Programmation impérative en OCaml

I. Éléments de programmation impérative

1. Premiers éléments

Type `unit`

Nous allons beaucoup utiliser le type **`unit`** qui correspond à l'absence de valeur. C'est très utile pour des fonctions qui ont un effet, mais ne renvoient pas de résultat. Par exemple, la fonction `print_int` a pour type **`int -> unit`**, ce qui signifie qu'elle prend en entrée un entier, et ne renvoie rien. Entre temps, elle aura affiché la valeur de l'entier passé en argument.

On peut faire suivre une expression de type **`unit`** par une autre expression (d'un type quelconque) en les séparant par un point-virgule.

```
# print_int ;;      (* affichage d'entier *)
- : int -> unit = <fun>
# print_newline ;; (* passage à la ligne *)
- : unit -> unit = <fun>
# print_int 12 ;;
12- : unit = ()
# print_int 12 ; "poulet" ;;
12- : string = "poulet"
# print_int 12 ; print_newline () ;;
12
- : unit = ()
```

Notons que pour un test **`if ... then ... else`** dont les deux expressions sont de type unit, on peut supprimer le **`else`** si l'on veut ne rien faire si le teste s'évalue à `false`.

```
# if 3 mod 2 = 0 then print_string "bizarre" else () ;;
- : unit = ()
# if 3 mod 2 = 0 then print_string "bizarre" ;;
- : unit = ()
```

Boucles `for`

Commençons par un premier type de boucle, que nous allons illustrer par un exemple :

```
for i = 1 to 10 do
    print_int i ;
    print_newline ()
done ;;
```

Entre le **do** et le **done**, on a une expression de type **unit**. Notons que les bornes sont toujours incluses et que l'on a obligatoirement un pas de 1. On peut aussi aller en décroissant en remplaçant le **to** par **downto**.

Références

Avant de présenter les boucles **while**, présentons les références qui permettent d'avoir des variables dont la valeur évolue au cours du temps.

```
let factorielle n =
    let f = ref 1 in
    for i = 1 to n do
        f := !f * i
    done ;
    ! f
;;
```

- ⋆ On créée une référence en écrivant `ref` suivi de sa valeur initiale (en la mettant dans un **let**, cela permet de lui donner un nom).
- ⋆ On obtient sa valeur en faisant précéder la référence par `!`.
- ⋆ On change sa valeur à l'aide de `:=`.

Boucles **while**

Pour finir, on dispose de la boucle **while** dont la syntaxe est sans surprises.

```
let compte n =
    let k = ref n
    and cnt = ref 0 in
    while !k > 0 do
        incr cnt ; (* remplace cnt := !cnt + 1 *)
        k := !k / 2
    done ;
    !cnt
;;
```

2. Tableaux

De nombreux algorithmes impératifs portent sur des tableaux, voici quelques commandes de base.

```
# let a = [| 4; 8; 15; 18; 23; 42 |] ;;
val a : int array = [|4; 8; 15; 16; 23; 42|]
# a.(3) ;;
- : int = 18
# a.(3) <- 16 ;;
- : unit = ()
# a ;;
- : int array = [|4; 8; 15; 16; 23; 42|]
# Array.length a ;;
- : int = 6
# let b = Array.make 10 true ;;
val b : bool array =
  [|true; true; true; true; true; true; true; true; true; true|]
# Array.init 5 (fun i -> 2 * i + 1) ;;
- : int array = [|1; 3; 5; 7; 9|]
```

II. Analyse d'algorithmes

Nous allons étudier trois propriétés qui sont à la base de l'analyse d'algorithmes.

1. Terminaison

La première question importante est de déterminer si l'exécution de l'algorithme va se terminer ou non, la terminaison étant en général ce que l'on désire.

Notons que ce qui peut entraîner une exécution infinie est la présence de boucles, et plus précisément de boucles **`while`**. Un outil efficace pour prouver qu'une telle boucle termine est d'utiliser un *variant de boucle*.

Définition Un **variant de boucle** associé à une boucle **`while`** est une quantité v qui est un entier naturel dont la valeur dépend de l'état du programme et telle que l'exécution du corps de la boucle fait décroître strictement la valeur de v.

