Premier chapitre

C. Charignon

Table des matières

I Cours **2**

1 Présentation de Caml **2**
1.1 Caractéristiques principales 2
1.2 Comparaison à Python 2

2 Récursivité **3**
2.1 Exemples 3
2.1.1 Factorielle 3
2.1.2 PGCD 3
2.2 Exemple de mauvaise fonction récursive 4

3 Fonctions à plusieurs variables **4**

II Exercices **5**

Première partie
Cours

1 Présentation de Caml

1.1 Caractéristiques principales

Les deux caractéristiques principales du langage Caml sont les suivantes :

- C'est un langage fonctionnel : il est conçu pour écrire de nombreuses petites fonctions qui s'appellent les unes les autres plutôt qu'une seule grosse fonction. En outre, il facilite un style utilisant peu de boucles et de variables. Ainsi durant toute la première partie du cours, nous n'utiliserons aucune variable et aucune boucle.
- C'est un langage fortement typé : chaque fonction aura un type précis, défini par le type de ses arguments et le type de son résultat. Une même fonction ne pourra pas prendre en argument une fois un entier et une fois un flottant par exemple. Et elle devra renvoyer dans tous les cas un résultat du même type.

1.2 Comparaison à Python

- Pas de return ! Il n'y en a pas besoin car le résultat renvoyé par une fonction est clairement apparent dans sa construction même.
- Tous les identifiants sont définis avec la commande **`let`** : y compris les fonctions. Sur ce point, c'est d'ailleurs Python qui est bizarre : une fonction est un objet comme les autres.
  En outre un **`let`** est suivi d'un **`in`** dont le but est d'indiquer la zone où l'identifiant sera définie, sauf en cas d'identifiant global.
- Syntaxe d'un test : **`if ... then ... else ...`**
  Pas de « elif » en Caml, mais il suffit d'écrire **`else if`** .
- Ce sont les parenthèses qui définissent les blocs, et non plus l'indentation. On peut également utiliser **`begin`** ↪ **`.... end`**. Cependant ceci ne sert que lorsqu'on utilise des variables et des boucles ; ce sera pour un chapitre ultérieur.
- Fin de la définition d'un identifiant global : deux points-virgule.
- L'opérateur de comparaison se note `=`. Il n'y a pas de `==`, une affectation étant clairement signalée grâce au **`let`**.
- Les arguments d'une fonction s'écrivent tous à la suite, séparés uniquement par un espace. Même pas de parenthèses autour. Nous verrons pourquoi plus loin.
- L'évaluation d'une fonction est prioritaire sur les autres opérations. Exemple : `f 2 * 3` signifie $f(2) * 3$. Donc pour calculer $f(2 * 3)$ il faudra taper `f (2*3)`.
- Chaque opération n'est définie que pour un type précis : par exemple `+`, `*\`, `/` fonctionnent pour les entiers (en particulier, `/` est la division euclidienne). Pour les flottants, ce sera `*.`, `+.`, et `/.`.
  Exception : les comparaisons fonctionnent quand même pour des types différents. Ainsi `x<y` ou `x=y` sera accepté que x et y soient flottants ou entiers.
  En effet, l'égalité et les relations d'ordre sont d'une telle importance en informatique qu'il est souvent utile que n'importe quel type soit muni d'une relation d'ordre.

2 Récursivité

Principe de Hofstadter : il faut toujours plus de temps que prévu, même en tenant compte du principe de Hofstadter.

Le langage Caml est particulièrement adapté à la programmation récursive. Traitons quelques exemples.

2.1 Exemples

2.1.1 Factorielle

La définition de la suite factorielle, notée $(n!)_{n\in\mathbb{N}}$ est la suivante :

$$\begin{cases} 0! = 1 \\ \forall n \in \mathbb{N},\ (n+1)! = (n+1) \times n! \end{cases}$$

(C'est une définition par récurrence.)
À l'aide d'un décalage d'indice on peut réécrire ceci de manière plus pratique :

$$\begin{cases} 0! = 1 \\ \forall n \in \mathbb{N}^*,\ n! = n \times (n-1)! \end{cases}$$

Notez le $\mathbb{N}^*$ au lieu de $\mathbb{N}$ pour le domaine de définition de la seconde égalité.

