Programmation itérative

MPSI - Prytanée National Militaire

Pascal Delahaye

7 février 2019

La programmation itérative (ou impérative) repose sur une série d'instructions excécutées de façon séquentielle par l'ordinateur. Ces instructions viennent modifier étape par étape les valeurs des variables pour finalement aboutir au résultat souhaité.

Les langages C, FORTRAN, PASCAL, ...utilisent ce style de programmation.
Même si Caml n'est pas un langage impératif (on verra qu'il s'agit d'un langage *fonctionnel*), il possède un certain nombre de fonctions qui permettent de mettre en place ce type de programmation.

Attention : même si les structures exposées dans ce chapitre sont incontournables dans un langage itératif, on les utilisera parfois (voire souvent) avec les autres modes de programmation.

Le choix d'un mode de programmation (en l'absence d'argument d'efficacité en faveur d'un mode ou d'un autre) doit avant tout être guidé par des considérations de simplicité et de clareté. N'oubliez pas que vos programmes seront amenés à être lus et interprétés par un tiers ou par vous-même!!

Enfin, même si le mode de programmation itérative semble au départ le plus naturel, on l'abandonnera progressivement au profit du mode de programmation fonctionnel récursif, beaucoup plus efficace et compréhensible avec un peu de pratique.

1 Les variables utilisées

1.1 Les constantes

Les constantes (définies par `let a = valeur ;;`) sont des variables immuables : leur valeur n'évolue pas au cours d'une session Caml. Les références en revanche sont des variables dont on pourra faire évoluer la valeur.

1.2 Les références

Une référence est en fait une structure de type enregistrement (ou produit) mutable à une seule composante dont le nom est `contents`.

OCaml
```
ref 1;;                          (* instruction *)
- : int ref = {contents = 1}     (* réponse de l'ordinateur *)
```

Plus concrêtement, eune référence correspond à un emplacement réservé dans la mémoire de l'ordinateur dans lequel on place la `valeur_initiale`. On définit une référence par la commande `let a = ref valeur_initiale ;;`
On peut se représenter ce type de variable comme une boîte dont le nom est celui de la référence et dont le contenu est une valeur qui peut varier au cours du temps.

Il est alors possible de :

1. Lire le contenu d'une référence (avec l'opérateur `"!"`)
2. Modifier le contenu de la référence (avec l'opérateur `":="`)

Exercice : 1

Familiarisation avec les références
Taper les instructions suivantes et commenter la réponse de Caml dans le tableau suivant :

Instructions	Réponse	Commentaires
let boite = ref 3 ; ;		
!boite ; ;		
boite :=5 ; ;		
!boite ; ;		
boite :=1.4 ; ;		
!boite ; ;		
boite2 := 4 ; ;		

A retenir ! !

1. Une référence doit être définie (instruction : `let boite = ref 3 ;;`) avant d'être utilisée.
2. On peut modifier le contenu initial d'une référence `boite` par l'instruction `boite :=` *newval;;*
3. Le type de la référence est déterminé par la valeur donnée lors de la définition et ne peut être modifié.
4. On a accès au contenu d'une référence `boite` par l'instruction `!boite`.

Remarque 1.

1. Pour incrémenter une référence `x` de 1, on utilisera donc la commande : `x := !x + 1` ou la commande `incr x`.
2. Si une fonction fait intervenir plusieurs références locales, on pourra utiliser la syntaxe :
   `let a = ref 0 and b = ref 1 in ...`

Instructions	Réponse	Commentaires
`let a = ref 0 and b = ref 1 in !a + !b ;;`		

Remarque 2. Pour faciliter la lisibilité de la ligne d'instruction, on reviendra à la ligne et on alignera les instructions de même niveau.

OCaml
```
let a = ref 0 and
    b = ref 1 in !a + !b ;;
```

Remarque 3. **Attention aux liaisons entre références !**

Instructions	Réponse	Commentaires
let a = ref 3 ; ;		
let b = a ; ;		
b := 5 ; ;		
!a ; ;		
let c = ref !a ; ;		
c := 3 ; ;		
!a ; ;		

Dans cet exemple, `a` et `b` désignent la même boite !...
Ainsi, si l'on change la valeur de la boite par l'intermédiaire de `b`, la valeur de `a` est elle aussi modifiée (et inversement !).
En revanche, `c` désigne une nouvelle boîte qui admet au départ la même valeur que `a`. Si l'on change la valeur de la boîte `C`, la valeur de la référence `a` n'est pas modifiée.

