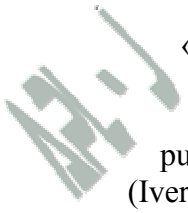
Le langage J est-il vraiment APL ?

par Sylvain Baron



« **J**, mais c'est APL » me disait Ken Iverson en 1990 autour d'une table où nous discutions (lui, Arthur Whitney et moi) pendant un après-midi du congrès APL de Copenhague. Cette année là en effet, sous le titre « APL\? », K. Iverson, E. McDonnell, A. Whitney et R. Hui présentaient publiquement leur interpréteur de **J** qu'ils venaient de produire dans le cadre de la société ISI (Iverson Software Incorporated) créée par Eric Iverson. Il est vrai que depuis cette date les textes et les communications sur le langage J se multiplient dans les instances APL et il semble difficile que notre revue « Les Nouvelles d'APL » y échappe complètement. Le langage J est-il vraiment APL? Cette question se pose naturellement puisque même Ken Iverson, par respect et humilité, réserve en public, et dans ses écrits, le nom d'APL à l'APL-ISO et nomme dialectes d'APL aussi bien APL2 que SHARP APL, APL PLUS III, APL68000, Dyalog APL et J. Et cependant ...

Cependant, il est impossible pour un observateur sincère de morceler APL et de le dissocier de la personne de Ken Iverson pour qui il représente l'effort incessant de toute une vie marquée par une conviction, une exigence, un talent extra-ordinaire (cette touche de génie qui donne presque toujours la vision de l'essentiel là où le talent ordinaire ne voit rien et passe à côté), une rigueur, une ténacité et un travail considérable. On verra aussi l'exceptionnelle unité de son oeuvre. Pour bien comprendre cela, il faut sans doute revenir quelques années en arrière.

Où en était-on vers 1973-1974, après le succès d'APL\360 puis des variables partagées (APLSV) ? On cherchait à étendre le langage au traitement d'objets qui soient autre chose que des tableaux rectangulaires de scalaires. Or si chacun s'accordait pour accepter la création de primitives nouvelles permettant la représentation scalaire d'un objet (enclore et son inverse, déclore), personne n'était satisfait de l'usage qui en résultait dans le contexte de l'APL\360. Les efforts des uns et des autres restaient peu convaincants, et nous nous souvenons tous d'un article de Phil Abrams (qui avait contribué à la naissance d'APL\360 dans l'équipe Falkoff-Iverson d'IBM), et qui fit l'ouverture du congrès APL de 1975 à Pise. Le titre en était : *What's wrong with APL ?* (*Que reprocher à APL ?*). Après avoir évoqué toutes les propositions d'extension de 1966 à 1973, on pouvait lire vers le début d'article :

« Mon sentiment actuel est que ces propositions sont trop compliquées. La plupart nécessitent beaucoup de mécanismes extérieurs à APL pour faire fonctionner correctement les extensions aux structures d'arbres. Beaucoup de ces propositions semblent s'effondrer pour le traitement des scalaires, c'est-à-dire pour les objets atomiques auxquels aucune structure n'est associée. La complexité et les carences envers les cas-limites renforcent ma conviction que personne n'a encore eu le même trait de génie pour les structures généralisées qu'Iverson a eu pour les tableaux rectangulaires ». Et dans sa conclusion : « Aujourd'hui, beaucoup des généralisations proposées ont soulevé au moins autant de problèmes qu'elles n'ont apporté de remèdes. [...] La prochaine étape est, peut-être, de recommencer à nouveau à partir des principes essentiels pour créer un langage qui est à APL ce qu'APL est à FORTRAN. Existe-t-il quelqu'un qui puisse accomplir le même saut de génie qui a donné naissance à APL ? Serons nous prêts pour cette nouvelle étape quand elle apparaîtra ? Ou, est-ce que le conservatisme APL nous a rendu aveugles aux nouvelles idées ? ». (P.Abrams, STSC : *What's wrong with APL ?* ACM APL75 Conference Proceedings, page 1 à 8).

