TP 1— PREMIERS PAS http://www.gallium.inria.fr/~scherer/teaching/colles/2012/tp1.pdf

Les questions en rapport avec les TP, Caml ou la programmation en général sont les bienvenues et peuvent être envoyées à gabriel.scherer@gmail.com. N'hésitez pas !

Le but de ces TPs est de vous donner, en complément du cours de votre professeur d'informatique, un peu de pratique de la programmation (plus particulièrement du langage Caml Light). Les premières séances seront consacrées à la découverte du langage.

1 INTRODUCTION

Le langage Caml Light est ce qu'on appelle un *langage fonctionnel*. Il est assez proche, dans sa conception, des mathématiques : assez formel, il repose sur la manipulation de fonctions. Il est cependant très polyvalent et intègre des concepts de nombreux langages de programmation. C'est un bon langage pour débuter.

1.1 UTILISATION DU TERMINAL

L'interaction avec le terminal Caml Light se déroule sous la forme d'un dialogue : l'utilisateur (ou utilisatrice) envoie une *phrase* caml, et le terminal répond. Quand le terminal attend une phrase, il le signale en affichant un symbole #. Pour signaler que vous avez terminé votre phrase, vous devez la conclure par un double point virgule.

Voyez l'exemple ci-contre : le terminal affiche le code qui lui a été envoyé (les lignes commençant par le symbole dièse), suivi par sa réponse (les lignes commençant par un tiret).

Les phrases envoyées ici sont toutes des *expressions* (on peut envoyer d'autres sortes de phrases, comme les déclarations que vous verrez ensuite). Quand on lui envoie une expression, le terminal répond en donnant son type et sa valeur, sous la forme suivante :
`- : <type> = <valeur>`

```
#2 + 2;;
- : int = 4
#cos 0.;;
- : float = 1.0
#2 = 3;;
- : bool = false
#vzjg3w;u21]3wqf<aepiju3[t;;
Erreur de syntaxe.
```

1.2 TYPES DES EXPRESSIONS

On ne peut effectuer une opération que sur des valeurs du bon type. Quelques exemples :

- opérations qui renvoient un `int` (nombre entier) : `+` `-` `*` `/` `mod` (modulo) `int_of_float`
- opérations qui renvoient un `float` (≃ réels) : `+.` `-.` `*.` `/.` `**` (puissance) `float_of_int`
- opérations qui renvoient un `bool` (booléen) : `<` `<=` `>` `>=` `=` `<>`
- opération qui renvoie un `string` (texte) : `^`

▷ **Question 1.** Examinez la liste de programmes donnée ci-contre: à votre avis, quel sera le type du résultat de chacun? Lesquels comportent une erreur? Pourquoi cette erreur a-t-elle lieu? Comment la corriger? ◁

```
(1 + 2) = 3;;
1 + (2 = 3);;
"Texte" + "autre texte";;
exp(1.234) < 1.5;;
2 * 3.141592;;
```

Remarque : Vous ferez très souvent l'erreur d'utiliser une opération ou une valeur du mauvais type. Surtout pour les flottants.

1.3 NOMMER DES VARIABLES

Voici une autre sorte de phrases : les *déclarations* de valeurs. Elles sont de la forme suivante (où <*expr*> désigne une expression quelconque) : `let <nom> = <expr>`

```
#let pi = 3.141592;;
pi : float = 3.141592
```

Un nom continue à désigner la valeur à laquelle il est attaché jusqu'à ce qu'une nouvelle définition le lie à une autre valeur. Cette modification n'a aucun effet sur les utilisations précédentes du nom, qui conservent leur ancienne valeur.

▷ **Question 2.** Essayer de prévoir le résultat de la suite d'instructions ci-dessous sans l'exécuter, puis vérifier les prédictions à l'aide du terminal. ◁

Il est important de bien choisir les noms donnés aux variables: `a`, `x` ou `bool` sont des noms qui n'apportent pas en général d'information pertinente à un lecteur humain. Au contraire, des noms comme `compteur`, `discriminant` ou `fini` donnent des informations sur l'algorithme utilisé. Il est en général préférable d'écrire plus pour avoir un programme plus lisible.

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

2 EXPRESSIONS

Les expressions que l'on a vues jusqu'à présent étaient simples : des constantes (`1`, `5.7`, `"blabla"`), des noms de variables, et des opérations (addition, test d'égalité, etc.). On verra ici trois sortes d'expressions plus évoluées : les fonctions, les expressions conditionnelles et les déclarations locales.

2.1 FONCTIONS

Une fonction est une expression de la forme suivante : `fun <nom> -> <expr>`

`Caml` ne connait pas de représentation textuelle pour les fonctions : à la place de leur valeur il affiche `<fun>`. Le type, par contre, est bien donné : la fonction ci-contre prend un paramètre de type `int` et renvoie une valeur de type `int`.