1.3 Les tableaux : `array`

Les tableaux sont une implémentation d'une structure de donnée usuelle appelée *Liste indexée*.
Concrêtement, il s'agit d'un ensemble ordonnée et de taille fixé d'emplacements dans la mémoire de l'ordinateur.
Un tableau est donc une collection ordonnée de références (composantes du tableau) auxquelles on a accès par un système d'indexation. Il sera possible de modifier ces composantes.

v.(0)	v.(1)	v.(2)	...	...	v.(5)

Structure d'un tableau `v`

Remarque 4. Vous remarquerez que les éléments d'un tableau de longueur n sont numérotés de 0 à $n-1$.

La définition d'une structure de données comprend des *fonctions de manipulation*. Il s'agit des fonctions de base indispensables pour travailler avec cette structure. Toutes les autres fonctions agissant sur la donnée sont alors programmées à l'aide de ces fonctions de manipulation.
Pour la structure *array*, les fonctions de manipulation sont les suivantes :

1. Les constructeurs (permettent de construire la donnée) : `v.(i) <- a ; Array.make ; [|...|]`
2. Les fonctions de sélection (permettent de récupérer l'information) : `v.(i) ; Array.length`
3. Les prédicats (permettent de tester la donnée) : `v.(i) = e`

Exercice : 2
Familiarisation avec les fonctions de manipulation des tableaux
Taper les instructions suivantes et commenter la réponse de OCaml dans le tableau suivant :

Instructions	Réponse	Commentaires
let v = [|3 ;-4 ;2 ;7 ;11 ;8|] ; ;		

Instructions	Réponse	Commentaires
Array.length v ; ;		
v.(5) ; ;		
v.(6) ; ;		
v.(0)<-100 ; v ; ;		
let v = Array.make 10 0 ; ;		

A retenir ! !

1. Les tableaux sont définis par `let v = [| ... |] ;;` et les composantes sont séparées par des `";"`.
2. Les composantes d'un tableau sont numérotées de 0 à $n-1$ (si n est la longueur du tableau)
3. Bien que les composantes d'un tableau soit des références, la syntaxe pour appeler leurs valeurs ou pour les modifier diffère de la syntaxe habituelle.
   - On utilise `v.i)` pour appeler la valeur de la $i+1$ ème composante du tableau `v`.
   - On utilise `v.(i)<-3` pour modifier la valeur de la $i+1$ ème composante du tableau `v`.
4. Pour créer un tableau de longueur n, on pourra utiliser la fonction `Array.make`.
5. On a accès à la taille d'un tableau en utilisant la fonction `Array.length`.
   La taille est une caractéristique d'un tableau et ne peut être modifiée.

Remarque 5. **Attention aux liaisons entre tableaux !**

Instructions	Réponse	Commentaires
let v1 = [|1 ;2 ;3|] ; ;		
let v2 = v1 ; ;		
v2.(0)<-0 ; ;		
v1 ; ;		

Dans cet exemple, v1 et v2 désigne le même emplacement dans la mémoire de l'ordinateur !...
Ainsi, si l'on change la valeur d'une cellule de l'emplacement par l'intermédiaire de v2, la valeur de v1 est elle aussi modifiée (et inversement !).

Exercice : 3

Familiarisation avec les matrices

Taper les instructions suivantes et commenter la réponse de OCaml dans le tableau suivant :

Instructions	Réponse	Commentaires
`let v = [|[|3;-4|];[|2;7|]|] ;;`		
`v.(1).(0) ;;`		
`v.(1).(0) <- 10;;`		
`v;;`		

Instructions	Réponse	Commentaires
`let v = Array.make_matrix 10 10 1 ;;`		

Remarque 6. On rappelle les conventions pour la manipulation des matrices :

1. On note n le nombre de lignes et p le nombre de colonnes.
2. On note i l'indice de ligne et j l'indice de colonne.
3. Le terme T_{ij} représente le coefficient de la matrice situé à la i ème ligne et à la j ème colonne.

2 Les structures de contrôle : `IF ... THEN ... ELSE ...`

Cette structure est le la forme : if *condition* then *action 1* else *action 2* ;;

Elle se traduit de la façon suivante :
Si le booléen *condition* est vrai, alors l'*action 1* est réalisée, sinon c'est l'*action 2*.