Ce texte avait laissé amers la plupart d'entre nous, chacun étant plus ou moins persuadé que jamais, en une vie, on ne pouvait voir deux fois le miracle APL, de la part d'Iverson ou de quiconque d'ailleurs. Il allait donc falloir, après un beau feu d'artifice, s'habituer à vivre dans une relative médiocrité.

On sait que les choses ont évolué. Après quelques prémisses annonciatrices apparaissaient entre autres : *Operators and functions*, K.Iverson, Research Report 7091, IBM Corp. March 1978, *Operators and enclosed arrays*, B. Bernecky and K.Iverson, I.P. SHARP, 1980, l'extension de SHARP APL aux opérateurs de composition, de dualité et de rang, *Rationalized APL*, K.Iverson, I.P. Research Report, March 1983, *A Dictionary of the APL language*, K.Iverson, ACM 1987 Conference Proceedings et, l'actuel aboutissement: **J**. C'est par la présentation succincte de quelques unes des extensions contenues dans J que nous essayerons de répondre à la question qui fait l'objet de cet article. Nous n'aborderons ici ni la question des notations, ni celle du vocabulaire, ni celle du système.

Nous partirons de trois grandes exigences qui existaient dès 1973 pour l'évolution d'APL: une évolution des structures de données (tableaux généralisés), la rationalisation du langage par la suppression d'anomalies (par exemple, les doubles crochets pour l'indexation et pour l'opérateur d'axes) et l'écriture directe de fonctions (dont les débuts sont apparus en 1976 avec ce qu'on appelait la notation *alpha* et *oméga*) prélude à la programmation fonctionnelle.

1 - Quelques primitives

La primitive «enclos» est représentée par le signe < monadique. Elle n'est pas permissive sur les scalaires. Ainsi, `2` est distinct de `<2` . La primitive inverse «déclos» est représentée par le signe > monadique. Elle est permissive : `>2` est identique à `2` .

Un tableau enclos se présente en J, à l'écran, entouré d'un rectangle.

```
      M                    V
1 2 3 4              9 0 9 7 8
5 6 7 8

      <M
┌─────────┐
│1 2 3 4  │
│5 6 7 8  │
└─────────┘

      <V
┌─────────┐
│9 0 9 7 8│
└─────────┘
```

La primitive « lien » notée par **`;`** (introduite dès 1983 dans SHARP APL) se définit à partir de la concaténation et de l'enclos de la façon suivante: `A;B` est `(<A),B` si `B` est un enclos ou un vecteur d'enclos et `(<A),<B` si `B` est simple. Ainsi la « strand notation » de type `A B C D` d'APL2, impossible si l'enclos n'est pas permissif, est le plus souvent équivalente à `A;B;C;D` .

Par exemple, `M;V` affichera côte à côte les deux valeurs encloses de `M` et `V`.

```
      M;V
```

1 2 3 4 5 6 7 8	9 0 9 7 8

La primitive « nombre » est notée # et, #M est identique à (⍳0)⍴1↑⍴1⌿M, c'est-à-dire le scalaire donnant la première dimension de M en général et 1 si M est un scalaire. En usage dyadique, c'est la réplication (dont la compression est un cas particulier).

```
     1 0 2 0 3 # 'abcde'
acceee
```

Les primitives dyadiques « droit » et « gauche » sont représentées par les crochets droit et gauche. Le crochet droit rend l'argument de droite et le crochet gauche rend l'argument de gauche. Utilisée de façon monadique, l'une ou l'autre rend son argument ( ]a ou [a rend a).

```
     1 2 3 ] 'abcd'
abcd
     1 2 3 [ 'abcd'
1 2 3
```

Ces fonctions, introduites depuis longtemps dans le domaine de la logique formelle, se révèlent être de prodigieux auxiliaires dans la programmation fonctionnelle.

