Revision: 1.13 Compilation Date: 2012/08/09 11:06:58 UTC

Ce document est une base pour constituer votre mémo sur le langage OCaml, il est donc à compléter et annoter en fonction de votre compréhension et maîtrise du langage.

Ce document n'est absolument pas un support de cours.

D'autre part, les constructions syntaxiques proposées dans ce document sont « compatibles » avec le langage OCaml : certaines restrictions sont utilisées dans ce mémo pour des raisons pédagogiques. Si vos connaissances du langage vont au delà de ce qui est présenté dans ce document, c'est tant mieux et c'est à vous de passer outre les restrictions proposées.

Dans ce document, les définitions globales se terminent toutes par un double point-virgule (;;). Les expressions se terminant par ;; sont des expressions déclarées globalement mais dont on ne récupère pas le résultat. Ce dernier ne peut donc être réutilisé.

On rappelle qu'en OCaml, les fonctions sont des valeurs au même titre que les autres valeurs telles que les chaînes de caractères par exemple.

1 Préliminaires

Les types de base

```
unit bool char int float string
```

Liste des mots-clefs (non utilisables en tant que nom de variable):

```
  and as  assert asr  begin class constraint  do done downto  else end
  exception  external false  for fun  function functor  if  in include
  inherit initializer land lazy let  lor lsl lsr lxor match method mod
  module mutable new object of open  or private rec sig struct then to
  true try type val virtual when while with
```

http://caml.inria.fr/pub/docs/manual-ocaml/manual044.html

2 Définitions de types

2.1 Définition d'alias

2.1.1 Nommer un type

```
type t1 = int ;;
type t2 = string ;;
type t3 = Char.t ;;
```

2.1.2 Nommer un type de n-uplet

```
type couple_entier_flottant = int * float ;;
type triplet_entier_flottant_entier = int * float * int ;;
type triplet_entier_flottant_chaine = int * float * string ;;
type ('a, 'b, 'c) tp3 = 'a * 'b * 'c ;;
type ('truc, 'machin, 'chose) tmc = 'truc * 'machin * 'chose ;;
type 'a triplet_entier_flottant_alpha = int * float * 'a  ;;
```

2.2 Définition d'un type somme

```
type enumeration = | A | B | C | D ;;
type constructeurs = | C1 of int | C2 of float | C3 of int * char ;;
type jeu_de_cartes = | Joker | Numero of int ;;
```

2.3 Définition d'un type enregistrement

```
type 'a ref = { mutable contents : 'a } ;;
type prenom_nom_age = { prenom : string ; nom : string ; mutable age : int } ;;
```

2.4 Définitions récursives de types

```
type nombre = | Entier of int | Flottant of float | Rationnel of ratio
and  ratio  = { numerateur : int ; denominateur : int } ;;
```

```
type 'a donnees_en_chaine = Rien | Quelquechose of 'a quelquechose
and 'a quelquechose = { donnee : 'a ; donnee_suivante : 'a donnees_en_chaine } ;;
```

Remarques

Après le mot-clef `of`, on peut mettre un nom de type ou bien un type de n-uplet. On ne peut pas mettre autre chose.

3 Définitions globales

3.1 Définition simple (let)

```
let nom = expression ;;
let nom1 = expression1
and nom2 = expression2 ;;
let nom_de_fonction param1 param2 = param1 + param2 ;;
```

3.2 Définition de fonctions récursives (let rec)

```
let rec nom_de_fonction param1 param2 =
  (if   (param1 = 0)
   then (param2)
   else (nom_de_fonction (param1 - 1) (param2 + 1)) ;;
```

```
let rec est_pair n =
  (if   (n = 0)
   then (true)
   else (est_impair (n-1)))
and est_impair n =
  (if   (n = 0)
   then (false)
   else (est_pair (n-1))) ;;
```

3.3 Définition d'exception

```
exception MonException ;;
type message = string ;;
exception MonException2 of message ;;
```

4 Expressions

4.1 Littéraux

4.1.1 Singleton (unit)

```
( )
```

4.1.2 Booléens (bool)

```
true false
```

Attention, **true** et **false** ne sont pas des noms de variables (ce sont des mots-clefs).

