Quelques particularités du langage OCaml

Cours – TP

I De la différence avec Python

I.1 Caml est un langage fonctionnel

Trois paradigmes de programmation importants sont les trois suivants :

- Programmation impérative : en informatique, il n'y a que des suites d'instructions.
- Programmation fonctionnelle : en informatique, il n'y a que des fonctions.
- Programmation objet : en informatique, il n'y a que des objets.

Évidement, se restreindre de façon exclusive à l'un de ces trois paradigmes n'est guère pertinent. Ainsi, **Python comme Caml supportent ces trois styles de programmation**.

L'idée de l'enseignement en classe préparatoire est :

1. d'utiliser Python pour bien illustrer la programmation impérative ;
2. d'utiliser Caml pour bien illustrer la programmation fonctionnelle.

I.2 Caml est un langage fortement typé

La plupart des langages informatiques sont **typés**, c'est-à-dire que les termes du langage ont tous un type.

- Python est **faiblement typé** : tous les termes ont un type mais
  - le type de retour d'une fonction peut changer,
  - le type des éléments d'une même liste peut changer,
  - des conversions de type automatiques sont possibles,
  - ⋮
- Caml est **fortement typé** : rien de ce qui précède n'est possible !

I.3 Quelques points de syntaxe

Le document d'accompagnement de ce chapitre est encore plus important que le chapitre lui-même : c'est l'aide mémoire que j'ai tapé de mes blanches mains et qui répertorie les points de syntaxe les plus importants en Caml. Le plus important est de pouvoir déclarer une variable avec sa valeur : on utilise pour cela la syntaxe `let ... = ...`.

II Quelques types en Caml

II.1 Le type `int`

C'est le type des entiers relatifs. Mais en fait non. Parce qu'en OCaml, conformément à ce que vous avez appris en informatique de tronc commun et contrairement à ce que fait Python, les entiers sont codés sur un nombre limités de bits (normalement 63, mais possiblement 31 suivant le processeur). Donc en fait `int` correspond plutôt à l'ensemble des entiers modulo 2^{63} qu'on note $\mathbb{Z}/2^{63}\mathbb{Z}$ en mathématiques. Voir l'aide mémoire pour la liste des fonctions les plus importantes sur les entiers.

Exercice 1

1. Déclarer une variable `a` de valeur `4611686018427387903`. *Passion : copier-coller.*
2. Comparer `a-1` et `a+1` à l'aide de `<`, `=` et `>`. Les afficher pour comprendre ce qui se passe !

Remarque 1 : Le plus grand entier représentable s'appelle `max_int`. Si les entiers sont représenté sur 63 bits (ce qui est normalement le cas sur votre version de OCaml), cet entier est `a`.

II.2 Le type `float`

C'est le type des flottants. Eux aussi sont codés sur un nombre limité de bits, normalement 64. Voir l'aide mémoire pour la liste des fonctions les plus importantes sur les flottants.

À cause du typage fort (ou grâce à celui-ci suivant le point de vue), les opérations sur les flottants ne sont pas les mêmes que sur les entiers !

Exercice 2 Comparer les flottants `4611686018427387902.` et `4611686018427387904.` à l'aide de `<`, `=` et `>`. Expliquer.

II.3 Le type `bool`

C'est le type des booléens (true, false). **Attention**, c'est **t**rue et **f**alse sans majuscule, contrairement au choix fait en Python. C'est l'occasion de préciser qu'en Caml les majuscules sont réservées aux classes et aux constructeurs. Voir l'aide mémoire pour la liste des fonctions les plus importantes sur les booléens.

Autre remarque : le mot-clé `and` ne correspond pas au "et logique", il est réservé à autre chose.

Comme on a commencé à le voir, en Caml tout type `'a` est muni d'opérations `<`, `=` et `>` de type `'a →'a →bool`.

Exercice 3 Vérifier que && et || ont bien la table de vérité attendue.

II.4 Caractères et chaînes de caractères

Contrairement à Python (et c'est mieux), Caml distingue les **caractères** des **chaines de caractères**. Le type des caractères est `char`, celui des chaînes de caractères est `string`. Voir l'aide mémoire pour la liste des fonctions les plus importantes sur les caractères et les chaînes de caractères.

Exercice 4

1. Stocker la chaîne de caractère `"Hello, world !"` dans une variable `bonjour`.
2. Extraire le cinquième caractère. Est-ce un `'o'` ? Pourquoi ?
3. Définir une chaine de caractère composée de mille `'a'`.

II.5 Le type `'a → 'b`

C'est le type des fonctions, ou plutôt ce sont les types des fonctions. En Python il n'y a qu'un seul type pour toutes les fonctions mais le typage fort interdit totalement ceci en Caml.

On peut définir des fonctions en Caml avec le mot clé `function` (c'est l'équivalent du `lambda` de Python) mais on a déjà vu qu'on pouvait définir des variables, et donc en particulier avec le mot-clé `let`. On peut aussi définir des fonctions non récursives avec le mot-clé `let`.

