Une variété d'expressions des algorithmes pour mieux apprendre à raisonner

Jean-Pierre Peyrin

UNE VARIÉTÉ D'EXPRESSIONS DES ALGORITHMES POUR MIEUX APPRENDRE À RAISONNER

Jean-Pierre PEYRIN

Tout langage de programmation privilégie un mode d'expression (actionnel, fonctionnel, relationnel,.). Toute pratique intensive d'un mode d'expression conduit à ne comprendre une analyse qu'en termes de ce mode d'expression, et l'on confond alors la solution d'un problème avec l'expression de cette solution. Il faut se donner les moyens de comprendre différentes expressions d'une même solution pour capter l'essentiel de cette solution et prendre ainsi le recul nécessaire à une bonne programmation. Rester trop lier à une forme (un mode d'expression) empêche de bien comprendre le fond (une solution algorithmique).

L'atelier a proposé :

- Une observation de la variété des expressions algorithmiques et de la variété des styles de programmation (mise en évidence des modèles de calcul ; définition d'une programmation impérative dans laquelle le modèle de calcul est explicite).

- Une découverte de la cohérence des schémas algorithmiques fondamentaux. La nécessité d'une définition de l'ensemble des informations à traiter conduit à l'analyse par cas (définition par extension) et à l'analyse récurrente (définition par compréhension). Les modalités d'application du raisonnement par récurrence conduisent à structurer "mentalement" les informations en séquence ou en arbre. La structure des algorithmes de traitement d'un ensemble d'informations repose donc sur une structure abstraite possible de cet ensemble.

- Une étude, à titre d'exemple, de la structure (type abstrait) de séquence. Définition fonctionnelle du type séquence en termes des constructeurs des objets du type ; expression fonctionnelle des schémas de parcours et de recherche (fonctions récursives). Définition actionnelle des séquences ; expression actionnelle des schémas de parcours et de recherche (actions itératives). Correspondance entre les deux expressions ; expression (fonctionnelle) de l'invariant de l'itération et explicitation (actionnelle) du modèle de calcul ; confrontation des points de vue "sémantique" et "opératoire" [Pey88].

Le texte qui suit ne peut qu'être réducteur. Plutôt que de reprendre toute la progression d'exemples présentée lors de l'atelier, il n'en présente qu'une partie significative et commentée.

1. LA RÉCURRENCE : il y a deux manières d'utiliser la récurrence [Sch84].

1.1. La première manière : (ex. : tout entier de la forme 10^i-1 est divisible par 9).

base *(i = 0)* : 10^0-1 = 1-1 = 0 = 0 **x 9**
récurrence :
hypothèse *(i = n)* : 10^n-1 = k **x 9**
hérédité *(de n à n+1)* : 10^{n+1}-1 = (10 x 10^n)-1 = ...
= (10 x k + 1) **x 9**

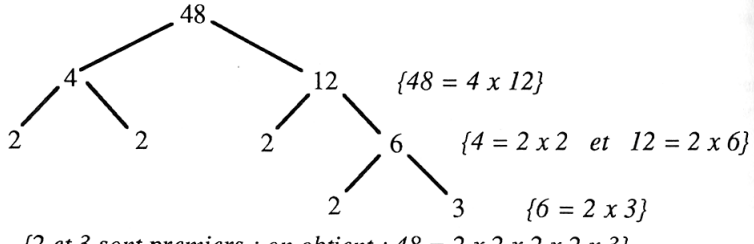
Une « trace » de la vérification de la propriété pour les entiers montre un calcul « séquentiel ». Cette manière d'utiliser la récurrence conduit au type séquence. Par exemple :

i	0	1	2	3	4	...
10^i-1	0	9	99	999	9999	...

N.B. : le type séquence est développé, ci-après, au § 2.

1.2. La deuxième manière : (ex. : tout entier supérieur à 1 peut être décomposé sous forme d'un produit de facteurs premiers).

base : tous les nombres premiers
récurrence :
hypothèse : propriété vraie pour tout entier k tel que $2 \leq k < n$
hérédité : - si n n'est pas premier, il existe deux entiers a et b vérifiant : $n = a \times b$ et $1 < a < n$ et $1 < b < n$
- a et b vérifient la propriété, par hypothèse de récurrence ; on en déduit la propriété pour n.



