MASTER EN SCIENCES INFORMATIQUES

Implémentation des suites et ensembles pour un langage de programmation permettant l'expression d'assertions

Wenzi, Menayame

Award date:
1977

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix - NAMUR
INSTITUT D'INFORMATIQUE

IMPLEMENTATION DES SUITES ET ENSEMBLES POUR UN LANGAGE DE PROGRAMMATION PERMETTANT L'EXPRESSION D'ASSERTIONS

WENZI Menayame

Mémoire présenté en vue de l'obtention du grade de licencié et maître en informatique.

Année académique 1976 - 1977

Que toutes les personnes qui ont contribué de loin ou de près à l'élaboration de ce mémoire trouvent ici l'expression de mes sincères remerciements. Nous pensons tout spécialement à Monsieur Baudouin LECHARLIER, promoteur de ce mémoire, pour l'intérêt qu'il a porté à ce travail et pour ses conseils fructueux.

TABLE DES MATIERES

I. INTRODUCTION

Ce mémoire s'insère dans le cadre de travail proposé par D. FISETTE ((1)) : fournir un environnement logiciel permettant la vérification de certaines propriétés des programmes - à savoir les assertions (conditions pré-post et invariants, essentiellement) - pendant leur exécution.

Nous rappelons que ce cadre de travail visait essentiellement la classe des différents programmes du cours "SEMINAIRE DE PROGRAMMATION" donné en 1ère licence et que c'est dans ce contexte bien déterminé que D. FISETTE a défini un langage formel permettant l'expression des assertions.

Comme il s'avère quasiment impossible qu'un langage formel puisse exprimer directement (sans programmation) TOUTE assertion écrite en langage normal, l'expression des assertions sera GENERALEMENT réduite, pour des programmes compliqués, à une activité de programmation.

Donc, D. FISETTE a finalement défini un langage de PROGRAMMATION d'assertions (baptisé L2) et ce langage a permis comme on le verra, une traduction directe de la grande majorité des assertions de la classe des programmes citée plus haut.

Les limites inhérentes à ce langage proviennent naturellement de la nécessité de passer par une formalisation de ces assertions et aussi du fait que, d'après le théorème de Rice, il existe des assertions qui ne sont pas calculables ! ((2)).

Pour permettre une expression plus ou moins aisée des assertions, ce langage devrait en plus manipuler d'autres concepts beaucoup plus riches que ceux du langage de programmation dans lequel le programme à vérifier est écrit (ce langage est baptisé L1). Aussi y trouve-t-on des opérations et structures de données plus complexes comme les SUITES et ENSEMBLES et, toutes les manipulations possibles, aussi complexes soient-elles, de toutes sortes afférentes à ces structures.

Il a semblé que ces nouvelles structures de données devaient, avant de commencer à songer à l'implémentation de ce langage, retenir toute notre attention :

c'est pour cela que le présent mémoire est écrit; il étudie essentiellement certaines implémentations de ces structures de données, celles des suites et des ensembles d'entiers, de caractères et de booléens.

On admettra, après avoir étudié dans leurs détails algorithmiques les différentes assertions contenues dans le cours mentionné plus haut et traduit ces as-

sertions dans le langage L2, qu'il est certain que leur vérification pendant l'exécution pose d'énormes problèmes de coûts;
principalement, cette vérification prend, dans la majorité des cas beaucoup plus de temps que le programme lui-même.
Il était donc impératif, dans une perspective de contrôle de ces coûts, de rechercher des implémentations efficaces de ces suites et ensembles et des méthodes performantes de réalisation des différentes opérations qui ne gonflent pas considérablement le temps d'exécution et qui ne sont pas de grosses consommatrices de l'espace mémoire.

La première partie de ce mémoire est consacrée au rappel des éléments importants du langage L2.
On y trouvera dans la deuxième partie :

- la définition des structures de représentation finalement proposées,
- les règles de traduction des différentes primitives de L2 en pascal et
- la gestion dynamique de la mémoire.

Enfin, la troisième partie concernera essentiellement les mesures et les tests.

II. RAPPEL

D. FISETTE s'est donc livré dans son mémoire à une construction progressive, par des exemples de plus en plus compliqués, des concepts de ce langage devant permettre l'expression des assertions.
Ce travail de définition devrait se faire sur base d'un langage (habituel) de programmation déterminé (donc, L1). L'auteur de la définition de L2 a utilisé comme langage L1 une partie de LSD80 ((8)) qui, tout en ressemblant fort à pascal est moins riche que lui.
D. FISETTE est parti des notions élémentaires de tableau et de variable simple pour la construction de ce langage. Il a crée à partir de ces éléments de base des concepts qui, tout en étant suffisamment généraux (en vue d'extension à d'autres problèmes) permettent une traduction aisée de la classe de problèmes considérée.
Nous rappelons que la traduction des assertions en ce langage formel est d'une manière générale une activité de programmation.

* Dans sa démarche vers une définition complète du L2, on s'est aperçu que ces concepts devaient être très riches,
plus riches que ceux de la partie du LSD80 considérée.
On y trouvera donc des opérateurs et des structures de données, absents dans LSD80, comme l'exponentiation, la factorielle, les suites et ensembles, des opérations multiples et complexes sur ces structures,...

* En plus de types de variables admis par LSD80 (entier, caractère, booléen), on trouvera dans L2 six nouveaux types :
type suite d'entiers, type suite de caractères, type suite de booléens, type ensemble d'entiers, type ensemble de caractères et type ensemble de booleéns.
En outre, on augmentera le type entier de 2 valeurs supplémentaires :
zmin et zmax (respectivement moins infini et plus infini).

* Les instructions de L2 sont :
  - une partie des instructions de LSD80, syntaxiquement et sémantiquement ((8)),
  - l'instruction let qui syntaxiquement s'écrit :

let var be exp

où exp est une expression d'un type quelconque et var, une variable.

La sémantique est : déclaration de la variable var
(de même type que le type de l'expression exp) et
affectation de la valeur (si elle est déterminée) de
l'expression "exp" à la variable "var".

Comme les assertions font souvent référence à des valeurs (des variables) autres que les valeurs courantes, cette instruction servira à mémoriser ces valeurs,

- la fonction def qui s'écrit :

def (liste d'expression de désignation de variables ou de tableaux).

Elle est, comme le souligne D. FISETTE, la précondition UNIVERSELLE de toute assertion traduite en L2.

En langage normal, toutes les variables d'une assertion sont supposées avoir reçu une valeur;

en langage formel, cette supposition doit être explicitement vérifiée.

Dans l'instruction def (a),

si a est une variable simple, alors def(a) est vrai si "a" a été initialisé; faux sinon;

si a est un tableau alors def(a) est vrai si tous les éléments ont été initialisés; faux sinon,

- la pseudo-instruction variant et la fonction term dont les syntaxes sont : variant (id,"exp1",exp2) et
term (id) qui

sont utiles pour vérifier la terminaison des programmes (contenant au moins une boucle).

Variant spécifiera l'expression arithmétique exp1, appelé le variant, qui est la quantité qui décroîtra, à chaque tour de boucle, au moins de exp2, qui est une autre expression arithmétique (une constante fixée). Id sera le nom du variant correspondant à la boucle.

Term vérifiera, par l'intermédiaire de "id" que la valeur actuelle du variant (exp1) a bien décru de la valeur de exp2 au moins, par rapport à sa valeur précédente (de exp1).

* Dans le langage L2, la partie déclarative des variables n'est pas différente de celle de LSD80, elle est simplement augmentée des nouveaux types cités ci-avant.

* L'intrusion des structures de données plus complexes, à savoir les suites et les ensembles a, d'un côté, facilité pleinement certains types de raisonnements courants en démonstration des programmes mais elle a obligé, de l'autre côté, l'auteur de L2 à construire des expressions SIMPLES et convenant à leur manipulation.
Nous pourrons classer en gros, ces expressions de base en 2 catégories :
- les expressions de génération des suites et ensembles et,
- les expressions de manipulation de ces suites et ensembles.
Nous parlerons, dans ce rappel, plus spécifiquement des suites et ensembles d'entiers, la généralisation étant évidente pour les suites et ensembles d'autres types.

A. Expressions De Génération Des Suites Et Ensembles

De même qu'il faut qu'une variable existe et ait une valeur avant de l'utiliser dans une expression, il est aussi évident que pour utiliser les éléments d'une suite ou d'un ensemble, il faut les avoir crées auparavant.
On peut générer ces structures
soit à partir des valeurs des variables ou des tableaux du programme L1,
soit à partir d'autres suites et ensembles qui existent déjà.

a. Toute expression contenant les facteurs [exp] ou {exp} génère respectivement une suite ou un singleton composé du seul élément qui est issu de l'évaluation de l'expression "exp"
(exp est une expression arithmétique quelconque).

b. Toute expression contenant les facteurs [exp1..exp2] ou {exp1..exp2}

(soit v1 la valeur résultant de l'évaluation de l'expression arithmétique exp1,
soit v2 la valeur résultant de l'évaluation de l'expression arithmétique exp2)
générera respectivement une suite ou un ensemble dont les éléments sont ceux de l'intervalle v1..v2.

c. Toute expression contenant les facteurs [tab[exp1..exp2]]ou {tab[exp1..exp2]} générera,
avec v1 comme l'évaluation de l'expression arithmétique exp1 et v2 comme l'évaluation de l'expression arithmétique exp2,
d'abord la suite [v1..v2] et ensuite,
dans le premier cas, la suite composée des seuls éléments du tableau tab dont les indices sont spécifiés dans l'intervalle v1..v2 et,
dans le deuxième cas, l'ensemble composé des seuls éléments du tableau tab dont les indices sont spécifiés dans l'intervalle v1..v2.

d. Toute expression de la forme [tab] ou {tab} exprime respectivement la génération d'une suite ou d'un ensemble contenant tous les éléments du tableau tab.

e. Soit Sk, l'expression de désignation d'une suite, alors toute expresde la forme [tab[Sk]] ou {tab[Sk]} générera respectivement la suite ou l'ensemble composé des seuls éléments du tableau tab dont les indices sont les éléments de Sk.

