Programmation impérative en Caml

C. Charignon

Il est temps de voir comment on programme de manière impérative (c'est-à-dire avec des boucles et des variables, comme dans le tronc commun avec Python).

Table des matières

Première partie

Cours

1 Introduction

Tout d'abord, il est prouvé que les styles de programmation impératifs et récursifs sont équivalent : tout programme écrit dans un style peut aussi être écrit dans l'autre. Cependant, certain programmes sont plus facilement codés dans un style.

Voici les éléments de programmation utilisés en priorité en programmation récursive :

- Nombreuses fonctions (récursives).
- Les informations sont transmises via des arguments supplémentaires.
- Types persistants.

Et en programmation impérative :

- Boucles.
- Procédures. En Caml, une procédure renvoie `()`, de type `unit` (c'est le `None` de Python).
- Les informations sont conservées dans des « variables ».
- Types modifiables (mutables).

Remarque : J'ai déjà eu l'occasion de le dire mais je le redis : Le mot « variable » a un sens assez flou en informatique et en science de manière générale... Je rechigne presonnellement à l'utiliser pour un objet persistant qui ne peut donc pas varier. Vous avez eu l'occasion de vous en rendre compte : Caml utilise le terme «identificateur»

2 Boucles

Voici la syntaxe Caml des boucles :

```
for i= 0 to n-1 do
    (*faire des trucs de type unit*)
done;
```

```
while condition do
    (*faire des trucs de type unit*)
done;
```

Quelques importants :

- On ne peut couper une boucle « for » au milieu de son exécution. (Pas de return, donc on ne peut pas écrire return au milieu d'une boucle.) Si vous ne savez pas à l'avance combien d'itérations seront nécessaires, utilisez une boucle conditionnelle, ou une fonction récursive.
- Le contenu d'une boucle doit être de type unit, puisqu'il n'est pas possible de renvoyer une valeur au milieu d'une boucle.

  Soit `f :int -> unit` la *procédure* utilisée dans la boucle. Alors le code :

  ```
  for i= 0 to n-1 do
      f i
  done;
  ```

  Équivaut à `f 0; f1; … ; f (n-1);`

  N'oubliez pas qu'entre deux instructions (c'est-à-dire des expressions de type `unit`) il faut mettre un point-virgule.

  Enfin, la boucle entière est donc un élément de type `unit`.
- Pour l'instant, les seules instructions que nous connaissons sont l'affichage (`print_int` et variantes). C'est assez anecdotique : en général, nous utiliserons des instructions qui permettent de modifier une variable mutable. Nous n'avons pour l'instant pas utilisé de variable mutable en Caml. Dans les parties suivantes, nous en présentons les deux types les plus élémentaires : les références et les tableaux.

3 Références

Une référence est la manière la plus simple de créer une variable modifiable. Voici la syntaxe sur un exemple :

```
let ma_variable = ref 0 in (* Je crée une référence à un entier. Type int ref. *)
    ma_variable := 3;  (* Je modifie la variable. C'est une instruction (type unit). Notez
    ↪ bien le :=  *)
    print_int !ma_variable (* J'affiche le « contenu » de la variable. Notez le point d'
    ↪ exclamation pour l'obtenir *)
;;
```

Pensez qu'une référence est un tableau à une case.

4 Tableaux

Les tableaux fonctionnent comme en Python, à ceci près qu'ils sont de longueur fixe, et qu'ils ne peuvent contenir que des éléments du même type. Il s'agit donc en fait plutôt de tableaux de numpy.

Voici un petit exemple, voir la fiche de syntaxe donnée en début de semestre pour les autres fonctions utiles.

```
let exemple = [|1;2;3|] ;;
exemple.(0);; (* Accéder à un élément *)
exemple.(0) <- 2;; (* modifier un élément *)
```

5 Copie superficielle

Comme en Python, les objets de type mutable ne sont pas copiés par défaut, et peuvent être modifiés par une procédure.

Exemple :

```
let augmente x=
  x:= x+1
;;

let x=ref 2 in
    augmente x;
    x;;
```

Remarque : Le fonction `augmente` existe déjà dans Caml : c'est `incr`.

Exemple 2

```
let x=[|1;2;3|] in
    let y=x in
        y.(1)<- 0;
        x;;
```

:

6 Création d'un type enregistrement

Il s'agit de la deuxième méthode pour créer un type en Caml (la première était le type « somme » vu au chapitre sur les arbres).

```
type contact = { nom: string ; adresse: string ; telephone: int};;
```

Pour créer un élément de ce type :

```
let MX = {nom = "monsieur X"; adresse = "rue Z"; telephone=1234567890};;
```

Pour récupérer la valeur d'un champ, on utilise le ".", ce qui rappelle Python : `MX.nom;;`

A priori les types ainsi créés sont persistants. Pour les rendre modifiable, rajouter «`mutable`» :

```
type contact = { nom: string ; mutable adresse: string ; mutable telephone: int};;
```

Ici on considère que le nom n'est pas susceptible de changer : seules l'adresse et le téléphone sont mutables.