Et cette définition peut être traduite presque mot à mot en Caml :

```
let factorielle n =
    if n=0 then
      1
    else
      n * factorielle (n-1)
;;
```

2.1.2 PGCD

Rappelons le lemme d'Euclide, qui justifie l'alogrithme éponyme, qui permet de calculer un plus grand dénominateur commun à deux entiers :

Lemme 2.1. *Soit $(a,b,q,r) \in \mathbb{Z}^4$ telles que $a = bq + r$. Alors $a \wedge b = b \wedge r$.*

Démonstration :

- Tout diviseur de b et de q divise aussi $bq + r$ c'est-à-dire a.
- Réciproquement, tout diviseur de a et b divise aussi $a - bq$, c'est-à-dire r.

Ainsi les couples (a,b) et (b,r) ont-ils les mêmes diviseurs communs, et donc les même plus grands diviseurs communs. □

Et rappelons ce cas particulier simple :

Lemme 2.2. *Soit $(a,b) \in \mathbb{Z}^2$ tel que $b = 0$. Alors $a \wedge b = a$.*

Démonstration : Tout entier divise 0. En conséquence, les diviseurs communs à a et 0 sont les diviseurs de a. □

Ces deux lemmes donnent la fonction suivante :

```
let pgcd a b =
  if b = 0 then
    a
  else
    pgcd b (a mod b)
;;
```

2.2 Exemple de mauvaise fonction récursive

cf exercice : 1

L'exemple classique de mauvais emploi de la récursivité est le calcul d'une suite récurrente double, comme la suite de Fibonacci. Soit $F \in \mathbb{N}^{\mathbb{N}}$ telle que $\begin{cases} F_0 = 0, F_1 = 1 \\ \forall n \in \mathbb{N}, F_{n+2} = F_{n+1} + F_n \end{cases}$.

Remarque : Nombre de manières de monter un escalier par pas de 1 ou 2 marche(s).

Une fonction naïve serait :

```
let rec fiboNaif n =
    match n with
        |0 -> 1
        |1 -> 1
        |_ -> fiboNaif (n-1) + fiboNaif (n-2)
;;
```

On se rend vite compte que chaque valeur de F_k est recalculée un grand nombre de fois, d'où un grand nombre de calculs inutiles.

Notons pour tout $n \in \mathbb{N}$ C_n le nombre d'additions effectuées par `fiboNaif n`. Vu le programme,

$$\begin{cases} C_0 = 0 = C_1 \\ \forall n \in [\![2, \infty[\![, C_n = C_{n-1} + C_{n-2} + 1 \end{cases}.$$

Sans ce $+1$, ça serait une suite récurrente double, et nous saurions calculer son terme général par le cours de math. La méthode ici consiste à chercher une constante k telle que $C - k$ soit vraiment une suite à récurrence linéaire double, c'est-à-dire telle que $\forall n \in [\![2, \infty[\![, (C_n - k) = (C_{n-1} - k) + (C_{n-2} - k)$. Manifestement, $k = 1$ convient. La règle générale est qu'il suffit de prendre pour k une constante qui vérifie la même relation de récurrence que la suite.

On pose donc $u = C + 1$ (c'est-à-dire $\forall n \in \mathbb{N}$, $u_n = C_n + 1$). Donc $u_0 = 1, u_1 = 1$ et $\forall n \in \mathbb{N}$, $u_{n+2} = u_{n+1} + u_n$.

La suite u est une brave suite à récurrence double linéaire. Son équation caractéristique est $r^2 = r + 1$ d'inconnue $r \in \mathbb{C}$, dont les racines sont $\dfrac{1+\sqrt{5}}{2}$ et $\dfrac{1-\sqrt{5}}{2}$. Donc il existe deux constantes $(\alpha, \beta) \in \mathbb{R}^2$ telle que

$$\forall n \in \mathbb{N}, u_n = \alpha \left(\frac{1+\sqrt{5}}{2}\right)^n + \beta \left(\frac{1-\sqrt{5}}{2}\right)^n$$

Dès lors :

$$C_n = u_n - 1 \underset{x\to\infty}{\sim} \alpha \left(\frac{1+\sqrt{5}}{2}\right)^n \underset{n\to\infty}{=} O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right).$$

C'est une complexité exponentielle ! (sauf si $\alpha = 0$, mais ce n'est pas possible : la complexité tendrait vers 0.)