4.1.3 Caractères (char)

Un caractère est une valeur d'un octet exactement.

```
'\000'    '\001'              '\255'
'\x00'    '\x01'              '\xFF'
   'a'       'b'      'c'         'z'
   'A'       'B'      'C'         'Z'
   '0'       '1'      '2'         '9'
```

4.1.4 Nombres entiers (int)

```
 0  1  2  3  4    max_int
   -1 -2 -3 -4    min_int
 0x0 0xFEDCBA9
 0o0 0o12345670
```

4.1.5 Nombres à virgule (float)

```
 0.0 espilon 0.42  1. 4.2 1e23 1.32e23 1e1 1e0 infinity neg_infinity nan
```

4.1.6 Chaînes de caractères (string)

Chaque élément d'une chaîne de caractères est une valeur d'un octet exactement

```
"" "plop" "bonjour␣le␣monde\n" "\000\023Hello\023\000"
```

Accès à un élément d'une chaîne de caractères

```
(String.make 42 'X').[(24/2+1)]
```

```
"XXXX".[2]
```

Modification d'un élément d'une chaîne de caractères

```
(* test d'égalité qui s'évalue à true *)
(let x = String.make 2 'X' in (x.[1] <- 'Y'; x)) = "XY"
```

4.2 Application

Les applications de fonctions se font en notation préfixe.

```
(fonct1 param1)
(fonct2 param2 param3 param4)
(fonct3 param5 param6 param7 param8 param9)
(fonct4 (fonct5 param1) (fonct6 param42 param23))
```


4.3 Application infixe

Un certain nombre d'opérateurs sont binaires et s'utilisent en position infixe.

```
+   -   *   /   mod lsl  lsr  asr land  lor  lxor (* opérateurs sur les entiers *)
+.  -.  *.  /.  ** (* opérateurs sur les flottants *)
=   <>  >   <  (* opérateurs polymorphes *)
&& ||  (* opérateurs booléens *)
^ (* concaténation de 2 chaînes de caractères *)
@ (* concaténation de 2 listes  *)
```

```
[ (1+2); (2+3) ] @ [ 7; 9 ; (int_of_string ("1" ^ "1"))]
```

L'application infixe peut se transformer en application préfixe:

```
( @ ) [ (1+2); (2+3) ] [ 7; 9 ; (int_of_string ("1" ^ "1"))]
```

Les opérateurs infixes peuvent être utilisés en tant que valeur :

```
List.fold_left ( + ) 0 [ ((+) 1 2); 3; (6/2) ] (* rend 9 *)
```

4.4 Conditionnelle

```
(if (expr) then (expr) else (expr))
```

4.5 Séquence

```
(expr1; expr2; expr3)
```

```
begin expr1; expr2; expr3 end
```

4.6 Définition locale

```
(let nom = expr in expr)
```

```
(let fonct x y z = x + y + z in expr)
```

```
(let rec fact n = (if (n = 0) then (1) else (n * fact (n-1))) in expr)
```

```
(let rec even n = (n  =  0) || (odd  (n - 1))
 and     odd  n = (n <>  1) || (even (n - 1))
 in expr)
```

4.7 Boucle conditionnelle

```
(while expr_booleenne do
  expr_de_type_unit
 done)
```

4.8 Boucle bornée

```
(for identifiant = expr1_de_type_int to expr2_de_type_int do
  expr_de_type_unit
 done)
(* Si (expr1_de_type_int > expr2_de_type_int)
   alors on ne rentre pas dans le corps de la boucle. *)
```

```
(for identifiant = expr1_de_type_int downto expr2_de_type_int do
  expr_de_type_unit
 done)
(* Si (expr1_de_type_int < expr2_de_type_int)
   alors on ne rentre pas dans le corps de la boucle. *)
```

4.9 N-uplets

4.9.1 Construction (allocation)

```
(expr1, expr2)
(expr1, expr2, expr3, expr4, expr5)
```

L'accès aux champs nécessite le filtrage par motifs.

4.9.2 Enregistrements (n-uplets avec champs nommés)

Construction (ou allocation)

```
{ champ1 = expr1 ; champ2 = expr2 }
```

Accès en lecture

```
(expression).nom_du_champ
```

Accès en écriture

```
(expression).nom_du_champ <- expr_nouvelle_valeur
```

4.10 Listes

La liste vide: `[ ]`

`[(expr1); (expr2); (expr2); ]` ≡ `((expr1)::(expr2)::(expr3)::[ ])`

Remarque: `[ ]` et `::` sont des constructeurs (comme ceux des types sommes) à syntaxe particulière.

