Sc.N.76/119 B

UNIVERSITE DE NANCY I

THESE

UNIVERSITE NANCY BIBLIOTHÈQUE

pour obtenir le diplome de docteur-ingenieur

REALISATION D'UN INTERPRETE COMPLET DU LANGAGE APL SUR UN MINI-ORDINATEUR

par

Jean-Jacques GIRARDOT
François MIREAUX

Soutenue le 27 Septembre 1976

MEMBRES DU JURY

President : M.Pair
Examinateurs : MM. Coulon
Derniame
Griffiths
Guiboud-Ribaud
Mahl

BIBLIOTHEQUE SCIENCES NANCY 1
D 095 183161 6

UNIVERSITE DE NANCY I

THESE

pour obtenir le diplome de docteur-ingenieur

REALISATION D'UN INTERPRETE COMPLET DU LANGAGE APL SUR UN MINI-ORDINATEUR

par

Jean-Jacques GIRARDOT
François MIREAUX

Soutenue le 27 Septembre 1976

MEMBRES DU JURY

President : M. Pair
Examinateurs : MM. Coulon
Derniame
Griffiths
Guiboud-Ribaud
Mahl

Nous tenons à remercier :

Monsieur Claude PAIR, Président de l'Institut National Polytechnique de Lorraine, dont l'aide nous a été précieuse tout au long de la réalisation de cette thèse, et qui nous a fait l'honneur d'en présider le Jury.

Messieurs Daniel COULON, Jean-Claude DERNIAME, et Michael GRIFFITHS, qui se sont intéressés à notre travail, et ont bien voulu faire partie du Jury.

Monsieur Robert MAHL, qui est à l'origine de ce travail et en a guidé le démarrage alors qu'il était Directeur du Département Informatique de l'Ecole des Mines de Saint.Etienne.

Monsieur Serge GUIBOUD-RIBAUD, Directeur du département Informatique de cette même Ecole, et qui nous a permis de faire aboutir ce projet.

Madame BONNEFOY, qui a assuré la frappe de notre manuscrit, ainsi que Monsieur LOUBET, qui en a effectué le tirage.

o
o o

S O M M A I R E

0=0=0=0=0=0=0=0=0=0

INTRODUCTION.

1 - Avant-propos
2 - Historique de la réalisation
3 - Financement des recherches
4 - Plan de la thèse

PREMIERE PARTIE : UTILISATION D'APL SUR T1600

I - Description d'APL/1600
II - Utilisation d'APL/1600 pour des applications scientifiques
III - Applications graphiques d'APL/1600
IV - Utilisation des fonctions système et autres extensions d'APL/1600

DEUXIEME PARTIE : CONCEPTION ET MISE AU POINT

1 - CONCEPTION DU CALCULATEUR APL
A - aspects généraux
I - Les objets APL
II - Le dynamisme de l'exécution
III - Les approches possibles
IV - Notre approche

B - L'interprète
I - Les approches possibles
II - La machine naïve
III - Le code exécutable
IV - La conception de la gestion de mémoire
V - Le fonctionnement

2 - DESCRIPTION DE L'IMPLEMENTATION

A - Représentation interne des objets APL
B - Gestion de la mémoire
C - Interprétation
D - Changements de contexte

3 - MESURES ET OPTIMISATION

A - Mesures et amélioration des algorithmes
B - Optimisation de la structure de l'interprète

TROISIEME PARTIE : MATERIEL ET INGENIERIE.

1 - LE MATERIEL
2 - INGENIERIE

I - La méthode de programmation
II - Utilisation du T1600
III - Utilisation d'un gros ordinateur

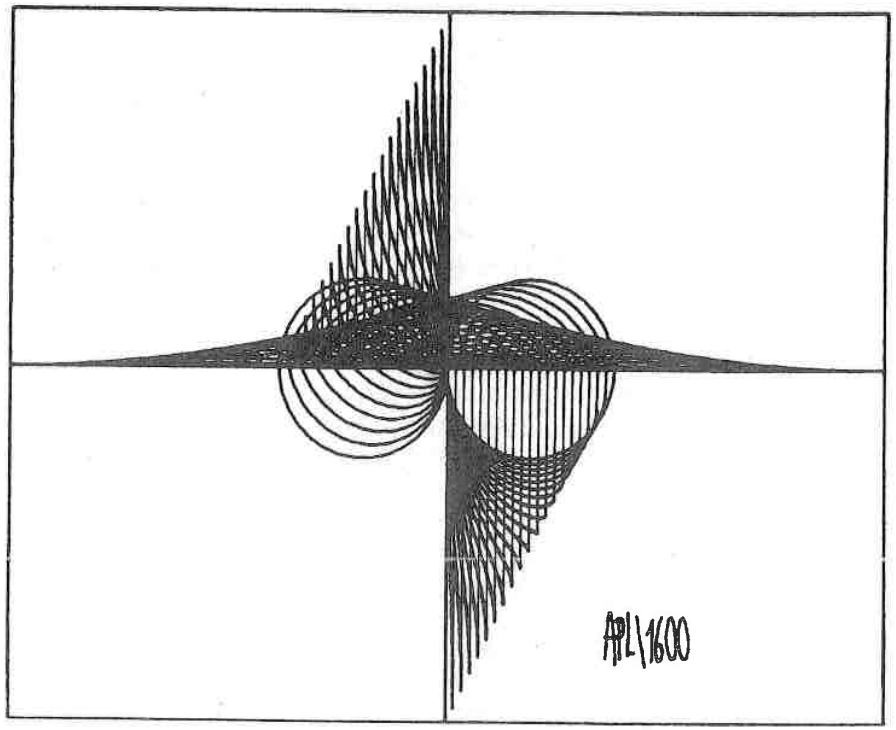
CONCLUSION

1 - Intérêt de la réalisation
2 - Evolution récente et perspective à long terme

ANNEXES

1 - Deux applications d'APL/1600

INTRODUCTION



1 - AVANT PROPOS.

K.E. IVERSON a défini le langage APL en 1962 dans un ouvrage intitulé "A Programming Language". Malgré les difficultés nouvelles que posait ce langage aux implémenteurs, des interprètes APL furent rapidement disponibles chez IBM, d'abord à titre expérimental, en 1965, puis commercialisés sur ordinateur 1130 et 360 (A3, A4). En effet, ce langage s'était d'emblée révélé intéressant pour toutes les applications réclamant plus de programmation que de longs calculs.

Depuis cette époque, d'autres interprètes APL ont vu le jour chez d'autres constructeurs (B2, B3, B4, B8). Cependant, réalisés sur de gros ordinateurs, ils réclament des ressources hors de proportion avec les moyens habituels des ingénieurs ou des étudiants.

Dans cette optique, il a paru intéressant de réaliser un interprète APL sur une petite machine, partagée entre plusieurs utilisateurs.

Réalisé sur ordinateur T1600, aux Mines de Saint-Etienne, cet interprète a nécessité cinq hommes-années aux auteurs. Les principales difficultés étaient dues à la petite taille de la mémoire centrale, à l'inadéquation du jeu d'instructions, et au manque de souplesse du logiciel disponible sur le calculateur.

De plus, les auteurs ont jugé intéressant d'augmenter la puissance du langage par des opérateurs supplémentaires. C'est ainsi qu'ont été incorporées des facilités pour la génération d'images, la gestion d'une tablette graphique, l'auto-exécution d'instructions générées par programme.

L'interprète a été testé sur de nombreux exemples extraits de la littérature (D1, D5), ou inventés par les utilisateurs d'APL/1600, élèves et chercheurs de l'Ecole des Mines de Saint-Etienne.

La maniabilité du produit a conduit de nombreux non informaticiens (chercheurs en gestion, métallurgie, réalisation de matériel audio-visuel) à s'y intéresser.

Enfin, des mesures de performances ont montré la vélocité de l'interprète, atteinte par la mise en oeuvre de techniques nouvelles d'optimisation.

2 - HISTORIQUE DU PROJET APL 1600.

De nombreuses étapes ont jaloné ce projet. Il y eut tout d'abord un début de réalisation d'interprète APL pour gros ordinateur (Philips P1100), qui a constitué le projet long de J.J. GIRARDOT au cours de sa 3ème année comme élève à l'Ecole des Mines de Saint-Etienne (1971-72). L'analyseur syntaxique fonctionnait en une passe, en utilisant un analyseur lexicographique simple. Celui-ci reconnaissait identificateur et constantes, et décodait les opérateurs APL, présentés sous forme de mots clé (comme RHO, IOTA, etc...).L'analyseur syntaxique lui-même tenait compte de la valeur sémantique associée à chaque symbole. Fonctionnant de la gauche vers la droite, il était capable de reconnaître les erreurs de syntaxe, mais imposait des contraintes artificielles sur le langage interprété.

Une fois la phrase APL reconnue correcte, le module d'exécution effectuait les calculs et imprimait éventuellement le résultat (les données étaient forcément scalaires et de type réel). Cependant, cette première version avait eut l'avantage de dégager un certain nombre de principes généraux, dont il a été tenu compte dans les réalisations ultérieures.

Au cours de l'année 72-73, les premiers travaux de conception ont été effectués pour l'APL/1600. Ces travaux ont été menés en collaboration entre l'Ecole des Mines de Nancy et l'Ecole des Mines de Saint-Etienne. Au cours de cette période, un certain nombre d'algorithmes ont été programmés et mis au point sous APL/1130, tant à l'ISIN (Institut des Sciences de l'Ingénieur de Nancy) qu'à la MAS (Manufacture d'Armes de Saint-Etienne), où des ordinateurs 1130 dotés d'interprètes APL étaient disponibles. C'est ainsi que furent effectués les travaux suivants :

- Mise au point de la première version de la gestion de mémoire, par François Dominique ARMINGAUD [B6], au moyen de fonction APL réalisant la gestion de listes chaînées dans une mémoire de travail.

- Ecriture par J.J. GIRARDOT d'un mini-interprète APL en APL, effectuant un certain nombre d'opérateurs scalaires (plus, moins, multiplié et divisé) et de restructuration (take, drop, et ratate), par une méthode d'interprétation arborescente optimisée. Cette méthode dont l'inspiration est venue début 72 après des contacts avec P. Abrams [C1], consiste à construire d'abord l'arbre

correspondant à l'expression APL que l'on désire évaluer, puis calculer successivement le rang, les dimensions et enfin les valeurs des éléments du résultat. A chaque fois, une seule valeur est remontée à la demande le long de l'arbre, ce qui permet d'éviter des calculs inutiles, avec cette nuance que si le résultat d'un sous-arbre est entièrement nécessaire, il sera évalué en une seule fois (par appel récursif de la procédure d'évaluation), et remplacé dans l'arbre par son résultat temporaire. Cette méthode n'a finalement pas été retenue dans la conception d'APL 1600, car elle nécessitait en particulier de disposer d'un interpréteur entièrement résident, ce qui n'était pas possible dans le cas du T1600.

- Ecriture des opérateurs de restructuration, sous une forme directement transposable en PL1600, avec utilisation de procédures très générales d'accès à l'élément et de contrôle de l'itération. Tous les opérateurs APL ont ainsi été décrits et mis au point, ce qui a permis par la suite de les introduire très vite dans l'interprète actuel.

En même temps débutaient les premiers essais de programmation de l'interprète lui-même. Ce furent d'abord à Nancy la première version de la gestion de mémoire (F.D. Armingaud et F. Mireaux) ainsi que la gestion des fonctions (Mergault), et à Saint-Etienne la gestion de la Table des symboles et la génération de code interne ainsi que les entrées sorties sur télétype, car on ne disposait pas alors de terminaux APL.

Compte tenu de l'insuffisance du système d'exploitation proposé par Télémécanique, un ensemble d'outils a dû être mis au point au cours de l'année 1973 par les auteurs (CF. IIIè partie).

A partir de Juillet 1973, les recherches ont été menées à temps plein à Saint-Etienne par les auteurs. C'est ainsi qu'ont été réalisés successivement :

- un premier prototype, comportant les opérateurs scalaires élémentaires, et une nouvelle gestion des fonctions opérant sur un terminal APL (Tektronix 4013), dès Octobre 1973

- en Mars 1974, une nouvelle version comportait de nouveaux opérateurs, ainsi que la possibilité de définir et d'utiliser dans les expressions APL des fonctions définies par l'utilisateur.

- au mois d'Avril, cette première version de l'interprète faisait l'objet d'une transposition sous le système d'exploitation standard de la Télémécanique, BOS/D, à l'usine d'Echirolles. Vers cette époque, la connexion entre le T1600 et le Philips P1175 avait été réalisée par J.F. Chambon et B. Le Bihan, ce qui permit d'accélérer notablement les travaux de mise au point de la logicielle du T1600 (cf. 3ème partie).

- en Juin 1974, tous les opérateurs disponibles sous APL/360 étaient implémentés. Cette version comportait en outre l'utilisation des commandes de sauvegarde et de chargement de zones de travail, à partir du disque.

- en septembre 1974, APL était en démonstration au SICOB sur le stand de la Télémécanique Electrique.

De Novembre à Janvier, eurent lieu les premières tentatives d'intégration de l'interprète APL sous le système de temps partagé de la Télémécanique, TSM. Ces essais ont permis de voir quelles étaient les modifications à apporter à ce système pour permettre une utilisation agréable d'APL en multi-utilisateur.

En même temps, au cours du premier semestre de l'année 1975, le système APL était assorti de commandes nouvelles permettant l'introduction et l'utilisation sous APL de programmes en langage machine T1600, désignés sous le terme de Fonctions Système. Celles-ci permettent de disposer sous APL de possibilités coûteuses ou non prévues, comme le formattage de données, l'accès à un système de fichiers, etc...

Depuis mars 1975, APL est proposé aux utilisateurs par la Télémécanique, dans sa version mono-console utilisable sous BOS/D. La version multi-console opérationnelle depuis Septembre 1975 est utilisée à l'Ecole des Mines de Saint-Etienne par des élèves, des chercheurs des différents laboratoires, notamment le Centre de Technologie Educative pour les essais d'un système de réalisation d'images et de films semi-animés, et quelques utilisateurs extérieurs occasionnels.

3 - FINANCEMENT DES RECHERCHES.

Ces recherches ont pu être menées grâce aux organismes suivants, qui ont apporté un concours financier à l'opération :

1) Direction de la Technologie, de l'Environnement Industriel et des Mines (Ministère de l'Industrie et de la Recherche) : a payé le matériel utilisé et certains frais de personnel.

2) Délégation à l'Informatique (Ministère de l'Industrie et de la Recherche) : a passé successivement 3 conventions de recherche, dont l'une pour la conception de l'interprète avec l'Ecole des Mines de Nancy, et deux pour sa réalisation avec l'Ecole des Mines de Saint-Etienne.

3) La Télémécanique Electrique : prête 8 K mots de mémoire à l'Ecole des Mines de Saint-Etienne depuis Janvier 1975.

4 - PLAN DE LA THESE.

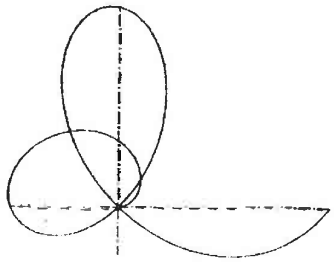
Dans une première partie, nous effectuerons une brève présentation du langage APL, en insistant sur les particularités introduites dans notre réalisation. Nous donnerons ensuite, un certain nombre d'exemples d'utilisations, concernant notamment les applications scientifiques et graphiques du système APL/1600, présentant également l'emploi des fonctions systèmes. Nous terminerons par une brève perspective d'avenir, en particulier en ce qui concerne l'utilisation future du produit ainsi que ses extensions possibles.

Dans une seconde partie, nous essayerons tout d'abord de dégager les idées générales ayant présidées à la conception de l'interprète. Après avoir précisé la notion de calculateur APL, ainsi que les diverses contraintes dues aux choix préalables et à l'implantation sur une petite machine, nous définirons plus précisément la structure du calculateur. C'est ainsi que seront explicitées la nature des objets manipulés, les structures de données mises à contribution, et enfin ce qu'est l'état du calculateur. Ensuite, seront abordés plus en détail les algorithmes utilisés, en particulier pour la gestion de la mémoire. Nous terminerons en précisant certains détails de fonctionnement du système APL/1600, et en présentant quelques mesures effectuées.

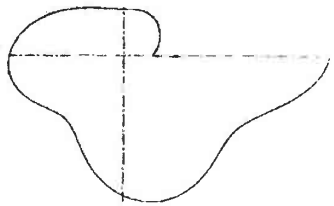
La troisième partie traitera plus spécifiquement des outils logiciels que nous avons été conduits à réaliser pour permettre une écriture aisée de l'interprète. Cette description recouvre les améliorations apportées au système d'exploitation fourni par la Télémécanique, la présentation des outils de recouvrement et d'aide à la mise au point, et enfin l'utilisation de la connexion avec un gros ordinateur pour la maintenance des programmes composant l'interprète.

Nous conclurons enfin, en l'intérêt de la réalisation effectuée, en faisant un bref survol des développements ultérieurs possibles.

PARTE I



RHO=TETA×SIN(TETA+COS(TETA))



RHO=TETA+COS(TETA×SIN(TETA))

TETA VARIANT DE 0 A 2 PI

DESCRIPTION D'APL/1600

Nous présentons d'abord le langage APL en résumant la démarche qui a conduit Iverson à le définir et en dégageant les principales caractéristiques du langage. Cette description n'est destinée qu'à donner un avant-goût du langage sans en montrer la puissance réelle. Il existe des ouvrages beaucoup plus complets sur le sujet, tels que " Le langage APL " de B. ROBINET (A2).

Nous comparons ensuite notre implémentation avec plusieurs implémentations bien connues notamment APL-SV ceci du point de vue spécifications externes et non mesures de performances. Nous montrons les restrictions que nous avons été amenés à faire, principalement dues à la faible taille de l'ordinateur ainsi que les extensions développées portant sur l'introduction de nouveaux opérateurs, ou la généralisation du domaine de validité d'opérateurs déjà existants.

I - UNE INTRODUCTION BREVE AU LANGAGE APL :

En 1962 paraissait un ouvrage de K.E. Iverson, intitulé " A Programming Language " (A1), qui décrivait ce qui allait devenir le langage APL. Les premières implémentations sur IBM/1130 (A3) et sur IBM/360 (A4) ont fixé l'alphabet et la syntaxe d'APL, tels qu'ils sont universellement employés de nos jours.

a) L'alphabet :

Le langage APL résume un effort accompli en vue de formaliser un certain nombre de notations mathématiques. La notion d'opérateur recouvre le sens plus général de fonction mathématique. Iverson justifie lui-même son approche,dans son introduction au colloque APL de l'IRIA, en 1971 (A6), en citant les exemples suivants :

Notation mathématique	Signification	Ecriture APL
$-X$	opposé	-X
$\|X\|$	valeur absolue	|X
$X!$	factorielle	!X
$\frac{1}{X}$	inverse	÷X
$\text{Log } X$	logarithme	⍟X
e^X	exponentielle	*X
AB	produit	AxB
$\frac{A}{B}$	division	A÷B
A^B	puissance	A*B
$\text{Log}_A B$	logarithme en base A	A⍟B
$\text{Max}(A,B)$	maximum de A et B	A⌈B

En particulier, la distinction est établie entre les opérateurs monadiques (un argument), et les opérateurs dyadiques (deux arguments), qui peuvent avoir une représentation commune (témoins les exemples ci-dessus), sans que ce soit une cause d'ambiguité, et ceci grâce à une syntaxe simple et rigoureuse.

Un grand nombre d'opérateurs APL représentant les fonctions mathématiques usuelles sont ainsi fournis à l'utilisateur. Ces opérateurs sont dits "scalaires". Les données sur lesquelles ils s'appliquent sont des tableaux, au sens habituel du terme (exemple : FORTRAN).

Un même effort est fait pour normaliser la représentation des constantes numériques. Ainsi, une écriture telle -3 représente habituellement le nombre négatif moins trois. En fait, elle peut également s'interpréter comme une expression calculant l'opposé du nombre positif trois. Cette ambiguité est levée en APL, où il existe le signe ¯ pour indiquer une constante négative : ¯3. De même, les notations telles que $1{,}25 \times 10^3$ ou 10^{-5} vont s'écrire 1.25E3 et 1E¯5.

Cette distinction est rendue nécessaire par le fait qu'une constante APL peut être non seulement scalaire (c'est-à-dire, ne comporter qu'un seul élément), mais aussi vectorielle. Les différents éléments de la constante sont alors séparés par un blanc : 3 5 8 représente une constante vectorielle de trois éléments. La distinction entre l'opérateur moins (-) et le signe négatif (¯) permet de différencier des expressions comme 3-1 (calcul de la valeur 2), et 3¯1 (constante vectorielle de deux éléments, trois et moins un).

Enfin, de même que dans tout langage de programmation, l'utilisateur peut donner à ses variables des noms symboliques. Classiquement, un identificateur APL est une suite de lettres et de chiffres, commençant par une lettre, et dont la longueur maximale est laissée à la discrétion de l'implémenteur.

b) La syntaxe :

Les opérateurs que nous avons vus étaient dits scalaires car ils s'appliquaient élément par élément à leurs opérandes. Ainsi, l'expression :

1 2 3 4 + 5 6 7 8

à la valeur

6 8 10 12

Il est possible de modifier la manière dont s'appliquent les opérateurs scalaires. Prenons la notation mathématique :

$$\sum_{i=1}^{N} V_I$$

qui représente la somme des éléments du vecteur v.

En APL, nous écrirons :

+/V

Cette expression est dite : réduction par l'opérateur plus, du vecteur V.

De la même façon $\prod_{i=1}^{N} V_i$ devient x/V.

Un autre exemple d'extension des opérateurs scalaires est le produit interne. Ainsi, si *A* et *B* sont des tableaux à 2 indices, un exemple de produit interne est l'expression A+.×B qui effectue le produit matriciel ordinaire.

Alors que les opérateurs scalaires conservent la structure de leurs opérandes, d'autres opérateurs permettent de la modifier : ce sont les opérateurs mixtes. Ainsi l'opérateur ⍳ s'appliquant à un entier *N* va fournir comme résultat un vecteur de *N* éléments dont les valeurs seront les entiers consécutifs de 1 à *N*. l'opérateur ⍴ restructure l'opérande de droite de la manière spécifiée par l'opérande de gauche. Par exemple, l'expression 3 3⍴⍳9 crée une matrice carrée de taille 3x3. et composée des éléments 1 à 9. L'affectation ← peut s'employer dans une expression *APL* au même titre que n'importe quel autre opérateur.

Une variable est définie à partir du moment ou l'utilisateur lui a donné une valeur par l'opérateur d'affectation :

< identificateur > ← < expression APL >

Il n'y a aucune déclaration de type ou de taille. La variable prend à chaque affectation les caractéristiques du résultat de l'expression APL. Elle peut prendre successivement le type numérique puis caractère, passer de scalaire à un tableau de rang et de taille quelconques ou inversement.

Signalons pour terminer qu'une expression APL est évaluée de la droite vers la gauche sans aucune priorité entre les opérateurs. Ces règles, qui peuvent paraître arbitraires ou restrictives, ont l'avantage d'être simple à retenir. En outre, il est toujours possible d'utiliser des parenthèses pour modifier l'ordre de l'évaluation.

Si, pour un certain problème, l'utilisateur a besoin d'un opérateur particulier, et qu'il ne trouve pas cet opérateur dans le langage lui-même, il lui est possible de le définir sous forme d'une fonction. Ainsi, pour définir un opérateur calculant la moyenne des éléments d'un vecteur, il écrira :

```
∇R←MOY V
R←(+/)÷⍴V
∇
```


Il peut ensuite utiliser cet opérateur ainsi défini comme un des opérateurs standard du langage :

2+*MOY* 2 3 4 5 va lui rendre la valeur 5.5

Dans les implémentations actuelles, APL se présente généralement sous forme d'un langage conversationnel. Toute expression entrée au pupitre est évaluée par l'interpréteur, et son résultat rendu à l'utilisateur.

c) <u>L'apport d'APL a la programmation</u> :

APL est un langage de programmation pour les gens qui ne veulent pas faire de la programmation. Il permet, en particulier, une bonne diminution du bruit de fond dû à la programmation pratique d'un algorithme. On peut désigner sous le terme de bruit de fond d'un programme toutes les instructions qui ne sont pas essentielles à la compréhension de l'algorithme, mais sont imposées par le langage choisi, en particulier les déclarations, les indices des boucles, etc... Un exemple aidera à faire comprendre cette notion [D6] :

Un classique calcul de circuit électrique est l'expression :

$$M=A^t xPxA$$

dans laquelle A et P sont des matrices carrées d'ordre N.

En FORTRAN on peut écrire :

```
      DØ 10 I = 1,N
      DØ 10 J = 1,N
10    M(I,J)  = 0
      DØ 20 I1=1,N
      DØ 20 J1=1,N
      DØ 20 I=1,N
      DØ 20 J=1,N
20    M(I1,J1)=M(I1,J1)+P(I,J)*A(I,I1)*A(J,J1)
```

En APL on écrira :

M←(⍉*A*)+.×*P*+.×*A*

D'une manière générale, la concision et la transparence d'APL font que l'on obtient une meilleure vue globale de l'ensemble du programme. Un exemple n'a bien sûr jamais rien démontré, mais il apparait intuitivement que l'écriture et la mise au point d'un programme APL sont plus rapides que celles d'un programme FORTRAN. D'un point de vue pratique, des gains de temps de l'ordre de 10 ne sont pas exceptionnels. Il est courant de voir des sociétés de service en informatique, devant fournir un certain logiciel pour un certain matériel, écrire et mettre au point en APL cette application avant de la transcrire dans le langage désiré pour l'exploitation.

II - DESCRIPTION COMPARATIVE DU LANGAGE IMPLEMENTE :

Cette comparaison porte sur les spécifications externes et non sur les performances de l'interprète.

Bien qu'il n'existe pas à proprement parler de normes pour le langage APL, l'implémentation réalisée sur l'ordinateur IBM 360 [A4] a jusqu'à présent servi de modèle de référence.

Les restrictions de notre implémentation résultent d'une part des caractéristiques de l'ordinateur hôte, d'autre part de choix spécifiques visant à minimiser l'encombrement de certaines structures de données, plus que de la réduction du domaine d'applications des opérateurs eux-mêmes.

C'est ainsi que le nombre d'indices pour les tableaux est limité à 4, le nombre d'éléments d'un tableau à 32767, le nombre de lignes d'une fonction à 320 environ.

Ces restrictions ne sont pas réellement significatives : il est rare de trouver des fonctions APL de plus d'une cinquantaine de lignes, ou des applications nécessitant des tableaux de plus de 4 indices. Par contre, la représentation des nombres réels ou double mots, soit 32 bits, ne donne que 6 ou 7 chiffres significatifs et semble insuffisante pour des calculs complexes ou demandant une bonne précision.

Seuls quelques opérateurs comme le laminage ou la fonction ⍕ n'ont pas été implémentés. Certains toutefois, comme le domino (inversion de matrice, résolution de système binaire) répondent à des spécifications restrictives. Dans le cas de l'opérateur I-beam, interface entre l'utilisateur et le système APL, les spécifications sont différentes d'une implémentation à l'autre mais les fonctions remplies sont presque identiques.

L'accès à un système de gestion de fichier qui existe maintenant sur pratiquement toutes les implémentations n'a pas encore été réalisé, mais la possibilité pour un utilisateur d'inclure dans le système APL des programmes écrits dans un langage de programmation quelconque devrait permettre d'écrire facilement l'interface nécessaire.

De même les variables partagées, qui n'ont pas de sens dans une version monoconsole, seront implémentées dans la version multi-console, dans le cadre de l'utilisation d'une mémoire virtuelle câblée.

La taille des zones de travail est comparable à celle des autres implémentations puisqu'elle peut atteindre 64K octets.

Les extensions apportées visent essentiellement à utiliser toutes les possibilités de la configuration. Ainsi l'utilisateur a accès à tous les périphériques aussi bien en entrée qu'en sortie.

Les primitives graphiques implémentées et constamment améliorées facilitent l'utilisation d'APL comme outil graphique et constituent la première approche d'un système graphique complet.

Au niveau du langage le domaine d'application de l'opérateur "execute" (⍎) a été étendu à l'édition de fonction et aux commandes systèmes, permettant ainsi la gestion dynamiques des zones de travail et la création automatique de fonctions.

Des opérateurs récemment apparus dans le langage, comme le "scan" (\) analogue à la réduction, ou le formattage de données (⍕) sont également implémentés.

En ce qui concerne l'édition de texte on peut citer comme extension intéressante la possibilité d'insérer dans une fonction des lignes extraites d'une autre fonction.

Enfin, dans le cadre de l'utilisation d'APL pour l'enseignement, une hiérarchie a été introduite entre les utilisateurs : il n'y a plus un utilisateur privilégié et les autres, mais différentes classes d'utilisateurs, avec chacune leurs prérogatives particulières. De nouvelles commandes systèmes permettent de contrôler cette hiérarchie.

UTILISATION D'APL/1600 POUR DES APPLICATIONS SCIENTIFIQUES

****

La littérature contient de nombreux exemples d'applications scientifiques du langage APL, depuis la recherche opérationelle jusqu'à la preuve automatique de théorèmes en passant par les statistiques. C'est pourquoi, ce chapitre n'est destiné qu'à donner un avant goût des possibilités d'APL/1600 en montrant sur quelques exemples simples ce qu'on pourrait appeler "l'esprit" du langage APL, ou comment s'écrivent en APL quelques algorithmes élémentaires. Après quelques fonctions plus élaborées nous présenterons une application complète tirée d'un article publié par P. Abrams et W.M.Mc Keeman [D5] permettant le calcul des polytopes dérivés dans un espace à n dimensions.

CALCUL DE e.

On connaît la définition du nombre irrationnel e :

$$e = \sum_{n=0}^{\infty} \frac{1}{n!}$$

ou encore

$$1 + \sum_{n=1}^{\infty} \frac{1}{n!}$$

La série convergeant très rapidement, on obtient une bonne approximation de e en écrivant

$$e = 1 + \sum_{n=1}^{10} \frac{1}{n!}$$

ce qui s'écrit en APL :

```
      1+÷/÷!⍳10
2.718282
```

Décomposons cette expression en sachant qu'elle s'interprète de droite à gauche :

```
      ⍳10
1  2  3  4  5  6  7  8  9  10

      ÷!⍳10

1  .5  .1666666  4.166666E¯2  8.333333E¯3  1.388888E¯3  1.984127E¯4
      2.480159E¯5  2.755732E¯6  2.755732E¯7

      +/÷!⍳10

1.718282

      1++/÷!⍳10

2.718282
```

Etudions la convergence de la série en écrivant

```
      1++\÷!⍳15

2  2.5  2.666667  2.708333  2.716666  2.718055  2.718254  2.718279
      2.718281  2.718282  2.718282  2.718282  2.718282  2.718282
      2.718282
```

ce qui donne les différentes approximations de e pour N allant de 1 à 15 : compte tenu de la précision de la machine, on atteint la meilleur approximation possible déjà pour n = 9.

APPROXIMATION DE pi

De même le développement limité de Arctg(x) dont le terme général peut s'écrire pour n $\geqslant$ 1

$$U_n = (-1)^{n-1} \frac{x^{2n-1}}{2n-1}$$

permet d'obtenir un encadrement de pi. En effet Arctg(1) = π/4 d'où pour n pair on peut écrire :

```
      N←200
      X←-/÷¯1+2×⍳N
      Y←X+÷1+2×N
```



```
      X
3.136593
      Y
3.146568
```

On peut donner une approximation de pi avec :

```
      X+2÷1+2×N
3.14159
```

APPLICATION DU PRODUIT EXTERNE.

L'opérateur produit externe est extrèmement précieux : il permet d'appliquer un opérateur scalaire dyadique entre toutes les paires d'éléments des deux opérandes Ainsi une table de multiplication s'écrit :

```
      (⍳10)∘.×⍳10
1   2   3   4   5   6   7   8   9   10
2   4   6   8   10  12  14  16  18  20
3   6   9   12  15  18  21  24  27  30
4   8   12  16  20  24  28  32  36  40
5   10  15  20  25  30  35  40  45  50
6   12  18  24  30  36  42  48  54  60
7   14  21  28  35  42  49  56  63  70
8   16  24  32  40  48  56  64  72  80
9   18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90  100
```

De même l'expression :

```
      ⍉(0,⍳5)∘.!0,⍳5

1   0   0   0   0   0
1   1   0   0   0   0
1   2   1   0   0   0
1   3   3   1   0   0
1   4   6   4   1   0
1   5   10  10  5   1
```

donne les coefficients du binome. L'opérateur de transposition (⍉) ne sert qu'à présenter le résultat sous une forme plus habituelle.

Le produit externe est souvent associé à une réduction pour sélectionner une partie d'un ensemble ; par exemple le calcul de tous les diviseurs d'un entier n s'écrit :

```
      (0=(⍳N)∘.|N)/⍳N←396
1  2  3  4  6  9  11  12  18  22  33  36  44  66  99  132  198  396
```

sachant que pour A et B positifs la définition de l'opérateur résidu (|) est :

```
      B-A×⌊B÷A
```

D'où on déduit facilement l'expression donnant le pgcd de plusieurs nombres:

```
      ⌈/(∧/0=I∘.|U)/I←⍳⌊/U←36 30 45
3
```

Enfin décomposons l'algorithme de calcul des nombres premiers inférieurs à N donnés par l'expression :

```
      (2≤+⌿0=(⍳N)∘.|⍳N)/⍳N←20
1  2  3  5  7  11  13  17  19

      0=(⍳N)∘.|⍳N
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0
0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1
0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1
0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
```



```
      +⌿0=(⍳N)∘.|⍳N
1  2  2  3  2  4  2  4  3  4  2  6  2  4  4  5  2  6  2  6

      2≤+⌿0=(⍳N)∘.|⍳N
1 1 1 0 1 0 1 0 0 0 1 0 1 0 0 0 1 0 1 0
```

STATISTIQUES ELEMENTAIRES.