Remarque : Bien sûr, on peut calculer α, mais ce n'est pas nécessaire si on s'intéresse seulement à l'ordre de grandeur.

cf exercice : 6 pour une version efficace du calcul de cette suite.

Remarque : Autre méthode pour se débarrasser de la constante, en ne faisant plus les calculs exacts mais seulement une minoration : On a $\forall n \in \mathbb{N}$, $C_{n+2} \geqslant C_{n+1} + C_n$, $C_0 = 0$ et $C_1 = 1$. Notons u la suite telle que $u_1 = 1$, $u_2 = 2$ et $\forall n \in \mathbb{N}$, $u_{n+2} = u_{n+1} + u_n$, on montre alors par récurrence simple que $\forall n \in \mathbb{N}$, $C_n \geqslant u_n$.

3 Fonctions à plusieurs variables

Ce petit paragraphe indépendant a pour but de préciser l'usage de fonctions à plusieurs variables, et en particulier d'expliquer pourquoi les différents arguments ne sont pas séparés par des virgules et entourés de parenthèses.

Prenons la fonction suivante :

```
let addition x y =
    x+y
;;
```

Son type est `int -> int -> int`, et cela signifie qu'elle prend deux entiers en entrée et renvoie un autre entier.

Mais cette écriture avec ces deux flèches peut sembler bizarre. En fait, Caml considère qu'il y a une parenthèse à droite : le type est donc `int -> (int -> int)`.

Cela signifie que lorsqu'on lui donne un entier, la fonction "addition" renvoie une nouvelle fonction, de type `int -> int`. Autrement dit, si on lui donne un entier, on obtient la fonction qui attend le deuxième entier.

Par exemple `addition 1` est la fonction qui attend un entier y pour renvoyer $1 + y$.

```
let plusUn = addition 1;;

plusUn 3;;
```

Deuxième partie

Exercices

Exercices : récursivité élémentaire

Exercice 1. * Calcul approché de racine

Soit $a \in \mathbb{R}^{+*}$. Soit $u \in \mathbb{R}^{\mathbb{N}}$ la suite vérifiant $u_0 = a$ et $\forall n \in \mathbb{N}$, $u_{n+1} = \frac{u_n}{2} + \frac{a}{2u_n}$. On démontre facilement que u converge très rapidement vers $\sqrt{a}$ (c'est la méthode de Newton).

Écrire une fonction prenant en argument a et n et calculant u_n.

Exercice 2. ** Calcul approché de racines 2 : suites récurrentes croisées

Soit $a \in \mathbb{R}^{+*}$. Soient $(u, v) \in (\mathbb{R}^{\mathbb{N}})^2$ les deux suites définies par : $\begin{cases} u_0 = 1 \text{ et } v_0 = a \\ \forall n \in \mathbb{N}, u_{n+1} = \frac{2u_n v_n}{u_n + v_n} \text{ et } v_{n+1} = \frac{u_n + v_n}{2} \end{cases}$

(On démontre facilement que ces deux suites convergent vers $\sqrt{a}$.)

1. Écrire deux fonctions récursives u et v pour calculer u_n et v_n en fonction de n.
2. Jusqu'à quelle valeur de n obtenez-vous un résultat en temps raisonnable ? Comment obtenir une version efficace ?
3. (***) On démontre (cf TD de maths sur les suites) que $\forall n \in \mathbb{N}$, $u_n \leqslant \sqrt{b} \leqslant v_n$. Écrire une variante de la fonction précédente qui prend en entrée non plus n mais un réel $\varepsilon \in \mathbb{R}^{+*}$ et qui renvoie un couple (u_n, v_n) qui encadre $\sqrt{a}$ avec une précision d'au moins ε.

Exercice 3. * Factorielle et dépassement de mémoire

Reprendre la fonction `factorielle` écrite en cours.