Proposition 1

Toute boucle ayant un variant termine.

Preuve

Par l'absurde, une boucle qui ne terminerait jamais entraînerait, en prenant les valeurs successives de v, une suite d'entiers naturels infinie et strictement décroissante, ce qui est impossible.

Exemple On considère la fonction

```
let appartient v t =
        let n = Array.length t in
        let p = ref 0 in
        while !p < n && t.(!p) <> v do
            p := !p + 1
        done ;
        !p < n
```

Un variant de la boucle est `n - !p`, la boucle admet un variant donc elle termine.

Exercice 1 Prouver la terminaison de la fonction suivante :

```
let mystere n =
    let c = ref 0
    and v = ref n in
    while !v <> 0 do
        c := !c + 1 ;
        v := !v / 2
    done ;
    !c
```

Notons que l'on ne sait pas toujours prouver la terminaison d'un programme en général. Considérons par exemple la fonction suivante, correspondant à la *conjecture de Syracuse* (remarquez au passage l'utilisation fonctionnelle du test).

```
let syracuse n =
    let v = ref n in
    while !v <> 1 do
        v := if !v mod 2 = 0 then
                !v / 2
             else
                (3 * !v) + 1
    done
;;
```

Personne ne sait à l'heure actuelle si cette fonction termine. D'après le mathématicien hongrois Paul Erdős, « les mathématiques ne sont peut-être pas encore prêtes pour de tels problèmes. »

Question 2 Concrètement, que signifie le fait que l'on ne sait pas si cette fonction termine ?

En fait, Alan Turing a prouvé qu'il n'existe pas d'algorithme général permettant de dire si un programme termine ou non. On appelle cela le *problème de l'arrêt*.

2. Correction

Une autre question important est de s'assurer que le résultat de l'algorithme est bien celui que l'on espére. Pour cela, il faut pouvoir énoncer et prouver des propriétés du résultat, ce qui est un peu délicat en présence de boucle `while` où l'on ne sait pas à l'avance combien d'itérations vont être effectuées.

Une méthode pour faire cela est d'utiliser un *invariant de boucle*.

Définition Un **invariant de boucle** est un prédicat P dépendant de l'état du programme tel que si P est vérifié au début du corps de la boucle (et que le test de boucle est vérifié), alors il est aussi vérifé en fin du corps de la boucle.

Proposition 2

Si une boucle admet un invariant P, et que cet invariant est vérifié juste avant l'exécution de la boucle, alors il est encore vérifié en sortie de boucle.

Exercice 3 Considérons le court programme suivant, où est `i` est une référence d'entier :

```
...
i := 0 ;
while !i < 10 do
    i := !i + 1
done ;
...
```

1. Montrer que le prédicat `!i` ≤ 10 est un invariant de la boucle.
2. Quelle est la valeur de `!i` en sortie de boucle ?

Exercice 4 Prouver que la fonction suivante renvoie la factorielle de l'entier passé en argument.

```
let factorielle n =
    let i = ref 1
    and f = ref 1 in
    while !i < n do
        i := !i + 1 ;
        f := !f * !i
    done ;
    !f
;;
```

Invariant pour les boucles `for`

La méthode précédente s'adapte facilement aux boucles **for** en considérant que les programmes suivants sont équivalents.

```
for i = debut to fin do
    instructions
done
```

```
let i = ref debut in
while !i <= fin do
    instructions ;
    i := !i + 1
done
```

3. Complexité

Une fois que l'on s'est assuré que notre algorithme termine et renvoie la bonne valeur, on peut se poser de l'efficacité de notre algorithme. On peut, pour cela, mesurer la quantité de ressources utilisées pour exécuter l'algorithme en fonction de la taille des données en entrée.

