CENG 2010 - Programming Language Concepts

Weeks 1-2: A Short Introduction to OCaml

Burak Ekici

March 6 - March 13, 2023

Table of Contents

OCaml

General-purpose, strongly typed programming language

OCaml

General-purpose, strongly typed programming language

- successor of Caml Light (itself successor of Caml),

OCaml

General-purpose, strongly typed programming language

- successor of Caml Light (itself successor of Caml),
- part of the ML family (SML, F$\#$, etc.)

OCaml

General-purpose, strongly typed programming language

- successor of Caml Light (itself successor of Caml),
- part of the ML family (SML, F#, etc.)
- designed and implemented at Inria Rocquencourt by Xavier Leroy and others

OCaml

General-purpose, strongly typed programming language

- successor of Caml Light (itself successor of Caml),
- part of the ML family (SML, F#, etc.)
- designed and implemented at Inria Rocquencourt by Xavier Leroy and others
- Some applications: symbolic computation and languages (IBM, Intel, Dassault Systèmes), static analysis (Microsoft, ENS), file synchronization (Unison), peer-to-peer (MLDonkey), finance (LexiFi, Jane Street Capital), teaching

The First Program

- `hello.ml`

```
let () = print_string "hello␣world!\n"
```

where

```
let () = ...
```

is the first entry of your program (like the “main” in C)

The First Program

- `hello.ml`

  ```
  let () = print_string "hello␣world!\n"
  ```

  where

  ```
  let () = ...
  ```

  is the first entry of your program (like the “main” in C)

Compilation	`% ocamlopt -o hello hello.ml`
Execution	`% ./hello`
Output	`hello word!`

Variable Declarations

- introducing a global variable

```
let x = e
```

Variable Declarations

- introducing a global variable

```
let x = e
```

- differences with respect to usual notion of variable:
  1. necessarily initialized

Variable Declarations

- introducing a global variable

```
let x = e
```

- differences with respect to usual notion of variable:
  1. necessarily initialized
  2. type not necessarily declared but inferred

Variable Declarations

- introducing a global variable

```
let x = e
```

- differences with respect to usual notion of variable:

1. necessarily initialized
2. type not necessarily declared but inferred
3. cannot be assigned afterwards

Variable Declarations

- introducing a global variable

```
let x = e
```

- differences with respect to usual notion of variable:
  1. necessarily initialized
  2. type not necessarily declared but inferred
  3. cannot be assigned afterwards
- versus:

Variable Declarations

- introducing a global variable

```
let x = e
```

- differences with respect to usual notion of variable:
  1. necessarily initialized
  2. type not necessarily declared but inferred
  3. cannot be assigned afterwards
- versus:

Java	OCaml
`final int x = 42;`	`let x = 42`

Variable Declarations

- introducing a global variable

```
let x = e
```

- differences with respect to usual notion of variable:
  1. necessarily initialized
  2. type not necessarily declared but inferred
  3. cannot be assigned afterwards
- versus:

Java	OCaml
`final int x = 42;`	`let x = 42`

- example:

```
let x = 1 + 2
let y = x * x

let () =
  print_int x;
  print_int y;
```

where semicolon (“;”) is the expression separator – sequencing

References

a variable to be assigned is called a reference; it is introduced with `ref`

```
let a = ref 1
let b = ref 2

let () =
  print_int !a;
  print_int !b;
  b := !a + 3;
  print_int !b;
```

Expressions and Statements

no distinction between expression/statement in the syntax : only expressions

usual constructs:

- conditional

```
let i = 1

let () =
  if i = 1 then print_int 1 else print_int 2
```

Expressions and Statements

no distinction between expression/statement in the syntax : only expressions

usual constructs:

- conditional

```
let i = 1

let () =
  if i = 1 then print_int 1 else print_int 2
```

- for loop

```
let a = ref 1

let () =
  for ind = 1 to 5 do a := !a + ind done
```

Unit Type

- expressions with no meaningful value (assignment, loop, . . .) have type unit

Unit Type

- expressions with no meaningful value (assignment, loop, ...) have type unit
- this type has a single value, written ()

Unit Type

- expressions with no meaningful value (assignment, loop, ...) have type unit
- this type has a single value, written ()
- it is the type given to the else branch when it is omitted

Unit Type

- expressions with no meaningful value (assignment, loop, ...) have type unit
- this type has a single value, written ()
- it is the type given to the else branch when it is omitted
  - correct

```
let () =
  if !a > 0 then a := 0
```

Unit Type

- expressions with no meaningful value (assignment, loop, ...) have type unit
- this type has a single value, written ()
- it is the type given to the else branch when it is omitted
  - correct

```
let () =
  if !a > 0 then a := 0
```

  - incorrect

```
let () =
  2 + (if !a > 0 then 1)
```

Local Variables

- in C or Java, the scope of a local variable extends to the bloc:

```
{
  int x = 1;
  ...;
}
```

Local Variables

- in C or Java, the scope of a local variable extends to the bloc:

```
{
  int x = 1;
  ...;
}
```

- in OCaml, a local variable is introduced with `let in`:

```
let x = 12 in x * x
```

Local Variables

- in C or Java, the scope of a local variable extends to the bloc:

```
{
  int x = 1;
  ...;
}
```

- in OCaml, a local variable is introduced with `let in`:

```
let x = 12 in x * x
```

- as for a local variable:

Local Variables

- in C or Java, the scope of a local variable extends to the bloc:

```
{
  int x = 1;
  ...;
}
```

- in OCaml, a local variable is introduced with `let in`:

```
let x = 12 in x * x
```

- as for a local variable:
  - necessarily initialized

Local Variables

- in C or Java, the scope of a local variable extends to the bloc:

```
{
  int x = 1;
  ...;
}
```

- in OCaml, a local variable is introduced with `let in`:

```
let x = 12 in x * x
```

- as for a local variable:
  - necessarily initialized
  - type inferred

Local Variables

- in C or Java, the scope of a local variable extends to the bloc:

```
{
  int x = 1;
  ...;
}
```