Avec X1 et X2 étant n'importe quelle expression signalée aux points a, b, c, d et e (ci-dessus), c'est-à-dire n'importe quelle expression de désignation de suite ou d'ensemble,
- l'expression CONCAT (X1,X2) génère la suite issue de la concaténation X1|X2,
X1 et X2 doivent être des expressions de type suite;

- l'expression UNION (X1,X2) génère l'ensemble issu de l'union X1 $\cup$ X2,
  X1 et X2 doivent être des expressions de type ensemble;
- l'expression INTER (X1,X2) exprime la génération de l'ensemble issu de l'intersection X1 $\cap$ X2,
  X1 et X2 doivent être des expressions de type ensemble;
- l'expression DIFFER (X1,X2) générera l'ensemble issu de la différence X1 $\setminus$ X2,
  X1 et X2 étant des expressions de type ensemble;
- les expressions [id of X1|bexp] ou {id of X1|bexp} créeront respectivement la suite ou l'ensemble issu des éléments id de X1 pour lesquels bexp(id) est vrai,
  bexp étant une espression booléenne et X1, une expression de désignation d'une suite;
- la pseudo-instruction let pourra aussi être utilisée pour générer des suites ou des ensembles,
  ainsi let X0 be X1 générera :
  - la suite X0 si X1 est une expression de type suite et,
  - l'ensemble X0 si X1 est une expression de type ensemble.

En respectant les types, les expressions de base précitées peuvent être combinées pour produire des expressions plus complexes.

B. Expressions De Manipulation Des Suites Et Des Ensembles

Pour simplifier la lecture de ce paragraphe, supposons que les suites et les ensembles visés par les différentes manipulations existent déjà, il s'agira alors d'y faire référence au moyen de leurs variables de désignation :

Sk et Sj, pour les suites,

Ek et Ej, pour les ensembles.

a. Expressions de type entier

1. max (min) for id in Ek of aexp(x);
   avec id, un identificateur de même type que les éléments de Ek et,
   aexp(x), une expression arithmétique entière fonction de x,
   cette expression donne le maximum (minimum) de l'ensemble des valeurs de l'expression aexp(id) quand id puise successivement toutes ses valeurs dans Ek;

2. sum (prod) for id in Sk of aexp(x);
   avec id et aexp(x) ayant les mêmes significations que ci-dessus, cette expression fournit la somme (produit) de toutes les valeurs de l'expression aexp(id) quand id puise successivement toutes ses valeurs dans Sk;

3. first (Sk) renvoie le premier élément de Sk si Sk $\neq \emptyset$;

4. long (Sk) donne la longueur de Sk;

5. last (Sk) renvoie le dernier élément de Sk si Sk $\neq \emptyset$;

6. card (Ek) renvoie $|Ek|$;

7. elt (Ek) fournit un élément quelconque de Ek si Ek $\neq \emptyset$;

8. max (Ek) donne l'élément maximum de Ek si Ek $\neq \emptyset$;

9. min (Ek) donne l'élément minimum de Ek si Ek $\neq \emptyset$;

b. Expressions de type booléen

1. for all id in Ek : bexp(x);
   cette expression est vraie ssi bexp(id) est vrai
   quelle que soit la valeur de l'identificateur id
   (de même type que les éléments de Ek) puisée dans Ek, bexp(x)
   étant donc une expression booléenne fonction de x;

2. there exists id in Ek : bexp(x);
   cette expression est vraie ssi il existe une valeur de id
   (un identificateur de même type que les éléments de Ek),
   lorsque celui-ci puise toutes ses valeurs dans Ek, pour laquelle
   bexp(id) est vrai,
   bexp(x) étant une expression booléenne fonction de x;

3. empty (Sk), cette expression est vraie si long (Sk) = 0;

4. empty (Ek), cette expression est vraie si Ek = ∅;

5. égal (Sk,Sj), cette expression est vraie si Sk = Sj;

6. égal (Ek,Ej), cette expression est vraie si Ek = Ej;

7. ssuite (Sk,Sj), cette expression est vraie si Sk est une sous-suite de Sj;

8. segment (Sk,Sj), cette expression est vraie si Sk est un segment de Sj;

9. permut (Sk,Sj), cette expression est vraie si Sk est une permutation de Sj (et vice versa);

10. préfixe (Sk,Sj), cette expression est vraie si Sk est un préfixe de Sj;

11. suffixe (Sk,Sj), cette expression est vraie si Sk est un suffixe de Sj;

12. tricrois (Sk), cette expression est vraie si Sk est trié en ordre croissant;

13. tridec (Sk), cette expression est vraie si Sk est trié en ordre décroissant;

14. tristrcrois (Sk), cette expression est vraie si Sk est trié en ordre strictement croissant;

15. tristrdec (Sk), cette expression est vraie si Sk est trié en ordre strictement décroissant;

16. inclus (Ek,Ej), cette expression est vraie si $Ek \subseteq Ej$.

c. Expressions de type suite

1. head (Sk), cette expression renvoie la suite Sk sans son dernier élément si $Sk \neq \emptyset$;

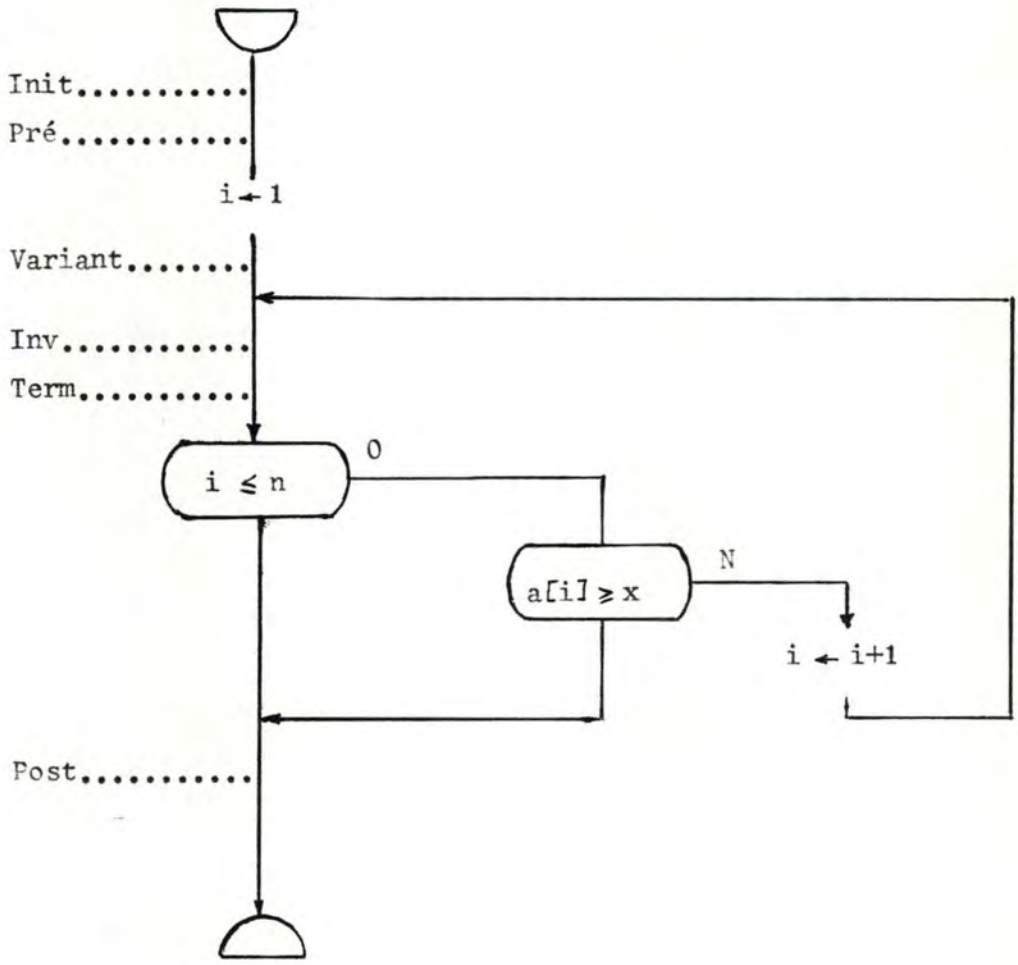
2. tail (Sk), cette expression renvoie la suite Sk sans son premier élément si $Sk \neq \emptyset$.

C. Exemples

Illustrons l'utilisation de ce langage dans 2 programmes concrets. On trouvera d'autres programmes plus compliqués dans le dernier chapître (TESTS) du présent mémoire.

1. SPECIFICATION : Soient a[1..n] un vecteur d'entiers trié par ordre croissant ($n \geq 0$) et x, une variable entière possédant une valeur.
   On doit affecter à la variable i
   - le plus petit entier i_o tel que $1 \leq i_o \leq n$ et $a[i_o] \geq x$, s'il existe un tel entier,
   - la valeur n+1 sinon.