1. Écrire une fonction récursive prenant deux entiers x et n et renvoyant x^n.
2. Quelle est la plus grande valeur de n pour laquelle $n!$ est calculable ?
3. Votre version de Caml fonctionne-t-elle en 32 bits ou en 64 bits ?

Exercice 4. **! Coefficients binomiaux

On souhaite écrire une fonction `coeffBinomial` prenant deux entiers n et p et renvoyant $\binom{n}{p}$. On propose 3 méthodes. Les essayer, et identifier la plus efficace des trois.

1. Écrire une version en utilisant une fonction `factorielle n` .
2. Écrire une version récursive en se basant sur la relation de Pascal.
   Concernant cette version :
   (a) Étudier le calcul de $\binom{7}{3}$: remplir un triangle de Pascal en inscrivant dans chaque case combien de fois le coefficient correspondant a été calculé. En déduire le nombre total d'appels récursifs à la fonction qui a été effectué.
   (b) Montrer que de manière générale, le nombre d'addition pour calculer $\binom{n}{p}$ grâce à la relation de Pascal est supérieur à $\binom{n}{p} - 1$.
3. Enfin, écrire une troisième version en démontrant puis utilisant la formule (dite « du pion ») : $\forall (n, p) \in (\mathbb{N}^*)^2$, $\binom{n}{p} = \frac{n}{p}\binom{n-1}{p-1}$.
   Quelle est la complexité de cette dernière version ?

Exercice 5. **! Tours de Hanoï

Soit $n \in \mathbb{N}^*$. On dispose de n anneaux $A_0, ..., A_{n-1}$ qu'on peut empiler sur trois tours T_0, T_1, T_2. Les anneaux sont de taille strictement croissante : $A_0 < ... < A_{n-1}$.

Au début du jeu, tous les anneaux sont sur T_0, rangés dans l'ordre décroissant (A_{n-1} en bas, A_0 en haut).

Le but du jeu est de déplacer tous les anneaux vers la tour T_1, mais en s'assurant qu'à chaque instant, chaque anneau repose sur un anneau plus gros.

1. Résoudre le problème pour $n = 2$ et 3.
2. Supposons qu'on soit capable de déplacer le bloc des $n - 1$ plus petits anneaux d'une colonne vers une autre. Comment pourrait-on en déduire une méthode pour déplacer les n anneaux ?

3. En déduire un algorithme récursif pour résoudre le problème. L'écrire tout d'abord en français, puis le rédiger en Caml en utilisant des instructions de type `print_int i; print_string "->"; print_int j;`.
4. Calculer la complexité de l'algorithme.
5. (***) Démontrer que l'algorithme est optimal, c'est-à-dire que tout algorithme permettant de résoudre le jeu à une complexité d'au moins $2^n - 1$ anneaux déplacés.

Exercice 6. **! Fibonacci efficace : utilisation d'une fonction auxiliaire

Soit $F \in \mathbb{N}^{\mathbb{N}}$ telle que $F_0 = 0, F_1 = 1$ et $\forall n \in \mathbb{N}$, $F_{n+2} = F_{n+1} + F_n$.

On a vu en cours que la programmation récursive naïve de cette suite est inefficace. On propose ci-dessous deux méthodes pour résoudre ce problème. Pour chacune on aura besoin de définir une fonction auxiliaire.

1. **Fonction auxiliaire qui utilise un argument supplémentaire :**
   On utilise une fonction `fiboAux` qui prend en entrée deux termes consécutifs f_k et f_{k+1} et un entier n, et qui renvoie f_{k+n}.
2. **Fonction auxiliaire qui renvoie une valeur supplémentaire :**
   On utilise une fonction auxiliaire `fiboAux` qui renvoie non pas seulement F_n, mais le couple (F_n, F_{n+1}).

Écrire les deux fonctions correspondantes, et calculer leur complexité.

Exercice 7. ** Dichotomie

1. Programmer l'algorithme de dichotomie pour rechercher un encadrement d'une zéro d'une fonction vérifiant les hypothèses du théorème des valeurs intermédiaires.
2. Quelle est l'opération utilisée dans votre code la plus coûteuse en temps ?
3. Vérifier que votre programme n'effectue jamais deux fois cette opération sur les mêmes valeurs, et si ce n'est pas le cas, le corriger.

