TP 1— Partir sur de bonnes bases

http://www.liafa.jussieu.fr/∼nath/tp1/tp1.pdf

Les questions sont les bienvenues et peuvent être envoyées à `nathanael.fijalkow@gmail.com`.

Ce TP est largement inspiré de deux TP donnés par Gabriel Schérer.

Ces TP d'informatique servent à vous préparer pour les concours aux épreuves contenant de la programmation `caml`: il s'agit principalement d'épreuves *écrites* (Centrale, Mines-Ponts et surtout X), puisque seul au concours INFO des ÉNS il y a une épreuve sur machine. Donc l'objectif de ces TP est de vous apprendre bien sûr à programmer en `caml`, mais surtout à écrire sur du papier (avec un crayon, si si) du code `caml` lisible, clair, indenté, et à le commenter (fonctionnement, preuve de correction et terminaison, analyse de complexité).

L'objectif de ce premier TP est modeste : implémenter une exponentiation rapide et un tri rapide. Ces deux fonctions servent très souvent, et vous serez certainement amenés à les recoder par la suite, parfois avec des variantes, donc il est important de bien en connaître les écueils.

1 Récursivité

En *programmation* on dit qu'une fonction est *récursive* lorsqu'elle apparaît dans sa propre définition (cette phrase devrait faire hurler le matheux qui est en vous, mais qu'importe). 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 `fact n` est exécutée, elle va appeler `fact (n-1)`, qui va appeler `fact (n-2)` et ainsi de suite jusqu'à 0. Il va donc y avoir $n+1$ appels à `fact` pour calculer `fact n`.

```
let rec fact n = match n with
  | 0 -> 1
  | _ -> n * fact (n - 1) ;;
```

▷ **Question 1.** Écrire une fonction `fibo` récursive tel que `fibo n` calcule le n-ième terme de la suite de Fibonaci, avec $u_0 = 0$ et $u_1 = 1$. ◁

La fonction `fibo` que vous venez d'écrire a une complexité désastreuse, alors qu'elle ressemble fort à `fact`. Pourquoi? Analysons la pile de récursivité de ces deux fonctions. Pour `fact`, elle a la forme :

$$\texttt{fact n} \to \texttt{fact (n - 1)} \to \cdots \to \texttt{fact 0}$$

Pour `fibo`, elle a une forme plus compliquée :

$$\texttt{fibo n} \to \begin{array}{l}\texttt{fibo (n-2)}\\ \texttt{fibo (n-1)}\end{array} \to \begin{array}{l}\texttt{fibo (n-2)}\\ \texttt{fibo (n-3)}\\ \texttt{fibo (n-2)}\end{array} \to \dots$$

▷ **Question 2.** Quelle est la complexité de `fibo n` en fonction de n? Proposer une amélioration en utilisant des couples, quelle est sa complexité? ◁

On s'intéresse maintenant au problème de l'exponentiation. On veut un algorithme qui étant donnés x entier et $n \geq 0$, calcule x^n. On peut faire simple, et calculer x^2, puis x^3, puis x^4, jusqu'à x^n, mais ceci donne un algorithme linéaire, pas très satisfaisant. On utilise la récursivité : on sait que $x^0 = 1$, $x^{2n} = (x^n)^2$ et $x^{2n+1} = x \cdot (x^n)^2$.

▷ **Question 3.** Écrire une fonction `exponentiation` récursive tel que `exponentiation x n` calcule x^n. ◁

La fonction `exponentiation` que vous venez de coder vous sera d'une utilité constante, et certainement pas que cette année!

▷ **Question 4.** On s'intéresse au nombre de multiplications effectuées par cet algorithme. Quelle est la complexité de `exponentiation x n` en fonction de n (ici, on ne demande qu'un ordre de grandeur)? ◁

2 Types somme

Un petit rappel sur les types. Parmi les types de bases, ceux que l'on utilise souvent sont : `int` (entier), `float` (flottant), `char` (caractère), `string` (chaîne de caractères) et `unit`. Il est possible en `caml`de définir de nouveaux types. Si `t1` et `t2` sont des types `caml`, `t1 * t2` est le type des couples d'un élément de type `t1` et d'un élément de type `t2`. 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`. Décrivons maintenant les types somme.

Un type somme est constitué d'un ensemble de valeurs associées à des *constructeurs*. Ci-contre, `contact` est décrit par trois 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.

```
#type contact = Tel of int | Mail of string | Inconnu;;
Type contact defined.

#Mail "nathanael.fijalkow@gmail.com";;
- :  contact = Mail "nathanael.fijalkow@gmail.com"
#Inconnu;;
- :  contact = Inconnu
```

2.1 Pattern matching

En lisant un contact, on veut faire quelque chose comme "si c'est un numéro de téléphone, alors téléphoner, sinon envoyer un mail". Ici l'instruction conditionnelle `if then else` n'est pas suffisante : on voudrait dire `if contact = Tel`, mais ça ne marche pas, car contact est de la forme `Tel` *`entier`*. Le *pattern matching*, ou *filtrage de motif*, donne cette liberté : "si contact est de la forme `Tel` *`numéro`*, alors téléphoner à `numéro`".