ORGANIGRAMME :



LES ASSERTIONS TRADUITES EN L2 PEUVENT ETRE :

* INIT : def (x,a);

* PRE : let données be [x,a]; tricrois ([a]);

* VARIANT : variant (boucle, "n-i", 1);

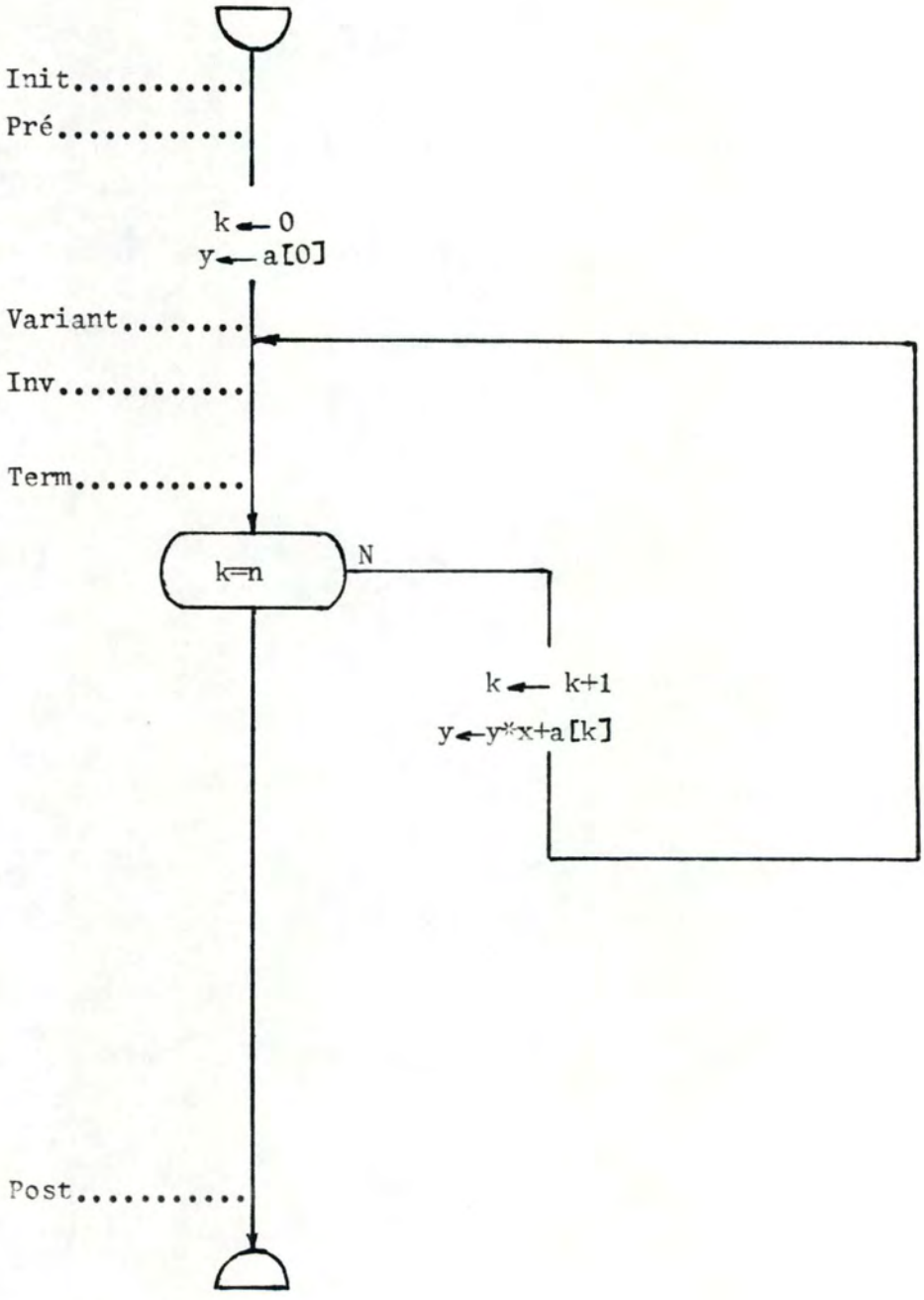
* INV : $1 \leq i \leq n$ and for all j in {1..i-1}: (a[j] < x) and
égal (données, [x,a]);

* TERM: term (boucle);

* POST : let ens be {j in {1..n} | a[j] ≥ x}; égal (données,[x,a]) and
(empty (ens) ⟹ i=n+1) and
(not (empty (ens)) ⟹ i=min (ens))

2. SPECIFICATION : Soit un vecteur d'entiers a[0..n] ($n \geqslant 0$) et soit une variable entière x, initialisée; il s'agit d'affecter à la variable y la valeur du polynôme de degré n (qui a pour coefficients les éléments de a) évalué en la valeur de x.

ORGANIGRAMME :



LES ASSERTIONS TRADUITES EN L2 PEUVENT ETRE :

* INIT : def (x,a);

* PRE : let a_o be [a]; let x_o be x;

* VARIANT : variant (boucle, "n-k", 1);

* INV : $0 \leq k \leq n$ and $x=x_o$ and égal (a_o, [a]) and
y=sum for i in [0..k] of (a[i]*x**k-i);

* TERM : term (boucle);

* POST : $x=x_o$ and égal (a_o, [a]) and
y=sum for i in [0..n] of (a[i]*x**n-i);

III. STRUCTURES DES DONNEES

1. GENERALITES

On voit directement que l'on pourrait profiter des avantages offerts par les structures de tableau et de pointeur :
en effet, à notre avis, en jouant sur les combinaisons de ces deux éléments, on devrait pouvoir trouver des structures des données plus ou moins adéquates pour les suites et ensembles.

Le pointeur sera nécessaire pour exprimer essentiellement la dimension dynamique des suites et des ensembles (la taille de ceux-ci pouvant constamment varier) et, le tableau pourra essentiellement servir à ranger les éléments des suites et à stocker ceux des ensembles.

Il est généralement difficile de construire, à partir de ces deux éléments une repésentation efficace qui soit la plus proche de la définition mathématique de l'ensemble,
c-à-d une représentation qui n'aurait aucune notion d'ordre sous-jacente et dont le temps d'accès à ses différents éléments, quasiment négligeable, serait le même; mais quand les éléments à enregistrer sont connus et leur nombre est petit (cfr représentation de l'ensemble de caractères), on pourrait approcher cette définition mathématique à l'aide de la table de présence.
Cette difficulté est une des raisons pour lesquelles nous déciderons que tout ensemble sera toujours représenté par une structure triée.

L'optique adoptée sera d'examiner deux types de structures :
- un premier, consommant la place mémoire nécessaire pour stocker ses éléments, et
- un deuxième pouvant, dans certains cas où les éléments à représenter se succèdent d'UNE CERTAINE MANIERE (ces cas sont très nombreux dans le cours de séminaire de programmation donné en 1ère licence) épargner de la place mémoire,

et d'étudier dans les 2 cas comment se comportent le temps d'exécution, l'occupation de la mémoire et la complexité de rédaction des algorithmes.

La technique de la table de présence (cfr représentation des ensembles en pascal) nous sera très utile pour représenter les ensembles de caractères et ceux de booléens.

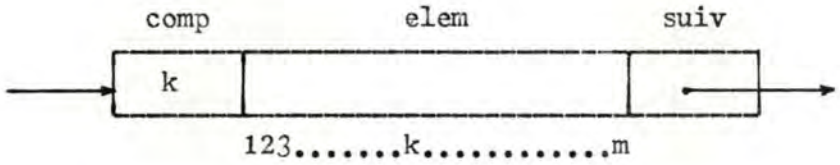
Nous croyons que toute organisation des éléments à représenter (en arbre, ...) autre qu'une organisation en "succession normale" de ces éléments serait inefficace pour notre propos et s'adaptera difficilement aux différentes opérations exécutables sur les suites et les ensembles.
Il nous serait finalement très difficile de définir avec les structures de données que nous fournit un langage comme le pascal, une autre représentation (plus efficace en temps et en consommation place mémoire) des suites et des ensembles qui ne tiendrait pas directement ou indirectement d'une des structures mentionnées plus haut ou des deux.

Pour les 2 types de repésentations qu'on envisagera, nous ne préciserons toutes les règles de traduction des primitives de L2 en pascal qu'en ce qui concerne les suites et les ensembles d'entiers; les autres règles de traduction dans les cas des suites de caractères et de booléens s'en déduisant aisément, il n'y a donc aucun intérêt à s'y attarder; quelques règles seulement seront données en ce qui concerne les ensembles de caractères et de booléens.

Pour exprimer ces règles, nous utiliserons un pseudo-langage, sans partie déclarative (pour la facilité d'expression), proche de pascal.

2. REPRESENTATION ETALEE

La représentation étalée sera une structure qui consommera autant de places mémoire qu'il y a d'éléments pour la suite ou l'ensemble à représenter, donc chacun de ces éléments sera PHYSIQUEMENT représenté.
Elle sera comme suit :



et appartiendra au type :

```
type ensuite = record
                 comp : integer;
                 elem : array [1..m] of X;
                 suiv : ^ensuite
                 end;
```

X peut être le type entier, caractère ou booléen.

Règles de représentation :

Les v (v > 0) différents éléments de la suite ou de l'ensemble à représenter seront stockés dans $\lceil v/m \rceil$ (°) structures de type ensuite, ni plus ni moins. Le tableau "elem" de chaque structure "ensuite" composant la représentation de la suite ou de l'ensemble comportera exactement m éléments sauf éventuellement celui de la dernière structure, auquel cas il aura ses $m*\lceil v/m \rceil - v$ derniers éléments vides.

- comp précisera le nombre d'éléments se trouvant dans le tableau "elem",
- m sera un paramètre qui définira la taille du tableau "elem", ce paramètre pourra être un élément intéressant dans le raffinement de cette représentation; en effet, selon les valeurs qu'il peut prendre, on observera différentes valeurs pour le temps d'exécution et pour l'espace mémoire consommé,
- suiv donnera accès au reste éventuel de la suite ou de l'ensemble.