Soit V un vecteur consignant les résultats d'une mesure sur une certaine population. La fonction suivante permet de connaître la répartition de cette population dans N classes de même largeur.

```
      ∇CLASSE[⎕]∇
    ∇ Z←U CLASSE N
[1]   Z←+⌿(⍳N)∘.=+⌿(Z+0,(((⌈⌿U)-Z←⌊⌿U)÷N)×⍳N-1)∘.≤U
    ∇
```

Prenons par exemple un vecteur de 100 éléments compris entre 1 et 10 pris aléatoirement :

```
      U←?100⍴10
      U
4  8  8  7  2  9  10  3  10  8  7  6  3  3  2  10  1  3  2  4  8  4  9
   10  4  8  1  7  3  1  10  9  6  2  2  3  7  9  10  4  6  3  2
   10  7  8  8  4  10  7  8  5  10  4  2  6  6  2  8  5  4  5  8  6
   2  5  10  8  6  9  5  4  5  5  10  2  4  2  9  6  3  9  7  4  6
   3  8  10  2  4  9  2  8  8  10  3  2  7  1  2
```

Le découpage en cinq classes donne la répartition :

```
      U CLASSE 5
19  22  15  22  21
```

Le découpage en 10 classes affine cette répartition :

```
      U CLASSE 10
4  15  10  12  7  9  8  14  8  13
```

C'est en fait le nombre d'éléments de V ayant les valeurs de 1 à 10 :

```
      +/V=5
7
      +/V=7
8
```

Si M est une matrice (n,p) dont chaque colonne contient les résultats des mesures de n variables pour une expérience, la fonction MATCOV calcul la matrice des variances et covariances de ces variables.

```
      ∇MATCOV[⎕]∇
    ∇ R←MATCOV M
[1]   R←((M+.×⍉M)÷J)-V∘.×V←(+/M)÷J←¯1↑⍴M
    ∇

      M
31.1      34.3      56.7      80.2      93.9      110.7
23        25.8      43.1      62.1      73.6      87.1
67        71        73        79        88        91
33.87     38.51999  77.28     117.36    139.3499  168.3299

      MATCOV M
877.9542  705.9229  254.1308  1485.469
705.9229  567.669   204.8076  1194.3
254.1308  204.8076  77.47168  429.126
1485.469  1194.3    429.126   2513.558
```

Si on suppose que la dernière variable est fonction linéaire des autres on peut calculer les coefficients de la regression :

```
      ∇RESULT[⎕]∇
    ∇ R←RESULT M
[1]   R←0 ¯1↓MATCOV M
[2]   M←¯1 0↓R∘.,R[1↑⍴R;]
[3]   R←(⍉M)+.×V
    ∇

      V←RESULT M
      V
1.375  .515625  ¯.2792969
      V+.×(¯1 0↓M)
35.90898  40.63555  79.79726  120.2308  142.4843  171.7074
```

Remarque : L'opérateur ∘ sert de séparateur d'expressions (cf. chapitre traitant des extensions).

CADRAGE DE TEXTE :

Jusqu'à maintenant nous n'avons manipulé que des données numériques, mais la plupart des opérateurs d'APL s'appliquent également aux chaînes de caractères. Donnons pour exemple cette fonction qui cadre un texte à droite et à gauche pour une longueur de ligne donnée. Pour un résultat correct le texte doit commencer par un caractère non blanc et il faut avant l'appel supprimer tous les blancs non significatifs.

```
      ∇TEXTE[⎕]∇
    ∇ M←N TEXTE V;O;C;H;I;A;B;Q;L;W;Z
[1]   O←'AEIOUY';C←'BCDFGHJKLMNPQRSTVWXZ';H←'LMNRS'∘.=W←' '∘Z←1∘N←N-W←4
[2]   →10×⍳(I←N)>⍴V
[3]   →6×⍳(A∊' ,;:.?-')∨V[I+1]=' '∘A←V[I]∘B←V[I+1]∘L←''
[4]   Q←((∨/V[I+⍳((I↓V)⍳' ')-1]∊O)=1)∧(I≠N)∧B∊C∘L←'-'
[5]   →3×⍳B∘I←I-B←~(Q=1)∧(A=B)∨(V[I+2]∊O)∧(A∊O)∨A∊H
[6]   →9×⍳(0=A)∨0=B←(+/Q)-A←+/×\⌽Q←' '=L←N↑(I↑V),L
[7]   Q←Q∧((B|A)+B×0=B|A)≥+\Q
[8]   L←N↑(,L∘L[;B]←Q∘L[;1]←1∘L←L=' ')/,L←⍉((B←1+⌈A÷B),N)⍴L
[9]   →2∘Z←0∘N←N+W×Z∘V←(V[1+I]=' ')⌽I↓V∘M←M,L
[10]  M←((1+L(⍴M)÷N),N)⍴M,V,(N←N+W×Z)⍴' '
    ∇
```

REMARQUE EXPLICATIVE.

Les lignes 3 à 6 construisent dans la variable Q la ligne à insérer à la suite avec coupure éventuelle du dernier mot.

Les lignes 7 et 8 distribuent les blancs de fin de ligne au milieu de celle-ci pour le cadrage à droite.

```
      TX
ACTUELLEMENT, NOUS CONNAISSONS DEUX GRANDES CATEGORIES D'APPLICATIONS, E
      XIGEANT L'ALLOCATION DYNAMIQUE DE LA MEMOIRE: LA SIMULATION ET LES
       SYSTEMES D'ORDINATEURS ADMETTANT UN HAUT DEGRE DE SIMULTANEITE (T
      EMPS REEL ET MULTIPROGRAMMATION NOTAMMENT). CES APPLICATIONS SONT
      D'AILLEURS ETROITEMENT APPARENTEES ET DIFFERENT SURTOUT PAR LEUR C
      LIENTELE: INFORMATICIENS ICI, ET LA, UTILISATEURS QUELCONQUES. D'A
      UTRES APPLICATIONS EXISTENT DEJA, ET L'OUTIL FERA PEUT-ETRE, PLUS
      TARD, NAITRE LE BESOIN.
      45 TEXTE TX
    ACTUELLEMENT, NOUS CONNAISSONS DEUX GRAN-
DES CATEGORIES D'APPLICATIONS, EXIGEANT L'AL-
LOCATION  DYNAMIQUE DE LA MEMOIRE: LA SIMULA-
TION  ET LES SYSTEMES D'ORDINATEURS ADMETTANT
UN  HAUT DEGRE DE SIMULTANEITE (TEMPS REEL ET
MULTIPROGRAMMATION  NOTAMMENT).  CES APPLICA-
TIONS SONT D'AILLEURS ETROITEMENT APPARENTEES
ET  DIFFERENT SURTOUT PAR LEUR CLIENTELE: IN-
FORMATICIENS  ICI,  ET LA, UTILISATEURS QUEL-
CONQUES. D'AUTRES APPLICATIONS EXISTENT DEJA,
ET  L'OUTIL FERA PEUT-ETRE, PLUS TARD, NAITRE
LE BESOIN.
      70 TEXTE TX
    ACTUELLEMENT,  NOUS CONNAISSONS DEUX GRANDES CATEGORIES D'APPLICA-
TIONS, EXIGEANT L'ALLOCATION DYNAMIQUE DE LA MEMOIRE: LA SIMULATION ET
LES  SYSTEMES  D'ORDINATEURS  ADMETTANT  UN HAUT DEGRE DE SIMULTANEITE
(TEMPS  REEL  ET  MULTIPROGRAMMATION NOTAMMENT). CES APPLICATIONS SONT
D'AILLEURS  ETROITEMENT  APPARENTEES  ET  DIFFERENT  SURTOUT  PAR LEUR
CLIENTELE:  INFORMATICIENS  ICI,  ET  LA,  UTILISATEURS  QUELCONQUES.
D'AUTRES  APPLICATIONS  EXISTENT DEJA, ET L'OUTIL FERA PEUT-ETRE, PLUS
TARD, NAITRE LE BESOIN.
```

CALCUL DE POLYTOPES :

Cette application est une adaptation des fonctions proposées par P. Abrams et W.M. Mc Keeman dans le numéro 36 (Juillet/Août 1970) de la revue Cegos Informatique : "Computer display of the derived polytopes".

Les polytopes réguliers sont la généralisation dans l'espace n-dimensions des polyèdres tels que cube tétraèdre, octaèdre. Des transformations géométriques simples appliquées à ces polytopes réguliers génèrent des figures semi-régulières appelées "polytopes dérivés. Nous n'entrerons pas dans les détails des algorithmes. Les fonctions ci-dessous calculent deux séries de polytopes dérivés : dans un espace à 3 dimensions cela correspond aux polytopes dérivés du cube (measureplot) et de l'octaèdre(crossplot).

```
      ∇CROSSPLOT[⎕]∇
    ∇ PTS←N CROSSPLOT G;U;P
[1]   N←1 NVERTS U←(2*¯.5)×0,+\((N-1)⍴2)⊤G
[2]   1 PERMS⍴U⍺PTS←(N,⍴U)⍴0⍺P←0
[3]   PTS←CPOLYT
    ∇

      ∇MEASUREPLOT[⎕]∇
    ∇ PTS←N MEASUREPLOT G;U;P
[1]   N←1 NVERTS U←.5+(2*¯.5)×0,+\((N-1)⍴2)⊤G
[2]   1 PERMS⍴U⍺PTS←(N,⍴U)⍴0⍺P←0
[3]   PTS←CPOLYT
    ∇
```

On calcule les coordonnées de tous les sommets du polytope en permutant, avec ou sans changement de signe, les coordonnées d'un point standard (fonctions NVERTS, PERMS). Il suffit de relier ensuite les sommets distants d'une unité(fonction CPOLYT).

```
      ∇NVERTS[⎕]∇
    ∇ R←T NVERTS U;I;K;N
[1]   N←⍴U⍺K←I←R←1
[2]  NSL1:→(U[I]=U[I+1])/NSL2
[3]   K←0⍺N←N-K⍺R←R×K!N
[4]  NSL2:→NSL1×⍳(⍴U)>I←I+1⍺K←K+1
[5]   R←R×2*T×+/U≠0
    ∇

      ∇PERMS[⎕]∇
    ∇ S PERMS N;I
[1]   →(N≠0)/P1⍺I←1
[2]   →0⍺PTS[P;]←U⍺P←P+1
[3]  P1:→(I>U⍳U[I])/P2
[4]   S PERMS N-1⍺U[I,N]←U[N,I]
[5]   →(~S∧U[N]≠0)/P3
[6]   S PERMS N-1⍺U[N]←-U[N]
[7]   U[N]←-U[N]
[8]  P3:U[N,I]←U[I,N]
[9]  P2:→(N≥I←I+1)/P1
    ∇
```



```
      ∇CPOLYT[⎕]∇
    ∇ T←CPOLYT;J;I;EDGES;C
[1]   EDGES←⍳0⍺I←1⍺T←(0,1+¯1↑⍴PTS)⍴0
[2]   J←I↓,9.999996E¯5>|¯1+(+/(PTS-(⍴PTS)⍴PTS[I;])*2)*.5
[3]   →2×⍳(1↑⍴PTS)>I←I+1⍺EDGES←EDGES,,⍉(2,(⍴J),(⍴J)⍴I),J←I↓,J/⍳⍴J
[4]  E1:T←T,[1]PTS[EDGES[J,I←1+J←(0≠EDGES)⍳1];],0 1
[5]  E2:→E3×⍳(⍴EDGES)<J←EDGES⍳C⍺EDGES[I,J]←0⍺C←EDGES[I]
[6]   →E2⍺T←T,[1]PTS[EDGES[I←J+¯1+2×2|J];],1
[7]  E3:→E1×0≠+/EDGES
    ∇
```

On obtient finalement une matrice dont chaque ligne contient les coordonnées d'un sommet et un indicateur disant si le point est relié au précédent (1) ou non relié (0).

On remarque que la fonction PERMS est récursive.

```
      3 MEASUREPLOT 0
 ¯.5   .5   .5   0
 ¯.5   ¯.5  .5   1
  .5   ¯.5  .5   1
 .5    .5   .5   1
 .5    .5   ¯.5  1
 ¯.5   .5   ¯.5  1
 ¯.5   .5   .5   1
 .5    ¯.5  ¯.5  0
 .5    .5   ¯.5  1
 ¯.5   ¯.5  ¯.5  0
 ¯.5   .5   ¯.5  1
 ¯.5   ¯.5  ¯.5  1
  .5   ¯.5  ¯.5  0
  .5   .5   .5   1
```

Toutefois, si ces calculs peuvent permettre une étude des propriétés géométriques des figures obtenues ils ne deviennent vraiment intéressant que dans la mesure où il est possible de visualiser une projection plane de celles-ci. Nous verrons dans le chapitre suivant comment il est possible de réaliser cette image sur l'écran d'une console tektronix grâce aux opérateurs graphiques implémentés dans APL/1600.

DEVELOPPEMENTS GRAPHIQUES

****

L'utilisation des possibilités graphiques d'un terminal comme le Tektronix 4013 ou 4015 ouvre à APL un champ d'applications très intéressant, dans la mesure où un certain nombre de primitives de manipulation graphique sont inclues dans le langage.

Ces primitives se présentent dans l'implémentation actuelle sous forme de deux opérateurs : l'un permet la lecture des coordonnées d'un point désigné par l'utilisateur, l'autre effectue l'affichage d'une image.

Après une brève description de ces nouveaux opérateurs, nous verrons dans ce chapitre comment ces outils simples permettent de visualiser des résultats, construire des images, tracer des courbes. Nous terminerons en décrivant les premiers essais d'une application développée en collaboration avec le centre de technologie éducative de l'Ecole des Mines de Saint-Etienne, pour la conception et la réalisation de schémas animés entrant dans la composition d'audio-visuels.

Définition des opérateurs graphiques.

Les entrées-sorties graphiques se font au moyen de l'opérateur I-beam (I) d'opérande 16. La forme monadique permet d'entrer les coordonnées de point repèré par la "croisée" de l'écran

Forme monadique.

Elle s'écrit :

```
I16
```

Il est possible de déplacer à l'aide de deux molettes la "croisée" qui apparaît alors sur l'écran. Quand l'utilisateur a repéré le point voulu il tape un caractère quelconque sur le clavier la " croisée " disparaît alors et le résultat est un vecteur numérique de trois éléments : le code ASCII du caractère suivi des coordonnées x et y du point avec :

$$0 \leqslant x \leqslant 1023$$

$$0 \leqslant y \leqslant 800$$

ce qui correspond au quadrillage visible de l'écran.

Nous donnerons dans le chapitre suivant un exemple d'utilisation de cet opérateur.

Forme dyadique :

Elle s'écrit :

```
16IA
```

et son action dépend de la structure de A. Dans tous les cas le résultat est A.

a) A est vide : effacement de l'écran.

b) A est un vecteur . A doit alors avoir un nombre pair d'éléments tels que

$$0 \leqslant A(i) \leqslant 1023$$

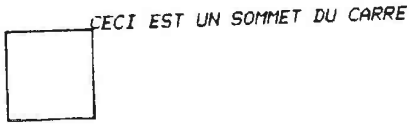
Ce vecteur est considéré comme une suite de coordonnées x, y. L'opérateur affiche la ligne continue joignant ces points.

Pour tracer un carré on peut écrire :

```
A←16I100 100 200 100 200 200 100 200 100 100
```

Si A n'a que deux éléments il y a positionnement au point indiqué mais celui-ci n'apparaît pas ce qui permet, par exemple, d'écrire une chaîne de caractères à l'endroit voulu de l'écran en écrivant :

```
'CECI EST UN SOMMET DU CARRE'⍎16I200 200
```



c) A est une matrice de deux colonnes. Chaque ligne représente un point et l'écriture :

```
16IA
```

est équivalente à

```
16I,A
```


d) A est une matrice d'au moins trois colonnes. Les deux premières colonnes sont considérées comme une suite de coordonnées la dernière comme un indicateur de tracé, les autres sont ignorées.

Si l'indicateur vaut 0 il y a positionnement au point x, y sans trace, s'il vaut 1 la droite joignant ce point au précédent est tracée.

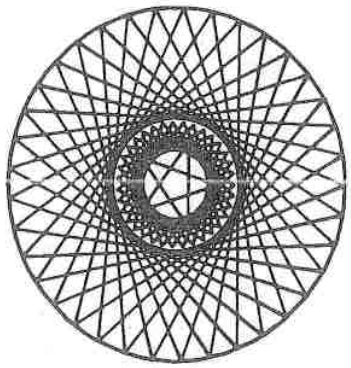
Cette dernière forme permet de définir des images complexes discontinues. Les coordonnées doivent toujours être positives et inférieures à 1024. Sachant que si y est supérieur à 850, le point n'apparaît pas.

Traçons par exemple deux lignes parallèles :

```
      A←4 3⍴100 100 0 400 100 1 200 200 0 500 200 1
      A
100  100  0
400  100  1
200  200  0
500  200  1
      A←16IA
```

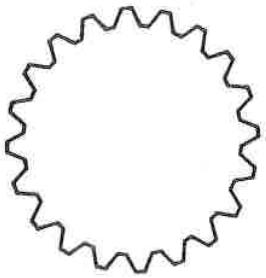
Des fonctions simples peuvent permettre de tracer des figures plus complexes :

```
      0 700 600 200 POL 100 1
      0 700 600 200 POL 37 23
      0 700 600 50  POL  5  3
      0 700 600 70  POL  35 11
```



```
      ∇POL[⎕]∇
    ∇ C POL N
[1]   U←(C[1]×(○2)÷360)+((○2)×N[2])×(0,⍳N[1])÷N[1]
[2]   U←16I⌈,⍉(2,(⍴U))⍴(C[2]+C[4]×20U),C[3]+C[4]×10U
    ∇
```

```
      ∇RDT[⎕]∇
    ∇ R←RA RDT ND
[1]   R←((ND←ND+1),1)⍴0(0,⍳ND←4×ND)+2×ND⍟RA←RA+0 0 0 0 ¯20 ¯20 ¯20 ¯20
[2]   R[1;3]←0⍟R←(((ND,2)⍴RA)×(20R),10R),1
    ∇

      R←150 RDT 20
      R[;1 2]←R[;1 2]+400
      0⍴16⍳R
```



Tracé des polytopes.

Nous pouvons donc maintenant projeter les polytopes calculés dans le chapitre précédent. Toutefois les calculs donnant une figure symétrique par rapport à l'origine et dont les arêtes ont une longueur unité, il faut effectuer des rotations, une affinité et une translation pour pouvoir afficher cette figure de manière correcte.

On peut, par exemple générer une matrice de rotations par

```
      ∇GENROT[⎕]∇
    ∇ M←GENROT A;I;J;T;U;M1
[1]   M←(⍳3)∘.=⍳3⍟I←1
[2]   G1:J←I+1
[3]   G2:M←M+.×M1⍟M1[I,J;I,J]←2 2⍴T,U,(-U←10A),T←20A⍟M1←(⍳3)∘.=⍳3
[4]    →(3≥J←J+1)/G2
[5]    →(3>I←I+1)/G1
    ∇

      MRT←GENROT0+11
```

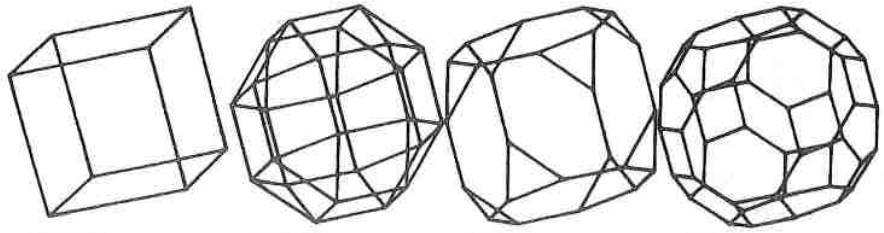
Puis écrire une fonction qui affiche une image :

```
      ∇AFF[⎕]∇
    ∇ R←U AFF PTS
[1]   R←30 ¯30+2↑U⍟16⍳U APPLY((0 ¯1↓PTS)+.×MRT),PTS[;4]
    ∇

      ∇APPLY[⎕]∇
    ∇ R←L APPLY Y;X;P
[1]   R←Y[;1 2]
[2]   R←⌈(R-X)×L[3]÷(⌈/P)-X←L/P←,R
[3]   R[;1]←R[;1]+L[1]⍟R[;2]←R[;2]+L[2]
[4]   R←R,Y[;4]
    ∇
```

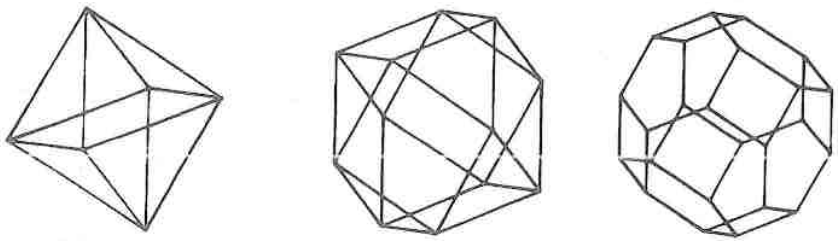

Puis une fonction donnant plusieurs exemples mis en page : Les variables Cij représentent les polytopes dérivés calculés par i crossplot j et Mij ceux calculés par i measureplot j (cf. chapitre précédent).

```
      ∇DISP[⎕]∇
    ∇ DISP
[1]   'C(3,1)'⍟16⍳0 100 250 AFF C31⍟16⍳0
[2]   'C(3,2)'⍟16⍳380 100 250 AFF C32
[3]   'C(3,3)'⍟16⍳720 100 250 AFF C33
[4]   'M(3,0)'⍟16⍳0 500 250 AFF M30
[5]   'M(3,1)'⍟16⍳255 500 250 AFF M31
[6]   'M(3,2)'⍟16⍳510 500 250 AFF M32
[7]   'M(3,3)'⍟16⍳765 500 250 AFF M33
    ∇
```

L'exécution de DISP donne :



M(3,0) M(3,1) M(3,2) M(3,3)



C(3,1) C(3,2) C(3,3)

Tracé de courbes

Autre application plus immédiate : tracer une ou plusieurs courbes en les cadrant dans l'écran.

La fonction AXES calcule une fois pour toutes une variable permettant de tracer les axes.

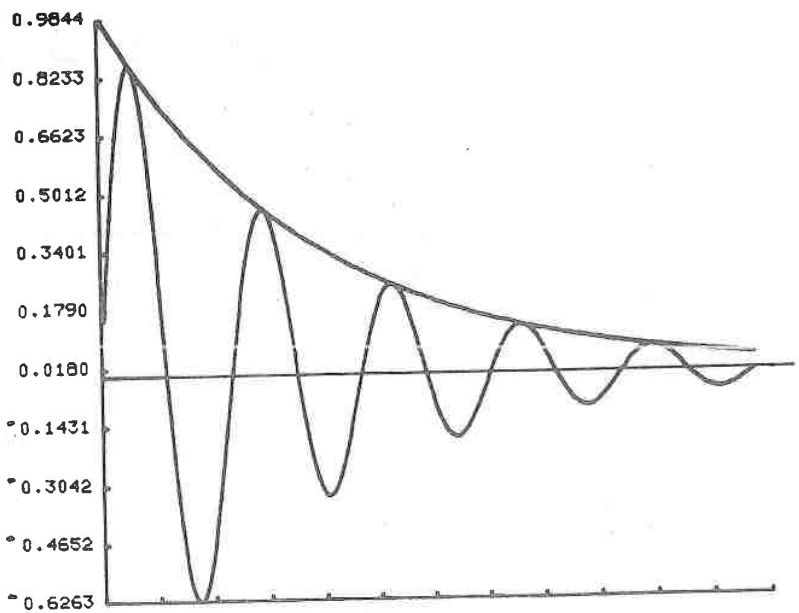
```
      ∇AXES[⎕]∇
    ∇ AXES;X
[1]   X←(⍎[1](33 2⍴70 ¯70)+33 2↑⍎X),[1]X←⍉↖180 110,[1]37 2⍴0 5 0 ¯5 66 0
    ∇
```

La fonction TRACE trace les courbes représentées par les colonnes de la matrice argument. S'il n'y a qu'une courbe à tracer, on peut la laisser sous forme de vecteur.

```
      ∇TRACE[⎕]∇
    ∇ TRACE M;J;X;AY;MIN;MAX;LY
[1]   M←110+M+(MAX←(⌈/⌈/M)+10)+66⍟M←M-MIN←L/L/M←(2↑(⍴M),1)⍴M
[2]   LEGX⍟16I177 75⍟16I⍉⍟16I10⍟LY←¯1↑⍴M⍟AY←1↑⍴M
[3]   (31⍴1 0 0)↖FOR⍟11 1⍴⍎MIN+MAX×¯1+⍳11⍟16I0 810
[4]   →TR×⍳1=×MIN×MIN+MAX×10⍟J←1⍟X←(AY,1)⍴180+(66+⌈AY+12)×⍳AY
[5]   0⍴16I180,MAX,1000,MAX←110+(0-MIN)+MAX+66
[6]  TR:→TR×LY⍳J←J+1⍟0⍴16IX,M[;J]
    ∇
```

L'utilisateur doit définir deux variables globales : FOR qui sert de format pour l'édition de l'échelle de l'axe des ordonnées et LEGX, chaîne de caractère, qui est écrite sous l'axe des abscisses.

```
X←200 1⍴(10×0⍳200)+200
Y←10X
Y←(Y×X),X←⍟-X+10
TRACE Y
```



ESSAI DE TRACE DE COURBE

Construction et animation de schémas.

Nous allons créer un langage de définition de figures qui sera compilé par une fonction APL pour donner une autre fonction APL. Il sera ainsi possible de construire puis d'animer des schémas plus ou moins complexes. Pour ce faire l'utilisateur doit d'abord définir son schéma sous forme d'arbre dans une chaîne de caractère de notre langage de deuxième niveau : les niveaux de l'arbre sont indiqués par le parenthesage, les frères sont séparés par des virgules ; les feuilles sont des identificateurs représentant chacun une composante du schéma. Ainsi dessinons par exemple un piston en définissant la variable MODEL :

```
      MODEL
⎕(ROUE,BRAS,⍥(TRAN TETR)⎕PISTON,⎕CORPS,⎕CORPS)
```

Devant chaque noeud de l'arbre il est possible de mettre un "quad" (⎕) ce qui permettra de mettre en place les différentes parties du schéma en appliquant aux noeuds une matrice de transformation.

L'expression :

```
⍥(TRAN TETR)⎕PISTON
```

indique que l'utilisateur veut appliquer la translation de vecteur TETR sur le piston pour l'animation. TETR est un vecteur et TRAN une fonction calculant la matrice correspondante :

```
      ∇TRAN[⎕]∇
    ∇ R←TRAN X
[1]   R←3 3⍴1 0 0 0 1 0,X,1
    ∇
```

Les deux parties du corps du piston sont des carrés qui resteront fixes ainsi que ROUE qui est un cercle.

BRAS est une fonction APL :

```
      ∇BRAS[⎕]∇
    ∇ R←BRAS
[1]   R←0,[1]2 3⍴(100×20TETA),(100×10TETA),1
[2]   TETR←¯1↑R[3;2 1]←0,R[3;1]+250×20¯10.4×10TETA
[3]   TETR←(TETR-350),0
    ∇
```

Elle calcule le bras du piston ainsi que la translation TETR en fonction de l'angle TETA.

Nous allons d'abord construire un certain nombre de variables de travail et une fonction définissant le schéma à l'aide des deux fonctions suivantes :

```
      ∇DEFINE[⎕]∇
    ∇ X DEFINE Y;U;P;M;Z;I;J;N;OPF;NLF
[1]   IQ←0+(⍳3)∘.=⍳3⍟OPF←'∇',X,'_F['⍟NLF←0
[2]   ⍎'∇',X,'_←',X,'_F∇'
[3]   ⍎X,'←Y'⍟(X,'_')COMP Y←⍎,Y
[4]   U←⍴I←1+J←0⍟Y←Y,' '⍟P←(⍴Y)⍴0⍟M←0 1⍴''
[5]   →7 7 8 8 9 10 10[Z←(Y[J←J+1]='([])⍷⎕⎕')/⍳7]
[6]   →(12×⍳J≥⍴Y),5
[7]   →5⍟I←¯1⍟U←U,I
[8]   →5⍟U←¯1↓U⍟I←¯1↑U
[9]   →5⍟I←I+1
[10]  M←(((1⍴⍴M),N)↑M),[1](N←(⍴Z)[¯1↑⍴M)↑Z←'MS'[1+Z=7],X,'_',1 0⍕U,I
[11]  →5⍟P[J]←1⍴⍴M
[12]  ⍎X,'_N←M'⍟⍎X,'_P←P'
    ∇

      ∇COMP[⎕]∇
    ∇ R←X COMP Y;U;S;I;M;J
[1]   M←X,'←'
[2]   I←1⍟U←,((0=+\((Y='(')∨Y='[')-(Y=')')∨Y=']')×(Y=',',')))/⍳⍴Y
[3]  LPP:S←Y[S[J]+⍳S[J+1]-(S←0,1↓S,1↑⍴Y)[J←1⍴S←I,U]+1]⍟R←''
[4]   →TTQ×⍳'0'≠1⍴S⍟J←1 0⍕I
[5]   →ERR×⍳'('≠1⍴S←1↓S
[6]   R←R,((T←∧\1↓+\(S='(')-S=')')/S),')AP '
[7]   →ERR×⍳')'≠1⍴S←(~T)/S
[8]   S←1↓S
[9]  TTQ:→ENQ×⍳'⎕'≠1⍴S
[10]  R←('M',X,J,' AP '),R⍟S←1↓S⍟⍎'M',X,J,'←IQ'
[11] ENQ:→TEX×⍳'('≠1⍴S
[12]  →ERR×⍳')'≠¯1↑S
[13]  →END⍟R←R,(X,J)COMP 1↓¯1↓S
[14] TEX:→IM×⍳'['≠1⍴S
[15]  →ERR×⍳']'≠¯1↑S
[16]  →END⍟R←R,1↓¯1↓S
[17] IM:→NQU×⍳'⎕'≠1⍴S
[18]  →ERR×⍳1≠⍴S
[19]  →END⍟R←R,'S',X,J⍟⍎'S',X,J,'←1 3⍴0'
[20] NQU:→NDUP×⍳'$'≠1⍴S
[21]  S←(X,J)COMP S←⍎1↓S
[22] NDUP:→END⍟R←R,S
[23] ERR:'SYNTAX PROBLEM: ',S
[24]  ⍎'→'
[25] END:⍎OPF,(⍕NLF),']',M,R,'∇'⍟NLF←NLF+1
[26]  →LPP×⍳(I←I+1)≤1+⍴U⍟M←X,'←',X,'[1]'
[27]  R←X
    ∇

      'TT' DEFINE 'MODEL'

      ∇TT_F[⎕]∇
    ∇ TT_←TT_F
[1]   TT_1←ROUE
[2]   TT_1←TT_1,[1]BRAS
[3]   TT_1←TT_1,[1]MTT_13 AP(TRAN TETR)AP PISTON
[4]   TT_1←TT_1,[1]MTT_14 AP CORPS
[5]   TT_1←TT_1,[1]MTT_15 AP CORPS
[6]   TT_←MTT_1 AP TT_1
    ∇

      TT
⎕(ROUE,BRAS,∘(TRAN TETR)⎕PISTON,⎕CORPS,⎕CORPS)
```



```
      TT_P
1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
      2  0  0  0  0  0  0  3  0  0  0  0  0  0  4  0  0  0  0  0  0

      TT_N
MTT_1
MTT_13
MTT_14
MTT_15
```

La fonction TT-F est définie récursivement dans COMP.

La fonction AP effectue l'application d'une transformation plane, définie pour une matrice(3,3), à une image :

```
      ∇AP[⎕]∇
    ∇ R←T AP M
[1]   R←(0 ¯1↓((0 ¯1↓M),1)+.×T),M[;¯1↑⍴M]
    ∇
```

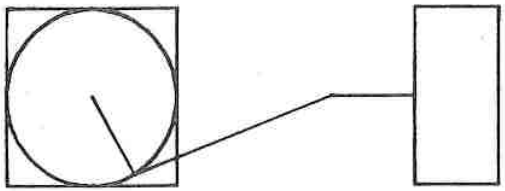
Nous pouvons maintenant utiliser la fonction SHOW pour mettre en place le schéma

```
      ∇SHOW[⎕]∇
    ∇ SHOW X;N;T;I;POS;NM;Q;P;S
[1]   I←16⌶0⌈1023⌊I←⍎X,'_F'⍟16⌶10⍟I←IQ⍟POS←(⍎X,'_P'),0⍟NM←⍎X,'_N'
[2]   ⎕←S←⍎X⍟16⌶0 800
[3]   →3×⍳0=Q←POS[Q]⍟P←P[Q←1⍴(P≠' ')/⍳⍴P←(⍴S)↑⎕]
[4]   →5 0 1 6 7 8 9 10 11 3[⍳'TSDRHI↓↑⎕'⍳P]⍟T←⍎N←,NM[Q;]
[5]   →2⍟⍎N,'←T'⍟T←T+.×TRAN(¯2↑⌶16)-¯2↑⌶16
[6]   →2⍟⍎N,'←T'⍟T←(ROT⎕)+.×T
[7]   →2⍟⍎N,'←T'⍟T←(HOMO⎕)+.×T
[8]   →2⍟⍎N,'←IQ'
[9]   →2⍟I←T
[10]  →2⍟⍎N,'←I'
[11]  →2⍟⍎'S',1↓N,'←T'⍟T←DESSIN
    ∇

      ∇HOMO[⎕]∇
    ∇ R←HOMO X
[1]   R←3 3⍴X[1],0 0 0,X[2],0 0 0 1
    ∇

      ∇ROT[⎕]∇
    ∇ R←ROT X
[1]   R←3 3⍴(2○X),(1○X),0,(-1○X),(2○X),0 0 0 1
    ∇
```

Après avoir tracé l'état actuel du schéma, SHOW écrit la chaîne de définition et se met en attente d'entrée. Il est alors possible d'indiquer des transformations à effectuer sur les noeuds en écrivant au dessous des quads le caractère T pour translation, H pour affinité, R pour rotation, etc...

Ainsi après avoir translaté tout le schéma initialement centré en zéro et mis en place le piston,obtient :

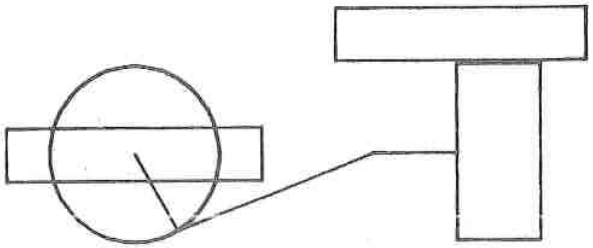
```
⎕(ROUE,BRAS,∘(TRAN TETR)⎕PISTON,⎕CORPS,⎕CORPS)
                                  H
⎕:
      1.5 .3
```



et on indique le produit de deux affinités de rapports 1.5 suivant X et .3 suivant Y sur une partie du corps

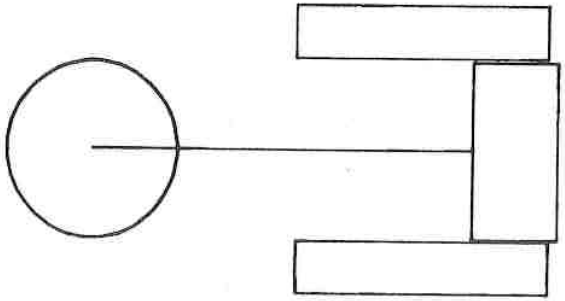
on met en place cette partie après avoir effectué les mêmes affinités sur l'autre partie.

```
⎕(ROUE,BRAS,∘(TRAN TETR)⎕PISTON,⎕CORPS,⎕CORPS)
                                        T
```

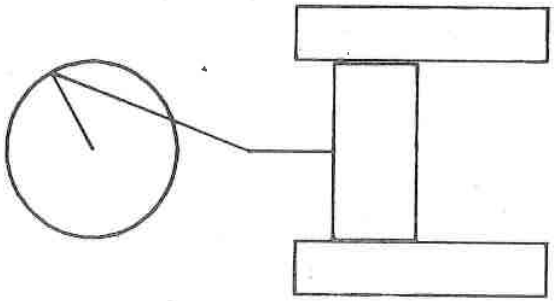


Il ne reste plus qu'à mettre en place la deuxième partie du corps. Les translations sont indiquées par l'utilisateur par deux entrées successives au "cross-hair" comme on le voit dans la ligne 5 de la fonction SHOW. Maintenant, la fonction DESSINE permet d'afficher le schéma pour différentes valeurs de TETA :


```
      ∇DESSINE[⎕]∇
    ∇ DESSINE X
[1]   I←16I0⌈1023⌊⍉X,'_F'
    ∇

      TETA←0

      DESSINE 'TT'
```



```
      TETA←2×○÷3

      DESSINE 'TT'
```



Ceci ne donne qu'un bref aperçu des possibilités de cette application apparemment rudimentaire. Il est notamment possible d'effectuer des rotations sur tout ou partie de l'arbre, de réinitialiser la matrice associée à un noeud ou encore de compléter après coup certaines parties du schéma, symbolisées par le caractère ⍞ .