{2 et 3 sont premiers ; on obtient : 48 = 2 x 2 x 2 x 2 x 3}

Une trace du calcul effectué montre un calcul « arborescent ». Cette manière d'utiliser la récurrence conduit au type arbre. Par exemple, pour 48, on obtient la "trace" suivante :
N.B. : le type arbre n'est pas développé dans ce texte.

2. LE TYPE SÉQUENCE (expression fonctionnelle) :

2.1. Définition d'une séquence d'éléments :

constructeurs *(ils permettent de construire toutes les séquences)* :

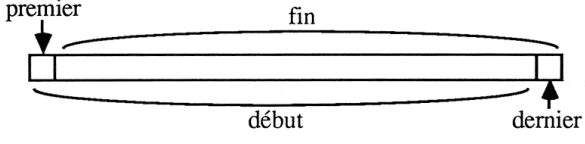
[ ] : une fonction → une séquence d'éléments *{séquence vide}*
- o - : une fonction (un élément, une séquence d'éléments) → une séquence d'éléments *{ajout en tête}*
- . - : une fonction (une séquence d'éléments, un élément) → une séquence d'éléments *{ajout en queue}*

exemples : [ ] **.** a = [a] ; a **o** [b c d e] = [a b c d e]

sélecteurs *(associés aux constructeurs)* :

vide : une fonction (une séquence d'éléments) → un booléen
p**remier** : une fonction (une séquence non vide d'éléments) → un élément
fin : une fonction (une séquence non vide d'éléments) → une séquence d'éléments
dernier : une fonction (une séquence non vide d'éléments) → un élément
début : une fonction (une séquence non vide d'éléments) → une séquence d'éléments

exemples : vide([ ])=vrai ; vide([a b])=faux ; premier([a b c])=a ; fin([a b c])=[b c] ; début([a b c d])=[a b c] ; dernier([a b c d])=d



2.2. Schémas de parcours d'une séquence *(chaque élément est traité une fois)* :

forme 1 : PF : la fonction (S : une séquence d'éléments) → une valeur

{Parcours Fonctionnel}
selon S
vide(S) : **v**
non vide(S) : **f**(PF(début(S)), dernier(S))

forme 2 : PF(S) = PFI(**v**, S)
{la fonction PF nécessite une fonction intermédiaire PFI}

PFI : la fonction (Acc : une valeur ; S : une séquence d'éléments) → une valeur
{Parcours Fonctionnel Itératif avec utilisation d'un Accumulateur}
selon S
vide(S) : Acc
non vide(S) : PFI(**f**(Acc, premier(S)), fin(S))

N.B. : la valeur **v** (valeur du parcours d'une séquence vide) et la fonction **f** (expression récurrente du parcours) sont les mêmes dans les deux schémas.

2.3. Exemple : nombre d'occurrences de la lettre 'A' dans un texte.

forme 1 : NbaF : la fonction (T : un texte) → un entier ≥ 0
selon T
vide(T) : **0**
non vide(T) : NbaF(début(T)) **+ (si** dernier(T)**='A' alors 1 sinon 0)**

par exemple : NbaF("MA") = NbaF("M") + 1 = (NbaF("") + 0) + 1 = 0 + 0 + 1 = 1

forme 2 : NbaF(T) = NbaFI(**0**, T)

NbaFI : la fonction (Acc : un entier ≥ 0 ; T : un texte) → un entier ≥ 0
selon T
vide(T) : Acc
non vide(T) : NbaFI(Acc **+ (si** premier(T)**='A' alors 1 sinon 0)**, fin(T))

par exemple : NbaF("MA") = NbaFI(0, "MA") = NbaFI(0, "A") = NbaFI(1, "") = 1

traductions en Logo, par exemple, de ces algorithmes :

forme 1 :

```
POUR NBAF  :T
      SI VIDEP :T          [RETOURNE 0]
            [SI DERNIER :T = 'A'          [RETOURNE (NBAF SD :T)+1]
                                    [RETOURNE (NBAF SD :T)] ]
FIN
```

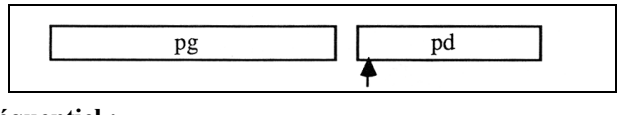
forme 2 :

```
POUR NBAF  :T
       RETOURNE NBAFI  0  :T
FIN

POUR NBAFI  :ACC  :T
       SI VIDEP  :T       [RETOURNE  :ACC]
            [SI PREMIER :T = 'A' [RETOURNE NBAFI ( :ACC+1) SP :T]
            [RETOURNE NBAFI :ACC  SP :T]   ]
FIN
```

