Cours Informatique MP.

Caml (le minimum).

Antoine MOTEAU
antoine.moteau@wanadoo.fr

. . . /Caml.tex (2003)
. . . /Caml.tex Compilé le vendredi 09 mars 2018 à 16h 40m 32s avec LaTeX.
Compilable avec LaTeX, PDFLaTeX, LuaLaTeX, XeLaTeX.

CAML (le minimum)

Table des matières

Eléments pour CAML Light (le minimum)

1 Environnement (sous Windows)

1.1 Installation de Caml Light (CamlWin), de CMDCAML

. . . (voir la doc)

1.2 Compilation directe de programmes Caml

1. Extrait raccourci simplifié de commandes de compilation :

   Dans une fenêtre Dos, on produit un fichier `.exe` depuis le fichier Caml `truc.ml` par la commande

   - `camlc -o truc.exe truc.ml` , si `truc.ml` est autonome
   - `camlc -o truc.exe bibli1.zo bibli2.zo ... biblin.zo truc.ml` , si `truc.ml` utilise les bibliothèques non standard `bibli1.zo bibli2.zo ... biblin.zo` (ordre d'énoncé signifiant).

2. Limitations :

   - Le programme DOS ainsi compilé ne peut pas utiliser de fenêtre graphique (utiliser OCaml).
   -

1.3 Utilitaire CamlWin pour Caml Light

- Dans CamlWin, on exécute pas à pas chaque élément de programme.
- CamlWin permet l'utilisation d'une fenêtre graphique.

Recommandé : utiliser un éditeur de texte extérieur pour l'écriture du programme.

1.4 Utilisation de l'éditeur CMDCAML avec la fenêtre CamlWin

CMDCAML (Editeur Caml light), en liaison avec CamlWin, permet d'écrire le code et d'évaluer les constituants d'un programme, morceau par morceau, par transfert ("copier-coller-exécuter" automatique) dans la fenêtre **Camlwin**, par exemple :

- "évaluer sélection" (bouton droit de la souris) **Problèmes avec Win 2000, XP ...**.
- "évaluer tout" (non recommandé en cours de conception . . .) **Problèmes avec Win 2000, XP ...**.
- "copier" + "coller" dans CamlWin + "exécuter" dans CamlWin .

  Dans ce cas, au préalable, ouvrir **CamlWin** depuis **CMDCAML** par le menu "fenetre" + "Commande Caml (Ctrl+O)" .

Avec "évaluer sélection" ou "évaluer tout", normalement, à la première tentative, CMDCAML devrait lancer automatiquement la fenêtre CamlWin si elle n'est pas déjà lancée, mais cela ne marche pas bien avec Windows 2000, Windows XP, et on perd parfois la fin du fichier (suite du bloc évalué) dans **CMDCAML** !

IMPORTANT : sauvegarder souvent le travail en cours !

1.5 Utilisation d'un autre éditeur

On peut utiliser bloc-Note (recommandé), Wordpad, Word etc . . .

Il faut avoir au préalable lancé, de façon indépendante, CamlWin.

Ensuite, depuis l'éditeur, "copier" + "coller" dans CamlWin + "exécuter" dans CamlWin .

