Courrier des Lecteurs

M. Gilles Hervy

à propos des nombres premiers

Intéressé depuis longtemps par les nombres premiers, je n'ai pas pu résister à l'envie d'écrire en APL PLUS III un des algorithmes proposés par Gérard LANGLET dans le numéro 16 des Nouvelles d'APL (fonction PREMS page 44 de son article « Du bon usage de la simplicité »). Le REPEAT-UNTIL de ma fonction remplace la récursivité pour chaîner, par devant, les nombres premiers de 1 à racine de n.

Je l'ai fait surtout pour bien comprendre moi-même l'algorithme (je n'avais pas de doute sur le résultat ...), et pour me faire une idée précise du temps d'exécution sur ma machine favorite (un Pentium 75 avec 40 Moctets de mémoire vive).

L'utilisation de "WHERE" en ligne [8] permet surtout d'aller plus loin car il n'y a pas à créer le vecteur **⍳n** , c'est aussi pour préserver de l'espace mémoire qu'il y a une réaffectation de la variable "p" sur cette même ligne, rendue efficace en rejetant **p←~p** hors de cette boucle. Pour ceux qui n'auraient pas la fonction WHERE, mettre un commentaire en ligne [8], et enlever celui de la ligne [7] (WHERE se trouve dans l'espace de travail ASMFNS fourni avec APL PLUS III).

```
   ∇ r←Premier n;i;p;⎕io
[1]  ⍝ Nombres premiers inférieurs ou égaux à n
[2]  r←⍴⎕io←1
[3]  :REPEAT
[4]   p←n⍴0 ◇ i←¯2 ◇ p[1]←1
[5]   :WHILE n≥i×i ◇ p←p∨n⍴i↑1 ◇ i←-p⍳0 ◇ :ENDWHILE
[6]   p←~p
[7]  ⍝:UNTIL 1≥n←¯1+1↑r←(p←p/⍳n),r
[8]  :UNTIL 1≥n←¯1+1↑r←(p←WHERE p),r
   ∇
```

C'est exactement le crible d'Eratosthène où, à chaque pas du REPEAT-UNTIL, on trouve la liste des nombres premiers entre racine de n et n (ce sont les nombres marqués par le vecteur binaire p en ligne[6]).

En utilisant la ligne [8], et avec un WSSIZE de 1Mo, on peut ainsi avoir :

```
      ⎕wa
992156
      ai←⎕ai ◇ R←Premier 1640000 ◇ 2⊃⎕ai-ai
11.15   ⍝ Le temps d'exécution en secondes
      ⍴R
123942
```

et avec la ligne [7] pour la même place libre :

```
      ⎕wa
992156
      ai←⎕ai ◇ R←Premier 220000 ◇ 2⊃⎕ai-ai
1.21
      ⍴R
19618
```

Et maintenant, juste pour le plaisir !!!...

```
      )clear 13000000
CLEAR WS

      )copy 60 prem Premier WHERE
SAVED 11/30/1995 17:07:12
```

```
      ai←⎕ai ◇ R←Premier 20000000 ◇ 2⊃⎕ai-ai
352.34

      ⍴R
1270607
```

On trouve les 1 270 607 nombres premiers inférieurs à vingt millions en moins de 6 minutes ! Tout tient dans `p←p⍱n⍴i↑1` en ligne [5] avec `i` négatif. Merci Gérard...

Courrier à propos de l'Editorial de S. Baron, `Les Nouvelles d'APL`", numéro 16, page 4.

b1) **De M. Claude Henriot, CH-Savièse**

Elucubrations sur un "Thème" binaire. (voir Note)

A première vue, cette fonction programmée `PP`, pourrait se résoudre par une compression. J'ai vite déchanté car la transformation `PP1[3]` n'a pas été facile à codifier.

Je présente cette solution annexe à titre informatif pour celui qui voudrait utiliser les fonctions de `BTR` (Binary to Reduce) et `RTB` (Reduce to Binary).

Ce problème de `PP` et la solution se trouvent dans VECTOR Volume 1, 1984.
- Prize Competition Vol.1.1, page 105-106 par David Ziemann
- Competition Result Vol.1.3, page 123-125
- et dans ma boîte d'utilitaires GPF1.

Je me suis donc tourné vers une solution de logique binaire qui, en résumé, est la suivante :

- Faire un vecteur binaire à zéro, avec seulement la première position de chaque séquence de 1 à 1, selon le 1er exemple :

```
donnée ←0 0 0 1 1 0 0 1 1 1 1 0 0 0 1 1 1 0 0 0 0 1 1 0
nouveau ←0 0 0 1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0
```

- puis expansion par l'opérateur d'addition, plus 1 :

```
1+ +\ nouveau
resultat←1 1 1 2 2 2 2 3 3 3 3 3 3 4 4 4 4 4 4 4 5 5 5
```

- enfin le modulo 2 donnera un vecteur binaire :

```
resultat←1 1 1 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 1 1 1
```

- le produit binaire avec la donnée correspond à `PP` :

```
solution=produit←0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 0
```

Vous allez sûrement me poser la question :
>> comment ai je calculé la séquence "`nouveau`" <<
Simplement en fouillant dans ma boîte d'utilitaires GPF1, pour y trouver une fonction donnant 20 algorithmes binaires. Je l'ai construite sur la base d'un article dans le même VECTOR Volume 1, 1984.
- Boolean Difference Operations par J. Barman.
- Vol. 1.1, Appendix, page 138-141
- Vol. 1.2, Technical Correspondance, N. Thomson.