Alors on peut modifier un champ par la commande `<-` (même syntaxe que pour un tableau, mais pas pour une référence).

```
MX.telephone <- 0123456789;;
```

Remarque : Le type référence est défini ainsi :

```
type 'a ref = {mutable content : 'a };;
```

7 Exemple : tableaux redimensionnables

Exercice 1. ** Tableaux redimensionnables (à la Python)

Le type « `List` » de Python est un tableau que l'on peut agrandir si besoin. Une telle structure est appelée «tableau redimensionnable», ou «tableau dynamique».

On propose de l'implémenter par un type enregistrement :

```
type 'a tabRedim ={mutable longueur : int; mutable donnees : 'a vect};;
```

Ainsi, nos tableaux redimensionnables seront constitués d'un entier appelé «longueur», et d'un vecteur appelé «donnees». Le principe est que le tableau «donnees» sera en général plus grand que nécessaire : les cases inemployées à la fin pourront servir lorsqu'on voudra agrandir le tableau.

1. Écrire une fonction `creeTabRedim` prenant en entrée un entier n et un élément x et renvoyant un tableau redimensionnable de n cases contenant des x.

   Le tableau `donnees` créé sera créé deux fois plus grand que nécessaire pour laisser de la place aux agrandissements ultérieurs.
2. Écrire les fonctions de lecture et écriture.
3. Écrire une fonction pour rajouter un élément à la fin d'un tableau. La plupart du temps, il suffira d'incrémenter la longueur, et d'écrire l'élément à rajouter dans la case vide suivante de `donnees`.

   Cependant, si `donnees` est déjà plein, on créera un nouveau tableau, deux fois plus grand, et on y recopiera les données précédentes.
4. Quelle est la complexité de l'ajout d'un élément à la fin du tableau ?
5. Soit $n \in \mathbb{N}$. On part d'un tableau vide. Et on y rajoute l'un après l'autre 2^n éléments. Quelle est la complexité totale ?

   On dit que l'ajout d'un élément dans un tableau a un coût *amorti* en $O(1)$.

1.
2.
3.
4. Comptons par exemple le nombre d'écritures dans le tableau. Ainsi il faut la plupart du temps 1 écriture de tableau pour rajouter un élément. Cependant, lorsqu'il faut recopier l'ancien tableau dans un nouveau, cela fait $n+1$.
5. Voici l'état de notre tableau au cours des premières étapes (on suppose ici qu'un tableau vide est créé déjà avec une case vide)
   - Une case vide
   - Une case pleine
   - Une case pleine, une vide
   - Deux cases pleines
   - Trois pleines, 4 vides
   - 4 pleines
   - 5 pleines, 3 vides
   - 8 pleines

- 9 pleines, 7 vides
- …
- 2^n pleines

Il y a eu $n-1$ agrandissements. L'agrandissement pour passer de 2^i cases à 2^{i+1} cases nécessite 2^i copies d'anciennes données. Le coût total des agrandissements est donc de :

$$\sum_{i=0}^{n-1} 2^i$$

qui vaut $2^n - 1$.
Ce à quoi il convient de rajouter le coût des écritures des nouveaux éléments, soit 2^n.
Coût total : $2^{n+1} - 1$.
Ainsi, le coût moyen d'ajout pour chaque élément est $\frac{2^{n+1}-1}{2^n}$, soit $2 - 2^{-n}$, donc environ 2.
On dit que le coût amorti est en $O(1)$. Nous avons donc le même ordre de grandeur de complexité, même si en moyenne le coût est double de celui d'un tableau simple.

Deuxième partie
Exercices

Exercice : programmation impérative en Caml

1 Type enregistrement

Exercice 2. ** Calculs physiques et homogénéité

On propose de calculer des grandeurs physiques : il s'agira de flottants munis d'une unité. L'unité sera représentée par un tableau de 7 éléments, dans l'ordre suivant `[|m;kg;s;A;K;cd;mol|]`. Par exemple, une accélération est en $m.s^{-2}$: son unité sera représentée par le tableau `[|1;0;-2;0;0;0;0|]`.

Une grandeur physique est alors formée d'un flottant et d'une unité. On utilise un enregistrement :

```
type grandeur = { valeur: float; unite : int array};;
```

Par exemple la constante g pourra être définie par :

```
let g ={ valeur = 9.81; unite = [|1;0;-2;0;0;0;0|] };;
```

1. Écrire des fonction d'addition, opposé, inverse et produit pour ce type. On renverra des messages d'erreur lorsque les unités sont incompatibles.
2. *Exemple :* Un objet de masse $2kg$ et de volume $0.1m^3$ est plongé dans l'eau. Calculer son accélération.
   Écrire un prédicat `flotte` prenant en entrée la masse et le volume d'un objet et qui détermine si il flotte ou pas.
3. Écrire enfin les fonction exponentielle et cosinus.

Exercice 3. ** Jouons avec le type unit

La fonction `List.iter` permet d'appliquer une *procédure* à chaque élément d'une liste.

1. Utilisez-la pour créer une procédure `affiche_liste` qui affiche le contenu d'une liste d'entiers.
2. Reprogrammez-la.
3. Reprogrammez-la en vous basant sur un `List.fold`.

Quelques indications

2 1. Écrire une fonction pour additionner deux tableaux termes à termes.
2. La poussée d'Archimède exerce une force « vers le haut » de norme égale au poids d'eau déplacé par l'objet.
3. Les fonctions exponentielle et cosinus doivent prendre des nombres sans unité en argument. Renvoyer une erreur si ce n'est pas le cas.

Quelques solutions

2