2 Caml Light (éléments)

2.1 Mots clé de base (non exhaustif)

`let`	déclaration (préalable).	Exemple : `let a = 1 in ...`
`in`	suite éventuelle de `let`.	
`and`	mot de liaison.	Exemple : `let a = ref 0 and b = 2 in ...`
`ref`	déclaration de variable référence (à contenu modifiable).	
`!`	dé-référencement.	Exemple : `let a = ref 0 in ...; a := !a + 1; ...`
`:=`	affectation de valeur à une variable référence.	
`=`	élément de déclaration ou test.	Exemple : `if !a = b then a := !a + 1;`
`<-`	affectation de valeur à une composante de vecteur, de chaîne, à un champ mutable.	Exemple : `v.(i) <- v.(i) + 1; s.[i] <- 'a';`
`{..}`	ensemble à champs nommés.	Exemple : `type enrg = {car :char; compte :int};;`
`mutable`	déclaration : composant (champ) modifiable sur place. Exemple : `type enrg2 = {mutable car2 :char; mutable compte2 : valeur};;`	
`.`	déclinaison.	Exemples : `enrg2.compte2 <- 1 + enrg2.compte2; v.(i) <- v.(i) + 1; s.[i] <- 'a';`
	point décimal.	Exemples : `w.(i) <- 12.17 +. 23.61;;`
`match ... with`	filtrage.	
`|`	alternative de filtrage.	
`when`	complément ou précision dans un fitrage.	
`as`	alias (autre nom).	Exemple : `let essai = function (a,b) as ab -> a + snd ab;;`
`function`	fonction à une seule variable.	Exemple : `let f = function a,b -> a + b;;`
`fun`	fonction curifiée	Exemple : `let f = fun a b -> a + b;;`
`rec`	déclaration (fonction récursive).	Exemple : `let rec som = function L -> [] -> 0 | h::t -> h + som t and prod = function L -> [] -> 1 | h::t -> h * prod t;;`
`begin ... end`	déclaration : grouper une séquence d'instructions en une seule.	
`( instructions )`	idem `begin ... end`, mais pas toujours lisible, à ne pas confondre avec	
`( ... , ... )`	couple, n-uplet.	Exemple : `let (a,b) = (0,1) in ...`
`fst , snd`	premier, second d'un couple.	Exemple : `snd (1,2);; let (_,_,c) = (1,2,3) in c;;`
`,`	séparateur dans un n-uplet.	
`;`	séparateur (d'instructions, d'éléments dans une liste).	
`;;`	terminateur (de déclaration, de fonction).	
`()`	résultat "rien" (de type `unit`).	
`_` (isolé)	valeur "n'importe quoi".	Exemple : `let (_,_,c) = (1,2,3) in c;;`
`__` (double)	"déclinaison" (de bibliothèque).	Exemple : `list__mem a [1,2,5,10];;`
`:`	élément de déclaration.	Exemple : `let Myfonction (i : int) = ...`
`==`	identité structurelle.	Exemple : `type caractere == char;;`
`prefix`	déclaration (opérateurs).	
`try ... with`	traitement des exceptions.	
`exception`	définition des exceptions.	
`(* ... *)`	commentaires. Attention : `(*` et `*)` sans espace, et veiller à bien fermer !	
`where`	déclaration après coup ... (n'existe pas en OCaml).	

2.2 Variables (minimum)

2.2.1 Variables constantes nommées (exemples)

```
let a = 2 in ...
```
le contenu de a est non modifiable.

```
let v = make_vect 10 0 in ... ; v.(3) <- 17;...
```
les composantes de v sont modifiables !

2.2.2 Variables références (exemples)

```
let a = ref (-2) in ... ; a := !a + 1; ...
```

`a:= 1 +!a;;` (sans espace) est une erreur.

2.3 Types simples

- Caractères (type `char`). Exemple : `let c = 'a';`
  Fonctions utiles : `int_of_char` , `char_of_int` , `char_for_read`
- Booléens (type `bool`), de valeurs `true` ou `false`.
  Opérateurs : "et" : `&` ou `&&` (mais pas `and`) , "ou" : `or` ou `||` , "non" : `not`
- Entiers (type `int`, de `min_int = -1073741824` à `max_int = 1073741823`).
  Opérateurs : `+` , `-` , `*` , `/` , `quo`, `mod` , `<` , `<=` , `>` , `>=` mais pas `**` ni `^`
  Attention : `max_int + max_int` vaut `... -2` !
- Floats (type `float`, de ? à ?).
  Opérateurs pointés : `+.` , `-.` , `*.` , `/.` , `**` , `=.` , `<.` , `<=.` , `>.` , `>=.`
- Constantes particulières :
  `false` , `true`
  `()` soit "rien" (de type unit), à ne pas confondre avec _ (isolé) qui signifie "n'importe quoi".
  `[]` liste vide.