Exercice : 4
Redéfinir en langage OCaml la fonction *valeur absolue*.

Remarque 7. Vérifier les remarques suivantes :

1. L'instruction `else` n'est pas indispensable lorsque la sortie est de type `unit`.
2. Les deux actions sont nécessairement de types identiques.

Si les actions comportent plusieurs instructions, on prendra soin de les encadrer par `begin ... end`.

OCaml
```
let signe x = if x>0 then
                        begin print_int x;
                              print_string " est positif";
                        end
                      else
                        begin print_int x;
                              print_string " est negatif ou nul"
                        end ;;
```

Remarque 8. Les mots clés `begin ... end` sont indispensables pour une bonne interprétation de vos instructions par la machine. Les instructions comprises entre `begin ... end` doivent toutes être de type `unit`.

Si le contrôle porte sur plus de 2 conditions, on pourra ajouter else à la suite des différentes sorties possibles.

OCaml
```
let signe x = if x>0 then print_string "positif" else
              if x=0 then print_string "nul"     else
                          print_string "négatif";;
```

A retenir ! !

1. Dans une structure de contrôle, le `else` est indispensable sauf lorsque la sortie est de type `unit`.
2. Dans une structure de contrôle, les deux sorties possibles sont de types identiques (sauf dans le cas où l'une aboutit à un message d'erreur : `failwith "..."`).
3. Lorsqu'une action comporte plusieurs instructions, celles-ci doivent être :
   - de type `unit`
   - être encadrées par les mots clés `begin ... end`
   - être séparées par des "`;`"
4. Lorsque le contrôle porte sur plus de 2 conditions on pourra utiliser la structure if ... then ... else if ... then ... else if .... mais on préfèrera employer si possible la structure fonctionnelle `match ... with ...` (vue dans un chapitre antérieur)
5. On demande à l'ordinateur d'afficher une chaîne de caractères avec l'instruction `print_string`

Remarque 9. **Equivalent fonctionnel d'une structure de contrôle**

L'instruction `if` *condition* `then` *action 1* `else` *action 2* peut aussi se programmer sous la forme :

OCaml
```
match condition with                        ou     match expression with
              | true  -> action1                                  | val1 -> action1
              | false -> action2 ;;                               | val2 -> action2 ;;
```

Exercice : 5

Construire une fonction permettant de comparer à l'aide de l'ordre lexicographique, deux vecteurs du plan.

Exercice : 6

Fonction de Dirac

Construire en Caml la fonction Dirac qui a tout x réel associe sa fonction de dirac δ_x qui vaut 1 en x et 0 ailleurs.

1. Avec la structure de contrôle itérative
2. Avec la structure de contrôle fonctionnelle

Remarque 10. **Message d'erreur**

Lorsque l'on veut afficher un message d'erreur (pour indiquer par exemple que certaines valeurs ne peuvent être prises comme argument), on pourra utiliser la commande `failwith`.

Par exemple :

OCaml
```
let f x = if x = 0. then failwith "la fonction n'est pas définie pour cette valeur"
                    else 1. /. x ;;
let a = f 0. ;;
a ;;
f 2. ;;
```

Attention : le déclenchement d'un tel message d'erreur entraîne l'arrêt de la procédure et a pour conséquence que l'objet qui était éventuellement en processus de déclaration n'a pas été effectivement déclaré.

Remarque 11. En pratique la structure `if ... then ... else ...` sera souvent remplacé par la structure fonctionnelle `match ... with ....`

3 Les boucles : FOR

Cette structure est le la forme : for i = *début* to *fin* do *action(s)* done ; ;

Elle se traduit de la façon suivante :
Effectuer l'*action* pour les valeurs de i allant de *début* à *fin*.

Remarque 12. Le `to` peut-être remplacé par `downto` qui a pour effet de décrémenter l'identificateur.

`Exemple 1.` Pour calculer la somme des 10 premiers entiers, on écrira :

OCaml
```
let n = ref 0 in
    for i = 1 to 10 do n:=!n+i
                    done;
!n;;
```

Vous pourrez transformer cet algorithme en une fonction ...