(°) $i \leftarrow \lceil x \rceil$ est une expression telle que
i=x si x est entier,
i= (partie entière de x) + 1 sinon.

Nous donnons ci-après un ensemble de règles systématiques possibles pour traduire en pascal les différentes opérations et manipulations sur les suites et les ensembles.
Ce seront donc finalement des procédures et des fonctions que pourrait suivre un interpréteur de L2.

A. SUITES D'ENTIERS

Il suffit de loger tous les éléments devant constituer la suite dans des structures de type "ensuite" (cfr page 16).
Soit p, le pointeur du début de la représentation de la suite non vide ainsi créée, le z-ème élément de cette suite sera obtenue comme suit :

```
w ← z;
i ← ⌈z/m⌉;
w ← w - ((i-1) * m);
j ← 1;
tant que j<i faire | p ← p^.suiv;
                   | j ← j + 1
le z-ème élément de la suite est p^.elem[w].
```

Dans les pseudo-algorithmes qui suivent ci-dessous, les paramètres et les arguments p et q sont les pointeurs de début des suites désignées par Sk et Sj. Ces pseudo-algorithmes seront accompagnés si nécessaire d'une brève explication lorsque la traduction ne nous paraît pas évidente.
Notons toutefois qu'une suite d'entiers est vide ssi le pointeur correspondant est nil.

1. Expressions De Manipulation Des Suites d'Entiers

a. EMPTY (Sk) : vérifie si la suite Sk est ∅,

```
fonction EMPTY (p) : booléen;
    début
    si p ≠ nil alors empty←faux
                sinon empty←vrai
    fin
```

b. FIRST (Sk) : renvoie le 1er élément de la suite Sk.
Précondition : p ≠ nil,

```
fonction FIRST (p) : entier;
    début
    first←p^.elem[1]
    fin
```

c. LONG (Sk) : donne |Sk|,

```
fonction LONG (p) : entier;
    début
    l←0
    tant que p ≠ nil faire | l←l + p^.comp
                           | p←p^.suiv
    long←l
    fin
```

d. LAST (Sk) : renvoie le dernier élément de la suite Sk.
Précondition : p ≠ nil,

```
fonction LAST (p) : entier;
    début
    tant que p^.suiv ≠ nil faire   p←p^.suiv
    last←p^.elem  p^.comp
    fin
```

e. EGAL (Sk, Sj) : teste si les suites Sk et Sj sont égales,

```
fonction EGAL (p, q) : booléen;      (°)
    début
    eg ← vrai
    tant que p ≠ nil et q ≠ nil et eg
            faire | i ← 1
                  | tant que i ≤ p^.comp et eg
                  |         faire | si p^.elem[i] ≠ q^.elem[i]
                  |                |       alors eg ← faux
                  |                |       sinon i ← i+1
                  | si i > p^.comp alors p ← p^.suiv
                  | si i > q^.comp alors q ← q^.suiv
    si p ≠ nil ou q ≠ nil alors eg ← faux      (°°)
    si eg alors egal ← vrai
          sinon egal ← faux
    fin
```

f. PERMUT (Sk, Sj) : vérifie si Sk et Sj sont des permutations l'une de l'autre.

Si l'on doit directement travailler sur les représentations de ces deux suites Sk et Sj, respectivement des longueurs m et n, qui ne sont généralement pas triées,
alors on ne peut pas avoir une complexité théorique en temps de moins de $\mathcal{O}(mn)$.
On décidera alors, en vue de rendre cette fonction plus facile, de trier d'abord (par un tri rapide, le quicksort par exemple) les représentations de ces suites à l'aide d'une structure intermédiaire (tableau);
on pourrait ainsi n'avoir qu'une complexité théorique d'environ $\mathcal{O}(m \log m)$ (si $m \geq n$).

Soit donc SORT (p, table, i), une procédure
- qui transforme la représentation de la suite (d'entiers) dont le pointeur de début est p en un tableau "table" d'entiers (cfr fonction FORMTAB page 21),
- qui trie ce tableau et
- qui met la longueur de ce tableau dans la variable entière i,

alors la fonction PERMUT peut se présenter comme suit :

(°) : au point de vue esthétique et temps d'exécution, nous avons préféré cette version à la version récursive !

(°°) : ce test sert essentiellement à detecter les cas où la suite Sk serait un préfixe de la suite Sj (ou vice versa) mais n'auraient pas la même longueur.

```
fonction PERMUT (p, q) : booléen;
    début
    perm ← vrai
    SORT (p, tab1, i1)
    SORT (q, tab2, i2)
    i ← 1
    si i1 ≠ i2 alors perm ← faux
               sinon tant que  i ≤ i1 et perm
                             faire si tab1 [i] ≠  tab2 [i]
                                     alors  perm ← faux
                                     sinon  i ← i + 1
    si perm alors permut ← vrai
            sinon permut ← faux
    fin
```

g. SSUITE (Sk, Sj) : vérifie si Sk est une sous-suite de Sj

```
fonction SSUITE (p, q) : booléen;
    début
    i ← 1
    j ← 1
    tant que p ≠ nil et q ≠ nil
         faire | tant que  j ≤ q^.comp et p^.elem[i] ≠ q^.elem[j]
               |          faire  j ← j + 1
               | si j > q^.comp alors | q ← q^.suiv
               |                       | j ← 1
               |                sinon  | i ← i + 1
               |                       | si i > p^.comp
               |                       |    alors  i ← 1
               |                       |           p ← p^.suiv
               |                       | j ← j + 1
               |                       | si j > q^.comp
               |                       |    alors  j ← 1
               |                       |           q ← q^.suiv
    si p = nil alors ssuite ← vrai
               sinon ssuite ← faux
    fin
```

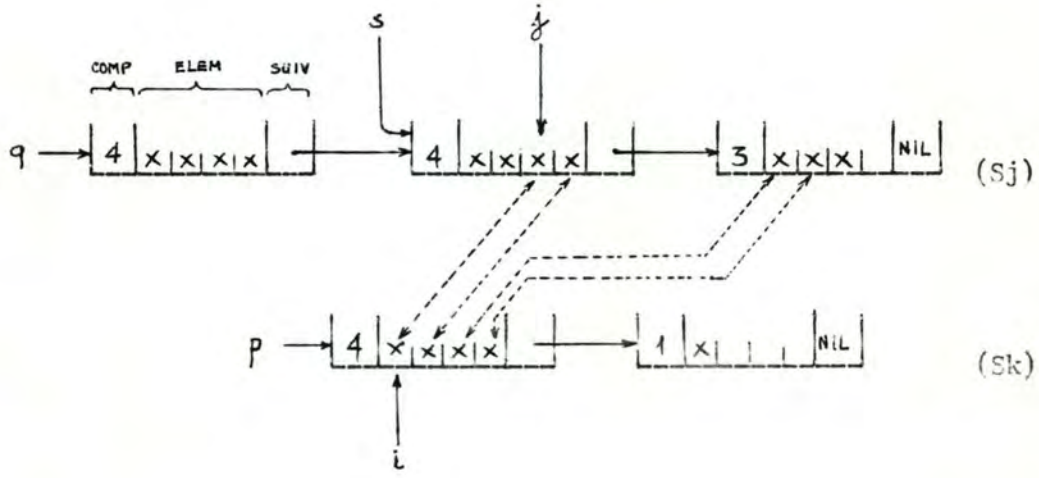
h. SEGMENT (Sk, Sj) : vérifie si Sk est un segement de Sj,

Définissons COMPAR (p, i, s, j) comme une fonction booléenne qui renverrait "vrai" si partant de l'élément s^.elem j de Sj, on trouve la même succession d'éléments que ceux de Sk dont le pointeur est, comme on l'a déjà dit, p.

Cette fonction pourrait être définie comme suit :

```
fonction COMPAR (p, i, s, j) : booléen;
début
eg←vrai
tant que p ≠ nil et s ≠ nil et eg
        faire | si p^.elem [i] ≠ s^.elem [j]
              |       alors  eg← faux
              |       sinon | i←i + 1
              |              | si i > p^.comp alors | p← p^.suiv
              |              |                      | i← 1
              |              | j←j + 1
              |              | si j > s^.comp alors | s← s^.suiv
              |              |                      | j←1
si p ≠ nil alors eg←faux
si eg alors compar←vrai
     sinon compar←faux
fin
```

Soient p, i, s et j;
si une telle sucession d'éléments existe à partir de l'élément s^.elem j dans Sj, on aura comparé ces éléments, indiqués schématiquement ci-dessous par des flèches à traits discontinus, deux à deux avec ceux de Sk :



La fonction SEGMENT pourrait alors être :

```
fonction SEGMENT (p, q) : booléen;
|début
|j ← 1
|seg ← faux
|d1 ← LONG (p)
|d2 ← LONG (q)
|si d2 ≥ d1
|   alors | si d2≠0 alors |l ← 1
|          |              |d ← faux
|          |              |tant que l ≤ d2-d1+1 ou d=faux      (°)
|          |              |      faire |d ← COMPAR (p, l, q, j)
|          |              |            |l ← l + 1
|          |              |            |j ← j + 1
|          |              |            |si j  q^.comp
|          |              |            |   alors |q ← q^.suiv
|          |              |            |         |j ← 1
|          |              |si d alors seg ← vrai
|          |          sinon  seg ← vrai
|si seg alors segment ← vrai
|       sinon segment ← faux
|fin
```

i. PREFIXE (Sk, Sj) : vérifie si Sk est un préfixe de Sj,

Soit la fonction COMPAR définie ci-avant (cfr page 18.(3)), la fonction PREFIXE pourrait simplement être :

```
fonction PREFIXE (p, q) : booléen;
|début
|préfixe ← COMPAR (p, 1, q, 1)
|fin
```

