Lycée Thiers **Année 2014-15**

Sup MPSI **Option Informatique**

CAML LIGHT 0 - PREMIER CONTACT

S'initier au langage de programmation CAML Light, et découvrir les principaux aspects de sa syntaxe au travers d'exemples simples : tel est l'objectif du présent document et de ceux qui le suivent. La consultation d'ouvrages de référence est vivement encouragée pour étudier plus avant ce langage (cf. dernière section). Par ailleurs, la lecture du présent texte doit être accompagnée de l'expérimentation sur machine : ce n'est qu'en essayant les exemples proposés, en explorant des variantes de ceux-ci et en observant les réponses du système (et notamment les messages d'erreurs ...) qu'il sera possible d'assimiler les rudiments du langage.

On peut obtenir le présent texte (et d'autres à venir) sur le net, à l'adresse suivante :

http://info-mpsi.weebly.com

On y trouvera aussi un aide-mémoire, écrit par Michel QUERCIA, professeur au lycée Carnot (Dijon).

1. L'interface

Bien que CAML Light ne soit plus maintenu depuis début 2002, on considère que le niveau de stabilité atteint est très satisfaisant. Il ne faut, par ailleurs, pas perdre de vue qu'il s'agit d'un langage destiné à l'apprentissage de certains aspects fondamentaux de l'algorithmique et de la programmation, et en aucun cas au développement d'applications professionnelles. Pour ce dernier objectif, on peut se tourner vers **Objective Caml** (Ocaml pour les intimes), dont le développement reste à ce jour très actif.

A la base, CAML Light est disponible à l'adresse suivante :

http://caml.inria.fr/caml-light/release.fr.html

Mais depuis quelques années, on voir apparaître sur le net des solutions plus ergonomiques, offrant des fonctionnalités appréciables (comme la coloration syntaxique, par exemple).

Ces divers logiciels permettent de programmer avec CAML Light en mode interactif : chaque phrase tapée par l'utilisateur est typée, compilée, évaluée, puis le résultat de cette évaluation est affiché.

Deux fenêtres sont généralement regroupées dans une même interface :

- Entrée (où l'on tape des phrases en CAML)
- Sortie (où apparaissent les résultats des évaluations)

Une troisième fenêtre, dédiée au graphisme, peut éventuellement s'ajouter aux précédentes (le mode graphique sera présenté dans un document ultérieur).

Pour Windows comme pour MacOSX, on pourra utiliser l'interface (WinCaml ou MacCaml, selon le cas) développée par Jean Mouric et téléchargeable à l'adresse suivante :

http://jean.mouric.pagesperso-orange.fr/

2. Expressions et Types

2.1. **Premières phrases en CAML.** Une phrase peut occuper plusieurs lignes ; elle se termine par un **double** point-virgule. Attention : pour aller à la ligne suivante, on presse la touche *Return* ; tandis que pour déclencher l'évaluation de la phrase, on presse sur la touche *Enter* (ou bien *Control-Return*, ou encore (sur Mac) *Command-Return*).

```
2005+1;;
- : int = 2006
7*11*13;;
- : int = 1001
91/7;;
- : int = 13

355. /. 113.;;
- : float = 3.14159292035

355 / 113;;
- : int = 3
355 mod 113;;
- : int = 16

2. ** 10.;;
- : float = 1024.0

1 = 2;;
- : bool = false
77*13 = 11 * 91;;
- : bool = true
(1 = 2) or (3 < 4);;
- : bool : true
(1 = 2) & (3 < 4);;
- : bool = false
not (3 <> 4);;
- : bool = false

"Ceci est une ";;
- : string = "Ceci est une "
"Ceci est une " ^ "phrase complète.";;
- : string = "Ceci est une phrase complète."
```

Le résultat de l'évaluation de chaque phrase comporte trois indications :
– le nom de la variable globale qui vient d'être définie (voir section 3 ci-après) ou, à défaut, un simple tiret.
– le type du résultat (voir section 2.3 ci-après).
– la valeur du résultat.

2.2. **Commentaires.** Il est conseillé de placer des **commentaires** dans le code source : un commentaire consiste en du texte, ignoré par le système. Ce texte est donc exclusivement destiné à l'humain qui le lira. L'emploi de commentaires rend les sources d'un programme lisibles par un tiers (qui, bien souvent, est l'auteur ... quelques temps plus tard).