Ma fonction donne tout de suite un exemple de la variante demandée, soit "First One at 1":

```
'FO1' ⍙BDO 1 0 1 0 1 0 1 0 1
      1 0 1 0 1 0 1 0 1
```

- De plus, elle crée une variable globale simulant une DDF d'I-APL :

```
⍙BDO
      FO1: B>¯1↓0,B        ⍝ à transcrire en PP2
```

```
   ∇ PP2[⎕]∇
[0]  R←PP2 BIN
[1] ⍝ EXERCICE DE THEME PROPOSE DANS LE N°16, PAGE 4.
[2]  R←BIN∧2|1++\BIN>¯1↓0,BIN      ⍝ 'FO1' ⍙BDO BIN
   ∇ 1995-10-24 09.10.47
```

Malheureusement, je ne suis pas satisfait par ce résultat qui n'est que partiel. Lorsque l'on crée une fonction, surtout si elle doit être utilitaire, donc générale, il faut penser à cette généralité.

Dans le problème posé de PP, il est bon de prévoir au moins les deux cas :
- paquets des 1 de rang pair, et de rang impair
- par extension, on peut prévoir tous les 3 paquets etc...

J'ai défini, arbitrairement, que l'entier scalaire en argument de gauche indique la variante :

- LA pair donne rang pair
- LA impair donne rang impair
- pas de LA donne 0

et voici la fonction programmée PPP :

```
   ∇ PPP[⎕]∇
[0]  R←EO PPP BIN
[1] ⍝ EXERCICE DE THEME PROPOSE DANS LE N°16, PAGE 4.
[2]  ⍎(2≠⎕NC 'EO')/'EO←0'
[3]  R←BIN∧2|(1+0⌈⌊EO)++\BIN>¯1↓0,BIN ⍝ 'FO1'⍙BDO BIN
   ∇ 1995-10-30 20.42.41
```

Note :

La notion de programme implique un programme principal (Main routine) et des sous-programmes (Subroutines). En APL il n'existe pas de programme car toute fonction programmée (même la première d'une application) peut être appelée par une autre fonction, qui peut elle-même être appelée.

Donc PP est une fonction programmée utilitaire (pour ne pas dire une sous-fonction).

Annexe :
Les GPF (General Purpose Functions) utilisées dans cette réponse, sont la propriété du soussigné, mais à la disposition des APListes intéressés.

```
   ∇ PP1[⎕]∇
[0]  R←PP1 BIN;LGT;RED;RHO
[1] ⍝ EXERCICE DE THEME PROPOSE DANS LE N°16, PAGE 4.
[2]  RHO←⍴RED←⍙BTR BIN
[3]  LGT←(0 1 3 5)[⎕IO+4|RHO]+6×⌊RHO÷4
[4]  R←⍙RTB(LGT⍴1 0 1 0 1 1)\RED
   ∇ 1995-10-24 08.58.03

   ∇ ⍙BTR[⎕]∇
[0]  R←⍙BTR B;D
[1] ⍝ BINARY TO REDUCED LENGTH VECTOR. (÷ ⍙RTB)
[2]  R←D-⎕IO,¯1↓D←D/⍳⍴D←(B,2)≠0,B←∈B
[3] ⍝ FROM VECTOR VOL.1
   ∇ 1992-04-25 21.17.38

   ∇ ⍙RTB[⎕]∇
[0]  B←⍙RTB R
[1] ⍝ BINARY FROM REDUCED VECTOR. (÷ ⍙BTR)
```

```
      [2]  B←R/(⍴R)⍴0 1
      [3] ⍝ VSAPL IS B←≠\(⍳+/R)∈⎕IO++\R
          ∇ 1992-04-26 17.08.48

          - (÷ ΔααΔ) signale la fonction inverse associée

          ∇ ΔBDO[⎕-1]∇
      [0]  Z←Y ΔBDO X;B;E;F;S
      [1] ⍝ BOOLEAN DIFFERENCE OPERATIONS. VECTOR VOL ⍝ Fera l'objet d'un
article séparé
          ∇ 1992-04-25 21.35.07
```

b1) De M. Gérard Langlet

La proposition de S. Baron tombe à pic pour que j'en profite pour compléter et illustrer les recommandations de mon article « *Du bon usage de la simplicité* » Les Nouvelles d'APL, N° 16, pp. 25-45, par la maxime suivante :

« Jamais un problème purement **binaire** en numérique ne traiteras...[1] »

Dans les implantations qui traitent le binaire en bits, le fait d'utiliser un seul vecteur entier intermédiaire multiplie l'espace nécessaire en mémoire au moins par 16 sinon par 32 ! En ces temps d'austérité, « Sus au Gaspi ! »

On peut même ajouter tout de suite qu'il convient de regarder si l'**idiome universel** (cher à notre cœur et fort économe de ressources) n'apporterait pas immédiatement **la** solution en binaire pur... sans tergiverser [2] :

```
      ∇ V←PP V
[1]     V←V∧~≠\V>0,¯1↓V
      ∇

      PP 0 0 0 1 1 0 0 1 1 1 1 0 0 0 1 1 1 0 0 0 0 1 1 0
0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 0

      PP 1 0 1 0 1 0 1 0 1
0 0 1 0 0 0 1 0 0
```

Inutile de gaspiller plus d'un nom dans la table des symboles (V suffit).

