APL
vu du Ciel

Bernard Legrand
pour AFAPL

14 Juin 2006

InterFluences

InterFluences
Centre d'affaires NCI
55, rue Sainte Anne - 75002 Paris

01 48 69 19 52
contact@interfluences.fr

Ce document a été composé sous
Microsoft Word ©

Les exemples APL ont été réalisés
au moyen de **Dyalog APL** Version 10
distribué par Dyalog Ltd.
www.dyalog.com

Imprimé par Office Parisien
7, rue Notre-Dame des Victoires
75002 Paris
01 40 20 43 22
officeparisien@wanadoo.fr

1. Préambule

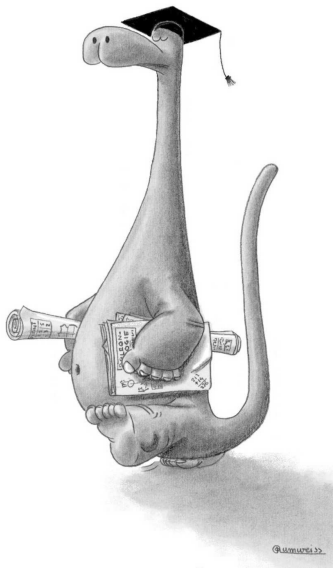
Ce document a été spécialement rédigé pour une conférence organisée le 22 Juin 2006 à l'initiative de AFAPL, Association Francophone pour la promotion du langage APL, conférence spécialement dédiée à notre ami Henri Sinturel, qui nous a quittés trop tôt.

Cette conférence ne s'adresse pas à des habitués du langage : tout ce qui va être dit leur est familier depuis des années, et je n'ai rien à leur apprendre. Je vais seulement tenter de faire découvrir ce bel outil intellectuel à des personnes qui n'en ont jamais entendu parler, avec le secret espoir de faire un ou deux adeptes, et ainsi passer le témoin.

En effet, la présentation qui va suivre se veut avant tout un pont jeté entre deux générations :

- La génération des "diplodocus" : ceux qui ont connu l'informatique avant le PC, avant même l'apparition des premiers écrans (pensez-donc, ils ont même connu la carte perforée !), et qui ont trouvé en APL un moyen de traiter tous les problèmes que les grandes équipes informatiques étaient alors dans l'incapacité de traiter dans des délais raisonnables. Cette génération est en voie d'extinction.

- La génération de ceux qui n'ont connu que le micro-ordinateur, Internet et l'hypertexte, ceux qu'on a nourris dès l'école à grand renfort d'Excel et qui pourraient penser que tout peut se faire en trois clics de souris.



Mais décrire le langage APL, que ce soit en 3 ou en 30 pages, est aussi difficile que décrire une rencontre de tennis ou le vol d'une mouette : le document écrit est incapable de transmettre ce que procure l'observation directe. Aussi les pages qui suivent ne donnent-elles qu'une vision très déformée et très parcellaire de toute la richesse d'APL.

C'est en quelque sorte APL vu de très haut ; vu du Ciel. Salut à toi, Henri.

J'espère rester fidèle à l'esprit d'APL, et à l'enthousiasme de celles et ceux qui depuis des années en font l'éloge et la promotion.

Je vais aussi essayer de répondre aux questions les plus souvent posées, aux critiques les plus souvent formulées, mais sans pouvoir embrasser toutes les facettes du débat. Aussi j'invite tous ceux que ce langage intrigue à passer une heure avec moi autour d'un écran d'ordinateur.

Attachez vos ceintures, nous partons !

2. Le premier contact est aisé

Pour plus de clarté, dans les pages qui suivent, les textes frappés par l'opérateur apparaissent en rouge, tandis que la réponse de l'ordinateur reste en noir.

La première image que donne APL, c'est celle d'une calculette :

```
      27 + 53
80
      1271 - 708
562
      59 × 8
475
      86 ÷ 4
21.5
```

Première surprise, la multiplication est représentée par un vrai signe `×`, et non par l'abominable étoile `*` qui prévaut désormais dans tous les langages informatiques. Idem pour la division.

Pour créer une variable, il suffit de frapper au clavier le nom qu'on veut lui donner, d'indiquer par la flèche `←` qu'on lui attribue une valeur, et de frapper à la suite la ou les valeurs.

Par exemple `TVA ← 19.6` (lire : *TVA* reçoit `19.6`)
ou encore `Années ← 1952 1943 1986 2005`

Pour connaître la valeur d'une variable, il suffit d'en frapper le nom, comme on le voit ici.

```
      Années
1952 1943 1986 2005
```

3. Une approche globalisée

APL présente cette particularité de savoir travailler spontanément entre deux ensembles de nombres de même "forme" (de même taille). Ci-dessous, par exemple, on introduit dans deux variables la liste des prix de 5 produits, et la quantité achetée pour chacun de ces produits :

```
      Prix ← 5.2  11.5  3.6  4  8.45
      Qtes ← 2    1     3    6  2
```

Si on multiplie ces deux variables l'une par l'autre, les éléments sont multipliés terme à terme pour produire un résultat de même taille :

```
      Couts ← Prix × Qtes
      Couts
10.4  11.5  10.8  24  16.9
```

Cette approche globalisée économise la plupart des "boucles" qui alourdissent tant la programmation dans tous les langages en usage actuellement.

Cette propriété reste vraie pour des tableaux de valeurs, et cela quelle qu'en soit la taille, ou le nombre de dimensions.

Ainsi, un Directeur des ventes fait des prévisions de ventes pour 4 produits sur les 6 mois à venir, et les consigne dans la variable *Prévu*.
Au bout des 6 mois, il consigne une réalité un peu différente dans la variable *Réel*.

```
        Prévu                          Réel
150 200 100  80  80  80        141 188 111  87  82  74
300 330 360 400 500 520        321 306 352 403 497 507
100 250 350 380 400 450        118 283 397 424 411 409
 50 120 220 300 320 350         43  91 187 306 318 363
```

Il est évident que le premier réflexe de notre Directeur sera de vouloir connaitre les écarts, ce qu'il obtiendra le plus simplement du monde en écrivant :

```
      Réel-Prévu
¯9  ¯12   11   7   2   ¯6
21  ¯24   ¯8   3  ¯3  ¯13
18   33   47  44  11  ¯41
¯7  ¯29  ¯33   6  ¯2   13
```

Notez que le signe moins attaché aux valeurs négatives est décalé vers le haut, afin de le distinguer du signe moins de la soustraction.

Pour obtenir un résultat similaire au moyen d'un langage informatique traditionnel, il faudrait plusieurs instructions qui masquent le but du calcul derrière des arcanes de programmation. Voici par exemple ce qu'on écrirait en PASCAL :

```
DO UNTIL I=4
   DO UNTIL J=6
      ECARTS(I,J):=REEL(I,J)-PREVU(I,J)
   END
END.
```

Eh bien, croyez-le ou non, il se trouve, au sein des structures universitaires françaises, des personnes pour soutenir que cette seconde manière d'écrire les choses est plus simple, ce qui relève de la maxime bien connue des Shadoks :

"*Pourquoi faire simple quand on peut faire compliqué ?*".