2.4 Types structurés prédéfinis (indices à partir de 0)

- Couples, *n*-uplets
  Exemples : `let ab = (2, 3) and abc = (4,5,6) in ...`
  Fonction utiles : `fst` , `snd` (pour un couple) mais pas pour un triplet et il n'y a pas de "third" !
  Pour un triplet, il faut utiliser une "description" nommée :
  `let (a,b,c) = (4,5,6) in ...; let u = (a+b)*c in ...`
- Vecteurs (indice à partir de 0). Fonctions importantes : `make_vect`, `vect_length`.
  Exemples : `let v0 = [| 1; 2; 3 |]; v0.(2) <- 5; v0;; (* ->  [| 1; 2; 5 |]; *)`
  `let v = make_vect 7 0 in ...; for i=1 to vect_length v - 1 do v.(i) <- v.(i) + i done;...`

  Attention : `let v = make_vect 10 (make_vect 10 0);;` conduit à un grave problème :
  `v.(2).(3) ← 2; v.(7).(3) ← 7; print_int v.(2).(3);; (* PB, c'est 7 ! *)`

  Par contre la conception suivante est correcte :
  ```
  let v = map_vect (fun x -> make_vect 10 0) (make_vect 10 0);;
  v.(2).(3) <- 2;; v.(7).(3) <- 7;; print_int v.(2).(3);;  (* OK, c'est 2 *)
  ```
- Matrices (indices à partir de 0). Fonctions importantes : `make_matrix`, `vect_length`.
  Exemples : `let M = make_matrix 5 9 0 in ...;`
  `M.(0).(2) <- 3; vect_length M; (* 5 *); vect_length M.(0); (* 9 *)...`
- Chaînes de caractères (indice à partir de 0). Fonctions, opérateurs : `string_length`, `^` (concaténation).
  Exemples : `let s = "essai" in ... ; s.[3] <- 'u'; s ^ "s glaces."; ...`
- Listes . Opérateurs : `::` (cons), `hd` , `tl`, `[... ; ... ]` .
  Exemples :
  `let L = 1 :: 2 :: 3 :: [4;5] in ...;  hd L; (* c'est 1 *)  tl L; (* c'est [2;3;4;5] *)`
  Il y a aussi `list_length` (à éviter) , @ (concaténation), à éviter.

2.5 Types structurés définis par l'utilisateur (exemples élémentaires)

2.5.1 Type ensemble à champs nommés (ou "enregistrement") ; exemple :

```
type Ascci = { caractere : char ; code : int };;
let a = { caractere = 'a'; code = 97 };;  print_char a.caractere;; print_int accu.code;;
```

2.5.2 Types à champs alternatifs (sélectionnables) ; exemple :

```
type Arbre =   Feuille of int
             | Noeud of Arbre * int * Arbre
;;
let a = Noeud (Noeud (Feuille 0, 3, Noeud( Feuille 4, 5, Feuille 3)), 6, Feuille 4);;
match a with
    Feuille i       -> ...
  | Noeud (g, e, d) -> ... ;;
```

2.6 Types à éléments mutables (modifiables en place)

- C'est le cas des composantes d'un vecteur, d'une chaîne de caractères : `v.(i) <- 12;` / `s.[i] <- 'a';;`
- Ensemble à champs nommés mutables ; exemple :

```
type Accumulateur = { mutable caractere : char ; mutable compte : int };;
let accu = { caractere = 'a' ; compte = 97 };;
accu.caractere <- 'b'; accu.compte <- accu.compte + 1;  accu;;
```

- Type à champs alternatifs (sélectionnables) mutables ; exemple :

```
type Arbre =   Feuille of mutable int
             | Noeud of mutable Arbre * int * Arbre
;;
let a = Noeud ( Noeud ( Feuille 0, 3, Noeud( Noeud(Feuille 1,4,Feuille 2), 5, Feuille 3))
              , 6, Feuille 4);;
```

Mise des étiquettes de feuilles à 0, par effet de bord :

```
let rec mise_a_zero_F a =  match a with
  | Feuille i     -> i <- 0
  | Noeud (g,e,d) -> mise_a_zero_F g; mise_a_zero_F d;
;;
(* mise_a_zero_F : Arbre -> Arbre = <fun>  *)

mise_a_zero_F a;;  (* résultat : - : unit = () *)
```

mais le paramètre a été modifié :

```
a;;
(* résultat : - : Arbre =
   Noeud (Noeud (Feuille 0, 3, Noeud (Noeud (Feuille 0, 4, Feuille 0), 5, Feuille 0)),
          6, Feuille 0)          *)
```