A noter que si la fonction > devenait indisponible, l'expression α∧~ω produit le même résultat que α>ω pour des arguments booléens.

Pour obtenir les paquets impairs, il suffit d'ôter dans la fonction la négation logique, (qu'entre parenthèses on peut aussi écrire 1≠ des fois que la touche ~ soit cassée).

Foin de glose ! C'est tout pour aujourd'hui.

c) de M. Claude Henriod, CH-Savièse

à propos de divers articles du Numéro 16.

I) ⍝ ETUDE DU "JUSTE" CALCUL DE LA "BETE".

⍝ APL2 Version PS/2 1.02 TIMING selon APL232.exe

Cette étude est basée sur l'article des "Nouvelles d'APL"
dans le N° 16 , présentée par Gérard A. Langlet, Page 51 & sq.

Je n'ai pas modifié l'interpréteur, mais créé, depuis plus de 10 ans, des fonctions programmées en APL qui s'exécutent "JUSTE" sur plusieurs systèmes:

- MPCALC et MP2CALC. Pour un article ultérieur.
  sur : APL68000/Wicat+Ampère, I-APL/PC, APL2/PS2/DOS

(impl. APL2/OS2 et APL*PLUS II non testées).

Pour le problème de Gérard, j'ai d'abord ajouté une fonction "FOIS" dans mon WS, laquelle se contente de multiplier en boucle. Puis, après quelques réflexions
nocturnes, j'ai rajouté une fonction MPOWER.
L'amélioration du « timing » est déjà significative.

CONCLUSIONS:
Faire confiance aux propositions du même Gérard A.L[3]. en page 45 du même n° 16 reçu récemment.

- En APL, éviter les boucles, si possible.
- Dépenser quelques temps de réflexion pour gagner des heures.

- Par contre, si MPOWER est beaucoup plus encombrant que FOIS, il permet de limiter le nombre de FOIS, l'encombrement étant dû à la préparation du calcul.

Ici, je n'ai pas cherché l'optimum, mais seulement essayé de démontrer comment on peut gagner du temps par la pensée.
>> La devise bien connue d'IBM est : "THINK" <<

```
      ⍝ EXECUTION PS/2 486/DX2 AVEC ⎕WA=12 MO, APL232.
      ⍝ LE TEMPS EFFECTIF N'EST PAS IMPORTANT,
      ⍝ C'EST LE RAPPORT 1/20 QUI COMPTE.

      bete←0 666    ⍝ 0  est la caractéristique du MP

B_FOIS ← 'BEBETE+bete FOIS bete'
M_POWR ← 'BETE+bete MPOWER bete'

      ⊃5 XTIME¨
  AVG CPU:  121.232  Sec. for  5   times of:  B_FOIS
  AVG CPU:  6.074    Sec. for  5   times of:  M_POWR

      ⍝ calcul en Multi-Précision. MPU_nit = 1E7
     ⊃5 ⍙TAK BETE
      0   27154 1759288 7128558 2608745   ⍝  5↑BETE
4202427 2504000 2868395 6305088  598016   ⍝ ¯5↑BETE[4]

       BEBETEBETE
1
       ⍝ C Q F D
```

PRINCIPE du calcul:

```
 ⍝ Je prends d'abord un cas simple : N⋆4
      - soit 3 multiplications : N × N × N × N
      - on peut en 2 multiplications : ×/ 2⍴ ×/ 2⍴ N
   Même exercice avec : N⋆5
      - soit 4 multiplications : N × N × N × N × N
      - je fais en 2 multiplications : ×/(1⍴N),2⍴×/2⍴N

La forme générale de cette équation à 4 niveaux sera :
      ( ×/(N⋆b4),( ×/(N⋆b3),( ×/(N⋆b2), . . .
            ( ×/(N⋆b1),( N)⋆a1 ) ⋆a2 )⋆a3 )⋆a4 )
            ⍝ -premier niveau- ⍝

Nous verrons plus loin que l'on a toujours :
      a1=1 et b1=0, d'où premier niveau = N
[algo 1]
      (×/(N⋆b4),(×/(N⋆b3),(×/(N⋆b2),(N)⋆a2)⋆a3)⋆a4)
```

Il faut maintenant trouver la valeur des termes en a et b, et le nombre de couples. A partir de l'exposant

P complet on fait une réduction:

```
      A ← ⌊ P⋆.5 ◇ B ← P-A⋆2
      puis P ← A , et l'on recommence.
```