2 - L'écriture des fonctions

La flèche d'affectation se note =. en J. Elle s'applique aux données comme aux fonctions ou aux opérateurs:

```
     a =. 2 3 4
     b =. 5 6 1

     plus =. +

     a plus b
7 9 5
```

Le caractère : est l'opérateur de fabrication de fonction. A gauche, on met la définition monadique et à droite, la définition dyadique.

```
     fonc =. ! : +          ( ! est la primitive factorielle )

     fonc 4
24
     3 fonc 4
7
```

Notons qu'on peut faire une définition à la volée, ou n'en pas faire du tout.

```
     (fonc=.  ! : +) 4
24
     3  ! : +  4
7
```

Cette écriture directe des fonctions (avec ou sans l'opérateur (`:`)) n'est évidemment pas la seule façon de programmer. L'opérateur (`:`) avec des arguments numériques permet d'écrire des fonctions avec autant de lignes que l'on veut, incorporant si on le souhaite des structures de contrôle telles que if...then..else..., do...while... etc. Mais nous ne voulons pas faire un cours de J.

Les fonctions peuvent être composées entre elles avec de nombreux opérateurs. La composition la plus naturelle est celle d'un train de fonctions que les mathématiciens définissent ainsi : la somme de deux fonctions f et g se note f + g et, (f + g) x veut dire : f(x) + g(x). Ken Iverson étend cela à toutes les fonctions avec les syntaxes suivantes:

```
a (f g h) b    est (a f b) g (a h b)
  (f g h) b         est (f b) g (h b)          (On reconnaît le cas de (f + g) x)
  a (f g) b         est  a f (g b)
    (f g) b         est  b f (g b)
```

Cette règle s'applique à un train de fonctions de longueur quelconque par un parenthésage de droite à gauche :

`f g h r s t` est équivalent à `f (g h(r s t))` . En langage J, on appelle `(f g h)` une fourche (fork) et `(f g)` un hameçon (hook).

L'opérateur «Attache», noté `&`, attache deux fonctions pour en produire une seule avec la règle de composition suivante:

```
a f&g b   est     (g a) f (g b)
f&g b     est    f (g b)
```

Cet opérateur peut aussi attacher un nombre à une fonction.

```
     plusdeux=. 2&+                  (Addition de 2)

     plusdeux 5  12  0  1
7  14  2  3
```

```
     sin=. 1&o.              (car 1 o. x est sin x)
     cos=. 2&o.              (car 2 o. x est cos x)
     tan=. 3&o.              (car 3 o. x est tan x)
```

La division (et l'inverse) se note `%` et Pi fois se note `o.`
Pi divisé par 4 se note : `x=. o. %4`

```
     (sin % cos) x  (fourche identique à (sin x)%(cos x))
1
```

```
     tan x
1
```

Un opérateur dyadique comme `&` est appelé une *conjonction* en J. Un opérateur monadique comme `/` est appelé un *adverbe*. Un verbe produit par un opérateur peut être monadique ou dyadique. `+/` représente toujours la « réduction » quand on l'utilise de façon monadique. L'adverbe `/` est néanmoins rebaptisé « insertion ».

```
a
2 3 4
     +/a
9
```

Mais utilisé de façon dyadique, le verbe résultant est le produit extérieur.

```
b
5 6 1
     a  +/b
7  8 3
8  9 4
9 10 5
```

Ainsi disparaît une ancienne anomalie de notation (le produit extérieur était un opérateur monadique noté par un couple de symboles placés à gauche !).

Donnons quelques autres exemples de création de fonctions.

Connaissant le « nombre » (`#`), la « somme » (`+/`) et la « division » (`%`), on peut une fois de plus écrire la moyenne arithmétique.:

```
     moyenne=. +/ % #

     moyenne 5 4 1 6
4
```

`(+/ % #) x` est la fourche `(+/x) % (#x)`, c'est-à-dire la somme de `x` divisée par le nombre d'éléments de `x`.

Enfin, pour laissez rêveur, donnons un exemple (pris parmi des centaines) figurant à la page 49 du dictionnaire J de Ken Iverson. Anticipons sur l'opérateur de rang (noté `"`) qui permet de créer à partir d'une variable quelconque `b`, la fonction `b"_` qui rend la constante `b` . Exemple, si `b=. 'Y'`

```
     ( b"_ ) 'abcdef'
Y
     'ABC' ( b"_ ) 6 7 8 9
Y
```

Créons la fonction `m=. '-'"_` qui rend le signe moins. Et créons la fonction *moins* suivante avec un train de cinq fonctions :

```
     moins=. [ , m , ]
     'a' moins 'b'
a-b

     liste=. 'abcdefg'
     moins / liste
a-b-c-d-e-f-g
```

Interprétons d'abord, selon la règle, le train `[, m ,]` comme `[,(m,])`.