Remarque 13. Lorsque l'on demande d'effectuer plusieurs instructions à chaque passage de boucle, on veillera à terminer chaque instruction par un " ;".
On pourra structurer le programme selon le modèle suivant :

OCaml
```
for i =1 to 3 do print_int i;
                 print_newline();
              done;;
```

Exercice : 7

Utiliser une boucle `for` pour construire différentes fonctions permettant de :

1. afficher les éléments d'un tableau de longueur n en les séparant par des tirés.
2. épeler un mot.
3. afficher tous les entiers pairs de 1 à n en revenant à chaque fois à la ligne (commande : `print_newline()`;.
4. calculer $n!$.
5. calculer la somme des entiers compris entre n et m.
6. compter le nombre de fois où l'on rencontre la valeur x dans une *liste* donnée.
7. calculer la moyenne d'une *liste* de nombre.
8. déterminer si un nombre n est parfait ou non.
9. déterminer une procédure MAX donnant le maximum des composantes d'un tableau.
10. déterminer une procédure PREM donnant les nombres premiers premiers compris entre 1 et n.

4 Les boucles : WHILE

Cette structure est le la forme : while *condition* do *action(s)* done ; ;

Elle se traduit de la façon suivante :
Effectuer l'*action* tant que la *condition* est vérifiée.

Remarque 14. Attention au risque de boucle infinie lorsque la *condition* est toujours vérifiée.
Pour cela, vous vous assurerez en particulier que l'*action* modifie bien le contenu de la *condition*.
Vous penserez à utiliser l'option "interrupt OCaml" dans le menu OCaml si toutefois cela vous arrive.

Remarque 15. **Condition multiple**

La syntaxe utilisée pour une condition multiple est :
- `&&` pour "et"
- `||` pour "ou"

`Exemple 2.` Construction d'une fonction de a donnant le plus petit entier n tel que $a^n > 10^6$:

OCaml
```
let fonct a = let n = ref 1. in
              while a**(!n)<10.**6. do n:=!n +.1.
                                    done ;
              print_string "Le plus petit entier n tel que " ;
              print_float a ;
              print_string "^n est supérieur à " ;
              print_float (10.**6.) ;
              print_string " est : " ;
              print_float (!n) ;
              print_newline() ;;
```

Exercice : 8

(∗) NIVEAU FACILE ! Construire des procédures itératives permettant de :

1. déterminer les n premiers nombres premiers.
2. déterminer la longueur d'une liste.
3. déterminer le plus petit entier n tel que la somme des $\frac{1}{k}$ pour k variant de 1 à n dépasse une valeur A.
4. déterminer le nombre d'étapes nécessaires pour atteindre la valeur 1 en appliquant l'algorithme de Syracus à une valeur entière n.

Exercice : 9

(∗∗) NIVEAU MOYEN ! Construire des procédures itératives permettant de :

1. déterminer par dichotomie si un tableau v de composantes triées admet une de ses composantes égale à e.
2. déterminer la solution d'une équation de la forme $f(x) = 0$ à ε près par dichotomie.
3. déterminer la valeur de a^n en utilisant le codage binaire de n (algorithme d'exponentiation rapide)

5 Preuve d'un programme itératif

Pour tout programme, nous devons pouvoir nous assurer que :

1. Les boucles intervenant dans le programme se terminent bien. (la terminaison du programme)
2. Le programme donne effectivement le résultat attendu. (La correction du programme)

Si ce travail doit principalement s'effectuer lors de la conception de celui-ci, nous devons cependant pouvoir a postériori effectuer ces vérifications, même si, comme nous allons le voir, ceci ne sera malheureusement pas toujours possible ...

DÉFINITION 1 : *Prouver* un algorithme consiste à :

1. Montrer que l'algorithme se termine pour toute valeur de ses arguments. (La terminaison)
2. Déterminer le résultat de l'algorithme et vérifier qu'il correspond au résultat attendu. (La correction)

5.1 Indécidabilité de la terminaison

Au début du siècle dernier, Gödel a prouvé qu'il était impossible de concevoir un algorithme (une méthode) permettant de déterminer si un programme donné se termine pour l'ensemble des valeurs possibles de ses arguments ou pas. Pour justifier ceci, Gödel propose une démonstration par l'absurde.