Pour une utilisation intensive de ces fonctions, il est nécessaire d'optimiser au maximum la partie calcul de l'image. Or ce calcul consiste principalement en une série d'applications de transformations linéaires à des images selon un procédé de multiplication de matrices précédé et suivi de restructurations de matrices. Ce procédé que nous dénommerons "pseudo-multiplication de matrices" est effectué par la fonction AP qui, écrite en APL, prend beaucoup de temps calcul, alors que l'algorithme est très simple. Nous verrons dans le chapitre suivant qu'il a été possible de remédier à cet inconvénient en incluant cette pseudo-multiplication comme un opérateur du langage sous forme de fonction système.

FONCTIONS SYSTEMES ET AUTRES EXTENSIONS

****

Au vu des derniers développements des systèmes APL existant, et pour éviter de faire d'APL/1600 un langage "fermé", les auteurs ont jugé intéressant de faciliter l'écriture et l'intégration d'un nouveau type de fonctions dites "fonctions systèmes". Ces fonctions dont l'utilisation obéit à la syntaxe courante d'APL sont en fait des programmes écrits dans un langage de programmation quelconque et conservés, dans les zones de travail, sous forme de programme objet. Sans alourdir l'interprète elles permettent des extensions faciles et une "personnalisation" de ses possibilités, l'utilisateur ayant ainsi la possibilité de programmer ses propres opérateurs de manière aussi performante que les opérateur du langage, d'accéder aux services d'une gestion de fichiers ou d'utiliser au mieux toutes les options de sa configuration. Nous ne donnerons pas ici un mode d'emploi détaillé permettant l'écriture et l'intégration de ces fonctions, ce qui sortirait du cadre de ce chapitre, mais, après quelques exemples de fonctions systèmes développées pour les besoins de nos applications, nous décrirons plus en détail une fonction générale de formattage dont les spécifications sont proches de celle d'APL/PLUS. Nous terminerons enfin en donnant un aperçu de quelques unes des extensions les plus intéressantes et les plus significatives apportées au langage pour les auteurs, notamment dans l'édition des fonctions et les spécifications de l'opérateur "dequote" ou "execut". (Ω)

QUELQUES FONCTIONS SYSTEMES.

La fonction AP décrite au chapitre précédent peut être réécrite en une cinquantaine d'instructions machine sous la forme d'un programme PL 1600 :

```
IR,NC:=0;
DO; << CALCUL DU RESULTAT
  IF(NC=2)THEN      << ON LAISSE LA DERNIERE COLONNE
    &PFRES(RX):=&PFOP2(IR);
  ELSE  FLTR:=FZ;K:=2;I1,RA:=NC+RA;I2:=IR;
    DO;       << CALCUL D'UN ELEMENT
      IF(K=0)THEN RFL:=F1;
      ELSE RFL:=&PFOP2(I2);
      END;
      FLTR:=RFL*&PFOP1(I1)+FLTR;
      DECR K;EXIT ON(K<0);
      INCR I1;INCR I1;INCR I2;INCR I2;
    END;
    &PFRES(IR):=FLTR;
  END;
  INCR IR;INCR IR;INCR NC;
  IF(NC>2)THEN NC:=0 END;
  EXIT ON(IR=LGRES);
END; << DO
```

Son utilisation divise par deux le temps de calcul d'une image composite par les fonctions du chapitre précédent.

De même pour utiliser les possibilités de tracé incrémental de l'écran tektronix, les auteurs avaient écrit une fonction APL permettant de tracer des lettres et des chiffres de tailles variables. Mais, relativement lente elle nécessitait trois variables globales, dont l'une de taille assez importante contenant la définition de chaque caractère. Sous forme d'une fonction système elle est devenue plus économique en place, et, divise par dix les temps d'exécution.

```
      ∇DEM[⎕]∇
    ∇ DEM
[1]   9 ECRIT 'VOICI UNE SERIE'⍝16I10 700⍝16I10
[2]   8 ECRIT 'DE LETTRES'⍝16I200 550
[3]   7 ECRIT 'QUE VOUS POUVEZ'⍝16I100 400
[4]   6 ECRIT 'UTILISER A VOTRE GUISE'⍝16I0 260
[5]   5 ECRIT 'VOYEZ LE BRICK GEANT QUE'⍝16I50 160
[6]   4 ECRIT 'J''EXAMINE PRES DU WHARF.'⍝16I150 90
[7]   3 ECRIT '0123456789 .:,!?=()'⍝16I300 40
    ∇
```

VOICI UNE SERIE

DE LETTRES

QUE VOUS POUVEZ

UTILISER A VOTRE GUISE

VOYEZ LE BRICK GEANT QUE

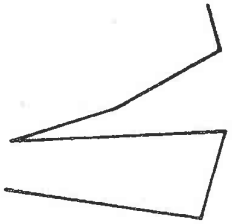
J'EXAMINE PRES DU WHARF.

0123456789 .:,!?=()

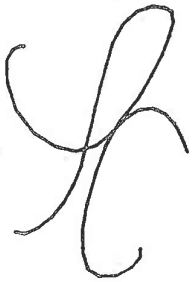
L'inconvénient du mode incrémental est qu'il est pratiquement impossible d'appliquer des transformations simples aux images obtenues. Calculer à partir de la définition incrémentale un caractère traçable avec l'opérateur graphique classique devient vraiment prohibitif alors qu'une fonction système donne un résultat quasi immédiat, permet de faire entrer des phrases dans la définition des images et même d'obtenir plusieurs jeux différents de caractères, par des déformations géométriques des modèles initiaux.

L'utilisation du mode graphique de l'écran révèle rapidement à l'utilisateur les limites des possibilités d'entrées graphiques grâce au "cross-hair" mais la connection d'une tablette augmente considérablement ces possibilités. Encore à ces premiers balbutiements, l'intégration de cette tablette a été très facilement effectuée grâce à une fonction système permettant l'entrée sous tous les modes possibles :

```
      A←GRT 3
        A
680   915  0
890   902  1
928   775  1
807   709  1
683   671  1
935   684  1
908   584  1
677   615  1
```



```
      B←GRT 7

      ''
        'B
55  3
```



Pour terminer ces quelques exemples simples signalons encore une fonction système transformant une fonction APL en matrice de caractère :

```
      M←FCTM 'DEM'

      M
DEM
9 ECRIT 'VOICI UNE SERIE'⍕16I10 700⍕16I10
8 ECRIT 'DE LETTRES'⍕16I200 550
7 ECRIT 'QUE VOUS POUVEZ'⍕16I100 400
6 ECRIT 'UTILISER A VOTRE GUISE'⍕16I0 260
5 ECRIT 'VOYEZ LE BRICK GEANT QUE'⍕16I50 160
4 ECRIT 'J''EXAMINE PRES DU WHARF.'⍕16I150 90
3 ECRIT '0123456789 .:,↓?=()'⍕16I300 40

      ⍴M
8  45
```

UNE FONCTION DE SORTIES FORMATTEES.

Les possibilités d'édition d'APL sont assez pauvres malgré l'opérateur dyadique enquote.

```
      X
1.25  100  1.4468

      12 0 ⍕ X
          1          100            1

      12 3 ⍕ X
       1.250      100.000        1.447

      10 ¯2 ⍕ X
   1.2E00     1.0E02     1.4E00
```

Aussi l'utilisation d'une fonction de formattage telle que FMT est elle bienvenue pour toutes sortes d'applications dès que l'utilisateur désire des sorties un peu sophistiquées.

Si nous reprenons l'exemple du premier chapitre de la fonction calculant les coefficients du binôme il est certain que la forme suivante donnée au résultat est plus agréable :

Exemple : triangle de Pascal.

```
(-⍟3×¯1⍳⍳N+1)⍟ 'BI6' FMT ⍟(0,⍳N)∘.!0,⍳N←10

                                  1
                               1     1
                            1     2     1
                         1     3     3     1
                      1     4     6     4     1
                   1     5    10    10     5     1
                1     6    15    20    15     6     1
             1     7    21    35    35    21     7     1
          1     8    28    56    70    56    28     8     1
       1     9    36    84   126   126    84    36     9     1
    1    10    45   120   210   252   210   120    45    10     1
```


Examinons de plus près les possibilités de cette fonction.

La fonction système FMT, pour l'édition formattée de tableaux de données APL, est écrite en PL1600. Elle effectue, dans une large mesure, des appels à certains sous-programmes de l'interprète.

Elle est dyadique, et admet comme opérande gauche une chaîne de caractères représentant un format, comme opérande droit une donnée APL, ou une liste de données séparées par des point- virgules, et situées entre parenthèses.

Elle peut travailler en deux modes :

- dans le cas où elle est le dernier "opérateur" de la ligne APL, les sorties sur le terminal se font ligne par ligne, ce qui permet d'éviter les cas de saturation mémoire dus à un résultat trop gros.

- dans le cas où elle est suivie d'autres opérateurs, il y a génération effective d'une matrice résultat.

Les spécifications de format sont très voisines de celles de fortran : formats I, E et F pour les valeurs numériques, A pour les valeurs de type caractère, insertion de blancs et de chaînes de caractères :

```
      M←3 3⍴⍳9
      '⍕I3,3X,E9.3,⎕CHAINE⎕,F5.2' FMT M
  1    2.00E0  CHAINE 3.00
  4    5.00E0  CHAINE 6.00
  7    8.00E0  CHAINE 9.00

      ⍴⎕←'⍕I3' FMT M
  1  2  3
  4  5  6
  7  8  9
3  9
```

De plus, l'utilisation de spécifieurs permettant d'agir sur la zone éditée, offre un attrait supplémentaire ;

```
      X←¯12345.678 ¯123 ¯0.35 0 15
      '⍕I10' FMT X
    ¯12346
      ¯123
```

```
      'ZCF⍬+⍬M⍬-⍬I15' FMT X
-00,000,012,346
-00,000,000,123
+00,000,000,000
+00,000,000,000
+00,000,000,015

      'N⍬          ⍬CI20' FMT X
  -12,346
     -123
                    0
                    0
                   15

      'R⍬∘ ⍬M⍬-$⍬F⍬$⍬CF15.2' FMT X
∘ ∘ -$12,345.68
∘ ∘ ∘ ∘-$123.00
∘ ∘ ∘ ∘ ∘-$0.35
∘ ∘ ∘ ∘ ∘ $0.00
∘ ∘ ∘ ∘ ∘$15.00
```

Il est également possible d'éditer simultanément plusieurs données :

```
      N←4 6⍴'NUTS  BOLTS SCREWSTOTAL '
      P←76 142 37
      C←.05 .1 .06
      Q←8 4 2
      F←'6A,⍬| ⍬,I3,F7.2,I5,F10.2'
      F FMT (N;P;C;Q;E,+/E←C×Q)
NUTS  |  76   0.05    8       0.40
BOLTS | 142   0.10    4       0.40
SCREWS|  37   0.06    2       0.12
TOTAL |                       0.92
```

Enfin, les dépassements de capacité, ou l'emploi d'un format incorrect, se traduisent par des astérisques dans le champs d'édition, les autres erreurs étant signalées de la manière habituelle :

```
      'AC' FMT 12
**
**
      'I5' FMT 1E12
*****
      'MAUVAIS' FMT 13
SYNTAX ERROR
      'MAUVAIS' FMT13
                   ^
      1 2 3 FMT 1 2 3
DOMAIN ERROR
      1 2 3 FMT 1 2 3
      ^
```


QUELQUES ASPECTS DES EXTENSIONS.

Tout au long des chapitres précédents l'observateur attentif et averti se sera posé quelques questions sur des opérateurs ou des expressions inhabituelles.

Ainsi l'opérateur ⍺ remplace des expressionstelles que

A,0⍴B ou A AFTER B avec :

```
      ∇R←A AFTER B
[1]   R←A
[2]   ∇
```

L'opérateur a d'ailleurs la même définition que cette fonction.

Plus importantes sont les extensions de l'opérateur dequote (⍎) qui permet aussi d'excécuter des commandes systèmes ou des définitions de fonction.

Exemple :

```
      ⍎')VARS'
M       X

      ⍎')FNS'
AFTER   ALPH    AP      CLOSE   DEM     ECRIT   FCTM    FMT
GRPH    GRT     INIT    PCHEND  PCHN    PCHS    PCHSTR  TY

      ⍎'∇AFTER[⍬]∇'
    ∇ R←A AFTER B
[1]   R←A
    ∇
```

Ceci permet une gestion dynamique des zones de travail de même que la génération automatique et la modification dynamique des fonctions comme nous l'avons vu au chapitre précédent.

Contrairement aux implémentations sur grosses machines ou l'utilisateur ne dispose généralement que de son terminal pour effectuer les entrées-sorties APL/1600 permet d'utiliser tous les périphériques de la configuration par un simple changement de l'unité d'entrée ou de sortie. En cas d'erreur le système recommute automatiquement sur le terminal

```
      14⍎N
```

permet de commuter les entrées sur l'unité symbolique ou physique de numéro N. N est un entier positif ou nul inférieur à 255.

De même : `15⍎N`

commute les sorties sur l'unité N.

Dans les deux cas, le résultat est le numéro de l'ancienne unité affectée.

L'éditeur de texte standard d'APL est relativement peu puissant. Sans doute encore insuffisantes, les deux extensions suivantes en augmentent toutefois agréablement les possibilités.

Tout d'abord, il est possible d'insérer dans une fonction tout ou partie d'une autre fonction par la commande :

[<NOM DE FONCTION> (N) (P)]

n et p sont des numéros de lignes facultatifs.

Examinons les différentes possibilités de cette commande :

[<NOM DE FONCTION>]	insère tout le corps de la fonction
[<NOM DE FONCTION> N]	insère la ligne n de la fonction
[<NOM DE FONCTION> N P]	insère les lignes n à p de la fonction

n peut être nul auquel cas on insère également l'en-tête.

L'insertion se fait à partir de la ligne courante en gardant l'incrément courant.

Ex : [2.1] [F] insère toute la fonction F à partir de la ligne 2.1 en incrémentant les numéros de lignes de 0.1.

```
[2.1] [NVERTS]
[2.1] N←⍴V⍺K←I←R←1
[2.2] NSL1:→(V[I]=V[I+1])/NSL2
[2.3] R←0⍺N←N-K⍺K←R×K!N
[2.4] NSL2:→NSL1×1(⍴V)>I←I+1⍺K←R←1
[2.5] R←R×EXT×+/V≠0
[2.6]
```

```
[2.8] [NVERTS 0 10]
[2.8] R←T NVERTS V;I;R;N
[2.9] N←⍴V⍺K←I←R←1
[3]   NSL1:→(V[I]=V[I+1])/NSL2
[3.1] R←0⍺N←N-K⍺K←R×K!N
[3.2] NSL2:→NSL1×1(⍴V)>I←I+1⍺K←R←1
[3.3] R←R×EXT×+/V≠0
[3.4]
```

```
[3.5] [NVERTS 1]
[3.5] N←⍴V⍺K←I←R←1
[3.6]
```

Les lignes soulignées sont les commandes frappées par l'utilisateur.

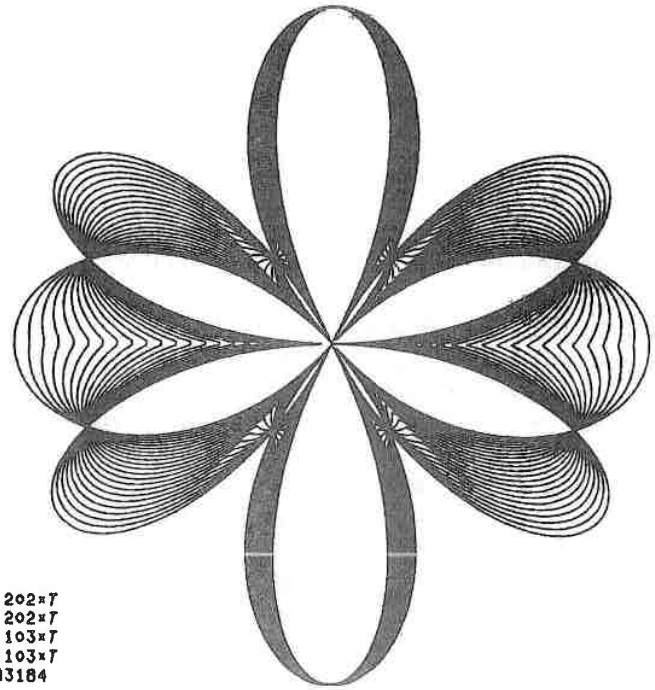
Ensuite, il est possible de définir une chaîne de caractère qui sera ensuite insérée en mode édition de ligne, à chaque rencontre du caractère delta :

```
      ⍫SEG500 [⎕]
    ⍫ SEG500
[1]   R5←1I←0
[2]   →2×1I<⍴I+1⍺R5←R5,0=B4+I
[3]   [←R5←R5-1 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0
    ⍫
[4]   ⍺←NBSWAP
[4]   [2⎕10]
[2]   →2×1I<⍴I+1⍺R5←R5,0=B4+I
                  //⍺//⍺

[2]   →2×1I<⍴I+1⍺NBSWAP←NBSWAP,0=B4+I
[3]
```

Cette définition reste valable jusqu'à nouvelle définition ou fermeture de la fonction.

```
[4]   [3⎕10]
[3]   [←R5←R5-1 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0
        //⍺//⍺
[3]   [←NBSWAP←NBSWAP-1 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0

[4]   ⍺←NUMPROC
[4]   [1⎕10]
[1]   R5←1I←0
        /⍺
[1]   R5←1NUMPROC←0
```

PARTE II



X1=(20T)×|202×T
Y1=(10T)×|202×T
X2=(20T)×|103×T
Y2=(10T)×|103×T
T ∈0 6.283184

CHAPITRE 1

CONCEPTION DU CALCULATEUR APL

Dans sa première partie, ce chapitre a pour but de mettre en lumière les caractéristiques générales du langage APL, et leurs implications au niveau de la réalisation.

Il étudie dans sa seconde partie quels choix restent possibles dans le cadre de celle-ci, et donne les grandes lignes de l'implémentation.

A) ASPECTS GENERAUX.

Ecrire un interprète APL revient à concevoir un calculateur dont le langage machine est APL, et à le simuler sur une machine existante.

Il est aisé de définir, à la lumière des caractéristiques de ce langage, quelles sont les fonctions que doit remplir un tel calculateur APL : principalement, il doit permettre l'accès et la manipulation des objets du langage, et assurer la sécurité du traitement.

A partir de ces spécifications très générales, nous déduirons les caractéristiques communes à tous les interprètes APL, et après une rapide étude des implémentations déjà réalisées, nous donnerons les buts principaux de notre implémentation.

I - LES OBJETS APL :

Les objets manipulés par le calculateur APL sont en fait très simples : la puissance du langage est en effet due plus à la puissance des opérateurs qu'à la complexité des structures de données manipulables.

On distingue deux types d'objets :

- les objets de type opérande sur lesquels s'appliquent les opérateurs du langage et les fonctions.
- les objets de type fonction qui permettent de créer des opérateurs complexes.

. Les opérandes :

La seule structure reconnue par l'interprète est le tableau, qui peut être de rang 0 (opérande scalaire) ou plus (vecteur, matrice .....). Cependant, APL permet à l'utilisateur de manipuler facilement les tableaux booléens, numériques ou de type caractère.

L'accès aux valeurs des éléments est séquentiel ou aléatoire aussi bien en lecture qu'en écriture.

Le calculateur doit être également capable de créer et de détruire dynamiquement des opérandes.

. Les fonctions :

L'utilisateur peut définir et utiliser des fonctions, suite ordonnées d'expressions APL qui, sauf branchement explicite, sont exécutées en séquence. Ces fonctions "utilisateur", interprétées à chaque exécution sont peu performantes par rapport à un programme compilé. Il nous a donc paru utile de créer un second type de fonction, dite fonction système ou fonction intégrée dont le maniement est identique pour l'utilisateur mais qui sont en fait des programmes écrits dans un langage quelconque, compilés en dehors de l'interprète, et exécutés, en langage machine, sous son contrôle.

Les fonctions utilisateurs n'ont qu'un point d'entrée et peuvent posséder un espace local* c'est-à-dire un certain nombre de noms dont les valeurs associées n'existeront que pendant l'exécution de la fonction. Font implicitement partie de cet espace local le résultat et le ou les arguments (tous optionnels). Les arguments sont donc passés par valeur.

* On appelle espace l'ensemble des noms et des valeurs associées accessibles à un instant donné : la mémoire du calculateur APL étant en fait un espace de noms plutôt qu'un espace d'adresse. Un nom peut représenter aussi bien une variable qu'une fonction.

Ces fonctions nécessitent d'une part un éditeur de texte permettant de les créer et de les modifier, d'autre part un mécanisme permettant de masquer et de démasquer l'espace global et d'enchaîner l'exécution des lignes.

Le maniement des fonctions système est beaucoup plus simple car elles n'ont pas d'espace local et s'exécutent sous le contrôle de l'unité centrale du calculateur hôte lui-même. Les seules actions possibles pour le calculateur APL sont l'insertion dans le système, la destruction et le lancement de l'exécution.

Signalons pour terminer que les types et les valeurs des objets associés aux identificateurs ne sont pas fixes, la seule contrainte étant qu'un identificateur doit transiter par l'état indéfini pour passer du type opérande au type fonction et réciproquement.

II - LE DYNAMISME DE L'EXECUTION.

Sauf pour les fonctions, il n'y a pas en APL de déclaration de type, de rang ou de dimensions : tous les tableaux sont crées dynamiquement, et chaque nouvelle affectation à une variable peut modifier tout ou partie de ses caractéristiques pré-existantes.

Par ailleurs,

La programmation récursive est permise en APL, et il est possible de déclarer explicitement une variable locale. Ceci implique donc la possibilité de gérer une pile des appels, et plus généralement, une pile des contextes associés à chacun d'eux. Le même problème est soulevé par l'existence de l'opérateur execute, qui lui aussi peut s'utiliser récursivement.

Tous ces problèmes se compliquent encore du fait que, la localité étant une propriété attachée au nom, ce nom peut désigner indifféremment une fonction ou une variable : il est possible qu'un nom désigne une variable à un certain niveau, et une fonction à un autre niveau : le mécanisme de sauvegarde et de restauration s'applique donc à n'importe quel objet APL.

De ces quelques considérations et de tout ce que nous avons dit précédemment sur le langage nous pouvons tirer les caractéristiques générales communes à tous les interprétes APL.

a) Le code machine :

Celui-ci va rester très proche du langage APL, les seules modifications consistant en :

- passage d'un nom externe à une adresse.
- transformation des constantes, pour leur donner une forme interne normalisée.

b) La mémoire du calculateur :

Alors que, sur les calculateurs habituels, l'unité de travail est le mot mémoire, sur le calculateur APL, où une instruction machine s'applique à une donnée toute entière, l'unité d'allocation sera le segment.

c) La gestion de la mémoire :

Du fait du dynamisme du langage, il n'est pas possible d'utiliser des méthodes statiques d'allocation. Ce sera la charge du calculateur que de déterminer, avant chaque instruction machine, quelles seront les caractéristiques du résultat. Il sera donc nécessaire, à ce niveau là, de faire appel à une gestion dynamique de la mémoire, dont les primitives seront par exemple :

- obtenir un segment de taille désirée.
- libérer un segment.

d) Les données :

Alors que sur une machine classique, les données sont indifférenciées (c'est-à-dire que le contenu d'un mot mémoire peut être une instruction machine, une adresse, un nombre, un caractère, etc...), sur le calculateur APL au contraire, toutes les caractéristiques de chaque segment sont connues du calculateur (1). Ceci devra lui permettre entre autre de choisir la représentation optimale pour une donnée, de manière transparente à l'utilisateur (le terme d'optimalité étant bien sûr relatif aux critères de l'implémentation : gain de place, gain de temps, etc....).

---

(1) Dans cet ordre d'idée, on peut signaler la présence, sur les ordinateurs Philips de la série P1000, d'instructions machine opérant sur une pile dont chaque élément occupe 64 bits. Les deux derniers bits opérandes indiquent si l'on a affaire à un réel, un entier, un logique, ou une adresse, les instructions machine s'adaptent aux types des données sur lesquelles elle opèrent.

e) Le système d'interruption :

Le terme dans le cas présent ne s'applique pas aux entrées-sorties, mais à la fonction de traitement des erreurs. On retrouvera alors les diagnostiques habituels, mais plus complets, car le calculateur a une meilleure conscience des caractéristiques des données.

Le système devra également tenir compte des mal-fonctions du calculateur ("Erreur Système").

De plus, il sera conçu dans l'optique d'une reprise du traitement après intervention de l'utilisateur.

On voit donc que ce système d'interruption devra être beaucoup plus sophistiqué que ce qui existe sur les calculateurs classiques.

III - LES APPROCHES POSSIBLES :

Nous ne chercherons pas ici à résumer les nombreux travaux menés sur APL à travers le monde. Nous rappellerons simplement les deux grands types possibles d'approche à une telle réalisation.

1) La réalisation d'un maximum de fonctions au niveau matériel :

Dans ce domaine, la réalisation la plus spectaculaire est le MCM70 [6] qui est en fait un micro ordinateur ayant l'aspect d'un calculateur de poche. Il interprète un sous-ensemble du langage APL, grâce à une mémoire de micro-programmes, d'une capacité de 18 K octets.

Il comporte également une mémoire programmable, pouvant contenir les données et les fonctions de l'utilisateur, d'une taille maximale de 16 K octets.

De plus, il est possible de connecter un disque à cartouche, ce qui assure les fonctions de sauvegarde et de restauration de zones de travail.

Il est également intéressant de citer l'approche de A-HASSIT, J.W. LAGESCHULTE et L.E. LYON [C6], qui consiste en la micro-programmation d'un ordinateur IBM 360/25.

Dans cette réalisation, une partie des fonctions sont micro-programmées. Ce sont :

- gestion de la table des symboles
- procédure d'entrées et de sorties de caractères
- conduite de l'interprétation
- certains opérateurs, pour des éléments scalaires et vectoriels.

L'ensemble de ces micro-programmes définit donc une machine APL restreinte. La taille de ces micro-programmes est de 16 K octets.

Les autres fonctions sont réalisées par un superviseur, écrit dans cet APL restreint.

L'intérêt de cette réalisation est démontré par des mesures de temps d'exécution : par exemple, dans le cas de la somme de 2 vecteurs, l'expression APL : "A+B" est plus rapide à exécuter, que la suite d'instructions effectuant le même traitement au moyen d'une boucle sur un 360/25 ordinaire, dès que le nombre d'éléments dépasse une dizaine.

Que peut-on conclure de ces deux exemples de réalisations, dont le principe de base est une intervention au niveau matériel ?

Tout d'abord, qu'elles démontrent la faisabilité et l'intérêt d'un calculateur APL.

Ensuite, débordant le cadre du langage APL, elles montrent qu'il est possible de concevoir des ordinateurs dotés d'un langage machine de haut niveau, et que celui-ci, loin de diminuer la capacité de traitement de l'ordinateur, en multiplie au contraire les possibilités. Cet aspect est d'ailleurs souligné par HASSIT, LAGESCHULTE et LYON, puisque le titre de leur article est :

" IMPLEMENTATION OF A HIGH LEVEL LANGUAGE MACHINE ".

2) La réalisation des fonctions à un niveau purement logiciel :

C'est, jusqu'à présent, l'approche la plus fréquemment utilisée, probablement parce qu'elle est à la portée de tout chercheur disposant d'un ordinateur.

A ce stade là, les justifications des implémenteurs sont d'ordres divers :

a) Réalisation d'un système APL complet, destiné à une exploitation effective. Ces implantations ne font, en général, l'objet d'aucune innovation.

b) Application de nouveau algorithmes, pour l'optimisation de l'interprétation ou la représentation des structures de données. L'expérience montre que de tels interprètes sont rarement utilisés, même lorsque la réalisation a été poussée jusqu'à un stade où le produit est devenu opérationnel.

La raison en est, le plus souvent, qu'un seul aspect du problème a été correctement abordé, (celui qui relevait de ces nouveaux algorithmes), et que la réalisation pèche par les autres côtés, ce qui interdit son exploitation.

c) Résolution des problèmes inhérents au langage selon les méthodes habituelles, mais de nouveaux problèmes sont introduits par des conditions inhabituelles de réalisation : par exemple, celle-ci a lieu sur un petit ordinateur, ou une gestion des fichiers est incluse dans l'interprète, ou encore de nouveaux opérateurs sont introduits dans le langage, etc...

C'est à cette dernière approche que se rattache notre réalisation, notre but étant de réaliser un interprète sur une petite machine ce qui suppose remplir les trois conditions suivantes :

- un langage complet
- un système fiable
- un système extensible réduisant les inconvénients dus à la faible taille du calculateur.

IV - NOTRE APPROCHE :

Nous allons maintenant reprendre les différents points mis en évidence dans le paragraphe a, en indiquant, à la lumière des expériences décrites au paragraphe b, les différents choix effectués.

Les descriptions des méthodes employées resteront bien évidemment fort concises, les algorithmes intéressants étant détaillés dans les chapitres suivants.

a) Un langage APL complet :

Les spécifications du langage ont été choisies identiques à celle d'APL/360. Bien que la comparaison ait déjà été effectuée dans la première partie, rappelons que les opérateurs récemment inclus dans le langage, tel execute, ⍎ , enquote, ⍕ , l'inversion de matrice, ⌹, ont été implémentés.

De plus, quelques extensions intéressantes ont été implémentées (Cf. 1ère partie).

b) Un système fiable :

Il va de soi que nous n'essayerons pas de démontrer que notre interprêteur APL est correct, la preuve d'un tel programme étant hors de la portée de quiconque, du moins dans l'état actuel de la science informatique.

Du moins avons-nous essayé de formuler un certain nombre d'assertions sur les différents algorithmes utilisés, ainsi que sur le fonctionnement général de l'interprète, et disposé dans les modules des séquences d'instructions permettant de vérifier que les données sont bien conformes aux assertions.

De plus, les principaux algorithmes ont été simulés et testés en APL/1130, sous une forme voisine de leur formulation PL1600, ce qui nous a permis d'obtenir rapidement une première version opérationnelle de l'interprète.

c) Un système extensible :

L'existence et la survie d'un système comme APL impliquent un dialogue constant avec l'utilisateur. En effet, une fois les contraintes d'implémentation définies par les concepteurs du système, un certain nombre de choix sont possibles. Or, chaque utilisateur aimerait voir les développements ultérieurs du constructeur s'orienter dans la direction que lui, utilisateur, préconise. C'est pourquoi, lorsque l'on pose aux utilisateurs la question " que manque-t-il dans notre système APL ", les réponses sont si contradictoires. En fin de compte, pour le constructeur, il n'est pas possible d'effectuer tous les développements demandés par les utilisateurs, de même qu'il n'est pas possible dans un langage d'inclure tous les aspects possibles de la programmation, sous peine de créer une indomptable hydre de Lerne (le triste exemple de PL/1 le montre bien). C'est ainsi qu'au cours de ces cinq dernières années, l'évolution d'APL s'est limitée à l'introduction dans le langage de deux nouveaux opérateurs (l'inversion de matrices et la résolution de systèmes linéaires, ainsi que le formattage de résultats), et de la possibilité de faire appel à une gestion de fichiers.

Le problème se pose en ces termes : comment augmenter les services rendus à l'utilisateur d'un système APL ? Dans le cas d'un langage classique, disons FORTRAN, la réponse est simple : la fourniture de sous-programmes bien écrits et efficaces, voire même d'applications entièrement réalisées, permet de résoudre les problèmes des clients. Dans le cas d'un système APL, le problème est légèrement différent : ce n'est pas en bourrant les WS des utilisateurs de fonctions fournies par le constructeur que l'on résoudra leurs problèmes. Car n'importe quel utilisateur APL averti peut écrire en APL des fonctions tout autant astucieuses que celles du constructeur, et certainement mieux adaptées à ses travaux. Non, le problème est le suivant : APL n'est pas compétitif lorsque l'on doit exécuter une fonction de manière répétitive, autrement dit lorsqu'un algorithme simple n'est pas disponible sous forme d'une primitive du langage.

A ce niveau, le problème (faire en sorte qu'un algorithme exprimé en APL puisse être exécuté sur un ordinateur à une vitesse acceptable) admet deux solutions : puisque seul le code machine est efficace, soit générer du code machine à partir d'une fonction (compilation), soit permettre à l'utilisateur d'exécuter sous APL du code machine. La première solution, quoique plus séduisante, pose des problèmes qui sont loin d'être résolus. La seconde solution était, et de loin, la seule susceptible de fournir une réponse rapide et efficace. De plus, cette seconde solution a l'avantage de répondre à la question : combien va coûter à un utilisateur l'implémentation d'un nouvel opérateur qu'il n'utilisera pas ? La question devient : l'utilisateur est-il disposé à travailler pour "cabler" une fonction qu'il utilise fréquemment ?

Or, la réponse est assez simple : une fois la fonction opérationnelle en APL, la transcription dans un langage comme PL1600 présente assez peu de difficultés, même en respectant l'interface avec l'interprète. En revanche, l'utilisateur dispose maintenant d'une fonction aussi performante qu'un opérateur implémenté directement dans l'interprète. Cette fonction peut être conservée dans une zone de travail inactive tant que l'on ne s'en sert pas, ne pénalisant nullement les autres utilisateurs.

L'introduction d'une telle fonction sous le système APL ne présente aucune difficulté : l'utilisateur dispose d'une nouvelle commande qui lui permet de charger un programme objet dont les liens ont été édités, et de le reconfigurer ainsi sous forme d'une fonction système.

B) L'INTERPRETE.

Après une rapide présentation de différentes méthodes d'optimisation applicables à un interprète APL, et une justification des choix effectués dans ce domaine, nous donnerons une description simplifiée de "l'unité centrale" de notre calculateur APL.

La fin de cette seconde partie sera consacrée à certains aspects spécifiques de notre réalisation, la traduction du texte source et l'utilisation d'une mémoire hiérarchisée à deux niveaux.

I - LES APPROCHES POSSIBLES :

Le langage APL permet beaucoup de concision dans la rédaction des programmes, mais nécessite souvent la manipulation de structures de données importantes pour un résultat simple. Ainsi dans le calcul de 3↑A+B, A et B étant des vecteurs, l'utilisateur ne s'intéresse qu'aux trois premières valeurs du vecteur A+B. Il semble donc intéressant d'optimiser le calcul d'une expression en limitant les opérations effectuées.