Remarque 2 : En Caml l'application d'une fonction `f` à un argument `a` se note `f a`. Parenthèses non nécessaires.

Exercice 5 Écrire une fonction `carre : float→float` qui calcule le carré de son argument.

Remarque 3 : Si on veut écrire une fonction de deux arguments, on écrira plutôt une fonction à un argument qui retourne une fonction à un argument, comme ici pour une fonction `plus : int→int→int` correspondant à l'addition :

```
let plus x y = x+y;;
```

Un mécanisme important en Caml est le **filtrage** (*pattern-matching*) : on peut distinguer la forme d'un argument à l'aide de la syntaxe `match ... with ...`. Voir l'aide-mémoire.

Exercice 6 Voyons si on a compris comment ça marche :

1. Écrire, sans utiliser if, une fonction `chi0 : int → int` qui à 0 associe 1 et à tous les autres entiers associe 0.
2. **Réécrire** la fonction || sans utiliser if. L'appeler `ou`.

Et enfin, surtout, surtout : on peut définir des fonctions récursives avec le mot-clé `let rec`.

Exercice 7 Écrire une fonction récursive `fact : int → int` permettant de calculer la factorielle de son argument.

II.6 Le type `unit`

C'est le type de "rien". L'astuce utilisée en informatique pour dire qu'une fonction ne renvoie rien, c'est de dire que cette fonction renvoie quelque chose, qui est "rien", et qu'on note () en Caml.

Exercice 8 1. Faire afficher le type de `()`.

2. Écrire une fonction prenant un argument de type arbiraire, et ne retournant rien.
3. Écrire une fonction constante prenant 0 argument et retournant l'entier 2.
   Attention, ce n'est pas pareil qu'une variable égale à 2.

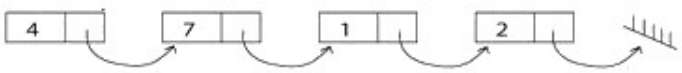
En général, une fonction qui ne renvoie rien fait quand même quelque chose quelque part : on dit qu'elle a un **effet de bord**. Un exemple d'effet de bord facile à illustrer c'est l'écriture sur un fichier ou l'affichage sur la sortie standard.

Exercice 9

Écrire une fonction sans argument, qui ne retourne rien, mais qui a pour effet de bord d'afficher le message `Hello, world !` sur la sortie standard. **Attention, ça n'a rien à voir avec une fonction qui retourne une chaîne de caractères !**

II.7 Le type `'a List`

Le type `'a list` qui correspond aux **listes chaînées** d'éléments de type `'a`. Une liste chaînée d'éléments est ou bien la liste vide, ou bien un couple formé d'un élément (sa tête) et d'un pointeur sur une liste chainée (sa queue).



Avantage : on peut facilement insérer un élément au début (en tête) d'une liste chaînée.
Inconvénient : pour accéder au $i^{\text{ième}}$ élément d'une liste chainée, il faut suivre i pointeurs.
Voir l'aide mémoire pour la liste des fonctions les plus importantes sur les listes.

À cause du typage fort (ou grâce à celui-ci suivant le point de vue), tous les éléments d'une liste doivent avoir le même type !

Il y a deux types de `'a list` :

— la liste vide `[]` ;

— les listes non vides de la forme `h::t` où `h` est l'élément de tête de la liste (de type `'a`), et `t` la queue de la liste (de type `'a list`) correspondant à la liste de départ privée de son élément de tête.

Ceci permet le pattern-matching sur les listes, youpi.

Exercice 10 On va commencer les choses sérieuses et essayer de comprendre comment sont codées certaines fonctions bien utiles, c'est un classique de concours de nous demander de les réécrire.

1. Que fait `List.length` ? Écrire une fonction (nécessairement récursive) `longueur` ayant le même comportement.
2. Que fait `List.map` ? Écrire une fonction (nécessairement récursive) `mapper` ayant le même comportement.

II.8 Le type `'a array`

C'est le type des **tableaux** d'éléments de type `'a`. Lorsqu'on définit un tableau, on doit spécifier sa longueur et celle-ci ne changera plus jamais. Un nombre de cases mémoire adjacentes correspondant est alors réservé pour les éléments.
Avantage sur les listes : l'accès au $i^{\text{ième}}$ élément est gratuit.
Inconvénient : le nombre d'éléments est figé, on ne peut pas le changer.
Voir l'aide mémoire pour la liste des fonctions les plus importantes sur les tableaux.

À cause du typage fort (ou grâce à celui-ci suivant le point de vue), tous les éléments d'un tableau doivent avoir le même type !

Exercice 11 Écrire une fonction `recherche_dichotomique : 'a → 'a array → bool` qui prend comme argument un élément de type `'a` et un tableau de type `'a array` **supposé trié**, et rend comme résultat le booléen indiquant si l'élément est ou pas dans le tableau, en utilisant une recherche dichotomique. On peut le faire avec une boucle while (présentées plus bas), et c'est d'ailleurs plus naturel, mais faisons l'effort de le coder par récursivité.