4.11 Tableaux

Le tableau vide: `[| |]`

`[| (expr1); (expr2); (expr2); |]`
≡ `(let x = Array.make 3 expr1 in x.(1) <- expr2; x.(2) <- expr3; x)`

4.11.1 Accès en lecture d'une case

```
(expr).(numero_de_case)
```

```
([| 1;3;54;2;42 |]).(4)
```

4.11.2 Accès en écriture d'une case

```
(expr).(numero_de_case) <- (expr_nouvelle_valeur)
```

```
([| 1;3;54;2;42 |]).(4) <- 23
```


4.12 Filtrage par motifs

Attention à l'homogénéité des types : chaque motif dans un filtrage doit filtrer des expressions d'un même type.

4.12.1 Filtrage d'entiers

```
 match expr with
| 1 -> expr
| 2 -> expr
| _ -> expr
```

4.12.2 Filtrage de nombres à virgule

```
 match expr with
| 1.2 -> expr
| 3.2e10 -> expr
| _ -> expr
```

4.12.3 Filtrage de caractères

```
 match expr with
| 'a' -> expr
| 'b' .. 'z' -> expr
| _ -> expr
```

Les caractères sont les seuls à pouvoir bénéficier de la notation `..` (deux points consécutifs).

4.12.4 Filtrage de chaînes de caractères

```
 match expr with
| "Bonjour" -> expr
| "Au revoir" -> expr
| _ -> expr
```

Le filtrage des chaînes de caractères ne concerne pas les « expressions rationelles ». Par exemple, il n'est pas possible d'exprimer un motif filtrant une chaîne **quelconque** commençant par "Bonjour" et se finissant par "Bye.".

4.12.5 Filtrage des constructeurs de types sommes

```
match expr with
| None -> expr
| Some x -> expr
```

4.12.6 Filtrage des enregistrements

Les champs filtrés doivent impérativement exister.

```
match expr with
| { champ1 = truc ; champ2 = machin } -> expr
```

```
match expr with
| { prenom = p ; nom = n ; age = a } ->
     "Son prénom est "  ^ p ^ ", son nom est " ^ n ". "
     ^ "Et cette personne a " ^ string_of_int a ^ " ans."
```

```
match expr with
| { prenom = p ; nom = n ; age = 1 } ->
     "Son prénom est "  ^ p ^ ", son nom est " ^ n ". "
```

```
    ^ "Et␣cette␣personne␣a␣1␣an."
| { prenom = p ; nom = n ; age = a } ->
    "Son␣prénom␣est␣"  ^ p ^ ",␣son␣nom␣est␣" ^ n ".␣"
    ^ "Et␣cette␣personne␣a␣" ^ string_of_int a ^ "␣ans."
```

```
match expr with
| { identite = i ; biens = [| |] } ->
    i ^ "␣n'a␣aucun␣bien"
| { identite = i ; biens = b  } ->
    i ^ "␣a␣" ^ string_of_int (Array.length b) ^ "␣objets␣parmi␣ses␣biens."
```

4.12.7 Filtrage liant (lier une variable à une valeur)

```
(match expr1 with | nom -> expr2)
```

Cette construction équivaut (à peu près) à :

```
(let nom = expr1 in expr2)
```

4.12.8 Multifiltrage et Filtrage anonyme

```
let rec fact n = match n with
 | 0 | 1 -> 1
 | _ -> n * fact (n - 1) ;;
```

4.12.9 Filtrage des listes

```
let rec est_de_longueur_paire lst = match lst with
 | [ ] -> true
 | [ e ] -> false
 | x :: y :: reste -> est_de_longueur_paire reste ;;
```

```
let est_de_longueur_3 lst = match lst with
 | [ _ ; _ ; _ ] -> true
 | _ -> false ;;
```

4.12.10 Filtrage des tableaux

On ne peut filtrer qu'un nombre de cases constant pour un tableau (attention, c'est donc différent des listes) :

```
let est_de_taille_3 tab = Array.length tab = 3 ;;
let est_de_taille_3_bis tab = match tab with
 | [| _ ; _ ; _ |] -> true
 | false ;;
```