(°) : le test l ≤ d2-d1+1 permet de sortir de la boucle dès que le nombre d'éléments dans Sj (à partir de l'élément q^.elem [j] jusqu'à la fin de la représentation de la suite) devient inférieur au nombre d'éléments dans Sk.

j. SUFFIXE (Sk, Sj) : vérifie si Sk est un suffixe de Sj,

Un des moyens de réaliser cette opération est d'abord de passer, s'il y a lieu, les (LONG (q) - LONG (p)) éléments de Sj (cfr procédure PASSER ci-dessous) pour ne comparer ensuite que les (LONG (p)) éléments de Sk avec les (LONG (p)) derniers éléments de Sj.

```
procédure PASSER (q, j, d)        (°)
    début
    tant que d ⩾ 1 faire | j ← j + 1
                         | si j > q^.comp alors | q ← q^.suiv
                         |                     | j ← 1
                         | d ← d - 1
    fin
```

Soit la fonction COMPAR définie ci-avant (cfr page 18.(3)), alors la fonction SUFFIXE pourrait être :

```
fonction SUFFIXE (p, q) : booléen;
    début
    d ← LONG (q) - LONG (p)
    si d ⩾ 0 alors | i ← 1
                   | j ← 1
                   | si d > 0 alors PASSER (q, j, d)
                   | suffixe ← COMPAR (p, i, q, j)
             sinon   suffixe ← faux
    fin
```

k. TRICROIS (Sk) : vérifie si Sk est une suite triée en ordre croissant,

```
fonction TRICROIS (p) : booléen;
    début
    tric ← vrai
    maxval ← zmin
    tant que p ≠ nil et tric
          faire | i ← 1
                | tant que  i ⩽ p^.comp et tric
                |       faire | si maxval ⩽ p^.elem [i]
                |              |      alors | si maxval < p^.elem [i]
                |              |            |      alors maxval ← p^.elem [i]
                |              |            | i ← i + 1
                |              |      sinon tric ← faux
                | si i > p^.comp alors p ← p^.suiv
    si tric alors tricrois ← vrai
            sinon tricrois ← faux
    fin
```

(°) : d représente la différence LONG (q) - LONG (p).

l. TRISTRCROIS (Sk) : vérifie si Sk est une suite triée en ordre strictement croissant,

```
fonction TRISTRCTOIS (p) : booléen;
début
trisc ← vrai
maxval ← zmin
tant que p ≠ nil et trisc
     faire | i ← 1
           | tant que  i ≤ p^.comp et trisc
           |      faire | si maxval < p^.elem [i]
           |           |      alors | maxval ← p^.elem [i]
           |           |            | i ← i + 1
           |           |      sinon trisc ← faux
           | si i > p^.comp alors p ← p^.suiv
si trisc alors tristrcrois ← vrai
         sinon tristrcrois ← faux
fin
```

m. TRIDEC (Sk) : vérifie si Sk est une suite triée en ordre décroissant,

```
fonction TRIDEC (p) : booléen;
début
trid ← vrai
minval ← zmax
tant que p ≠ nil et trid
     faire | i ← 1
           | tant que  i ≤ p^.comp et trid
           |      faire | si minval ≥ p^.elem [i]
           |           |      alors | si minval > p^.elem [i]
           |           |            |      alors  minval ← p^.elem [i]
           |           |            | i ← i + 1
           |           |      sinon trid ← faux
           | si i > p^.comp alors p ← p^.suiv
si trid alors tridec ← vrai
        sinon tridec ← faux
fin
```

n. TRISTRDEC (Sk) : vérifie si Sk est une suite triée en ordre strictement décroissant,

```
fonction TRISTRDEC (p) : booléen;
début
trisd ← vrai
minval ← zmax
tant que p ≠ nil et trisd
      faire | i ← 1
            | tant que i ≤ p^.comp et trisd
                  faire | si minval > p^.elem [i]
                        |       alors | minval ← p^.elem [i]
                        |              | i ← i + 1
                        |       sinon trisd ← faux
            | si  i > p^.comp alors  p ← p^.suiv
si trisd alors tristrdec ← vrai
         sinon tristrdec ← faux
fin
```

o. SUM FOR <id> IN Sk OF <expr-arith> : calcul de la somme $\sum$ expr-arith (id) pour tout id $\in$ Sk, expr-arith étant une fonction de x : expr-arith (x),

```
fonction SUMFOR (p, id, expr-arith) : entier;
début
s ← 0
tant que p ≠ nil faire | i ← 1
                       | tant que  i ≤ p^.comp
                       |       faire | id ← p^.elem [i]
                       |              | s ← s + expr-arith (id)
                       |              | i ← i + 1
                       | p ← p^.suiv
sumfor ← s
fin
```

p. PROD FOR <id> IN Sk OF <expr-arith> : calcul du produit $\prod$ expr-arith (id) pour tout id $\in$ Sk, expr-arith étant une fonction de x : expr-arith (x),

```
fonction PRODFOR (p, id, expr-arith) : entier;
début
v ← 1
tant que p ≠ nil faire | i ← 1
                       | tant que  i ≤ p^.comp
                       |       faire | id ← p^.elem [i]
                       |              | v ← v * expr-arith (id)
                       |              | i ← i + 1
                       | p ← p^.suiv
prodfor ← v
fin
```

2. Expressions Génératrices Des Suites d'Entiers

Remarque : (i) L'expression [Sk,Sj] ou CONCAT (Sk, Sj) pourrait être vue sous différents aspects quant à la manière de construire la suite résultante.

Soit E, l'expression dans laquelle se trouve l'expression CONCAT (Sk, Sj) :

* on pourrait l'interpréter comme
  Sk ← CONCAT (Sk, Sj) (1)
  ou
  Sj ← CONCAT (Sk, Sj);
  dans (1) par exemple, on rattacherait Sj à Sk et on renverrait le pointeur p de la suite Sk.

  Avantage : Cela est accompli par une opération simple et rapide.

  Inconvénient : Nécessite une opération de détachement des suites et une utilisation des compteurs de références ((3)), ((4)) si l'on doit accéder aux seuls éléments de Sk ou de Sj après l'évaluation de E,

* on pourrait l'interpréter comme une opération générant une 3ème suite (Sr) différente de Sk et de Sj :
  Sr ← CONCAT (Sk, Sj),
  on renverrait donc le pointeur r de cette troisième suite.

  Avantage : Clarté et facilité de programmation; les 2 suites Sk et Sj restant inchangées.

  Inconvénient : Consommation de la place mémoire.

Grâce à sa clarté, nous choisirons cette dernière interprétation.

(ii) Les autres expressions [exp], [exp1..exp2], [tab], [tab [Sk]], ... ne posent aucun problème spécial.

Pour toutes les expressions de génération, nous utiliserons la fonction FORMSUIT (expr) ci-dessous dont la spécification est :
de produire la représentation étalée de la suite demandée en fonction de l'expression de génération "expr" et de renvoyer le pointeur r de début de la suite ainsi construite (cfr page 20.(6)).

Pour ne pas encombrer inutilement cette fonction, nous supposerons que toutes les variables ont été correctement initialisées et laisserons de côté tous les contrôles qui ne nous intéressent pas directement.
Soient r, suite et lien trois variables de type ensuite (cfr page 16).

```
fonction FORMSUIT (expr) : ^ensuite;
DEBUT
case expr of
[exp] : début
        v ← évaluation (exp)
        new (suite)
        r ← suite
        r^.elem [1] ← v
        r^.comp ← 1
        r^. suiv ← nil
        formsuit ← r
        fin

[exp1..exp2] : début
               v ← évaluation (exp1)
               w ← évaluation (exp2)
               r ← nil
               si v ≤ w alors new (suite)
                              r ← suite
                              lien ← suite
                              répéter j ← 1
                                      tant que j ≤ m et v ≤ w faire suite^.elem [j] ← v
                                                                    j ← j + 1
                                                                    v ← v + 1
                                      suite ^.comp ← j - 1
                                      si v ≤ w alors new (suite)
                                                     lien^.suiv ← suite
                                                     lien ← suite
                                                sinon suite^.suiv ← nil
                              jusqu'à ce que v > w
               formsuit ← r
               fin
```

```
[tab] : début
        r ← nil
              {g = borne inférieure du tableau tab}
              {d = borne supérieure du tableau tab}
        si g ≤ d alors | new (suite)
                       | r ← suite
                       | lien ← suite
                       | répéter | j ← 1
                       |         | tant que j ≤ m et g ≤ d faire | suite^.elem [j] ← tab [g]
                       |         |                               | j ← j + 1
                       |         |                               | g ← g + 1
                       |         | suite^.comp ← j - 1
                       |         | si g ≤ d alors | new (suite)
                       |         |                | lien^.suiv ← suite
                       |         |                | lien ← suite
                       |         |          sinon | suite^.suiv ← nil
                       | jusqu'à ce que g > d
        formsuite ← r
        fin
```

```
[tab [Sk]] : début
             i ← 1
             j ← 1
             r ← nil
             louer ← vrai
             firstime ← vrai
             tant que p ≠ nil faire | tant que j ≤ m et i ≤ p^.comp faire | si louer alors | new (suite)
                                                                                           | si firstime
                                                                                           |    alors | r ← suite
                                                                                           |          | firstime ← faux
                                                                                           |    sinon | lien^.suiv ← suite
                                                                                           | lien ← suite
                                                                                           | louer ← faux
                                                                         | suite^.elem [j] ← tab [p^.elem [i]]
                                                                         | j ← j + 1
                                                                         | i ← i + 1
                                                    | si j > m alors | suite^.comp ← j - 1
                                                                     | louer ← vrai
                                                                     | j ← 1
                                                    | si i > p^.comp alors | p ← p^.suiv
                                                                           | i ← 1
             si r ≠ nil alors | si j ≠ 1 alors suite^.comp ← j - 1
                              | suite^.suiv ← nil
             formsuit ← r
             fin
```