On vient de voir qu'APL sait travailler entre deux données de même taille ; il sait aussi travailler entre une donnée quelconque et une valeur unique, isolée, qu'on appellera un ***scalaire***.

Tel serait le cas si on voulait calculer les montants d'une TVA à 19.6% appliquée sur notre variable *Prix* vue plus haut.

Qu'on l'écrive : `Prix×0.196` , ou qu'on l'écrive : `0.196×Prix`
le résultat sera le même : `1.0192 2.254 0.7056 0.784 1.6562`

Un résultat qu'il faudrait arrondir, mais ce n'est pas notre propos ici.

4. De nouveaux symboles

L'intelligence humaine ne se laisse pas enfermer dans quatre ou cinq opérations de base, même si telle est bien la limite imposée par la plupart des langages de programmation même les plus modernes.

Le génial créateur d'APL, Kenneth E. Iverson a donc adjoint au jeu de symboles que nous partageons tous en commun, un certain nombre de nouveaux symboles :

La fonction "*Maximum*" rend le plus grand de deux nombres, ou de deux ensembles de nombres comparés terme à terme ; il y a aussi, on s'en doute, un symbole "*Minimum*".

```
      75.6 ⌈ 87.3
87.3

      11 28 52 14 ⌈ 30 10 50 20
30 28 52 20
```

La fonction "*Minimum*" s'emploie de même :

```
      11 28 52 14 ⌊ 20
11 20 20 14
```

APL comporte au total environ 70 symboles. Sachant que certains signes se déclinent sous deux formes, on peut ergoter longtemps sur le nombre exact de signes ; retenez juste cet ordre de grandeur.
Rien de bien inquiétant à cela : nombre de ces signes nous sont familiers (pensez à × ou > ou encore + et −, mais aussi ! et bien d'autres).

5. Le double usage des symboles

Ce n'est pas une particularité d'APL ; l'algèbre nous a familiarisé avec le fait qu'un symbole aussi banal que le signe moins puisse avoir deux usages :

Dans l'expression *a = x-y* il représente l'opération "*soustraction*"
Tandis que dans *a = -y* il représente l'opération "*opposé de*", qui change le signe

La première forme sera dite usage "***dyadique***" du symbole.
La seconde forme sera dite usage "***monadique*** " du symbole.

Tel est également le cas en APL, où la plupart des symboles peuvent avoir les deux usages.

Par exemple, pour connaître la forme (les dimensions) d'un objet, on utilise la lettre grecque Rho (⍴), qu'on peut lire "*forme de* ...", dans son usage monadique :

```
      ⍴ Prix
5                    Prix  comporte 5 éléments

      ⍴ Prévu
4 6                  Prévu  comporte 4 lignes de 6 éléments
```

Utilisé de manière dyadique, le même symbole sert à organiser des valeurs sous une forme imposée.
Par exemple, si nous voulons créer le tableau ci-contre ⇨

```
25 60
33 47
11 44
53 28
```

il nous faut indiquer deux informations :

- d'abord la ***forme*** à donner au tableau : `4 2` (4 lignes sur 2 colonnes)
- ensuite le ***contenu*** du tableau : `25 60 33 47 11` etc..

C'est le symbole Rho qui va conjuguer ces deux informations entre elles :

```
      Tab ← 4 2 ⍴ 25 60 33 47 11 44 53 28
```

Une nouvelle variable, *Tab*, est ainsi créée, qui est le tableau désiré.
C'est ainsi qu'on été constituées les variables *Prévu* et *Réel* utilisées plus haut.

Conventions

En APL, on a pris l'habitude d'appeler ***Vecteur*** une liste de valeurs, qu'elle soit composée de nombres, comme *Prix* et *Qtes*, ou de lettres comme *'Il était une fois'*. On appelle ***Matrice*** un tableau à deux dimensions, comme *Prévu* ou *Tab*, et ***Scalaire*** une valeur isolée, que ce soit un nombre comme `456.18` ou une lettre unique, comme *'Q'*.

6. La réduction, notation unificatrice

Souvenez-vous, nous avons calculé des coûts : `10.4  11.5  10.8  24  16.9`

Mais combien avons-nous dépensé au total ? Les mathématiciens, gens créatifs, ont de longue date inventé le symbole ∑ , toujours délicieusement assorti d'indices de début et de fin, ce qui impose une typographie peu compatible avec l'usage élémentaire d'un traitement de texte.

En APL, l'opération se note ainsi :

```
      +/ Couts
73.6
```

Simple, non ? À quoi serviraient des indices de début et de fin ? On totalise tous les éléments de la donnée et on n'en parle plus !

On dira qu'on a procédé à une "***Réduction par la somme***" de la variable `Couts`.

Cherchons à mieux comprendre ce qui s'est passé.

Quand nous écrivons une instruction telle que : `+/ 21 45 18 27 11`
C'est comme si nous écrivions : `21 + 45 + 18 + 27 + 11`
et nous obtenons la somme : `122`

En fait, tout se passe comme si on avait "*infiltré*" le signe + entre les valeurs traitées.

Mais alors, si nous écrivons : `×/ 21 45 18 27 11`
C'est comme si nous écrivions : `21 × 45 × 18 × 27 × 11`
et nous obtiendrons le produit : `5051970`

De même, si nous écrivons : `⌈/ 21 45 18 27 11`
C'est comme si nous écrivions : `21 ⌈ 45 ⌈ 18 ⌈ 27 ⌈ 11`
et nous obtiendrons le plus grand terme : `45`

La ***Réduction*** appartient à une catégorie particulière de symboles, qu'on appelle les ***opérateurs***, alors que tous les autres symboles (+ - × ⌈ > ⍴ ...) représentent des ***fonctions*** (*addition*, *soustraction* ... *maximum* ... *forme*, etc.).

Les opérandes d'une fonction sont des données : `Prix × Qtes`
L'opérande gauche d'un opérateur est une fonction : `+/ Couts`

(une définition plus rigoureuse sortirait du cadre simplificateur de cet exposé).

A ce stade, disons que la Réduction permet d'effectuer autant d'opérations différentes qu'on peut placer de signes opératoires différents (ou de noms de programmes !) à sa gauche : c'est une notion d'une très grande généralité.
Songez en effet qu'en mathématiques, on a inventé ∑ pour la somme, ¶ pour le produit, *Min* et *Max* pour le minimum ou le maximum (et encore certains notent-ils *Inf* et *Sup* !).

En APL, le seul signe / suffit à **unifier** toutes ces notations !

APL comporte en tout 6 opérateurs dans les versions les plus pauvres, 9 dans la version Dyalog APL qui a servi à rédiger ce document.

7. Premier programme

Nous décidons de calculer la moyenne de la suite de nombres que voici :

```
      Val ←22 37 41 19 54 11 34
```

Il nous faut pour cela diviser l'un par l'autre deux termes :

- d'une part la ***somme*** de ces valeurs : `+/Val` qui donne `218`
- d'autre part leur ***nombre*** : `⍴Val` qui donne 7

On peut écrire le calcul en une seule formule : `(+/Val)÷(⍴Val)`

Comme ce type de travail a de bonnes chances de nous servir souvent, il est préférable de mémoriser cet enchainement de calculs sous forme d'un programme.
En APL on préfère l'appellation ***fonction définie*** au mot programme.
C'est un outil supplémentaire qui vient s'ajouter au jeu des fonction fournies par le langage sous forme de symboles (`+ × ⌈ > ⍴`), qu'on appelle ***fonction primitives***.