L'idée de cette démonstration est la suivante :

1. On peut montrer que toute fonction calculable (issue d'un algorithme) peut-être codée par un entier que nous appelerons `code(f)`.
2. Supposons qu'il existe une fonction `termine : int -> bool` qui admet pour argument le code d'une fonction calculable f et qui renvoie la valeur `true` si la fonction f se termine et la valeur `false` sinon.
3. On pourrait alors construire la fonction récursive `absurbe()` suivante :

OCaml
```
let absurde () = while termine(code(absurde())) do done;;
```

4. Que cette fonction se termine ou pas, on aboutit à une absurdité.
   Cela prouve que la fonction `termine()` ne peut exister.

A titre d'exemple, aucune théorie n'a encore pu démontrer que l'algorithme suivant se terminait et rien ne permet d'affirmer qu'il est possible de le savoir !

`Exemple 3.` **Algorithme de Syracuse**
Calcul du terme général de $(u_n)_{n\in\mathbb{N}}$ définie par $\left\{\begin{array}{l} u_0 \in \mathbb{N}^* \\ u_0 > 1 \end{array}\right.$, et $u_k = \left\{\begin{array}{ll} u_k/2 & \text{si } u_k \text{ est pair} \\ 3.u_k + 1 & \text{sinon} \end{array}\right.$ tant que $u_n \neq 1$.

5.2 Principe général pour prouver une boucle

1. **Identification des variables :**
   On identifie les éléments variables de la boucle (références) : a, b ...
2. **Conjecture :**
   On recherche des relations donnant la valeur de chacune des variables précédentes en fonction du rang i de l'itération. On pourra noter a_i, b_i ... le contenu des variables a et b après i itérations.
3. **Validation :**
   On valide les relations conjecturées par récurrence.
4. **Sortie :**
   (a) **Vérification de la sortie de boucle :**
   On s'assure que la condition de sortie de boucle est bien vérifiée au bout d'un certain nombre d'itérations. (Immédiat dans la cas des boucles "for")
   Dans le cas où la boucle termine, on détermine au passage le nombre d'itérations $i = n$ effectuées.
   (b) **Valeurs de sortie de boucle :**
   On détermine les valeurs des grandeurs variables à la sortie de la boucle : a_n, b_n ...

`Exemple 4.` $(*)$ Considérons l'algorithme suivant ayant pour objectif le calcul de la somme des n premiers entiers :

OCaml
```
let somme n =
              let resultat = ref 0 in
              for i = 1 to n do resultat:=!resultat + i done;
              !resultat;;
```

1. Terminaison : Comme il s'agit ici d'une boucle "FOR", l'algorithme se termine après n itérations.

2. Conjecture et récurrence : Pour prouver la validité du résultat obtenu, on identifie les éléments variables de la boucle (ici la référence "resultat") et on définit la propriété $P(i)$ suivante :

$$P(i) : \quad \text{"Après } i \text{ itérations, nous avons : } \quad resultat_i = \sum_{k=0}^{i} k\text{".}$$

   On démontre par récurrence, que $P(i)$ est vraie pour tout $i \in [\![0, n]\!]$.

3. Sortie de boucle : après n itérations, la référence "resultat" contient la valeur $\displaystyle\sum_{k=0}^{n} k$.
   Or cette valeur correspond au résultat rendu par la fonction et correspond bien au résultat attendu.

Exercice : 10

(∗) Considérons l'algorithme suivant :

OCaml
```
let f n =
       let x = ref n and y = ref n in
       while not(!y=0) do x:=!x+2 ;
                          y:=!y-1 ;
                               done ;
       !x ;;
```

1. Identifier les éléments variables de la boucle et conjecturer une propriété $P(i)$: "Après i itérations ..."
2. En déduire que la boucle `while` se termine et donner le résultat obtenu pour tout argument $n \in \mathbb{N}$.

Exercice : 11

(∗) Soit $P = [|a_0;\ a_1;\ \ldots; a_n|]$ et $x \in \mathbb{R}$.
Prouver que le programme suivant renvoie bien la valeur $a^n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$.

OCaml
```
let f p x = let s = ref 0 and
               n = (Array.length p -1) in
           for k = 0 to n do s := x * !s + p.(n-k)
                          done ;
           !s ;;
```

Exercice : 12

(∗) Soit $n \in \mathbb{N}$.
Prouver que le programme suivant renvoie bien la décomposition binaire de n sous la forme d'une liste de coefficients.