Mise des étiquettes de nœuds à 0, par effet de bord :

```
let rec mise_a_zero_N a =  match a with
  | Feuille i           -> ()
  | Noeud ((g,e,d) as n) -> mise_a_zero_N g; mise_a_zero_N d; n <- (g,0,d)
;;
(* mise_a_zero_N : Arbre -> unit = <fun> *)

mise_a_zero_N a;;  (* résultat : - : unit = () *)
```

mais le paramètre a été modifié :

```
a;;
(* résultat : - : Arbre =
   Noeud (Noeud (Feuille 0, 0, Noeud (Noeud (Feuille 0, 0, Feuille 0), 0, Feuille 0)),
          0, Feuille 1)           *)
```

2.7 Structures de programation

2.7.1 Bloc d'instructions : `begin ... ; ... end` ou `( ... ; ... )`

2.7.2 Alternative

- `if ... then` *une instruction ou un bloc d'instruction* `else` *une instruction ou un bloc d'instruction* .
  Exemple : `let xor a b = if a then not b else b;;`
- `if ... then` *une instruction ou un bloc d'instruction* .

  Le `else` n'étant pas spécifié, il est implicitement de la forme `else ()`, de type `unit`

  Par exemple : `if a = 1 then 2;;` est refusé car le `else` implicite renvoie `()` de type `unit` $\neq$ `int`.

2.7.3 Filtrage, sélection multiple, selon descripteurs

- Cas des listes,

```
match xx with                    (* cas de base pour les listes *)
  | [ ]                 -> ...
  | a::q                -> ... ;;

match xx with                    (* extension au cas de base, avec cas intermédiaires *)
  | [ ]                 -> ...
  | [a]                 -> ...
  | a::b::q when a < 10 -> a + b + ...
  | a::q                -> ... ;;
```

- autres : voir fonctions, arbres, ...

2.7.4 Boucles

- `for k = 1 to 10 do ... ; .... ; .... done; ...`
- `for k = 10 downto 10 do ... ; .... ; .... done; ...`
-
```
let fini = ref false            (* condition à initialiser au préalable *)
in while not !fini
   do ... ; ... ; ...           (* la condition doit évoluer pour terminer la boucle *)
   done;
```

2.8 Fonctions

2.8.1 Fonctions à un paramètre : function

```
let rec factorielle = function          (* let rec factorielle n = match n with *)
    0 -> 1
  | n -> n * factorielle (n-1)
;;
(* factorielle : int -> int = <fun> *)

let rev liste = renverse [] liste       (* let rev = function liste -> renverse [] liste     *)
  where rec renverse accu = function    (*   where rec renverse = fun accu x -> match x with *)
      []   -> accu                      (*   where rec renverse accu x = match x with        *)
    | a::q -> renverse (a::accu) q
;;
(* rev : 'a list -> 'a list = <fun> *)

let rec do_list f = function            (* let rec do_list f x = match x with         *)
  | []   -> ()                          (* let rec do_list = fun f x -> match x with  *)
  | a::q -> f a; do_list f q
;;
(* do_list : ('a -> 'b) -> 'a list -> unit = <fun> *)

let print_int_list = do_list (function i -> print_int i; print_char ' ')
;;
(* print_int_list : int list -> unit = <fun> *)
```

2.8.2 Fonction curifiées (paramètres en "cascade") : fun

```
let rec renverse = fun accu liste -> match liste with    (* let rec renverse accu = function *)
      []   -> accu                                       (* let rec renverse accu liste = match liste with *)
    | a::q -> renverse (a::accu) q
;;
(* renverse : 'a list -> 'a list -> 'a list = <fun> *)

let it_list = fun f accu x -> match x with           (* let it_list f accu x = match x with *)
   | []   -> accu                                    (* let it_list f accu = function       *)
   | a::q -> it_list f (f accu a) q
;;
(* it_list : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun> *)

let somme = it_list (fun a b -> a+b) 0;;     (* Paramètre "implicite" et fun obligatoire ici *)
(* somme : int list -> int = <fun> *)
```

2.9 Fonctions auxiliaires internes à une fonction (exemples)

1. Calcul de la somme des éléments d'une liste d'entiers : on introduit une fonction auxiliaire avec, en paramètre supplémentaire, un accumulateur de somme partielle, initialisé à 0 :
   - Déclaration préalable de la fonction auxiliaire (`let ... in ...`) :

     ```
     let somme liste =
       let rec sommer accu = function           (* let rec sommer accu x = match x with *)
            []   -> accu
          | b::q -> sommer (accu + b) q
       in sommer 0 liste
     ;;
     (* somme : int list -> int = <fun> *)
     ```