Dans cette direction il faut citer l'approche de P.S. Abrams exploitant les propriétés mathématiques du langage pour simplifier les calculs. Sa méthode repose sur deux processus :

- l'évaluation différée
- la modification automatique des expressions pour obtenir des formes standards réduisant notamment une suite quelconque d'opérateurs de sélection (prendre, laisser, transposer ...) à une composition de trois opérateurs de base.

Telle qu'il la décrit la méthode suppose des restrictions dans la définition des opérateurs : par exemple, elle n'est possible que pour une indication par des scalaires ou j-vecteurs (*). De plus, elle n'est valable que si l'effort (temps calcul, complexité des algorithmes) pour effectuer les modifications puis l'évaluation est plus faible que l'évaluation directe : ceci n'est vrai que si une partie importante du processus est microprogrammée.

S'inspirant de cette méthode J.J. GIRARDOT a mis au point et testé en APL sur un calculateur IBM 1130 une méthode d'interprétation en arbre minimisant le nombre d'opérations à effectuer, ainsi que le nombre d'accès mémoire requis.

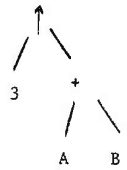
---

* Un J-vecteur est une suite entière croissante ou décroissante par pas de 1 de longueur et d'origine quelconque.

Les trois principes de bases de cette méthode sont :

a) tout élément est calculé une fois au maximum.
b) si une donnée est telle que l'on utilisera une fois au plus chacun de ses éléments , on n'allouera pas d'espace mémoire pour cette donnée et chaque élément sera calculé., utilisé puis détruit aussitôt.
c) si une donnée est telle que l'on utilisera plusieurs fois chacun de ses éléments, alors ceux-ci seront calculés et un emplacement mémoire leur sera alloué.

Explicitons la méthode en reprenant l'exemple précédent 3 ↑ A+B dont la représentation en arbre est :



Le résultat d'une expression est déterminé par :

- son type (numérique, caractère...)
- son rang
- ses dimensions
- sa valeur.

Considérons maintenant que chaque noeud de l'arbre est un opérateur qui peut poser des questions à ses fils dans l'arbre.

L'opérateur "prend" (↑) veut déterminer les caractéristiques du résultat.

a) calcul du rang. Le rang est celui de l'expression de droite. "Prend" s'adresse donc à "plus".
   - "plus" constate que ses fils sont des feuilles et fournit aussitôt le résultat (rang) demandé.
b) Calcul des dimensions. Les dimensions dépendent de la valeur de l'expression de gauche. "Prend" en demande le calcul qui donne pour résultat 3.
c) Calcul du résultat. Chaque opérateur effectue une transformation linéaire des indices de l'élément demandé du résultat et/ou calcule sa valeur.

Partant de la racine chaque noeud va interroger ses fils pour déterminer le type, le rang et les dimensions de son résultat.

Pour calculer ensuite la valeur, il suffit de déterminer la transformation linéaire à effectuer sur les coordonnées d'un élément des résultats. Cette fonction est la composition des transformations effectuées par chaque opérateur.

<u>Exemple</u> : 3 ↑ X f(u) = u

2 ⌽ X f(u,v) = v,u

La simulation en APL de cette machine optimisée et d'une machine "naïve" toutes deux acdeptant les opérateurs binaires : +,-,x,÷,↑,↓,⌽ et ⌽ a donné lieu à des essais très satisfaisants. Ainsi le calcul de 2 2 ↑ A+B ou A et B sont des matrices 4,4 est environ trois fois plus rapide sur la machine optimisée que celui de A+B sur la machine naïve.

Toutefois, le problème se complique considérablement si on étend la méthode au langage complet.

Il devient notamment nécessaire de rendre la machine "auto-adaptative" à la complexité de l'expression de manière à ne pas pénaliser trop lourdement les calculs simples pratiquement non optimisables.

Très intéressante du point de vue théorique ou pour des réalisations expérimentales, ces méthodes nous ont donc semblé peu praticables dans le cadre de notre implémentation ou plus exactement auraient nécessité des études hors de notre propos initial.

Une dernière approche, beaucoup plus pratique, consiste à détecter des "idiomes" c'est-à-dire des expressions simples souvent utilisées comme "I ← I+1", "ρρ", "ρ," et de les traiter comme un seul opérateur. Ceci revient à "compiler" en quelque sorte certaines parties d'une expression APL pour réduire les manipulations sur les données et donc les temps de calcul.

Se pose alors un problème de choix des idiomes et d'équilibre entre leur nombre et leur complexité d'une part, et d'autre part les avantages obtenus et la plus ou moins grande lourdeur du mécanisme.

Pour des raisons de simplicité de mise en oeuvre et de taille des algorithmes nous avons donc choisi d'implémenter une machine "naïve" dont nous allons maintenant présenter les grandes lignes.

II - <u>LA MACHINE NAIVE</u> :

La machine d'exécution a en entrée un programme qui est une expression APL codée, résultat de l'analyse lexicale contenant des références à des objets contenus soit dans le programme lui-même (constantes) soit dans la table des symboles. (Cf. paragraphe suivant).

Au cours de l'interprétation la machine va créer des objets temporaires, résultats de calculs intermédiaires, dont la durée de vie n'excèdera jamais le temps d'exécution de l'expression. Lorsqu'ils disparaissent les descripteurs eux-mêmes doivent disparaître contrairement aux variables de l'utilisateur. La machine d'exécution dispose pour cela d'une table de descripteurs temporaires qui évolue comme une pile et dont chaque entrée a la taille d'un descripteur.

Un analyseur parcourt le code exécutable et décode les unités syntaxiques les unes après les autres, créant pour les constantes un descripteur temporaires pour homogénéiser les traitements.

Il donne en sortie le type de l'unité : opérateur, séparateur, opérande et sa valeur qui est soit l'opérateur soit une référence au descripteur de l'objet APL (temporaire ou non).

Une dernière structure de données est utilisée alors par la machine d'exécution : une pile de doubles mots où vont éventuellement être rangées les sorties de l'analyseur jusqu'à ce qu'un opérateur ait toutes les données nécessaires à son traitement.

Les décisions sont prises grâce à une table de décisions dont les entrées sont le type de l'unité sommet de pile et le type de la dernière unité décodée par l'analyseur.

Lorsqu'un traitement quelconque est exécuté la pile est vidée de toutes les données nécessaires et le sommet devient, à la sortie, le résultat de ce traitement.

Autour de ce mécanisme simple se greffent des procédures de contrôle traitant notamment l'initialisation et la fin de l'interprétation, l'enchaînement des lignes des fonctions mais surtout l'appel et le retour de ces fonctions. Ces dernières procédures constituent ce que nous appellerons le mécanisme de changement de contexte. En effet, au niveau de la machine d'exécution elle-même l'appel d'une fonction revient à abandonner l'exécution du programme en cours pour exécuter une ou plusieurs lignes de fonctions puis reprendre l'exécution de ce programme. Au niveau supérieur nous avons vu que les fonctions pouvaient avoir des variables locales et donc qu'il fallait masquer pendant leur exécution un certain nombre de variables. Il s'agit donc de sauvegarder à la fois le contexte interne et une partie du contexte de la machine d'exécution.

Le niveau de récursivité étant indéfini il faut mettre en place une gestion de pile "infinie" capable de sauvegarder et de restaurer le contexte de la machine d'exécution et utilisant la gestion de mémoire. Ce contexte comporte notamment le programme en cours d'exécution et son pointeur et les différentes tables utilisée pile d'exécution et table des descripteurs temporaires. (Cf. chapitre suivant).

III - LE CODE EXECUTABLE :

a) Structure du code intermédiaire :

APL autorise des opérations entre scalaires, vecteurs, tableaux, etc... Les dimensions et le type même d'un opérande peuvent changer au cours de l'interprétation d'une manière imprévisible. Ces possibilités interdisent l'utilisation d'un compilateur pour générer du code efficace. Le langage intermédiaire est donc choisi très proche du langage Source. Soit la ligne APL suivante :

$$R \leftarrow A + 2$$

Il est possible de codifier cette ligne, en remplaçant les noms par une référence à la table des symboles, les constantes par leur représentation interne. Mais il est difficile d'aller plus loin. Ainsi, si l'on veut utiliser comme code interne une forme polonaise, on aura par exemple :

$$2\ A\ \underset{\substack{\downarrow \\ \text{dyadique}}}{+}\ R \leftarrow$$

Malheureusement, s'il s'avère qu'à l'exécution A est une fonction, la forme polonaise de la ligne devient :

$$2\ \underset{\substack{\downarrow \\ \text{nonadique}}}{+}\ A\ R \leftarrow$$

et l'on voit que la première forme donnera des résultats incorrects à l'exécution (1).

Le langage intermédiaire choisi est donc très proche de la représentation externe. Les noms sont remplacés par leur numéro dans la table des symboles, les constantes par leur représentation interne, les opérateurs par leur code caractère. Ainsi la ligne ci-dessus sera codifiée de la manière suivante :

$$2 + A \leftarrow R \quad \text{fin de ligne}$$

Un autre avantage de cette représentation voisine du langage source est qu'il est facile de reconstituer la ligne introduite par l'utilisateur. Ceci est nécessaire lorsque l'on désire éditer une fonction ou indiquer une erreur.

---

(1) Il est à remarquer qu'une notation polonaise impose en outre d'avoir des notations différentes pour représenter les formes monadiques et dyadiques d'un même opérateur.

Une ligne APL est, en fait, codifiée sous forme d'une liste d'unités syntaxiques, suivie de la table des constantes de la ligne. L'intérêt de cette table est que l'on peut alors représenter les constantes au moyen d'unité de taille fixe.

Les unités syntaxiques peuvent correspondre à des opérateurs ou séparateurs, des identificateurs, des constantes ou des commentaires. Les opérateurs et séparateurs sont codifiés sur un octet. Ils sont représentés par leur code caractère interne, qui a été choisi de telle sorte que le bit 0 soit à 0. Les autres unités syntaxiques sont codifiées sur 1 mot. Le bit 0 est à 1 pour indiquer la longueur.

Les bits 1 et 2 indiquent la nature de l'unité :

00 : non utilisé
01 : identificateur
10 : constante
11 : commentaire.

Les bits 3 à 15 ont la signification suivante :

- identificateur : numéro dans la table des symboles
- constante ou commentaire : pointeur dans la table des constantes de la ligne.

Dans une même ligne, des constantes identiques ne sont pas dupliquées, ce qui diminue en général la taille de la ligne codée. Une constante est précédée d'un descripteur indiquant son type et son nombre d'éléments. Un élément de type caractère occupe un octet. Un élément de type numérique est représenté sous forme flottante et occupe deux mots.

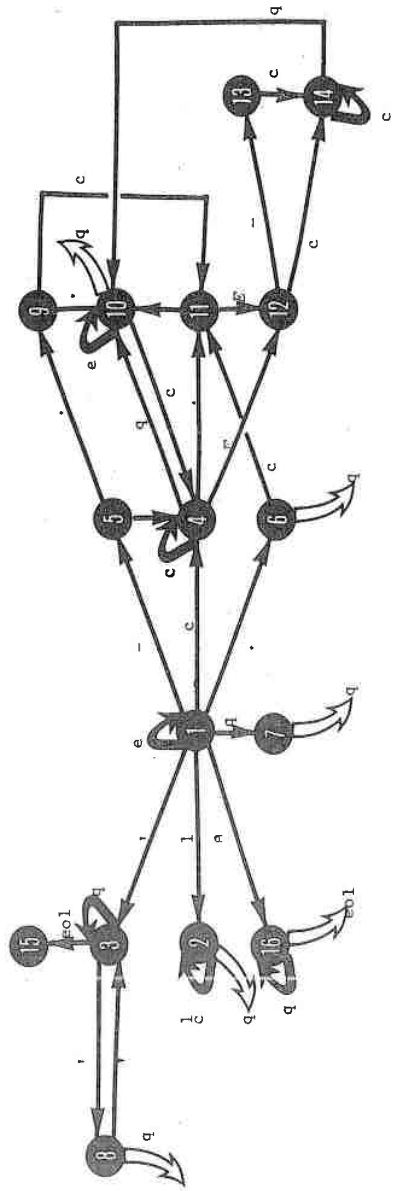
b) Traduction ou analyse lexicographique :

La reconnaissance des unités syntaxiques nécessaires à la génération du code exécutable est effectuée grâce à un automate d'état fini.

Chaque transition de l'automate est représentée par six valeurs :

- l'état actuel
- l'état futur
- les caractères limites pour passer à cet état
- le type de l'unité syntaxique en sortie correspondant } éventuellement 0
- le numéro de la routine sémantique à effectuer } éventuellement 0

AUTOMATE D'ANALYSE LEXICOGRAPHIQUE



eol	fin de ligne
e	espace
c	chiffre de 0 à 9
l	lettre, lettre soulignée, chiffre souligné
E	notation exponentielle
-	signe negatif pour les constantes
a	commentaire
'	apostrophe
.	point decimal ou operateur
q	caractère quelconque

L'automate reconnaît 5 types d'unité syntaxique :

- identificateur
- opérateur
- constante numérique
- constante caractère
- commentaire

Les constantes sont calculées au fur et à mesure de l'analyse.

En sortie l'automate donne le type d'unité syntaxique et sa valeur : longueur, suite des valeurs numériques ou chaîne de caractères.

Dans le cas d'un identificateur le générateur du code exécutable fait appel à la procédure de gestion de la table des symboles qui lui donne pour résultat le numéro correspondant au symbole qui sert de nom interne après l'avoir éventuellement introduit dans la table des identificateurs. Ce nom interne sert d'index danx la table des descripteurs utilisés au cours de l'interprétation.

IV - LA CONCEPTION DE LA GESTION DE LA MEMOIRE.

La petite taille de l'ordinateur nécessite l'utilisation d'une mémoire secondaire en l'occurence un disque pour gérer les données de l'interprète. Il y a donc deux niveaux de gestion :

- le niveau mémoire centrale
- le niveau disque .

En théorie seule deux primitives sont nécessaires :

-"obtenir de la place"
-"liberer de la place"

ce qui conduit à 2 classes de blocs en mémoire : les blocs libres et les blocs occupés.

En fait, les 2 niveaux de mémoire entraînent la nécessité de créer un nouveau type de bloc et deux autres primitives : l'interprète a besoin d'être sûr qu'une donnée va rester en mémoire un certain temps, cette donnée va donc être mise hors classe : on dit qu'elle est activée. Mais à ce moment sa place ne peut-être réutilisée il faut donc que l'interprète signale qu'il ne l'utilise plus. D'ou les deux primitives :

- " activer "
- " désactiver "

Pour limiter les transferts mémoire centrale-mémoire secondaire il est souhaitable d'indiquer pour les blocs occupés mais non activés et donc utilisables s'ils ont ou non une copie sur disque. Comme le gestionnaire n'a aucun moyen de contrôler si une donnée est modifiée ou non par l'interprète, celui-ci dispose d'une dernière primitive - "activer pour modifier".

En résumé, les blocs-mémoires peuvent prendre quatre états :

- libres
- occupés sans copie sur mémoire secondaire
- occupés avec copie sur mémoire secondaire
- activés

Et cinq primitives permettent à l'interprète de faire passer un segment de données d'un état à un autre.

- obtenir de la place
- libérer de la place
- activer
- désactiver
- activer pour modifier.

La gestion du disque, beaucoup plus simple, est complètement transparente à l'interprète sauf pour certaines commandes systèmes.

Le gestionnaire de mémoire dispose sur disque d'une zone pouvant atteindre 64 K octets. En fait, quand nous disons la taille de la zone de travail est 64 K octets nous ne considérons que la taille de la zone disque.

Mais cette zone est allouée par "increment" ou bloc de 4 K octet.

Une allocation en un seul bloc nécessiterait une réorganisation du disque par compactage sinon à chaque opération,du moins lors d'une impossibilité de trouver de la place, ce qui, compte tenu de la première remarque et du dynamisme du langage risque d'être fréquent surtout si, comme c'est le cas actuellement, les zones de travail actives sont gérées dans le même espace disque que les zones de travail inactives. Or cette opération nécessitant beaucoup d'échanges avec le disque est coûteux en temps. La solution d'une allocation par incrément permet une gestion beaucoup plus souple et facile du disque de sauvegarde.

Quelle est la justification de cette méthode ?

Les deux hypothèses de bases sont :

- la zone de stockage est limitée
- les zones de travail sont rarement pleines

Il est donc intéressant lors d'une sauvegarde de récupérer l'espace non effectivement utilisé d'où le découpage en incrément.

La méthode a deux inconvénients :

- taux de remplissage moindre
- compactage automatique très difficile.

Ceci du fait que les incréments ne sont pas obligatoirement contigüs.

Le taux de remplissage varie en fait avec la taille moyenne des segments de données. Tant que celle-ci reste faible (quelques secteurs) il est satisfaisant.

Compte-tenu des remarques précédentes il serait intéressant d'étudier plus précisément sur des exemplesréels l'utilisation du disque de sauvegarde. D'ores et déjà, il semble que la première hypothèse (zone de stockage limitée) ne soit plus valable dans la mesure ou l'utilisateur dispose d 'unitésde disque à cartouches interchangeables et de grandes capacités.

V - LE FONCTIONNEMENT :

L'interprète APL 1600 travaille en deux modes : exécution et définition.

En mode exécution, toute phrase APL introduite au terminal est aussitôt analysée, puis exécutée. Les erreurs éventuelles sont signalées.

En mode définition, l'usager peut enregistrer, modifier ou éditer une suite de phrases APL. Aucune exécution n'a lieu à ce moment là. Le texte introduit est conservé pour utilisation ultérieure. En revenant au mode exécution, l'utilisateur peut exécuter cette suite d'instructions, dite FONCTION.

LE MODE TERMINAL :

Une fois la ligne introduite par l'utilisateur, il faut renvoyer soit un message d'erreur, soit le résultat.

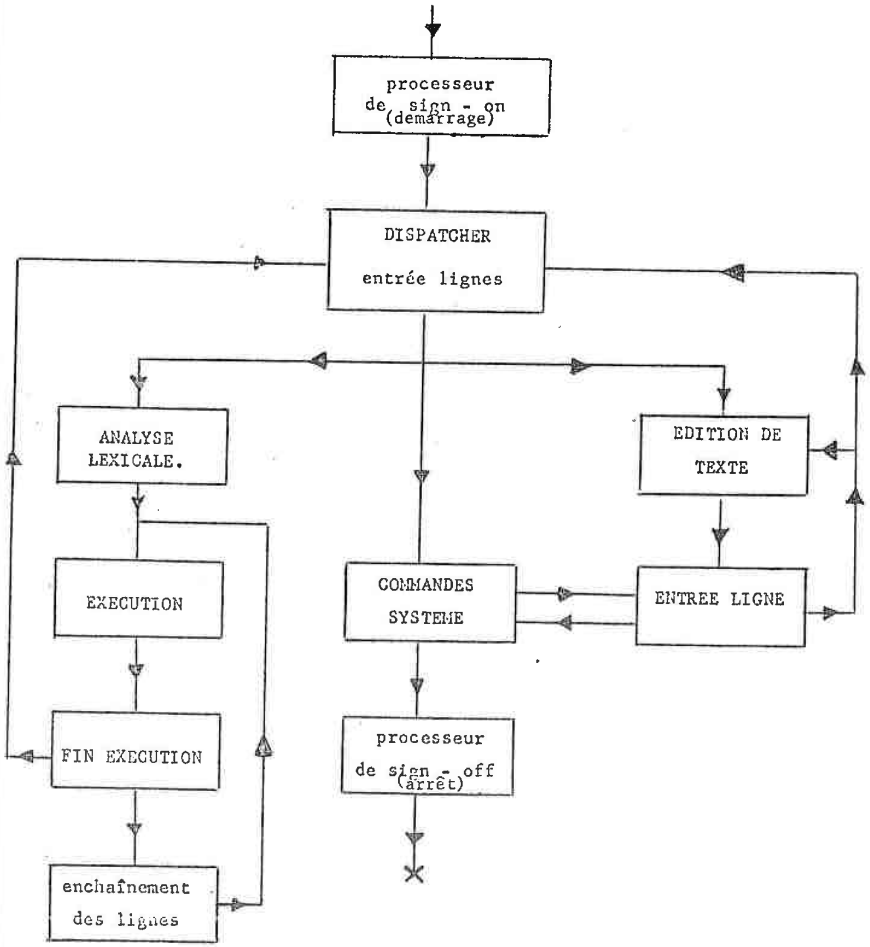
Voyons de quelle manière s'enchaînent les opérations.

La ligne introduite par l'utilisateur est transcodée du "code terminal" en "code caractères interne", puis le résultat est envoyé à l'analyseur lexicographique, qui génère du "code exécutable" grâce à un automate d'état fini. A ce niveau là, les identificateurs ont été remplacés par une référence à la table des symboles, les constantes par des références à des descripteurs créés en fin de ligne. Un certain nombre d'erreurs ont déjà été détectées et éliminées : caractère illégal (ou, plus exactement, superposition illégale de caractères), constante invalide, ligne trop longue, table des symboles saturée.

Le "code exécutable" sert d'entrée à la "machine d'exécution", qui travaille avec une pile (la pile d'exécution), et une table permettant de stocker des renseignements sur les résultats intermédiaires (table des descripteurs temporaires). Un analyseur, parcourt le code exécutable et fournit une à une à la machine d'exécution les unités syntaxiques. Une table de décision permet de choisir la procédure à utiliser. Cette décision est prise en fonction du type de l'unité située au sommet de la pile, et du type de l'unité en entrée. Les décisions peuvent être les suivantes :

- empiler l'unité qui arrive
- exécuter un opérateur
- effectuer l'appel d'une fonction de l'utilisateur
- ... etc

La procédure de fin d'exécution vide éventuellement la pile, puis fait un appel au contrôleur ou enchaîne sur l'exécution de la ligne suivante.



ENCHAINEMENT LOGIQUE SIMPLIFIE DE L'INTERPRETEUR

LE MODE DEFINITION :

En mode définition, les lignes frappées sont transformées en code exécutable, puis conservées sous forme de segments de plusieurs lignes par le gestionnaire de mémoire.

Il est possible de modifier et supprimer des lignes d'une fonction, d'en insérer de nouvelles, etc...

CHAPITRE 2

DESCRIPTION DE L'IMPLEMENTATION

Ce chapitre s'intéresse tout particulièrement aux structures de données utilisées dans l'interprète : après la description de la représentation interne des objets APL, une seconde partie est consacrée à la gestion de la mémoire.

La troisième partie montre de quelle manière ces structures sont utilisées dans la conduite de l'interprétation.

La quatrième partie, enfin est consacrée aux problèmes des changements de contextes, et décrit la structure de la pile.

REPRESENTATION INTERNE DES OBJETS APL

Un objet APL est défini pour l'utilisateur par son nom, à ce nom l'interpréteur associe un descripteur 4 mots et travaille ensuite sur ce descripteur sauf pour certaines procédures de la machine de commande.

Ces descripteurs se trouvent soit dans la table des descripteurs contenue dans la table des symboles, soit pour les objets créés dynamiquement par l'interpréteur lui-même au cours de l'exécution d'une ligne dans une table de descripteurs temporaires. Ils sont construits par les procédures effectuant les opérateurs : l'affectation par exemple pour les variables ou par la gestion des fonctions au moment de la première ouverture pour les fonctions.

On distingue deux types principaux de descripteurs correspondants aux objets de type opérande (type 1) et aux objets de type fonction (type 2). Dans la première catégorie se rangent les descripteurs des variables, constantes et temporaires, les descripteurs des fichiers et des groupes ; elles est caractérisée par le bit 0 du mot 0 à zéro. Les descripteurs des fonctions sont de type deux et ont le bit 0 du mot 0 à un.

Dans les deux cas le mot zéro défini le type précis de l'objet et un certain nombre de renseignements sur son état.

Les mots 1, 2 et 3 servent d'informations complémentaires ou de pointeurs dans la zone donnée sur des tables ou des valeurs.

I - REPRESENTATION DES VARIABLES CONSTANTES ET TEMPORAIRES :

On ne considère que trois types d'opérandes : les logiques, les réels et les chaînes de caractères, qui peuvent tous être scalaires ou tableaux de rang au plus égal à 4. Dans le but de simplifier les traitements ces objets sont toujours sous leur forme calculée : il n'y a aucun retard dans les calculs.

Exemple : ι 500 est stocké sous la forme des 500 premiers entiers et non sous une forme compactée.

Dans une première version deux formes de stockage seulement ont été utilisées :

- Le stockage sous format réel standard (32 bits) pour les objets de type logique ou réel.
- Le stockage sous format caractère (1 caractère par octet) pour les objets de type chaîne de caractère.

En effet, des instructions de chargement et rangement d'octets et de réels sur 2 mots existent sur le T1600 et permettent une manipulation facile de ces 2 types d'opérandes. Il nous a semblé inutile d'ajouter un type entier au moins dans un premier temps. Les logiques rangés sous format réel perdent évidemment beaucoup de place puisqu'ils occupent 32 bits au lieu d'un, par contre, cela limite à deux tailles d'éléments et permet de simplifier énormément les traitements des opérateurs: en effet, un logique peut intervenir indifféremment comme logique ou comme réel de valeur 0 ou 1 dans une opération, le ranger sous forme réelle évite donc des tests et des conversions.

Toutefois, dans la version actuelle les logiques sont représentées sur 1 bit, sans que les temps d'exécution s'en soient ressentis. Cette modification a d'ailleurs amené les auteurs à systématiser l'accès aux variables, et facilitera désormais l'introduction de nouveaux types : entier, double précision, etc...

La valeur d'un scalaire est indiqué directement dans les mots 1 et 2 du descripteur, le mot 3 est alors inutilisé. Dans le cas des tableaux les mots 1 et 2 contiennent la taille du tableau sur mots ou sur octets selon son rang.

Le mot 3 est un pointeur sur la valeur qui se trouve dans la zone gérée par la gestion de mémoire pour les variables et temporaires ou dans le code exécutable pour les constantes.

DESCRIPTEURS

Descripteur d'une variable, constante ou temporaire:

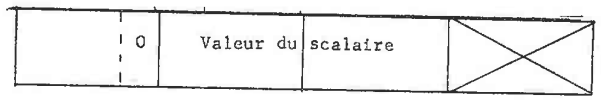
Mot 0:	0 1 . . 4 5 . . . . . D . .
Desc. de type 1	0 . . . . .
Type	. X X X . . . . .
Descripteur protégé	. . . . 1 . . . . .
Donnée protégée	. . . . . 1 . . . . .
Rang	. . . . . X X X

Type: 0 : logique
1 : caractére
2 : réel
3 : indéfini

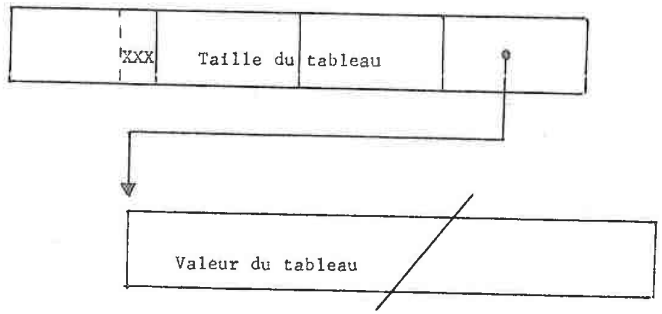
Bits 4 et 5: 1 0 : variable
0 1 : constante
0 0 : temporaire

Rang: 0 à 4

Descripteur scalaire:



Descripteur tableau:



II - REPRESENTATION DES FONCTIONS :

Une fonction est une suite de lignes frappée par l'utilisateur et utilisable après définition comme un opérateur APL classique. Les lignes sont analysées au fur et à mesure de leur entrée et sont stockées sous forme de code exécutable ceci oblige à décoder une ligne avant de la lister mais permet une exécution plus rapide des fonctions.

Pour compacter l'information tout en gardant suffisamment de souplesse à l'accès et à la modification des lignes celles-ci sont rangées dans des segments de longueur fixe de 1 ou plusieurs secteurs disque. Une table à deux mots par entrée permet de retrouver la ligne voulue connaissant son numéro :

- Le premier mot est le numéro de la première ligne du segment
- Le second le pointeur sur le segment

La première entrée de la table est spéciale, elle contient le nombre d'entrées utilisées et le premier numéro de ligne libre.

A part cette table qui définit la fonction elle-même il est nécessaire de connaître à l'appel d'une fonction les noms des variables locales pour sauvegarder les variables globales correspondantes. On construit donc également deux tables de variables locales :

- Une table contenant outre le nom de la fonction (global) la liste de toutes les variables explicitement déclarées comme locales dans la ligne zéro de la fonction ainsi que le nom du résultat et des arguments éventuels (noms implicitement locaux). A l'appel de la fonction toutes ces variables seront mises à indéfinies sauf les arguments.

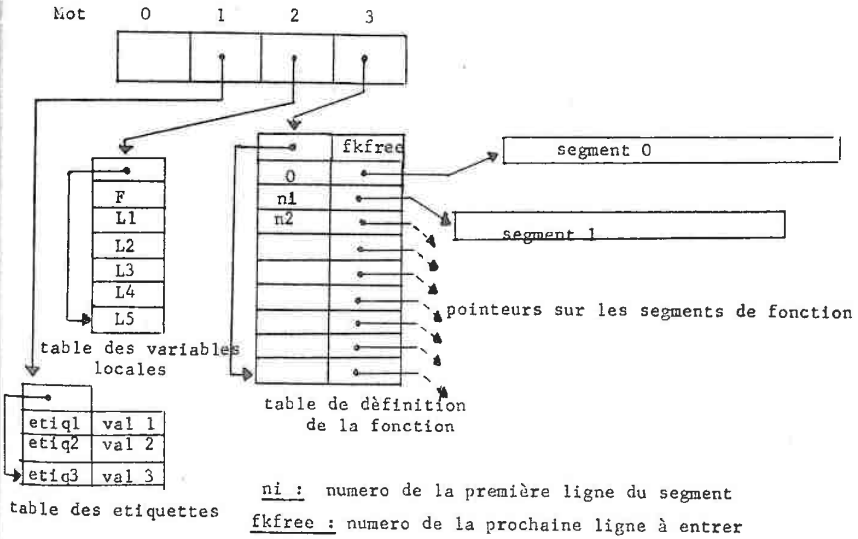
- Une table des étiquettes, variables implicitement locales, contenant également la valeur de ces étiquettes c'est-à-dire le numéro de la ligne où elle apparaissent en tant qu'étiquettes. Elles sont initialisées à cette valeur à l'appel de la fonction.

Le mot 1 du descripteur pointe sur la table des étiquettes, le mot 2 sur la table des variables locales et le mot 3 sur la table de définition de la fonction.

Le mot 0 indique si la fonction a ou non un résultat, un ou deux arguments si elle est verrouillée, pendante ou suspendue et enfin si son exécution doit être tracée sur certaines lignes ou arrêtée avant certaines lignes.
(Dans ces derniers cas on indique au niveau de chaque ligne si elle doit être tracée ou si on doit s'arrêter avant de l'exécuter).

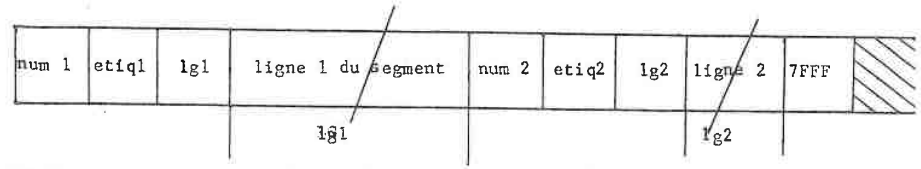
Descripteur d'une fonction:

Mot 0:	0 1 2 3 4 5 6 7 8 9 A . . . . .
Desc. type 2	1 . . . . .
Fonction	. 0 . . . . .
Resultat	. . X . . . . .
Paramètre droit	. . . X . . . . .
Paramètre gauche	. . . . X . . . . .
Fonction vèrouillée	. . . . . X . . . . .
Fonction pendante	. . . . . X . . . . .
Fonction suspendue	. . . . . X . . . . .
Vecteur de trace non vide	. . . . . X . . . . .
Vecteur stop non vide	. . . . . X . . . . .
Ligne 0 definie	. . . . . X . . . . .



ni : numero de la première ligne du segment
fkfree : numero de la prochaine ligne à entrer

Structure d'un segment de fonction:



etiqi : numero du descripteur de l'etiquette attachée à la ligne si elle existe sinon '3FFF

III - REPRESENTATION INTERNE D'UNE FONCTION SYSTEME :

Une fonction systeme est représentée par un descripteur de quatre mots, pointant sur trois données de la gestion de mémoire. Elles représentent respectivement une zone de données dynamiques, la table de translation des adresses, et le code objet de la fonction. Les actions effectuées sur ces tables sont les suivantes :

- Données dynamiques : leur gestion est à la charge de la fonction système, qui doit donc les activer et les désactiver. Après chargement d'une fonction, le pointeur sur cette donnée est initialisé à la valeur -1.
En cas d'erreur, on effectue un RLSE sur cette zône.

- Table de translation des adresses : elle est utilisée à l'appel de la fonction système, une fois le code objet activé. Elle permet la translation des adresses relatives, en fonction de la nouvelle implantation en mémoire du code objet. Elle est désactivée une fois cette translation effectuée. Elle ne subit que des RQST et RLSE.

- Code objet : il est activé à l'appel de la fonction, et désactivé en fin d'exécution seulement. L'activation s'effectuant par un RQST, c'est à la charge de la fonction d'effectuer un MOD si elle désire retrouver dans le local les données dans leur dernier état.
Le code objet est désactivé après appel de la procédure de retour.

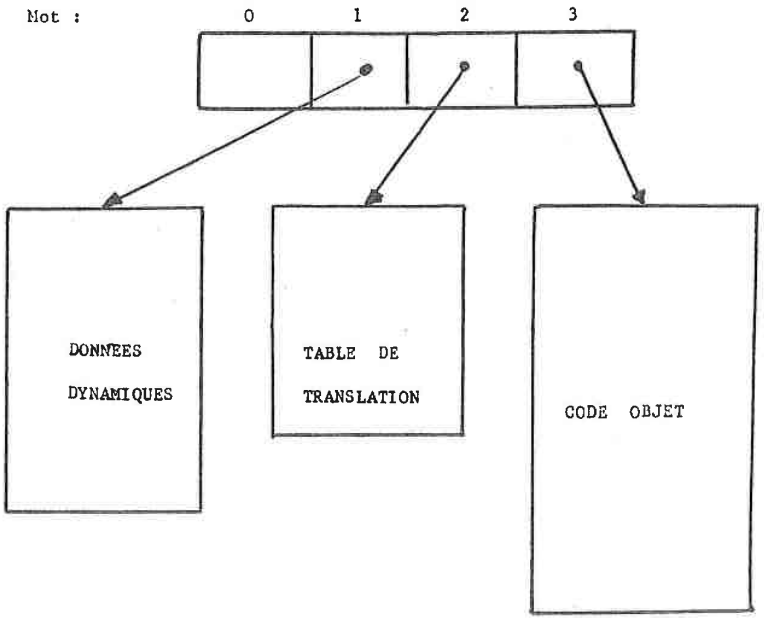
Les bits 14 et 15 du descripteur vont contenir le type de la fonction système : en effet, certaines considérations syntaxiques imposent parfois de savoir si une fonction est niladique, monadique ou dyadique. La connaissance du type de la fonction, fourni au chargement de celle-ci, permet alors de lever l'indétermination. Les valeurs sont les suivantes :

XX = 0 : la fonction n'a pas d'arguments
XX ≠ 0 : la fonction a des arguments.