{p est le pointeur de Sk}

```
[id of Sk | bexp] : | début
                    | r ← nil
                    | i ← 1
                    | j ← 1
                    | firstime ← vrai
                    | louer ← vrai
                    | tant que  p ≠ nil
                    |           faire | tant que  j ≤ m et i ≤ p^.comp
                    |                  |           faire | id ← p^.elem [i]
                    |                  |                 | si bexp (id) alors | si louer alors | new (suite)
                    |                  |                 |                    |                | si firstime
                    |                  |                 |                    |                |    alors | r ← suite
                    |                  |                 |                    |                |          | firstime ← faux
                    |                  |                 |                    |                |    sinon | lien^.suiv ← suite
                    |                  |                 |                    |                | lien ← suite
                    |                  |                 |                    |                | louer ← faux
                    |                  |                 |                    | suite^.elem [j] ← p^.elem [i]
                    |                  |                 |                    | j ← j + 1
                    |                  |                 | i ← i + 1
                    |                  | si  j > m alors | suite^.comp ← j - 1
                    |                  |                 | louer ← vrai
                    |                  |                 | j ← 1
                    |                  | si  i > p^.comp alors | p ← p^.suiv
                    |                  |                       | i ← 1
                    | si  r ≠ nil alors | si j ≠ 1 alors  suite^.comp ← j - 1
                    |                   | suite^.suiv ← nil
                    | formsuit ← r
                    | fin
                                                                  {p est le pointeur de la suite Sk}
```

---

bexp (id) est l'évaluation de l'expression booléenne bexp (x) dépendant de x, avec x = id

```
CONCAT (Sk, Sj) : début
                  i ← 1
                  j ← 1                                  {p est le pointeur de la suite Sk}
                  r ← nil                                {q est le pointeur de la suite Sj}
                  cpt ← 1
                  louer ← vrai
                  firstime ← vrai
                  s ← p
                  répéter | tant que s ≠ nil
                          |       faire | tant que j ≤ m et i ≤ s^.comp
                          |              |       faire | si louer alors | new (suite)
                          |              |             |                | si firstime alors | r ← suite
                          |              |             |                |                   | firstime ← faux
                          |              |             |                | sinon lien^.suiv ← suite
                          |              |             |                | lien ← suite
                          |              |             |                | louer ← faux
                          |              |             | suite^.elem [j] ← s^.elem [i]
                          |              |             | i ← i + 1
                          |              |             | j ← j + 1
                          |              | si j > m alors | suite^.comp ← j - 1
                          |              |                | louer ← vrai
                          |              |                | j ← 1
                          |              | si i > s^.comp alors | s ← s^.suiv
                          |              |                      | i ← 1
                          | si r ≠ nil alors si cpt = 2 alors | si j ≠ 1 alors suite^.comp ← j - 1
                          |                                   | suite^.suiv ← nil
                          | cpt ← cpt + 1
                          | s ← q
                  jusqu'à ce que cpt > 2
                  formsuit ← r
                  fin

FIN
```

3. Autres Expressions

TAIL (Sk) et HEAD (Sk) :

Soit FORMTAB (p, table), une fonction dont la spécification est :
- de transformer la représentation de la suite pointée par p en un tableau "table" et,
- de renvoyer la longueur de ce tableau :

```
fonction FORMTAB (p, table) : entier;
    début
    i ← 0
    tant que p ≠ nil faire |for j:=1 to p^.comp
                           |    do | i ← i + 1
                           |       | table [i] ← p^.elem [j]
                           |       | j ← j + 1
                           |p ← p^.suiv
    formtab ← i
    fin,
```

soit la fonction FORMSUIT comme spécifiée page 19,

TAIL (Sk) qui renvoie l'adresse r de la queue de Sk (Sk sans son premier élément) pourra s'écrire comme suit :

```
fonction TAIL (p) : ^ensuite;
    début
    k ← FORMTAB (p, table)
    tail ← FORMSUIT ( [table [2..k]])
    fin
```

HEAD (Sk) qui renvoie l'adresse r de la tête de Sk (Sk sans son dernier élément) pourra se traduire comme suit :

```
fonction HEAD (p) : ^ensuite;
    début
    k ← FORMTAB (p, table)
    head ← FORMSUIT ( [table [1..k - 1]])
    fin
```

B. SUITES DE CARACTERES

Etant donné C, l'ensemble des caractères défini dans la définition du langage LSD80 ((8)).
Se limitant strictement aux opérations permises sur les caractères, on pourrait, en utilisant les primitives ORD et CHR induire toutes les modifications adéquates sur les algorithmes déjà développés pour les suites d'entiers,
alors, il n'est pas du tout intéressant de mettre dans ce mémoire les pseudo-algorithmes définis pour les suites de caractères.

C. SUITES DE BOOLEENS

Etant donné l'ensemble B = {true, false}, la logique des algorithmes déjà développés pour les suites d'entiers reste valable pour les suites de booléens et avec l'aide des fonctions ORD et CHR, il n'y a aucune difficulté à y apporter les aménagements adéquats. Dès lors, il serait inutile d'inclure dans ce mémoire tous les pseudo-algorithmes développés pour ces suites de booléens.

D. ENSEMBLES D'ENTIERS

On aura pratiquement les mêmes pseudo-algorithmes que pour les suites d'entiers mais on pourrattt en plus apporter une certaine amélioration dans les expressions de manipulation, surtout au point de vue de la complexité temporelle, en profitant du fait que les représentations des ensembles sont des structures toujours triées.
Dans la traduction de chaque expression de L2 capable de générer un ensemble d'entiers, on y trouvera, s'il y a lieu, un processus qui trierait les éléments de la représentation de l'ensemble en question.
Soient p et q, les pointeurs sur les représentations des ensembles désignées respectivement par Ek et Ej. Notons qu'un ensemble d'entiers est vide si et seulement si le pointeur correspondant est nil.

1. Expressions De Manipulation Des Ensembles d'Entiers

a. EGAL (Ek, Ej) : teste si Ek et Ej sont égaux,
cfr fonction EGAL (Sk, Sj) page 18.(1)

b. CARD (Ek) : renvoie |Ek|,
cfr fonction LONG (Sk) page 18.(0)

c. EMPTY (Ek) : teste si Ek est $\emptyset$,
cfr fonction EMPTY (Sk) page 18.(0)

d. ELT (Ek) : renvoie un élément de Ek.
Convenons-nous de renvoyer, par exemple, toujours le premier élément de la représentation de l'ensemble.
Précondition : p $\neq$ nil,

```
fonction ELT (p) : booléen;
    début
    elt ← p^.elem [1]
    fin
```

e. INCLUS (Ek, Ej) : teste si Ek ⊆ Ej

fonction INCLUS (p, q) : booléen; (°)

```
début
i ← 1
j ← 1
tant que p ≠ nil et q ≠ nil
        faire | tant que i ≤ p^.comp et j ≤ q^.comp
              |       faire | si  p^.elem [i] = q^.elem [j]
              |              |       alors | i ← i + 1
              |              |             | j ← j + 1
              |              |       sinon   j ← j + 1
              | si i > p^.comp alors | p ← p^.suiv
              |                      | i ← 1
              | si j > q^.comp alors | q ← q^.suiv
              |                      | j ← 1
si p = nil alors inclus ← vrai
           sinon inclus ← faux
fin
```

f. MAX (Ek) : renvoie i ∈ Ek tel que i > j pour tout j ∈ Ek, i ≠ j,

fonction MAX (p) : entier;

```
début
si p = nil alors  max ← zmin
           sinon | tant que p^.suiv ≠ nil faire  p ← p^.suiv
                 | max ← p^.elem [p^.comp]
fin
```

g. MIN (Ek) : renvoie i ∈ Ek tel que i < j pour tout j ∈ Ek, i ≠ j,

fonction MIN (p) : entier;

```
début
si p = nil alors min ← zmax
           sinon min ← p^.elem [1]
fin
```

(°) : cette version est préférable à la version récursive car elle est meilleure en temps d'exécution !

h. MAX FOR <id> IN Ek OF <expr-arith> : calcul de
max {expr-arith (id)}
pour tout id ∈ Ek,

```
fonction MAXFOR (p, id, expr-arith) : entier;
|début
|maxi ← zmin
|tant que p ≠ nil faire |i ← 1
|                       |tant que i ≤ p^.comp
|                       |      faire |id ← p^.elem [i]
|                       |            |k ← expr-arith (id)
|                       |            |si maxi < k alors maxi ← k
|                       |            |i ← i + 1
|                       |p ← p^.suiv
|maxfor ← maxi
|fin
```

i. MIN FOR <id> IN Ek OF <expr-arith> : calcul de
min {expr-arith (id)}
pour tout id ∈ Ek,

```
fonction MINFOR (p, id, expr-arith) : entier;
|début
|mini ← zmax
|tant que p ≠ nil faire |i ← 1
|                       |tant que i ≤ p^.comp
|                       |      faire |id ← p^.elem [i]
|                       |            |k ← expr-arith (id)
|                       |            |si mini > k alors mini ← k
|                       |            |i ← i + 1
|                       |p ← p^.suiv
|minfor ← mini
|fin
```

j. FOR ALL <id> IN Ek :<(bexp)> renvoie "vrai" si
bexp (id) = vrai pour tout id ∈ Ek,

```
fonction FORALL (p, id, bexp) : booléen;
    début
    frl ← vrai
    tant que  p ≠ nil et frl
              faire | i ← 1
                    | tant que i ≤ p^.comp et frl
                    |           faire | id ← p^.elem [i]
                    |                  | v ← bexp (id)
                    |                  | si v=faux alors  frl ← faux
                    |                  |          sinon  i ← i + 1
                    | p ← p^.suiv
    si frl alors forall ← vrai
           sinon forall ← faux
    fin
```

k. THERE EXIXTS <id> IN Ek : <(bexp)> renvoie "vrai" si
bexp (id) = vrai pour au moins
un id ∈ Ek,

```
fonction THEREXISTS (p, id, bexp) : booléen;
    début
    thex ← faux
    tant que p  ≠ nil et thex = faux
              faire | i ← 1
                    | tant que i ≤ p^.comp et thex = faux
                    |           faire | id ← p^.elem [i]
                    |                  | v ← bexp (id)
                    |                  | si v=vrai alors  thex ← vrai
                    |                  |          sinon  i ← i + 1
                    | p ← p^.suiv
    si thex alors therexists ← vrai
           sinon therexists ← faux
    fin
```

2. Expressions Génératrices Des Ensembles D'Entiers

Définissons TRIER (table, n), une fonction qui trie (rapidement) le tableau "table" de taille n et qui renvoie le pointeur r de la représentation étalée de l'ensemble (trié) issu de ce tableau.