   - Déclaration repoussée de la fonction auxiliaire (`... where ...`) (n'existe pas en OCaml) :

     ```
     let somme = sommer 0              (* PARAMETRE IMPLICITE (NON DECRIT EXPLICITEMENT) *)
       where rec sommer accu = function  (* let rec sommer = fun accu x -> match x with *)
            []   -> accu
          | b::q -> sommer (accu + b) q
     ;;
     (* somme : int list -> int = <fun> *)
     ```

2. Renversement de liste (avec type faible) : fonction auxiliaire avec accumulateur.

   ```
   let rev =                          (* PARAMETRE IMPLICITE *)
    let rec renverse accu = function  (* let rec renverse = fun accu x -> match x with *)
         []   -> accu                 (* let rec renverse accu x = match x with       *)
       | a::q -> renverse (a::accu) q
    in renverse []
   ;;
   (* rev : '_a list -> '_a list = <fun> *) (* Type faible, qui sera fixé au premier appel *)
   let rev = renverse []              (* PARAMETRE IMPLICITE *)
     where rec renverse accu = function  (* where rec renverse = fun accu x -> match x with *)
         []   -> accu                    (* where rec renverse accu x = match x with       *)
       | a::q -> renverse (a::accu) q
   ;;
   (* rev : '_a list -> '_a list = <fun> *) (* Type faible, qui sera fixé au premier appel *)
   ```

   Remarque. Pour avoir un type polymorphe, il faut avoir un paramètre explicite :

   ```
   let rev liste = renverse [] liste
     where ....
   ;;
   (* rev : 'a list -> 'a list = <fun> *) (* Type polymorphe *)
   ```

2.10 ...

3 Relation d'ordre, ordre générique

3.1 Egalité, ordre générique (exemple)

```
12.35 < 15.0;;        (* -> true *)        "abc" < "abdef";;                  (* -> true  *)
(4,5) <= (4,6);;      (* -> true *)      (4,5,6) <= (3,8,0);;                 (* -> false *)
[1;2;3] < [1;2;3;4];; (* -> true *)      [1;2;5] <= [1;2;3;4];;               (* -> false *)
[1;2;3] = [1;2;1+2];; (* -> true *)      [1.0;2.0;3.0] = [1.0;2.0;1.0 +. 2.0];; (* -> true  *)
```

3.2 Passage d'un opérateur (d'ordre) comme paramètre à une fonction (exemple)

1. Tri rapide de liste (générique), avec ordre passé en paramètre :

```
let rec tri_rapide_list ordre = function   (* tri générique, polymorphe, d'une liste *)
  | []   -> []
  | [e]  -> [e]
  | e::r -> let l1,l2 = partition ordre e r in
            (tri_rapide_list ordre l1) @ (e::(tri_rapide_list ordre l2))
and    partition ordre e = function
  | []   -> ([],[])
  | d::r -> let l1,l2 = partition ordre e r in
            if ordre d e then (d::l1, l2) else (l1, d::l2);;
(* tri_rapide_list : ('a -> 'a -> bool) -> 'a list -> 'a list = <fun>           *)
(* partition : ('a -> 'a -> bool) -> 'a -> 'a list -> 'a list * 'a list = <fun> *)
```

2. Spécialisations non polymorphes (avec variables non nommées : type faible) :

   Le type exact des variables de type faible sera fixé au premier appel ...

```
let tri_rapide_np_list_croiss   = tri_rapide_list_gen ( prefix < )
and tri_rapide_np_list_décroiss = tri_rapide_list_gen ( prefix > );;
(* tri_rapide_np_list_croiss :   '_a list -> '_a list = <fun> *)      (* type faible *)
(* tri_rapide_np_list_décroiss : '_a list -> '_a list = <fun> *)      (* type faible *)

tri_rapide_np_list_croiss [1;5;4;12;0];;            (* int list = [0; 1; 4; 5; 12]  *)
tri_rapide_np_list_croiss;;                         (* int list -> int list = <fun> *)
tri_rapide_np_list_croiss [1.0;5.0;4.0;12.0;0.0];;  (* -> erreur de type            *)
```