`(a moins )` est `(a[b),a(m,])b` par application d'une fourche.
`a[b` rend l'argument gauche `a`.
`a(m,])b` est une autre fourche `(a m b),a]b`, c'est-à-dire `'-',b`.
et finalement,
`(a moins b)` est `a,'-',b` (quelles que soient les chaînes `a` et `b`)

La « réduction » de la liste par la fonction `moins` ne réduit rien. On a inséré `moins` entre chaque lettre de la liste et on a effectué l'instruction obtenue. C'est pourquoi, dans un APL généralisé, on ne peut plus parler de réduction. On parlera de l'opérateur (ou mieux de l'adverbe) d'insertion.

Cet exemple montre l'écriture d'une fonction, un peu moins triviale que `+`, qui s'écrit sans qu'aucun argument ne soit utilisé. Cette écriture est dite être la forme *tacite* de la fonction opposée à la forme *explicite* avec ses arguments qui existe bien sûr aussi en J. Enfin, il existe aussi un opérateur qui permet de transformer, quand c'est possible, une forme explicite en forme tacite.
Donnons un dernier exemple qui est utilisé à chaque page du dictionnaire. Supposons que l'on veuille le résultat des fonctions `f`, `g`, `h`, `r` et `s` auxquelles on applique les arguments `a` et `b`. On écrit `a (f ; g ; h ; r ; s) b` et on obtient 5 résultats enclos! Exemple, pour deviner ce que sont les opérations `*` et `^`.

```
5 ( + ; - ; % ; * ; ^) 2
```

7	3	2.5	10	25

On a la chaîne suivante: `(5+3);((5-3);((5%3);((5*3);(5^3))))` qui, compte tenu de la définition de la primitive *lien*, donne bien un vecteur des cinq résultats enclos.

3 - Les structures de données

Bien plus que le simple ajout de la primitive de représentation scalaire (enclos `<`), un vrai bouleversement est apparu vers 1983 dans la vision et dans la manipulation des données d'abord avec l'opérateur de rang sur lequel on reviendra, puis plus récemment avec une vue plus simple encore et qui s'impose dès qu'on veut étendre les

opérateurs à toutes les fonctions (mais qui, à part Ken, l'avait vu ?). Étudions l'ancienne propagation de type `+\` .

Si on regarde `+\2 1 4 3` comme `(2),(2+1),(2+1+4),(2+1+4+3)` , on peut aller plus loin et dire que c'est :

```
(+/2),(+/2 1),(+/2 1 4),(+/2 1 4 3)
```

Profondément, l'opérateur \ découpe la liste en morceaux de longueurs croissantes et, sur chaque morceau, on applique +/. Étendons l'APL : `\` devient un adverbe (opérateur) réalisant ce découpage préalable. A la place de `+/` mettons *enclos.*

```
      <\ 1 2 4 8 16
```

1	1 2	1 2 4	1 2 4 8	1 2 4 8 16

Si on déclot cette liste, chaque vecteur est complété de zéros pour être concaténé sur une dimension supplémentaire.

```
      > <\ 1 2 4 8 16
1 0 0 0  0
1 2 0 0  0
1 2 4 0  0
1 2 4 8  0
1 2 4 8 16
```

On aurait obtenu le même résultat avec la primitive *droit* (`]`).

```
      ]\ 1 2 4 8 16
1 0 0 0  0
1 2 0 0  0
1 2 4 0  0
1 2 4 8  0
1 2 4 8 16
```

Et, si on reprend la fonction *moins* créée plus haut:

```
      moins/ \ 'abcde'
a
a-b
a-b-c
a-b-c-d
a-b-c-d-e
```