Une fonction récursive est raisonnable dans ce cas.

```
[0]   Z←RSR2 P;R;S
[1]  ⍝ RECURSIVE LIST OF EXTENDED EXPONENT
[2]   R←S,P-(S←⌊P⋆÷2)⋆2
[3]   ⍎(1 0R)/'→0,Z←⊂R'
[4]   Z←(⊂R),RSR2 S
     ∇
```

Exemple :

```
      RSR2 86
       9 5  3 0  1 2  1 0  ⍝ Chaque ( a b )
En remplacement dans l'algo 1:      N ← 5
      ( ×/(N⋆5),( ×/(N⋆0),( ×/(N⋆2),( N )⋆1 )⋆3 )⋆9 )
1.292469707114106E60
      N⋆86
1.292469707114106E60
```

De même avec le nombre de la Bête:

```
      RSR2 666
 25 41  5 0  2 1  1 1  1 0
```

Et en décomposant encore :

```
      ⊃ RSR2¨25 41
 5 0  2 1  1 1  1 0
 6 5  2 2  1 1  1 0
```

Et en décomposant encore :

```
      ⊃ RSR2¨5 6
 2 1  1 1  1 0
 2 2  1 1  1 0
```

Et en décomposant encore :

```
      ⊃ RSR2¨2 3
 1 1  1 0
 1 2  1 0
```

Reprenons encore l'algo 1 avec exposant 86 :

Il faut se rappeler qu'en MultiPrécision le calcul de puissance se traduit par une boucle de multiplications successives : N FOIS P .

Le but sera donc de diminuer ces calculs en reprenant les résultats intermédiaires (jusqu'où ? A déterminer.)

Si, partant de l'algo 1, on prend puissance 86

```
( ×/(N⋆5),( ×/(N⋆0),( ×/(N⋆2),( N )⋆1 )⋆3 )⋆9 )
      (N⋆b2) donne (N⋆2) = (×/(N⋆1),(N)⋆1) ou (N×N)
      (N⋆2) en MultiPrécision est = N FOIS 2 ,
```

soit une multiplication N×N .

Donc pas nécessaire d'appliquer l'algo 1 pour P=2 .

Je vous laisse faire le même raisonnement avec les autres (N⋆P) .

Ici, une fonction récursive peut aussi être programmée, car nous voyons qu'il y a 4 niveaux pour P=86, et seulement 5 niveaux pour P=666.

Le lendemain, j'ai mis en pratique ce que je vous ai expliqué ci-dessus.

Première constatation si :

```
                                  Nombre de multiplications
      ⊃RSR2¨1 2 3 4       en boucle     avec réduction
1 0                          0               0
1 1  1 0                     1               1
1 2  1 0                     2               2
2 0  1 1  1 0                3               2
```

On voit que jusqu'à `P=3` il n'y a pas de différence. Par contre, à partir de `P=4`, il peut être judicieux de recalculer la réduction.

J'ai donc remplacé `MPOWER` par `MPLOWER` (terme combiné de Low et Power), d'où un temps relatif amélioré.

```
BETE1 + 'BET1+0 666 FOIS 0 666'
BETE2 ← 'BET2←0 666 MPOWER 0 666'
BETE3 ← 'BET3←(0 666) MPLOWER 666'
      ⊃10 XTIME¨BETE1 BETE2 BETE3
  AVG CPU:  123.005  Sec. for  10   times of:  BETE1
  AVG CPU:  6.404    Sec. for  10   times of:  BETE2
  AVG CPU:  2.631    Sec. for  10   times of:  BETE3

      BET1 BET2 ¨⊂BET3
1 1
      123 ÷ 6.4 2.6           ⍝ Rapport
19.21875 47.30769231
```

La fonction `XTIME` n'est pas nécessairement exacte (basée sur `⎕AI`), mais donne une évaluation raisonnable des ratios.

Commentaire de G. Langlet.

Pour élever un nombre (comme d'ailleurs une matrice) à une puisance entière, il suffit, sans tergiverser, d'exprimer l'exposant en binaire (qui en eût douté ?) :

```
      2⊥1 0 1 0 0 1 1 0 1 0
666
```

Il va suffire de 4 multiplications des puissances 2, 8, 16, 128 et 512 pour obtenir la puissance 666. Et, pour obtenir les puissances 2, 4, 8, etc... jusque 512, il suffir de multiplier par lui-même le nombre initial et dvitérer. Au passage, le résultat courant (initialement 1) est multiplié par la puissance concernée uniquement si le bit de la décomposition de lvexposant en base 2 vaut 1 pour ladite puissance; et cvest tout - il nvest nullement besoin de récursion.

Il faut 9 produits pour passer de ω (donc $\omega\star2^0$) à $\omega\star512$ (donc $\omega\star2^9$), puis, à partir de $\omega\star2$, il faut 4 produits pour multiplier par $\omega\star8$, puis par $\omega\star16$, puis par $\omega\star128$ et enfin par $\omega\star512$, donc en tout 13 produits et pas un de plus. (Le tout se programme en une ligne).

Selon le même principe, on peut en théorie élever un nombre (ou une matrice) à des

puissances défiant l'imagination.

