Programmation et Structures de Données 1995-96
TD 2

Frédéric Vivien, bureau 306, e-mail: fvivien

29 septembre 1995

1 Errata

Quelques petites mises-au-point suite au TD de la semaine dernière:

- Structure d'un *if* ... *then* ... *else* ...: Si vous devez écrire plusieurs instructions dans le *then* ou le *else*, celles-ci doivent être délimitées soit par des parenthèses, soit par les mots-clé *begin* et *end*.
- Variable et affectation: pour l'affichage des polynômes, certains ont écrit:

```
let affiche_polynome p =
    let drapeau = true in
      if drapeau then
         begin
         drapeau = false;
         ...
         end
      else
         begin
         ...
         end
      ...
      ;;
```

  L'expression *drapeau* = *false* n'est pas une affectation mais un test d'égalité qui renvoie une valeur booléenne ! La liaison statique que j'avais invoquée finit d'expliquer le comportement du programme.

- Pour pouvoir utiliser la commande *printf* il faut ouvrir le module qui la contient et qui s'appelle ...*printf*. L'ouverture du module se fait grâce à l'instruction suivante:

```
#open "printf";;
```

  Attention ! Le symbole '#' fait ici partie de la commande et n'est pas le '#' de l'invite. Ne pas oublier de le taper.

- Pour créer une matrice, utiliser la fonction *make_matrix*!

2 Fonctions récursives simples

2.1 Premiers exemples: factorielle et suite de Fibonacci

Ecrivez deux programmes calculant $factorielle(n)$ et $Fib(n)$.
Rappel: définition de la suite de Fibonacci:

$$\begin{cases} Fib(0) & = 1 \\ Fib(1) & = 1 \\ Fib(n+2) & = Fib(n+1) + Fib(n) \end{cases}$$

2.2 Fonctions récursives définies par un point fixe

La fonction factorielle peut s'écrire comme point fixe d'une fonctionnelle:

```
#  let phi f = function n -> if n = 0 then 1 else n * f(n-1);;
phi : (int -> int) -> int -> int = <fun>

#  let rec fact (n) = (phi fact) n;;
fact : int -> int = <fun>
```

Ecrivez la suite de Fibonacci comme point fixe d'une fonctionnelle.

2.3 Point fixe ?

La fonctionelle *psi* est définie comme suit:

```
#  let psi f = function n -> if n = 0 then 1 else f(n+1)/(n+1);;
psi : (int -> int) -> int -> int = <fun>
```

Calculez:

```
#  fact(3) ;;

#  psi fact 3 ;;

#  let rec new_fact (n) = (psi new_fact) n;;
```

Que vaut $new\_fact(3)$? Commentaires ?

3 Dans un sens ou dans l'autre...

3.1 Compter à l'endroit et à l'envers

Nous allons essayer de comprendre ici comment s'exécutent les appels à une fonction récursive. Pour ce faire, définissons une fonction qui énumère les nombres par ordre décroissant à partir de l'argument n et jusqu'à 1:

```
#  let rec compte_à_rebours n =
#    if n=0 then () else
#      begin
#        print_int n; print_string " ";
#        compte_à_rebours (n-1)
#      end;;
compte_à_rebours : int -> unit = <fun>

#  compte_à_rebours 10;;
10 9 8 7 6 5 4 3 2 1 - : unit = ()
```

Voyons maintenant ce qui se passe si on inverse l'appel récursif et les impressions:

```
#  let rec compte n =
#    if n=0 then () else
#      begin
#        compte (n-1);
```

```
#        print_int n; print_string " "
#      end;;
compte : int -> unit = <fun>
```

```
#  compte 10;;
1 2 3 4 5 6 7 8 9 10 - : unit = ()
```

Expliquez le comportement observé.