Un commentaire se doit d'être concis, et ne consiste pas en un "plagiat" du code source : il doit éclairer ce dernier, préciser les sous-entendus ainsi que le principe de la méthode employée, donner éventuellement des références ...

Les délimiteurs de commentaires sont `(*` et `*)`.

Le premier exemple de commentaire apparaît à la section 2.4.1, ci-dessous.

2.3. Qu'est-ce qu'un "type" ?

En CAML, la valeur de chaque expression appartient à un ensemble bien déterminé, appelé **type de l'expression**.

Les types de base sont : **`int`** (entiers), **`float`** (floating point numbers = nombres à virgule flottante. Ce sont des nombres décimaux [1]), **`bool`** (vrai / faux), **`char`** (caractère), **`string`** (chaine de caractères). D'autres seront présentés plus loin, au fur et à mesure des besoins ...

CAML est doté d'un **moteur d'inférence de type** : c'est au système (et non pas au programmeur) de déterminer les types des expressions (contrairement au langage C, par exemple). Ainsi, lorsque l'expression `x + y` est rencontrée, CAML déduit de la présence de l'opérateur '+' que `x` et `y` sont de type `int`.

Aucun "mélange des genres" n'est possible ! Exemple :

```
1 / 7.;;
Toplevel input:
>1 / 7.;;
>    ^^
This expression has type float,
but is used with type int.
```

ou encore :

```
1 /. 7;;
Toplevel input:
>1 /. 7;;
>^ This expression has type int,
but is used with type float.
```

2.4. Types numériques : d'inévitables imperfections.

2.4.1. *Débordement.* Concernant les types numériques (`int` et `float`), les nombres représentables forment des ensembles finis et donc bornés. Il y a donc risque de dépassement de capacité (*overflow* en anglais). Voici ce qui se passe avec une architecture 32 bits :

```
1000*1000*1000;;
- : int = 1000000000

1000*1000*1000*1000;; (* résultat correct : 1000000000000 *)
- : int = -727379968
```

Ce résultat, assez déroutant à priori, est lié à la représentation interne des nombres entiers. Cette question sera étudiée en classe. Retenons déjà que le type `int` est codé sur 31 bits ce qui correspond à l'intervalle suivant :

$$[|-2^{30},\ 2^{30}-1|] = [|-1\,073\,741\,824,\ 1\,073\,741\,823|]$$

Avec une architecture 64 bits, les calculs ci-dessus se font correctement car l'intervalle de validité devient $\left[\left|-2^{62},\ 2^{62}-1\right|\right]$; mais :

```
10000000000*10000000000;
- : int = -1457092405402533888
```

Pour le type `float`, les messages du système sont toutefois plus explicites :

```
100. ** 100.;;
- : float = 1e+200

(100. ** 100.) ** 100.;;
- : float = inf.0
```

1. Ils sont codés conformément à la norme ANSI / IEEE 745.

2.4.2. *Arrondis.* Tant qu'aucun dépassement de capacité n'intervient, les calculs effectués avec le type `int` sont exacts. Il n'en va pas de même pour le type `float` :

```
10000. +. 0.0001;;
- : float = 10000.0001

10000.0001 -. 10000.;; (* résultat correct = 0.0001 *)
- : float = 9.99999992928e-05
```

ou encore :

```
1. -. 3. *. (4. /. 3. -. 1.);; (* résultat correct = 0 *)
- : float = 2.22044604925e-16
```

Là encore, cette anomalie apparente est due au codage interne des nombres de type `float`. Une solution partielle, si l'on veut calculer sans perte de précision, consiste à utiliser le type `num` (voir le 4ème tome du poly).

2.5. **Multiplets.** Un multiplet est un expression dont le type est le produit cartésien de plusieurs types.

