TP 2— Programmation récursive

http://www.gallium.inria.fr/~scherer/teaching/colles/2012/tp2.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 !

Cette colle a été (comme la première, et sans doute les suivantes) adaptée d'une colle donnée par Victor Nicollet à LLG il y a quelques années.

1 Récursivité

On dit qu'une fonction est *récursive* lorsqu'elle apparaît dans sa propre définition. Cela est indiqué par le symbole `rec`, qui rend le nom de la fonction disponible à l'intérieur de sa définition (sans lui, le nom ne serait disponible qu'après la définition).

L'exemple ci-contre est une fonction récursive qui calcule $n!$. Lorsque `fac n` est exécutée, elle va appeler `fac (n-1)`, qui va appeler `fac (n-2)` et ainsi de suite jusqu'à 0. Il va donc y avoir $n+1$ appels à `fac` pour calculer `fac n`.

```
let rec fac = function
  | 0 -> 1
  | n -> n * fac (n - 1);;
```

La récursion est un outil pratique pour traduire en `Caml Light` des fonctions ou suites mathématiques définies par récurrence.

▷ **Question 1.** Écrire une fonction récursive simple qui calcule le terme n de la suite de Fibonnaci, avec $u_0 = u_1 = 1$. Quelle est sa complexité en fonction de n ? Peut-on l'améliorer (indice : utiliser des couples) ? ◁

La récursion est un très bon outil pour résoudre les problèmes présentant la structure suivante :

- un ou plusieurs cas de base très simples (par exemple ici : $0! = 1$)
- un cas général que l'on peut décomposer en sous-problèmes plus simples (ici, pour connaître $n!$ il suffit de connaître $(n-1)!$).

▷ **Question 2.** On sait que $x^0 = 1$, $x^{2k} = (x^2)^k$ et $x^{2k+1} = x(x^2)^k$. En déduire une fonction récursive pour calculer x^n (x et n donnés).
Indication : étudier la parité de n. ◁

2 Types somme

Il est possible en `Caml Light` de définir de nouveaux types. Un type somme est constitué d'un ensemble de valeurs associées à des *constructeurs*. Ci-contre, `contact` est décrit par deux constructeurs: `Tel`, associé à un entier, `Mail` associé à une chaîne de caractères, et `Inconnu` qui n'est associé à rien du tout; un contact sera donc soit un numéro de téléphone, soit une adresse mail, soit un inconnu (étrange...).

```
#type contact = Tel of int | Mail of string | Inconnu;;
Type contact defined.
#Mail "gabriel.scherer@gmail.com";;
- :  contact = Mail "gabriel.scherer@gmail.com"
```

Rappel : On suppose connu les tuples : si `ta` et `tb` sont des types `Caml`, `ta * tb` est le type des couples d'un élément de type `ta`, et d'un élément de type `tb`. Par exemple `(1, "blah") : int * string`. On peut faire des n-uplets à plus de deux membres, comme `(1, "blah", 3.5) : int * string * float`.

▷ **Question 3.** On souhaite écrire une fonction qui résout une équation du second degré donnée par un triplet (a, b, c). Définir un type capable de représenter zéro, une ou deux solutions, puis écrire la fonction renvoyant un objet de ce type. ◁

2.1 Pattern matching

Il est facile de construire des valeurs d'un type somme, mais comment les utiliser ensuite ? On va vouloir faire quelque chose comme "si c'est un numéro de téléphone, alors téléphoner, sinon envoyer un mail".

Cependant, l'instruction conditionnelle `if then else` n'est pas suffisante dans ce cas : on voudrait dire `if contact = Tel`, mais ça ne marche pas, car contact est de la forme `Tel` *`entier`*. Il faut pouvoir dire "si contact est de la forme `Tel` *`numéro`*, alors téléphoner à `numéro`". C'est le principe du *pattern matching*, ou *filtrage de motif*.

```
let contacter contact = match contact with
| Mail m -> "Envoyer un mail à "^m
| Tel t -> "Appeler le "^(string_of_int t);;
```

match $expr$ with
| $pattern_1$ -> $expr_1$
| $pattern_n$ -> $expr_2$
...