4.12.11 Filtrage des n-uplets

```
(match (expr1, expr2) with
 | (motif1, motif2) -> expr3)
```

```
(match (f a b c) with
 | ([ ], 42) -> expr3
 | ((e::tl), 23) -> expr4)
```

4.12.12 Filtrage par motifs combinés

Il est possible de combiner les motifs (quand cela a un sens, bien sûr).


```
type enr_exn = { une_exception : exn } ;;
type constr  = E of enr_exn * int ;;
let rec foobar trucs = match trucs with
 | [ E ({ une_exception = Not_found }, 1) ] -> Invalid_argument "cool"
 | [ E ({ une_exception = x }, 42) ] -> x
 | [ E ({ une_exception = _ }, _ ) ] -> failwith "pas␣bon"
 | chose :: autres_trucs -> foobar autres_trucs
 | [] -> failwith "vide?" ;;
(*val foobar : constr list -> exn*)
```

4.13 Exceptions

4.13.1 Lever une exception

Toutes les exceptions sont de type `exn`. Elles ne peuvent être levées qu'avec la fonction `raise` du module `Pervasives`.

```
val raise : exn -> 'a
```

```
(raise (LeNomDeLException))
(raise (Not_found))
(raise (Failure ("ça␣ne␣va␣pas␣du␣tout")))
```

4.13.2 Rattraper une exception

```
(try expression with
  | Not_found -> expr
  | Invalid_argument msg -> expr
  | Failure msg -> expr)
```

```
(try expression with
  | not_found -> expr) (*ici on rattrape **toutes** les exceptions
                          et on nomme ``not_found'' celle rattrapée *)
```

4.14 Surparenthésage

Il vaut mieux mettre des parenthèses supplémentaires qui n'ont pas d'effet plutôt que d'en omettre par erreur : `((expr))` ≡ `(expr)` mais `(expr)` ≢ `expr`
Dans ce mémo, il y a volontairement un surparenthésage qui vise à limiter les erreurs.

5 Remarques

Un nom de variable ne peut pas commencer par une majuscule, car les mots commençants par une majuscule sont réservés aux constructeurs des types sommes et aux modules. Le contexte permet de facilement distinguer les noms des types sommes des noms de modules.

Les mots-clefs ne peuvent pas être utilisés en tant que nom de variable.

6 Extrait de la Bibliothèque standard

Il y a des restrictions ici aussi. Documentation officielle des modules : `http://caml.inria.fr/pub/docs/manual-ocaml/libref/index_modules.html`