OCaml
```
let binaire n = let r = ref n and
                    l = ref [] in
    while !r <> 0 & !r <> 1 do l := (!r mod 2)::!l ;
                               r := !r / 2
                            done ;
    !r::!l ;;
```

Remarque 16. Pour certaines boucles, la démarche précédente peut poser des problèmes dus :
— au nombre très important de variables intervenant dans la boucle
— au fait que l'on ne puisse pas facilement exprimer le contenu des variables après la $i$ème itération
Dans ce cas, on peut utiliser une preuve utilisant la notion d'*invariant de boucle*.

5.3 Preuve à l'aide d'un invariant de boucle

DÉFINITION 2 : **Invariant de boucle**
On considère une boucle (`for` ou `while`) dans un algorithme donné.

1. On appelle *invariant de boucle* de cette boucle, toute relation ou tout système de relations R portant sur les différents éléments variables de la boucle et qui reste vraie à chaque itération i.
2. Lorsqu'un invariant de boucle permet de déterminer le résultat obtenu en sortie de boucle, celui-ci est appelé *L'invariant de boucle* de la boucle étudiée.

Les invariants de boucle d'un programme sont en général fournis par le programmeur lors de la livraison d'un programme. Cela permet à d'éventuels utilisateurs de vérifier par eux-même que le programme donné fonctionne effectivement.

PROUVER une boucle à l'aide d'un invariant de boucle donné

1. **Validation :**
   On vérifie par récurrence que l'invariant de boucle donné est bien un invariant de boucle
2. **Sortie :**
   (a) On vérifie que la boucle se termine bien.
   (b) On utilise l'invariant de boucle et la condition de sortie de boucle (dans le cas d'une boucle "while") pour déterminer les valeurs des grandeurs significatives à la sortie de la boucle.

Exercice : 13

(∗) Déterminer l'invariant de boucle du programme donné dans l'exercice précédent.
Prouver votre programme en utilisant cet invariant de boucle.

Remarque 17. On remarque que dans le cas de l'utilisation d'un invariant de boucle, la preuve de terminaison doit se faire indépendamment de l'utilisation de l'invariant de boucle.

Exercice : 14

(∗) Prouver l'algorithme d'Euclide à l'aide d'un invariant de boucle.

Exercice : 15

(∗) Prouver la fonction suivante et déterminer un invariant de boucle pour chaque boucle.
On supposera $y > 0$.

OCaml
```
let f x y =
      let r = ref 0 and s = ref y in
          while (!s > 0) do r := !r + x ;
                            s := !s -1 ;
                         done ;
      s := y - 1 ;
      let t = ref !r in
          while (!s > 0) do r := !r + (!t) ;
                            s := !s - 1 ;
                         done ;
      !r ;;
```

Exercice : 16

Egalité de Bezout

(∗∗) On rappelle l'algorithme d'Euclide qui permet de déterminer le PGCD de deux nombres entiers a et b.
On considère la suite $r_0 = a$, $r_1 = b$, $\ldots r_{n-1} \neq 0$, $r_n = 0$ où r_k est le reste de la division euclidienne de r_{k-2} par r_{k-1}.
On définit parallèlement la suite (q_n) telle que $r_{k-1} = q_k.r_k + r_{k+1}$ avec $r_{k+1} < r_k$.
Le PGCD de a et b est alors le dernier reste non nul (c'est à dire ici : r_{n-1}).

Donner une implémentation CAML de l'algorithme de Bezout permettant de déterminer les entiers u et v et $a \wedge b$ de l'égalité de Bezout $au + bv = a \wedge b$.

Conseils :

1. *Commencer par définir les suites (u_k) et (v_k) par récurrence.*
2. *Définissez clairement :*
   (a) les données de départ
   (b) la condition d'arrêt
   (c) l'ordre dans le calcul des différents termes des suites
3. *Prouvez cet algorithme à l'aide d'un invariant de boucle.*

Exercice : 17

(∗) Soient $a,\ b \in \mathbb{N}^*$.

Prouver en utilisant l'invariant de boucle `"a = bQ + R"` que le programme suivant renvoie bien la valeur le quotient et le reste de la division euclidienne de a par b.

OCaml
```
let div_eucl a b = let q = ref 0 and
                       r = ref a in
    while !r >= b do q := !q + 1 ;
                     r := !r - b
                done ;
    (!q , !r) ;;
```

Exercice : 18

(∗) Soit la suite (u_n) définie par $\begin{cases} u_0 = 1 \\ u_{n+1} = \frac{1}{2}(u_n + \frac{x}{u_n}) \end{cases}$.