III Quelques aspects impératifs

III.1 Définition d'une variable

On l'a vu, pour définir une variable on utilise le mot-clé `let`. Pour une variable locale c'est `let ... in ...` et pour une variable qui est une fonction définie récursivement c'est `let rec`.

III.2 Définitions simultanées et récursivité croisée

On peut définir des variables **simultanément** à l'aide du mot-clé `and`. Encore faut-il que cela ait un sens.

Exemple : Un exemple où ça n'a pas de sens et où il faut emboîter les `let ... in` :

Quel intérêt alors ?

Exemple : Un exemple où on **doit** utiliser le `and` : la récursivité croisée.

III.3 Modification d'une variable

Bien sûr, l'intérêt d'une variable en programmation impérative c'est d'être modifiée à chaque instant.
Et là il y a, comment dire, une mauvaise nouvelle :

On ne peut pas modifier une variable locale en Caml.

Ce n'est pas une blague, on ne peut vraiment pas modifier la valeur d'une variable en Caml, c'est un aspect fonctionnel de Caml. Heureusement, il y a aussi une **bonne** nouvelle :

On peut modifier la valeur d'une référence en Caml.

Une référence c'est quoi ? C'est essentiellement la même chose qu'un pointeur. Autrement dit : c'est l'adresse mémoire d'une case où il y a quelque chose. Lorsqu'on définit une variable qui est une référence, on ne pourra jamais modifier la valeur de cette variable donc **l'adresse mémoire de la case** où il y a quelque chose ne pourra jamais être changée. Par contre, on peut modifier **le contenu de la case**.

Exercice 12 Créer une variable `n` de type `int ref` pointant vers la valeur entière de votre mois de naissance (entre 1 et 12). Modifiez-la en lui ajoutant votre jour de naissance (entre 1 et 31).

III.4 Boucles !

Caml n'étant pas un langage **purement** fonctionnel, on peut – heureusement ! – y écrire des boucles `for` et `while`.
La syntaxe est rappelée dans l'aide-mêmoire.

Exercice 13 Écrire une fonction `v2 : int → int` qui calcule la valuation 2-adique d'un entier. C'est très facile (plus facile) à faire par récursivité mais faisons l'effort d'utiliser une boucle.

Remarque 4 : Les tableaux aussi sont des pointeurs. On peut modifier la $i^{\text{ième}}$ case d'un tableau (cf aide-mémoire).

Exercice 14 Créer un tableau de longueur 100 contenant les entiers de 0 à 99 dans cet ordre. On peut le faire par récursivité mais faisons l'effort d'utiliser une boucle.

Exercice 15 Réécrire une fonction `factorielle : int → int` avec une boucle et une variable de type `int ref`.

Remarque 5 : On n'a pas de sortie de boucle anticipée en Caml. Si on veut en faire quand même, on peut contourner cette particularité en rattrapant une exception. On y revient dans le paragraphe sur les exceptions.

III.5 Structures conditionnelles

On a déjà insisté sur l'efficacité de Caml pour faire du **filtrage** (*pattern matching*) à l'aide du mot-clé `match`. On peut totalement se passer des `if` quitte à utiliser des filtrages avec garde `match ... when...`.

Il existe toutefois un `if` un Caml qui s'utilise avec la syntaxe `if condition then instruction`$_1$ `else instruction`$_2$.

Exercice 16 Écrire une fonction `degre2 : float → float → float → float list` telle que `degre2 a b c` donne la liste des racines réelles du trinôme $aX^2 + bX + c$.

Attention, écrire `if condition then instruction`$_1$ (sans `else`) est interprété comme `if condition then instruction`$_1$ `else ()`. On ne peut donc l'utiliser que pour `instruction`$_1$ de type `unit`. Mieux vaux éviter.

Attention aussi, après le `then` et après le `else`, on doit ne mettre qu'une seule instruction ! C'est encore un aspect fonctionnel de Caml. Si on veut en mettre plusieurs, le plus lisible pour le correcteur est d'utiliser la syntaxe `begin ... end`.

Exercice 17 Proposer un exemple où il est agréable de pouvoir écrire plusieurs instructions dans une (ou plusieurs) des branches d'une instruction conditionnelle. L'implémenter.

III.6 Exceptions

On peut provoquer des erreurs en Caml. On les appelle des exceptions. Certaines existent déjà par défaut (`Not_found`, `Division_by_zero`, `Failure of string`, ...), et on peut en définir d'autres avec le mot-clé `exception`.
Remarquez que le nom d'une exception commence par une majuscule.

Exercice 18

1. Écrire une fonction `division_euclidienne : int` $\to$ `int` $\to$ `int*int` telle que `division_euclidienne a b` retourne le couple `(q,r)` obtenu avec / et `mod`.
2. Cette fonction provoque une exception pour `b`$= 0$. La modifier à l'aide d'une instruction conditionnelle pour qu'elle retourne le couple $(-1, -1)$ dans ce cas.
3. Plutôt que d'utiliser une instruction conditionnelle, obtenez le même résultat en rattrapant l'exception.