3. LE TYPE SÉQUENCE (expression actionnelle) [ScP 88].

On considère que la séquence est toujours "coupée" en partie gauche (pg) et droite (pd). L'élément courant est le premier de la partie droite. *On peut construire un deuxième modèle dans lequel l'élément courant est le dernier de la partie gauche ; ce modèle, utile également, ne sera pas étudié ici.*



3.1. Accès séquentiel :

Démarrer : une action *{donne accès au premier élément}*
{état initial : indifférent}
{état final : pg = [ ], pd = S}

Avancer : une action *{donne accès à l'élément suivant}*
{état initial : pg = g, pd = d, S = g&d}
{état final : pg = g.premier(d), pd = fin(d)}
ElémentCourant : une fonction → un élément *{ElémentCourant = premier(pd)}*
FinDeSéquence : une fonction → un booléen *{FinDeSéquence = (vide(pd))}*

3.2. Schéma de parcours d'une séquence (v et f sont les mêmes qu'au § 2.2.) **:**

```
PA : l'action (le résultat R : une valeur) {Parcours Actionnel}
        Démarrer
        Acc ← v   {valeur initiale de l'Accumulateur}
        tantque non FinDeSéquence : {invariant : Acc = PF(pg)}
                Acc ← f(Acc, ElémentCourant)   {modification de l'Accumulateur}
                Avancer
        R ← Acc
```

3.3. Exemple : nombre d'occurrences de la lettre 'A' dans un texte.

```
NbaA : l'action (le résultat R : un entier ≥ 0) {Nombre_de_A Actionnel}
        Démarrer
        Acc ← 0
        tantque non  FinDeSéquence : {Acc = NbaF(pg)}
                Acc ← Acc + (si ElémentCourant ='A' alors 1 sinon 0)
                Avancer
        R ← Acc
```

3.4. Une illustration de ces primitives (fichier et tableau en Pascal)

fichier :	f : file of .
Démarrer	reset(f)
Avancer	get(f)
ElémentCourant	f^
FinDeSéquence	eof(f)
tableau :	T : array [1 .. N] of . ; i : [1 .. N+1]
Démarrer	i := 1
Avancer	i := i+1
ElémentCourant	T[i]
FinDeSéquence	i = N+1

traductions en Pascal, par exemple, de l'algorithme du § 3.3. :

```
reset(Texte) ;                          i :=  1 ;
Acc := 0 ;                              Acc := 0 ;
while not eof(Texte) do                 while i ≠ N+1 do
begin                                   begin
    if Texte^ = "A" then Acc := Acc + 1 ;     if Texte[i] = "A" then Acc := Acc + 1 ;
    get(Texte)                              i :=  i+1
end ;                                   end ;
R := Acc                                R := Acc
```

4. UN EXEMPLE DE CONSTRUCTION D'ALGORITHME :

Le but de cet exemple est de montrer comment on peut construire systématiquement un algorithme ; l'approche fonctionnelle permet une analyse indépendante des représentations particulières et des techniques d'accès à l'information. L'expression fonctionnelle de la solution présente la sémantique de l'algorithme ; l'expression actionnelle en présente le fonctionnement. Les deux expressions sont ainsi complémentaires et ces deux "points de vue" sont indispensables au programmeur.

Enoncé : Etant donné un ensemble E de nombres entiers, muni d'une relation d'ordre total, on désire construire un ensemble E', à partir de E, en remplaçant chaque élément par la somme de tous les éléments qui le précèdent (selon la relation d'ordre) et de lui-même. On dira que E' = SommesCumulées(E).

Exemple : E = [2, 9, -1, 5, 4] donne E' = [2, 11, 10, 15, 19]

Approche fonctionnelle :

Les ensembles E et E' sont naturellement « structurés » en séquence, en prenant la relation d'ordre pour relation « suivant ». On veut utiliser un schéma de parcours de séquence et on doit, pour cela, identifier les composantes **v** et **f**.

On étudie d'abord **f**, c'est à dire l'expression récurrente du problème. On considère un cas général : E = D**.**d où D = début(E) et d = dernier(E). On prend une hypothèse (de récurrence) : on suppose savoir construire D' = SommesCumulées(D). Alors on peut affirmer que E' = D'**.** (dernier(D') + d).

