OBJECTIVE CAML (OCAML)

```
# 1+2*3;;
- : int = 7
# let pi = 4.0 *. atan 1.0;;
val pi: float = 3.14159
# let square x = x *. x ;;
val square : float -> float = <fun>
# square (sin pi) +. square (cos pi) ;;
- : float = 1
# let rec fib n =
#   if n<2 then 1 else fib(n-1) + fib(n-2);;
val fib : int -> int = <fun>
# fib 10;;
- : int = 89
```

OCAML DATA TYPES

bool, char, string

false, 'a', "Hello world"

```
# let l = ["one"; "two"];;
val l : string list = ["one"; "two"]

# "zero" :: l;;
- : string list = ["zero"; "one"; "two"]
```

PATTERN MATCHING

```
# let rec sort lst =
#     match lst with
#         [] -> []
#       | head::tail -> insert head (sort tail)
# ;;
val sort: 'a list -> 'a list = <fun>
```

Pattern Matching (continued)

```
# let rec insert elt lst =
    match lst with
      [] -> [elt]
    | head :: tail ->
        if elt <= head then elt :: lst
        else head :: insert elt tail ;;

val insert : 'a -> 'a list -> 'a list = <fun>

# sort l;;
- : string list = ["two"; "one"; "zero"]
```

Polymorphism

```
# sort [6;2;5;3];;
- : int list = [2;3;5;6]
# sort [3.14;2.718];;
- : float list = [2.718; 3.14]
```

IMPERATIVE STYLE

```
# type point = { mutable x: float;
                 mutable y: float };;

# let translate p dx dy =
      p.x <- p.x +. dx;
      p.y <- p.y +. dy;;
val translate : point -> float -> float -> unit = <fun>

# let p1 = {x=0.0; y=0.0};;
val p1 : point = {x=0; y=0}

# translate p1 1.0 2.0;;
- : unit = ()

# p1 ;;
- : point = {x=1; y=2}
```

Arrays

```
# let add_vect v1 v2 =
    let len = min (Array.length v1)
                  (Array.length v2) in
    let res = Array.create len 0.0 in
    for i = 0 to len-1 do
      res.(i) <- v1.(i) +. v2.(i)
    done;
    res;;
val add_vect : float array -> float array ->
                                  float array = <fun>

# add_vect [| 1.0; 2.0 |] [| 3.0; 4.0 |];;

- : float array = [| 4; 6 |]
```

Reference Cell in OCAML

```
# type 'a ref = {mutable contents: 'a};;
# let (!) r = r.contents;;
val (!) : 'a ref → 'a = <fun>
# let (:=) r newval = r.contents ← newval;;
val (:=) : 'a ref → 'a → unit = <fun>
```

```
# let x0 = ref 0 ;;
val x0: int ref = {contents = 0}
# x0 := !x0 + 2 ;;
-: unit = ()
# !x0 ;;
-: int = 2
```

Exceptions

```
# exception Empty_list ;;

# let head l =
    match l with
      [] -> raise Empty_list
    | hd :: tl -> hd ;;
val head : 'a list -> 'a = <fun>

# let first l =
    try
      head l
    with Empty_list -> "Empty!" ;;
val first : string list -> string = <fun>

# first l ;;
- : string = "one"
# first [] ;;
- : string = "Empty!"
```

Standalone Caml Programs

```
(* File fib.ml *)

let rec fib n =
    if n<2 then 1 else fib(n-1)+fib(n-2);;

let main() =
    let arg = int_of_string Sys.argv.(1) in
    print_int (fib arg);
    print_newline();
    exit 0;;

main();;
```

```
→ ocamlc -o fib fib.ml
→ ./fib 10
89
→ ./fib 20
10946
```

LABELS

```
# List.map ;;
-: f:('a -> 'b) -> 'a list -> 'b list = <fun>

# String.sub ;;
-: string -> pos:int -> len:int -> string = <fun>


# let f ~x ~y = x - y ;;
val f : x:int -> y:int -> int = <fun>

# f ~x:3 ~y:2 ;;
-: int = 1
# f 3 2 ;;
-: int = 1
```

CLASSES AND OBJECTS

```
# class point =
      object
        val mutable x = 0
        method get_x = x
        method move d = x <- x + d
      end;;

class point :
  object val mutable x : int
         method get_x : int
         method move : int -> unit  end

# let p = new point;;
val p : point = <obj>
```

CLASSES AND OBJECTS (CONTINUED)

```
# p#get_x;;
- : int = 0
# p#move 3;;
- : unit = ()
# p#get_x;;
- : int = 3
```

SELF REFERENCES

```
# class printable_point x_init =
    object (s)
      val mutable x = x_init
      method get_x = x
      method move d = x <- x + d
      method print = print_int s#get_x
    end;;

# let p = new printable_point 7;;
val p : printable_point = <obj>

# p#print;;
7- : unit = ()
```

Virtual Methods

```
# class virtual abstract_point x_init =
    object (self)
      val mutable x = x_init
      method virtual get_x : int
      method get_offset = self#get_x - x_init
      method virtual move: int -> unit
    end;;
class virtual abstract_point :
  int ->
  object
    val mutable x: int
    method get_offset : int
    method virtual get_x : int
    method virtual move: int -> unit
  end
```

VIRTUAL METHODS (continued)

```
# class point x_init =
    object
      inherit abstract_point x_init
      method get_x = x
      method move d = x <- x + d
    end;;

class point :
  int ->
  object
    val mutable x : int
    method get_offset : int
    method get_x : int
    method move : int -> unit
  end
```

INHERITANCE

```
# class colored_point x (c: string) =
    object
      inherit point x
      val c = c
      method color = c
    end;;

# let p' = new colored_point 5 "red";;
val p' : colored_point = <obj>

# p'#get_x, p'#color;;
- : int * string = 5, "red"

# let get_succ_x p = p#get_x + 1;;
val get_succ_x : <get_x : int; ..> -> int = <fun>
# get_succ_x p + get_succ_x p';;
- : int = 8
```

MULTIPLE INHERITANCE

```
# class printable_colored_point y c =
    object (self)
      val c = c
      method color = c
      inherit printable_point y as super
      method print =
        print_string "(";
        super#print;
        print_string ", ";
        print_string (self#color);
        print_string ")"
    end;;

# let p' = new printable_colored_point 10 "red";;
new point at (10, red)
val p' : printable_colored_point = <obj>
# p'#print;;
(10, red)- : unit = ()
```

Modules in OCaml

Structures to package related definitions and enforce a consistent naming scheme

```
# module m =
    struct
      type ...
      let ...
      exception ...
    end;;

module m :
  sig
    type ...
    val ...
    exception ...
  end
```

Signatures

interfaces for structures
- to hide private components

```
# module type T =
    sig
      type ...
      val ...
      exception ...
    end;;
module type T =
  sig
    type ...
    val ...
    exception ...
  end
```

Functors

"functions" from structures to structures, to express parameterized structures

```
# module m =
    functor (Elt: Sig) ->
        struct
          type...
          let...
        end;;

module m:
  functor (Elt: Sig) ->
      sig
        type...
        val...
      end
```

e.g. to implement a set as a sorted list, a parameter structure provides { element type. ordering function.