Il sortirait du cadre de ce document d'expliquer comment on procède pour définir un tel programme ; disons qu'il revêtirait peu ou prou la forme suivante :

```
      ∇ R ← Moyenne V
[1]   R ← (+/V)÷(⍴V)
      ∇
```

`Moyenne` est le nom du programme
`V` représente symboliquement la liste des valeurs qu'on lui apportera à droite
`R` représente le résultat élaboré par le calcul, et dont la valeur "sortira" au final

Les signes typographiques `∇` (appelés "carottes") servent uniquement à marquer le début et la fin de la forme imprimée du programme.

Une fois définie, cette fonction s'utilise de la manière la plus simple qui soit :

```
      Moyenne Val
31.1428571428
      Moyenne 12 74 56 23
41.25
```

On peut désormais l'incorporer dans une expression plus complexe :

```
      10×Moyenne 12 74 56 23
412.5
```

8. Indiçage

Reprenons notre vecteur de nombres : `Val ←22 37 41 19 54 11 34`

Pour en extraire le 4ème élément, nous écrirons : `Val[4]`

Dans d'autres langages on utilise des parenthèses au lieu de crochets ; ce n'est pas très différent.

Ce qui est nouveau, c'est qu'on peut extraire **plusieurs** éléments en une seule opération.

```
      Val
22 37 41 19 54 11 34
      Val[2 4 7 1 4]      On voit qu'on peut même extraire deux fois le même élément
37 19 34 22 19
```

Et, bien entendu, on peut de la même manière modifier un ou plusieurs éléments de *Val* désignés par leurs indices, à condition de fournir autant de valeurs que d'éléments à modifier (ou une valeur identique pour tous) :

```
      Val[3 5 1] ← 300 77 111
      Val
111 37 300 19 77 11 34                (on a marqué en bleu les éléments modifiés)
```

Il est très fréquent d'avoir besoin d'extraire les premiers éléments d'une liste de valeurs, par exemple les 5 premiers. Rien de plus simple :

```
      Val[1 2 3 4 5]
111 37 300 19 77
```

Mais si on devait extraire les 500 premiers termes d'une longue liste, frapper les entiers de 1 à 500 est naturellement impossible.

C'est pourquoi APL est doté du symbole ⍳ (Iota), qui produit la suite des *n* premiers entiers.

Ainsi, au lieu d'écrire `1 2 3 4 5 6 7 8`, il suffit d'écrire `⍳8`.

Et pour extraire les 500 premiers termes d'un grand vecteur, on écrira : `Big[⍳500]`

9. Calculer sans programmer

Les vingt salariés d'une entreprise sont classés en trois catégories hiérarchiques, qui sont tout simplement numérotées 1 2 3.
On a consigné dans deux variables les salaires et les catégories de ces salariés ; en voici un affichage partiel :

```
Salaires :    4225 1619 3706 2240 2076 1389 3916 3918 4939 2735
Catégories :     3    1    3    2    2    1    3    3    3    2
```

Ne voilà-t-il pas que ces salariés veulent être augmentés ! (où va notre pauvre monde ?).

Une indiscrétion nous a renseignés sur leurs prétentions : ils veulent un pourcentage d'augmentation différent pour chaque catégorie, selon le barème suivant :

Catégorie	Augmentation demandée
1	8%
2	5%
3	2%

Combien cela va-t-il coûter à l'entreprise ?

Constituons une variable qui contiendra les trois taux ci-dessus :

```
      Taux ← 8 5 2 ÷ 100          (rappelons qu'on peut diviser 3 nombres par un seul)
      Taux
0.08  0.05  0.02
```

Alors, comme le premier salarié est en catégorie 3, le taux qui s'appliquerait à lui est :

```
      Taux[3]
0.02
```

Poursuivons : les 5 premiers salariés étant respectivement en catégories 3 1 3 2 2, ils attendent donc les augmentations suivantes :

```
      Taux[3 1 3 2 2]
0.02 0.08 0.02 0.05 0.05
```

Plus généralement, les taux afférant à nos 20 salariés s'obtiendraient comme ceci :

```
      Taux[Catégories]
0.02 0.08 0.02 0.05 0.05 0.08 0.02 0.02 0.02 0.05 0.05 0.02 etc.
```

Ayant 20 taux, il suffit de les multiplier par les 20 salaires pour connaitre les augmentations individuelles :

```
      Salaires × Taux[Catégories]
84.5 129.52 74.12 112 103.8 111.12 78.32 78.36 98.78 136.75 etc.
```

Finalement, en additionnant le tout, on saura combien il en coûte à l'entreprise :

```
      +/ Salaires × Taux[Catégories]
2177.41
```

On constate que :

- cette formule reste valable quels que soient le nombre de salariés et de catégories
- le résultat a été obtenu sans écrire aucun programme
- et cette phrase peut se lire en Français le plus simplement du monde :

  Somme des Salaires multipliés par les Taux afférant aux Catégories

Cet exemple démontre clairement qu'il existe d'autres voies de raisonnement que celles qui prévalent dans l'enseignement de l'informatique depuis 40 ans, et qui sont hélas fort réductrices.
Cette différence, cette originalité, introduites par APL sont capitales, car elles concourent à l'ouverture d'esprit et à l'épanouissement intellectuel des personnes qui le pratiquent.

10. Nos amis les binaires

APL fait un très large usage des informations binaires. Elles sont le plus souvent créées au moyen des fonctions de relation telles que = ou > :

```
      Salaires > 3000
1 0 1 0 0 0 1 1 1 0 1 1 0 0 1 1 0 0 0 0

      Réel > Prévu
0 0 1 1 1 0
1 0 0 1 0 0
1 1 1 1 1 0
0 0 0 1 0 1
```

On voit instantanément les bons résultats commerciaux

Première nouveauté, APL est le **seul** langage informatique doté des 6 fonctions de relation que nous connaissons, représentées sous leur forme mathématique usuelle :

< ≤ = ≥ > ≠

Certes, les autres langages "se débrouillent" mais enfin, il nous semble qu'en ce début de 21ème siècle, il n'est pas totalement déraisonnable de demander que l'inégalité ≥ ne soit pas représentée par > = et que ≠ ne soit pas représenté par < > , que diable !

Naturellement, on peut se livrer sur ces informations binaires à toutes les opérations de l'algèbre de Boole et, là encore, les symboles utilisés sont ceux que pratiquent, partout dans le monde, les mathématiciens de toutes nationalités :

- La fonction ET se note bien ∧ (on note AND dans bien des langages)
- La fonction OU se note bien ∨ (on noterait OR dans ces langages)

Ainsi, si je cherche les personnes en catégorie 3 dont le salaire est inférieur à 4000 euros, je peux écrire :

```
      (Catégories=3)∧(Salaires<4000)
0 0 1 0 0 0 1 1 0 0 0 0 0 0 0 1 0 0 0 1
```

APL offre en fait **toutes** les fonctions de l'algèbre de Boole, y compris certaines fonctions comme NOR et NAND (NI et Non-ET) inutilisées en gestion, mais particulièrement utiles dans les automatismes électroniques.