L'instruction essaye les motifs à gauche des `->`, de haut en bas; lorsqu'un premier motif $pattern_n$ correspond (il "matche") à la valeur $expr$, les variables présentes
dans le motif reçoivent la valeur correspondant dans $expr$, et l'expression à droite du `->`, $expr_n$, est renvoyée.

▷ **Question 4.** Que fait la fonction `est_ce_moi` ? ◁

Il faut bien comprendre que les motifs permettent à la fois de faire des choix (comme `if..then..else`) et de nommer des variables (les identifiants présents dans le motif). Le motif `Mail m` par exemple va déclarer une nouvelle variable `m` valant l'adresse mail du contact (si le contact est bien de la forme `Mail mail`). Elle ne fait pas référence à une variable `m` précédente (qui, si elle existe, sera écrasée). En particulier, on ne peut pas mettre plusieurs fois la même variable dans un seul motif.

```
let est_ce_moi contact =
let mon_tel = 0102030405 in
let mon_mail = "gabriel.scherer@gmail.com" in
match contact with
| Mail mon_mail -> true
| Tel mon_tel -> true
| Mail _ | Tel _
| Inconnu -> false
```

Si un motif est constitué d'une seule variable, il matche n'importe quelle valeur, et lie la variable à cette valeur : `match a with b -> c` est équivalent (sans polymorphisme) à `let b = a in c`. Il existe aussi un motif acceptant tout, et ne déclarant pas de nouvelle variable : `_`
On peut aussi écrire des motifs pour les n-uplets : `(`$patt_1$`, `$patt_2$`, ...)` est un motif correspondant aux n-uplets dont les membres correspondent aux motifs $patt_1$, $patt_2$, ...

3 Listes

3.1 Types récursifs

Il est possible d'utiliser un type dans sa propre définition, sans même spécifier un `rec`.

On programme un robot qui se déplace dans le plan. Un programme du robot peut consister en deux choses: la commande `Stop`, qui marque la fin du programme, et la commande `Move (p,suite)`, qui indique au robot de bouger de se rendre au point `p : float * float` et d'exécuter ensuite le programme `suite`.

Définir un type `Caml Light` décrivant un programme. Quel est le programme correspondant aux mouvements $((1,2),(-1,-3),(2.2,-3))$?

▷ **Question 5.** Écrire une fonction qui, étant donnés un point de départ et un programme, calcule le point d'arrivée du robot. ◁

▷ **Question 6.** Écrire une fonction qui calcule la taille (le nombre de déplacements) du programme qu'on lui donne. ◁

3.2 Principe

Une liste est la généralisation du type récursif utilisé dans la question 5. La commande `Stop` est notée `[]`. Il s'agit de la liste vide, qui ne contient aucun élément. La commande `Move (x,p)` est notée `x ::p`: elle contient l'élément `x` appelé tête de la liste, et la liste `p` appellée queue de la liste. Ces deux constructions permettent à la fois d'écrire des expressions pour construire des listes, et des motifs pour les déconstruire.

On utilise une abbréviation: la liste des entiers de 1 à 3, `1 ::(2 ::(3 ::[]))`, peut s'écrire de manière plus lisible comme: `[1; 2; 3]`

Contrairement aux tableaux, les listes ne peuvent pas être modifiées. La manipulation des listes ne comporte donc que deux possibilités: parcourir la liste pour en extraire des informations, ou créer une nouvelle liste.

3.3 Parcourir une liste

La fonction ci-contre calcule la longueur d'une liste en la parcourant, élément par élément. Plus généralement, on parcourt une liste comme on étudie un type récursif : on écrit une fonction qui effectue un travail sur la tête de la liste et s'applique récursivement sur la queue de la liste.

```
let rec length = function
  | [] ->
    0
  | h ::t ->
    length t + 1;;
```

On a utilisé dans ce code la forme nouvelle `(function ...)`, équivalente à `(fun x -> match x with ...)`, sans nommer de variable `x`. Attention, `function` prend implicitement un paramètre (comme `fun`, mais on ne le nomme pas, on lui applique directement des motifs), vous l'oublierez de temps en temps !