La forme dyadique fixe la taille des morceaux. Par exemple:

```
3  <\  2 3 7 8 6 1 5
```

2 3 7	3 7 8	7 8 6	8 6 1	6 1 5

Et si on reprend la fonction `moyenne=. +/ % #` définie plus haut, on obtient ainsi la moyenne mobile de la série sur 3 positions :

```
     3 moyenne \ 2 3 7 8 6 1 5
4 6 7 5 4
```

Sur le même mode de pensée, le produit interne devient un mode de découpage avant d'être un mode de calcul.

```
     M                    N
2 3 4                3 4 5
2 5 1                2 1 2
                     6 3 4
```

Le produit interne est:

```
     M +/ . * N
36 23 32
22 16 24
```

Il reste très agréable d'être libre d'écrire simplement `x=. +/ . *` et d'obtenir, sans effort, le même résultat en tapant `M x N` (imaginez pouvoir écrire directement en APL : `x ← .+×`).

Le mode de découpage formel du produit intérieur peut être en partie compris en utilisant < au lieu de `+/` .

```
M < . * N
```

6 8 10	6 8 10
6 3 6	10 5 10
24 12 16	6 3 4

```
     M moyenne . * N
12     7.67   10.67
 7.33  5.33    8
```

Il y a bien d'autre opérateurs (infix, suffix, outfix, oblique,...) qui donnent un découpage particulier des données. Aucun ne tombe du ciel gratuitement. Ils ont tous été conçus vers 1978-1983 et testés dans de très nombreuses modélisations et finalement retenus ... parce qu'ils sont utiles (comme l'ont été et le sont la « réduction » et la « propagation », la « compression » ou l'expansion »).

Pour aller plus avant avec les structures des données encloses, présentons les deux autres opérateurs de composition de fonctions.

L'opérateur « sur » noté `@` défini ainsi :

```
a f@g b        est  f (a g b)
 f@g b         est     f (g b)
```

L'opérateur « sous » noté `&.` est l'opérateur de dualité défini ainsi:

```
a f&.g b       est  invf (g a)f(g b)      où invf est la fonction
                                          inverse de f
f&.g b         est     invf  f (g b)
```

On sait par exemple que la multiplication est duale de l'addition relativement au logarithme (dont l'inverse est l'exponentielle). En APL, cela s'écrit:

```
(3 × 4)    ↔   (⋆(⍟3) + (⍟4))
```

En J, la multiplication se note `*` et `^.` représente le logarithme.
Posons `log =. ^.`

```
(3*4)     ↔    3 +&.log  4
```

que l'on peut lire (3 fois 4) est équivalent à (3 plus 4) sous le logarithme.

L'opérateur *each*, noté (¨) en APL2, ne représente qu'un traitement *sous déclos*. Créons, pour l'exemple, la variable `A=. M;V` (`M` et `V` tels que ceux définis plus haut).

```
                    A
┌─────────┬─────────┐
│1 2 3 4  │9 0 9 7 8│
│5 6 7 8  │         │
└─────────┴─────────┘
```

```
+/&.> A
```

```
┌─────────┬──┐
│6 8 10 12│33│
└─────────┴──┘
```

(en J, `+/` travaille par défaut sur la 1ère dimension)

```
each=. &.>                    (définition d'un opérateur)

                          +/ each A
```

6 8 10 12	33

Prenons un dernier exemple d'usage naturel de sélection par compression qui produit des résultats de tailles inégales que l'on veut donc enclore.

La primitive `#:` est le décode à base 2 et la primitive `i.` est le iota en origine zéro. `#: i. 8` est équivalent à `⍉2 2 2⊥⍳8` (avec `⎕IO←0`) en APL.

```
     #:  i. 8
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1
```

Cette table présente toutes les extractions possibles de zéro à trois objets parmi 3. On peut l'utiliser pour comprimer le vecteur `2 3 4` à l'aide de la primitive de réplication `#` qui est de rang 1 comme on l'expliquera plus tard et qui s'applique terme à terme entre des cellules qui sont des vecteurs. En usage direct, les résultats des compressions sont complétés avec des `0` avant caténation :