Dans ce dernier cas, pour des valeurs de XX de 10 et 11, on vérifie que la fonction est utilisée en monadique ou dyadique respectivement. On ne fait pas de test pour XX valant 01.

Descripteur d'une fonction systeme:

Mot 0:	0 1 2 3 4 5 6 7 8 9 A B C D E F
Desc. type 2	1 . . . . .
Fonction systeme	. 1 . . . . .
	. . X . . . . .
- inutilisé -	. . . X X X X X X X X X X X . .
Type	. . . . . X X



Structure d'une entrée dans la table de translation

add (l_6)	info (16)

(..) : taille en bits

Le premier mot de la table de translation contient l'adresse de début de la fonction systeme.

Le dernier mot contient la valeur -1 indiquant la fin de la table.

Il faut ajouter à info l'adresse de début du code objet.

GESTION DE LA MEMOIRE

Ce chapitre décrit la gestion de la mémoire d'une zone de travail active, ainsi que l'évolution de sa conception au fur et à mesure de l'avancement du travail. En conclusion, on indiquera quelques extensions envisagées, notamment dans le but de permettre une certaine paramétrisation en fonction de la configuration disponible, ainsi qu'une meilleure utilisation de l'espace libre.

I - INTRODUCTION.

Au cours d'une session de travail, l'utilisateur définit un certain nombre de données (variables et fonctions). Ces données sont conservées dans une zone dite espace de travail (en anglais working space, d'où l'abréviation WS quelquefois utilisée). Un espace de travail est dit actif si l'utilisateur travaille effectivement sur les données qu'il contient. L'utilisateur peut également conserver une ou plusieurs copies de son espace de travail actif. Une telle copie est dite espace de travail inactif.

Tous les objets auxquels veut accéder l'utilisateur à un instant donné doivent donc tenir dans sa zone de travail active (ceci n'est plus tout à fait vrai, les derniers développements d'APL incluant un système d'accès à des fichiers externes au catalogue APL). Cette zone doit donc avoir une taille suffisante pour que l'interprète puisse traiter une quantité convenable de données et ne soit pas réduit à un calculateur de bureau. Dans le cas de APL/360, une zone de travail occupe 64 K octets. Sur un mini-ordinateur, il est impossible de disposer d'une telle place en mémoire centrale. Il faut donc prévoir un mécanisme logiciel de mémoire virtuelle faisant appel à une mémoire secondaire (disque à tête fixe ou à tête mobile).

L'utilisateur peut ainsi avoir accès sous APL/1600 à des zones de travail de 32 K mots, alors que la place disponible en mémoire centrale est de l'ordre de 8 K mots.

La gestion de mémoire est donc composée de deux parties relativement indépendantes, la gestion de la mémoire centrale et la gestion de la mémoire secondaire.

II - GESTION DE LA MEMOIRE CENTRALE.

A - Structure.

La structure de la mémoire a été initialement définie par M. F.D. ARMINGAUD.

La mémoire centrale libre est constituée d'une liste de blocs doublement chaînés.

Chaque bloc est constitué de six mots de controle (cinq en tête de bloc et un à la fin) encadrant la partie utile à l'interprète lui-même. Ces mots de contrôle se décomposent comme suit :

- les premiers et derniers mots contiennent la longueur du bloc.
- les mots 1 et 2 sont les pointeurs amont et aval dans la chaîne.
- les mots 3 et 4 définissent la classe du bloc.

Il existe trois classes distinctes de blocs allouables. Ces classes sont numérotées de zéro à deux.

Classe 0 : blocs libres

Classe 1 : blocs ayant une copie sur disque

- le mot 3 contient l'adresse du pointeur référençant le bloc.
- le mot 4 contient l'adresse en mémoire secondaire (disque) d'une copie conforme du bloc.

Classe 2 : blocs sans copie sur disque
L'allocation d'un bloc de classe 2 nécessite donc une écriture sur disque.

Les blocs actifs sont retirés de la chaîne, puisqu'ils ne sont en aucune manière allouables. Pour indiquer l'état du bloc, on positionne à 1 les bits zéro du premier et du dernier mot du bloc.

Les données contenues dans les blocs sont à la disposition de l'interprète, et ne sont jamais modifiées par le gestionnaire de mémoire. Par contre, celui-ci seul a accès aux mots de contrôle.

L'interprète dispose de quatre primitives de base :

- GET : lui alloue dynamiquement un bloc de la taille spécifiée.
- KILL : pour libérer la place occupée par la donnée spécifiée.
- RLSE : pour désactiver une donnée.
- RQST : pour activer une donnée

Il peut également utiliser les trois primitives suivantes :

- MOD : pour activer une donnée, mais en détruisant sa copie sur disque si elle existe (lorsque l'interprète désactivera la donnée par la suite, celle-ci passera en classe 2).
- COPY : copie une donnée sur disque.
- INIT : initialisation de la mémoire comme un bloc unique de classe 0.

B - Mécanisme d'allocation et de desallocation.

Les blocs non actifs sont chaînés dans l'ordre des classes 0, 1, 2. Le gestionnaire de mémoire dispose pour gérer cette liste de trois pointeurs :

- HEAD : pointe sur la tête de liste (premier bloc de classe 0).
- COP : pointe sur le dernier bloc de classe 0 ou 1.
- TAIL : pointe sur le dernier bloc de la liste.

L'idée initiale de F.D. ARMINGAUD était d'utiliser un processus parallèle à l'interpréteur, la coopérative, pour faire passer les blocs de classe 2 en classe 1 après une copie sur disque. Cette idée n'avait pas beaucoup de sens dans le contexte de temps partagé tel que le conçoit le moniteur TSM. En effet, sous ce moniteur, chaque utilisateur est représenté par un processus particulier. Lorsqu'un processus est inactif, les données sur lesquelles il travaille sont écrites sur disque dans la zone de swap-out. La coopérative ne pouvait donc pas travailler à son tour sur ces données. D'autre part, la version tournant sous BOS/D devait répondre à des critères de rapidité dans la mise au point, et de compatibilité avec la version TSM pour des questions de maintenance. La coopérative n'a donc pas non plus été implémentée dans la version BOS/D.

Le premier algorithme d'allocation effectuait une première passe sur toute la liste pour chercher un bloc de taille suffisante (y compris dans les blocs de classe 1 et 2), puis, si aucun ne convenait, libèrait tous les blocs de classe 1 d'abord, de classe 2 ensuite, soit libèrait un à un les blocs de classe 1 et 2 (l'algorithme était une sorte de "best fit" amélioré). Mais lorsque la conception de l'interpréteur a été achevée, l'existence à l'intérieur de blocs de références à d'autres blocs (c.f. gestion des fonctions, de la table des symboles et de la pile des contextes) a conduit à un algorithme beaucoup plus rigide : les blocs sont libérés dans l'ordre dans lequel ils ont été désactivés.

Finalement, en imposant qu'aucun bloc contenant des pointeurs ne se retrouve en classe 1, nous avons été conduits à l'algorithme suivant travaillant en deux passes :

- première passe : en se limitant aux blocs de classe 0 et 1, on alloue le premier bloc suffisamment grand trouvé (méthode du "first fit").

- seconde passe : on libère un à un les blocs de classe 1, puis 2 dans l'ordre du chaînage, en testant à chaque fois si l'on n'a pas obtenu par fusion un bloc de taille suffisante.

Si après ces deux passes, il n'a pas été trouvé de place, il se produit un déroutement vers la procédure d'erreur. En effet, dans l'état actuel de l'interprète, il n'est pas possible de déplacer les blocs actifs, car ceux-ci peuvent être référencés par plusieurs pointeurs.

Si le bloc trouvé est plus grand que la taille demandée, on le fractionne en deux, à condition que la différence soit supérieure à une taille fixée à la génération qui est au minimum de 7 mots (par exemple, dans la version actuelle, cette taille est de 12 mots).

La désactivation d'un bloc est une opération beaucoup plus simple. Après un "RLSE", un bloc de classe 1 est chaîné après le bloc pointé par COP, un bloc de classe 2 est chaîné en fin de liste. Après un "KILL", le bloc de classe 0 obtenu est chaîné en début de liste, et on cherche à le fusionner avec les deux blocs physiquement contigüs.

III - GESTION DE LA MEMOIRE SECONDAIRE.

La mémoire secondaire consiste en une partie d'un disque à têtes fixes ou à cartouches que nous désignerons sous le terme de "disque APL".

Le disque APL est géré par incréments de 2 K mots, l'incrément étant la plus petite portion allouable à un utilisateur. Lorsqu'un utilisateur se connecte sous le système APL, on lui alloue à sa demande de 0 à 16 incréments pour sa zone de travail active. Les adresses de ces incréments sont conservées dans une table.

Toutes les adresses disque utilisées dans une zone de travail sont relatives à celle-ci, et occupent un octet chacune. L'adresse se décompose en 4 bits pour le numéro d'incrément, et 4 bits pour le numéro de secteur dans l'incrément.

L'adresse disque complète d'une donnée contient également sa longueur. Celle-ci occupe sur l'octet gauche d'un mot, sept bits, et contient une valeur M définie par :

$M = (L + 15)/16-1$

L étant le nombre de mots occupés par la donnée

Le bit zéro d'une adresse disque reste à zéro. Dans le cas d'une donnée située en mémoire, ce bit est à un : il correspond au bit de post-indexation.

L'espace disque d'une zone de travail est gérée grâce à une table de bits.

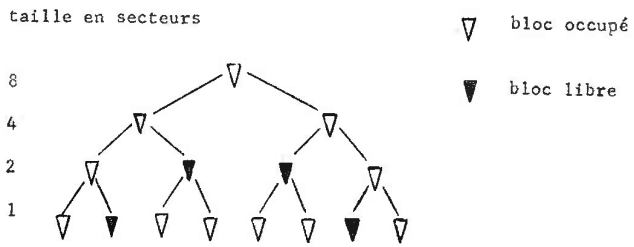
Dans la version initiale, la gestion se faisait par la méthode des blocs jumeaux*(E7, B6). La mémoire secondaire était considérée comme un ensemble de 16 blocs de 16 secteurs, plus 32 blocs de 8 secteurs, etc, jusqu'à 256 blocs de un secteur. On avait donc affaire à une structure de forêt binaire, dont seules les racines correspondaient à des blocs libres au départ. L'algorithme d'allocation décomposait éventuellement un bloc trop gros en deux plus petits, l'algorithme de désallocation s'efforcait ensuite de reconstituer des blocs les plus gros possibles.

---

* ou méthode dichotomique.

Cette méthode permettait de trouver rapidement une adresse disque disponible, mais la libération de la place non prise, puis la libération des secteurs occupés lorsqu'on détruisait une donnée étaient complexes.

De plus, la méthode ne permettait pas un compactage idéal des données : après une session de travail au cours de laquelle on avait effectué un grand nombre d'allocations et de désallocations, il se trouvait souvent que N secteurs contigüs ne soient pas allouables en tant que tels, parce que coupés par une ou plusieurs frontières de blocs, témoin l'exemple suivant :



Dans la configuration ci-dessus, on peut allouer plus de deux secteurs pour une donnée, alors qu'en réalité, il y en a six contigüs qui sont libres.

Nous avons donc été conduits à choisir une méthode plus simple qui consiste à représenter chaque incrément de 2 k mots (soit 16 secteurs) par un mot dans la table d'allocation, un bit à 1, indiquant que le secteur correspondant est libre, un bit à 0, qu'il est utilisé.

L'inconvénient précédent ne se retrouve plus que pour les frontières d'incréments, ce qui est beaucoup moins grave et de toute façon inévitable, puisqu'il n'a pas été prévu de pouvoir fractionner des données en mémoire secondaire. En effet, les incréments ne sont pas forcément contigüs sur le disque, ce qui fait l'intérêt de leur utilisation.

Des mesures et comparaisons entre les deux méthodes ont montré qu'elles étaient équivalentes en temps de traitement, et qu'effectivement le taux de remplissage de la zone de travail était meilleur avec la seconde méthode, surtout dans le cas général où la taille moyenne des données est inférieure à un demi k mots.

IV - EXTENSIONS ENVISAGEES.

a) Optimisation de l'utilisation de la mémoire centrale.

Il arrive parfois que l'erreur "WS, FULL, ERROR" se produise, bien que l'on ait suffisamment de place en mémoire. Ceci vient du fait que l'on ne peut pas fusionner deux blocs de classe zéro, s'ils sont séparés par un bloc actif, et que l'on ne peut pas déplacer celui-ci. La possibilité de déplacer ces blocs actifs est donc à l'étude, ce qui permettrait une utilisation optimale de l'espace libre. Ceci impose cependant une contrainte importante au niveau de l'écriture de l'interprète : les blocs actifs ne doivent être référencés que par un seul pointeur. Une solution serait de ne permettre l'emploi de multiples pointeurs sur un même bloc, qu'entre deux appels à la gestion de mémoire. Chaque procédure de l'interprète devrait donc obéir à la structure logique suivante :

- effectuer d'abord tous les appels nécessaires à la gestion de mémoire.
- ne multiplier qu'ensuite les pointeurs sur les objets actifs, en sachant que ce n'est qu'après le dernier appel que les valeurs de tous les pointeurs ne seront plus modifiées.

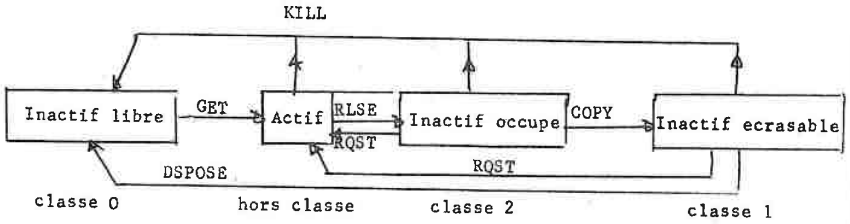
b) paramétrisation.

Certaines applications APL sont limitées non par la taille, mais par la masse des données. Le problème de l'utilisateur n'est plus de disposer de grosses données, mais d'un grand nombre de petites. On peut résoudre ce problème par une paramétrisation de la structure de l'adresse disque. Une adresse disque tient sur 15 bits. Ces 15 bits sont divisés en trois champs : longueur, numéro d'incrément, numéro de secteur. Actuellement ces champs occupent respectivement 7, 4 et 4 bits. En faisant varier la taille des différents champs, on peut augmenter la taille maximum d'une donnée, la taille maximum d'une zone de travail ou bien le nombre de données distinctes qu'on peut écrire sur disque.

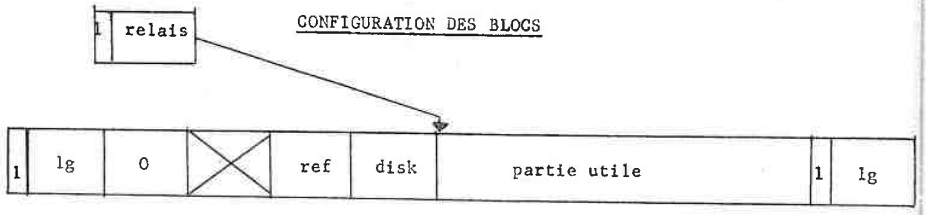
c) utilisation d'une mémoire virtuelle microprogrammée.

Un des développements en cours du système APL/1600 est l'utilisation d'un opérateur mémoire virtuelle mi-logicielle, mi-matérielle. Selon les résultats il pourrait être envisagé de remplacer la gestion de mémoire actuelle par cette mémoire virtuelle.

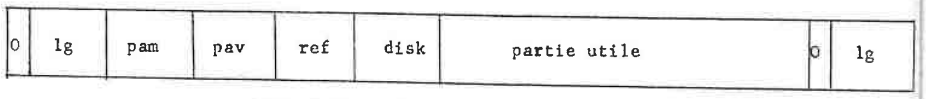
GRAPHE DE TRANSITION DES BLOCS



CONFIGURATION DES BLOCS

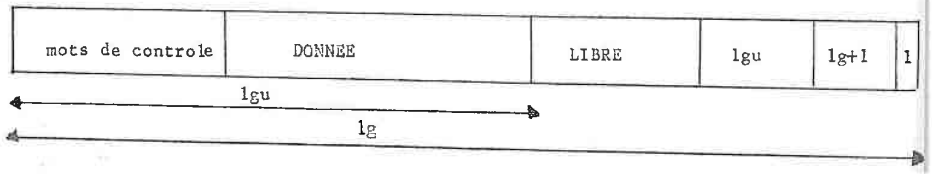


Bloc actif

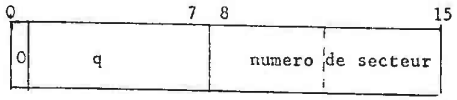


Bloc desactivé

LONGUEUR



ADRESSE DISQUE



$16 \times q \leqslant$ longueur donnée $< 16 \times (q+1)$

INTERPRETATION

La machine d'exécution travaille sur du code exécutable généré, soit à partir d'une ligne frappée par l'utilisateur (mode terminal, mode exécution de fonction, mode entrée de données) soit à partir d'une chaîne de caractères (mode "execute").

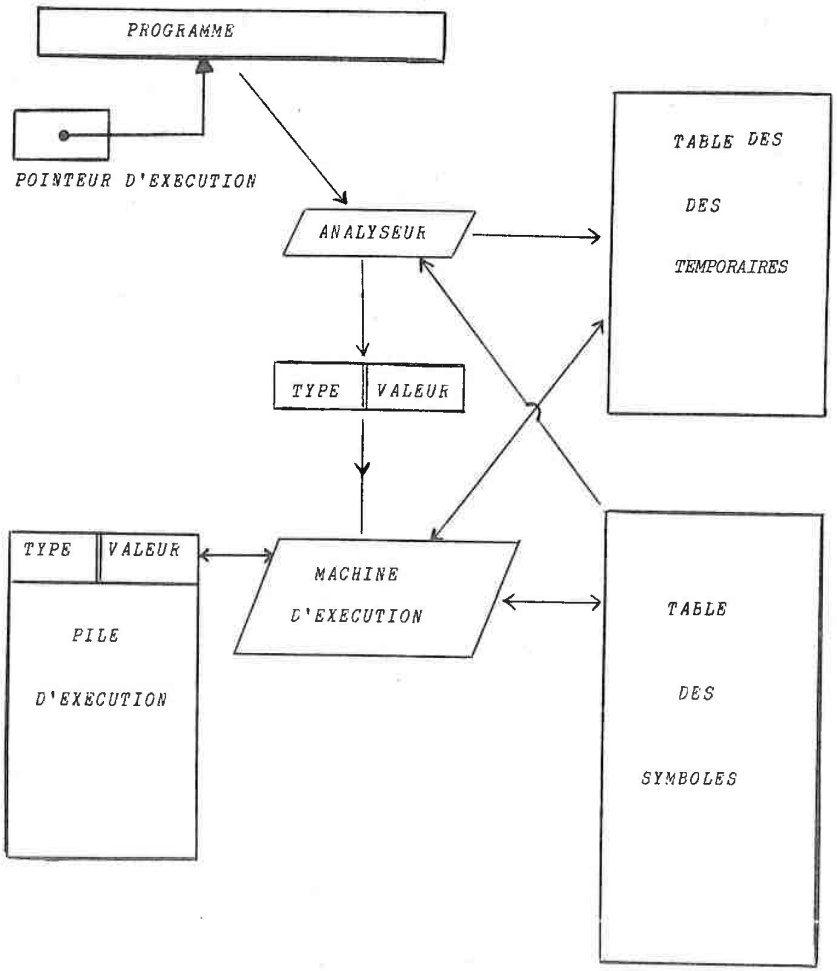
L'interprétation se fait de droite à gauche sans priorité d'opérateur sauf en cas de parenthèsage, le mécanisme en est très simple. Il utilise des tables qui sont à des adresses fixes en mémoire : le code exécutable, une pile d'exécution, une table de descripteurs temporaires, ainsi que la table de descripteurs de la table des symboles.

a) Procédure principale :

Le squelette de l'exécutant est composé d'une procédure principale très courte, qui ne fait que déterminer, à partir du type de l'unité syntaxique, qui est au sommet de la pile et de celui de l'unité suivante dans le code, quelle procédure elle appelle et ceci grâce à une table de décisions. Elle peut ainsi appeler 10 procédures différentes qui ont pour rôle d'empiler l'unité syntaxique en entrée, d'exécuter un opérateur ou d'effectuer l'appel à une fonction, de vérifier la correspondance des parenthèses et des crochets, d'indiquer une erreur de syntaxe, de terminer l'exécution d'une ligne ou d'appeler une fonction système.

Un analyseur examine le code, rendant à chaque appel l'unité syntaxique suivante sous la forme de deux mots : le type de celle-ci et sa valeur. La valeur est le code APL de l'opérateur ou un pointeur sur le descripteur de l'objet APL. Il y a 10 types d'unités syntaxiques: fin de ligne, opérateur, opérande, fonction niladique, fonction monadique, fonction dyadique, parenthèse ou crochet fermé, parenthèse ou crochet ouvert, point-virgule (indiquant un empilement des résultats qui ne seront écrits qu'en fin de ligne, ou séparant les opérandes d'une indexation), fonction système.

Cet analyseur construit également un descripteur temporaire pour les constantes rencontrées, la valeur restant dans le code exécutable sauf si la constante est scalaire.



SCHEMA DE L'INTERPRETATION
UTILISATION DES DIFFERENTES TABLES

Sommet pile \ Entrée	eol	oper.	opde.	fct.n	fct.m	fct.d	)]	([	;	fct.s
Fin de ligne	8	4	0	5	4	4	0	4	4	9
Operateur	2	2	1	5	2	2	0	2	2	2
Operande	8	0	4	4	6	0	0	3	0	0
Fonction nil.	4	4	4	4	4	4	4	4	4	4
Fonction mon.	4	4	4	4	4	4	4	4	4	4
Fonction dya.	4	4	7	5	4	4	0	4	4	9
)]	4	4	0	5	4	4	0	4	0	0
([	4	4	4	4	4	4	4	4	4	4
;	4	4	0	5	4	4	0	3	0	9
Fonction syst.	2	2	1	2	2	2	0	2	2	9

0 Empiler l'unité syntaxique en entrée.

1 Exécuter un opérateur dyadique.

2 Exécuter un opérateur monadique.

3 Vérifications sur les parenthéses et les crochets.

4 Erreur de syntaxe.

5 Appel d'une fonction niladique.

6 Appel d'une fonction monadique.

7 Appel d'une fonction dyadique.

8 Fin d'éxécution d'une expression.

9 Appel d'une fonction systéme(niladique).

TABLE DE DECISION DE L'EXECUTANT

b) Empiler une unité syntaxique.

Chaque entrée dans la pile est constituée des deux mots donnés par l'analyseur. Pour les variables, c'est donc une référence au descripteur qui est empilée et la variable n'est pas dupliquée mais simplement désactivée. Cette façon de faire a des conséquences dont il faut tenir compte dans certains cas particuliers :

Ex :

```
      A ← 1 2 3 4 5
      (A ← 1) + A
2
      A
1
```

Mais

```
      A ← 1 2 3 4 5
      (A ← 1) + A + 1
3 4 5 6 7
      A
1
```

Ce choix se justifie dans la mesure où les parenthèses inversent l'ordre normal d'interprétation :

$(A \leftarrow 1) + A$ est équivalent à $A + A \leftarrow 1$

Le nombre de temporaires empilés est limité arbitrairement à 16 par la taille choisie pour la table des descripteurs temporaires. Toutefois, ces temporaires sont attachés à la ligne de code, c'est-à-dire qu'en cas d'appel de fonction ils sont sauvegardés et on peut de nouveau avoir 16 temporaires pour chaque ligne de la fonction.

c) Exécution d'un opérateur.

La plupart des opérateurs pouvant être monadiques ou dyadiques,la détermination du type se fait juste au moment de l'exécution : s'il a deux opérandes l'exécutant appelle la procédure préparant l'exécution des opérateurs dyadiques, il appelle celle qui prépare l'exécution des opérateurs monadiques dans le cas contraire. Ces deux procédures sont pratiquement identiques, elles récupèrent l'opérateur puis activent le ou les opérandes en éclatant leurs descripteurs dans des tableaux de 11 mots prédéfinis : il y a un tableau pour chaque opérande, un pour le résultat et un pour chaque index éventuel. Chaque mot contient une des informations compactées dans le descripteur de 4 mots,associé à l'opérande : longueur, type, rang, dimensions ou valeur.

Les traitements des opérateurs sont en général assez voisins les uns des autres. Avant tout, il est nécessaire de vérifier la compatibilité des données pour les types, les rangs ou les dimensions. Sauf dans le cas des opérateurs scalaires,ces vérifications sont spécifiques de chaque opérateur. En même temps que s'effectuent ces vérifications, sont déterminées les caractéristiques du résultat. Ensuite, on demande l'espace mémoire qui lui est nécessaire.

A ce niveau deux cas peuvent se produire :

- le traitement de l'opérateur est indépendant de la structure de la donnée, c'est-à-dire que l'itération va être effectuée sur le nombre total d'éléments,que la donnée soit un scalaire, un vecteur ou une matrice ; c'est le cas en particulier des opérateurs scalaires monadiques ou dyadiques. Dans ce cas, l'itération est très simple : on boucle sur le nombre d'éléments.

- le traitement de l'opérateur est fonction de la structure de la donnée et l'accès aux éléments n'est pas forcément séquentiel. Il faut alors mettre en place un mécanisme d'accès plus compliqué, qui sera souvent plus lourd que le traitement lui-même. Ce mécanisme est en théorie identique pour tous les opérateurs, cependant il est souvent adapté aux caractéristiques de chaque opérateur afin d'optimiser le traitement. Après le traitement, il est parfois nécessaire d'effectuer une remise en forme de résultat. Puis on libère les opérandes avant d'affecter un descripteur temporaire au résultat.

d) Appel d'une fonction.

Les arguments éventuels sont d'abord sauvegardés en les dupliquant si ce sont des variables ou des constantes. (Les temporaires sont récupérés sans qu'on ait besoin de les dupliquer), puis on appelle la procédure de sauvegarde de contexte.

On sauvegarde ensuite les variables globales correspondant aux variables locales de la fonction en construisant une table dont chaque entrée est constituée par le numéro du descripteur de la variable suivi par ce descripteur lui-même, ce descripteur est ensuite mis à indéfini ou initialisé dans le cas d'une étiquette avec le numéro de la ligne ou elle apparaît.

Enfin on initialise les variables correspondant aux arguments avec les valeurs précédemment sauvegardées et on lance l'exécution de la fonction en transférant dans le code exécutable sa première ligne et en appelant la machine d'exécution.

e) Fin d'exécution d'une ligne.

Le traitement dépend du mode d'exécution de la ligne. En mode terminal, si la dernière opération effectuée n'est pas une affectation, on écrit tous les résultats restant dans la pile d'exécution en vérifiant toutefois qu'ils sont séparés par des point-virgules, puis on se branche au programme principal de l'interprète qui effectue les entrées standards. En mode entrée de donnée ou "execute" on restaure le contexte et on se rebranche à la machine d'exécution après avoir récupéré le résultat qui doit être unique.

En mode exécution de fonction, si la ligne est tracée on écrit le nom et le numéro de la ligne puis, le ou les résultats éventuels même si la dernière opération a été une affectation. S'il existe une ligne suivante, elle est transférée dans le code exécutable et on se branche à la machine d'exécution sinon on sort de la fonction.

Pour ce faire, on sauve le résultat éventuel, puis on restaure les variables globales en récupérant leurs descripteurs après avoir détruit les variables locales. Puis on restaure le contexte et on récupère le résultat dans un temporaire et on se rebranche à la machine d'exécution.

f) Appel d'une fonction système.

Si la fonction n'est pas en mémoire, elle est lue sur disque, puis la relocation est faite. Après avoir vérifié que le nombre d'argument est correcte, on effectue un branchement au point d'entrée.

Au retour, on effectue la libération des opérandes et on alloue un descripteur pour le résultat selon l'état des descripteurs éclatés, et on continue l'exécution.

g) Traitement des erreurs.

Lorsqu'une procédure quelconque détecte une erreur elle appelle une procédure d'erreur qui imprime le message correspondant et se branche à la procédure principale qui édite la ligne erronnée en indiquant où se trouve l'erreur puis désactivent toutes les données et détruit tous les temporaires de la ligne.

Le principe général est qu'aucune erreur ne doit être fatale et que l'utilisateur peut en cas d'erreur reprendre le travail après correction ou modification de ses fonctions et variables.

h) Gestion de la mémoire au niveau interprétation.

Le but recherché est de permettre l'utilisation maximum de la mémoire centrale disponible par les objets APL nécessaires à l'exécution d'une ligne tout en minimisant les échanges entre mémoire centrale et disque de sauvegarde. On s'efforce pour cela de suivre les deux propositions définies par R. ZAKS [C7] :

P1 : Ne pas créer un nouveau bloc s'il peut occuper une place déjà allouée.
P2 : Libérer la place dès que l'on n'a plus besoin de l'objet.

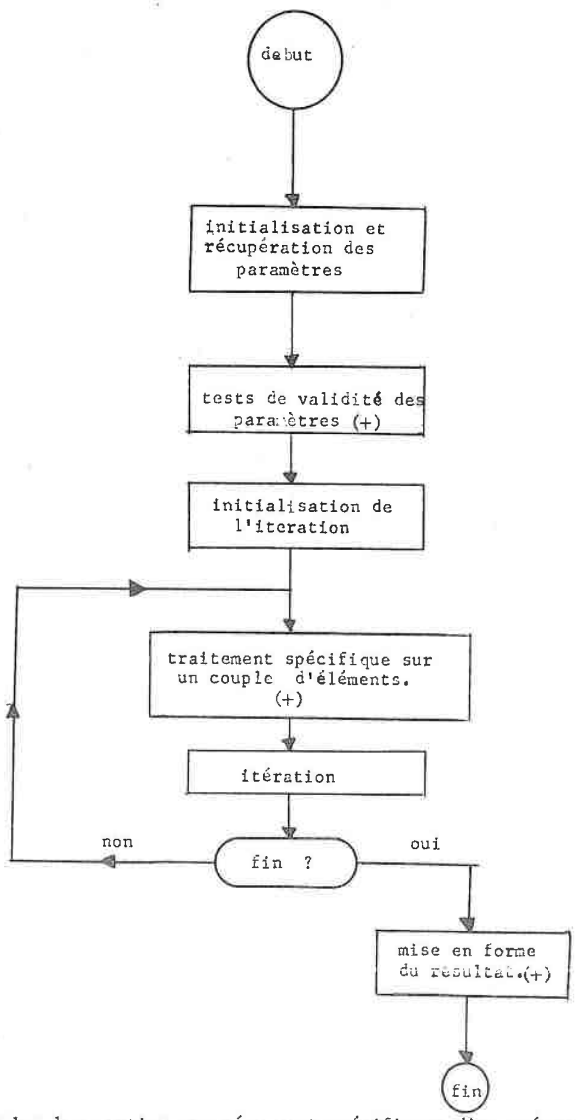
Au cours de l'exécution de la ligne, seules sont actives les données nécessaires à l'exécution de l'opérateur en cours. Ces données sont d'ailleurs désactivées ou détruites dès qu'elles ne sont plus utiles.

Chaque résultat d'opération est temporaire ; seule l'assignation attache à un nom une valeur permanente (jusqu'à la prochaine assignation). On s'efforce donc de calculer les résultats des opérateurs directement dans un des opérandes lorsque c'est possible ; il faut pour cela qu'un des opérandes soit temporaire et de même taille que le résultat, il faut de plus que les calculs se fassent séquentiellement sur les éléments des opérandes. Ceci est notamment le cas des opérateurs scalaires monadiques ou dyadiques.

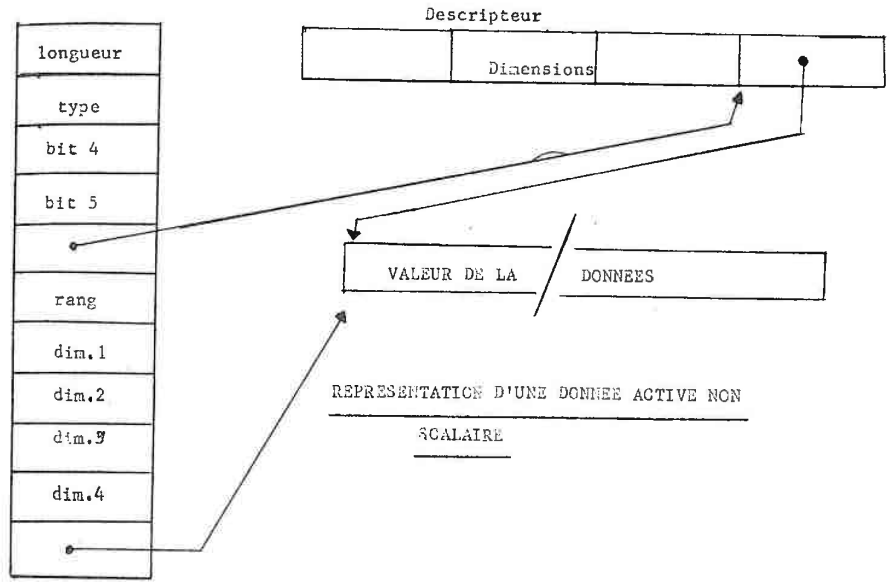
Lors d'une assignation on ne duplique la valeur affectée que si elle est constante ou non temporaire.

Ces précautions alourdissent légèrement le déroulement de l'interprétation mais évitent un trop grand fractionnement de la mémoire et permettent de travailler avec des données de taille plus importante.

SCHEMA DE TRAITEMENT D'UN OPERATEUR

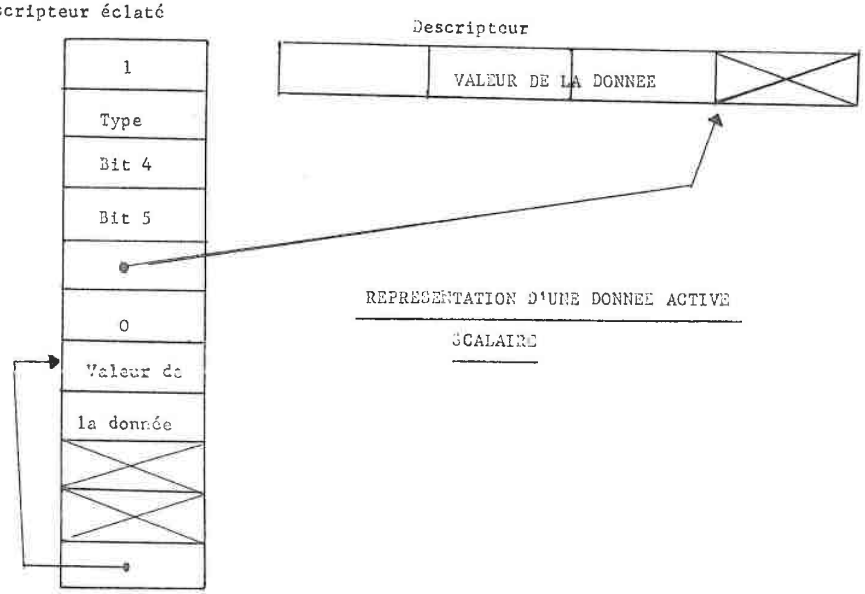


(+) Seules les parties marquées sont spécifiques d'un opérateur particulier.