A titre anecdotique, le OU-exclusif (souvent appelé XOR) se notera simplement ≠ car, comme chacun peut le constater, ce symbole effectue bien un OU-Exclusif :

```
      0 0 1 1 ≠ 0 1 0 1
0 1 1 0
```

Dernière particularité : ces vecteurs binaires peuvent être utilisés comme tels pour des usages que nous venons de découvrir, mais aussi pour des usages nouveaux et bien utiles : le dénombrement et la sélection.

Dénombrement

Ayant trouvé qui a un salaire inférieur à 2500 euros au moyen de l'expression suivante :

```
      Salaires<2500
0 1 0 1 1 1 0 0 0 0 0 0 1 1 0 0 1 0 1 0
```

il est tentant d'additionner tous ces 1 et ces 0 pour connaitre au final le nombre de personnes qui gagnent moins de 2500 euros :

```
      +/Salaires<2500
8
```

Sélection

On peut aussi se servir d'un vecteur binaire comme d'un "*masque*" qui servira à ne garder, dans une autre donnée, que les termes qui correspondent aux "1" du binaire :

```
      1 1 0 1 0 0 1/23 55 17 46 81 82 83
23 55 46 83
```

L'opération fonctionne à l'identique sur des données textuelles :

```
      1 1 0 1 0 0 1/'Bernard'
Bend
```

Cette opération, appelée ***Compression***, est spécialement utile pour extraire d'une variable les éléments qui correspondent à un critère donné. Par exemple, pour connaître les salaires des gens qui sont en Catégorie 2, on écrira :

```
      (Catégories=2)/Salaires
2240 2076 2735 3278 1339 3319
```

Puissant, n'est-ce pas ?

Découverte

Pour exercer notre science toute neuve, cherchons dans notre variable `Val` les emplacements des nombres supérieurs à 35.
Voici les étapes de notre démarche :

```
Val       vaut    22 37 41 19 54 11 34
Val>35    vaut    0  1  1  0  1  0  0
⍴Val      vaut    7
⍳⍴Val     vaut    1  2  3  4  5  6  7
```

Mettons en correspondance ces deux informations : `Val>35` ⇨ `0 1 1 0 1 0 0`
`⍳⍴Val` ⇨ `1 2 3 4 5 6 7`

On voit que si on élimine (par une compression) les termes qui correspondent aux zéros pour ne garder que ceux qui correspondent aux 1, on obtient bien les positions recherchées : `2 3 5`.

Le travail demandé sera donc effectué de la manière suivante :

```
      (Val>35)/⍳⍴Val
2 3 5
```

Cette expression sert dans des quantités de situations différentes.

Voici un usage très similaire, mais qui porte sur une donnée textuelle : nous allons chercher les emplacements des "a" d'une phrase ; la technique est la même.

```
      Phrase←'Le tango argentin ne passera pas de mode'
      (Phrase='a')/⍳⍴Phrase
5 10 23 28 31                                  (vous pouvez vérifier !)
```

11. Cachez ce signe que je ne saurais voir

Très fiers d'avoir pu trouver tous les "a", nous voudrions trouver toutes les voyelles.

Hélas, alors qu'on peut écrire `(Phrase='a')`, parce qu'on compare un vecteur à une valeur isolée, on ne peut pas écrire `(Phrase='aeiouy')`, parce que cela reviendrait à comparer terme à terme une phrase, qui contient 40 lettres, et "aeiouy" qui n'en comporte que 6.
Or on peut comparer 40 lettres à 40 autres lettres, ou les comparer à une seule, mais pas à 6.

Nous aurons donc recours à une nouvelle fonction : l'***appartenance***, qui se note `∈` ainsi que le veut l'usage en mathématiques.

L'expression `A∈B` indique par une réponse binaire quels sont les éléments de la variable `A` qui apparaissent (où que ce soit) dans la variable `B`. Et cela fonctionne quelles que soient les formes, dimensions, et natures des données `A` et `B` : une petite merveille !

Par exemple :

```
      5 7 2 8 4 9 ∈ 3 4 5 6
1 0 0 0 1 0
      'pissenlit'∈'jardin'
0 1 0 0 0 1 0 1 0                  (seuls deux "i" et le "n" figurent dans "jardin")
```

Pour les besoins de notre recherche, nous écrirons donc :

```
      (Phrase∈'aeiouy')/⍳⍴Phrase
2 5 8 10 13 16 20 23 26 28 31 35 38 40
```

On peut même utiliser l'appartenance entre un vecteur et un tableau (une ***matrice***), comme visible ci-après (la liste de villes est une variable qu'on suppose avoir introduite auparavant).

```
      Villes
Martigues
Paris
Strasbourg
Granville
Nantes
Fréjus
```

```
      Villes ∈ 'aeiouy'
0 1 0 0 1 0 1 1 0 0 0
0 1 0 1 0 0 0 0 0 0 0
0 0 0 1 0 0 1 1 0 0 0
0 0 1 0 0 1 0 0 1 0 0
0 1 0 0 1 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0
```

Attention : le résultat a toujours la même forme que la donnée de gauche :

```
      'aeiouy' ∈ Villes
1 1 1 1 1 0                    Aucune ville ne comporte de "y"
```

Mise au point

Des informaticiens m'ont souvent fait remarquer que ce symbole ∈ constituait à lui tout seul la preuve irréfutable qu'APL est un langage hautement mathématique, qu'on ne peut mettre entre toutes les mains. Je vous propose d'y réfléchir ensemble, voulez-vous ?

Même s'il n'est employé dans l'enseignement que depuis le début des années 60, ce symbole est apparu (si je ne m'abuse) avec l'apparition des mathématiques dites "modernes", pendant la seconde moitié du 19ème siècle. Sachant que nous sommes au 21ème siècle, disons qu'il a **au moins** 100 ans.
D'autre part, quand mon fils ainé avait 11 ans, l'appartenance était enseignée en classe de **CM2**. Je peux vous dire, pour avoir suivi de près sa scolarité, que ça ne posait aucun problème (même si depuis cet enseignement a été repoussé à plus tard).

Autrement dit, ces doctes informaticiens qui trouvent l'appartenance trop compliquée pour leur entendement, clament en fait haut et fort qu'ils ont plus de 100 ans de retard sur le Q.I. d'un gamin de 11 ans.
Dans ces conditions, en effet, je crois qu'on ne peut pas mettre APL entre toutes les mains ; en tout cas pas entre ces mains là ! Mais est-ce bien APL qu'il faut critiquer ?