Posons : `c=. # i. 8`

```
     c # 2 3 4
0 0 0
4 0 0
3 0 0
3 4 0
2 0 0
2 4 0
2 3 0
2 3 4
```

Mais si on applique l'enclos (`<`) *sur* la compression (`#`), avec l'opérateur `@`, on obtient :

`c <@# 2 3 4`

4	3	3 4	2	2 4	2 3	2 3 4

Ré-interprétons cette instruction. Appelons (`ci`) la ligne i du tableau `c` . (`ci`) est la cellule sur laquelle la primitive `#` travaille en bouclant et en exécutant, pour chaque ligne de c : `< (ci # 2 3 4)` selon le mécanisme de la conjonction `@` définie par `x f@g y ≡ f (x g y)`. L'instruction `<@#` se lit : enclos sur réplication.

Mettons le résultat dans `a.` `a=. c <@# 2 3 4`

Si on veut sommer chaque cellule, on écrit `+/ each a`, ou, directement (puisque qu'on a définit `each` par `&.>`) :

```
+/&.> a
```

0	4	3	7	2	6	5	9

Rappelons que `&.` est l'opérateur de dualité, et que `&.>` est la conjonction qui, appliquée à `f` , fait boucler `f&.>` sur chaque cellule enclose pour :

1 - déclore la cellule
2 - appliquer `f` au contenu de la cellule
3 - enclore le résultat

On peut ensuite déclore le tout :

```
      > +/&.> a
0 4 3 7 2 6 5 9
```

Ce qui aurait être obtenu directement par `c +/@# 2 3 4 .`

```
      c +/@# 2 3 4
0 4 3 7 2 6 5 9
```

Ceci montre combien trois choses sont intimement liées pour la structure des données: les primitives (incluant l'enclos), les opérateurs (s'appliquant à toute fonction) et la vision de l'argument comme tableau de scalaires ou vecteur de vecteurs ou tableau de vecteurs, etc. Cette vision des données dépend de ce qu'on appelle *le rang* d'une fonction. Ce rang, en SHARP APL comme en J, est modifiable par un opérateur. C'est un des instruments les plus puissants de la programmation.

Par exemple, si on écrit `+"_ 0` (où _ représente l'infini), cela veut dire que l'argument gauche de l'addition est vu comme un tout et que l'argument droit est vu comme une collection de scalaires (objets de rang 0). Si on écrit `+"1 1` , cela veut dire que chaque cellule des arguments gauche et droit sont de rang 1, c'est à dire qu'ils sont vus comme des structures de vecteurs.

Par exemple:

```
      a               b
1 2             10 20
3 4
      a +"_ 0   b
11  12
13  14

21  22
23  24
```

C’est équivalent à déclore le résultat de `a` pris comme un tout avec les scalaires de `b` .

```
1 2
3 4
```

```
+ each
```

```
10
```

```
20
```

```
     a +"1 1  b
11 22
13 24
```

C’est équivalent à déclore le résultat de `a` et `b` pris comme des vecteurs de vecteurs.

```
1 3
```

```
2 4
```

```
+ each
```

```
10 20
```

En reprenant l’exemple de l’extraction par la matrice binaire `c` donnée plus haut et en l’appliquant non plus au vecteur `2 3 4` mais à la matrice `w` suivante:

```
      w                  c
1 2 3                0 0 0
4 5 6                0 0 1
7 8 9                0 1 0
                     0 1 1
                     1 0 0
                     1 0 1
                     1 1 0
                     1 1 1
```

nous pouvons apprécier les effets différents suivants :

- appliquer toute la matrice `c` à chaque vecteur (i.e. chaque ligne) de `w` avec la compression `#` puis l'enclore.

```
c (<@#)"_ 1 w
```

3	2	2 3	1	1 3	1 2	1 2 3
6	5	5 6	4	4 6	4 5	4 5 6
9	8	8 9	7	7 9	7 8	7 8 9

3	2	2 3	1	1 3	1 2	1 2 3
6	5	5 6	4	4 6	4 5	4 5 6
9	8	8 9	7	7 9	7 8	7 8 9

- appliquer chaque vecteur (ligne) de `c` à `w` toute entière.

```
c (<@#)"1 _ w
```