Par exemple : E = [2, 9, -1, 5, 4] se décompose en D = [2, 9, -1, 5] et d = 4. L'hypothèse de récurrence nous donne D' = [2, 11, 10, 15]. Alors, dernier(D')=15 et 15+d = 19.

Cette expression récurrente suppose que D' n'est pas vide puisque l'on a besoin de son dernier élément, donc que D n'est pas vide, donc que E a au moins deux éléments. Lorsque E a un élément, la relation de récurrence ne s'applique pas ; il s'agit de la base de récurrence. Le cas où E est vide ne rentre alors pas dans la définition récurrente du problème et est donc examiné à part.

La valeur **v** correspondant à la base de récurrence est donc la valeur de SommesCumulées pour un ensemble d'un seul élément : cette valeur est l'ensemble lui-même.

Algorithme fonctionnel :

```
SommesCumulées : la fonction (E : une séquence d'entiers)
                                                → une séquence d'entiers
    selon E
        vide(E) :        [ ]
        non vide(E) :    SC(E)              {fonction intermédiaire
récurrente}
```

```
SC : la fonction (E : une séquence non vide d'entiers) → séquence non vide
d'entiers
        selon début(E)
                vide(début(E)) :        E
                non vide(début(E)) :    soit D' = SC(début(E))
                                        dans D'. (dernier(D') + dernier(E))
```

Algorithme traduit en Logo :

```
POUR SC :E
    SI VIDEP SD :E   [RETOURNE E]
                     [LOCAL "DP
                     DONNE "DP (SC SD :E)
                     RETOURNE LISTE :DP (DERNIER :DP +
DERNIER :E)]
FIN

POUR SOMMESCUMULEES :E
    SI VIDEP :E [RETOURNE [ ] ] [RETOURNE SC :E]
FIN
```

Approche actionnelle :

Les composantes **v** et **f** sont communes aux schémas fonctionnel et actionnel. Nous n'avons donc pas à recommencer l'analyse. Nous devons simplement appliquer le schéma actionnel de parcours d'une séquence.

Auparavant, nous devons décider de la représentation de l'information. Nous allons, par exemple, supposer que l'ensemble E est représenté par un tableau T défini sur [1.N]. Et nous allons supposer que E' est une modification de E. Le problème consiste donc à modifier le tableau T de telle façon que son état final représente SommesCumulées de la séquence représentée par son état initial.

Algorithme actionnel :

```
SommesCumulées :  l'action (la donnée-résultat T : un tableau d'entiers sur [1...N])
   si N ≥ 1 :          {non vide(E)}
            i ← 2      {i désigne l'élément courant ; T[1...i-1] est pg ; T[i...N] est
pd}
            tantque i ≠ N+1 :
                    T[i] ← T[i-1] + T[i]
                    i ← i+1
```

Algorithme SommesCumulées traduit en Pascal :

```
        if N >= 1 then
        begin
                i := 2 ;
                while i <> N+1 do
                begin
                        T[i] := T[i-1] + T[i] ;
                        i := i+1
                end
        end
```

5. CONCLUSION

Il faudrait beaucoup d'autres exemples pour concrétiser le propos. De toutes façons, ce texte n'est pas à lire, mais à décortiquer ; il est surtout un outil de travail destiné essentiellement aux formateurs. C'était d'ailleurs le sens de l'atelier : réfléchir sur la construction des algorithmes en confrontant différents points de vue.

Il n'y a pas « une » méthode de construction des algorithmes, mais une démarche méthodique. Comprendre les différents sens de ce que l'on écrit est le premier pas d'une telle démarche.

Jean-Pierre PEYRIN

Université Joseph Fourier,
Laboratoire de Génie Informatique,

IMAG - Campus, BP 53 X
38041 Grenoble cedex - France

REFERENCES

[Sch84] P.C. SCHOLL : *Algorithmique et représentation des données* (tome 3) ; récursivité et arbres. Masson, 1984.

[Pey88] J.P. PEYRIN : *Algorithmique - expression fonctionnelle du traitement des séquences*. Support d'un stage pour les enseignants de l'Enseignement Optionnel d'Informatique. CIAP, Grenoble, 1988.

[ScP88] P.C. SCHOLL, J.P. PEYRIN : *Schémas algorithmiques fondamentaux - séquences et itération*. Masson 1988.

[SFLM93] P.C. SCHOLL, M.C. FAUVET, F. LAGNIER, F. MARANINCHI : *Cours d'informatique - langages et programmation*. A paraître, 1993.