C. Charignon

Table des matières

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 la fonction est clairement apparent dans la construction de la fonction.
- Tout se définit avec la commande `let` : variables, constantes, ou fonctions. Sur ce point, c'est d'ailleurs Python qui est bizarre : une fonction est un objet comme les autres.
- Ce sont les parenthèses qui définissent les blocs, et non plus l'indentation. On peut également utiliser `begin .... end;`
- Fin d'une définition : deux points-virgule.
- L'opérateur de comparaison se note `=`. Il n'y a pas de `==`, une affectation étant clairement signalée par un `let`.
- Les arguments d'une fonction s'écrivent tous à la suite, séparés uniquement par un espace. Même pas de parenthèses autour.
- 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 `/.`.
  Remarque : les comparaisons fonctionnent quand même pour des types différents : `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.

Il est très facile de définir une suite vérifiant une relation de récurrence en suivant directement sa définition mathématique.

2.1 Exemples

- factorielle
- pgcd
- recherche dichotomique

2.2 Complexité et preuve

Prouver qu'une fonction récursive est correcte est beaucoup plus pratique que pour une fonction contenant des boucles. Il suffit d'utiliser une récurrence. On rédigear typiquement ainsi : «Pour tout $n \in \mathbb{N}$, posons $P(n)$: « `f n` termine et renvoie... » ».

Quant au calcul de complexité, il se ramène à l'étude d'une suite vérifiant une relation de récurrence.

Étudions les exemples précédents.

2.3 Exemple de mauvaise fonction récursive

cf exercice : 1

L'exemple typique de mauvais emploi de la récursivité est le calcul d'une suite récurrente double, par exemple 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).

La 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 est classique : on 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.

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 $\alpha = 0$, ce qui n'est pas réaliste : la complexité tendrait vers 0.)
Remarque : Bien sûr, il est très facile de 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.

N.B. Plus simple pour se débarrasser de la constante : 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 Quelques détails sur le langage

3.1 Portée des liaison

L'instruction `let x=... in` permet de définir `x`. Le terme variable n'est pas vraiment adapté en Caml, car une fois défini, ce `x` ne pourra pas être modifié. Il existe de véritable variables modifiables, mais nous en parlerons plus tard. Le terme technique employé est « liaison entre un identifiant et une valeur », ou plus simplement « identifiant ».

Après un `let maVariable = ... in` , l'identifiant `x` reste défini dans l'expression suivant le "in".

Remarque : Il y a un détail technique : c'est que la séquence est prioritaire sur le in. De même que `2+ 3* 2` est automatiquement compris comme `2+ (3*2)`. Par exemple, si `t` est un tableau, la formule `let x = 2 in t.(0) <- x ; x+1` est automatiquement comprise comme :

`let x = 2 in (t.(0) <- x ; x+1)`.

Par contre dans l'exemple :

```
for i=1 to 12 do
    let x=i in
        blabla
done;
x
```

Le x final n'est pas défini. La définition du x s'arrête au "done".

3.2 Fonctions à plusieurs variables

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;;
```

3.3 La séquence

On rappelle qu'on appelle "séquence" le fait de passer à l'opération suivante. En caml c'est `;`, et en python c'est juste passer à la ligne.

Sous Caml, toute expression a un type. Une expression qui ne renvoie rien (donc une « instruction ») est de type "`unit`". Typiquement, une affectation (`x:= !y`) est de type `unit`.

Il faut considérer que `;` est un simple opérateur. Son type est `unit -> a -> a`, ce qui veut dire qu'il prend une expression de type `unit` en premier, puis une expression de type quelconque ensuite, et que le résultat est de même

type que celui de la deuxième expression.

Exemples :

- `2;3` : Mal défini.
- Si x est un entier : `print_int x ; 2*x` : De type "int".
- Si t est un tableau de flottants : `t.(0)<- 1.2 ; t.(0)` est de type "float". Ou encore `t.(0)<- 1.2 ; t` est de type "float array".

3.4 Variables et constantes

- Une instruction de type «`let x=2;;`» définit une constante. Ainsi, x ne sera plus modifiable dans la suite.
  Remarque : On rappelle que le terme exact est de parler d'une "liaison identifiant-valeur". Dans l'exemple, la valeur sera l'entier 2, l'identifiant est x.
- Pour définir une variable, on utilise : `let x=ref 2;`. On dit ici que x est une référence à un entier.
  Et alors pour accéder au contenu de la variable, rajouter un point d'exclamation : `!x`.
  Et pour modifier le contenu, utiliser `:=`. Par exemple pour incrémenter x de 1 : `x:= !x +1;`. [1]
- Enfin, le mot clé `and` permet de définir deux identifiants à la fois :
  `let x=... and y=... in (....);`
  Remarque : Dans le `and`, les deux identifiant sont vraiment définies *en même temps*. Par exemple `let x=2 and y=x+1;` ne fonctionne pas car x n'est pas encore défini au moment de définir y.

Deuxième partie
Exercices

1. Ceci dit pour incrémenter une référence à un entier, on dispose de la commande `incr`.

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 $\forall n \in \mathbb{N}$, $u_{n+1} = \dfrac{u_n}{2} + \dfrac{a}{2u_n}$. 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 ?

Exercice 3. * Factorielle et dépassement de mémoire

1. Écrire une fonction prenant en argument un $n \in \mathbb{N}$ et calculant $n!$.
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.
3. Étudier le calcul de $\binom{7}{3}$ par la fonction précédente : 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.
4. Montrer que de manière générale, le nombre d'addition pour calculer $\binom{n}{p}$ grâce à la relation de Pascal est supérieur à $\binom{n}{p} - 1$.
5. Enfin, écrire une troisième version en démontrant puis utilisant la formule (dite « du pion ») : $\forall (n, p) \in (\mathbb{N}^*)^2$, $\binom{n}{p} = \dfrac{n}{p}\binom{n-1}{p-1}$.

   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_1, 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. On donnera la suite d'opérations de type `bouge i j` à effectuer pour gagner.
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. Les nombre $c_0, c_1, ...$ s'appellent les nombre de Catalan.
   Justifier que $\forall n \in \mathbb{N}$, $c_n = \sum_{k=0}^{n-1} c_k c_{n-1-k}$.
   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. Écrire une fonction `maxi` qui prend en entrée deux fonction f et g et qui renvoie la fonction $\max(f, g)$.
4. 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 !

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 :

```
1         let f x =
2            let g y = ...  in
3            ...
4
```

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