3. Spécialisations polymorphes (variables nommées : type polymorphe) :

```
let tri_rapide_p_list_croiss l   = tri_rapide_list_gén ( prefix < ) l (* polymorphe *)
and tri_rapide_p_list_décroiss l = tri_rapide_list_gén ( prefix > ) l (* polymorphe *)  ;;
(* tri_rapide_p_list_croiss   : 'a list -> 'a list = <fun> *)
(* tri_rapide_p_list_décroiss : 'a list -> 'a list = <fun> *)

tri_rapide_p_list_croiss [1;5;4;12;0];;                  (*[0;1;4;5;12] *)
tri_rapide_p_list_croiss [1.0;5.0;4.0;12.0;0.0;3.0;2.0];;(*[0.0;1.0;2.0;3.0;4.0;5.0;12.0]*)
```

4 Définition d'opérateurs (exemple)

```
max_int;;  (* - : int = 1073741823 *)          max_int + max_int;;  (* - : int = -2 *)

let add_intB = fun
    _ y when y = max_int -> max_int
  | x _ when x = max_int -> max_int
  | x y                  -> x + y  ;;  (* add_intB : int -> int -> int = <fun> *)

add_intB 2 3;; (*  - : int = 5 *)
add_intB max_int 3;; add_intB 2 max_int;; add_intB max_int max_int;; (* - : int = 1073741823 *)

let prefix +! = add_intB;;            (* +! : int -> int -> int = <fun> *)

2 +! 3;;  (* - : int = 5 *)    max_int +! 2;;  max_int +! max_int;;  (* - : int = 1073741823 *)
```

5 Utilisation de bibliothèques en Caml Light

5.1 Bibliothèques du système Caml

5.1.1 Exemple : utilisation de la bibliothèque graphique usuelle

```
#open "graphics";;  (* bibliothèque graphique de Caml *)

type 'a arbre =  F                                  (* Feuille, sans valeur associée      *)
               | N of 'a arbre * 'a * 'a arbre      (* valeur du père entre les deux fils *)
;;
let graph_dx = (2 * fst (text_size "0") )/2 and graph_dy = 2 * snd (text_size "0")
;;
let rec ecart = function
  | F          -> 1
  | N(g, _, d) -> (ecart g) + (ecart d) ;;      (* ecart : 'a arbre -> int = <fun> *)

let dessine_arbre sp t =                                    (* dessin modeste ... *)
     let  x = (size_x () /2) and y = (size_y () - graph_dy)
     in clear_graph (); moveto x y; dessine_noeud x y t
   where rec dessine_noeud x y = function
     | F          -> () (* lineto x y *)
     | N(g, p, d) ->
         let zg = max 1 (ecart g) and zd = max 1 (ecart d ) in
         lineto x y; dessine_noeud (x - zd * graph_dx) (y - graph_dy) g;
         moveto x y; dessine_noeud (x + zg * graph_dx) (y - graph_dy) d;
         let s = sp p in let (u,v) = text_size s in moveto (x-u/2) (y-v/2); draw_string s
;;
(* dessine_arbre : ('a -> string) -> 'a arbre -> unit = <fun> *)

let a = N( N( N(F,(1,1),F),(2,2),N(F,(3,3),F)),  (4,4),  N( N(F,(5,5),F),(6,6),N(F,(7,7),F)))
in dessine_arbre (function (x,y) -> string_of_int x) a;;
```

5.1.2 Exemple : utilisation de la bibliothèque d'éditions formatées standard

```
#open "printf";;   (* bibliothèque standard de Caml *)

printf "entier: %4d et float: %7.4f" (-12) 12.25;;              (* output on std_out *)
printf "floats en notation exp: %8.4e ; %12.6e" 12.25 1356.07;;
printf "booléen %b" (3=1+2);;                                   (* booléen true- : unit = () *)

let s = sprintf "entier: %4d et float: %7.4f" (-12) 12.25;;     (* output dans une chaîne *)
```

5.2 Bibliothèques définies par l'utilisateur

5.2.1 Exemple : utilisation des bibliothèques construites pour les TP Automates n 1,2,3,4

Bibliothèques `AutomBk` ($k = 1,2,3,4$)

AutomBk.mli, AutomBk.zi	Interface : source, compilé ;
AutomBk.zo	Objet : compilé ;
AutomBib.hlp	fichier d'aide.

dans le dossier `"F:/InfoMP/TP-info/Automats/Bibli";;`

```
#directory "F:/MP/TP-info/Automats/Bibli";;  (* Chemin de la bibliothèque *)
load_object "AutomB1";;                      (* nom primaire des fichiers de la bibliothèque *)
#open "AutomB1";;
....
```

5.2.2 Exemple : Bibliothèque Grapharb (.../InfoMP/Cours/Arbres/Dessin-Arbres/)

Dessins améliorés d'arbres binaires avec auto-positionnement ...