- in OCaml, a local variable is introduced with `let in`:

```
let x = 12 in x * x
```

- as for a local variable:
  - necessarily initialized
  - type inferred
  - immutable

Local Variables

- in C or Java, the scope of a local variable extends to the bloc:

```
{
  int x = 1;
  ...;
}
```

- in OCaml, a local variable is introduced with `let in`:

```
let x = 12 in x * x
```

- as for a local variable:
  - necessarily initialized
  - type inferred
  - immutable
  - but scope limited to the expression following in

"`let in =`" Expression

- `let x = e1 in e2`

  is an expression

“`let in =`” Expression

- `let x = e1 in e2`

  is an expression
- its type and value are those of e2

“`let in =`” Expression

- `let x = e1 in e2`

  is an expression
- its type and value are those of e2
- in an environment where x has the type and value of e1

"`let in =` " Expression

- `let x = e1 in e2`

  is an expression
- its type and value are those of e2
- in an environment where x has the type and value of e1
- Example:

  ```
  let x = 1 in (let y = 2 in x + y) * (let z = 3 in x * z)
  ```

Example

Java	OCaml
`{`	
`  int x = 1;`	`let x = ref 1 in`
`  x = x + 1;`	`x := !x + 1;`
`  int y = x * x;`	`let y = !x * !x in`
`  System.out.print(y);`	`print_int y`
`}`	

Table of Contents

Interactive Compiler

- ```
  % ocaml
  OCaml version 4.14.0
  Enter #help;; for help.
  ```
  ```

Interactive Compiler

- ```
  % ocaml
  OCaml version 4.14.0
  Enter #help;; for help.
  ```
- ```
  # let x = 1 in x + 2;;

  - : int = 3
  ```

Interactive Compiler

- ```
  % ocaml
  OCaml version 4.14.0
  Enter #help;; for help.
  ```
- ```
  # let x = 1 in x + 2;;
  ```
  ```
  - : int = 3
  ```
- ```
  # let y = 1 + 2;;
  ```
  ```
  val y : int = 3
  ```
  ```

Interactive Compiler

- ```
  % ocaml
  OCaml version 4.14.0
  Enter #help;; for help.
  ```
- ```
  # let x = 1 in x + 2;;
  ```
  ```
  - : int = 3
  ```
- ```
  # let y = 1 + 2;;
  ```
  ```
  val y : int = 3
  ```
- ```
  # y * y;;
  ```
  ```
  - : int = 9
  ```

Table of Contents

Syntax

- ```
  # let f x = x * x;;
  ```
  ```
  val f : int → int = <fun>
  ```
  ```

Syntax

- ```
  # let f x = x * x;;
  val f : int → int = <fun>
  ```
  - body = expression (no return)
  ```

Syntax

- `# let f x = x * x;;`

  `val f : int → int = <fun>`

  - body = expression (no return)
  - type is inferred (types of argument x and result)

Syntax

- ```
  # let f x = x * x;;
  ```

  ```
  val f : int → int = <fun>
  ```

  - body = expression (no return)
  - type is inferred (types of argument x and result)

  ```
  # f 4 ;;
  ```

  ```
  - : int = 16
  ```
  ```

Syntax

- ```
  # let f x = x * x;;
  val f : int → int = <fun>
  ```
  - body = expression (no return)
  - type is inferred (types of argument x and result)

  ```
  # f 4 ;;
  - : int = 16
  ```

- versus:
  ```

Syntax

- `# let f x = x * x;;`

  `val f : int → int = <fun>`

  - body = expression (no return)
  - type is inferred (types of argument x and result)

  `# f 4 ;;`

  `- : int = 16`

- versus:

Syntax

-
```
# let f x = x * x;;
val f : int → int = <fun>
```

  - body = expression (no return)
  - type is inferred (types of argument x and result)

```
# f 4 ;;
- : int = 16
```

- versus:

Java	OCaml

`static int f(int x)`
`{`
`  return x * x;`
`}`
	`let f x = x * x`

Procedure

a procedure = a function whose result type is `unit`

Procedure

a procedure = a function whose result type is `unit`

- ```
  # let x = ref 1
    let set v = x := v;;

  val x : int ref = {contents = 1}
  val set : int -> unit = <fun>
  ```
  ```

Procedure

a procedure = a function whose result type is `unit`

- ```
  # let x = ref 1
    let set v = x := v;;

  val x : int ref = {contents = 1}
  val set : int -> unit = <fun>
  ```

- ```
  # set 3;;

  - : unit = ()
  ```

Procedure

a procedure = a function whose result type is `unit`

- ```
  # let x = ref 1
    let set v = x := v;;

  val x : int ref = {contents = 1}
  val set : int -> unit = <fun>
  ```

- ```
  # set 3;;

  - : unit = ()
  ```

- ```
  # !x;;

  - : int = 3
  ```
  ```

Functions without Arguments

takes an argument of type `unit`

Functions without Arguments

takes an argument of type `unit`

- ```
  # let reset () = x := 0;;
  val reset : unit -> unit = <fun>
  ```
  ```

Functions without Arguments

takes an argument of type `unit`

- ```
  # let reset () = x := 0;;
  val reset : unit -> unit = <fun>
  ```
- ```
  # reset ();;
  - : unit = ()
  ```

Functions with Several Arguments

takes an argument of type `unit`

Functions with Several Arguments

takes an argument of type unit

- ```
  # let f x y z = if x > 0 then y + x else z - x;;
  ```

  ```
  val f : int → int → int → int = <fun>
  ```
  ```

Functions with Several Arguments

takes an argument of type unit

- `# let f x y z = if x > 0 then y + x else z - x;;`

  `val f : int → int → int → int = <fun>`

- `# f 1 2 3;;`

  `- : int = 3`

Local Functions

- function local to an expression

Local Functions

- function local to an expression
  - `# let sqr x = x * x in sqr 3 + sqr 4 = sqr 5;;`