7 8 9	4 5 6	4 5 6 7 8 9	1 2 3	1 2 3 7 8 9	1 2 3 4 5 6	1 2 3 4 5 6 7 8 9

On conçoit sans peine la richesse considérable de cette notion de rang qui donne toutes les visions possibles d'une même donnée. Cette notion impose au langage au moins trois choses:

- que chaque primitive ait des rangs (monadique, gauche et droit) *par défaut* (par exemple `+` , `-` etc. sont de rang 0, l'inversion de matrice est de rang (droit) 2);
- que des règles d'héritage soient pourvues lors de la composition de fonctions avec les opérateurs;
- enfin, qu'une primitive donne l'information sur le rang d'une fonction (composée ou non).

Donner un sens complet et formel à cette possibilité de jouer sur le rang a pris un temps considérable durant le début des années 1980, et quelques hésitations ont encore dû être corrigées au début du langage J entre 1990 et 1993.
A posteriori et à l'usage, l'idée de jouer sur le rang s'avère une idée à proprement parler merveilleuse. Ce concept de rang d'une fonction est encore étranger aux mathématiques qui n'en ont pas besoin car toutes les fonctions y sont scalaires *par définition* (on sait que les fonctions multiformes dans le plan complexe ne sont pas des fonctions et que les surfaces de Riemann ont pour but précisement d'en faire des fonctions, c'est-à-dire de leur faire donner une image et une seule par objet de départ. De même, les classes d'équivalence remplacent à bon escient toute une collection par un ensemble (vu comme un scalaire), les fonctions vectorielles prennent en fait les vecteurs comme des scalaires, etc).

Nous ne multiplierons pas les exemples. Ceux qui viennent d'être présentés suffisent à donner une idée de l'homogénéité des idées nouvelles introduites sur les structures de données. On voit aussi à quel point ces idées se servent toutes les unes des autres de façon très naturelle en s'imbriquant intimement au point qu'il est difficile de trouver pour chacune des exemples purs et isolables.

4 - La rationalisation d'APL

Les impropriétés d'APL dont certaines ont été évoquées au cours du texte ont été complètement éliminées. Les crochets de l'opérateur d'axe (comme dans `⌽[1]` ou `+/[1]`) sont remplacés par l'opérateur de rang et ceux de l'indexation (comme pour `M[I;J]`) par la primitive { que nous ne présenterons pas, bien que sa syntaxe et sa richesse soient éblouissantes.

D'autre part, il n'y a plus de fonctions niladiques; chaque fonction est ambivalente (monadique ou dyadique) même si elle est composée de plusieurs primitives et de plusieurs opérateurs.

Comme on l'a vu, l'écriture de fonction peut se passer de tout éditeur (il en existe toujours un bien sûr) ou de fonctions de type `⎕FX` qui rendait un nom au lieu d'une fonction ! Au premier niveau, la définition directe d'une fonction est simplissime. Ainsi, `fonc=.+ * -` définit une fonction qui fait le produit de la somme et de la différence de ses arguments (on rappelle qu'en J le symbole `*` désigne la multiplication). `A fonc B`, défini par une *fourche*, réalise bien le calcul suivant : `(A+B)*(A-B).`

L'affectation sur sélection (du type `A[I;J]←M` ou `(1 1⍉M)←V`) est obtenue par l'opérateur `}` appelé *amende* avec la syntaxe `x m} y` pour mettre les éléments de `x` aux places spécifiées par `m` dans `y` (où `m` peut-être n'importe quel argument convenable de la très riche primitive d'indexation `{`).
Les règles d'évaluation sont simples pour les primitives comme pour les opérateurs. La création de nouvelles primitives (comme `#` pour *nombre* et *réplication*) libère des symboles qui sont libres pour systématiser l'écriture; ainsi `+/` qui doit être monadique et dyadique peut prendre un sens dyadique qui fait disparaître la notation impropre du produit extérieur.