6 Construction d'une bibliothèque utilisateur pour Caml Light

Exemple élémentaire : Bibliothèque BibCaml

BibCaml0.ml	Fichier initial (éventuel) ;
BibCaml.mli, BibCaml.zi	Interface : fichiers source, compilé ;
BibCaml.ml, BibCaml.zo	Objet : fichiers source, compilé ;
TBibCaml.ml	Tests ... .

dans le dossier `"F:/InfoMP/BibCamlD";;`

1. Facultatif : écrire le fichier BibCaml0.ml (format texte), autonome ; comportant les définitions, les fonctions, des tests. Ce fichier sera ensuite découpé en fichiers interface, objet, test.

   La compilation de BibCaml0.ml permet de relever explicitement la description des fonctions que l'on veut exporter.

2. Ecrire le fichier d'interface (BibCaml.mli) (format texte), comportant les éléments que l'on veut inclure dans l'interface :
   - Type public (on reprend tout le type, que l'on exclus du fichier .ml)
   - Type privé (abstrait) : simple déclaration du nom du type (déclaré totalement dans l'objet)
   - Déclarations de fonctions : Value + description , sans le `= <fun>;`

```
(* BibCaml.mli : Fichier source de l'interface de la bibliothèque BibCaml *)

type 'a arbre =  F                                  (* Feuille, sans valeur associée      *)
               | N of 'a arbre * 'a * 'a arbre      (* valeur du père entre les deux fils *)
;;
(* type 'a arbre;;     (* pour un type abstrait déclaré comme ci-dessus dans l'objet *) *)
value dessine_arbre : ('a -> string) -> 'a arbre -> unit;;
value dessine_int_arbre : int arbre -> unit;;
(* Fin BibCaml.mli *)!
```

3. Ecrire le fichier objet BibCaml.ml. Par exemple par transformation de BibCaml0.ml en BibCaml.ml :
   - Supprimer les éléments de BibCaml0.ml qui ont été complètement inclus dans l'interface BibCaml.mli (on pourrait aussi les mettre entre commentaires).
   - Supprimer les tests inclus dans BibCaml0.ml (les transporter dans un fichier de test).

```
(* BibCaml.ml : Fichier source de l'objet de la bibliothèque BibCaml *)
#open "graphics";;                          (* Bibliothèque usuelle de Caml *)
let graph_dx = (2 * fst (text_size "0") )/2   (* privé, non connu à l'extérieur *)
and graph_dy = 2 * snd (text_size "0")        (* privé, ... *)
;;
let rec ecart = function                      (* privé, ... *)
  | F          -> 1
  | N(g, _, d) -> (ecart g) + (ecart d);;     (* ecart : 'a arbre -> int = <fun> *)

let dessine_arbre sp t =          (* dessin modeste ... *)       (* déclaré public *)
     let  x = (size_x () /2) and y = (size_y () - graph_dy)
     in clear_graph ();
     moveto x y; dessine_noeud x y t
   where rec dessine_noeud x y = function
     | F          -> () (* lineto x y *)
     | N(g, p, d) ->
         let zg = max 1 (ecart g) and zd = max 1 (ecart d ) in
         lineto x y; dessine_noeud (x - zd * graph_dx) (y - graph_dy) g;
         moveto x y; dessine_noeud (x + zg * graph_dx) (y - graph_dy) d;
         let s = sp p in let (u,v) = text_size s in moveto (x-u/2) (y-v/2); draw_string s
;;
let dessine_int_arbre t = dessine_arbre string_of_int t        (* déclaré public *)
;;
(* Fin de BibCaml.ml *)
```