Pour appliquer un argument à une fonction, il suffit de l'écrire à la suite. Les fonctions étant des valeurs comme les autres, on peut aussi les nommer.

```
#fun x -> x + x;;
- :  int -> int = <fun>
#(fun x -> x * x) 4;;
- :  int = 16
#let carre = fun x -> x * x;;
carre :  int -> int = <fun>
#carre 5;;
- :  int = 25
```

▷ **Question 3.** Reprendre la fonction `carre` ci-dessus, et l'utiliser pour écrire la fonction polynômiale $p(x) = x^2 + 2x + 1$. Calculer $p(1)$ et $p(2)$. ◁

Comme les déclarations de fonctions sont très courantes dans un programme `Caml`, il existe une syntaxe spéciale qui évite de devoir écrire `let f = fun x -> ...` à chaque fois; on peut écrire à la place : `let f x = ...` Les deux phrases suivantes sont strictement équivalentes :

```
let carre = fun x -> x * x ;;
let carre x = x * x ;;
```

2.2 CONDITIONS

Les conditions sont des expressions de la forme : `if <expr> then <expr> else <expr>`

L'exemple ci-contre définit la fonction valeur absolue en utilisant la position de x par rapport à zéro.

```
let abs x = if x >= 0 then x else -x ;;
```

▷ **Question 4.** Écrire la fonction définie par:

$$f(x) = \begin{cases} x/2 & \text{si } x \text{ est pair} \\ 3x+1 & \text{si } x \text{ est impair} \end{cases}$$

◁

Remarque : L'expression conditionnelle dans son ensemble possède, comme toutes les expressions `Caml`, un type. C'est aussi le type des expressions dans les deux branches ("then" et "else") de la condition. En particulier, ces dernières doivent avoir le même type et la phrase suivante est incorrecte :

```
if 2 = 3 then 2 else "toto"
```

2.3 Définitions locales

Parfois, on veut nommer une valeur pour alléger les notations, mais on n'a pas besoin de conserver la valeur pour tout le reste du programme. On utilise alors une *déclaration locale*, qui porte seulement sur l'expression suivante, et pas tout le reste du programme :
`let <nom> = <expr> in <expr>`

```
#let truc = 3+6 * 6+3 in truc / 3+1 + truc;;
- :  int = 57
#truc;;
L'identificateur truc n'est pas défini.
```

Une déclaration locale est, prise dans son ensemble, une expression `Caml` comme les autres. En particulier, elle peut se trouver à l'intérieur d'une fonction, et on peut mettre une déclaration locale dans une déclaration locale.

```
let double x = x + x in
  1 + (let a = 1 in (let b = double a in 1 + b));;
```

C'est très utile, mais il faut prendre garde de ne pas écrire des choses complètement illisibles.

▷ **Question 5.** Écrire une fonction qui donne le n-ième terme de la suite de fibonnaci en utilisant la formule suivante :

$$\phi = \frac{1+\sqrt{5}}{2} \qquad u_n = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}}$$

◁

Attention : une déclaration locale doit être suivie par une *expression*, et non par une phrase OCaml quelconque (déclaration de valeur, de type...). Ainsi, le code ci-contre, qui définit `deux_pi`, est incorrect.

```
let double x = x + x in
  let deux_pi = double pi;;
```

▷ **Question 6.** Comment le corriger ? ◁

3 Traitement des fonctions à plusieurs paramètres

Une fonction est une valeur comme les autres. Elle peut par exemple être passée en argument d'une autre fonction, ou renvoyée comme valeur de retour. Ici, `deux_fois` prend une fonction `f` en paramètre et renvoie une autre fonction (`fun x -> ..`).

Si l'on fait un peu attention, on peut aussi voir `deux_fois` comme une fonction prend deux arguments, f et x, et renvoie $f(x)+f(x)$. On peut d'ailleurs écrire (et c'est strictement équivalent) : `let deux_fois f x = f x + f x`.

```
# let deux_fois f = (fun x -> f x + f x);;
# let augmenter_deux_fois =
    let augmenter n = n + 1 in
      deux_fois augmenter in
  augmenter_deux_fois 3 ;;
- :  int = 8
# deux_fois (fun n -> n + 1) 3 ;;
- :  int = 8
```

Cette méthode permet d'exprimer des fonctions à plusieurs arguments. On peut la généraliser à un nombre quelconque (mais fixé à l'avance) d'arguments. Elle s'appelle la *curryfication*, et correspond à l'isomorphisme, en mathématiques, entre $(A \times B) \to C$ et $A \to (B \to C)$.

Remarque : Avec les fonctions à plusieurs arguments, on voit apparaître des types de la forme `int -> int -> int`. Dans ces cas là il faut lire (la flèche est associative à droite) `int -> (int -> int)`.