6.1 Module Pervasives

```
val raise : exn -> 'a
val invalid_arg : string -> 'a
val failwith : string -> 'a
exception Exit
val ( = ) : 'a -> 'a -> bool
val ( <> ) : 'a -> 'a -> bool
val ( < ) : 'a -> 'a -> bool
val ( > ) : 'a -> 'a -> bool
val ( <= ) : 'a -> 'a -> bool
val ( >= ) : 'a -> 'a -> bool
val compare : 'a -> 'a -> int
val min : 'a -> 'a -> 'a
val max : 'a -> 'a -> 'a
val ( == ) : 'a -> 'a -> bool
val ( != ) : 'a -> 'a -> bool
val not : bool -> bool
val ( && ) : bool -> bool -> bool
val ( || ) : bool -> bool -> bool
val ( ~- ) : int -> int
val succ : int -> int
val pred : int -> int
val ( + ) : int -> int -> int
val ( - ) : int -> int -> int
val ( * ) : int -> int -> int
val ( / ) : int -> int -> int
val ( mod ) : int -> int -> int
val abs : int -> int
val max_int : int
val min_int : int
val ( land ) : int -> int -> int
val ( lor ) : int -> int -> int
val ( lxor ) : int -> int -> int
val lnot : int -> int
val ( lsl ) : int -> int -> int
val ( lsr ) : int -> int -> int
val ( asr ) : int -> int -> int
val ( ~-. ) : float -> float
val ( +. ) : float -> float -> float
val ( -. ) : float -> float -> float
val ( *. ) : float -> float -> float
val ( /. ) : float -> float -> float
val ( ** ) : float -> float -> float
val sqrt : float -> float
val exp : float -> float
val log : float -> float
val log10 : float -> float
val cos : float -> float
val sin : float -> float
val tan : float -> float
val acos : float -> float
```



```
val asin : float -> float
val atan : float -> float
val atan2 : float -> float -> float
val cosh : float -> float
val sinh : float -> float
val tanh : float -> float
val ceil : float -> float
val floor : float -> float
val abs_float : float -> float
val mod_float : float -> float -> float
val frexp : float -> float * int
val ldexp : float -> int -> float
val modf : float -> float * float
val float : int -> float
val float_of_int : int -> float
val truncate : float -> int
val int_of_float : float -> int
val infinity : float = infinity
val neg_infinity : float = neg_infinity
val nan : float = nan
val max_float : float = 1.79769313486231571e+308
val min_float : float = 2.22507385850720138e-308
val epsilon_float : float = 2.22044604925031308e-16
type fpclass = | FP_normal | FP_subnormal| FP_zero | FP_infinite | FP_nan
val classify_float : float -> fpclass
val ( ^ ) : string -> string -> string
val int_of_char : char -> int
val char_of_int : int -> char
val ignore : 'a -> unit
val string_of_bool : bool -> string
val bool_of_string : string -> bool
val string_of_int : int -> string
val int_of_string : string -> int
val string_of_float : float -> string
val float_of_string : string -> float
val fst : 'a * 'b -> 'a
val snd : 'a * 'b -> 'b
val ( @ ) : 'a list -> 'a list -> 'a list
type in_channel = in_channel
type out_channel = out_channel
val stdin : in_channel = <abstr>
val stdout : out_channel = <abstr>
val stderr : out_channel = <abstr>
val print_char : char -> unit
val print_string : string -> unit
val print_int : int -> unit
val print_float : float -> unit
val print_endline : string -> unit
val print_newline : unit -> unit
val prerr_char : char -> unit
val prerr_string : string -> unit
val prerr_int : int -> unit
val prerr_float : float -> unit
val prerr_endline : string -> unit
val prerr_newline : unit -> unit
```

```
val read_line : unit -> string
val read_int : unit -> int
val read_float : unit -> float
type open_flag = | Open_rdonly | Open_wronly | Open_append | Open_creat
       | Open_trunc | Open_excl | Open_binary | Open_text | Open_nonblock
val open_out : string -> out_channel
val open_out_bin : string -> out_channel
val open_out_gen : open_flag list -> int -> string -> out_channel
val flush : out_channel -> unit
val flush_all : unit -> unit
val output_char : out_channel -> char -> unit
val output_string : out_channel -> string -> unit
val output : out_channel -> string -> int -> int -> unit
val output_byte : out_channel -> int -> unit
val output_binary_int : out_channel -> int -> unit
val output_value : out_channel -> 'a -> unit
val seek_out : out_channel -> int -> unit
val pos_out : out_channel -> int
val out_channel_length : out_channel -> int
val close_out : out_channel -> unit
val open_in : string -> in_channel
val open_in_bin : string -> in_channel
val open_in_gen : open_flag list -> int -> string -> in_channel
val input_char : in_channel -> char
val input_line : in_channel -> string
val input : in_channel -> string -> int -> int -> int
val really_input : in_channel -> string -> int -> int -> unit
val input_byte : in_channel -> int
val input_value : in_channel -> 'a
val seek_in : in_channel -> int -> unit
val pos_in : in_channel -> int
val in_channel_length : in_channel -> int
val close_in : in_channel -> unit
type 'a ref = 'a ref = { mutable contents : 'a; }
val ref : 'a -> 'a ref
val ( ! ) : 'a ref -> 'a
val ( := ) : 'a ref -> 'a -> unit
val incr : int ref -> unit
val decr : int ref -> unit
val string_of_format : ('a, 'b, 'c, 'd, 'e, 'f) format6 -> string
val format_of_string : ('a,'b,'c,'d,'e,'f)format6 -> ('a,'b,'c,'d,'e,'f)format6
val exit : int -> 'a
val at_exit : (unit -> unit) -> unit
val do_at_exit : unit -> unit
```