12. Une fonction à tout faire

Nous avons cherché la position de lettres ou de nombres dans des vecteur par une méthode très utile, mais qui présente de petits défauts que nous n'exposerons pas ici. Il existe une autre méthode, qui fait appel à la forme dyadique du symbole ⍳ .

```
      Vec ← 15 40 63 18 27 40 33 29 40 88    Vecteur dans lequel on cherche
      Vec ⍳ 29 63 40 33 50                   Valeurs cherchées
8 3 2 7 11                                   Positions trouvées
```

Il est bien vrai que 29, 63, 40 et 33, apparaissent respectivement en positions 8, 3, 2 et 7

Première surprise : la valeur 40 est présente 3 fois dans `Vec`, mais seule sa première apparition est mentionnée. C'est le prix à payer pour qu'à chaque valeur cherchée on puisse faire correspondre sa position ; comment ferait-on si, cherchant 5 nombres, on obtenait 7 réponses ?

Seconde surprise, la valeur 50 se voit attribuer la position 11 ... dans un vecteur qui ne comporte que 10 termes ! C'est de cette manière que la fonction ***Occurrence*** (le `⍳` dyadique) signale les valeurs absentes.
Au premier abord, cela semble étrange ; en réalité, c'est une caractéristique qui va faire de cette fonction un symbole à tout faire.

Un exemple

Un constructeur automobile décide de proposer à ses clients des réductions par rapport au prix catalogue (on voit bien à quel point l'exemple est imaginaire !)

Le taux de ces réductions dépendra du département, conformément à la table suivante :

Département	Réduction
17 50 59 84 89	9 % 8 % 6 % 5 % 4 %
Autres	2 %

Le problème consiste à calculer le taux de réduction auquel peut prétendre un client potentiel qui habite dans un département `D` ; par exemple `D←84`.

Commençons par créer deux variables :

```
DEP   ← 17 50 59 84 89
REDUC←  9  8  6  5  4  2
```

Cherchons où se trouve le 84 dans la liste des départements favorisés:

```
      DEP⍳D
4
```
⇦ 84 est bien le 4ème département de la liste

Cherchons à présent le taux de réduction correspondant par indiçage :

```
      REDUC[4]
5
```
⇦ Ce client a droit à 5% de réduction ; c'est bien cela

On aurait pu écrire directement : `REDUC[DEP⍳D]`

Si un client habite un département quelconque (75, 45, ou 93), l'expression `DEP⍳D` donnera dans tous les cas la réponse 6, et `REDUC[6]` ira toujours chercher le taux 2%, ce qui est conforme à notre attente.

Tout l'intérêt de cette approche est qu'elle est ***vectorielle***. Supposons en effet que cette annonce publicitaire attire les foules, et que `D` ne soit plus un scalaire mais un vecteur, la solution reste valable :

```
      D←24 75 89 60 92 50 51 50 84 66 17 89
      REDUC[DEP⍳D]
2 2 4 2 2 8 2 8 5 2 9 4
```

Tout cela sans programme, ni "boucle" ni "test" ; le lecteur qui a la connaissance d'autres langages de programmation pourra faire la comparaison sans peine.

Généralisons

En réalité, l'expression que nous venons de frapper exprime un algorithme de "changement de référentiel". Pas de panique, le terme est pompeux, mais le concept est simple : on est passé d'une liste de numéros de départements (ensemble initial) à une liste de taux de réduction (ensemble final).

Imaginons que l'ensemble initial soit un alphabet fait de minuscules et de majuscules, et que l'ensemble final ne soit composé que de majuscules :

```
      Almin
abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ'

      Almaj
ABCDEFGHIJKLMNOPQRSTUVWXYZ ABCDEFGHIJKLMNOPQRSTUVWXYZ*

     Fable←'Le Petit Chaperon-Rouge a bouffé le Loup'
```

L'expression vue plus haut produit la conversion minuscules ⇨ majuscules.

```
     Almaj[Almin ⍳ Fable]
LE PETIT CHAPERON*ROUGE A BOUFF* LE LOUP
```

Comme on pouvait s'y attendre, les caractères - et é, qui étaient absents de l'alphabet initial ont été remplacés par le * de l'alphabet final, mais la conversion a bien fonctionné. On peut améliorer très sensiblement cette solution.

Une fois encore, la démarche intellectuelle qui sera mise en œuvre pour résoudre un problème informatique sera totalement différente des voies traditionnellement enseignées, et le développeur y gagnera beaucoup en hauteur de vue.

13. Après le fond, la forme

Les langages informatiques traditionnels ne savent pas traiter les tableaux de nombres. Ils savent bien les mémoriser, mais au moment de les traiter, ils ne peuvent traiter que nombre après nombre. Rien de bien étonnant, dans ces conditions, à ce que ces langages ne s'intéressent pas aux modifications qu'il serait possible d'apporter à la forme des données.

C'est tout le contraire en APL, qui dispose de multiples outils pour travailler sur la forme des données ; nous n'en verrons que quelques uns.

Prend et Laisse

Les fonctions Prend (↑) et Laisse (↓) servent, comme leur nom le dit si bien, à prendre ou à laisser tomber une partie de variable. Nous ne montrerons ici que des exemples portant sur des vecteurs, mais toutes les autres formes de données peuvent être traitées de manière similaire.

Rappelons que `Vec` vaut `15 40 63 18 27 40 33 29 40 88`

```
4 ↑ Vec
15 40 63 18                    On a bien pris les 4 premiers éléments du vecteur

      5 ↓ Vec
40 33 29 40 88                 On a laissé tomber les 5 premiers éléments
```

Si l'opérande gauche est négatif, ces mêmes opérations touchent cette fois la queue du vecteur :

```
      ¯3 ↑ Vec
29 40 88                       On a pris les 3 derniers éléments du vecteur
```

```
      ¯7 ↓ Vec
15 40 63
```

Si on laisse tomber les 7 derniers éléments, il ne reste bien entendu que les 3 premiers, ce que donnerait `3↑Vec`
Les deux fonctions sont donc redondantes.

A quoi cela peut-il servir ?

Imaginons une entreprise dont on a noté, au fil des 12 années écoulées, la fulgurante (?) croissance du chiffre d'affaires en millions d'euros (d'où le nom `Came` donné à la variable) :

```
      Came
56 59 67 64 60 61 68 73 78 75 81 84
```

On veut calculer les écarts constatés d'une année sur l'autre ; comment faire ?

`1↓Came` donne `59 67 64 60 61 68 73 78 75 81 84`
`¯1↓Came` donne `56 59 67 64 60 61 68 73 78 75 81`

On voit qu'il ne reste qu'à soustraire terme à terme :

```
      (1↓Came)-(¯1↓Came)
3 8 ¯3 ¯4 1 7 5 5 ¯3 6 3
```

Sans programme ni boucle ; tout cela est très léger !

Au lieu d'une soustraction, une division eût permis, avec quelques aménagements de bon sens, de calculer des taux de croissance plutôt que des écarts :

```
      100×((1↓Came)÷(¯1↓Came))-1
5.35  13.56  ¯4.48  ¯6.25  1.67  11.47  7.35  6.85  ¯3.85  8  3.70
```

Miroirs et transpositions

APL est également doté de fonctions qui permettent de retourner une donnée en la faisant pivoter autour d'un axe de symétrie bien visible dans la forme même du symbole. Cela s'applique aussi bien à des données numériques que textuelles ; nous en ferons la découverte en appliquant ces fonctions à la variable `Villes` déjà rencontrée.