    `- : bool = true`

Local Functions

- function local to an expression
  - ```
    # let sqr x = x * x in sqr 3 + sqr 4 = sqr 5;;
    - : bool = true
    ```
- function local to another function
    ```

Local Functions

- function local to an expression
  - ```
    # let sqr x = x * x in sqr 3 + sqr 4 = sqr 5;;
    ```
    ```
    - : bool = true
    ```
- function local to another function
  - ```
    # let pythagorean x y z =
        let sqr n = n * n in
        sqr x + sqr y = sqr z;;
    ```
    ```
    val pythagorean : int → int → int → bool = <fun>
    ```

Functions as First-Class Citizens

function = yet another expression, introduced with `fun`

Functions as First-Class Citizens

function = yet another expression, introduced with `fun`

- ```
  # fun x → x+1
  - : int → int = <fun>
  ```
  ```

Functions as First-Class Citizens

function = yet another expression, introduced with `fun`

- ```
  # fun x → x+1
  - : int → int = <fun>

  # (fun x → x+1) 3;;
  - : int = 4
  ```
  ```

Functions as First-Class Citizens

function = yet another expression, introduced with `fun`

- ```
  # fun x → x+1
  - : int → int = <fun>

  # (fun x → x+1) 3;;
  - : int = 4
  ```
  ```

Functions as First-Class Citizens

function = yet another expression, introduced with `fun`

- ```
  # fun x → x+1
  - : int → int = <fun>

  # (fun x → x+1) 3;;
  - : int = 4
  ```

internally

```
let f x = x+1;;
```

is identical to

```
let f = fun x → x+1;;
```
```

Partial Application

```
fun x y → x*x + y*y
```

is the same as

```
fun x → fun y → x*x + y*y
```

Partial Application

```
fun x y → x∗x + y∗y
```

is the same as

```
fun x → fun y → x∗x + y∗y
```

one can apply a function partially

Partial Application

```
fun x y → x*x + y*y
```

is the same as

```
fun x → fun y → x*x + y*y
```

one can apply a function partially

- ```
  # let f x y = x*x + y*y;;
  val f : int → int → int = <fun>
  ```
  ```

Partial Application

```
fun x y → x*x + y*y
```

is the same as

```
fun x → fun y → x*x + y*y
```

one can apply a function partially

- ```
  # let f x y = x*x + y*y;;
  val f : int → int → int = <fun>
  ```
- ```
  # let g = f 3;;
  val g : int → int = <fun>
  ```

Partial Application

```
fun x y → x*x + y*y
```

is the same as

```
fun x → fun y → x*x + y*y
```

one can apply a function partially

- ```
  # let f x y = x*x + y*y;;
  val f : int → int → int = <fun>
  ```
- ```
  # let g = f 3;;
  val g : int → int = <fun>
  ```
- ```
  # g 4;;
  - : int = 25
  ```
  ```

Partial Application (cont'd)

- a partial application is a way to return a function

Partial Application (cont'd)

- a partial application is a way to return a function
- but one can also return a function as the result of a computation

Partial Application (cont'd)

- a partial application is a way to return a function
- but one can also return a function as the result of a computation
- ```
  # let f x = let x2 = x * x in fun y → x2 + y * y;;
  val f : int → int → int = <fun>
  ```
  ```

Partial Application (cont'd)

- a partial application is a way to return a function
- but one can also return a function as the result of a computation
-
```
# let f x = let x2 = x * x in fun y → x2 + y * y;;
val f : int → int → int = <fun>
```
- a partial application of f computes x∗x only once

Example (Partial Application)

- ```
  # let count_from n = let r = ref (n-1) in fun () → incr r; !r;;
  ```

  ```
  val count_from : int → unit → int = <fun>
  ```
  ```

Example (Partial Application)

- ```
  # let count_from n = let r = ref (n-1) in fun () → incr r; !r;;
  val count_from : int → unit → int = <fun>

  # let c = count_from 0;;
  val c : unit → int = <fun>
  ```
  ```

Example (Partial Application)

- ```
  # let count_from n = let r = ref (n-1) in fun () → incr r; !r;;
  val count_from : int → unit → int = <fun>

  # let c = count_from 0;;
  val c : unit → int = <fun>
  ```

- ```
  # c ();;
  - : int = 0
  # c ();;
  - : int = 1
  ```

Higher-Order Functions

- functions that input and output functions

```
# let riemann (f: float → float) (a: int) (b: int) (n: int): float =
  let a = ref a in
  let s = ref 0.0 in
  let x = (b - !a)/n in
  a := !a + x;
  while !a ≤ b do
    s := !s +. (f (float !a) *. float x);
    a := !a + x
  done;
  !s;;

val riemann : (float → float) → int → int → int → float = <fun>
```

Recursive Functions

- in OCaml, it is idiomatic to use recursive functions, for

Recursive Functions

- in OCaml, it is idiomatic to use recursive functions, for
  - a function call is cheap

Recursive Functions

- in OCaml, it is idiomatic to use recursive functions, for
  - a function call is cheap
  - tail calls are optimized

Recursive Functions

- in OCaml, it is idiomatic to use recursive functions, for
  - a function call is cheap
  - tail calls are optimized
- Example:

```
let zero f =
  let rec lookup i = if f i = 0 then i else lookup (i+1)
  in lookup 0
```

Recursive Functions

- in OCaml, it is idiomatic to use recursive functions, for
  - a function call is cheap
  - tail calls are optimized
- Example:

```
let zero f =
  let rec lookup i = if f i = 0 then i else lookup (i+1)
  in lookup 0
```

- recursive code ⇒ clearer, simpler to justify

Polymorphism

- refers to many forms

Polymorphism

- refers to many forms
- polymorphic function = act over values with many different types

