Programmation et Structures de Données 1995-96
TD 1

Frédéric Vivien, bureau 306, e-mail: fvivien

22 septembre 1995

1 Premier contact

Le lancement de CAML s'effectue par la commande `camllight`. Le symbole `#` est une *invite*, qui comme son nom l'indique vous invite à entrer une commande, terminée par deux points-virgules successifs. Tapez par exemple :

```
# 1;;
- : int = 1
```

Essayez quelques calculs simples.

2 Les types de bases

```
# let b = 1 = 1 ;;
b : bool = true
# b = true ;;
- : bool = true
#"abc" ;;
- : string = "abc"
#`a` ;;
- : char = `a`
```

Exercice: définir deux entiers c et d, définir respectivement deux entiers e et f comme la somme et le produit de c et d.

3 Définitions

3.1 Définitions globales

`let` permet des *définitions* :

```
#let x = 1;;
x : int = 1
#x;;
- : int = 1
```

Le nom `x` est *lié* à la valeur 1. Dans ce cas, toute utilisation du nom `x` retourne la valeur à laquelle `x` était lié au moment de l'utilisation.

3.2 Définitions locales

Tous les objets définis plus haut sont *globaux*, c'est-à-dire que leur définition est valable en tout point du programme. Une définition temporaire (*locale*) est prescrite par la construction `let .. in`:

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

#let y = 2 in x+y;; (* Hors de cette ligne, y n'est pas defini *)
- : int = 5

#x ;;
- : int = 3

#y ;;
> Toplevel input:
>y ;;
>^
> Variable y is unbound.
```

Plusieurs variables locales peuvent être définies simultanément à l'aide du mot clé *and*:

```
#let s="Bonjour " and t="a tous!" in s^t ;;     (* notez le AND *)
- : string = "Bonjour a tous!"
#s ;;
> Toplevel input:
>s ;;
>^
> Variable s is unbound.
```

3.3 Liaisons statiques

Exercice: étudiez les programmes suivants ; quelles sont les valeurs retournées?

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

4 Les fonctions

4.1 Fonctions d'une variable

Ecrivons une fonction qui calcule le carré d'un nombre:

```
#let carre x = x * x ;;
carre : int -> int = <fun>
```

Remarquez que l'expression tapée n'est plus anonyme. Pour utiliser cette fonction, il faut procéder comme suit :

```
#carre 2;;
- : int = 4
```

Les fonctions peuvent être écrites de plusieurs manières, comme illustré ci-dessous sur la fonction identité.

```
#let identite = function x -> x ;;
identite : 'a -> 'a = <fun>
#let identite x = x ;;
identite : 'a -> 'a = <fun>
```

Nous verrons dans les cours/TDs suivants comment interpréter ces messages de CAML .

4.2 Fonctions de plusieurs variables

Ecrivons une fonction qui retourne la somme de deux nombres :

```
#let add x y = x + y ;;
add : int -> int -> int = <fun>
#add 3 4;;
- : int = 7
```

Nous aurions pu également écrire :

```
#let add2 (x, y) = x + y ;;
add2 : int * int -> int = <fun>
```

Mais cette fonction n'a à proprement parler qu'un seul paramètre, qui est une paire de deux nombres. Toute tentative de lui passer deux paramètres se soldera par une erreur :

```
#add2 3 4 ;;
> Toplevel input:
>add2 3 4 ;;
>^^^^
> Expression of type int * int -> int
> cannot be used with type int * int -> 'a -> 'b
#add2 (3, 4);;
- : int = 7
```

```
#add (3, 4);;                        (* Inversement, pour add... *)
> Toplevel input:
>add (3, 4);;
>     ^^^^
> Expression of type 'a * 'b
> cannot be used with type int
```

4.3 Définitions locales de fonctions

Nous avons vu au paragraphe 3.2 qu'une variable pouvait être définie localement. Il en va de même des fonctions.

Ecrivons une fonction qui élève un nombre à la puissance quatre au moyen d'une fonction qui élève un nombre au carré:

```
#let puissance_4 x =
     let puissance_2 y = y * y in
       (puissance_2 x) * (puissance_2 x);;
puissance_4 : int -> int = <fun>
```

ou mieux:

```
#let puissance_4 x =
     let puissance_2 y = y * y in
       (puissance_2 (puissance_2 x));;
puissance_4 : int -> int = <fun>
```

4.4 Liaisons statiques

On retrouve dans la définition des fonctions le problème de liaison statique rencontré lors de la définition des variables au paragraphe 3.3:

```
#let x = 1;;
x : int = 1
#let f y = x + y ;;          (* x est lie' a` la valeur 1 dans f *)
f : int -> int = <fun>
#let x = 88 ;;
x : int = 88
#f 2 ;;
- : int = 3
```

5 Vecteurs

Les tableaux ou “vecteurs” sont créés et utilisés de la façon suivante :

```
#let t = [| 1 ; 2 ; 3 |] ;;
t : int vect = [|1; 2; 3|]

#let q = make_vect 4 "bleu" ;;        (* quatre fois la chaine bleu *)
```

```
q : string vect = [|"bleu"; "bleu"; "bleu"; "bleu"|]
#vect_length q;;
- : int = 4
#q.(1);;
- : string = "bleu"
#t.(1);;
- : int = 2

#q.(1) <- "rouge" ;;                          (* attention au <- *)
- : unit = ()

#q.(1) <- 3;;                                 (* mauvais type *)
> Toplevel input:
>q.(1) <- 3;;
>         ^
> Expression of type int
> cannot be used with type string

#q ;;
- : string vect = [|"bleu"; "rouge"; "bleu"; "bleu"|]
```

Un vecteur est en fait une *référence* à une succession de valeurs de même type. Ainsi, par analogie avec le paragraphe 3.3 observe-t-on :

```
#let v = make_vect 4 1 ;;
v : int vect = [|1; 1; 1; 1|]
#let f x = v.(1) + x ;;
f : int -> int = <fun>
#f 2 ;;
- : int = 3
#v.(1) <- 0 ;;
- : unit = ()
#f 2 ;;
- : int = 2
```

6 Boucles For et conditionelles

La boucle `for` :

```
#for i=1 to 5 do print_int i done ;;
12345- : unit = ()

#for i = 0 to vect_length(q) - 1 do print_string q.(i) done ;;
bleurougebleubleu- : unit = ()
```

Conditionelle `if` *condition* `then` *instruction-1* `else` *instruction-2* :

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

```
abs : int -> int = <fun>

#abs (-8);;
- : int = 8
```

7 Sortie de Caml

```
#quit() ;;
```

8 Exercices sur les polynômes

Un polynôme sera manipulé sous la forme du vecteur de ses coefficients.

1. Récupérer une fonction qui affiche un polynôme:

   ```
   cp   ~fvivien/Enseignement/Sujets/source\_TD1.ml   .
   ```

2. Ecrire une fonction qui affiche un polynôme sous une forme plus "humaine", genre $3x\hat{}3 + 2x + 4$ au lieu de $3x\hat{}3 + 0x\hat{}2 + 2x\hat{}1 + 4$.

3. Ré-écrire cette fonction en utilisant une fonction définie localement qui affiche un monôme.

4. Ecrire une fonction qui effectue la somme de deux polynômes.

5. Ecrire une fonction qui effectue le produit de deux polynômes.