Rappelons que le simulateur « JUSTE » est écrit en APL-**ISO** pour pouvoir fonctionner tel quel a priori dans n'importe quelle implantation. On n'a pas touché à l'interprète. Tous les exemples présentés dans l'article du N° 16 des Nouvelles d'APL ont été composés en APL*PLUS PC et pourraient fonctionner aussi bien sur un vieux PC-AT (la taille de la ZONE VIERGE dépasse à peine 400 kilo-octets). Economie et Austérité obligent.

II) A propos de divers points du numéro 16

Notes personnelles concernant certains éléments du numéro 16 des "Nouvelles d'APL".

Du bon usage de la Simplicité. Les conclusions en page 45.

La deuxième proposition de Gérard A.Langlet me semble incomplète.
Je suggère une conclusion supplémentaire:

>> La fonction qui utilise moins de primitives APL sera plus performante <<[5]

Exemple: ∇FA fait 10 boucles de 50 primitives
∇FB fait d'abord 100 primitives 1 fois
et 10 boucles de 10 primitives

Pour deux raisons au moins FB sera certainement plus performante que FA :
- L'exécution de 200 primitives au lieu de 500.
- La préparation aura demandé plus de réflexion, donc probablement une meilleure approche du problème et de la codification.

Simplicité de la fonction IOTA. Page 31.
Question : Le >>Diamant<< est-t-il nécessaire ? [6]
[1] R←R+⍳⍴R←V/V←+\V

Quelques Techniques de Programmation. Pages 17 et 18.

Je me porte en faux (tout au moins sur certains points) avec Eric Lescasse, et je vais justifier ma position.

1. Les boucles.

Ma proposition, avec APL2-IBM-Family, indépendante de ⎕IO.

```
→NEXT,I←0
DEB: . . . . .
. . . . .
→(condi)/NEXT
. . . . .
NEXT: . . . . .
→(N≥I←I+1)/DEB
⍝ sinon FIN
```

Justification et explications en 2.

2. Les branchements.

- Il existe au moins 5 primitives possibles, pour un *GOTO* conditionnel, que je détaillerai ci-dessous. et deux raisons déterminant le choix :
1° Le timing (temps d'exécution) n'est pas représentatif car pour une primitive le temps total se compose de deux paramètres :
-un temps fixe *(F)*, c'est l'interprétation et la préparation de l'instruction dépendant d'un interpréteur donné un temps variable *(V)*, c'est l'exécution qui dépendra surtout des arguments associés à l'instruction Le

rapport *(F+V)/V* n'est pas le même avec 10000 octets ou 1 bit.
Le temps *V* est donc négligeable dans un *GOTO*.

2° Utiliser toujours, ou presque, le même idiome de débranchement comportant:

```
LABEL , condition_logique , → , primitive
```

La relecture par soi-même ou un autre en sera facilitée.

Résumé des divers idiomes de codification :

```
a)→ LABEL ×⍳ condi
b)→ LABEL [⍳ condi]
c)→ (L_A,L_B,L_C,L_D,L_E)['ABCD'⍳argument]
d)→ condi ↑ LABEL  ou (~condi) ↓ LABEL
e)→ condi ↓ LABEL ou (~condi) ↑ LABEL
f)→ condi ⍴ LABEL
g)→ n ⍴ LABEL
h)→ condi / LABEL
```

Mon opinion :

a) Très mauvais ! *iota*_0 donne toujours *vide*, soit *GOTO* en séquence, mais *iota*_1 donne ⎕IO (1 ou 0) et *LABEL_fois_0 = GOTO 0 !*

b) Evite le piège du ⎕IO ci-dessus.

c) Rejoint la proposition d'Eric Lescasse en page 18 mais... la logique d'APL2-IBM exige la parenthèse dans ce cas (avec ou sans ,).
Ma proposition, variante de b), donne en plus une sortie pour tous les cas hors *ABCD*.

d) e) En ajoutant encore les variantes de *not_condi*, la compréhension du GOTO si TRUE ou FALSE devient un casse-tête.

Par contre j'utilise occasionnellement le Take pour une condition **trinaire** :

```
d3)→ ( × valeur) ↑ PLUS,MINUS
 ⍝ Signe ZERO en séquence
```

f) Je n'ai rien contre, sauf que "Reshape" ou réorganisation n'est pas explicite d'un choix conditionnel.
>> La bonne instruction au bon endroit ! <<

g) Rejoint le 1. de ces techniques. Si n vaut 5 ou 10, rien à dire.
Mais si n=10000 ou plus, imaginez le vecteur ? en I-APL par exemple.

h) Que j'ai volontairement gardé pour la bonne bouche n'est, à première vue, pas meilleur que d) ou f), mais...
avec la réduction / il est possible de faire aussi une *condition_multiple*.

```
i) → (condi_1,condi_2,condi_3) / LA , LB , LC
```

Exemple: L'argument N doit être 1, 2, 3, ou plus grand. Dans ce cas je préconise la réduction:

```
→ (3 2 1 0 < N) / LN, L3, L2, L1 ⍝ en séquence N=0 ou négatif
```

Donc, pour ma part, j'ai adopté la réduction. A vous de faire mieux !