Polymorphism

- refers to many forms
- polymorphic function = act over values with many different types
- ```
  # let f x = x;;
  ```
  val f : $\alpha \rightarrow \alpha$ = <fun>
  ```

Polymorphism

- refers to many forms
- polymorphic function = act over values with many different types
- `# let f x = x;;`

  `val f : α → α = <fun>`

- `# f 3;;`

  `- : int = 3`

Polymorphism

- refers to many forms
- polymorphic function = act over values with many different types
- `# let f x = x;;`

  `val f :` $\alpha \to \alpha$ `= <fun>`

- `# f 3;;`

  `- : int = 3`

- `# f true;;`

  `- : bool = true`

Polymorphism

- refers to many forms
- polymorphic function = act over values with many different types
- `# let f x = x;;`

  `val f : α → α = <fun>`

- `# f 3;;`

  `- : int = 3`

- `# f true;;`

  `- : bool = true`

- `# f print_int;;`

  `- : int → unit = <fun>`

Polymorphism

- refers to many forms
- polymorphic function = act over values with many different types
- ```
  # let f x = x;;
  val f : α → α = <fun>
  ```
- ```
  # f 3;;
  - : int = 3
  ```
- ```
  # f true;;
  - : bool = true
  ```
- ```
  # f print_int;;
  - : int → unit = <fun>
  ```
- ```
  # f (print_int 20; Printf.printf "\n");;
  20
  - : unit = ()
  ```
  ```

Polymorphism (cont'd)

- OCaml always infers the most general type

Polymorphism (cont'd)

- OCaml always infers the most general type

Polymorphism (cont'd)

- OCaml always infers the most general type

```
# let compose f g = fun x → f (g x);;
val compose : (α → β) → (γ → α) → γ → β = <fun>
```

Table of Contents

Arrays

- ```
  # let a = Array.make 10 0;;
  ```
  ```
  val a : int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
  ```
  necessarily initialized
  ```

Arrays

- ```
  # let a = Array.make 10 0;;
  ```
  ```
  val a : int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
  ```
  necessarily initialized
- ```
  # let a = [| 1; 2; 3; 4 |];;
  # a.(1);;
  ```
  ```
  - : int = 2
  ```

Arrays

- ```
  # let a = Array.make 10 0;;
  ```
  ```
  val a : int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
  ```
  necessarily initialized
- ```
  # let a = [| 1; 2; 3; 4 |];;
  # a.(1);;
  ```
  ```
  - : int = 2
  ```
- ```
  # a.(1) ← 5;;
  ```
  ```
  - : unit = ()
  ```
  ```

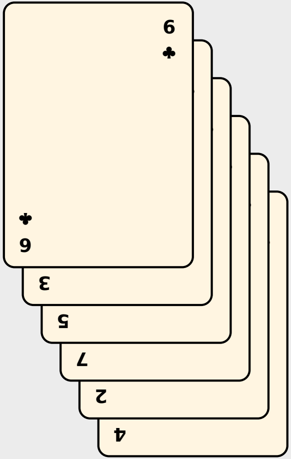
Arrays

- ```
  # let a = Array.make 10 0;;
  ```
  ```
  val a : int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
  ```
  necessarily initialized
- ```
  # let a = [| 1; 2; 3; 4 |];;
  # a.(1);;
  ```
  ```
  - : int = 2
  ```
- ```
  # a.(1) ← 5;;
  ```
  ```
  - : unit = ()
  ```
- ```
  # a;;
  ```
  ```
  - : int array = [|1; 5; 3; 4|]
  ```

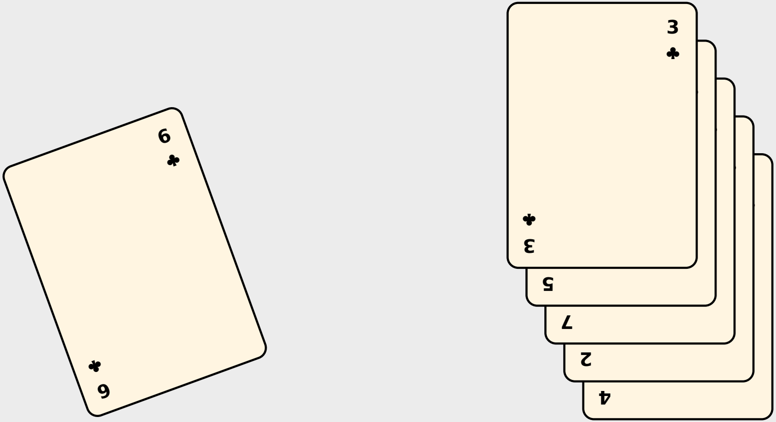
Example

Java	OCaml
`int[] a = new int[42];`	`let a = Array.make 42 0`
`a[17]`	`a.(17)`
`a[7] = 3;`	`a.(7) <- 3`
`a.length`	`Array.length a`

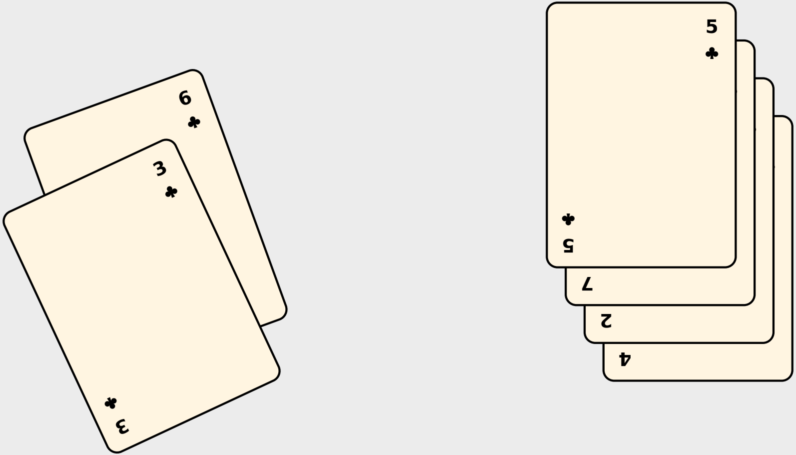
Example (Insertion Sort)