Les expressions {exp} , {exp1..exp2} et {id of Ek | bexp} auront rigoureusement les mêmes pseudo-algorithmes que ceux des expressions de génération des suites [exp] , [exp1..exp2] et [id of Sk | bexp] respectivement; elles fourniront toujours en effet, des représentations (d'ensembles) triées.

Dans les autres cas, pour des raisons d'efficacité, on ne gardera pas les mêmes pseudo-algorithmes que ceux définis pour les suites. Il est plus rapide de produire d'abord un tableau dans lequel les opérations de tri et d'élimination des copies pourront être directement accomplies.

Donc, on ne peut pas brutalement annexer ces deux opérations à la fin des pseudo-algorithmes développés pour les suites d'entiers pour produire ceux des ensembles, auquel cas, on devra passer par une opération supplémentaire et inutile de formation de tableau (cfr fonction FORMTAB page 21) avant d'appliquer les deux opérations (de tri et d'élimination des copies).

Plus précisement, dans la fonction TRIER (table, n), on trouvera (cfr page 27) :

- le tri proprement dit qui sera accompli par la procédure QSORT (table, li, ls) dont la spécification sera :
  trier les éléments table[li], table[li+1], ..., table[ls] du tableau "table" (li ≤ ls) par un tri rapide en ordre croissant;
- l'élimination des copies qui sera accomplie par la fonction ELIMINER (table, n) dont la spécification est :
  * d'éliminer toute copie dans le tableau "table" de taille n, trié et
  * de renvoyer une valeur i, entière de sorte que les éléments de ce tableau qui étaient tels que :
    table [1] ≤ table[2] ≤ ..... ≤ table [n]

deviennent tels que :
table[1]' < table[2]' < ..... < table[i]' (avec i ≤ n) et
{table[1], table[2], ... , table[n]} =
{table[1]', table[2]', ... , table[i]'};

- la <u>génération</u> de l'ensemble demandé qui sera faite par la fonction GENERER (table, g, d).
  Celle-ci fournira le pointeur r de la représentation étalée de l'ensemble des éléments du tableau "table" dont les indices sont compris dans l'intervalle [g..d]

Soient r, s, ens, lien des variables de type <u>ensuite</u> (cfr page 16), pour toutes les expressions de génération, nous utiliserons la fonction <u>FORMENS</u> (expr) dont la spécification est :

produire la représentation étalée de l'ensemble demandé en fonction des différentes expressions "expr" de génération et de renvoyer le pointeur r de début de l'ensemble ainsi généré (cfr page 28.(0)).

Pour des raisons de clarté, nous supposerons que toutes les variables ont été correctement initialisées et laisserons de côté tous les contrôles qui ne nous intéressent pas directement.

```
fonction ELIMINER (table, n) : entier;
    début
    i ← 2
    j ← 1
    tant que i ≤ n faire | si table [j] ≠ table [i]
                         |        alors | j ← j + 1
                         |               | table [j] ← table [i]
                         | i ← i + 1
    si n > 0 alors  eliminer ← j
             sinon  eliminer ← 0
    fin
```

```
fonction GENERER (table, g, d) : ^ensuite;
    début
    si  g ≤ d alors | louer ← faux
                    | new (ens)
                    | r ← ens
                    | lien ← ens
                    | répéter | j ← 1
                    |         | tant que j ≤ m et g ≤ d
                    |         |       faire | si louer
                    |         |             |    alors | new (ens)
                    |         |             |          | lien^.suiv ← ens
                    |         |             |          | lien ← ens
                    |         |             |          | louer ← faux
                    |         |             | ens^.elem [j] ← tab [g]
                    |         |             | j ← j + 1
                    |         |             | g ← g + 1
                    |         | ens^.comp ← j - 1
                    |         | si g ≤ d alors louer ← vrai
                    |         |           sinon ens^.suiv ← nil
                    | jusqu'à ce que  g > d
              sinon | r ← nil
    générer ← r
    fin
```

```
fonction TRIER (table, n) : ^ensuite;
    début
    QSORT (table, 1, n)
    trier ← GENERER (table, 1, ELIMINER (table, n))
    fin
```

```
fonction FORMENS (expr) : ^ensuite
    DEBUT
    case expr of
    {tab [ Sk]} : début
                  j ← 0
                  tant que  p ≠ nil faire | i ← 1
                                          | tant que i ≤ p^.comp faire | j ← j + 1
                                          |                            | table [j] ← tab [p^.elem [i]]
                                          |                            | i ← i + 1
                                          | p ← p^.suiv
                  formens ← TRIER (table, j)
                  fin

    {tab} :  début
             formens ← TRIER (tab, n)                    (°)
             fin

    {Sk} :   début
             j ← 0
             tant que  p ≠ nil faire | i ← 1
                                     | tant que  i ≤ p^.comp faire | j ← j + 1
                                     |                             | table [j] ← p^.elem [i]
                                     |                             | i ← i + 1
                                     | p ← p^.suiv
             formens ← TRIER (table, j)
             fin
```

(°) : n est la taille du tableau "tab"

{exp} : cfr [exp] page 20.(0) où l'instruction "formsuit ← r" sera remplacée par l'instruction "formens ← r"

{exp1..exp2} : cfr [exp1..exp2] page 20.(0) où l'instruction "formsuit ← r" sera remplacée par l'instruction "formens ← r"

```
INTER (Ek, Ej) : début
                 k ← 0
                 i ← 1
                 j ← 1
                 tant que p ≠ nil et q ≠ nil faire | tant que i ≤ p^.comp et j ≤ q^.comp
                                                   |       faire | si p^.elem [i] = q^.elem [j]
                                                   |              |      alors | k ← k + 1
                                                   |              |            | table [k] ← p^.elem [i]
                                                   |              |            | i ← i + 1
                                                   |              |            | j ← j + 1
                                                   |              |      sinon si p^.elem [i] < q^.elem [j]
                                                   |              |                  alors i ← i + 1
                                                   |              |                  sinon j ← j + 1
                                                   | si i > p^.comp alors | i ← 1
                                                   |                      | p ← p^.suiv
                                                   | si j > q^.comp alors | j ← 1
                                                   |                      | q ← q^.suiv
                 formens ← GENERER (table, 1, k)        (°)
                 fin
```

(°) : il est évident que dans ce cas, le tableau "table" n'a pas besoin d'être trié !

```
UNION (Ek, Ej) : début
                 j ← 0
                 cpt ← 1
                 s ← p
                 répéter | tant que s ≠ nil faire | i ← 1
                         |                        | tant que i ≤ s^.comp faire | j ← j + 1
                         |                        |                            | table [j] ← s^.elem [i]      (°)
                         |                        |                            | i ← i + 1
                         |                        | s ← s^.suiv
                         | cpt ← cpt + 1
                         | s ← q
                 jusqu'à ce que cpt > 2
                 formens ← TRIER (table, j)
                 fin

DIFFER (Ek, Ej) : début
                  k ← 0
                  i ← 1
                  j ← 1
                  tant que p ≠ nil et q ≠ nil faire | tant que i ≤ p^.comp et j ≤ q^.comp
                                                    |       faire | si p^.elem [i] < q^.elem [j]
                                                    |              |      alors | k ← k + 1
                                                    |              |            | table [k] ← p^.elem [i]
                                                    |              |            | i ← i + 1
                                                    |              |      sinon | si p^.elem [i] > q^.elem [j]
                                                    |              |            |      alors  j ← j + 1
                                                    |              |            |      sinon | i ← i + 1
                                                    |              |            |            | j ← j + 1
                                                    | si i > p^.comp alors | i ← 1
                                                    |                      | p ← p^.suiv
                                                    | si j > q^.comp alors | j ← 1
                                                    |                      | q ← q^.suiv
                  tant que p ≠ nil faire | tant que i ≤ p^.comp faire | k ← k+1
                                         |                            | table [k] ← p^.elem [i]
                                         |                            | i ← i + 1
                                         | si i > p^.comp alors | i ← 1
                                         |                      | p ← p^.suiv
                  formens ← GENERER (table, 1, k)      (°°)
                  fin

FIN
```

(°) : cette façon de faire est, pour la primitive UNION, beaucoup moins longue que le parcours parallèle des représentations, comme cela est fait pour INTER.

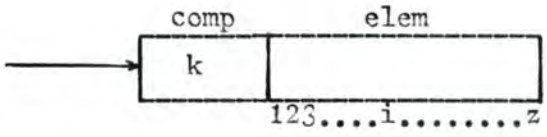
(°°) : le tableau "table" n'a pas besoin d'être trié, il est déjà en ordre strictement croissant !