Nous allons utiliser le mécanisme de *trace* de Caml pour suivre les appels récursifs et les impressions. Ce mécanisme imprime à l'écran les appels successifs d'une fonction, ainsi que les résultats que cette fonction calcule. Traçons par exemple la fonction `successeur`:

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

```
#  trace "successeur";;
The function successeur is now traced.
- : unit = ()
```

```
#  successeur 2;;
successeur <-- 2
successeur --> 3
- : int = 3
```

Ayant compris le fonctionnement du mécanisme de trace, étudions le déroulement des appels à `compte` et `compte_à_rebours`:

```
#  trace "compte"; trace "compte_à_rebours";;
The function compte is now traced.
The function compte_à_rebours is now traced.
- : unit = ()
```

Vérifiez votre analyse du fonctionnement de ces fonctions en évaluant:

```
#  compte 3;;
```

```
#  compte_à_rebours 3;;
```

3.2 Épeler à l'endroit et à l'envers

Écrivons quelques fonctions récursives sur les chaînes de caractères.

Voici une fonction qui épelle des mots à l'envers (en commençant par le dernier caractère et en s'appelant récursivement sur le caractère précédent):

```
#  let rec épelle_envers_aux s i =
#    if i>=0 then
#      begin
#        print_char (nth_char s i); print_char ` `;
#        épelle_envers_aux s (i-1)
#      end;;
épelle_envers_aux : string -> int -> unit = <fun>
```

```
#  let épelle_envers s = épelle_envers_aux s (string_length s - 1);;
épelle_envers : string -> unit = <fun>
```

```
#  épelle_envers "programmation";;
n o i t a m m a r g o r p - : unit = ()
```

En vous inspirant de ce qui a été fait précédemment, écrivez deux fonctions différentes qui épellent un mot à l'endroit.

3.3 Les palindromes

Un palindrome est un mot qui se lit aussi bien à l'endroit qu'à l'envers. Pour déterminer si un mot est un palindrome, il suffit de vérifier que ses premiers et derniers caractères sont identiques, puis de vérifier que le mot situé entre ces deux caractères est lui aussi un palindrome. Il faut aussi remarquer qu'un mot réduit à un seul caractère et la chaîne vide sont aussi des palindromes.

Indication: la fonction *sub_string chaîne début longueur* extrait de la chaîne argument la sous-chaîne partant de l'indice *début* et de longueur *longueur*.

Ecrivez une fonction qui reconnaît des nombres palindromes.

4 Equivalence entre boucles et récursions

4.1 Dé-récursion au moyen de boucles

Ré-écrivez les fonctions des parties 2 et 3 en remplaçant les appels récursifs par des boucles.

4.2 Ré-écriture simple de boucles *for*

Ré-écrire la boucle ci-dessous à l'aide d'un appel récursif:

```
#let boucle f a b = for i = a() to b() do f(i) done ;;
boucle : (int -> 'a) -> (unit -> int) -> (unit -> int) -> unit = <fun>
```

Vérifiez la correction de votre solution en comparant le résultat de l'exécution des deux programmes avec les fonctions suivantes:

```
#let f n = print_int n;;

#let a ()  = (print_int 1515; 3);;

#let b ()  = (print_int 1789; 7);;
```

4.3 Dé-récursion à l'aide d'un itérateur

On définit l'itérateur suivant:

```
#let rec iter n f x =
#    if n = 0 then         x
#             else iter (n-1) f (f x);;
iter : int -> ('a -> 'a) -> 'a -> 'a = <fun>
```

Utiliser l'itérateur dans les cas suivants:

```
#let f (x) = (x+1);;

#let g = iter 10 f ;;

#g (0);;

#let f (x) = (print_int x; x+1);;

#let g = iter 10 f ;;

#g(0);;
```

Ecrire la suite de Fibonacci à l'aide de cet itérateur. Indication: On pourra exprimer la suite à l'aide d'une fonction ϕ telle que:

$$(u_{n+2}, u_{n+1}) = \phi(u_{n+1}, u_n)$$