4. (Re-)compilation { dans la fenêtre CamlWin, par "file compile".
   dans une fenêtre Dos, par `camlc -c fichier.ml`
   (a) Compiler en premier le fichier interface BibCaml.mli ce qui (re-)crée BibCaml.zi
   (b) Compiler ensuite le fichier objet BibCaml.ml. Puisque BibCaml.zi existe, il ne sera pas re-créé et ses définitions sont utilisées pour la compilation et la (re-)création de BibCaml.zo
5. Ecrire un fichier de test, TBibCalm.ml, et tester (dans CamlWin).

```
(* TBibCaml.ml : Test de la bibliothèque BibCaml *)
#directory "F:/InfoMP/BibCamlD";;   (* Chemin de la bibliothèque BibCaml *)
load_object "BibCaml";;             (* Maj/minuscules significatives !   *)
#open "BigCaml";;
let a = N(N(N(F,(1,1),F),(2,2),N(F,(3,3),F)), (4,4), N(N(F,(5,5),F),(6,6),N(F,(7,7),F)))
in dessine_arbre (function (x,y) -> (string_of_int x)^(string_of_int y) ) a;;
let b = N( N( N(F,1,F),2,N(F,3,F)),  4,   N( N(F,5,F),6,N(F,7,F)))
in dessine_int_arbre b;;
(* Fin de BibCaml0.ml *)
```

6. Ne pas oublier de faire une documentation, par exemple un fichier d'aide, avec des exemples simples.

7 Production d'un programme exécutable (.exe) en Caml light

Production d'un exécutable DOS, sans pouvoir utiliser de fenêtre graphique ... (pour cela, passer à OCaml !).

```
(* ASCIICHR.ml : Edition à l'écran du code ASCII de caractères saisis au clavier *)
exception Fini;;  (* exception définie par l'utilisateur *)

let rec vider = function     (* let rec vide flux = match flux with *)
  |[< ' '\n' >]   -> ()
  |[< 'c ;  r >]  -> vider r
  |[< >]          -> ()
;;   (* vider : 'a stream -> unit = <fun> *)

let rec traite flux = match flux with     (* terminaison dès le premier A, reste ignoré *)
  |[< ' 'A' ; r >] -> print_int (int_of_char 'A'); print_string " "; vider r; raise Fini
  |[< ' '\n' >]    -> ()                  (* retour chariot  *)
  |[< 'c ; r >]    -> print_int (int_of_char c); print_string " "; traite r
  |[< >]           -> raise Fini
;;   (* traite : char stream -> unit = <fun> *)

let run () = let flux_d'entree = stream_of_channel std_in in
   try while true                          (* boucle infinie terminée par exception *)
       do print_string "?"; flush std_out;
         try traite flux_d'entree; print_newline ();
         with | Parse_error -> print_string "Erreur de syntaxe"; print_newline ()
              | Failure s   -> print_string ( "Erreur: " ^ s ); print_newline ()
       done
   with Fini -> print_string "Fin du programme !"; print_newline ()
;;   (* run : unit -> unit = <fun> *)

run ();;
(* Fin de ASCIICHR.ml *)
```

Compilation : `F:\InfoMP\ASCIICHRdir> camlc -o ASCIICHR.exe ASCIICHR.ml`

(production des fichiers ASCIICHR.zi et ASCIICHR.zo intermédiaires puis du fichier ASCIICHR.exe).

8 Utilisation de fichiers en Caml Light

8.1 Exemple élémentaire : fichiers de caractères

```
let dir_name = "F:/InfoMP/Fich";;

let rec lire i =
   try let a = input_char i in a :: (lire i)
   with End_of_file -> []
;;
(* lire : in_channel -> char list = <fun> *)

let ch_in = dir_name ^ "/test-1.txt";;
let Myfile_in = open_in ch_in;;
string_of_char_list (lire Myfile_in);;
close_in Myfile_in;;
(*--------------*)

let rec ecrire o = function
    | []   -> ()
    | a::q -> output_char o a; ecrire o q
;;
(* ecrire : out_channel -> char list -> unit = <fun> *)

let ch_out = dir_name ^ "/test-2.txt";;
let Myfile_out = open_out ch_out;;
let u = char_list_of_string "Ceci sera le contenu du fichier test-2.txt.";;
ecrire Myfile_out u;;
close_out Myfile_out;;
```

8.2 Exemples

Voir la fin du TP Bitmap (compression d'images .bmp)

$< \mathscr{F}\mathscr{I}\mathscr{N} >$