▷ **Question 7.** Écrire une fonction qui renvoie la somme de trois paramètres. ◁

Notations On a déjà vu que l'on peut écrire `let f x = a` à la place de `let f = fun x -> a`. Selon le même principe, on peut simplifier l'écriture de fonctions à plusieurs paramètres :

- on peut compresser une suite de `fun`, par exemple `fun x -> fun y -> z` s'écrit aussi `fun x y -> z`.
- on peut faire rentrer plusieurs `fun` dans un `let` : `let f = fun x y -> z` s'écrit aussi `let f x y = z`.

Il faut bien comprendre que ces notations représentent exactement le même code, et savoir faire ces transformations dans les deux sens pour raisonner sur les programmes.

3.1 Application partielle

```
# let somme x y = x + y;;
# let somme' x = fun y -> x + y;;
# let somme_3 = somme' 3;;
# somme_3 2;;
- int :  5
```

Au risque de me répéter : il n'y a pas à proprement parler de "fonction à deux arguments" en Caml. Il y a par contre des fonctions qui prennent un argument, puis renvoient une fonction qui prend un deuxième argument et renvoie un résultat. En particulier, fournir un seul argument à une fonction Caml est toujours défini, vous n'aurez jamais d'erreur du style "Vous n'avez pas fourni assez d'arguments".

Par exemple, dans le code ci-contre, les fonctions `somme` et `somme'` sont exactement identiques, mais la deuxième permet mieux de comprendre ce que signifie `somme 3` : c'est la fonction qui, quand on lui donne un paramètre, lui ajoute 3. On parle dans ce cadre d'*application partielle* : on a donné seulement une partie des arguments à la fonction.

3.2 Placement des parenthèses

J'ai été assez surpris de constater en donnant des colles qu'une des principales sources d'erreurs chez les élèves est le *placement des parenthèses* en Caml. C'est regrettable, parce que ce placement obéit à des règles très simples et logiques, qu'il suffit de comprendre une fois pour savoir toujours où placer des parenthèses. La source d'erreur est le fait que le placement des parenthèses en Caml est *différent* de la notation mathématique habituelle.

Les parenthèses utilisées autour des expressions n'ont *pas de signification*. Elles servent uniquement à éliminer les ambiguités, comme en mathématiques (différence entre $1 + 2 * 3$ et $(1 + 2) * 3$).

Pour appeler une fonction `f` avec un argument `x`, on écrit simplement `f x`. Vous pouvez aussi écrire `(f x)`, `(f)x`, `f(x)` ou `(f)((x))` si cela vous fait plaisir, mais contrairement à la notation mathématique, les parenthèses autour du `x` ne sont *pas* nécessaires, et je vous déconseille de les mettre.

Le problème de la notation mathématique est son extension aux appels de fonction à plusieurs arguments, comme par exemple `g x y` : on donne à la fonction `g` l'argument `g`, puis `y`. Si l'on veut être précis, on dit que l'application de `x` à `g` renvoie une fonction (`g` est de la forme `fun a -> fun b -> c`), et que c'est à cette fonction qu'on applique y. On pourrait donc aussi écrire `let h = g x in h y` ou `(g x) y`.

```
let somme x y = x + y;;
somme
  somme somme 2 3 4
  somme 2 somme 3 4;;
```

C'est ici que la notation mathématique à laquelle vous êtes habitués risque de vous faire faire des erreurs. `g x y` est équivalent à `(g x) y`, mais **pas** à `g(x y)` ! Cette deuxième écriture signifie qu'on applique `y` à `x`, et qu'on donne le résultat à `g`. Par contre, si vous voulez écrire $f(g(x))$, c'est bien `f (g x)` et pas `f g x` !

▷ **Question 8.** Rajoutez les parenthès nécessaires pour que le code ci-dessus soit correct. ◁

4 Quelques adresses

Il est possible de télécharger `Caml Light` à l'adresse suivante:

`http://caml.inria.fr/download.fr.html`

Le manuel de référence en anglais, qui n'est pas vraiment adressé aux débutants en informatique, se trouve quant à lui sur:

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

Ce manuel contient cependant des pages qui vous seront très utiles quand vous maîtriserez un peu mieux le langage, à savoir les descriptions des bibliothèques (l'ensemble des fonctions préprogrammées dans `Caml Light`). Les principales sont :

The core library : `http://caml.inria.fr/pub/docs/manual-caml-light/node14.html`
The standard library : `http://caml.inria.fr/pub/docs/manual-caml-light/node15.html`

Le livre "Le langage Caml", qui a été récemment mis en ligne en libre accès. Écrit par deux des concepteurs du langage, c'est un excellent outil d'apprentissage, qui va au delà du niveau attendu en classes préparatoires. N'hésitez pas à y jeter un oeil si vous chercher une explication alternative à un concept que vous avez du mal à comprendre, ou des exercices d'entraînement supplémentaires :

`http://caml.inria.fr/pub/distrib/books/llc.pdf`