4. Divers. Règle 2. Page 22.

Sur mon PS2-486-DX2-66MHz avec IBM-APL232, les tests de `aaa ∈ 0 et aaa = aaa = 1` me donnent un rapport de 1/2 et non 1/5.

- Dans ma logique j'aurais remplacé `aaa = aaa = 1 par aaa =× aaa` qui prend un temps équivalent.
- De plus, ⎕MON n'existe pas dans APL*PLUS II Version 5.1

5. Les Curiosités. Page 23, 2ème cas. Il faut rappeler ou préciser que:
- *Zilde* n'existe pas en APL-IBM
- Ce premier élément extrait sera un *scalaire*, et pas un vecteur

```
Si: OBJET ← dimensions ⍴ n'importe_quoi
```

alors: dimensions ⍴ OBJET
et si: dimension est vide, '' ou (⍳0) ou ⍬
alors: l'objet n'a pas de dimension, c'est un scalaire Un scalaire n'est jamais vide. En APL2, ce scalaire peut contenir un tableau, etc
Si: mes explications vous donnent la Grosse Tête,
alors: calmez-vous devant votre écran et demandez à APL de vous convaincre.

6. Un conseil supplémentaire de mon cru.

Il concerne surtout une *(expression_conditionnelle)* mais est aussi valable ailleurs en programmation :

- Dans une relation logique il y a deux variables. Généralement l'une est
- une constante (sdf) "Self Defining Value" et l'autre une variable calculée dans une expression.

Par exemple:est-ce que *VEC* est un vecteur ? `(( ⍴⍴ VEC) = 1)`
ou: 3 fois les dimensions d'un tableau a-t-il une valeur plus grande que X ? `( ∨/ (( ⍴TABLEAU ← (expression)) ×3) >X)`

Règle: >> Toujours mettre la sdf à gauche, ce qui diminue les ( ), entre autres <<
Ces deux exemples donnent : `( 1 = ⍴⍴ VEC)( ∧/ X < 3× ⍴TABLEAU ← expression)`

Souvent, la réorganisation d'une expression fait apparaître d'autres simplifications. Concerne: N.B. page 54 des "Nouvelles d'APL" n° 16.
La dernière phrase : (*Cette . . . globalement*) L'opérateur "CHAQUE" d'APL2 peut être répété sur une ligne comme le montre la fonction ci-dessous tirée d'un livre en APL2:

```
      ∇B_HUFFMAN[⎕]∇
[0]  TREE←B_HUFFMAN PAIRS;SET
[1]   TREE←⍳0
[2]   SET←(⊂¨B_MAKEBTREE¨1⊃¨PAIRS),¨2⊃¨PAIRS
[3]   TREE←B_HUFFTREE SET
    ∇ 1994-06-19 19.14.07
```

Mais . . . à mon avis c'est une mauvaise écriture car un seul opérateur *chaque* (ou "*each*") peut très bien être utilisé globalement :

```
      ∇C_HUFFMAN[⎕]∇
[0]  TREE←C_HUFFMAN PAIRS;SET
[1]   TREE←⍳0
[2]   SET←C_MAKECTREE¨PAIRS   ⍝ La ligne est extraite
[3]   TREE←B_HUFFTREE SET
    ∇ 1995-11-04 22.50.19

      ∇C_MAKECTREE[⎕]∇
[0]  SETT←C_MAKECTREE PAIR
[1]  ⍝ FUNCTION FOR EACH PAIR. stmt [2] de B_HUFFMAN
[2]   SETT←(⊂B_MAKEBTREE 1⊃PAIR),2⊃PAIR
    ∇ 1995-11-04 22.50.10

Petit exemple d'exécution:
      PAIR_5
  A 0.25    B 0.2    C 0.05    D 0.1    E 0.4

      (⎕←C_HUFFMAN PAIR_5)  B_HUFFMAN PAIR_5
 EACDB   E     ACDB   A     CDB   CD   C     D      B
1     ⍝ C.Q.F.D.
```

Note : Dans cet exemple les temps sont équivalents, mais sur un grand vecteur PAIR il n'en sera pas de même.

Conclusion : Gérard doit faire juste une adaptation pour une juste FONCTION_PROGRAMME.

```
Canton de l'Humour helvéto-mathématique (Polytechnicum de Zurich?)
```

de Claude Henriod, CH-Savièse

Mon humour suisse ou mon amour pour la Suisse m'ont inspiré le petit idiome suivant (selon la page 94 du N° 16):

```
      ⎕IO ← ×FP ← ↑PL ← PRIME_NUMBERS_LIST
      FP  2
      220 ≤ ⍴PL

      (FP⊃PL×(((FP⊃PL)⍟FP)⊃PL)⊃PL)⊃PL
1291
   ⍝ Création de la Suisse des Trois Cantons Primitifs
```

d) de M. Gérard Langlet, BP 36, F-Jouy en Josas

à propos de la visite du Dr James A. Brown à Paris
(30 octobre 1995, IBM Rives de Seine).