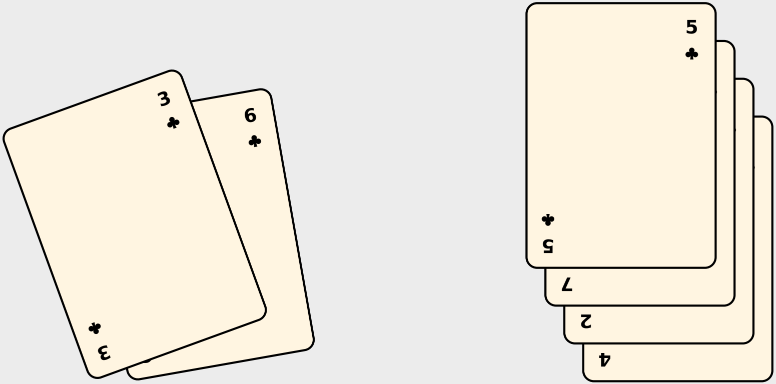
Example (Insertion Sort)



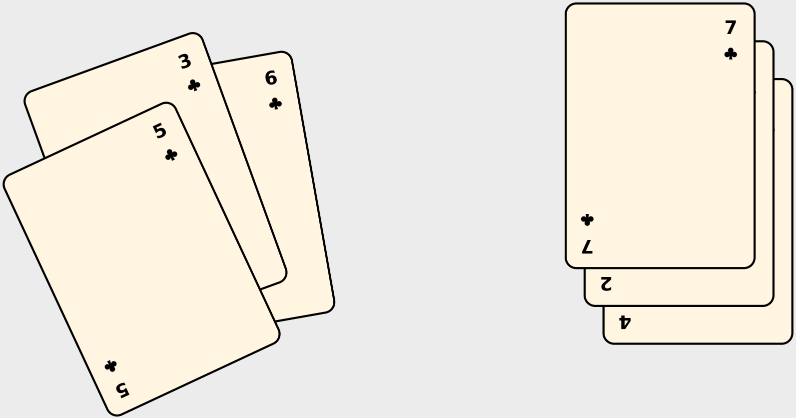
Example (Insertion Sort)



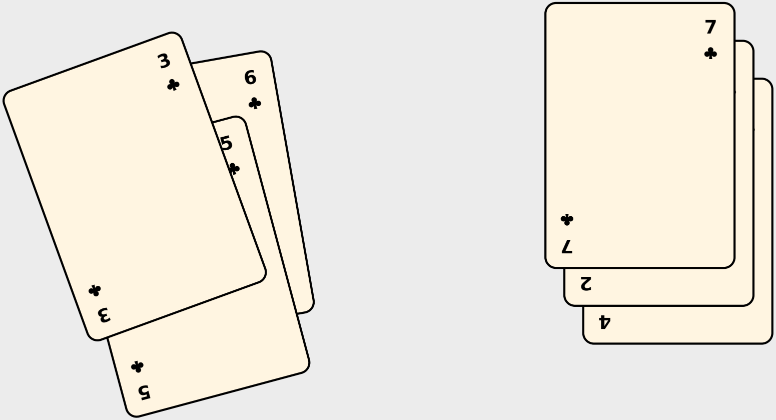
Example (Insertion Sort)



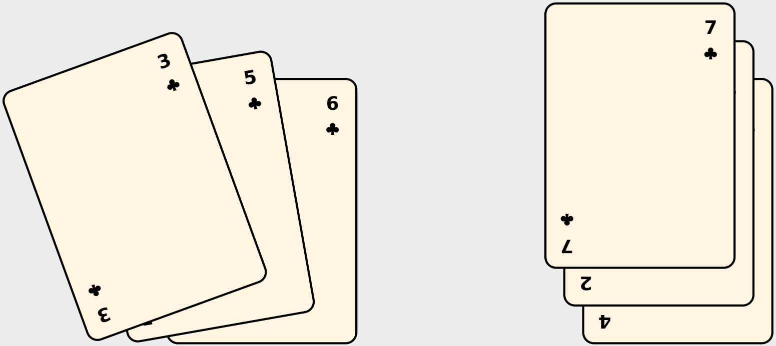
Example (Insertion Sort)



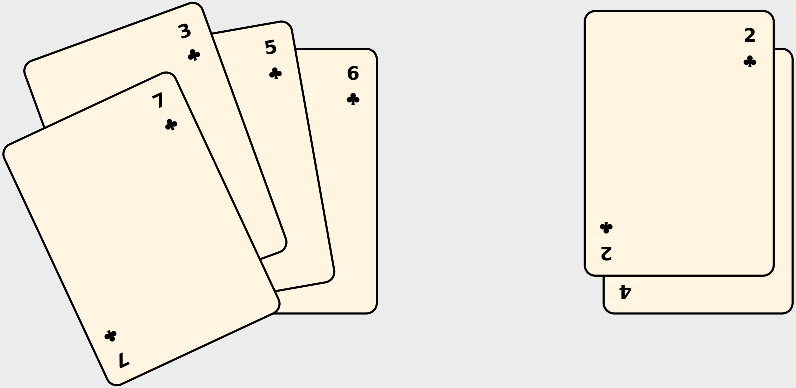
Example (Insertion Sort)



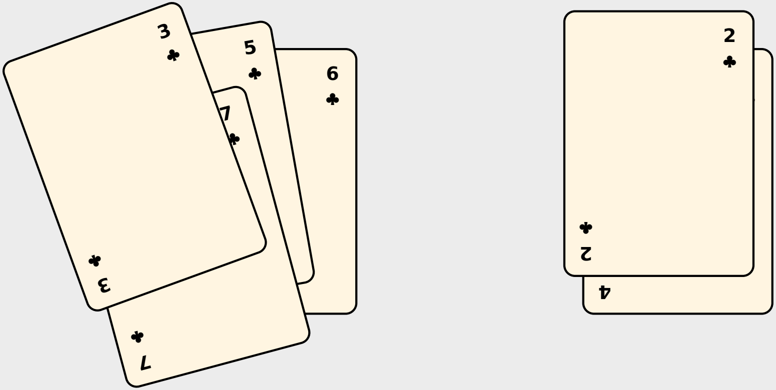
Example (Insertion Sort)