REPRESENTATION D'UNE DONNEE ACTIVE NON SCALAIRE

Descripteur éclaté



REPRESENTATION D'UNE DONNEE ACTIVE SCALAIRE

Descripteur éclaté

CHANGEMENTS DE CONTEXTE

Ce chapitre décrit la structure de la pile utilisée pour les sauvegardes de contextes ainsi que les justifications du choix de celle-ci, puis le mécanisme de changement de contexte et ses conséquences sont explicitées.

I - INTRODUCTION.

L'interpréteur peut fonctionner selon 5 modes différents :

- 0 mode terminal
- 1 mode entrée de données
- 2 mode exécution de fonction
- 3 mode exécution d'une chaîne de caractères
- 4 mode définition de fonction

Chaque passage d'un mode à un autre nécessite la sauvegarde d'un certain nombre d'informations caractérisant l'état de l'interpréteur : on appelle contexte ces informations nécessaires à la reprise de l'exécution.

Le niveau de récursivité pouvant être élevé et la taille des informations sauvegardées variable mais importante, le mécanisme de changement de contexte doit inclure une gestion dynamique d'une structure de type pile, cette gestion utilisant la gestion de mémoire de manière standard et servant simplement d'intermédiaire entre celle-ci et l'interprète.

II - GESTION DE LA PILE.

La première solution consiste à chaîner derrière un pointeur de pile les informations empilées (figure 1). Cette solution conduit à un fractionnement important de la mémoire centrale en petits blocs. Elle est donc coûteuse en temps et en place.

Une solution améliorée permettant d'éviter ce trop grand fractionnement de la mémoire consiste à "bloquer" les informations dans des données de taille fixe multiple de 128 mots soit un secteur disque ce qui optimise également l'occupation du disque (figure 2).

Cette structure relativement simple à gérer serait satisfaisante si elle ne devait être utilisée effectivement que comme une pile. En fait, l'utilisateur peut à tout moment demander à visualiser l'état de cette pile, il faut donc pouvoir l'explorer sans la détruire.

Pour éviter une trop grande lourdeur à cette opération, nous avons été conduits à adopter une structure un peu plus complexe utilisant une table de pointeurs. Le pointeur de pile permet d'accéder à une table contenant les pointeurs sur les différents segments de la pile. Ces segments sont crées au fur et à mesure des besoins et détruits dès qu'ils sont de nouveau vides.

La taille d'un segment de pile est en principe fixe et déterminée à la génération du système, toutefois, si on veut empiler une donnée de taille supérieure on crée alors un segment spécial pour la ranger, ce segment ne contiendra que cette donnée. Dans le cas général, on remplit le segment sommet de pile avec les informations à empiler au fur et à mesure puis quand il est plein un nouveau segment est crée par l'intermédiaire de la gestion de mémoire et on lui fait correspondre l'entrée suivante dans la table des pointeurs. Empiler une information consiste en fait à mettre cette information suivie de sa longueur dans le segment sommet de pile.

Cette structure est explicitée dans la figure 3.

III - CHANGEMENT DE CONTEXTE.

Nous avons vu qu'il y a changement de contexte chaque fois qu'on passe d'un mode de fonctionnement à un autre, ou chaque fois qu'on appelle une fonction. Quelles sont les informations qui doivent être sauvées ? Tout d'abord le mode actuel de fonctionnement et les différentes tables utilisées : pile d'exécution, table des descripteurs temporaires. Puis selon le cas le nom et le numéro de la ligne en cours si on exécute une fonction ou bien le code exécutable si celui-ci a été tapé par l'utilisateur ou vient d'une chaîne de caractères.

Enfin, fait également partie du contexte la table des variables globales correspondant aux variables locales d'une fonction. En effet, ces variables globales ne seront restaurées que quand on dépilera le contexte correspondant à l'appel de la fonction. En d'autres termes, les variables locales à une fonction masquent pour tout contexte de niveau supérieur la valeur des variables globales correspondantes.

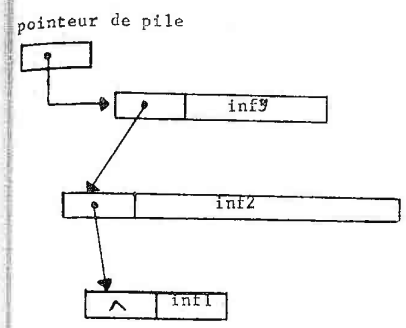


Fig.1

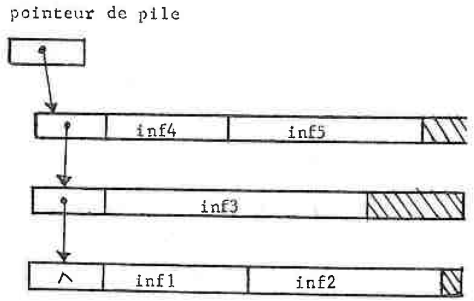


Fig.2

Solutions envisagées

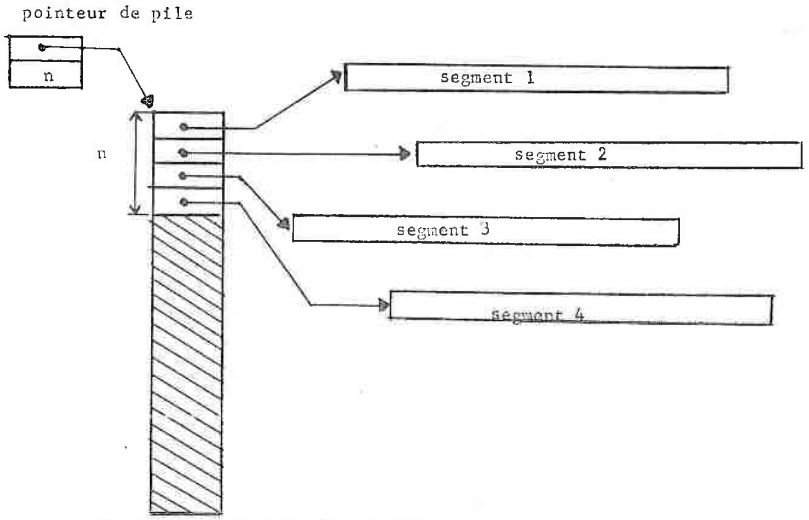


Table des pointeurs de segments

Fig.3

Solution adoptée

Lorsqu'on exécute une ligne de fonction,on ne sauve pas le code,par contre, on sauvegarde 2 bits du descripteur indiquant si la fonction est pendante ou suspendue : une fonction est dite pendante si son contexte a été empilé pour une cause normale, elle est dite suspendue si elle a été interrompue pour l'une des raisons suivantes.

- l'utilisateur a interrompu une sortie par la touche "attention"
- l'utilisateur a interrompu l'exécution par cette même touche
- l'interprète a détecté une erreur dans une ligne de la fonction
- l'interprète s'est arrêté avant d'exécuter une ligne conformément aux instructions de l'utilisateur (mode arrêt pour la mise au point des fonctions)

Une fonction peut être à la fois pendante et suspendue.

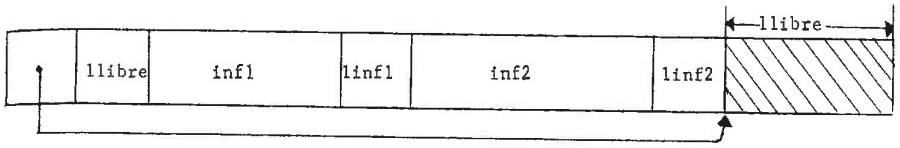
On ne peut détruire ou modifier une fonction pendante, car cela risquerait de rendre invalide les informations rangées dans la pile et donc de provoquer des erreurs lors du dépilage. Ces bits servent donc de protection pour le système.

Toutes les informations ne sont pas toujours nécessaires, toutefois, elles sont toujours sauvegardées pour que tous les contextes aient une structure identique.

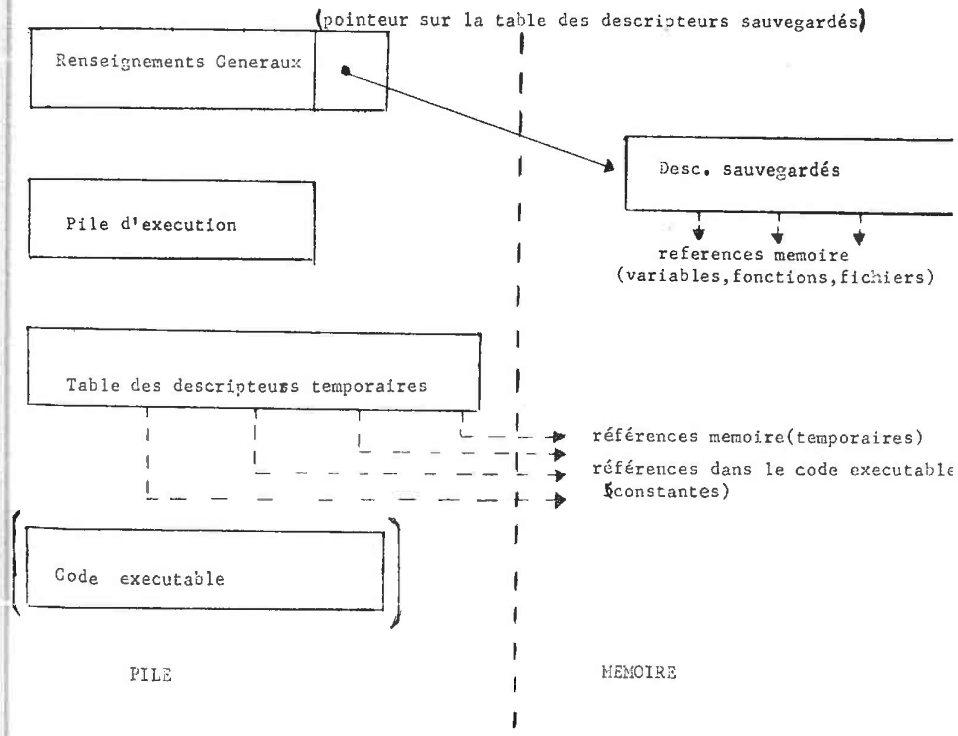
La manipulation normale d'un contexte se fait par deux procédures :
- sauvegarder un contexte
- restaurer un contexte

Une procédure spéciale intégrée à la machine de commande traite la visualisation.

STRUCTURE DE LA PILE



Structure d'un segment de pile



Structure d'un contexte

CHAPITRE 3

MESURES ET OPTIMISATION

Une fraction relativement importante de la durée de la réalisation a été consacrée aux tests et aux mesures de performances de l'interprète. Dans certains cas, ces mesures se sont révélées utiles pour mettre en évidence certaines faiblesses de conception, et ont conduit les auteurs à repenser partiellement ou totalement certains aspects de la réalisation.

Ce chapitre traite plus particulièrement des mesures et optimisations, effectuées tant au niveau local, c'est-à-dire au niveau des algorithmes utilisés pour la programmation des différentes fonctions du calculateur APL, qu'au niveau global, sur la structure adoptée pour l'assemblage des différents modules constituant l'interprète.

MESURES ET AMELIORATION DES ALGORITHMES

Un certain nombre de mesures de temps d'exécution ont été effectuées afin de tester les performances de l'interprète. Les plus significatives d'entre elles sont indiquées ici, avec certaines justifications concernant les algorithmes employés.

I - Exécution d'une fonction de la littérature.

La fonction utilisée est la suivante :

```
    ∇ R←C N
[1]   R←(⌈N÷2)⌽(¯1+⍳N)⍉(¯1+⍳N)⌽(N,N)⍴1N×N
    ∇
```

Elle génère un carré magique d'ordre N pour N impair.
Les temps indiqués sont ceux écoulés entre le lancement de la fonction et l'impression de la première ligne du résultat.
N indique la taille de la matrice résultat (le nombre d'éléments étant N au carré), les temps sont donnés en secondes.

N	T	N	T	N	T	N	T
3	1	5	1	7	2	9	2
11	3	13	3	15	3	17	3
19	4	21	4	23	4	25	4
27	4						

Il semblerait à première vue que T soit proportionnel à N. Or, si l'on examine l'algorithme, on constate qu'il comporte essentiellement trois rotations (l'une de ⌈N÷2 et les deux autres de ¯1+⍳N), portant sur une matrice de N^2 éléments. Le temps d'exécution devrait donc être proportionnel à N^2 et non à N. Comme on a également tenu compte du temps d'édition de la première ligne du résultat, édition portant sur la première ligne de la matrice, et donc sur N éléments, on voit que le temps d'exécution de la fonction est masqué par le temps d'édition du résultat. Ceci est donc un facteur dont il faudra tenir compte dans les différentes mesures effectuées.

2 - Temps de réaction de la gestion de mémoire.

La fonction suivante ne fait intervenir que les opérateurs rho (donnant la taille d'un élément) et iota, le générateur d'indices. Ces opérateurs sont traités dans le même segment : il n'y aura donc pas d'accès disque pour changer de segment. Rho est traité en quelques instructions-machines. Iota ne prend que cinq ou six instructions par élément généré. La fonction teste donc essentiellement la gestion de mémoire, puisque l'on demande 1 mot, puis N mots, puis 1 mot, etc...

```
    ∇ R←F N
[1]   R←⍴⍳⍴⍳⍴⍳⍴⍳⍴⍳⍴⍳⍴⍳⍴⍳⍴⍳⍴⍳⍴⍳⍴⍳⍴⍳⍴⍳⍴⍳⍴⍳⍴⍳⍴⍳⍴⍳⍴⍳N
    ∇
```

Les temps observés sont les suivants :

N	T	N	T	N	T	N	T
100	1	200	2	400	2	800	2
1000	2	1200	3	1400	3	1600	3
1800	3	2000	3	2200	3	2400	3

La courbe obtenue est sensiblement linéaire. L'ordonnée à l'origine de son approximation linéaire donne une valeur de l'ordre de 1,8 seconde. Ceci mesure donc le temps de réaction du système (exécution de la fonction, écriture du résultat et appels à la gestion de mémoire). Le temps d'exécution d'une fonction ineffective est de l'ordre d'une seconde environ. Compte tenu de la faible précision des mesures effectuées, il est contestable d'attribuer l'écart moyen de 0,8 secondes au travail de la gestion de mémoire.

Dans le même ordre d'idée, on peut rapporter les mesures effectuées avec les fonctions suivantes :

```
    ∇ R←F N
[1]   R←1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,⍳N
    ∇
```

N	T	N	T	N	T	N	T
100	1	200	2	300	3	400	3
500	3	600	4	700	4	800	4

Le travail demandé à la gestion de mémoire est un peu plus grand dans ce cas précis, puisqu'après la libération d'un segment de taille N est demandé un segment de taille N+1, ce qui impose des fusions avec les blocs contigüs.

```
    ∇ R←F N
[1]   R←⌽⌽⌽⌽⌽⌽⌽⌽⌽⌽⌽⌽⌽⌽⌽⌽⌽⌽⌽⌽⌽⌽⌽⍳N
    ∇
```

N	T	N	T	N	T	N	T
100	2	200	3	300	3,5	400	4
500	4,5	600	5	700	5,5	800	6
900	6,5	1000	7,5	1100	8	1200	9

Cet exemple, comme les précédents, permet de mettre en évidence la part du temps de traitement qui est due à l'opérateur ⌽ , et donc proportionnelle à N, et celle qui est due à l'exécution d'une fonction et aux appels à la gestion de mémoire, indépendante de N.

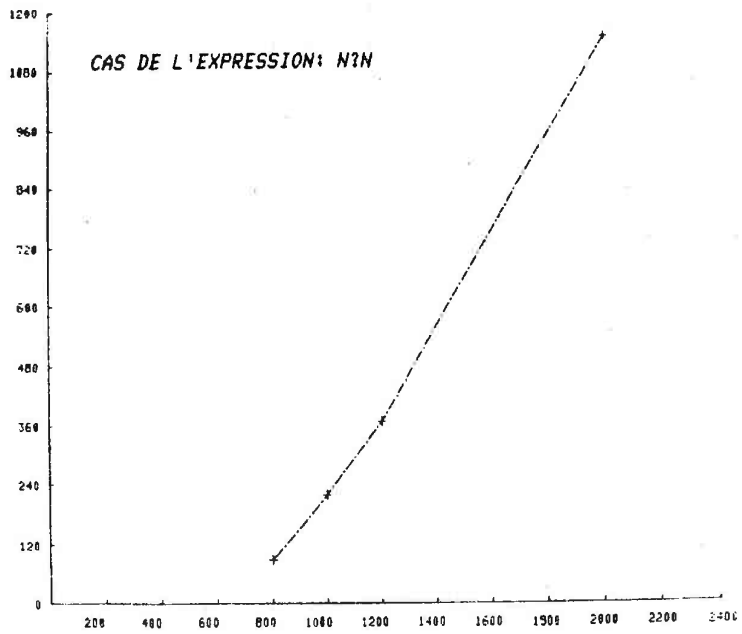
3. Performances de certains opérateurs.

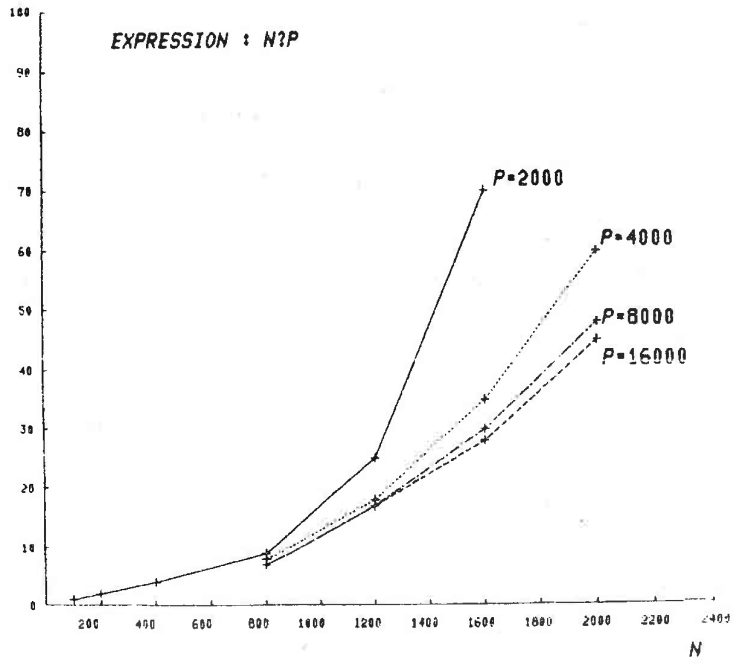
Les tests effectués sur les différents opérateurs du langage ont prouvé la non-optimalité de certains algorithmes choisis. Le fait était flagrant pour les opérateurs ? dyadique (tirage aléatoire exhaustif) et ⍋ et ⍒ , tris ascendants et descendants.

a) Le tirage aléatoire.

L'expression N?P, avec N ⩽ P doit fournir N des P éléments du vecteur ⍳P. L'algorithme était le suivant : tirer un nombre au hasard K pris entre 1 et P, regarder s'il existait déjà dans le vecteur résultat. Si non, l'ajouter. Si oui, essayer K+1 modulo P.

Les temps d'exécution dépendent manifestement de N et de P. On obtient, pour différentes valeurs, les résultats suivants :





P = 2000

N	T	N	T	N	T	N	T
100	1	200	2	400	4	800	9
1200	25	1600	70	2000	1152		

P = 4000

800	8	1200	18	1600	35	2000	60

P = 8000

800	7	1200	17	1600	30	2000	48

P = 16000

800	7	1200	17	1600	28	2000	45

Dans le cas particulier de l'expression N?N (permutation aléatoire des N premiers entiers), les résultats sont les suivants :

N	T	N	T	N	T	N	T
800	90	1000	220	1200	370	2000	1152

Les remarques qui s'imposent à l'examen de ces résultats sont les suivants :

- l'algorithme est mauvais
- il est surtout mauvais lorsque N est voisin de P, ce qui est grave, des expressions du type N?N étant souvent employées dans des problèmes de simulation.

L'algorithme a donc été remplacé par le suivant :

- Si N est supérieur ou égal à P/16, on utilise une table de P bits, avec la signification suivante : le bit K vaut 0 si le nombre K a déjà été introduit dans le vecteur résultat, 1 dans le cas contraire. Il est donc possible, après tirage d'un nombre aléatoire, K, de savoir en quelques instructions s'il est déjà présent ou non dans le vecteur résultat. Si le nombre est déjà présent dans le résultat, on recherche directement le premier bit positionné à droite de la position K. Ceci permet d'accélérer le traitement, surtout dans des cas comme N?N, où les collisions deviennent nombreuses pour les derniers éléments.

- Si N est inférieur à P/16, le problème des collisions devient beaucoup moins grave. La recherche du nombre K dans le vecteur résultat s'effectue par clé. Celle-ci est calculée comme l'octet droit de K, la recherche s'effectuant au moyen de l'instruction RANK, micro-programmée, qui permet de rechercher un octet dans une zone. Si l'octet est trouvé, on vérifie que le mot correspondant a bien la valeur K, et on essaye alors K+1 modulo P. Dans le cas contraire, on relance l'instruction à partir de la position suivante.

Le seuil de P/16 choisi correspond à la raison suivante : le résultat est fourni à l'utilisateur sous forme réelle, occupant donc 2xN mots. Or, le calcul s'effectuant sur des entiers, il reste N mots disponibles, soit 16xN bits. Lorsque cela est possible, on utilise cette zone comme indicateur (premier cas).

Il est à remarquer que l'algorithme est identique dans les deux cas, seule la programmation diffère.

Avec la nouvelle implantation de ces algorithmes, les temps d'exécution deviennent : beaucoup plus satisfaisants :

P = 2000

N	T	N	T	N	T	N	T
400	1	800	1	1200	1	2000	1
P = 1600							
400	1	800	1	1200	1	2000	1

b) Les tris ascendant et descendant :

L'expression ⍋ *V*, *V* étant un vecteur numérique, doit fournir un résultat *R* tel que *V* [*R*] contienne tous les éléments de *V* dans l'ordre ascendant. Autrement dit, ⍋ *V* fournit le vecteur d'indices triant *V* dans l'ordre ascendant. De plus, les indices correspondants à des éléments égaux doivent être classés du plus petit au plus grand. De même, ⍒ *V* fournit le vecteur d'indices triant *V* dans l'ordre descendant.

⍋ 2 3 2 5 ⟷ 1 3 2 4 et ⍒ 2 3 2 5 ⟷ 4 2 1 3

On voit donc que ⍒ *V* n'est égal à ⌽⍋ *V* que si tous les éléments de *V* sont distincts.

L'algorithme utilisé pour le tri était le suivant : une table de *N* éléments indiquait si l'élément correspondant du vecteur *V* avait déjà été introduit dans le vecteur résultat. L'algorithme était donc proportionnel à *N* au carré.

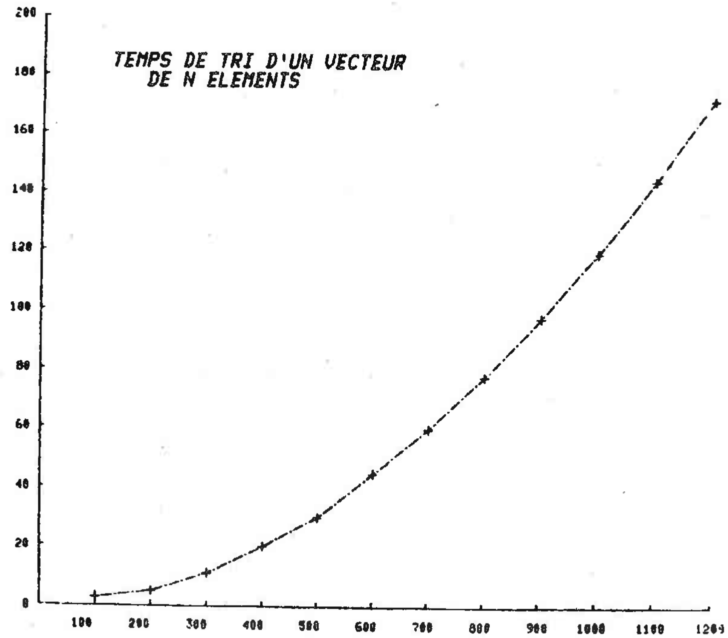
Les temps obtenus pour le tri d'un vecteur de taille *N* sont les suivants : (cas d'un tri ascendant).

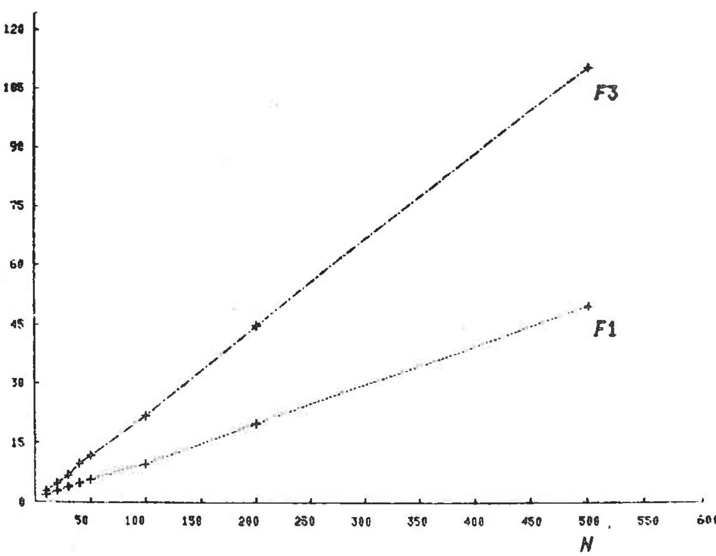
N	T	N	T	N	T	N	T
100	3	200	5	300	11	400	20
500	30	600	45	700	60	800	78
900	98	1000	120	1100	145	1200	173
1300	203	1400	235				

Les temps sont similaires pour le tri descendant. Citons :

200	6	400	20	800	76	1200	170

Là encore, le problème était simple : il était nécessaire de faire appel à des algorithmes plus élaborés. Une première distinction a été faite, basée sur le type des données : dans le cas d'une donnée logique, l'opération de tri consiste à prendre les indices de tous les éléments égaux à zéro, puis de ceux égaux à un. On obtient ainsi un algorithme travaillant en un temps proportionnel à N. Dans le cas d'une donnée de type réel, l'algorithme retenu a été celui de l'arbre binaire, car il était possible de l'implémenter sans que l'on ait à faire appel à la gestion de mémoire. En effet, le résultat est un vecteur de taille N réels, il nécessite 2 N mots pour sa représentation. Les calculs s'effectuant en entier, il reste N mots disponibles qui vont contenir les éléments de l'arbre.





Les résultats des nouveaux algorithmes sont les suivants :

Pour un vecteur à trier réel :

N	T	N	T	N	T	N	T
200	1	400	2	600	3	800	4
1000	5	1200	6	1400	7	1600	8

Pour un vecteur logique :

400	1	800	1	1200	1	1600	1

4 - Fonctions bouclant sur elles-même.

Le problème posé était le suivant : le temps de mise en place d'une ligne de fonction pour exécution est-il négligeable devant le temps d'exécution des opérateurs de la ligne ? On a donc mesuré les temps d'exécution d'une fonction ineffective, et ceux d'une fonction semblable, mais comportant en outre l'exécution d'un certain nombre d'opérateurs supplémentaires. Ces deux fonctions sont les suivantes :

```
    ∇ F1 N
[1]    I←0
[2]    →2×ιN>I←I+1
    ∇
    ∇ F2 N
[1]    I←0
[2]    →2×ιN>I←I+1α9↑⌽9↑⌽9↑⌽9↑⌽9↑⌽9↑⌽9↑⌽9↑⌽9↑⌽9↑⌽9↑⌽9↑⌽9↑⌽9
       ↑⌽9↑⌽9↑⌽9↑⌽9↑⌽9↑⌽9↑⌽9ρ'*'
```

Ces deux fonctions exécutent une fois leur ligne 1 et N fois leur ligne 2. Elles comportent les mêmes instructions de gestion de la boucle. La fonction F2 comporte en outre l'exécution de 20 opérateurs prendre (↑) et symétrie (⌽), portant sur une petite donnée de type caractère, de taille 9.

Les temps d'exécution sont les suivants :

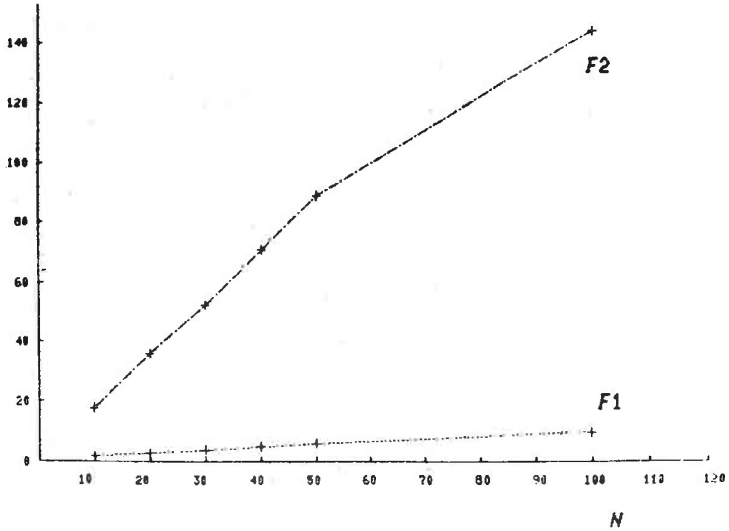
Fonction F1

N	T	N	T	N	T	N	T
10	2	20	3	30	4	40	5
50	6	100	10	200	20	500	50
1000	103						

Fonction F2

10	18	20	36	30	53	40	71
50	89	100	145				

Il faut signaler que les opérateurs ↑ et ⌽ ont été choisis parce qu'ils étaient situés dans des segments différents, ce qui imposait pour la fonction F2 40 chargements de segments supplémentaires.



On voit dans ce cas précis que le temps imputable à l'exécution de la fonction (i.e chargement et lancement d'une ligne) est faible devant le temps d'exécution des opérateurs de la ligne. Ce résultat n'est cependant valable que s'il y a beaucoup d'opérateurs dans la ligne, autrement dit si la ligne est suffisamment longue. Ce résultat expérimental correspond d'ailleurs à une attitude que les utilisateurs d'APL ont adopté depuis bien longtemps (c.f. les algorithmes de la revue APL QUOTE-QUAD).

5 - Utilisation de l'opérateur exécute

L'opérateur execute ⍎, permet une chaîne de caractères APL comme une expression entrée au terminal. La fonction F3 a été écrite afin d'en tester les performances :

```
      ∇ F3 N
[1]     I←0
[2]     ⍎'→2×⍳N>I←I+1'
      ∇
```

Si nous comparons avec la fonction F1 définie pour le test 4, nous constatons que les actions supplémentaires suivantes sont effectuées :

- les deux changements de contexte, pour passer du mode fonction au mode exécute, et réciproquement.

- transformation de la ligne source en code exécutable.

Cette transformation, qui est exécutée une fois pour toutes à la définition de la fonction, doit être répétée ici à chaque passage dans la ligne 2 de la fonction.

Les résultats obtenus sont les suivants :

N	T	N	T	N	T	N	T
10	3	20	5	30	7	40	10
50	12	100	22	200	45	500	111
1000	223						

On constate, sur le diagramme de comparaison des fonctions F1 et F3 que l'opérateur exécute ne coûte pas trop cher, même dans un cas comme celui-ci, où il ne s'imposait nullement. En réalité, le temps de mise en oeuvre de l'opérateur est souvent plus faible que celui d'appel d'une fonction. Il est ainsi rentable de remplacer un appel de fonction par un opérateur exécute lorsque cela est possible.

6 - Utilisation de fonctions récursives.

Les utilisateurs hésitent souvent à utiliser des fonctions récursives, car ils ont peur que cela leur coûte trop cher en temps machine. Or, le test d'une procédure récursive comme IOTA fait apparaître un certain nombre de résultats curieux :

```
    ∇ R←IOTA N
[1]   →FIN×⍳N≤1
[2]   →0×R←(IOTA N-1),N
[3]  FIN:R←1
    ∇
```

N	T	N	T	N	T	N	T
0	1	1	1	2	1	3	1
4	2	5	2	6	3	7	3
8	3	9	3	10	4	50	14
100	35	150	45	200	63	250	85
300	105						

Compte tenu du fait qu'un appel de IOTA N nécessite l'exécution de 2 N lignes ainsi que de 2 N changements de contexte, on constate que contrairement à ce que l'on pouvait penser, la récursicité n'est pas du tout coûteuse en temps machine. Par contre, à chaque appel il est nécessaire de sauvegarder les variables locales de la fonction, et l'on voit que la récursivité sera grosse consommatrice de place mémoire : ainsi, dans une zone de travail de 16 k mots (vierge sauf la fonction IOTA), l'expression IOTA 250 provoque un débordement de la mémoire. Il en est de même dans une zone de 32 k, pour l'expression iota 350.

L'utilisation de la récursivité n'est donc pas intéressante lorsqu'elle peut être remplacée par des boucles. Par contre, si le niveau de récursivité reste faible, (comme par exemple dans le cas d'un tri par fusion récursif), la récursivité fournit un instrument intéressant et relativement peu coûteux.

Conclusions sur les tests.

Bien que des comparaisons précises n'aient pas pu être effectuées, les tests ont montré que APL/1600 était relativement performant (par exemple vis à vis de APL/360 tournant sous CP-CMS). De plus, ils ont permis de décider dans quels domaines faire porter les efforts. Ainsi, nous avons été amenés à reconcevoir les algorithmes de tri et de tirage aléatoire.

Il faut enfin signaler que ces tests ont été effectués sur le T1600 des Mines de Saint-Etienne, où étaient disponibles le flottant microprogrammé, ainsi qu'un disque à têtes fixes. D'après certaines mesures effectuées sur une autre installation, avec flottant programmé et disque à cartouche, il semblerait qu'il faille multiplier par un facteur de 3 les temps obtenus.

OPTIMISATION DE LA STRUCTURE DE L'INTERPRETE

La version initiale de l'interprète était prévue pour une configuration disposant de 16K mots en mémoire centrale, ce qui, du fait de la taille du système d'exploitation, n'autorisait que 5K de processeurs pour APL. Cette contrainte conduisait à définir une racine de 1K, et 14 branches de recouvrement, de 4K chacune.

Une fois la mise au point terminée, les auteurs se sont intéressés à évaluer le pourcentage du temps pris par la lecture des branches de recouvrement.

Devant l'ordre de grandeur de ce résultat (70% de la durée totale de l'exécution), une étude approfondie a été entreprise, conduisant à une réorganisation de la structure de l'interprète, qui a permis un gain de 10 à 100 sur le nombre de lectures de branche, ce qui divise par 3 ou 4 les temps d'exécution.