Il y a quelques années, j'avais mis beaucoup d'espoir dans une possible diffusion libre du produit TryAPL2 d'IBM, correctement converti en français, dans la communauté francophone. D'ailleurs j'ai proposé un didacticiel complet, documenté (Zone GENERI), à la Bourse d'Echange de Logiciels du Congrès APL94 d'Anvers.

Depuis cette dernière conférence de J. Brown, il semblerait que l'une des tendances du développement futur d'APL2 chez IBM serait sa mise à disposition... sous Windows. Pourquoi pas ?

Il y a quelques mois, j'ai acheté un ordinateur portable « Thinkpad » IBM et quelle ne fut pas ma surprise de le recevoir livré tout prêt à fonctionner sous Windows (ce système se chargeant avec une bannière IBM européenne en lieu et place de celle, classique, de Microsoft). Ceci représentait sans aucun doute un signe de changement dans la continuité.

Ayant alors essayé de faire fonctionner TryAPL2 dans cet environnement préchargé avec la bénédiction du constructeur de l'engin, je constatai - amèrement - un refus de Windows avec émission du message : « Ce logiciel a violé l'intégrité du système... etc. », en bref une fin de non recevoir avec laquelle on vous conseille, suite à une « faute de protection générale », de fermer toutes les applications en cours (qui risquent d'avoir été endommagées), puis de recharger Windows et le reste. Agréable, non ?

Déjà, au cours de la précédente conférence de J. Brown à Paris-La Défense en 1993, j'avais posé la question suivante : « Envisagez-vous de diffuser une version de TryAPL2 plus ou moins compatible avec APL2 sous OS2 ? ». La réponse avait été : « Non, parce qu'il existe maintenant une version d'appel du produit (in English « APL2/2 Entry ») complète, en dehors de quelques restrictions (pas de « support », pas d'interface avec SQL) à un prix remarquablement bas ».

A demi-convaincu, je m'empressai toutefois d'acquérir cette version Entry d'APL2 sous OS2 et commandai un ordinateur DX2 avec OS2 préinstallé.

Si, alors, je pus faire fonctionner, à peu près correctement, TryAPL2 comme une application DOS sous OS2, je m'aperçus vite que la lenteur d'exécution n'incitait guère à persévérer dans cette expérience.

Ensuite, le but était de reconvertir **aisément** mes démonstrations (une quarantaine) fonctionnant entièrement en français - commandes APL et messages d'erreur inclus - vers APL2/OS2 Entry.

Je constatai alors que cela aurait requis une charge de travail énorme, pour différentes raisons (dont l'utilisateur ne pouvait être a priori conscient parce qu'à ma connaissance, il n'existe aucune mise en garde), entre autres le fait que la syntaxe d'APL2/2 diffère parfois de celle d'APL2 tout court - donc de celle de

TryAPL2 . Beaucoup plus gênant était le fait que les fichiers ⋆.LOG possèdent des formats incompatibles entre eux (APL2 PC sous DOS par rapport à APL2 sous OS2).

Maintenant, effectivement, l'avenir pédagogique d'APL2 (aussi bien que celui des autres implantations d'APL et de ses dialectes) n'a des chances d'exister que si un pédagogue, arrivant avec une disquette dans la poche, peut s'en servir sur n'importe quelle machine mise à sa disposition, sans que sa démonstration tourne au fiasco, quelle que soient les causes de la cagade et les excuses qu'il trouve.

de M. G. Langlet:

Le dernier courrier de M. Gilles Hervy dans le N° 17 suggère que l'on peut, une bonne fois pour toutes, utiliser une fonction « ru » (pour « *repeat until* ») avec la syntaxe e ru c , l'argument e étant une chaîne de caractères contenant une expression APL exécutable, et l'argument c étant une autre chaîne de caractères contenant une autre expression logique exécutable. Ainsi, la sous-fonction devient-elle acceptable par n'importe quelle implantation d'APL :

```
      ∇ ∆∆e ru ∆∆c;∆∆l
[1]     ⍎∆∆e ◇ ∆∆l←~⍎∆∆c ◇ →∆∆l/1
      ∇
```

Il est judicieux de n'utiliser que des variables n'entraînant pas de conflit avec les variables des expressions exécutables, d'où le préfixe ∆∆ des noms de variables dans cette fonction.

Soient alors la fonction « Premie » et la sous-fonction wh pour « *while* » ::

```
      ∇ R←Premie N;I;P;⎕IO
[1]     R←⍴⎕IO←1 ◇ 'P←N⍴0◇I←¯2◇P[1]←1◇''P←P∨N⍴I↑1◇I←-P⍳0''wh''N≥
I×I''◇P←~P'
          ru '1≥N←¯1+1↑R←(P←WHERE P),R'
      ∇

      ∇ ∆∆e wh ∆∆c;∆∆l
[1]     ∆∆l←⍎∆∆c ◇ ⍎∆∆l/∆∆e ◇ →∆∆l/1
      ∇
```

La sous-fonction WHERE est simplement :

```
      ∇ R←WHERE R      ou bien la fonction en langage-machine présente dans la
[1]     R←R/⍳⍴,R       zone ASMFNS de certaines versions d'APL*PLUS PC.
      ∇
```