▷ **Question 7.** On utilise maintenant des listes pour programmer le robot. Décrire les mouvements $((1,2),(-1,-3),(2.2,-3))$, puis réécrire les fonctions calculant le point d'arrivée et le nombre de mouvements. ◁

3.4 Créer une liste

On sait créer manuellement des listes de longueur donnée. Comment créer des listes de longueur arbitraire?

Créer une liste est fondamentalement un problème récursif: si on sait construire la queue de la liste, on peut construire la liste en y ajoutant simplement la tête.

Par exemple, la fonction ci-contre calcule la liste des carrés des entiers plus petits que n.

```
let rec carres = function
  | 0 ->
    []
  | n ->
    (n*n)::(carres (n-1));;
```

▷ **Question 8.** Écrire une fonction qui calcule la liste de diviseurs premiers d'un entier n, apparaissant autant de fois dans la liste que dans n. ◁

3.5 Transformation de listes

On ne peut pas transformer les listes, puisqu'on ne peut pas les modifier! En revanche, on peut créer une liste à partir des données lues dans une autre. Pour cela, il suffit en général d'écrire une fonction qui lit des données dans une liste, et qui renvoie une liste.

Une modification peut impliquer de remplacer la tête par une autre tête, d'insérer des éléments avant ou après la tête, ou même d'éliminer la tête pour ne garder que la queue de la liste. Des exemples de modification sont donnés ci-contre.

```
let rec ajoute x = function
  | [] ->
    []
  | t::q ->
    (t+x)::(ajoute x q);;

let rec supprime_impairs = function
  | [] | [_] ->
    []
  | impair::pair::reste ->
    pair::(supprime_impairs reste);;
```

▷ **Question 9.** La fonction `map` prend en argument une liste `[x1; x2; ... xn]` et une fonction `f` et renvoie la liste `[f(x1); f(x2); ... f(xn)]`. Réecrivez vous-même cette fonction. ◁

▷ **Question 10.** Écrire la fonction `rev_append`, telle que `rev_append` ajoute a, retournée, devant b : `rev_append [1;2;3] [4;5;6] = [3; 2; 1; 4; 5; 6]`

En déduire (et écrire) une fonction `rev` qui retourne une liste. ◁

▷ **Question 11.** Écrire une fonction `insert` qui, étant donnés une liste triée `l` et un élément `e`, renvoie la liste triée contenant `e` ainsi que les éléments de `l` (on parle d'insérer un élément dans une liste triée).

En déduire (et écrire) une fonction qui applique le tri par insertion pour trier une liste. ◁

▷ **Question 12.** Implémenter une queue FIFO (on ajoute les éléments à une extrémité, on les retire à l'autre) et les opérations `enqueue` et `dequeue` (complexité constante amortie) pouvant contenir un nombre arbitraire d'éléments.

Indice : utiliser un couple de listes, les éléments "de devant" et les éléments "de derrière". Attention à l'ordre ! ◁

▷ **Question 13.** À faire à la fin, s'il vous reste du temps.
Regarder le type, comprendre le fonctionnement et recoder les fonctions suivantes :

- do_list
- map2
- index
- assoc
- split, combine
- (difficile) it_list, list_it

◁

Les formes `let` acceptent en fait tout motif à la place de l'identifiant (qui est un cas particulier de motif). Si la valeur liée ne correspond pas au motif, une erreur est levée.

```
#let a::b = [];;
Uncaught exception:  Match_failure ("", 0, 15)
```

4 Questions difficiles

▷ **Question 14.** On dit qu'une fonction utilise la récursion terminale si elle renvoie le résultat d'un appel récursif, plutôt que le résultat d'une autre opération. Un exemple est donné ci-contre.

```
let dernier = function
  | [] -> failwith "Erreur"
  | [x] -> x
  | _::q -> dernier q
```

Expliquer pourquoi de telles fonctions peuvent être très simplement réécrites sous la forme d'une boucle `while`, et consomment une quantité fixe de mémoire.

Proposer une méthode pour transformer des fonctions comme `facto`, `combine` ou `map` en fonctions à récursion terminale. Est-ce possible pour toutes les fonctions? ◁

▷ **Question 15.** Écrire un programme qui provoque l'erreur suivante: ◁

```
This expression has type truc, but is used with type truc.
```

▷ **Question 16.** Dessiner un chameau ◁