Par exemple :

```
"Hugo","Victor";;
- : string * string = "Hugo", "Victor"

1,3.14;;
- : int * float = 1, 3.14

1,2,3,4;;
- : int * int * int * int = 1, 2, 3, 4

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

La virgule ',' est le *constructeur* de multiplet.

Des parenthèses peuvent être utilisées pour améliorer la lisibilité ou éviter une ambiguïté :

```
(1,2,3,4);;
- : int * int * int * int = 1, 2, 3, 4
```

3. Variables et Liaisons

```
let a = 12;;
a : int = 12
```

La **variable** a est **globale**.

Une expression du type `let identificateur = expr` est appelée une **liaison globale**. La valeur de l'expression `expr`, au moment ou elle est évaluée, est liée à l'identificateur (on parle de liaison statique) :

```
let b = 2 * a;;
b : int = 24

let a = 0;;
a : int = 0

b;;
- : int = 24
```

A moins de la redéfinir (par une phrase du genre `let a = 456;;`), il n'est pas possible de modifier la valeur de a [2]. Les variables de CAML correspondent aux constantes d'un langage comme PASCAL. L'analogue d'une variable au sens du PASCAL (c'est-à-dire modifiable) existe en CAML, c'est une **référence** (voir le tome 1 du poly).

Le système prédéfinit un certain nombre de variables globales. Par exemple le plus petit entier représentable et le plus grand :

```
min_int;;
- : int = -1073741824

max_int;;
- : int = 1073741823
```

La portée d'une variable peut être limitée ; on dit alors qu'elle est **locale** :

```
let b = 1 in
  a + b;;
- : int = 124

b;;
Toplevel input:
>b;;
>^
The value identifier b is unbound.
```

(unbound = non lié)

La construction `let ... in ...` est appelée une **liaison locale**.

Dans l'exemple ci-dessus, on constate que la portée de la variable `b` est limitée à l'expression `a+b`.

Hors du contexte où elle est définie, une variable locale n'a donc pas d'existence. Mieux : si une variable de même nom existe dans le contexte englobant, celle-ci devient *temporairement* inaccessible :

```
let a = 5;;
a : int = 5

let a = 6 in a+1;;
- : int = 7

a;;
- : int = 5
```

Les liaisons locales peuvent être emboîtées :

```
let x = 3 in
  let y = 4 in
   let z = 5 in x*y*z;;
- : int = 60
```

Des liaisons (locales ou non) peuvent être simultanées. On utilise pour cela le mot-clef `and` (à ne pas confondre avec l'opérateur booléen `&`) :

```
let x = 3 and y = 4 and z = 5 in x*y*z;;
- : int = 60
```

Attention, deux liaisons connectées par `and` sont, en quelque sorte, parallèles :

2. Des variables qui ne varient pas ! La terminologie n'est donc pas idéale ...

```
let x = 3;;
x : int = 3

let x = x + 1 and y = x+1 in x*y;; (* liaisons simultanées *)
- : int = 16

let x = x+1 in
  let y = x+1 in x*y;; (* liaisons emboîtées *)
- : int = 20
```

Signalons pour finir qu'il existe une alternative à la syntaxe `let variable = valeur in expr`, à savoir :

`expr` **`where`** `variable = valeur`

Par exemple :

```
3*(x+y) where x = 10 and y = 1;;
- : int = 33
```

En cas d'imbrication de plusieurs liaisons, cette syntaxe est cependant déconseillée car moins lisible.

4. Fonctions

Une fonction en CAML correspond assez bien à l'idée que l'on se fait d'une fonction en mathématiques. Elle accepte un (ou plusieurs) argument(s) et renvoie un résultat. Les arguments et le résultat appartiennent tous à des types bien déterminés ; d'ailleurs, lors de la définition d'une fonction, CAML nous informe du type de celle-ci.

4.1. Ma première fonction ...

```
let deux_fois = function x -> 2*x;;
deux_fois : int -> int = <fun>
```

ou, syntaxiquement plus léger mais qui revient exactement au même :

```
let deux_fois x = 2*x;;
deux_fois : int -> int = <fun>
```

Le mot-clef `let` a déjà été rencontré à la section 3 : il introduit la définition de la fonction.

Le système évalue la phrase et répond que `deux_fois` est une fonction [3] qui, à un entier, associe un entier. C'est le sens de la notation `int -> int`.

Essayons :

```
deux_fois (10 + 1);;
- : int = 22
```

L'expression `10+1` est évaluée, puis sa valeur (= 11) est substituée au paramètre formel `x`, qui lui correspond dans la définition de la fonction.

L'emploi des parenthèses est optionnel, à condition de respecter une règle : en CAML, l'application d'une fonction, qui est notée par simple juxtaposition, possède la plus forte priorité :

```
deux_fois 11;;
- : int = 22

deux_fois (10 + 1);;
- : int = 22

deux_fois 10 + 1;;
- : int = 21
```

3. fun comme function.