Variable initiale	Renversement Gauche-Droite (Miroir)	Renversement Haut-Bas (Miroir)	Permutation Lignes-Colonnes (Transposition)
`Villes`	`⌽Villes`	`⊖Villes`	`⍉Villes`

`Martigues`
`Paris`
`Strasbourg`
`Granville`
`Nantes`
`Fréjus`
	`seugitraM`
	`siraP`
	`gruobsartS`
	`ellivnarG`
	`setnaN`
	`sujérF`
		`Fréjus`
		`Nantes`
		`Granville`
		`Strasbourg`
		`Paris`
		`Martigues`
			`MPSGNF`
			`aatrar`
			`rrrané`
			`tiantj`
			`issveu`
			`g biss`
			`u ol`
			`e ul`
			`s re`
			`g`

Les symboles retenus sont particulièrement parlants, et ne demandent aucun effort de mémorisation. Ils peuvent aussi être utilisés en mode dyadique, avec des effets différents, mais tout aussi intéressants.

14. Retour à l'école primaire

Revenons à l'époque où nous apprenions nos tables de multiplication. Je me souviens qu'en ces temps reculés, proches du paléolithique, mon instituteur, pour s'assurer que nous connaissions bien toutes nos tables, nous faisait calculer le tableau de toutes les multiplications possibles des entiers de 1 à 9 :

×	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
etc.	etc.								

Vous voyez, je n'ai pas oublié ! Ce travail, cet enchaînement d'opérations, peut-être l'avez-vous fait, comme moi. Et puis nous nous sommes dépêchés d'oublier ce pensum, passant ainsi à côté d'un outil très puissant, qu'APL met à notre disposition sous le nom de ***Produit externe***.

Si on analyse bien notre travail, il consiste à combiner deux à deux les éléments des deux vecteurs représentés en rouge, au moyen de l'opération marquée en haut à gauche. Voyons ce que cela donnerait en changeant un peu les valeurs :

×	8	5	15	9	11	40
5	40	25	75	45	55	200
4	32	20	60	36	44	160
10	80	50	150	90	110	400
3	24	15	45	27	33	120

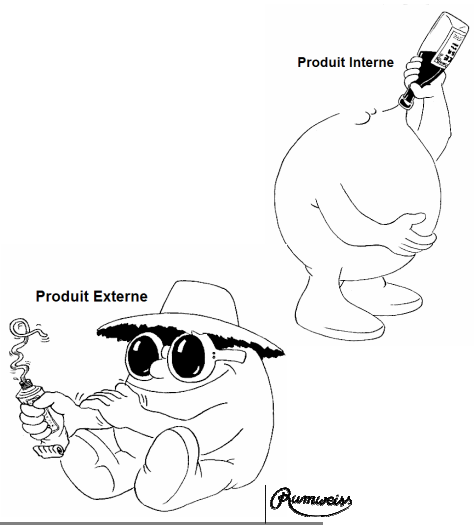
Cette opération se note comme ceci en APL :

```
      5 4 10 3∘.×8 5 15 9 11 40
40 25  75 45  55 200
32 20  60 36  44 160
80 50 150 90 110 400
24 15  45 27  33 120
```

Imaginez maintenant que vous pouvez remplacer le signe *Multiplication* par de nombreuses autres fonctions ou programmes que vous auriez définis vous-même, et vous comprendrez que, comme la *Réduction*, déjà rencontrée, le *produit externe* est un ***opérateur*** d'une incroyable puissance.

Commençons par quelques amusettes :

(⍳5)∘.=(⍳5)	(⍳5)∘.<(⍳5)	(⍳5)∘.≥⍳5)
1 0 0 0 0	0 1 1 1 1	1 0 0 0 0
0 1 0 0 0	0 0 1 1 1	1 1 0 0 0
0 0 1 0 0	0 0 0 1 1	1 1 1 0 0
0 0 0 1 0	0 0 0 0 1	1 1 1 1 0
0 0 0 0 1	0 0 0 0 0	1 1 1 1 1



Et passons aux applications pratiques

Un exemple

Supposons que le vecteur *Ages* contienne les âges des 400 personnes consultées lors d'un sondage. Nous aimerions comptabiliser combien il y a de personnes dans chacune des tranches suivantes.

0 - 25 - 30 - 35 - 45 - 50 - 55 - 65 et plus

On décide en outre que les personnes qui sont juste sur une limite de tranche seront comptabilisées dans la tranche inférieure.
Voici un aperçu des données :

```
Ages     ⇨   32 19 50 33 23 65 46 26 31 58 51 23 51 36 28 42 ... etc
Tranches ⇨   0 25 30 35 45 50 55 65
```

Nous allons réaliser le produit externe *Tranches ∘.< Ages* , et voici les premiers éléments du calcul présentés comme dans les pages précédentes :

<	32	19	50	33	23	65	46	26	31	58	51	23	51	36	28	42	34	... etc
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
25	1	0	1	1	0	1	1	0	1	1	1	0	1	1	1	1	1	
30	1	0	1	1	0	1	1	0	1	1	1	0	1	1	0	1	1	
35	0	0	1	0	0	1	1	0	0	1	1	0	1	1	0	1	0	
45	0	0	1	0	0	1	1	0	0	1	1	0	1	0	0	0	0	... etc
etc.																		

Si on totalise ce tableau binaire, on obtient, pour chaque ligne le nombre de personnes qui on plus de 0 années, plus de 25 ans, plus de 30 ans, etc. Procédons à ce calcul :

```
cum ← +/Tranches∘.<Ages
```

Dans l'exemple très réduit figuré ci-dessus, *cum* vaudrait : `17 14 12 8 6 4`

Autrement dit, 12 personnes ont plus de 30 ans. Mais parmi elles, on a compté les 8 qui ont plus de 35 ans. Pour savoir combien de personnes ont entre 30 et 35 ans, il faut faire `12-8`, qui donne 4.
Si on veut répéter ce calcul pour toutes les tranches, il faut procéder à la série de soustractions que voici :

```
          17 14 12 8 6 4      Ce vecteur, c'est cum
moins     14 12  8 6 4 0      Celui-ci, c'est cum amputé d'un élément, et suivi d'un zéro
donne      3  2  4 2 2 4      Autrement dit on a calculé cum-1↓cum,0
```

Bien que nous ne l'ayons pas encore dit, nous découvrons que la virgule permet d'accoler des valeurs les unes aux autres, c'est une opération appelée ***Concaténation***.

Si on travaille non plus sur ce petit échantillon de données, mais sur les 400 personnes, voici ce qu'on obtient :

```
      cum←+/Tranches∘.<Ages
      cum-1↓cum,0
83 65 79 79 32 17 36 9
```

Tout cela sans véritable programmation, et ça marche quels que soient le nombre de personnes et le nombre de tranches ... le bonheur, je vous dis !

Le produit externe permet de trouver des solutions atypiques à des problèmes très classiques.