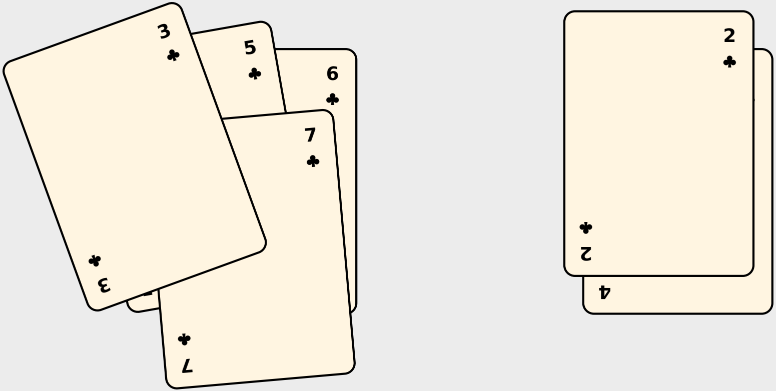
Example (Insertion Sort)



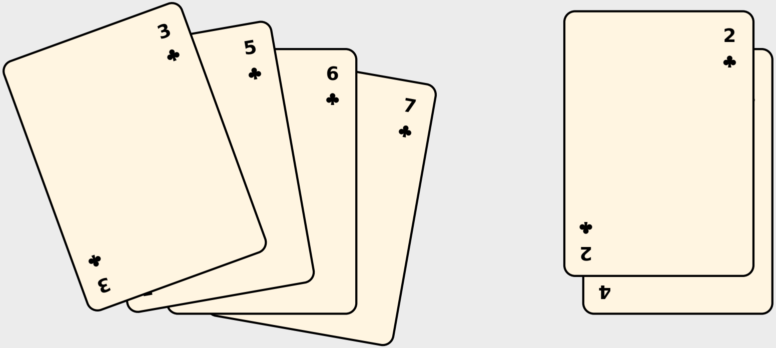
Example (Insertion Sort)



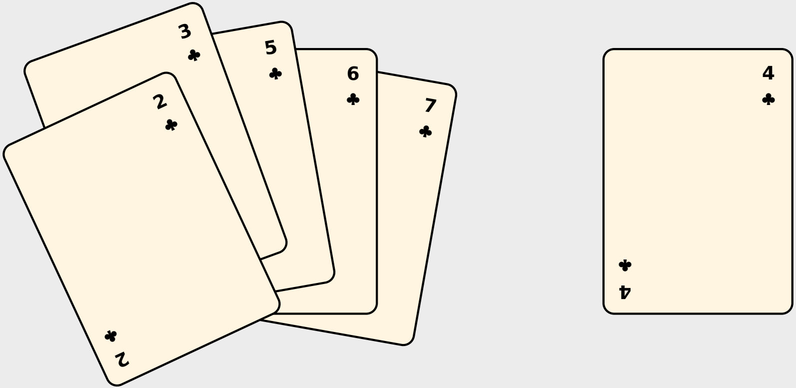
Example (Insertion Sort)



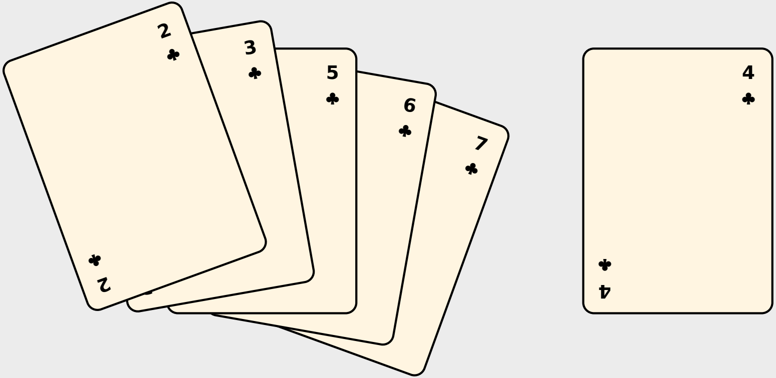
Example (Insertion Sort)



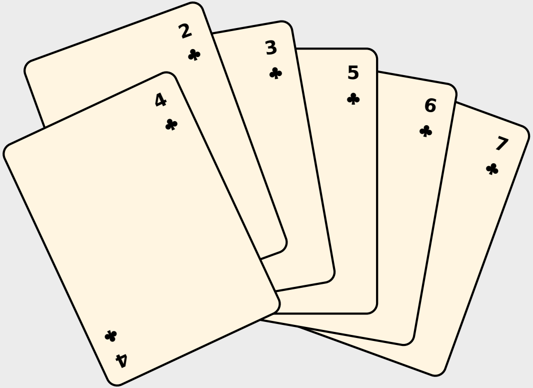
Example (Insertion Sort)



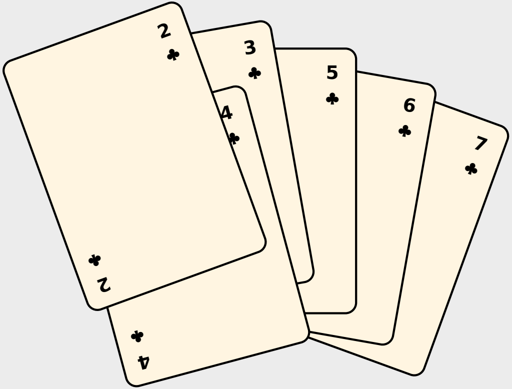
Example (Insertion Sort)



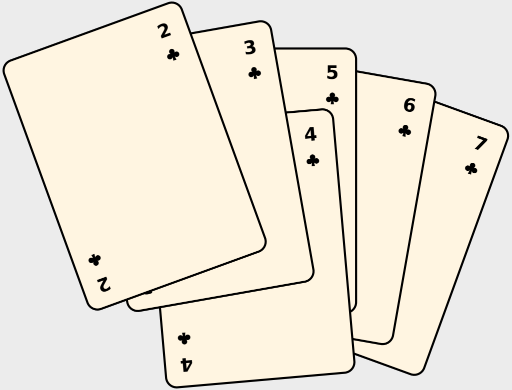
Example (Insertion Sort)