```
deux_fois -2;;
Toplevel input:
>deux_fois -2;;
>^^^^^^^^^
This expression has type int -> int,
but is used with type int.

#deux_fois (-2);;
- : int = -4
```

D'une manière générale, l'expression `x y z` désigne `(x y) z`.

Pour obtenir `x (y z)`, il faut le demander explicitement :

```
let u x = x + 1;;
u : int -> int = <fun>

let v x = x * x;;
v : int -> int = <fun>

u v 3;;
Toplevel input:
>u v 3;;
>  ^ This expression has type int -> int,
but is used with type int.

u(v 3);;
- : int = 10
```

Signalons enfin qu'il n'est pas nécessaire de **nommer** une fonction pour s'en servir :

```
(function x -> 2*x) 5;;
- : int = 10
```

4.2. **Le type `unit`.** Une fonction peut très bien n'agir que par **effet de bord**, et ne retourner aucun résultat (c'est l'analogue d'une procédure du PASCAL). Par exemple, `print_int` [4] affiche l'entier qui lui est passé en argument :

```
print_int (1+2+3+4);;
10- : unit = ()
```

Pour des raisons de cohérences, une telle fonction renvoie tout de même un résultat. Le type de ce résultat est `unit` ; il ne comporte qu'une seule valeur, notée `()`. Le type `unit` est l'analogue du type *void* du C ; on peut considérer que c'est le "rien" ....

Un autre exemple est la fonction `trace`, qui permet de suivre pas à pas le déroulement des appels d'une fonction donnée (utile, paraît-il, pour debugger un programme ...). Nous en reparlerons plus loin.

Certaines fonctions, quant à elles, n'acceptent aucun argument ; voici un exemple stupide :

```
let f () = 1;;
f : unit -> int = <fun>

f();;
- : int = 1
```

et un autre, plus utile, qui mesure le temps écoulé depuis le début de la session en cours [5]

4. Et de même : `print_float`, `print_string`, `print_char`.
5. ou bien, sous Linux et MacOS X, le temps CPU.

```
sys__time ();;
- : float = 26.59
```

Noter l'emploi du **double** trait de soulignement : `time` est une fonction implémentée dans le module `sys` ; on y accède soit par `sys__time`, soit simplement par `time`, à condition d'avoir d'abord chargé le module par la directive `#open` (pour refermer un module, on utilise la directive #close).

D'autres fonctions, enfin, n'acceptent aucun argument et ne renvoient rien (mais n'en sont pas inutiles pour autant : elles agissent par "effet de bord") ; c'est le cas de la fonction `print_newline` :

```
print_int 2;;
2- : unit = ()

print_int 2; print_newline ();;
2
- : unit = ()

print_newline;;
- : unit -> unit = <fun>
```

4.3. **Fonctions de conversion.** Il est souvent utile de convertir une expression d'un certain type vers son équivalent dans un autre type (lorsque cette conversion a un sens).

• Entre entiers et flottants : `int_of_float` et `float_of_int` sont (presque) réciproques l'une de l'autre ; la première accepte un argument de type `float` et renvoie l'entier correspondant (obtenu par troncature) ; la seconde fait l'inverse (sans perte de précision cette fois) :

```
int_of_float 3.56;;
- : int = 3

float_of_int 3;;
- : float = 3.0
```

• Entre entiers et chaines de caractères (^ est l'opérateur de concaténation des chaines) :

```
int_of_string ("12345");;
- : int = 12345

"3 fois 8 font " ^ string_of_int (3*8);;
- : string = "3 fois 8 font 24"
```

Bien sûr, il ne faut pas faire n'importe quoi :

```
int_of_string "3a8@!!";;
Uncaught exception: Failure "int_of_string"
```

• Entre flottants et chaines de caractères :

```
float_of_string "2.71828";;
- : float = 2.71828

"Pi est peu different de " ^ string_of_float (4.0 *. atan (1.0));;
- : string = "Pi est peu different de 3.14159265359"
```

Remarque. Les fonctions `int_of_string` et `float_of_string` sont utiles, par exemple, lorsqu'un programme se déroule de façon interactive : l'utilisateur est, à un moment donné, invité à taper une chaine de caractères, d'où seront ensuite extraites diverses sous-chaines. Certaines pourront être converties en entiers, en flottants, etc ...

• Entre entiers et caractères :

Un caractère est délimité par une paire d'accents graves (backquotes, Alt Gr 7 sur clavier français).