Exercice 8. ** Persistance d'un entier

Pour tout $n \in \mathbb{N}$, nous noterons $p(n)$ le produit des chiffres de n (en base 10).

1. Programmer la fonction p.
2. Démontrer que pour tout $n \in \mathbb{N}$, le nombre de chiffres de $p(n)$ est strictement inférieur au nombre de chiffres de n.
3. On appelle persistance d'un entier n le plus petit entier k tel que $p^k(n)$ n'a plus qu'un seul chiffre. Écrire une fonction qui calcule la persistance d'un entier.
4. Une conjecture prétend que la persistance d'un entier est toujours inférieure à 11. Écrire une fonction prenant en entrée un entier N et qui vérifie cette conjecture sur $[\![0, N[\![$.
5. (***) On peut augmenter la rapidité du programme précédent en gardant en mémoire la persistance de tous les entiers déjà traités.

Exercice 9. * Nombres de Catalan**

1. Écrire une fonction `somme` prenant en entrée un entier n et une fonction $f : \mathbb{N} \to \mathbb{N}$ et qui calcule $\sum_{k=0}^{n} f(k)$.
2. Pour tout $n \in \mathbb{N}$, on note c_n le nombre de manière d'arranger correctement n paires de parenthèse. Autrement dit il s'agit de suites formées de n fois le symbole `'('` et n fois le symbole `')'` telles que pour tout $i \in [\![0, n[\![$, parmi les i premiers symboles il y a au moins autant de `'('` que de `')'`.
   Les nombre $(c_n)_{n\in\mathbb{N}}$ s'appellent les nombre de Catalan.
   Justifier que $\forall n \in \mathbb{N}$, $c_n = \sum_{k=0}^{n-1} c_k c_{n-1-k}$.
3. En déduire une fonction prenant en entrée n et calculant c_n.

Exercice 10. * Opérations sur les fonctions**

Pour chacune des question suivantes, préciser le type de la fonction Caml écrite.

1. Écrire une fonction `sommeFonction` prenant en entrée deux fonction f et g de $\mathbb{Z}$ dans $\mathbb{Z}$ et renvoyant la fonction $f + g$.
2. Écrire une fonction `compose` prenant en entrée deux fonction f et g telle que l'ensemble d'arrivée de g est l'ensemble de départ de f et qui renvoie $f \circ g$.
3. Définir la fonction `plusUn` : $x \mapsto x + 1$ et la fonction carré. Puis en utilisant les fonctions précédentes, définir la fonction $x \mapsto (x + 2)^2 + x + 1$.

Quelques indications

1 On rappelle que les opérations sur les flottant en Caml sont `+.` `-.` `*.` `/.`.
Si votre programme ne parvient pas à dépasser $n = 30$ améliorez-le !

3 Dans une version 32 bits de Caml, le plus grand entier possible est $2^{30} - 1$. Dans une version 64 bits c'est $2^{62} - 1$.

4 5) Attention à l'ordre des opérations...

5 3) Il va falloir écire une fonction récursive plus générale que l afonction intialement souhaitée : mettre en argumente supplémentaire la tige de départ et la tige d'arrivée.

5) Pour déplacer n anneaux : il va falloir déplacer le plus gros. Ce qui impose : personne au dessus du plus gros, et une tige vide. Donc les $n-1$ autres anneaux ont été déplacés sur la troisième tige, le coût étant $\geqslant C_{n-1}$ par récurrence.

7 On peut écrire une fonction auxiliaire qui prend en argument supplémentaire les valeurs de f aux bords de l'intervalle de recherche.

9 Dans la formule, k représente la position où se ferme la première parenthèse ouverte.

10 On peut définir une fonction dans une autre :

```
let f x =
   let g y = ...  in
   ...
```

Ou alors utiliser l'analogue du « lambda » de Python, qui s'appelle ici **`fun`** et s'utilise ainsi : **`fun`** `x -> image_de_x` .