- ⋆ On s'intéressera principalement à la complexité **temporelle** qui mesure le temps de calcul nécessaire à l'exécution de l'algorithme, et à la complexité **spatiale** qui mesure la place mémoire nécessaire.
- ⋆ On distingue la complexité dans le meilleur des cas, dans le pire des cas et en moyenne. Seule la complexité dans le pire des cas est au programme, mais on évoquera parfois les deux autres.
- ⋆ La taille des données pourra être :
  - la valeur d'une variable entière ;
  - la taille d'un tableau ;
  - la longueur d'une chaîne de caractères ;
  - le degré d'un polynôme ;
  - etc.
- ⋆ En notant n la taille des données, on parlera de complexité…
  - constante quand elle est en $O(1)$;
  - logarithmique quand elle est en $O(\log n)$;
  - linéaire quand elle est en $O(n)$;
  - linéarithmique quand elle est en $O(n\log n)$;
  - quadratique quand elle est en $O(n^2)$;
  - polynomiale quand elle est en $O(n^a)$ pour un certain $a \geq 1$;

Pour indiquer la classe de complexité, on utilisera souvent les notations de Landau : étant donné deux fonctions f et g définie sur les entiers naturels, on a :

- ⋆ $f(n) = O\big(g(n)\big)$ si la suite $\frac{f(n)}{g(n)}$ est bornée (à partir d'un certain rang) ;
- ⋆ $g(n) = \Omega\big(f(n)\big)$ si $f(n) = O\big(g(n)\big)$;
- ⋆ $f(n) = \Theta\big(g(n)\big)$ si $f(n) = O\big(g(n)\big)$ et $f(n) = \Omega\big(g(n)\big)$.

III. Étude d'algorithmes

1. Algorithme sur les tableaux

Commençons par un algorithme très important, la recherche dichotomique dans un tableau trié.

```
let recherche_dicho t v =
    let a = ref 0
    and b = ref (Array.length t - 1)
    and trouve = ref None in
    while !trouve = None && !a <= !b do
        let c = (!a + !b) / 2 in
        if t.(c) < v
        then
            a := c + 1
        else if t.(c) > v
        then
            b := c - 1
        else
            trouve := Some c
    done ;
    !trouve
;;
```

Question 5 Prouver que l'algorithme termine.

Question 6

1. Prouver que la boucle **while** admet pour invariant :

$$\texttt{!trouve} = \texttt{None} \implies \left(\forall\, p \in [\![0, n-1]\!],\, \texttt{t.}(p) = \texttt{v} \implies \texttt{!a} \leq p \leq \texttt{!b}\right)$$

   où n est la taille du tableau.

2. En déduire la correction de l'algorithme.

Question 7 Déterminer la complexité de l'algorithme, en fonction de n.

2. Algorithmes de tri

Nous allons maintenant étudier deux algorithmes de tri sur des tableaux.

2.1. Tri par sélection

```
let tri_selection t =
    let n = Array.length t in
    for i = 0 to n - 2 do
        let cand = ref i in
        for j = i + 1 to n - 1 do
            if t.(j) < t.(!cand)
            then
                cand := j
        done ;
        if !cand > i
        then begin
            let ti = t.(i) in
            t.(i) <- t.(!cand) ;
            t.(!cand) <- ti
        end
    done
;;
```

Question 8 Quelle est la complexité de l'algorithme (la terminaison est évidente) dans le pire des cas ? Dans le meilleur ?

Question 9

1. Montrer que la conjonction suivant est un invariant de la boucle extérieure :
   - ⋆ le contenu du tableau est une permutation du tableau de départ
   - ⋆ **et** le sous-tableau allant des indices 0 à $i-1$ est ordonné de façon croissante
   - ⋆ **et** pour tout $j \in [\![0, i-1]\!]$ et tout $k \in [\![i, n-1]\!]$, on a `t.(j)` $\leq$ `t.(k)`.
2. En déduire que l'algorithme trie effectivement le tableau.

2.2. Tri par insertion

```
let tri_insertion t =
    for i = 1 to Array.length t - 1 do
        let ti = t.(i)
        and j = ref (i - 1) in
        while !j >= 0 && t.(!j) > ti do
            t.(!j + 1) <- t.(!j) ;
            decr j
        done ;
        t.(!j + 1) <- ti
    done
;;
```

Question 10 Justifier que la boucle `while` termine.

Question 11 Prouver que l'algorithme trie bien le tableau passé en argument, à l'aide d'un invariant judicieux.

Question 12 Quelle la complexité dans le pire des cas ? Dans le meilleur ?