```
int_of_char ('A');; (* renvoie le code ASCII du caractère 'A' *)
- : int = 65

char_of_int (97);; (* renvoie le caractère dont le code ASCII est 97 *)
- : char = 'a'
```

4.4. **Fonctions mathématiques.** Le module `float`, qui est chargée au lancement du système, prédéfinit notamment diverses fonctions mathématiques d'usage courant. Voici les principales, illustrées à l'aide d'exemples :

```
sqrt (144.);; (* square root = racine carrée *)
- : float = 12.0

let e = exp (1.);;
e : float = 2.71828182846
log (e *. e);; (* ATTENTION : log et non pas ln *)
- : float = 2.0

let pi = 4. *. atan(1.);;
pi : float = 3.14159265359
sin(pi /. 2.);;
- : float = 1.0
cos(pi /. 3.);;
- : float = 0.5

floor (-2.1);; (* partie entière par défaut. floor = plancher *)
- : float = -3.0
ceil (-2.1);; (* partie entière par excès. ceil = plafond *)
- : float = -2.0
```

4.5. **Fonctions de sélection des composantes d'un couple.**

```
let jumeaux = (17,19);;
jumeaux : int * int = 17, 19
```

Les fonctions prédéfinies `fst` (first) et `snd` (second) renvoie respectivement la première et la seconde composante d'un couple.

```
fst jumeaux;;
- : int = 17

snd jumeaux;;
- : int = 19
```

Noter que ces deux fonctions s'appliquent à deux couples de types quelconques :

```
let prénom_age = ("Gustave", 21);;
prénom_age : string * int = "Gustave", 21

fst prénom_age;;
- : string = "Gustave"

snd prénom_age;;
- : int = 21
```

On dit que `fst` et `snd` sont **polymorphes** (ce sujet sera développé dans un document ultérieur).

5. Plus sur les fonctions ...

5.1. **Fonctionnelles.** On nomme fonctionnelle toute fonction qui renvoie comme résultat ... une fonction.

Par exemple :

```
let f a = function b -> 1 + a*b;;
f : int -> int -> int = <fun>
```

ou, ce qui revient exactement au même :

```
let f a b = 1 + a*b;;
f : int -> int -> int = <fun>
```

`f` est une fonction qui, à un entier, associe une fonction qui, à un entier, associe un entier (ouf). C'est précisément ce qu'indique le type `int -> int -> int`, qu'il faut lire `int -> (int -> int)`.

Observons que cette associativité à droite de -> est en cohérence avec l'associativité à gauche de l'application des fonctions :

Puisque `f a b` désigne `(f a) b`, il est clair que `f a` doit désigner une fonction de type `int -> int` et que `f` est du type `int -> (int -> int)`.

Par exemple :

```
(f(4))(3);;
- : int = 13

f(4)(3);;
- : int = 13

f 4 3;;
- : int = 13

f 4;;
- : int -> int = <fun>

let g = f 4;;
g : int -> int = <fun>

g 3;;
- : int = 13
```

5.2. **Curryfication.** Reprenons la fonction `f` de la section 5.1 :

```
let f a b = 1 + a*b;;
f : int -> int -> int = <fun>
```

et comparons avec :