15. Je ne vous dirai pas tout

Au fil de ces quelques pages, nous n'avons fait que survoler les bases d'APL, et effleurer certaines idées fortes qui expliquent l'attrait de ce langage. Il nous resterait mille choses à voir !

Il faudrait parler du ***produit interne***, qui généralise très largement le produit matriciel, dont nos étudiants ne retiennent que des formules apprises par cœur, comme une litanie dont on n'a jamais bien compris le sens, à grands renforts de "*sigma de* A*ij* B*jk*". Mais concrètement, dans les problèmes de la vie courante, à QUOI sert ce bourrage de crâne ?
Pour avoir utilisé APL comme outil d'enseignement, je peux vous assurer qu'on peut enseigner l'algèbre linéaire de manière rapide et concrète, et faire découvrir aux étudiants qu'on peut s'en servir pour comparer les paniers de deux ménagères. Après cela, ils ne l'oublient plus !

Il faudrait parler des *tableaux généralisés*, de la fonction *Execute*, il faudrait ... il faudrait environ 400 pages ... ce n'est pas notre but aujourd'hui.

Laissez-moi encore vous donner un rapide éclairage sur l'usage très atypique des fichiers de données en mode "*inversé*".

16. Structurer des données

On a bien compris, au fil des pages précédentes, que la seule manière pertinente d'organiser les données en APL, c'est de les grouper par nature : une variable contiendra des noms, une autre des salaires, une autre encore des matricules, etc ...

Or, généralement, ce n'est **pas du tout** ainsi que l'information est organisée dans les fichiers de texte traditionnels. Si on prend l'exemple d'un fichier de personnel, sur chaque ligne de texte on trouve, à la queue-leu-leu toutes les informations qui concernent un même individu : nom, prénom, matricule, salaire, etc. Cela ressemble un peu à ceci :

```
Sabatier      Eugene     1 1933  2997 E D 4 2737 93 C
Depond        Alain      1 1943  1732 E C 0 1489 77 C
Laure         Rose       2 1967  3813 E D 0 2082 75 C
Japroutsy     Véronique  2 1962  3115 E M 3 1934 77 U
Perdoux       Véronique  2 1961  1685 M D 0 2559 94 U
Trinque       Kate       2 1968  1747 E C 0 2902 92 P
Foucault      Jean       1 1934  2962 M M 3 1641 94 U
Fossey        Nicole     2 1961  2370 E C 0 1640 94 A
Boudinoy      Juliette   2 1945  2705 E M 4 1131 75 U
Louvier       Laurence   2 1932  1972 E M 2 2228 93 U
```

Dans des données ainsi organisées, les informations chiffrées (salaire, année de naissance, ...) sont enregistrées sous forme textuelle : on ne peut faire de calculs qu'après conversion en nombres. Les traiter sous cette forme pénaliserait gravement APL.
Le fichier de l'exemple ci-dessus contenait 11 renseignements pour 1000 personnes ; le fichier avait donc 1000 "*enregistrements*".

En APL on a recours à la méthode suivante : on découpe le contenu du fichier en 11 "bandes" verticales ne contenant chacune qu'une seule nature d'information (nom, prénom, etc.), on convertit les données chiffrées en nombres véritables, puis on enregistre chacune des informations sous forme de 11 enregistrements dans un nouveau fichier.

On passe ainsi de 1000 fois 11 informations disparates à 11 fois 1000 informations homogènes.
Pour cette raison, on dit que ce nouveau fichier est un ***Fichier inversé*** (on parle aussi parfois de *fichier-vecteur*).

Dans la pratique, les choses sont un peu plus complexes. Quand le nombre de personnes devient très grand (par exemple 500.000), il n'est pas conseillé de stocker 500.000 valeurs dans un seul enregistrement. On segmente alors chaque information et on range, par exemple, les salaires dans 50 enregistrements de 10.000 salaires, puis les années de naissance dans 50 enregistrements de 10.000 années, etc. Le fichier obtenu ressemble à une sorte de mille-feuilles, mais le principe reste le même.

Comment s'en servir ?

Si on avait de simples petites variables, il serait aisé de les traiter comme vu dans les pages précédentes. Je vous montrerai comment faire afficher tout ou partie de ces données en écrivant de petites fonctions qui donnent une souplesse d'interrogation incomparable.

Par exemple, pour extraire les personnes dont le salaire (variable *SAL*) est compris entre 1800 et 3500 euros et dont la situation familiale (variable *SIF*) est 'M', on pourrait écrire :

```
Edite Gens (SAL Entre 1800 3500) Et (SIF = 'M')
```

On a utilisé les couleurs suivantes :

Rouge	Fonctions (programmes)
Bleu	Variables
Noir	Autres

Le résultat pourrait avoir la forme suivante :

Prénom	Nom	Sexe	A.Naiss	Salaire	Statut	SiF	Dept
Véronique	Japroutsy	2	1962	3115	E	M	77
Jean	Foucault	1	1934	2962	M	M	94
Juliette	Boudinoy	2	1945	2705	E	M	75
Laurence	Louvier	2	1932	1972	E	M	93

Grâce à de petits programmes comme *Edite*, *Gens*, *Entre*, *Et*, mais aussi : *Ou*, *Sauf*, *Selon*, *Tous*, *Déciles*, on peut effectuer très facilement des interrogations sur les données. On peut, bien entendu, enrichir à volonté ce vocabulaire.

Mais, me direz-vous, ce n'est pas un bien grand prodige à traiter avec aisance des variables relatives à 100 ou 200 personnes. Qu'adviendra-t-il si on doit traiter 10.000, 100.000 personnes, ou bien plus encore ?
C'est ici que les fichiers inversés prennent tout leur sens.

En effet, on peut effacer les petites variables (*NOM*, *SEX*, *MAT*, *SAL*, *SIF*, *ENF*, etc), puis créer (au moyen d'un générateur automatique) autant de minuscules programmes d'une seule instruction chacun, qui vont lire sur le fichier inversé l'information correspondante.

Autrement dit, jusqu'ici, le fait de frapper SAL faisait afficher le contenu de la variable SAL, c'est-à-dire quelques dizaines de salaires. A présent quand on frappe SAL, on exécute un programme qui lit le fichier inversé, et rapatrie quelques milliers ou dizaines de milliers de salaires.

L'utilisateur n'est pas bousculé dans ses habitudes, et peut continue à utiliser sa batterie de petits programmes d'interrogation. Il peut continuer à l'enrichir : un programme qui "marche" sur une variable de 10 ou 20 valeurs saura travailler identiquement sur 10.000 ou 100.000 valeurs.

Vous ai-je déjà dit qu'APL c'est vraiment magique ?

A présent, comme promis, je réponds à certaines des questions que l'on me pose le plus souvent.

17. FAQ

Je voudrais conclure en essayant d'apporter ma réponse à certaines questions qu'on m'a posées cent fois. Je dis bien MA réponse : je ne prétends détenir aucune vérité.

APL est-il un outil professionnel ?