Montrer, à l'aide de l'invariant de boucle "$0 \leq u_n - \sqrt{x} \leq \frac{u_1}{2^{n-1}}$" que le programme calculant la suite (u_n) permet d'obtenir une approximation de $\sqrt{x}$ à ε près.

Aide : On pourra remarquer que pour tout $n \geq 1$, on a $\sqrt{x} \geq \frac{x}{u_n}$ et $u_{n+1} - \sqrt{x} = \frac{1}{2}((u_n - \sqrt{x}) - (\sqrt{x} - \frac{x}{u_n}))$.

Remarque 18. La méthode de validation décrite précédemment devient malheureusement rapidement compliquée dès lors que l'algorithme itératif fait intervenir des branchements conditionnels en cascade ou des boucles imbriquées.

6 Exercices en classe

6.1 Exercice 1

On considère la suite (u_n) définie par la récurrence suivante : $u_0 = 1$ et $u_{n+1} = \sum_{k=0}^{n} u_k.u_{n-k}$ pour $n \geq 0$.

On souhaite ici construire une fonction donnant la valeur u_n de la suite.

1. Spécification de la fonction : Effectuer une analyse du problème posé
   (a) Nom de la fonction :
   (b) Arguments de la fonction (noms et types) :
   (c) Sortie de la fonction (nom et type) :
   (d) Construction de l'algorithme :
      i. Valeurs nécessaires pour obtenir le terme u_n ?
      ii. Comment obtenir ces valeurs ?
      iii. Comment stocker ces valeurs ?
      iv. Quel algorithme construire ?
2. Implémentation OCaml : Donner une implémentation OCaml de l'algorithme précédent.
3. Preuve : Prouver le programme obtenu.

6.2 Exercice 2

Il s'agit ici de déterminer un algorithme permettant de remettre les valeurs 1, 2 et 3 contenues dans un tableau donné, dans l'ordre (les 1 puis les 2 puis les 3).

1. Spécification de la fonction : Effectuer une analyse du problème posé

   (a) Nom de la fonction :
   (b) Arguments de la fonction (noms et types) :
   (c) Sortie de la fonction (nom et type) :
   (d) Construction de l'algorithme :
      i. Idée 1 : on peut
         A. Parcourir toutes les composantes du tableau
         B. Compter le nombre de 1, de 2 et de 3
         C. Modifier les composantes du tableau initial en plaçant les "1" puis les "2" et enfin les "3".
      ii. Idée 2 : on peut
         A. Construire une procédure permettant d'échanger les valeurs de deux composantes d'un tableau
         B. Parcourir une première fois les composantes du tableau et commencer par placer les "1" en tête de liste
         C. Parcourir de nouveau les composantes du tableau et placer les 3 en queue de liste
      iii. Idée 3 : on peut
         A. Construire une procédure permettant d'échanger les valeurs de deux composantes d'un tableau
         B. Parcourir une seule fois les composantes du tableau en plaçant les 1 en tête et les 3 en queue

2. Implémentation OCaml : Donner une implémentation CAML des algorithmes précédents.

3. Analyse de Complexité :Comparer les complexités spatiale et temporalle des 3 algorithmes précédents.

6.3 Exercice 3 : Pivot de Gauss et inversion d'une matrice

Considérons un système de cramer à n équations et n inconnues : $AX = B$.

Les données définissant le système seront stockées dans une matrice S comportant n lignes et $(n+1)$ colonnes que l'on obtient en collant la matrice B à la matrice A : $S = A|B$.

1. Triangularisation de la matrice :

   (a) Les sous-procédures :
      i. Construisez une fonction `pivot : float array array -> int -> int` qui renvoie l'indice de la première ligne L_i de S (compris entre j et n) tel que $S[i][j] \neq 0$. Cet élément sera appelé le "pivot".

      ii. Construisez une procédure `echange : float array array -> int -> int -> unit` qui échange les éléments des colonnes C_k telles que $k \geq i$ des lignes L_i et L_j.

      iii. Construisez une procédure `annule : float array array -> int -> unit` qui annule par OEL les coefficients des lignes L_k telles que $k > i$ de la colonne C_i.