Alors, on peut aisément implanter la fonction q2p (équivalent du verbe q: en langage J), qui décompose le nombre entier positif donné comme argument droit, en ses facteurs premiers :

```
      ∇ R←q2p N;J;L
[1]   J←Premie⌊N⋆0.5 ◇ L←0=J|N ◇ 'R←R,q2p N÷J'do 1≠J←×/R←L/J
 ◇ 'R←N'do 0=⍴R←R[⍋R] ◇ R←R~1
      ∇
```

(La sous-fonction do – ou if - exécute l'expression APL contenue dans son argument gauche, le nombre de fois précisé par l'argument droit; si l'argument droit est une condition logique, l'expression de gauche est exécutée 0 fois ou 1 fois.)

L'extension « sauf » (symbolisée par ~ diadique) est utilisée. Elle est admise par toutes les implantations courantes d'APL.

Exemples :

```
      q2p 4194303
3 23 89 683
```

```
      q2p 8126433
3 3 3 7 19 31 73

      q2p 11111111111
21649 513239
      ×/21649 513239
11111111111
```

Comparons alors avec ce que donne **J** pour le même nombre :

```
   q:11111111111
³domain error
³       q:1.11111e10
```

(**J** n'accepte pas que l'argument entier soit codé en virgule flottante.)

```
   q: inverse 21649 513239
1.11111e10

   inverse
puissance _1

   puissance
^:
```

Prière de se reporter à l'article de Chris Burke, traduit par Sylvain Baron, paru dans `Les Nouvelles d'APL,` N° 17 (déc. 1995), p. 15.

Note. La fonction `q2p` peut être simplifiée donc accélérée (notamment en faisant disparaître la récursivité et les redondances).

---

[1] Aphorisme qui s'appliquait déjà au « Quinto » en éliminant - a priori - tout recours au domino.

[2] L'idiome ≠⍀ est la propriété de tous les APListes...Il est capable de remplacer les banques d'idiomes les mieux gardées; il opère au grand jour et non pas en suisse.

[3] Merci !

[4] Merci (bis) !

[5] Réponse : Tout à fait d'accord, en général, mais n'est-ce point une évidence ? (G. Langlet).

[6] Réponse : Non, le diamant n'est pas nécessaire; si je l'emploie c'est a) parce qu'il est accepté dans toutes les implantations courantes, y compris APL2/2 et TryAPL2, b) le diamant c'est une assurance anti-Zone_Pleine, les variables intermédiaires créées en APL à l'insu de l'utilisateur n'étant pas détruites avant la fin d'exécution complète de l'instruction. (idem).

§ par exemple, en APL2/2, un espace est obligatoire entre un argument gauche numérique et le nom de fonction diadique qui suit.

§§ Les fichiers `*.LOG` d'APL2 sont le fruit d'une excellente idée initiale. Lorsque l'on ferme la session APL (à l'aide de `)OFF` en anglais, mais de `)FIN` en français, ce qui est beaucoup plus francophoniquement pédagogique, le tampon de session qui contient un certain nombre de pages - jusqu'à 64 kilo-octets en APL2 - de tout ce que vous avez pu faire - de bien ou de moins bien - dans toute la session, se trouve automatiquement sauvegardé dans un fichier de nom `APL2$SES.LOG` lequel se recharge automatiquement à l'appel suivant d'APL2. Ce fichier joue un peu le rôle de l'ancienne zone `CONTINUE` des APL sur grosses machines, mais, au lieu d'une restauration de l'environnement, on a une restauration

des événements pédagogiques de la session. L'intérêt majeur provenait de l'avantage que ces fichiers - pouvant par exemple contenir le listage complet (et effectué automatiquement à l'aide d'une toute petite fonction) d'une zone de travail entière, étaient totalement compatibles entre APL2/PC et TryAPL2. On disposait en outre d'une gestion des couleurs personnalisable, utile et agréable. Transférer une application d'APL2 PC vers TryAPL2 devenait un jeu d'enfant (on pouvait ignorer la sophistication - de mauvais goût - du passage par des fichiers *.OBJ (fichiers-objet), et même créer des fichiers `*.LOG` (à condition de respecter la norme ISO) à partir d'APL*PLUS PC ou, a priori, depuis n'importe quel autre APL, une fois que l'on avait déchiffré la structure intime - non documentée - desdits fichiers `*.LOG` d'APL2/PC. Un intérêt majeur d'opérer ainsi provenait du fait que TryAPL2 code les vecteurs et tableaux binaires en bits alors qu'APL*PLUS PC les code en entiers courts sur 2 octets, donc avec un gaspillage de mémoire, dramatique lorsqu'on ne dispose que de 300k pour travailler, d'un facteur 16. (On peut par exemple résoudre en TryAPL2 le Quinto d'ordre 1OO sans difficulté, et le visualiser semi-graphiquement sans troncature, car la largeur des lignes de session atteint 254 caractères par ligne, alors qu'APL*PLUS PC ne permet pas de se déplacer dans la session sans repli, au delà des 80 caractères affichables sur une même ligne).