```
let F (a,b) = 1 + a*b;;
F : int * int -> int = <fun>
```

`F` s'applique à un seul argument, de type `int * int` ; alors que `f` accepte deux arguments, chacun de type `int`.

Comme on l'a observé à la section 5.1, `f a` désigne une fonction : à savoir une fonction partielle de `F`, (celle qui à `b` associe `1+a*b`).

On dit que f est la version **curryfiée** de F. Le procédé par lequel on passe de F à f s'appelle la **curryfication**. Ces termes sont construits d'après le nom de Haskell Brooks Curry (1900 - 1982), mathématicien et logicien américain à qui l'on doit, entre autres choses, ce concept.

L'intérêt des fonctions curryfiées, outre le fait qu'elles sont **applicables partiellement**, est qu'elle sont en général **plus rapides**.

Ainsi, étant donnés trois ensembles E, F, G, l'idée sous-jacente à la curryfication / décurryfication est la bijection naturelle existant entre les ensembles $\left(G^F\right)^E$ et $G^{E\times F}$. Cette bijection est :

$$\left(G^F\right)^E \to G^{E\times F},\ u \mapsto [(e,f) \mapsto (u\,(e))\,(f)]$$

et sa réciproque est :

$$G^{E\times F} \to \left(G^F\right)^E,\ v \mapsto [e \mapsto [f \mapsto v\,(e,f)]]$$

5.3. **Opérateurs binaires.** En CAML, l'application d'une fonction utilise une notation préfixe (le nom de la fonction précède les arguments).

A toute fonction **d'arité** 2 (c'est-à-dire acceptant deux arguments), on peut associer un opérateur binaire infixe (dont le nom est placé entre les arguments) et vice versa. Parmi les opérateurs binaires prédéfinis, on trouve :

`+ - * / mod` (addition, soustraction, multiplication, quotient, reste) : pour le type `int`,

`+. -. *. /. **` (addition, soustraction, multiplication, division, exponentiation) : pour le type `float`,

`& or` (disjonction "ou", conjonction "et") : pour le type `bool` (`not` est un opérateur unaire),

`= <> < > <= >=` (égalité et inégalité [6], comparaisons strictes ou larges) : polymorphes.

et d'autres (comme par exemple `:=`) qui seront rencontrés ultérieurement.

Il est possible, à partir d'une fonction d'arité 2, de transformer son nom en un symbole d'opérateur infixe.

Par exemple, définissons une fonction "implique" :

```
let implique p q = (not p) || q;;
implique : bool -> bool -> bool = <fun>
implique true false;;
- : bool = false
```

puis, utilisons la directive [7] `#infix` (il faut taper le #) :

```
#infix "implique";;
```

On peut alors se servir de l'opérateur "implique" :

```
true implique true;;
- : bool = true
true implique false;;
- : bool = false
false implique true;;
- : bool = true
false implique false;;
- : bool = true
```

La transformation effectuée avec la directive `#infix` est réversible :

```
uninfix "implique";;
```

La fonction curryfiée, associée à un opérateur binaire infixe, est accessible via le mot-clef `prefix` :

6. structurelles ... l'(in)égalité physique est notée == (!=).
7. Il existe d'autres directives, notamment #open et #close, évoquées plus haut.

```
let plus = prefix +;;
plus : int -> int -> int = <fun>

plus 4 5;;
- : int = 9

let ajoute_un = plus 1;;
ajoute_un : int -> int = <fun>

ajoute_un 5;;
- : int = 6
```

Mais on peut aussi définir directement des opérateurs par l'intermédiaire de leur fonction associée [8] :

```
let prefix --> p q = (not p) || q;;
--> : bool -> bool -> bool = <fun>

true --> false;;
- : bool = false
```

6. Quelques références pour se documenter

L'nternet donne accès à beaucoup de documentation sur le langage CAML. On pourra consulter les liens suivants (liste non exhaustive !) :

• La page de CAML à l'INRIA :

http://caml.inria.fr/

et, plus particulièrement, celle consacrée à CAML Light (en anglais) :

http://caml.inria.fr/caml-light/index.fr.html

• Une FAQ (Frequently Asked Questions, ou Foire Aux Questions, selon les goûts) écrite par Pierre Weis en 1995, mais toujours très instructive (en français) :

http://caml.inria.fr/pub/old_caml_site/FAQ/index-fra.html

• Le manuel de la version 0.74 de CAML Light (en anglais) :

http://caml.inria.fr/pub/docs/manual-caml-light/

• Un tutoriel écrit par Michel Mauny (en anglais) :

http://caml.inria.fr/pub/docs/fpcl/

8. C'est ainsi que tous les opérateurs de CAML sont d'ailleurs définis.