 end
```

# Example

The module can be used as follows:

```
let q0 = IntQueue.empty
let q1 = IntQueue.enq q0 1
let q2 = IntQueue.enq q1 2
let (_,q3) = IntQueue.deq q2
let _ = IntQueue.print q1
let _ = IntQueue.print q2
let _ = IntQueue.print q3
```

The program prints:

```
1
1 2
2
```

# Example

The OCaml module system ensures the abstraction layer of the program:

```
let q4 = q1 @ [2]
```

produces a compile error:

```
Error: This expression has type IntQueue.t
 but an expression was expected of type 'a list
```

## Exercise 1: append

Implement a function that appends two lists:

```
append [1; 2; 3] [4; 5; 6; 7];;
- : int list = [1; 2; 3; 4; 5; 6; 7]
append [2; 4; 6] [8; 10];;
- : int list = [2; 4; 6; 8; 10]

let rec append l1 l2 = (* TODO *)
```

## Exercise 2: reverse

Implement a function that reverses a given list:

```
val reverse : 'a list -> 'a list = <fun>
reverse [1; 2; 3];;
- : int list = [3; 2; 1]
reverse ["C"; "Java"; "OCaml"];;
- : string list = ["OCaml"; "Java"; "C"]

let rec reverse l = (* TODO *)
```



## Exercise 3: nth-element

Implement a function that computes  $n$ th element of a list:

```
nth [1;2;3] 0;;
- : int = 1
nth [1;2;3] 1;;
- : int = 2
nth [1;2;3] 2;;
- : int = 3
nth [1;2;3] 3;;
Exception: Failure "list is too short".
```

```
let rec nth l n =
 match l with
 | [] -> raise (Failure "list is too short")
 | hd::tl -> (* TODO *)
```

## Exercise 4: remove-first

Implement a function that removes the first occurrence of an element from a list:

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

let rec remove_first a l =
 match l with
 | [] -> []
 | hd::tl -> (* TODO *)
```

## Exercise 5: insert

Implement a function that inserts an element to a sorted list:

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

```
let rec insert a l =
 match l with
 | [] -> [a]
 | hd::tl -> (* TODO *)
```

## Exercise 6: insertion sort

Implement a function that performs insertion sort:

```
let rec sort l =
 match l with
 | [] -> []
 | hd::tl -> insert hd (sort tl)
```

cf) Compare with “C-style” non-recursive version:

```
for (c = 1 ; c <= n - 1; c++) {
 d = c;
 while (d > 0 && array[d] < array[d-1]) {
 t = array[d];
 array[d] = array[d-1];
 array[d-1] = t;
 d--;
 }
}
```

# Summary

We've gone through the basics of OCaml programming.

- Expressions
- Names
- Functions
- Pattern matching
- Type inference
- Tuples and lists
- Variants (union data types)
- Exceptions
- Modules