6.2 Module List

```
val length : 'a list -> int
val hd : 'a list -> 'a
val tl : 'a list -> 'a list
val nth : 'a list -> int -> 'a
val rev : 'a list -> 'a list
val append : 'a list -> 'a list -> 'a list
val rev_append : 'a list -> 'a list -> 'a list
val concat : 'a list list -> 'a list
val flatten : 'a list list -> 'a list
```



```
val iter : ('a -> unit) -> 'a list -> unit
val map : ('a -> 'b) -> 'a list -> 'b list
val rev_map : ('a -> 'b) -> 'a list -> 'b list
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
val iter2 : ('a -> 'b -> unit) -> 'a list -> 'b list -> unit
val map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
val rev_map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
val fold_left2 : ('a -> 'b -> 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a
val fold_right2 : ('a -> 'b -> 'c -> 'c) -> 'a list -> 'b list -> 'c -> 'c
val for_all : ('a -> bool) -> 'a list -> bool
val exists : ('a -> bool) -> 'a list -> bool
val for_all2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool
val exists2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool
val mem : 'a -> 'a list -> bool
val memq : 'a -> 'a list -> bool
val find : ('a -> bool) -> 'a list -> 'a
val filter : ('a -> bool) -> 'a list -> 'a list
val find_all : ('a -> bool) -> 'a list -> 'a list
val partition : ('a -> bool) -> 'a list -> 'a list * 'a list
val assoc : 'a -> ('a * 'b) list -> 'b
val assq : 'a -> ('a * 'b) list -> 'b
val mem_assoc : 'a -> ('a * 'b) list -> bool
val mem_assq : 'a -> ('a * 'b) list -> bool
val remove_assoc : 'a -> ('a * 'b) list -> ('a * 'b) list
val remove_assq : 'a -> ('a * 'b) list -> ('a * 'b) list
val split : ('a * 'b) list -> 'a list * 'b list
val combine : 'a list -> 'b list -> ('a * 'b) list
val sort : ('a -> 'a -> int) -> 'a list -> 'a list
val merge : ('a -> 'a -> int) -> 'a list -> 'a list -> 'a list
```

6.3 Module Array

```
val length : 'a array -> int
val get : 'a array -> int -> 'a
val set : 'a array -> int -> 'a -> unit
val make : int -> 'a -> 'a array
val init : int -> (int -> 'a) -> 'a array
val make_matrix : int -> int -> 'a -> 'a array array
val append : 'a array -> 'a array -> 'a array
val concat : 'a array list -> 'a array
val sub : 'a array -> int -> int -> 'a array
val copy : 'a array -> 'a array
val fill : 'a array -> int -> int -> 'a -> unit
val blit : 'a array -> int -> 'a array -> int -> int -> unit
val to_list : 'a array -> 'a list
val of_list : 'a list -> 'a array
val iter : ('a -> unit) -> 'a array -> unit
val map : ('a -> 'b) -> 'a array -> 'b array
val iteri : (int -> 'a -> unit) -> 'a array -> unit
val mapi : (int -> 'a -> 'b) -> 'a array -> 'b array
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b array -> 'a
val fold_right : ('a -> 'b -> 'b) -> 'a array -> 'b -> 'b
val sort : ('a -> 'a -> int) -> 'a array -> unit
```

6.4 Module String

```
val length : string -> int
val get : string -> int -> char
val set : string -> int -> char -> unit
val make : int -> char -> string
val copy : string -> string
val sub : string -> int -> int -> string
val fill : string -> int -> int -> char -> unit
val blit : string -> int -> string -> int -> int -> unit
val concat : string -> string list -> string
val iter : (char -> unit) -> string -> unit
val escaped : string -> string
val index : string -> char -> int
val rindex : string -> char -> int
val index_from : string -> int -> char -> int
val rindex_from : string -> int -> char -> int
val contains : string -> char -> bool
val contains_from : string -> int -> char -> bool
val rcontains_from : string -> int -> char -> bool
val uppercase : string -> string
val lowercase : string -> string
val capitalize : string -> string
val uncapitalize : string -> string
type t = string
val compare : t -> t -> int
```

6.5 Module Char

```
val code : char -> int
val chr : int -> char
val escaped : char -> string
val lowercase : char -> char
val uppercase : char -> char
type t = char
val compare : t -> t -> int
```

Table des matières

Si vous avez des questions ou remarques, vous pouvez contacter l'auteur du document par courriel : Philippe.Wang@LiP6.fr