Nous en resterons là sur cet aspect de la rationalisation car les deux autres thèmes, l'écriture des fonctions et la structure des données que nous avons aussi survolés, ont participé aussi à l'illustration de cette idée.

Avant de conclure, il nous faut redire que nous avons volontairement écarté de notre préoccupation toutes les questions de notations, de vocabulaire et d'environnement. Presque tous les symboles (sauf quelque uns comme `+ - , > < ? ! |` etc.) et presque tous les noms (sauf plus, moins, supérieur, inférieur, etc.) ont changé. Il n'y a plus de « workspace » et l'on est sous un éditeur graphique WINDOWS, MacIntosh ou OS2). Inutile de dire que c'est un beau sujet de débat aussi. Mais rappelons-nous : nous nous demandions si le **langage** J était vraiment de l'APL.

Il y a un premier point de vue qui est celui présenté par Donald McIntyre en 1991 (*Mastering J*, ACM APL91 Conférence Proceedings): « J est-il un dialecte d'APL ? je pense que la réponse est simple. APL a commencé comme une forme exécutable de la notation d'Iverson; aujourd'hui J est la forme exécutable de la dernière version de la notation d'Iverson. Beaucoup de personnes ont contribué à l'évolution de systèmes APL disponibles commercialement, mais le principal architecte du langage est l'homme qui a publié *A Programming Language* en 1962 et qui nous donne aujourd'hui la version qu'il appelle J ». Ce point de vue est

celui de la continuité et il mérite grandement d'être retenu car les quelques exemples que nous avons donnés montrent à quel point tout ce qui fait APL est maintenu et que J en est un progrès.

Mais il y a l'autre point de vue. Celui que le texte de Phil Abrams cité à la deuxième page, en début d'article, permet de formuler quand il dit: « La prochaine étape est, peut-être, de créer un langage qui est à APL ce qu'APL est au FORTRAN ». Si J est ce nouveau langage, n'est-il pas si neuf et si riche qu'il est hors de portée d'être reconnu comme de l'APL par beaucoup d'APListes qui ont l'excuse, il est vrai, dans certains milieux, de n'avoir pas vu arriver une seule primitive nouvelle ni un seul opérateur nouveau depuis 10 ans et plus. En tout cas, il y a un tel saut entre J et APL que tous ceux qui s'y essayent retrouvent à la fois la nécessité d'un (presque) complet réapprentissage et les joies qu'on connaissait à programmer de beaux idiomes de six ou huit façons différentes.
Alors, continuité ou bouleversement ? C'est sans doute moins simple. On serait tenté de dire: bouleversement PARCE QUE continuité. Comme nous l'évoquions au tout début, c'est parce que Ken Iverson allie une exigence et un talent hors de l'ordinaire (cette touche de génie qui ... etc.), une rigueur et un travail acharné qu'il a pu donner tant de solutions audacieuses et novatrices à des problèmes difficiles. C'est aussi pour ces mêmes raisons qu'il a la chance de refonder avec Eric Iverson une équipe, où l'exceptionnelle qualité et le dévouement de Roger Hui lui ont permis la réalisation d'un interpréteur et d'un système J.

5 - Références

Phil Abrams, *What's wrong with APL ?* APL75 Conference Proceedings, Pisa.

R. Bernecky, K. Iverson, *Operators and enclosed arrays*, I.P. SHARP, 1980

Roger Hui, K. Iverson, E. McDonnell, A. Whitney, *APL\?* APL90, APL Quote-Quad 20.4, 1990

Roger Hui, K. Iverson, E. McDonnell, *Tacit definition*, APL91, APL Quote-Quad 21.4, 1991

Ken Iverson, *A Programming Language*, John Wiley&Sons, 1962

Ken Iverson, *Operators and functions*, IBM Research Report#RC7091, 1978

Ken Iverson, *Rationalized APL*, I.P SHARP Research Report 1, April 1983

Ken Iverson, *J Introduction and Dictionary*, Iverson Software Inc., 1994

Ken Iverson, *Concrete Math Companion*, Iverson Software Inc., 1995

Donald McIntyre, *Mastering J*, APL91, APL Quote-Quad 21.4, 1991