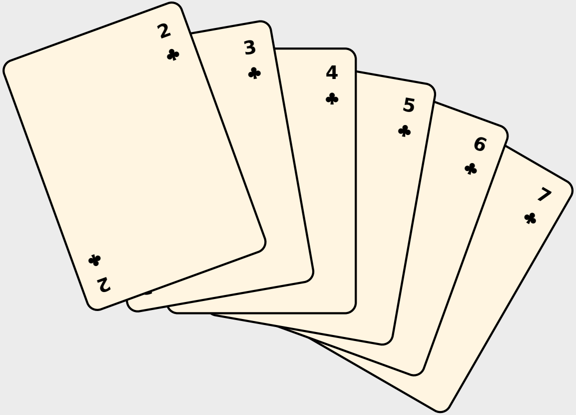
Example (Insertion Sort)



Example (Insertion Sort)



Example (Insertion Sort)



Example (Insertion Sort – iterative insertion)

```
let insertion_sort a =
  let swap i j =
    let t = a.(i) in a.(i) ← a.(j); a.(j) ← t
  in
  for i = 1 to Array.length a - 1 do
  (* insert element a[i] in a[0..i−1] *)
    let j = ref (i-1) in
    while !j ≥ 0 && a.(!j) > a.(!j+1) do
      swap !j (!j + 1); decr j
    done
  done
```

Example (Insertion Sort – recursive insertion)

```
let insertion_sort a =
  let swap i j =
    let t = a.(i) in a.(i) ← a.(j); a.(j) ← t
  in
  for i = 1 to Array.length a - 1 do
  (* insert element a[i] in a[0..i−1] *)
    let rec insert j =
      if j ≥ 0 && a.(j) > a.(j+1) then
      begin swap j (j+1); insert (j-1) end
    in
    insert (i-1)
  done
```

Table of Contents

Algebraic Data Types (Memory Representation)

- algebraic data type = union of several constructors

```
type formula =
  | True
  | False
  | And of formula * formula

# True;;

- : formula = True

# And (True, False);;

- : formula = And (True, False)
```

Pattern Matching (Algebraic Data Types)

- pattern matching generalizes to algebraic data types

```
# let rec eval = function
  | True          → true
  | False         → false
  | And (f1, f2) → eval f1 && eval f2;;

val eval : formula → bool = <fun>
```

Pattern Matching (Algebraic Data Types)

- pattern matching generalizes to algebraic data types

```
# let rec eval = function
  | True          → true
  | False         → false
  | And (f1, f2) → eval f1 && eval f2;;

val eval : formula → bool = <fun>
```

- patterns can be omitted or grouped:

```
let rec eval = function
  | True                                  → true
  | False                                 → false
  | And (False, _) | And (_, False) → false
  | And (f1, f2)                          → eval f1 && eval f2;;
```

Example (Positive Integers)

```
type pos =
  | XI: pos → pos
  | XO: pos → pos
  | XH: pos

let rec pos2Int (p: pos): int =
  match p with
    | XI k → 2 ∗ pos2Int k + 1
    | XO k → 2 ∗ pos2Int k
    | XH   → 1

let rec int2PosH (n: int) (m: int): pos =
  if n ≥ 1 && m ≥ 2 then int2PosH (n-1) (m-2)
  else if n ≥ 1 && m = 1 then XO (int2PosH (n-1) (n-1))
  else if n ≥ 1 && m = 0 then XI (int2PosH (n-1) (n-1))
  else XH

let int2Pos (i: int): pos =
  if i < 1 then failwith "input␣a␣non-negative␣integer"
  else int2PosH i (i+1)
```

Table of Contents

Lists

- lists = particular case of algebraic data type

Lists

- lists = particular case of algebraic data type
- predefined type of lists, α `list`, immutable and homogeneous

Lists

- lists = particular case of algebraic data type
- predefined type of lists, α `list`, immutable and homogeneous
- built from the empty list `[]` and addition in front of a list `::`

Lists

- lists = particular case of algebraic data type
- predefined type of lists, α `list`, immutable and homogeneous
- built from the empty list `[]` and addition in front of a list `::`
- ```
  # let l = 1 :: 2 :: 3 :: [];;
  val l : int list = [1; 2; 3]
  ```
  ```

Lists

- lists = particular case of algebraic data type
- predefined type of lists, α `list`, immutable and homogeneous
- built from the empty list `[]` and addition in front of a list `::`
-
```
# let l = 1 :: 2 :: 3 :: [];;
val l : int list = [1; 2; 3]
```
- shorter syntax
```
# let l = [1; 2; 3];;
```

Pattern Matching (Lists)

pattern matching = case analysis on a list

Pattern Matching (Lists)

pattern matching = case analysis on a list

- ```
  # let rec sum l =
      match l with
        | []     → 0
        | x :: r → x + sum r;;

  val sum : int list → int = <fun>

  # sum [1;2;3];;

  - : int = 6
  ```
  ```

Pattern Matching (Lists)

pattern matching = case analysis on a list

- ```
  # let rec sum l =
      match l with
        | []     → 0
        | x :: r → x + sum r;;

  val sum : int list → int = <fun>

  # sum [1;2;3];;

  - : int = 6
  ```

- shorter notation for a function performing pattern matching on its argument

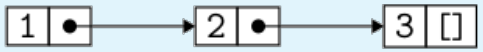
  ```
  let rec sum = function
    | []     → 0
    | x :: r → x + sum r;;
  ```
  ```

Lists (Memory Representation)

- OCaml lists = identical to lists in C or Java

Lists (Memory Representation)

- OCaml lists = identical to lists in C or Java
- the list [1; 2; 3] is represented as



Example (List Operations)

```
let rec myConcat (l1: α list)
                 (l2: α list): α list =
  match l1 with
    | []    → l2
    | x::xs → x :: myConcat xs l2

let rec myReverse (l: α list): α list =
  match l with
    | []    → []
    | x::xs → myConcat (myReverse xs) [x]
```