   (b) En déduire `triangularisation : float array array -> unit` qui triangularise la matrice S.

2. En déduire :
   (a) une fonction de résolution du système $AX = B$.
   (b) une fonction de calcul de la matrice A^{-1}.
   (c) une fonction de calcul du rang de la matrice A^{-1}.

7 Exercices d'entrainement

Ne disposant pas du temps nécessaire pour traiter en classe ces différents exercices, je vous invite à y réfléchir et à me proposer sur papier vos solutions.

Exercice : 19

Equation entière

Ecrire une fonction qui, à la donnée de $n \in \mathbb{N}$ retourne la liste des solutions entières de l'équation $x^2 + y^2 = n$.

Exercice : 20

Produit scalaire

Ecrire une fonction qui, à la donnée de deux vecteurs $\overrightarrow{u}$ et $\overrightarrow{v}$:

1. Vérifie qu'ils ont bien le même nombre de coordonnées.
2. Calcule, si c'est le cas, leur produit scalaire usuel.

Exercice : 21

Palindrome

Un palindrome est une phrase qui peut se lire aussi bien à l'endroit qu'à l'envers.
Exemple : "tu l'as trop écrasé césar ce Port salut" ou "la mariée ira mal".
Dans cet exercice, nous nous intéressons aux entiers dont l'écriture décimale est un palindrome.

1. Définir la fonction `renverse` qui renverse l'écriture décimal d'un entier. (`renverse 123 = 321`).
   Pour cela, vous pourrez commencer par convertir le nombre en la liste de ses chiffres considérés comme caractères.
2. En déduire le test `palindrome` sur les entiers.

Exercice : 22

Entiers amiables

Soit a un entier naturel. On appelle **diviseur propre de** a tout diviseur de a différent de a.
Deux entiers sont dits **amiables** si et seulement si chacun d'eux est égal à la somme des diviseurs propres de l'autre.

1. Ecrire une fonction permettant de déterminer la somme des diviseurs propres d'un entier
2. Ecrire une fonction permettant de déterminer tous les couples d'entiers amiables inférieurs ou égaux à 1500.

Conseil : Ne lancez pas tout de suite le calcul pour les entiers amiables jusqu'à 1500... Commencez par de plus petites valeurs, que vous pourrez augmenter progressivement en examinant attentivement l'évolution du temps de calcul. Une petite réflexion permet de ne pas écrire un algorithme inutilement long.

Exercice : 23

Ordre lexicographique

On définit sur $\mathbb{Z}^2$ un ordre total appelé l'*ordre lexicographique*.

1. Programmer une fonction `inf_lexic2` qui compare deux couples de $\mathbb{Z}^2$.
2. Proposer une généralisation pour des n_uplets de longueur n quelconque.

Exercice : 24

Coefficients binomiaux

1. Programmer une fonction produit qui calcul le produit des entiers de n à m.
2. Utiliser la fonction précédente pour construire une fonction `binom` calculant le coefficient binômial $\binom{n}{p}$.

Exercice : 25

Opérations sur les polynômes.

Ecrire les différentes fonctions permettant :

1. d'afficher un monôme de coef et de degré donné.
2. d'afficher un polynôme dont les coefficients sont données sous la forme d'un tableau.
3. d'additionner deux polynômes p et q et d'afficher le résultat sous la forme d'un polynôme.
4. de multiplier deux polynômes p et q et d'afficher le résultat sous la forme d'un polynôme.

Exercice : 26

Marche aléatoire

La fonction `random_int n` de Caml permet de tirer au hasard un entier compris entre 0 et $n-1$.

1. Expliquer comment utiliser :
   (a) la fonction `random_int 2` pour simuler un déplacement aléatoire le long de l'axe O_x
   (b) la fonction `random_int 4` pour simuler un déplacement aléatoire dans le plan.

2. Déplacement sur une droite.
   On suppose que l'on part de l'origine O.
   Construire un programme `marche1` en Caml d'argument n donnant :
   (a) La position atteinte au bout de n pas.
   (b) Le nombre de passage par l'origine.
   (c) L'abscisse du point le plus à droite atteint au cours de la marche.

3. Ecrire en Caml une fonction `marche2` qui simule une marche aléatoire d'au plus n pas dans le plan et qui donne le nombre de pas effectués avant le premier retour à l'origine.