LANGAGE OCaml – TP 3
Récursivité

Introduction

Dans les langages impératifs, l'utilisation de fonctions itératives est souvent privilégiée par rapport aux fonctions récursives à cause des problèmes d'occupation mémoire. Dans un paradigme déclaratif fonctionnel, l'utilisation des fonctions récursives est au contraire privilégiée.

Fonctions récursives

En OCaml, la définition d'une fonction récursive est introduite par l'adjonction du mot clé `rec` au mot clé `let`.

Récursivité simple

Exemple sur le calcul de la factorielle d'un nombre :

<pre>let rec fact_rec n =
    if n = 0 then 1
    else n*fact_rec (n-1);;
 	
fact_rec 5;;</pre>
	Version récursive du calcul de la factorielle d'un nombre.
	
	La version récursive permet une écriture plus compacte, tout en étant plus lisible puisque le recours aux références n'est plus indispensable.

<pre>let rec fact_rec_fil n =
    match n with
    | 0-> 1
    | _-> n*fact_rec_fil(n-1) ;;
 	
fact_rec_fil 5;;</pre>
	Autre version du calcul récursif de la factorielle d'un nombre, utilisant le filtrage pour identifier la condition d'arrêt des appels récursifs.
	
	La lisibilité du code est encore améliorée.

Différents types de récursivité peuvent être définis. Ils donnent lieu à des syntaxes particulières en OCaml.

Récursivité mutuelle (croisée)

Soient les suites $\begin{cases} u_{n+1} = \frac{1}{3}(u_n + 2 \times v_n) \\ v_{n+1} = \frac{1}{4}(u_n + 3 \times v_n) \end{cases}$ avec $\begin{cases} u_0 = 1 \\ v_0 = 1 \end{cases}$

En OCaml, il est possible de définir des fonctions mutuellement récursives en utilisant le mot clé and.

```
let rec un n =
  match n with
  | 0-> 1.
  | _-> (1. /. 3.) *. (un (n-1) +. 2. *. vn (n-1))
and
vn n =
  match n with
  | 0 -> 1.
  | _ -> 0.25 *. (un (n-1) +. 3. *. vn (n-1))
;;

un 2;;
vn 3;;
```

Sortie écran :

```
val un : int -> float
= <fun>

val vn : int -> float
= <fun>
```

Les deux fonctions sont définies.

On peut alors appeler un ou vn et obtenir le résultat de un ou vn :

```
# - : float = 1.
# - : float = 1.
```

Récursivité terminale

Pour rappel, la suite de Fibonacci est définie par :

Pour $n \in N$: $\begin{cases} u_0 = 1 \\ u_1 = 1 \\ u_n = u_{n-1} + u_{n-2} \end{cases}$

Ecrire la version récursive non terminale du calcul d'un élément de la suite de Fibonacci de rang n, de signature :

```
val fibo : int -> int = <fun>
```

Sortie attendue pour l'appel de fibo 8 :

```
- : int = 21
```

Cette fonction devra utiliser un filtrage.

```
let fibo_term n  =
   let rec fib_aux n a b =
      if n = 0 then a
      else fib_aux (n-1) b (a + b)
   in fib_aux n 0 1;;

fibo_term 8;;
```

La version récursive terminale du calcul d'un élément de la suite de Fibonacci de rang n, de signature `int -> int = <fun>` nécessite la définition d'une fonction récursive à l'intérieur du corps d'une autre fonction.
Sortie écran :

Exercices d'application directe

<table>
<tr><td><pre>let rec mystere1 n =
match n with
    | 0 -> print_newline()
    | _ ->
          print_int n;
          print_char ' ';
          mystere1 (n-1);
          print_int n;
          print_char ' ';;

mystere1 5;;</pre></td><td>Sortie écran et explications :</td></tr>
<tr><td><pre>let minuscule c =
char_of_int (32 + int_of_char
c);;

let rec mystere2 s i =
if i = String.length s then
print_newline()
else begin
   print_char s.[i];
      s.[i] <- minuscule s.[i];
      mystere2 s (i+1);
      print_char s.[i]
   end;;

mystere2 "BONJOUR" 0;;</pre></td><td>Sortie écran et explications :</td></tr>
</table>

Exercices

Exercice 1 : somme des entiers

Écrivez une fonction récursive `som_chiffres` de type `int -> int` qui calcule la somme des chiffres de l'entier passé en argument.

Exercice 2 : récursivité – division par 2^k

Écrire une fonction `log2` qui prend en argument un entier n, avec $n \geq 1$, et calcule le plus grand entier k tel que $2^k \leq n$. On proposera une version itérative et une version récursive.

Exercice 3 : coefficients binomiaux

Écrire une fonction récursive de signature `int -> int -> int` qui calcule le coefficient binomial $\binom{n}{k}$. Il faudra que cette fonction ait une complexité linéaire en appels récursifs.
On rappelle que si $k, n \in N^*, on\ a\ k\binom{n}{k} = n\binom{n-1}{k-1}$.

Exercice 4 : récursivité – nombres parfaits

Un nombre entier n ≥ 2 est dit parfait s'il est égal à la somme de ses diviseurs positifs (y compris 1) différents de lui-même.

Ecrire une fonction récursive `estparfait` de signature `val est_parfait : int -> bool = <fun>` indiquant si un entier n est parfait.

Pour tester votre fonction, les nombres parfaits inférieurs à 10000 sont : 6, 28, 496 et 8128.

Exercice 5 : ordre lexicographique

Écrivez une fonction récursive ordre_lexico de type string -> string -> bool qui détermine si une chaîne de minuscules est située avant une autre pour l'ordre lexicographique.

Vous pourrez utiliser la fonction sub, prédéfinie dans le module String, pour extraire une sous-chaîne d'une chaîne de caractères. `String.sub s a b` donne la sous-chaîne de `s` de longueur `b` démarrant à la position `a` (la position du premier caractère est 0).

Par exemple :
```
String.sub "Expression Logique et Fonctionnelle Evidemment" 22 13;;
 - : string = "Fonctionnelle"
```

Exercice 6 : programmation en C

Programmer les 5 exercices précédents en langage C.