```
let rec myMap (f: α → β) (l: α list): β list =
  match l with
    | []    → []
    | x::xs → f x :: myMap f xs

let rec myFoldr (f: α → β → β)
                (u: β) (l: α list): β =
  match l with
    | []    → u
    | x::xs → f x (myFoldr f u xs)
```

Table of Contents

Tuples

- usual notation:

```
# (1,2,3);;

- : int * int * int = (1, 2, 3)
```

Tuples

- usual notation:

```
# (1,2,3);;
- : int * int * int = (1, 2, 3)

# let v = (1, true, "hello", 'a');;
val v : int * bool * string * char = (1, true, "hello", 'a')
```

Tuples

- usual notation:

```
# (1,2,3);;

- : int * int * int = (1, 2, 3)


# let v = (1, true, "hello", 'a');;

val v : int * bool * string * char = (1, true, "hello", 'a')
```

- access to components

```
# let (a,b,c,d) = v;;

val a : int = 1
val b : bool = true
val c : string = "hello"
val d : char = 'a'
```

Tuples (cont'd)

- beneficial when it comes to returning several values:

```
# let rec division n m =
    if n < m then (0, n)
    else let (q,r) = division (n - m) m in (q + 1, r);;

val division : int → int → int * int = <fun>
```

Tuples (cont'd)

- beneficial when it comes to returning several values:

```
# let rec division n m =
    if n < m then (0, n)
    else let (q,r) = division (n - m) m in (q + 1, r);;

val division : int → int → int * int = <fun>
```

- function taking a tuple as argument

```
# let f (x,y) = x + y;;

val f : int * int → int = <fun>

# f (1,2);;

- : int = 3
```

Example (Currification)

```
let myCurry (f: (α∗β) → γ): (α → β → γ) =
  fun (x: α) → fun (y: β) → f (x, y)

let addP (t: (int∗int)): int = fst t + snd t

let myUnCurry (f: α → β → γ): ((α∗β) → γ) =
  fun (x: (α∗β)) → f (fst x) (snd x)

let add (x: int) (y: int): int = x + y
```

Table of Contents

Records

- `type complex = { re : float; im : float }`

Records

- `type complex = { re : float; im : float }`
- allocation and initialization are simultaneous:

```
# let x = { re = 1.0; im = -1.0 };;
val x : complex = {re = 1.; im = -1.}
# x.im;;
- : float = -1.
```

Mutable Fields

```
type person = { name : string; mutable age : int }
# let p = { name = "Martin"; age = 23 };;
val p : person = {name = "Martin"; age = 23}
# p.age ← p.age + 1;;
- : unit = ()
# p.age;;
- : int = 24
```

Example

<table>
<tr><th>Java</th><th>OCaml</th></tr>
<tr><td><pre>class T
{
  final int v;
  boolean b;

  T(int v, boolean b)
  {
    this.v = v;
    this.b = b;
  }
}

T r = new T(42, true);
r.b = false;
r.v</pre></td><td><pre>type t =
{
  v: int;
  mutable b: bool;
}

let r = { v = 42; b = true }
r.b ← false
r.v</pre></td></tr>
</table>

References

a reference = a record of that predefined type

References

a reference = a record of that predefined type

- `type ‘a ref = { mutable contents : ‘a }`

References

a reference = a record of that predefined type

- `type ‘a ref = { mutable contents : ‘a }`
- `ref`, `!` and `:=` are syntactic sugar

References

a reference = a record of that predefined type

- `type ‘a ref = { mutable contents : ‘a }`

- `ref`, `!` and `:=` are syntactic sugar
- only arrays and mutable fields can be mutated

Example (Rationals)

```
type rat =
{
  num  : int;
  denom: pos;
}

let rat2String (r: rat): string =
  string_of_int (r.num) ^ "/" ^ string_of_int (pos2Int (r.denom))

let printRat (r: rat): unit =
  printf "%s\n" (rat2String r)

let ratEq (r1: rat) (r2: rat): bool =
  r1.num * (pos2Int r2.denom) = r2.num * (pos2Int r1.denom)
```

Table of Contents

OOP Classes

- class = collection of attributes (data) and methods

OOP Classes

- class = collection of attributes (data) and methods
- example:

```
class dice = object
  val max = 6
  val mutable faceValue = 0

  method roll =
    Random.self_init();
    faceValue ← (Random.full_int max) + 1

  method set_faceValue v =
    faceValue ← v

  method get_faceValue =
    faceValue
end
```

Example (inheritance)

```
class rectangle (w: int) (h: int) = object (s)
  val mutable x = 0
  val mutable y = 0

  method get_x = x
  method set_x v = x ← v

  method get_y = y
  method set_y v = y ← v

  method area = w ∗ h
end

class square (w: int) = object (s)
  inherit rectangle w w as r

  method is_contained (px, py) =
    x ≤ px && px ≤ x + w &&
    y ≤ py && py ≤ y + w

end
```

Example (ad-hoc polymorphism)

```
class virtual shape (name: string) = object (s)
  val virtual mutable x: int
  val virtual mutable y: int
  method virtual area: int

  method describe =
    "shape␣is␣" ^name ^ "␣at␣" ^ "(" ^ string_of_int x ^ "," ^ string_of_int y ^ ")" ^
    "␣with␣the␣area␣=␣" ^  string_of_int (s#area)
end
```

Example (ad-hoc polymorphism (cont'd))

```
class rectangle (w: int) (h: int) = object
  inherit shape "rectangle" as r

  val mutable x = 0
  val mutable y = 0

  method get_x = x
  method set_x v = x ← v

  method get_y = y
  method set_y v = y ← v

  method area = w ∗ h
end
```

Example (ad-hoc polymorphism (cont'd))

```
class square (w: int) = object
  inherit shape "square" as r

  val mutable x = 0
  val mutable y = 0

  method get_x = x
  method set_x v = x ← v

  method get_y = y
  method set_y v = y ← v

  method area = w ∗ w
end
```

Thanks! & Questions?