Je ne parlerai que de trois exemples que mon associée ou moi-même avons réalisés :

- Le plan à long terme du groupe TOTAL, pendant 12 ans, au plus près de la Direction.
- La gestion des approvisionnements des chaînes de montage des 6 principales usines du groupe Renault du jour J+2 au mois M+3.
- La comptabilité "Réassurance" du groupe Allianz-AGF

Ces trois applications avaient en commun des caractéristiques qui les placent au niveau de grandes applications de niveau industriel :

- Elles étaient particulièrement sensibles, car elles étaient porteuses d'enjeux financiers considérables.
- Elles devaient avoir une fiabilité absolue. On ne peut pas se permettre d'arrêter une usine comme Flins ou Sandouville sous prétexte de bug informatique.
- Les deux premières étaient extrêmement mouvantes : les besoins évoluant en permanence, les programmes étaient en constante mutation.

Alors, la réponse est : oui, APL permet de réaliser, pour un coût humain très raisonnable, des applications sensibles, de grande envergure, avec le meilleur niveau de qualité et de fiabilité.

Quelle est le créneau d'APL aujourd'hui ?

Le créneau d'APL, ce sont toutes les applications ***très urgentes*** ou ***très mouvantes*** ; ce sont d'ailleurs généralement les mêmes.

Les équipes de développement traditionnelles ne travaillent que sur des cahiers des charges qui demandent parfois six mois de réflexion, à la suite de quoi les développements et les tests sont également très longs. Il faut un temps considérable pour avoir ce qu'on voulait ... et parfois on ne l'a même pas !
Que les besoins viennent à changer inopinément, et on repart pour d'interminables mois de travail et d'adaptations.

Hélas il est des problèmes qui ne peuvent pas attendre. Certains événements imprévus de la vie durent à peine deux mois, comme ce fut le cas de la première guerre du Golfe, c'est-à-dire bien moins longtemps qu'il n'en faut aux informaticiens pour adapter leur programmes à ces circonstances imprévues !
Cette grande mouvance ou cette grande urgence sont le vrai fonds de commerce d'APL. Car en APL, on peut développer au contact direct du demandeur, et l'impliquer dès le début dans le recadrage permanent de la pertinence de ce que fait le développeur. Par la suite, quand il faut procéder à des évolutions, c'est encore la rapidité de développement qui fait d'APL un outil tout spécialement adapté aux situations mouvantes.

Ce langage est-il lisible ?

Si APL était un langage hermétique, complexe, il n'aurait séduit que les petits génies de l'informatique, les surdoués à BAC+20, les fondus du bit et de l'octet.

Or, c'est un paradoxe, les informaticiens dans leur grande majorité n'ont jamais vraiment compris APL. Ceux qui l'ont utilisé avec bonheur n'avaient bien souvent aucune culture informatique, ou une connaissance assez légère ... et ils ont le plus souvent appris APL tout seuls. C'est dire à quel point le langage est porteur pour qui accepte de s'y plonger sans *a priori*.

Croire qu'une programmation en "*langage clair*" serait plus lisible relève soit de l'utopie, soit de la malhonnêteté intellectuelle. Car si je dis "*une fonction linéaire d'une variable est égale à la somme d'une constante et du produit de la grandeur variable par une seconde constante*.", c'est incontestablement du français, mais c'est parfaitement obscur, voire incompréhensible !

Mais si maintenant de dis $y=ax+b$ (notation pourtant ***abstraite*** et ***symbolique***), je sais être compris de la majorité de mes interlocuteurs, avec lesquels je partage une éducation commune. Il faut se rendre à l'évidence : ***tout est affaire de formation initiale***.

Les 80 lignes de C++ (ou de JAVA, ou de je ne sais quoi) que remplacent souvent 5 ou 6 lignes d'APL, sembleraient parfaitement obscures à quiconque n'a jamais étudié C++. Il faut comparer des choses comparables et cesser de juger APL à la lueur des opinions de personnes qui n'ont pas accepté de l'apprendre.
Précisons ce point. Si on fait lire à n'importe qui un poème de Pouchkine, accepterait-on du lecteur qu'il déclare que cette poésie est mauvaise sous prétexte qu'il ne sait pas déchiffrer le Russe ? Non, bien entendu ! C'est pourtant ce qu'on fait quand on demande à des informaticiens non formés à APL de porter un jugement sur la lisibilité de programmes écrits en APL. Forts de leur statut de professionnels, ils peuvent affirmer que ces programmes sont illisibles, ... et on les croit !

Convaincre ? : une mission impossible !

Pour être honnête, je dois convenir qu'APL possède un certain nombre de signes nouveaux, qui rendent son décryptage impossible par une personne **non formée**. Comment voudrait-on qu'un informaticien rompu à C++ ou PASCAL puisse comprendre une expression telle que :

```
R ← ((V⍳V)=⍳⍴V)/V
```

Et qui pourrait me croire quand je dis que cette expression ne nécessite pourtant **aucune** "lecture" ni **aucun** "décryptage" pour un APListe : elle se lit et se comprend instantanément, **en bloc**, tout comme le mot "PAPA" s'impose à notre esprit sans que nous ayons à le lire ou à le décrypter lettre à lettre comme le ferait un petit enfant.
Bien sûr, pour comprendre "PAPA", il faut avoir appris à lire ; il en va de même pour APL, il faut l'apprendre. Après tout, on apprend bien C++ ou PASCAL, pourquoi pas APL ?

En raison même de cette apparence cryptée, il est presque impossible de convaincre qui que ce soit de l'intérêt et de la beauté d'APL juste en lui montrant (comme j'ai pourtant essayé de le faire ici) quelques subtilités, quelques beaux algorithmes.

Ne cherchez à convaincre personne en montrant que vous faites en 10 symboles ce qui lui demanderait 100 instructions filandreuses : tout le monde préfère lire 100 lignes de bon (ou même de mauvais) Français que de rester muet devant 10 idéogrammes chinois !
Vous ne convaincrez que des personnes qui accepteront de s'investir et de se former.

Comment se former ?

N'importe qui peut, sans aucune formation, frapper 2+2 sur un clavier APL, et obtenir la réponse 4. Il ne faut pas en déduire pour autant, comme l'ont fait trop de laudateurs d'APL, que trois jours suffisent pour apprendre et pratiquer ce langage.

Outre la connaissance des éléments de base du langage, un usage correct d'APL suppose l'acquisition de méthodes d'organisation des données, de méthodes particulières de résolution des problèmes, qui sont spécifiques à APL. Cela ne s'apprend pas à la sauvette, ni en APL, ni en aucun autre langage.

Il faut consacrer à APL le même temps qu'on consacrerait à tout autre langage (2 à 3 semaines), et accepter de travailler au contact de professionnels, capables d'enseigner de bonnes pratiques.

18. Votre temps est venu

Les diplodocus, on le sait, sont condamnés à disparaître !

Tous, nous avons cru et croyons encore à APL ; aujourd'hui nous aimerions voir se manifester une relève jeune et dynamique, capable de trouver de nouveaux débouchés à APL, et de lui redonner crédibilité et légitimité.

Ce défi vous est lancé ; **votre temps est venu.**

Merci de m'avoir accompagné jusqu'ici.


Bon, moi je retourne
à mon C++