Description de la version initiale :

La faible taille disponible obligeait à dupliquer certains modules dans différentes branches, par exemple le noyau de l'interprète lui-même.

La répartition était la suivante :

- Une branche générale
- Une branche d'édition de fonctions
- Quatre branches de commandes
- Sept branches d'exécution
- Une branche d'édition des résultats, enchaînement de lignes, changement de contextes.

La répartition des opérateurs dans les différentes branches d'exécution avait été effectuée pour des raisons de facilité d'implémentation plutôt qu'en fonction de statistiques.

Les mesures effectuées :

Les mesures ont été obtenues de la manière suivante : le processeur gérant le recouvrement met à jour deux tables, l'une donnant pour chaque branche le nombre de lectures effectuées, l'autre comptant le nombre d'appels de chaque procédure.

Par ailleurs, le temps moyen de lecture d'une branche a été évalué à 35 ms, dans le cas d'un disque à tête fixe.

Le tableau suivant résume les résultats obtenus pour trois applications

APPLICATION	TEMPS D'EXECUTION	NOMBRE DE LECTURE	TEMPS DE LECTURE	POURCENTAGE
1	375 s	8 129	285 s	76%
2	105 s	2 325	82 s	78%
3	55 s	594	21 s	38%

A ce stade, il était évident qu'un gain au niveau du recouvrement entraînerait une augmentation notable des performances de l'interprète.

Par ailleurs, des considérations technico-commerciales nous ont ammenés à accepter une taille supérieure pour l'interprète. C'est ainsi qu'une première réorganisation des procédures a été tentée.

Une première solution :

Sans aucune modification des programmes, il a été possible de réduire à 5 le nombre total de branches, le nombre des segments d'exécution passant de 7 à 2.

La taille des branches passait alors à 6,5 K mots, et l'on avait :

- Une branche générale, comportant les commandes sur la zone de travail, les sorties de résultats et l'enchaînement des fonctions.
- Une branche servant à l'édition des fonctions.
- Une branche traitant les commandes relatives au catalogue APL.
- Deux branches d'exécution.

On constate alors que le nombre de lectures de branches est divisé par deux, alors que le temps de lecture des branches passe de 35 à 48 ms environ. (Bien entendu, cette nouvelle organisation entraine une légère diminution de la zone disponible pour les données).

Ce résultat intéressant,facilement obtenu nous a conduit à examiner plus en détail les statistiques de recouvrement, pour déterminer les points critiques et essayer d'y remédier.

Vers une solution améliorée :

L'examen du découpage des procédures et des statistiques d'appels montre que les principaux événements qui entrainent des lectures de branches sont, par ordre d'importance décroissante :

- Les enchaînements des lignes de fonctions.
- Les enchaînements des opérateurs.
- Les appels et les retours de fonctions.

Comme il n'était pas possible de trop augmenter la taille des processeurs (une version résidente contiendrait 24 K de programmes, ce qui, compte tenu des 4 K environ nécessité par les diverses tables de symboles et zones tampons, ne laisserait que 4 K mots disponibles pour les données !), il était nécessaire d'envisager certaines modifications dans les programmes.

Nous avons ainsi été ammenés à réaliser les deux transformations suivantes :

- Un regroupement des opérateurs les plus usuels, dans une des branches, l'autre ne contenant que les moins utilisés, comme les tris, l'inversion de matrice, la résolution des systèmes linéaires, etc...

- L'addition, dans les branches d'exécution, d'une version réduite du processeur d'enchaînement des lignes et de changement de contexte, (ce qui n'augmentait pas sensiblement la taille des branches)

On obtient ainsi une taille de 9 K mots pour l'interprète, avec 4 branches de 8 K, soit :

- Deux branches d'exécution.
- Une branche contenant les procédures générales, de sorties, de résultats et d'éditions de fonctions.
- Une branche traitant toutes les commandes systèmes.

Les nouvelles mesures ont montré des gains spectaculaires dans la majorité des cas. Ces résultats sont résumés dans le tableau suivant :

APPLICATION	NOMBRE DE LECTURES POUR LE SYSTEME 1	NOMBRE DE LECTURES POUR LE SYSTEME 2	POURCENTAGE
1	8129	16	0,2
2	1123	76	7
3	3647	6	0,2
4	2755	245	10
5	594	4	0,7

Le temps de lecture d'une branche est maintenant évalué à 55 ms.

Même dans les cas les plus défavorables, cette nouvelle structure économise au moins 80% du temps de lecture des branches, ce qui fait passer ce temps à moins de 15% du temps total d'exécution.

L'ORGANISATION DEFINITIVE

Une question se pose cependant : n'est-il pas possible d'augmenter la taille de la racine tout en gardant la même taille totale, et donc en diminuant la taille des branches ?

Une réorganisation de certaines procédures, notamment l'analyse des commandes, a permis, en reprenant l'ancienne version d'exécution, d'obtenir une structure de recouvrement incluant dans la racine le processus traitant les fins de lignes (enchaînement des lignes de fonction, appel et retour de fonction, changement de contexte). Cette structure comporte une racine de 2,5 K et 5 branches de 5,5 K, reprenant la même répartition que précédemment sauf la branche traitant les commandes, qui est partagée en deux. Vu les modifications faites, cette augmentation du nombre de branches n'entraîne aucune lecture supplémentaire. Par contre, le temps moyen de lecture redescend à 43 ms, soit un gain de l'ordre de 20 %.

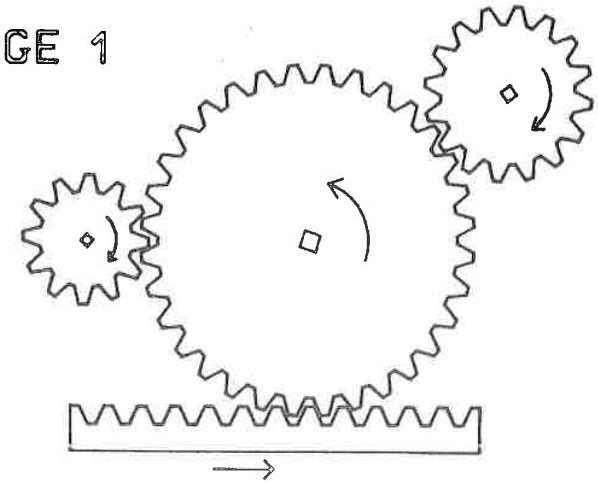
Pour un grand nombre d'application, le temps de lecture des segments de recouvrement va donc devenir négligeable devant le temps total de calcul, ce qui permettra de mieux mesurer le comportement de l'interprète lui-même, au niveau de la conception et de la logique des algorithmes.

CONCLUSION.

Les mesures effectuées ont permis de définir une configuration voisine de l'optimum dans le cas d'une configuration moyenne (16 à 24 K disponibles). La même étude est faisable pour une grosse configuration (28 à 32 K de mémoire), ce qui fera probablement apparaître l'intérêt d'un interprète d'une taille de 15 K environ, avec une partie résident de l'ordre de 5 K et des branches de 10 K. L'un des segments servira alors à l'exécution, le second remplissant toutes les autres fonctions.

PARTE III

IMAGE 1



III - MATERIEL ET INGENIERIE

Cette partie est consacrée plus spécifiquement à certains aspects pratiques de la réalisation.

Le premier chapitre effectue une présentation brève du matériel utilisé, en précisant la configuration nécessaire à l'utilisation d'APL.

Le second chapitre décrit plus en détail la méthode de programmation utilisée, ainsi que les problèmes posés par l'inadéquation du logiciel disponible, et les outils que nous avons été amenés à utiliser ou à mettre au point pour les résoudre.

o
o o
o

LE MATERIEL

Le T1600 est un petit ordinateur fabriqué par la Télémécanique Electrique. Il est adressable par mots, avec des capacités mémoire pouvant aller de 4 à 64 K mots. Les mots sont de 16 bits. L'utilisation de l'adressage indirect post-indexé permet l'accès aux octets. L'adressage se fait par l'intermédiaire de registres de base. Un certain nombre de registres rapides sont accessibles à l'utilisateur : A et B registres accumulateur et extension, X registre d'index, C, L et W registres de base, Y registre de travail, K pointeur de pile, P pointeur instruction et S registre d'état.

Il existe huit niveaux d'interruption, dont 2 pour les entrées-sorties, un pour l'horloge et un pour les déroutements de programmes. Certains peuvent comporter des sous-niveaux.

Un certain nombre d'options intéressantes sont à signaler :

- protection et relocation dynamique des programmes,
- gestion de tâches micro-programmées permettant de traiter des processus distincts.
- arithmétique flottante câblée.

Le terminal utilisé pour la mise au point de l'interprète est un Tektronix 4013 : c'est un terminal graphique comportant, en outre l'alphabet APL. D'autres terminaux APL peuvent cependant être utilisés sous réserve de modifier éventuellement les tables de conversions : Tektronix 4015, Anderson-Jacobson, IBM 2741, Télétype ASR ou KSR 38. Un module d'entrée-sortie a également été écrit pour un terminal sans caractères APL du type ASR33, mais la nécessité d'utiliser des mots-clés pour désigner des opérateurs paraît alors très contraignante et l'utilisation de terminaux APL semble pratiquement obligatoire pour un usage agréable.

L'utilisation d'APL nécessite enfin un minimum de 16 K mots de mémoire et un disque à têtes fixes ou à cartouches. La présence de l'option arithmétique flottante et un disque à tête fixe augmentent beaucoup les performances du système.

INGENIERIE

Ce chapitre décrit plus spécifiquement les outils logiciels, indépendants de l'interprète, que les auteurs ont été amenés à utiliser ou réaliser.

Nous dirons tout d'abord quelques mots sur le langage et la méthode de programmation utilisés, en essayant de montrer l'intérêt d'un langage de haut-niveau, à structure de bloc, pour une telle réalisation, ainsi que les avantages d'une programmation structurée.

Nous décrirons ensuite les problèmes posés par l'utilisation du système d'exploitation de base, et de quelle manière nous avons pu les contourner. Le système ainsi modifié, bien que très primaire encore, s'est avéré un outil de travail suffisant pour permettre la réalisation de l'interprète.

Nous parlerons également des modifications plus spécifiquement orientées vers APL, permettant en particulier l'utilisation de plusieurs tâches simultanément actives sous le système, ainsi que des processeurs de trace et de mise au point.

Pour terminer, nous signalerons l'existence d'un système de communication entre les deux ordinateurs du centre de calcul de l'Ecole des Mines de Saint-Etienne, en insistant sur l'apport d'une telle connexion à la réalisation de gros logiciels.

I - LA METHODE DE PROGRAMMATION.

Toute la programmation à l'exception du programme de recouvrement a été réalisée en PL1600. Ce langage de haut niveau voisin de PL/360 ou LP/70 est bien adapté au calculateur. Il permet la manipulation des registres et l'utilisation des instructions machines, mais fournit également un certain nombre de structures de contrôle fort appréciables : définition de procédure, de bloc, instructions if then else, do for, case of... etc.

Son utilisation a permis une réalisation rapide de l'interprète aussi bien parce que l'écriture des programmes est plus naturelle qu'en assembleur, que parce que ceux-ci une fois écrits sont beaucoup plus compréhensibles ce qui en facilite la mise au point.

A mesure que la réalisation progressait, une meilleure connaissance du code généré par le compilateur a permis aux auteurs d'optimiser ce code en choisissant les structures de contrôle les plus appropriées et en utilisant, en cas de besoin les instructions sur registres ou les instructions machines (en particulier pour certaines instructions sur nombres réels), non gérées par le compilateur, ce qui conduit à un code généré très proche de l'assembleur.

Finalement l'utilisation d'un langage de haut niveau empêchant les astuces de programmation mais ne conduisant pas, grâce à une utilisation judicieuse de toutes ses possibilités, à un code pléthorique s'est révélé très efficace.

La programmation a été structurée au maximum en découpant l'interprète en modules bien distincts : gestion de mémoire, exécution, opérateurs, génération de code,...

Chaque module remplit une fonction bien définie, mais seules les interfaces d'entrées et de sorties sont à respecter : il est possible de changer très facilement un algorithme ou une méthode sans avoir à réécrire tout l'interprète. Chaque module est lui-même découpé en procédures très courtes ayant un but très précis : aller chercher ou ranger un élément de variable, convertir un nombre. La programmation se faisant généralement ainsi : une fois défini l'algorithme utilisé, le module principal est écrit en supposant réalisées toutes les fonctions nécessaires : il décrit l'enchaînement de ces fonctions qui sont ensuite programmées à l'aide de procédures plus simples et le processus se ramifie jusqu'au niveau ou la procédure peut facilement être écrite à l'aide de quelques structures simples du langage.

Cette méthode de programmation nous a ainsi permis de modifier très facilement, par exemple, la gestion du disque pour la zone de travail active ou la représentation interne des variables logiques.

II - UTILISATION DU T1600.

a) - Adaptation du système

Les études préliminaires du projet avaient été faites avant la commercialisation du T1600 par la Télémécanique Electrique. La programmation a ainsi pu débuter dès mise à notre disposition de l'ordinateur. Cependant, le logiciel fourni par le constructeur était alors très réduit. L'utilisateur disposait en effet uniquement d'un système de base [F1] utilisant des rubans papier comme support de programme, sous lequel pouvaient travailler différents processeurs (éditeurs de texte, assembleur, compilateur PL1600, éditeur de lien, chargeurs). Chaque utilisation d'un de ces processeurs nécessitait son chargement à partir du lecteur de ruban. Chaque édition de texte, compilation etc... entrainait la lecture et la perforation de ruban de papier.

Ces outils étaient nettement insuffisants pour réaliser efficacement la programmation d'un logiciel aussi important qu'un interprète APL. Parallèlement à l'écriture des premiers modules de cet interprète les auteurs ont donc été conduits à modifier le système existant pour y inclure l'utilisation d'un disque à tête fixe, disponible sur notre configuration. Ceci a permis de mettre en place un mécanisme de chargement automatique des processeurs système d'une part, et une gestion de fichiers séquentiels très simple d'autre part pour le maintient des programmes sources et objets. Ce système amélioré au fur et à mesure des besoins notamment par l'adjonction d'un interface avec l'éditeur de liens, permettant l'édition de liens automatique de plusieurs programmes objets issus aussi bien du compilateur PL1600 que de l'assembleur, est toujours utilisé à l'Ecole des Mines de Saint-Etienne malgré la commercialisation par le constructeur de systèmes plus sophistiqués mais plus lourds à utiliser [F2]

b) - Les outils de mise au point.

En plus du système lui-même un certain nombre d'utilitaires ont été conçus pour faciliter la mise au point des programmes : visualisation de la mémoire, dé-assembleur permettant de vérifier le code généré par le compilateur, visualisation et modification d'un secteur disque.

Dès que les premiers modules de l'interprète, testés séparément ont été assemblés il a été nécessaire d'y inclure une trace permettant de suivre l'enchaînement des modules et le déroulement des actions. Cette trace

est optionnelle : le module qui l'effectue teste les clés du pupitre du calculateur pour déterminer les informations à imprimer ; ces arguments sont une identification de 2 caractères, une adresse et une longueur. Selon la position des clés, il revient immédiatement à la procédure appelante ou imprime l'identification suivi ou non par un vidage de la mémoire à partir de l'adresse indiquée et sur la longueur désirée.

L'interprète est lancé comme tâche esclave par un programme en mode maître intégré au système. Il est ainsi possible sans interrompre le fonctionnement de l'interprète de vider certaines parties de la mémoire ou de deassembler certaines parties de code grâce à des processeurs travaillant sous le numéro de tâche du système. Une gestion particulière de la tâche horloge permet de comptabiliser le temps d'unité centrale utilisé par la tâche APL : cette tâche horloge teste si la tâche logicielle en cours est la tâche APL et si oui incrémente un compteur spécial en plus du compteur de temps global.

Le fonctionnement sous forme de tâche esclave interrompue en cas de tentative de violation mémoire ou d'utilisation d'instruction privilégiée, a été précieuse dans la dernière phase de mise au point.

c) - Le mécanisme de recouvrement.

L'interprète APL, tel qu'il est conçu, afin de maintenir une compatibilité presque totale avec les autres implémentations, nécessite un ensemble de programmes dont l'encombrement est très supérieur à la taille mémoire disponible sur les configurations usuelles. Qui plus est, les nécessités de l'implantation sous un système sont telles que la taille souhaitable pour les programmes est de l'ordre de 4 à 5 K mots. Dans ces conditions un appel aux techniques de recouvrement s'impose. Pour 16 K mots de mémoire centrale.

Sous le système d'exploitation du calculateur T1600 disponible fin 1972 un tel dispositif n'existait pas. L'utilisation de recouvrement n'est devenu possible sur T1600 qu'avec le système BOSD début 1974. Un autre mécanisme plus performant, était déjà opérationnel pour l'interprète et c'est ce mécanisme que nous allons maintenant décrire. A ce niveau, deux conceptions sont possibles. On peut décider de figer la structure de l'interprète en créant des primitives telles que "chargement d'une branche", "retour à la racine". Ceci impose avant chaque appel de procédure un chargement de la branche dans laquelle elle se trouve. Il a donc été décidé d'utiliser comme dans la majorité des systèmes d'exploitations une méthode transparente à l'utilisateur ; celle-ci permet ainsi une programmation déchargée des soucis de gestion des chargements de branches et autres, la structure de l'interprète n'étant fixée en fait qu'au moment de l'édition de liens.

Le principe de fonctionnement est le suivant : les procédures mises en recouvrement sont regroupées en segments et sont toujours référencées par la section de données pointée par le registre de base RC. Cette section se trouve juste au début de la zone mémoire de recouvrement et chacune de ses entrées contient soit l'adresse de la procédure si celle-ci est présente en mémoire soit l'adresse de la procédure gérant le recouvrement dans le cas contraire. Cette procédure résidante détermine quelle était la procédure appelée et grâce à une table de correspondance charge le segment où elle se trouve et s'y branche. En même temps, des informations sont empilées pour le retour à la procédure appelante. Cette méthode est valable pour un nombre quelconque d'appels tant que la pile n'est pas saturée. L'inconvénient de la méthode réside dans l'impossibilité de passer des arguments de manière standard aux procédure mise en recouvrement. En effet, le compilateur PL1600 génère le passage des arguments par la pile et les manipulations de celles-ci, nécessairement faite par la procédure de recouvrement impose cette restriction qui se révèle peu gênante puisque les données sur lesquelles travaille l'interprète se trouvent dans une zone non mise en cause par le recouvrement (mémoire débanalisée du calculateur APL).

La procédure de recouvrement tient une comptabilité des chargements de segments et des appels de procédures ce qui a permis après quelques essais de déterminer la structure optimale du recouvrement compte-tenu de la taille choisie pour les segments.

Cette comptabilité a également permis de tester des structures de recouvrement pour des tailles de segments plus importantes adaptées à une configuration supérieure du calculateur T1600. (2ème partie chapitre 3).

III - UTILISATION D'UN GROS ORDINATEUR.

Parmi les moyens originaux mis en oeuvre pour cette réalisation, un très gros apport a été fourni par la connexion entre deux ordinateurs du centre de calcul de l'Ecole des Mines de Saint-Etienne, le Télémécanique T1600 et l'ordinateur Philips P1175.

Celui-ci est un gros ordinateur, d'une capacité mémoire de 608 K octets, effectuant des traitements multiprogrammés. En plus des périphériques classiques, comprenant un lecteur de cartes à 400 cartes/minutes, une imprimante à 1000 lignes/minutes, deux dérouleurs de bandes magnétiques et six unités d'une capacité de 30 M octets, il est équipé d'une unité de contrôle de communications gérant quatre écrans alphanumériques et le T1600.

La communication entre les deux ordinateurs a été réalisée par une équipe de chercheurs de l'Ecole des Mines de Saint-Etienne travaillant à la réalisation sur T1600 d'une station de transport pour le réseau Cyclades. Dans le cadre de cette réalisation, ils ont écrit sur T1600 un logiciel qui simule le comportement d'écrans alphanumériques connectés à l'unité de contrôle du P1175.

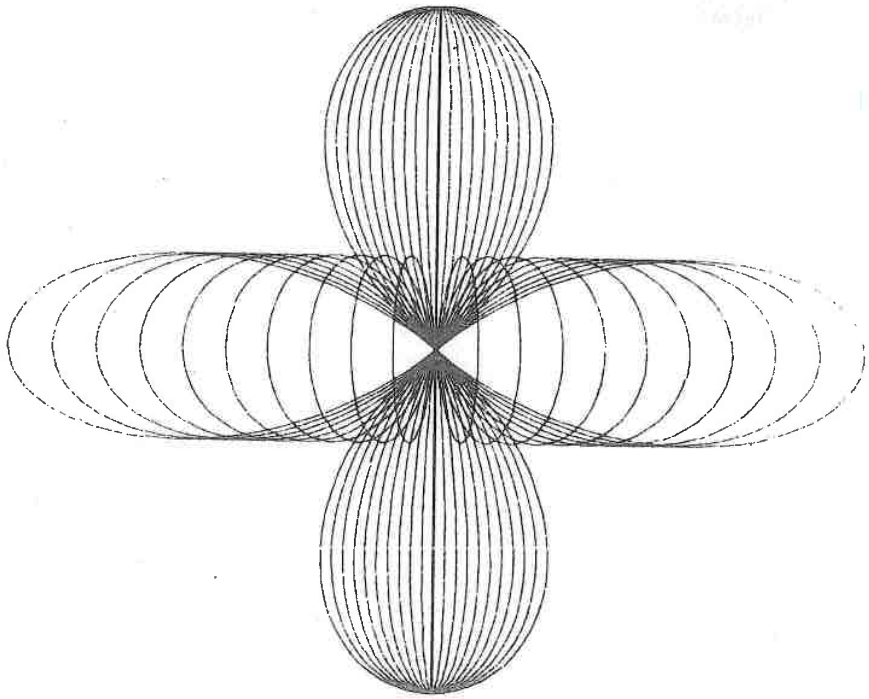
Afin de maintenir grâce au P1175 les programmes qui constituent le code source de l'interprète (ce qui n'était plus possible sur le T1600, du fait de la trop faible capacité de son disque), les auteurs ont écrit un ensemble de programmes permettant l'accès aux bibliothèques du P1175 à partir du T1600.

Cette communication avait ainsi résolu le problème de la conservation des modules et leur mise à jour pouvait avoir désormais lieu sur le P1175, ce qui déchargeait le T1600, arrivé à saturation. Mais ceci nécessitait alors l'utilisation des outils disponibles sur le P1175, malheureusement non conversationnels. Les auteurs ont ainsi été amenés à concevoir et à réaliser avec l'aide de MM. Thomas et Rossi (du centre de calcul), un éditeur de texte conversationnel opérant à partir des écrans et permettant la manipulation aisée des programmes situés sur bibliothèques du P1175. Accessoirement, cet éditeur est devenu un des outils informatiques les plus utilisés par les élèves et les chercheurs de l'Ecole des Mines [E8].

En plus de la conservation et de la manipulation des programmes, le P1175 a encore été mis à contribution pour les générations du système APL. Son macro-processeur très perfectionné a permis la génération automatique de tables pour le processeur de recouvrement, ainsi que l'incorporation automatique dans les programmes en langage PL1600 des blocs de description des données.

Terminons en signalant qu'un étudiant, dans le cadre d'un DEA préparé à l'Ecole des Mines, réalise actuellement sur l'ordinateur P1175 un analyseur syntaxique du langage PL1600. Cet analyseur permettra d'effectuer sur le P1175 en mode conversationnel non seulement l'édition, mais aussi l'analyse et la correction des programmes PL1600 destinés à être exploités sur l'ordinateur T1600.

CONCLUSION



CONCLUSION

Nous avons cherché, dans les pages précédentes, à présenter brièvement les multiples aspects de la réalisation de l'interprète APL sur le T1600.

Pourtant, bien que la conception du langage soit relativement séduisante pour l'esprit, et passe même aux yeux de certains pour la meilleure justification de son existence, la question pourrait se poser de savoir à qui APL est destiné, de mesurer son intérêt...

Conçu à l'origine comme une notation mathématique simplificatrice, APL est devenu, entre les mains d'IBM d'abord, de celles des fanatiques de ce langage ensuite, une arme à double tranchant, une langue à la fois concise et illisible, puissante et hermétique. Certains informaticiens "parlent" aujourd'hui APL avec peut-être la même foi qui animait jadis ceux qui ont conçu, aimé et utilisé l'Espéranto...

Paradoxalement, APL est également apparu comme une réponse, moins pour ceux qui s'intéressent à l'informatique en tant que science, car APL reste d'un faible intérêt théorique, que pour ceux qui cherchent une solution informatique simple à leurs problèmes de tous ordres : la prospérité de services APL, comme celui de la SLIGOS, le prouve amplement.

Alors, langage marginal, artificiel et inefficace, sans intérêt pratique ni théorique ? Ou, au contraire, réponse à tous les problèmes, panacée universelle ? Comme toujours, la vérité se situe entre ces deux bornes, et nous chercherons ici à mieux définir le véritable domaine d'application de ce langage tant par des considérations générales que par des exemples tirés de notre expérience.

Si l'on examine l'évolution des techniques numériques, il faut bien reconnaître que la réalisation de petits calculs, la mise au point de petits algorithmes, a toujours été une source de perte de temps et d'erreurs. Des outils comme la règle à calcul, le calculateur de poche,n'ont apporté que des solutions très partielles : faible précision pour la règle à calcul, pas de possibilités d'itération pour le calculateur de poche; D'autres outils plus évolués ont vu le jour, tels le calculateur programmable (Olivetti, Hewlett-Packard), avec d'autres inconvénients : programmation peu claire, faible puissance de calcul...

La transportabilité du matériel n'étant, en fin de compte, qu'une qualité secondaire, l'appel est fait le plus souvent à un ordinateur, travaillant par lots ou en temps partagé. Pourtant, si l'on excepte le cas idéal (et tout à fait chimérique) de l'utilisateur découvrant chez le constructeur le programme qui lui apportera immédiatement la solution, force est de constater que l'on n'a fait que remplacer un problème (fournir une puissance brute de calcul) par un autre (mettre en jeu cette puissance). Et le non-informaticien, jusqu'alors habitué à son premier problème, se trouve fort démuni devant le second ! De plus, outre la complexité de la mise en oeuvre de l'informatique "classique", il y a un facteur psychologique qui n'est pas à négliger : dans le cas d'un petit travail, l'utilisateur trouve normal d'avoir un faible délai de réponse, au prix d'un effort minimal. Il ne conçoit pas d'attendre une demi-journée pour un gain de quelques décimales (cas du traitement par lot) ni de se voir obligé de reprogrammer, dans le cadre de problèmes plus complexes, des algorithmes connus (exemple l'inversion de matrice) parce qu'il ne sait pas comment mettre en oeuvre les sous-programmes scientifiques fournis par le constructeur ! Enfin, il faut remarquer que l'apprentissage d'un langage comme FORTRAN ou COBOL représente un investissement que les utilisateurs hésitent un peu à fournir...

Quant au temps partagé, s'il est assez couramment disponible de nos jours et d'un emploi plus agréable, il reste en général assez mal conçu : les outils disponibles sont souvent identiques à ceux du traitement par lots (ce qui simplifie la tâche du constructeur) et donc mal adaptés. Si les éditeurs de texte sont conversationnels, les compilateurs le sont rarement, et la nécessité d'utiliser, en outre un langage de contrôle des travaux fait que, dans l'ensemble, ces outils ne sont pas d'une manipulation aisée.

C'est pourquoi sont nés des outils accomplissant toutes ces opérations d'une manière simplifiée, avec des possibilités plus ou moins élaborées. Ils sont (ou se veulent !) d'un maniement facile, rapide à acquérir, mais restent souvent spécifiques à un constructeur, une université, ou un système (BASIC, LSD, LSE, APL,...). Parmi ces outils, une fois éliminés les langages antiques (BASIC...) ou néo-antiques (LSD, LSE...) il faut bien admettre que celui qui offre le maximum d'attraits et de possibilités est encore APL.

L'intérêt du langage a été prouvé par les nombreuses réalisations qui ont été faites. Disponible chez IBM depuis 1968, il est de plus en plus couramment proposé par les autres fabricants d'ordinateurs, tels UNIVAC, BURROUGHS, DIGITAL EQUIPMENT... En France même, la SLIGOS réalise pour le compte de la CII un interprète APL pour IRIS 80, qui sera probablement disponible dès la fin de 1976.

Sur le marché français, toujours, un certain nombre d'interprètes ont déjà été réalisés par les universités et centres de recherches comme à Toulouse (CII 10070 [B2], [B3]), ou au CEA de Saclay (Mitra-15 [B5]). Pourtant, aucune de ces réalisations n'a été par la suite reprise par le constructeur puis proposée à d'autres utilisateurs. Faut-il voir dans ce fait un manque d'intérêt pour APL, ou plutôt un manque d'ouverture des constructeurs (1) envers les universités ?

C'est un peu pour cette raison qu'il nous a semblé intéressant d'effectuer la réalisation sur un mini-ordinateur français comme le T1600, destiné à avoir une grande diffusion, et dont le constructeur était disposé à commercialiser le produit plutôt que sur l'autre ordinateur du Centre de Calcul de Saint-Etienne le Philips P1175, qui appartenait à une gamme dont l'espérance de vie était très faible. De plus, le choix d'un mini-ordinateur reposait sur certaines hypothèses qui sont aujourd'hui devenues presque des lieux-communs.

Il est plus rentable d'acheter un mini-ordinateur, utilisé exclusivement pour faire de l'APL, si possible en temps partagé et sur plusieurs consoles, que de louer un terminal sur gros ordinateur, ou même de faire appel à une société de service spécialisée. Le mini-ordinateur est, en effet, disponible 24 heures sur 24 et d'un emploi plus simple, la lenteur (relative) des interactions est compensée par d'autres avantages, comme l'utilisation en multi-consoles, une accessibilité plus grande des périphériques classiques... Sans parler d'une fiabilité souvent supérieure !

---

(1) la CII en l'occurence...

A ce titre, il est significatif de voir apparaître sur le marché, des mini-calculateurs APL, tels le MCM 70, ou encore des terminaux APL intelligents comme celui annoncé par IBM.

Par ailleurs, le choix d'un mini entrait dans le cadre de projets à long terme pour le développement de l'informatique au sein de l'Ecole des Mines, et même sur le plan régional. Nos centres d'intérêts principaux étaient en particulier les domaines de l'enseignement et de la gestion.

L'enseignement, tout d'abord, du fait de la vocation de l'Ecole des Mines. En effet, un mini-ordinateur, travaillant en temps partagé, suffit largement aux besoins d'un petit groupe d'étudiants. APL répond bien à ce genre de besoins : largement conversationnel, avec d'excellentes facilités de mise au point, il est en outre fort bien assimilable par des élèves ayant subi la formation mathématique des classes préparatoires aux grandes écoles. De plus, seule une faible puissance de calcul est nécessaire : quelques dizaines de minutes de temps d'unité centrale sont une consommation typique pour une après-midi d'utilisation par une dizaine d'étudiants.

Le domaine de la gestion également, non seulement à cause de la formation donnée à une grande partie des élèves mais aussi du fait de nombreuses expériences de collaboration entre l'Ecole des Mines et les entreprises régionales. PME, mini-ordinateurs de gestion, informatique répartie sont des termes qui deviennent de plus en plus étroitement liés, et l'on est obligé de songer au fait que sur les petits ordinateurs dits " de gestion", l'aide apportée à l'utilisateur par le logiciel du constructeur est nettement insuffisante, à tel point, qu'un grand nombre d'applications sont encore programmées en langage d'assemblage !

Les expériences de gestion réalisées en APL ont suscité un vif intérêt de la part des PME, et nous avons pu constater, l'adéquation du langage à la résolution de ce genre de problème, dans la région stéphanoise, où nombre d'entreprises ne pouvaient admettre une utilisation en temps différé (du fait de commandes passées en grande partie par téléphone), et reculaient devant de nouvelles applications, pourtant nécessaires, à cause de la rigidité du système informatique défini : le flottement du franc, par exemple, ayant demandé des modifications "insurmontables" du logiciel pour certaines entreprises travaillant avec les pays étrangers limitrophes.

L'action entreprise par l'Ecole des Mines doit progressivement permettre aux PME d'organiser elles-mêmes leurs systèmes informatiques : développées par des chercheurs ou des élèves, les applications sont mises en oeuvres, d'abord à l'Ecole des Mines elle-même, puis chez le client, au moyen de terminaux APL connectés au T1600. Simultanément, une formation au langage APL des gestionnaires de l'entreprise les ammène à prendre en main l'exploitation. Le but final étant de former des utilisateurs avertis capables d'exploiter et de modifier sans problèmes leurs applications sur leurs propres machines.

Il est sans doute un peu trop tôt pour chiffrer l'aide qu'a pu apporter APL, tant pour la mise en place des applications que pour leurs exploitations, puisque ces projets en sont à l'heure actuelle, au stade du démarrage. De plus, cette comparaison tournera probablement autant autour de l'opposition mini-ordinateur/gros ordinateur que de celle APL/COBOL. Mais, quels que soient les résultats, ces expériences auront toujours apporté une vue et des conceptions nouvelles sur l'informatique dans l'entreprise.

EVOLUTION RECENTE ET PERSPECTIVE A LONG TERME.

Les conséquences de la publicité faite autour du langage APL/1600 ont conduit les auteurs à travailler à l'évolution du produit longtemps après que celui-ci fût devenu opérationnel. Ce fut d'abord pour des raisons de compatibilité avec des implémentations "de luxe", telles APL-SV ou APL-PLUS, puis pour tenir compte des désirs des premiers utilisateurs tant internes à l'Ecole des Mines, qu'externes (clients de la Télémécanique), ou encore pour permettre l'accès à des périphériques non standard (tables traçantes, tablettes graphiques, etc...)

APL/1600 est actuellement exploité en deux versions, mono-console sous les systèmes BOS/D et dérivés, et multi-consoles sous le système TSM.

La version mono-console est opérationnelle depuis Juin 1974 et proposée en clientèle par la Télémécanique depuis Mars 1975. L'utilisation de cette version est très souple, puisque son exploitation peut se faire aussi bien comme processeur normal du système (compilateur, éditeur de lien, etc...) que sous la forme de tâche, simultanément avec le traitement par lot. Son existence a eu le mérite de montrer très tôt l'intérêt de la réalisation par l'attention que lui portaient les utilisateurs potentiels à l'occasion de manifestations comme les Sicobs de 1974 et 1975 ou d'autres expositions locales.

La version multi-consoles est opérationnelle à l'Ecole des Mines depuis septembre 1975. Elle est à la disposition des utilisateurs internes depuis Janvier 1976, à raison d'un minimum de deux demi-journées par semaine. En plus de la console opérateur, cette version gère actuellement quatre terminaux APL de types différents (écrans et machines à écrire). La configuration sera prochainement étendue à dix terminaux.