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

match $expr$ with
$|\ pattern_1$ -> $expr_1$
$\vdots$
$|\ pattern_n$ -> $expr_n$
$\vdots$

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

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 de la forme `Mail mail`). Elle ne fait pas référence à une variable `m` précédente (qui, si elle existe, sera écrasée). 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 $patt_1$, $patt_2$, ...

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

▷ **Question 5.** Pourquoi la fonction `est_ce_moi` ne fait pas ce qu'elle devrait? ◁

3 Listes

3.1 Types récursifs

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

```
type programme_robot =
Stop | Move of (int * int) * programme_robot ;;
```

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 : int * int` et d'exécuter ensuite le programme `suite`.

Pour `'a` un type quelconque, le type `'a list` généralise le type précédent. 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` appelée queue de la liste. `[]` et `::` sont deux constructeurs, ils permettent à la fois d'écrire des expressions pour construire des listes, et des motifs pour les déconstruire.

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

3.2 Parcourir une liste

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

La fonction ci-contre calcule la longueur d'une liste en la parcourant, élément par élément.

Remarque : la forme `(function ...)` est 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.

3.3 Créer une liste

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

La fonction ci-contre calcule la liste des carrés des entiers plus petits que n.

▷ **Question 6.** Écrire une fonction qui calcule la liste des diviseurs premiers d'un entier n, apparaissant autant de fois dans la liste que dans n. ◁

3.4 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.

▷ **Question 7.** La fonction `map` (du module `List`) prend en argument une fonction `f` et une liste `[x1; x2; ... ; xn]` et renvoie la liste `[f(x1); f(x2); ... ; f(xn)]`. Écrire cette fonction. ◁

▷ **Question 8.** Écrire une fonction `rev` qui retourne une liste. Votre fonction est-elle efficace? J'en doute. Voici un conseil: écrire une fonction `rev_append` tel que `rev_append l1 l2` retourne `l1` et la concatène à `l2`. Avec cette fonction, écrire une nouvelle fonction `rev`, linéaire. ◁

On s'intéresse maintenant au problème de trier une liste. Nous supposerons ici qu'il s'agit d'une liste d'entiers, et on veut l'ordonner selon l'ordre naturel sur les entiers, croissant.

▷ **Question 9.** Écrire une fonction `insertion_tri` qui, étant donnés une liste triée `liste` et un élément `elt`, renvoie la liste triée contenant `elt` ainsi que les éléments de `liste` (on parle d'insérer un élément dans une liste triée).
En déduire (et écrire) une fonction qui effectue le tri par insertion pour trier une liste. ◁

Continuons avec le tri rapide. Rappelons le principe : étant donné une liste sous la forme `a::liste`, on se sert de `a` comme pivot. On construit deux listes, `linf` et `lsup`, la première contenant les élements de `liste` strictement inférieur à `a`, et la seconde tous les autres. On trie récursivement `linf` et `lsup`, puis on concatène les résultats.

▷ **Question 10.** Écrire une fonction `pivot` qui, étant donné un élément `a` et une liste `liste`, renvoie le couple de liste (`linf`,`lsup`) comme décrit ci-dessus.
En déduire (et écrire) une fonction qui effectue un tri rapide pour trier une liste. ◁

4 Question difficile

Considérons à nouveau le problème de l'exponentiation. On s'intéresse au nombre minimum de multiplications nécessaires pour calculer x^n. Pour définir le problème précisément, on suppose disposer d'une mémoire, qui initialement contient $x^0 = 1$ et $x^1 = x$. Chaque étape consiste à multiplier entre eux deux entiers disponibles dans la mémoire, et à ajouter l'entier obtenu à la mémoire. On ne s'intéresse qu'au nombre d'étapes minimal pour obtenir x^n, et pas aux coûts de gestion de cette mémoire.

Reprenons l'analyse de l'algorithme récursif. Voici une autre façon de comprendre cet algorithme en considérant l'écriture binaire de n en base 2. On part du bit le plus faible :

(1) si c'est un 0, alors on calcule récursivement $x^{n/2}$ en supprimant ce bit (division par 2), puis on retourne le carré de $x^{n/2}$, ce qui occasionne une multiplication supplémentaire;

(2) si c'est un 1, alors on calcule récursivement $x^{n/2}$ en supprimant ce bit (division par 2), puis on calcule le carré de $x^{n/2}$, que l'on multiplie à x, ce qui occasionne deux multiplications supplémentaires.

▷ **Question 11.** Notons $k+1$ le nombre de bits dans l'écriture binaire de n, et $j-1$ le nombre de 1. Exprimer le coût de l'algorithme en fonction de j et k. Exprimer k en fonction de n, et encadrer j. ◁

▷ **Question 12.** À partir de l'analyse précédente; identifier les pires cas, en fonction de n, de l'algorithme récursif. Trouver une valeur de n pour laquelle il existe un algorithme n'effectuant que des multiplications, calculant x^n avec strictement moins de multiplications (indication : $n = 2^4 - 1$). ◁

Considérons la question suivante : donner une méthode de construction d'un arbre (potentiellement infini) étiqueté par l'ensemble des entiers naturels, dont le chemin depuis la racine à l'entier n donne un chemin de multiplications optimal.

▷ **Question 13.** Donald (Knuth, à qui avez-vous pensé?) propose l'arbre suivant, comment est-il construit, pourquoi n'est-il pas optimal? La plus petite valeur pour laquelle cet arbre n'est pas optimal est $19\,879$... ◁