Le système d'exploitation sous lequel elle fonctionne, TSM a été à l'origine écrit pour le langage LSE puis modifié pour s'adapter au langage BASIC. Les auteurs n'ont que très légèrement modifié TSM en supprimant notamment du système les traitements spécifiques à BASIC et en introduisant quelques possibilités intéressantes comme la banalisation de tous les périphériques de la configuration. En fonction de cette expérience, la Télémécanique travaille à un nouveau système, à la fois plus proche du système BOS/D et mieux adapté à un langage comme APL. Ce produit sera probablement exploitable dès octobre 1976 sur SOLAR 16/40 et 16/65. Il pourra alors remplacer le système utilisé à l'Ecole des Mines qui aura été porté à seize consoles, environ, vers cette date.

Indépendamment de l'évolution du support de l'interprète, des modifications beaucoup plus importantes de l'interprète lui-même sont possibles, au niveau des spécifications tant internes qu'externes. Nous allons les passer rapidement en revue, en indiquant leur intérêt ainsi que les problèmes techniques qui se posent.

Du point de vue spécifications externes, deux axes de développement sont apparus nécessaires au cours de l'utilisation des réalisations actuelles :

- introduction d'une double précision pour les nombres réels
- utilisation d'un système de stockage de grandes quantités d'information.

- Double précision

Aussi bien pour des calculs de gestion que pour des calculs scientifiques, le nombre de chiffres significatifs, du fait d'une représentation interne des nombres sur 32 bits, limite les applications. Mais la double précision (64 bits) ne peut s'envisager qu'avec un opérateur câblé en cours de conception à la Télémécanique Electrique : un opérateur simulé coûterait trop cher en temps d'unité centrale et nécessiterait une réécriture presque complète d'une grande partie des programmes.

Pour diminuer l'encombrement des données, il est alors nécessaire d'introduire un type de représentation interne sous format entier (16 bits) : ceci complique les opérateurs scalaires par l'introduction d'un système de conversion automatique plus complexe que celui actuellement en place pour les logiques et les réels.

Une telle modification assurera 15 chiffres significatifs au lieu des 7 actuels.

- Stockage d'information.

Si nous n'employons pas le terme de fichier c'est que ce stockage peut s'envisager sous plusieurs formes. Tout d'abord, il est possible d'écrire un interface minimum avec un système de fichier classique : c'est ce qui est expérimenté à la Télémécanique avec le système de fichier FMS (File Management System). L'utilisateur dispose alors de fonctions système, lui permettant tous les types d'accès reconnus par FMS.

Une autre possibilité, réalisée à l'Ecole des Mines, est de simuler ce système de fichier en utilisant une mémoire virtuelle. L'application tourne actuellement avec la simulation de l'opérateur"mémoire-virtuelle". Elle a l'avantage de faciliter l'écriture des primitives identiques à celle d'APL * PLUS et le partage des fichiers ou des variables entre plusieurs utilisateurs.

Enfin cet opérateur de mémoire virtuelle permet d'envisager l'implémentation de structures de données plus complexes que les tableaux APL, se rapprochant d'une base de données, avec création d'un certain nombre de primitives de base, pour la définition et le traitement de ces structures. Ce dernier point est d'ailleurs l'objet du travail d'un chercheur de l'Ecole des Mines.

Parallèlement à ces modifications ou extensions qui seront sensibles à l'utilisateur, d'autres parties de l'interprète feront l'objet d'optimisation, qui, tout en restant transparentes à l'utilisateur, sont destinées à améliorer les performances ponctuelles de l'interprète.

Certaines de ces modifications sont déjà réalisées ou en cours de réalisation. La paramétrisation de la gestion de mémoire a permis d'augmenter la taille d'une zone de travail jusqu'à 256 K octets. Des optimisations de stockage des fonctions utilisateur et des fonctions système vont permettre des gains en place mémoire appréciables.

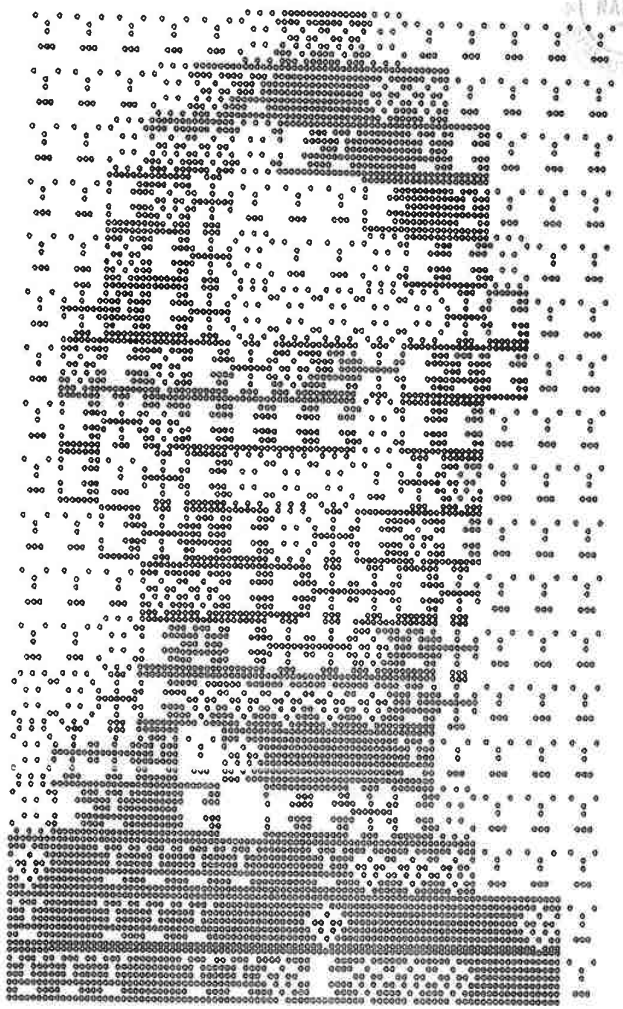
Un nouveau type de fonction a été crée : les fonctions intégrées. Ces fonctions systèmes, rangées dans une zone spéciale, automatiquement chargées dans la zone de travail lors de leur utilisation, n'occupent de la place qu'au moment de leur exécution et ne nécessitent aucun erdre de recopie.

Des mécanismes de mesures vont être mis en place, pour tester la structure de recouvrement en utilisation multi-consoles d'une part, et les performances du système de temps partagé d'autre part. Elles permettront de mieux définir les paramètres du prochain système.

Enfin, il serait possible d'utiliser cette version du système APL pour détecter les points de l'interprétation qu'il serait souhaitable d'optimiser, et même éventuellement pour tester plusieurs méthodes d'interprétation en remplaçant simplement le programme qui effectue celle-ci, sans modifier toute la structure du système APL complet.

En conclusion, nous pouvons dire que le but de notre travail, doter un petit ordinateur d'un outil particulièrement puissant APL, a été largement atteint, mais qu'il est toujours possible de parfaire cet outil et d'étendre ses possibilités d'une part,et d'autre part d'en promouvoir l'usage dans des domaines aussi divers que l'enseignement, la gestion ou la recherche scientifique, où nous pensons qu'il peut se révéler extrêmement utile.

ANNEXES



ANNEXES

Cette partie est consacrée à deux applications typiques,faisant intervenir des programmes qui tirent au mieux parti des caractéristiques du langage APL1600.

Plutôt que d'obtenir des programmes performants,le but de ces applications était de montrer la facilité avec laquelle de moyennes réalisations pouvaient être écrites et entiérement mises au point, pour un minimum de temps d'analyste-programmeur:une semaine pour le compilateur ZPL, deux jours pour le systéme de gestion de fichier.

La premiére de ces applications consiste en l'écriture d'un compilateur du langage APLGOL.Ce langage a été décrit dans (1) L'idée qui a présidé à sa conception était d'augmenter la puissance du langage APL en lui ajoutant une structure de bloc.

L'article original prévoyait deux syntaxes possibles,l'une voisine d'ALGOL 60,l'autre de PL1.Nous avons préfèrer réaliser ,pour notre part un compilateur analysant une syntaxe identique à celle de PL1600 (en fait trés voisine de celle de PL1).

La seconde application est un systéme de gestion de fichier écrit en APL et reposant sur quelques fonctions systéme simulant un opérateur cablé de mémoire virtuelle.La syntaxe choisie est celle (2) d'APL-PLUS,systéme de fichier utilisé sous un systéme APL réalisé sur IBM 360.Cette gestion de fichier a permis de tester une utilisation possible en APL d'une mémoire virtuelle permettant de ranger ,en dehors des zones de travail APL, des grandes quantités d'informations.

(1) APLGOL,an experimental structured programming language.
R.A. KELLEY IBM j.Res.Develop. JANVIER 1973

(2) Voir B7

LE LANGAGE ZPL

A-PRESENTATION:

ZPL est un langage d'aspect algolique,dont les instructions de base sont celles d'APL.

Le compilateur ZPL est en fait un préprocesseur, transformant un texte source en une ou plusieurs fonctions APL.De ce point de vue, un programme ZPL est ,une fois compilé, aussi performant que le programme APL équivalent.

B-SYNTAXE:

Un certain nombre de mots réservés ont été introduits dans le langage.Ils commencent par une lettre soulignée.Ce sont:

BEGIN	BY	CASE	DO
ELSE	END	FOR	IF
OF	OUT	PROCEDURE (ou PROC)	
REPEAT	RETURN	TO	THEN
UNTIL	WHILE		

C-FORME BNF:

```
<program>::= PROCEDURE <expression> ;<statment  list> END  (4)

<statment  list>::=<lstatment > |<lstatment> ;<statment  list>

<lstatment >::=<label>:<statment> | <statment>

<statment ::=<program>|<if stat.>|<for stat.>|<case stat.>|
             BEGIN <statment list> END |
             REPEAT <statment list> UNTIL <expression> END |
             WHILE <expression> DO <statment list> END |
             RETURN <expression> | RETURN |<expression>|<null>
<if stat.>::= IF <expression> THEN <statment list> END |
             IF <expression> THEN <statment list> ELSE
            <statment list> END
```

<for stat.>::= FOR <expression> TO <expression> DO <statment list> END|
FOR <expression> BY <expression> TO <expression>
DO <statment list> END

<case stat.>::= CASE <expression> OF <statment list> END|
CASE <expression> OF <statment list> OUT
<statment list> END

<expression>::= expréssion APL (1)(2)(3)

<label>::= identificateur APL

<null>::= expréssion vide

(1): Dans le cas d'une sortie mixte remplacer ";" par le symbole ∪

ex: *'VALEUR DE A';A*

s'écrit *'VALEUR DE A'∪A*

(2): Dans le cas d'une déclaration de procédure avec variables locales remplacer de même ";" par le symbole "," ou "∪".

ex: *PROCEDURE R←TRUC A,X,LG,Z;*

(Ces restrictions proviennent de l'usage du point-virgule comme délimiteur d'expréssions.Mais une indexation s'écrit normalement : *A[I;J]*).

(3): Les commentaires ne sont pas générés.

(4): La chaîne d'entrée doit se terminer par un "." ou un ";".

D-MISE EN OEUVRE:

Elle necessite une zone de travail de 24K minimum.Pour cela,on écrira au sign-on:

```
)nnnn      24
 nº de compte
```

Apres la connection on chargera la zone de travail APLGOL:

```
)LOAD 33 APLGOL
```

Il est possible de définir un texte source avec la fonction DEF:

ex:

```
      CHAINE←DEF
[001]
```

La fonction lit des chaînes de caractéres et les concaténe entre elles ,séparées par des "cr","lf" .L'insertion s'achéve sur une chaîne vide.

La compilation s'éffectue par un appel à la fonction APLGOL.

Ex:

Définition du texte source :

```
      FTRUC←DEF
[001]  PROCEDURE TRUC,A,I,S;
[002]     A←20?20; S←0;
[003]     FOR I←1 BY 3 TO ρA DO
[004]        S←S+A[I]
[005]     END;
[006]     'SOMME ' ∪ S;
[007]  END.
[008]
```

Le texte source est rangé dans la variable FTRUC:

```
      FTRUC
PROCEDURE TRUC,A,I,S;
   A←20?20; S←0;
   FOR I←1 BY 3 TO ρA DO
      S←S+A[I]
   END;
   'SOMME ' ∪ S;
END.
```

Compilation et essais d'éxecution:

```
      APLGOL FTRUC
PROCEDURE NAME : TRUC
OBJECT PROGRAM :
    ∇ TRUC;A;I;S
[1]   A←20?20
[2]   S←0
[3]   I←1
[4]  L001:→((ρA)<I)/L002
[5]   S←S+A[I]
[6]   →L001αI←I+3
[7]  L002:'SOMME ';S
    ∇
FIN DE COMPILATION.

      TRUC
SOMME 79
      TRUC
SOMME 85
```

Voici un autreexemple un peu plus complexe ,montrant la possibilité d'imbriquer des procédures dans une procédure principale,ici test d'une procédure récursive de calcul de factorielle:

```
      TX←DEF
[001]  PROCEDURE PRINCIPAL;
[002]     PROCEDURE R←FACT N;
[003]        IF N≤1 THEN
[004]           R←1
[005]        ELSE
[006]           R←N×FACT N-1
[007]        END;
[008]     END;
[009]     'FACTORIELLE 3 = ' ∪ FACT 3;
[010]     'FACTORIELLE 7 = ' ∪ FACT 7;
[011]  END.
[012]
      TX
PROCEDURE PRINCIPAL;
   PROCEDURE R←FACT N;
      IF N≤1 THEN
         R←1
      ELSE
         R←N×FACT N-1
      END;
   END;
   'FACTORIELLE 3 = ' ∪ FACT 3;
   'FACTORIELLE 7 = ' ∪ FACT 7;
END.

      APLGOL TX
PROCEDURE NAME : FACT
OBJECT PROGRAM :
    ∇ R←FACT N
[1]    →(~N≤1)/L001
[2]    R←1
[3]    →L002
[4]   L001:R←N×FACT N-1
[5]   L002:→⍳0
    ∇
PROCEDURE NAME : PRINCIPAL
OBJECT PROGRAM :
    ∇ PRINCIPAL
[1]    'FACTORIELLE 3 = ';FACT 3
[2]    'FACTORIELLE 7 = ';FACT 7
    ∇
FIN DE COMPILATION.

      PRINCIPAL
FACTORIELLE 3 = 6
FACTORIELLE 7 = 5040

      !3
6
      !7
5040
```


Quelques fonctions de la zone de travail APLGOL:

```
      ∇APLGOL[⎕]∇
    ∇ APLGOL IS;ETAT;LG;LLNB;L;LN;STAK;T;TY;TRANS;PROCNAME;LB;SVC;SVL;EXP;ID
[1]   INIT
[2]  MAIN:TRANS←TSY[;1]⍳ETAT
[3]   SKIPSPACES
[4]   TY←TYPE
[5]  INNER:→SYNT×⍳ETAT≠TSY[TRANS;1]
[6]   →OK×⍳(T=0)∨TY=T←TSY[TRANS;3]
[7]   →INNER,TRANS←TRANS+1
[8]  OK:→NOAB×⍳0=T
[9]   ABSORBE
[10] NOAB:→NOPUSH×⍳0=T←TSY[TRANS;4]
[11]  STAK←T,STAK
[12]  →EXEC,ETAT←TSY[TRANS;2]
[13] NOPUSH:→NOPOP×⍳0≠T←TSY[TRANS;2]
[14]  →0×⍳0=⍴,STAK
[15]  →EXEC,STAK←,1↓STAK⍺ETAT←1⍴STAK
[16] NOPOP:ETAT←T
[17] EXEC:→GEN×⍳0=T←TSY[TRANS;5]
[18]  ⍞'P',⍕T
[19] GEN:GENERE
[20]  →MAIN×⍳0≠ETAT
[21]  'FIN DE COMPILATION.'
[22]  →0
[23] SYNT:'ERREUR DE SYNTAXE, LIGNE ',⍕LLNB
[24]  'ETAT :  ';ETAT;'    TRANS :  ';TRANS
[25]  T,(30↑IS),T←5⍴'.'
    ∇
```

```
      ∇ABSORBE[⎕]∇
    ∇ ABSORBE
[1]   LN←0⍺IS←LN↓IS
    ∇
      ∇SKIPSPACES[⎕]∇
    ∇ SKIPSPACES
[1]   →0×⍳' '≠1⍴IS
[2]   →1⍺IS←1↓IS
    ∇
```

```
      ∇TYPE[⎕]∇
    ∇ R←TYPE
[1]   →(2 3 5 6 7 9 10 11 13 15 17 19 20,R←0)[';BCDEFIOPRTWU'⍳1⍴IS]
[2]   →0,R←LN←1
[3]   →0×⍳R←2×∧/'BY'=(LN←2)⍴IS
[4]   →0,R←3×∧/'BEGIN'=(LN←5)⍴IS
[5]   →0,R←4×∧/'CASE'=(LN←4)⍴IS
[6]   →0,R←5×∧/'DO'=(LN←2)⍴IS
[7]   →0×⍳R←6×∧/'END'=(LN←3)⍴IS
[8]   →0,R←7×∧/'ELSE'=(LN←4)⍴IS
[9]   →0,R←8×∧/'FOR'=(LN←3)⍴IS
[10]  →0,R←9×∧/'IF'=(LN←2)⍴IS
[11]  →0×⍳R←10×∧/'OF'=(LN←2)⍴IS
[12]  →0,R←18×∧/'OUT'=(LN←3)⍴IS
[13]  →0×⍳R←11×∧/'PROCEDURE'=(LN←9)⍴IS
[14]  →0,R←11×∧/'PROC'=(LN←4)⍴IS
[15]  →0×⍳R←12×∧/'REPEAT'=(LN←6)⍴IS
[16]  →0,R←13×∧/'RETURN'=(LN←6)⍴IS
[17]  →0×⍳R←14×∧/'TO'=(LN←2)⍴IS
[18]  →0,R←15×∧/'THEN'=(LN←4)⍴IS
[19]  →0,R←16×∧/'WHILE'=(LN←5)⍴IS
[20]  →0,R←17×∧/'UNTIL'=(LN←5)⍴IS
    ∇
```

FONCTIONS DE FICHIERS TYPE APL-plus

INTRODUCTION:

L'existence d'un opérateur de mémoire virtuelle cablé sur le T1600 et le SOLAR 16 a permis de developper aisement un systéme de gestion de fichiers sur l'interpréteur APL1600.

Le support de ces fichiers est un espace virtuel unique.

Ce systéme supporte 127 fichiers,chaque fichier peut avoir jusqu'à 2047 composantes numérotées de 1 à 2047.

La longueur maximale de chaque composante est fixée à 4096 mots de 16 bits soit donc: 2047 éléments de type numérique,
8191 éléments de type caractére,
32767 éléments de type logique.

Un fichier est accessible par un nom de fichier.Ce nom a au plus 16 caractéres,il est précédé du numero du propriétaire séparé par un espace (si le numéro de l'utilisateur est différent de celui du propriétaire);il est suivi eventuellement d'un mot de passe (128 caractéres au maximum) séparé par le caractére ":".

Syntaxe d'un nom de fichier:

(numero de compte) NOM (:mot de passe).

Au cours de l'utilisateur du fichier ,ce fichier est repéré par un numéro choisi par l'utilisateur et n'ayant une portée que jusqu'à la fermeture du fichier;ce numéro de fichier est compris entre 1 et 127.

INITIALISATION DE LA ZONE AFFECTEE A LA GESTION DE FICHIER:

L'initialisation se fait par un appel de la fonction OPENEV.

FERMETURE DE LA ZONE AFFECTEE A LA GESTION DE FICHIER:

La fermeture se fait par un appel de la fonction CLOSEV.

CREATION D'UN FICHIER:

La fonction FCREATE permet de créer et d'ouvrir un fichier :

nom de fichier FCREATE numero de fichier

Ex:

'TOTO' FCREATE 115

'781 *ARTHUR :PASS' FCREATE* 18

DESTRUCTION D'UN FICHIER:

Un fichier ne peut être détruit que s'il aété ouvert.

FERASE numéro de fichier

Ex:

FERASE 115

OUVERTURE D'UN FICHIER:

La fonction FOPEN permet d'attribuer un numéro de fichier et de s'attribuer l'accés au fichier.

nom de fichier FOPEN numero de fichier

Ex:

'TOTO' FOPEN 115

'781 ARTHUR :PASS' FOPEN 18

FERMETURE D'UN FICHIER:

La fonction FCLOSE permet de fermer un fichier et d'en libérer l'accés.

FCLOSE numero de fichier

Ex:

FCLOSE **117**

L'argument peut être un vecteur:il est possible de fermer plusieurs fichiers à la fois.

LECTURE D'UNE COMPOSANTE:

La lecture d'une composante se fait en indiquant le numero de fichier et le numéro de composante:

FREAD numero de fichier,numero de composante

Ex:

R←FREAD 118 3

ECRITURE D'UNE COMPOSANTE:

Comme pour la lecture ,en plus de la composante elle-même on indique le numéro du fichier et de la composante concernés.

composante FWRITE numéro de fichier,numero de composante

Ex:

(⍳1000) FWRITE 118 3

'CECI EST UNE CHAINE DE CARACTERES ' FWRITE 118 4

EFFACEMENT D'UNE COMPOSANTE:

On peut éffacer selectivement une composante d'un fichier.

FDROP numéro de fichier,numéro de composante

Ex:

FDROP 118 3

LISTE DES FICHIERS D'UN NUMERO DE COMPTE:

FLIB numéro de compte

Ex:

FLIB 33

JOJO

JULES

LISTE DES FICHIERS OUVERTS:

FNAMES

LISTE DES NUMEROS UTILISES:

FNUMS

TAILLE DU FICHIER:

FSIZE numéro de fichier

CHANGEMENT DE NOM OU DE MOT DE PASSE :

nom de fichier FRENAME numéro de fichier

Ex:

'TEMPS :MOI' FRENAME 118

```
      ∇FOPEN[⎕]∇
    ∇ N FOPEN PR;US;MAC;T;P
[1]   1 ⍎FERROR(1≠TY N)∨2≠TY PR
[2]   2 ⍎FERROR 1≠⍴⍴N
[3]   3 ⍎FERROR(1>⍴N)∨~(⍴PR←,PR)∊1 2
[4]   PR←1⍴PR⍺P←PR[2]⍺PR←2↓PR
[5]   1 ⍎FERROR(0≥PR)∨(128≤PR)∨PR≠⌊PR
[6]   8 ⍎FERROR PR∊⍎F[;1]
[7]   10 ⍎FERROR 4=1⍴⍴⍎F
[8]   N←1↓N←(T∨1⌽T←N≠' ')/N←' ',N
[9]   →L2×⍳~' '∊N
[10]  US←⍎(T←N⍳' ')⍴N
[11]  N←T↓N
[12]  9 ⍎FERROR ' '∊N
[13]  →L3
[14] L2:US←⍎US
[15] L3:MAC←VR 0 0,US
[16]  →L1×⍳1=TY MAC
[17]  MAC←0 24⍴''
[18] L1:6 ⍎FERROR~1∊T←(0 ¯8↓MAC)⍲.=16⍴N←(16↑N),1 TY,P
[19]  11 ⍎FERROR∨/N[17 18 19 20]≠MAC[T←T/⍳⍴T;17 18 19 20]
[20]  T←2 TY 20↓,MAC[T;]
[21]  ⍎F←⍎F,[1]PR,T
[22]  ⍎FL←⍎FL,[1]0 TY 27↓VR 0,T,0
      ∇
```

```
      ∇FERASE[⎕]∇
    ∇ FERASE N;US;PR;MAC;I;T
[1]   1 FERROR 2≠TY N
[2]   3 FERROR 1≠ρN←,N
[3]   1 FERROR(0≥N)∨(128≤N)∨N≠⌊N
[4]   5 FERROR(1ρρF)<N←F[;1]⍳N
[5]   PR←F[N;2]
[6]   T←,FL[N;]
[7]   N←N≠⍳1ρρF
[8]   F←N⌿F
[9]   FL←N⌿FL
[10]  US←⍎6ρN←VR 0,PR,0
[11]  MAC←VR 0 0,US
[12]  (0 0,US)VW(~(0 ¯8↑MAC)∧.=16⍴7↑N)⌿MAC
[13]  I←VR 0 0 0
[14]  I[PR]←1=0
[15]  0 0 0 VW I
[16]  (0,PR,0)VW⍳0
[17] L1:→0×⍳1024<I←T⍳1
[18]  (0,PR,I)VW⍳0
[19]  T[I]←0=1
[20]  →L1
      ∇
      ∇FERROR[⎕]∇
    ∇ N FERROR X;E;F
[1]   →0×⍳~X
[2]   →(L1+1)×⍳1=TY N
[3]   N←N×(N>0)∧(N<12)∧N=⌊N
[4]   F←'FILE '⍺E←' ERROR'
[5]  L1:→1+L1+N
[6]   ⍎31⍺⎕←'WRONG CALL'
[7]   ⍎31⍺⎕←F,'DOMAIN',E
[8]   ⍎31⍺⎕←F,'RANK',E
[9]   ⍎31⍺⎕←F,'LENGTH',E
[10]  ⍎31⍺⎕←F,'INDEX',E
[11]  ⍎31⍺⎕←F,'NOT OPENED'
[12]  ⍎31⍺⎕←F,'NOT FOUND'
[13]  ⍎31⍺⎕←F,'EXISTS ALREADY'
[14]  ⍎31⍺⎕←F,'NUMBER IN USE'
[15]  ⍎31⍺⎕←F,'SYNTAX',E
[16]  ⍎31⍺⎕←F,'RESERVATION',E
[17]  ⍎31⍺⎕←F,'ACCESS',E
      ∇
      ∇FWRITE[⎕]∇
    ∇ A FWRITE PR;T;I;N
[1]   1 FERROR 2≠TY PR
[2]   3 FERROR 2≠ρPR←,PR
[3]   1 FERROR(0≥N)∨(128≤N)∨N≠⌊N←PR[1]
[4]   5 FERROR~N∊F[;1]
[5]   4 FERROR(0≥I)∨(1024<I)∨I≠⌊I←PR[2]
[6]   T←F[;1]⍳N
[7]   (0,F[T;2],I)VW A
[8]   FL[T;I]←1=1
      ∇
```

```
      ∇VR[⎕]∇
    ∇ R←VR N
[1]   R←(M∧.=N←¯2↑N)⍳1
[2]   →L1×⍳R>1ρρM
[3]   R←⍎'V',∇R
[4]   →0
[5]  L1:R←⍳0
      ∇
      ∇VW[⎕]∇
    ∇ N VW A;R
[1]   R←(M∧.=N←¯2↑N)⍳1
[2]   →L1×⍳R≤1ρρM
[3]   M←M,[1]N
[4]  L1:⍎'V',(∇R),'←A'
      ∇
```



```
      ∇FCLOSE[⎕]∇
    ∇ FCLOSE N;P;I;T;US;MAC;A
[1]   1 FERROR 2≠TY N
[2]   3 FERROR~(ρN←,N)∊1 2
[3]   I←1ρNαP←¯1↑N
[4]   1 FERROR(0≥I)∨(128≤I)∨I≠⌊I
[5]   5 FERROR(1ρρF)<I←F[;1]⍳I
[6]   T←F[I;2]
[7]   MAC←27ρVR T←0,T,0
[8]   T VW MAC,A←,1 TY FL[I;]
[9]   I←I≠⍳1ρρF
[10]  F←I⌿F
[11]  FL←I⌿FL
[12]  →0×⍳1=ρN
[13]  N←MAC
[14]  MAC←VR 0 0,US←⍎6ρN
[15]  I←((0 ¯4↑MAC)∧.=7↑N)⍳1
[16]  MAC[I;17 18 19 20]←N[24 25 26 27]←1 TY,P
[17]  (0 0,US)VW MAC
[18]  T VW N,A
      ∇
      ∇FCREATE[⎕]∇
    ∇ N FCREATE PR;US;MAC;T;I;P
[1]   1 FERROR(1≠TY N)∨2≠TY PR
[2]   2 FERROR 1≠ρρN
[3]   3 FERROR(1>ρN)∨~(ρPR←,PR)∊1 2
[4]   PR←1ρPRαP←PR[2]αPR←2↑PR
[5]   1 FERROR(0≥PR)∨(128≤PR)∨PR≠⌊PR
[6]   8 FERROR PR∊F[;1]
[7]   10 FERROR 4=1ρρF
[8]   N←1↑N←(T∨1⌽T←N≠' ')/N←' ',N
[9]   9 FERROR ' '∊N
[10]  US←US
[11]  MAC←VR 0 0,US
[12]  →L1×⍳1=TY MAC
[13]  MAC←0 24ρ''
[14] L1:7 FERROR 1∊T←(0 ¯8↑MAC)∧.=16ρN←(16↑N),1 TY,P
[15]  I[T←(I←VR 0 0 0)⍳0]←1=1
[16]  0 0 0 VW I
[17]  (0 0,US)VW MAC,[1]N,1 TY,T
[18]  (0,T,0)VW(6 0∇US),' ',N,1 TY 1024ρ1=0
[19]  F←F,[1]PR,T
[20]  FL←FL,[1]0=1
      ∇
```

```
      ∇FNAMES[⎕]∇
    ∇ R←FNAMES;I
[1]   R←0 23ρ''⍺I←0
[2]  L1:→0×⍳(1ρρF)<I←I+1
[3]   R←R,[1]23ρVR 0,F[I;2],0
[4]   →L1
      ∇
      ∇FNUMS[⎕]∇
    ∇ R←FNUMS
[1]   R←F[;1]
      ∇
```

```
      ∇FLIB[⎕]∇
    ∇ R←FLIB N
[1]   1 FERROR 2≠TY N
[2]   3 FERROR 1≠ρN←,N
[3]   1 FERROR(0≥N)∨N≠⌊N
[4]   →L1×⍳2=TY R←VR 0 0,N
[5]   R←0 ¯8↑R
[6]   →0
[7]  L1:R←0 16ρ''
      ∇
```

BIBLIOGRAPHIE

1 - LA DEFINITION DU LANGAGE.

[A1] A Programming Language - K.E. IVERSON - 1962 John Wiley & sons

[A2] Le langage APL - B. ROBINET - 1971 - Editions Technip

[A3] APL/1130 Reference manual - R.S. CARBERRY, L.M. BREED, S.M. ROUCHER A.G. NEMETH, C.H. BRENNER - 1969 IBM

[A4] APL/360 Reference manual - K.E. IVERSON, A.D. FALKOFF - 1968 IBM Trad. C. Pinta, Y. Le borgne - 1969 IBM

[A5] APL-SV/360. - Manuels de présentation - ? - 1973 IBM

[A6] ALGEBRA AS A LANGUAGE/K.E. IVERSON/colloque APL de l'IRIA Sep. 1971

2 - LES IMPLEMENTATIONS D'APL

[B1] Notions sur le système APL/360 - P. Chomat - Sept. 1971 IUT Inf. Grenoble

[B2] Bulletin spécial de l'IRIA APL Mars 1971 :
- Les implémentationsd'APL - Y. Reynaud
- APL CII 90-80 - P. Maurice et P.C. Scholl

[B3] L'interprétation d'APL - E. LEVY - E. MARTIN et Y. REYNAUD - Mai 1972 U.E.R. d'informatique, Toulouse

[B4] APL/4004 implémentation - Eric B. IVERSON - APL Congress 1973

[B5] An APL interpreter for mini-Computers - Y.AMRAM, B. de COSNAC , J.L. GRANGER A. SMOUCOVIT - Centre d'étude nucléaires de Saclay 1973

[B6] Une gestion de mémoire pour APL - F.D. ARMINGAUD - Ecole d'été d'informatique de Neuchatel, 1972.

[B7] Description des fichiers du système APL * PLUS * FILE - SLIGOS INFORMATIQUE

[B8] APL/1600 rapport préliminaire - F.D. ARMINGAUD - G. CHANDLER 1974

3 - L'OPTIMISATION DE L'INTERPRETATION D'APL.

[C1] An APL machine - P. ABRAMS - Stanford University - Fev. 1970

[C2] A micro programmed APL machine - R. ZAKS - Thèse Ph.D. - Mai 1972

[C3] Un interpréteur APL avec génération et réutilisation de code machine. G. BATTAREL, M. DELBREIL et P. KALFON - Colloque APL de l'IRIA sept. 1971

[C4] Efficient évaluation of array Subscripts of array - A. Hassit et L.E. Lyon - IBM J R & D, Jan 1972

[C5] Un interpréteur APL optimisé - G. BATTAREL, M. DELBREIL, D. TUSERA - rapport de recherche 32 de l'IRIA, Oct. 1973.

[C6] Implémentation of High Level Language Machine - A. HASSIT, J.W. LAGESCHULTE, L.E. Lyon - Communications of the ACM, avril 1973.

[C7] Dynamic memory management from APL-Like languages - ZAKS

4 - LES UTILISATIONS D'APL

[D1] Revue APL QUOTE-QUAD, in SIGPLAN NOTICES - ACM

[D2] Software graphique pour Tektronix 4013 - S. BARON, S.R. BARTELS et G. MARTIN - CISI SACLAY, Mars 1974

[D3] Dossier APL - La presse informatique - 12 Mars 1973

[D4] Expériences d'enseignements avec APL - Colloque APL de l'IRIA, sept. 1971, Pages 251-307 et 445-460

[D5] Computer display of the derived polytopes - P. ABRAMS et W.M. Mc KEEMAN - Revue 36 de la CEGOS-INFORMATIQUE

[D6] APL in a two-step programming technique for developing complex programs - G. DEMARS, E. GIRARD, J.C. RAULT - APL Congress 73

5 - BIBLIOGRAPHIE GENERALE

[E1] Manageable software Engineering - R.W. Bemer - Proceeding of the third symposium on computer and Information Sciences, Dex 1969.

[E2] Les structures d'information et leurs représentation en mémoire - C. PAIR - Ecole d'été d'informatique de l'AFCET, Alès 1971.

[E3] L'allocation dynamique de la mémoire des ordinateurs - R. EHRMANN - DUNOD 1972

[E4] Cours de structures de données - R. MAHL - Ecole des Mines de Saint-Etienne 1973

[E5] Analyse d'un algorithme de gestion simultanée mémoire centrale disque de pagination - E. GELENBE, J. ENFANT, A. BRANDWAJN, D. POTIER - Rapport de recherche de l'IRIA , 19 mai 1973.

[E6] MCM/70, APL with 16-22 K bytes - M. Kutt - Datamation, Nov. 1973

[E7] Structures de données et algorithmes fondamentaux - C. PAIR. ENSMIM NANCY 1974

[E8 Un éditeur de texte conversationnel manuel d'utilisation - J.J. GIRARDOT F. MIREAUX, M. ROSSI, R. THOMAS

6 - DOCUMENTATION T1600

[F1] BOS - Manuel de référence - Télémécanique

[F2] BOS/D-Manuel de référence - Télémécanique

7 - APL/1600

7.1. - Rapport de contrats

[G1] Rapport final du 72-80-06

[G2] Rapport intermédiaire du 74-80-09

[G3] Rapport final du 74-80-09

7.2. - Manuels

[G4] Manuel de référence (1974)

[G5] Manuel d'exemples

[G6] Guide opérateur

[G7] Manuel de maintenance

[G8] Manuel d'utilisation des fonctions systèmes

[G9] Manuel d'utilisation de la fonction FMT

[G10] Manuel de référence (1976)

[G11] Une introduction au langage APL

7.3. - Applications

[G12] Analyse de données sous APL - J.P. SABY