E. ENSEMBLES DE CARACTERES

L'ensemble de type ensemble de caractères sera représenté par une seule structure de type ensc (ci-dessous) où le tableau "elem" est un tableau de booléens et sa taille sera z, le cardinal de l'ensemble C (cfr page 22).
Cette idée nous est suggérée par la façon dont le type SET est implémenté en pascal ((4)).

On aura donc :

```
type ensc = record
              comp : entier;
              elem : array [1..z] of booléen
            end
```



avec k : $1 \leq k \leq z$;

soit p, le pointeur sur la représentation d'un ensemble de caractères, sa signification sera :

si $1 \leq i \leq z$ alors le caractère (représenté par i) $\in$ représentation de l'ensemble si p^.elem [i] est vrai ou le caractère (représenté par i) $\notin$ représentation de l'ensemble si p^.elem [i] est faux;

k étant le cardinal de l'ensemble $\{$p^.elem [j] = vrai pour tout j : $1 \leq j \leq z\}$.

Ce faisant, on évite essentiellement de passer par la phase de tri à chaque opération susceptible de modifier le nombre d'éléments dans l'ensemble (nous rappelons que les ensembles devaient toujours être des représentations triées).

La réalisation des différentes primitives en pseudo-algorithmes selon cette représentation ne pose aucune espèce de problème; plusieurs primitives seront manifestement plus efficaces dans leur implémentation car on n'aura à manipuler qu'une seule structure.

Dans le souci de rendre les algorithmes définis sur cette implémentation beaucoup plus simples, nous conviendrons de représenter l'ensemble de caractères vide par une structure où les z éléments sont tous "faux".
Ainsi, soient r et ens deux variables de type ensc, pour générer par exemple les ensembles INTER (Ek, Ej) et DIFFER (Ek, Ej), nous pourrions respectivement avoir :

INTER (Ek, Ej):

```
début
i ← 1
k ← 0
new (ens)
tant que i ≤ z faire | si p^.elem [i] et q^.elem [i]
                     |      alors | ens^.elem [i] ← vrai
                     |             | k ← k + 1
                     |      sinon   ens^.elem [i] ← faux
                     | i ← i + 1
ens^.comp ← k
fin
```

DIFFER (Ek, Ej):

```
début
i ← 1
k ← 0
new (ens)
tant que i ≤ z faire | si p^.elem [i] et not (q^.elem [i] )
                     |      alors | ens^.elem [i] ← vrai
                     |             | k ← k + 1
                     |      sinon   ens^.elem [i] ← faux
                     | i ← i + 1
ens^.comp ← k
fin
```

F. ENSEMBLES DE BOOLEENS

Ces ensembles n'ont aucune utilité pratique.
Théoriquement, on peut tenir le même raisonnement que celui utilisé pour les ensembles de caractères en définissant le type :

```
type ensb = record
              comp : entier;
              elem : array [1..2] of booléen
            end.
```

On aurait pu très bien, en vue de rendre certains pseudo-algorithmes sur les suites et ensembles d'entiers plus clairs, définir une primitive d'accès à l'élément suivant comme par exemple :

```
procédure NEXT (p, i);
    début
    i ← i + 1
    si i > p^.comp alors  i ← 1
                          p ← p^.suiv
    fin
```

avec comme précondition : $p \neq nil$
(p étant une variable de type ensuite (cfr page 16) et i, une variable entière qui est l'indice du tableau "elem").

3. REPRESENTATION COMPACTEE

Elle sera une représentation qui, dans certains cas économisera de l'espace mémoire.

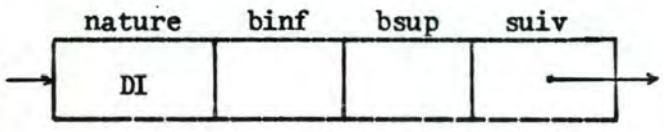
Elle pourra être considérée comme une amélioration de la représentation étalée, au point de vue de la consommation de la mémoire, par le fait qu'elle prendra en compte CERTAINES RELATIONS, à savoir essentiellement :

* progression ou régression par incrément de +1 ou de -1
  (sur les éléments dans le cas de suites et ensembles d'entiers ou sur le rang des éléments dans le cas de suites et ensembles de caractères ou de booléens) et
* répétition d'éléments

pouvant exister entre deux ou plusieurs éléments successifs parmi les différents éléments devant constituer la suite ou l'ensemble à représenter.

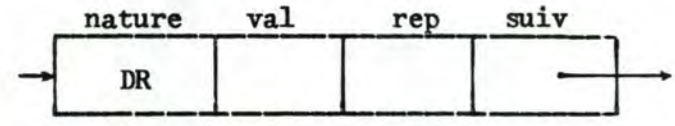
Elle sera composée de :

(1)



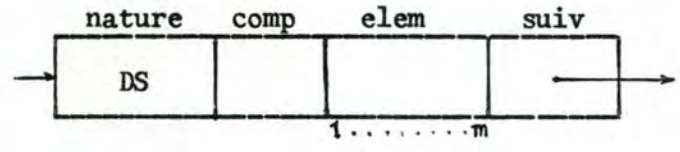
DI exprimera 2 types d'intervalles binf..bsup :
- binf < bsup ou binf > bsup, dans le cas des entiers,
- ORD (binf) < ORD (bsup) ou ORD (binf) > ORD (bsup), dans les cas des caractères et des booléens;

(2)



DR signifiera que l'élément (val) est répété successivement "rep" fois, avec rep 1;

(3)



DS signifiera qu'il s'agit des éléments quelconques ne présen-

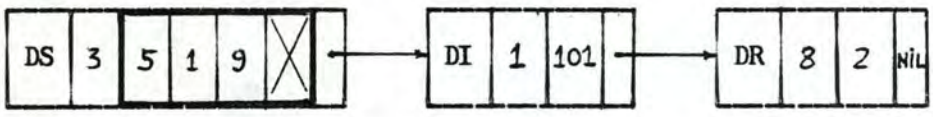
tant ni la caractéristique (1), ni la caractéristique (2);
ces éléments seront stockés dans le tableau elem (array [1..m]),
cfr représentation étalée page 16.

A. SUITES ET ENSEMBLES D'ENTIERS

On utilisera les types suivants :

```
type nat = (DS, DI, DR);
     ensuite = record
                 suiv : ^ensuite;
                 case nature : nat of
                    DI : (binf, bsup : entier);
                    DR : (val, rep : entier);
                    DS : (comp : entier; elem  : array [1..m] of entier)
                 end.
```

Ainsi, par exemple, la suite 5,1,9,1..100,101,8,8 sera représentée comme suit :



Nous aurions pu évidemment envisager d'autres relations (progression par pas > 1,...) mais nous nous limiterons à celles mentionnées plus haut car elles sont les plus visibles dans le cours de "Séminaire de Programmation".
Le compactage devra donc être accompli à chaque opération susceptible de modifier le nombre d'éléments dans la représentation d'une suite ou d'un ensemble.
Il est évident que pour les ensembles, on ne pourra jamais avoir ni de cas "DR", ni de cas "DI" avec binf > bsup.
On remarque donc que si parmi les éléments à représenter, il y a des segments assez longs d'éléments présentant les cas (1) ou (2) (cfr page 32), le gain en place mémoire est appréciable.

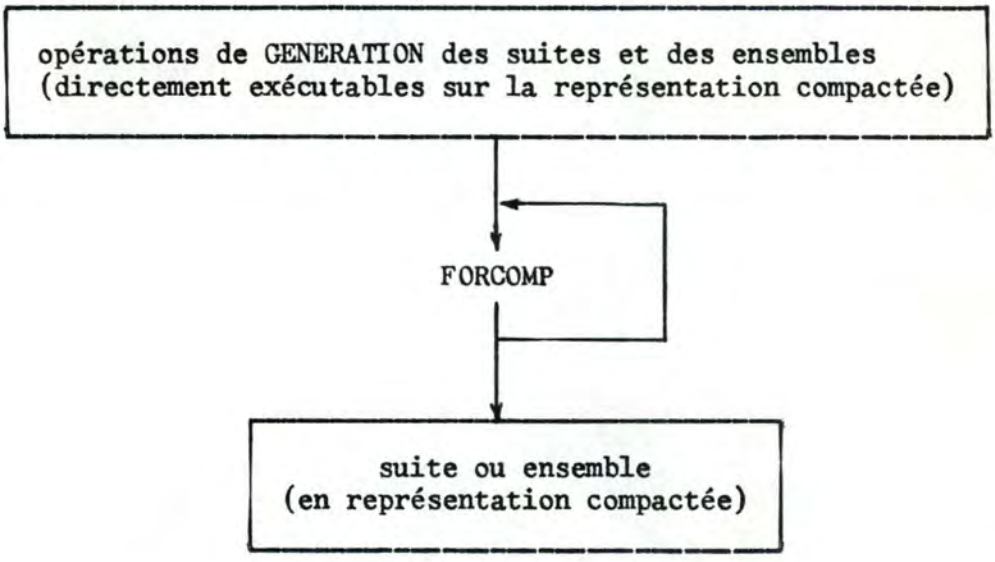
Nous donnons ci-après des règles de traduction des différentes primitives L2 en pseudo-algorithmes dans la représentation compactée, accompagnées éventuellement d'une explication dans les cas où le raisonnement présenté n'était pas évident.

L'algorithme de FORMATION et de COMPACTAGE des suites et ensembles (fonction FORCOMP) qui est le moteur de cette représentation aura comme spécification :

créer la représentation compactée des suites et des ensembles et,
en renvoyer le pointeur.

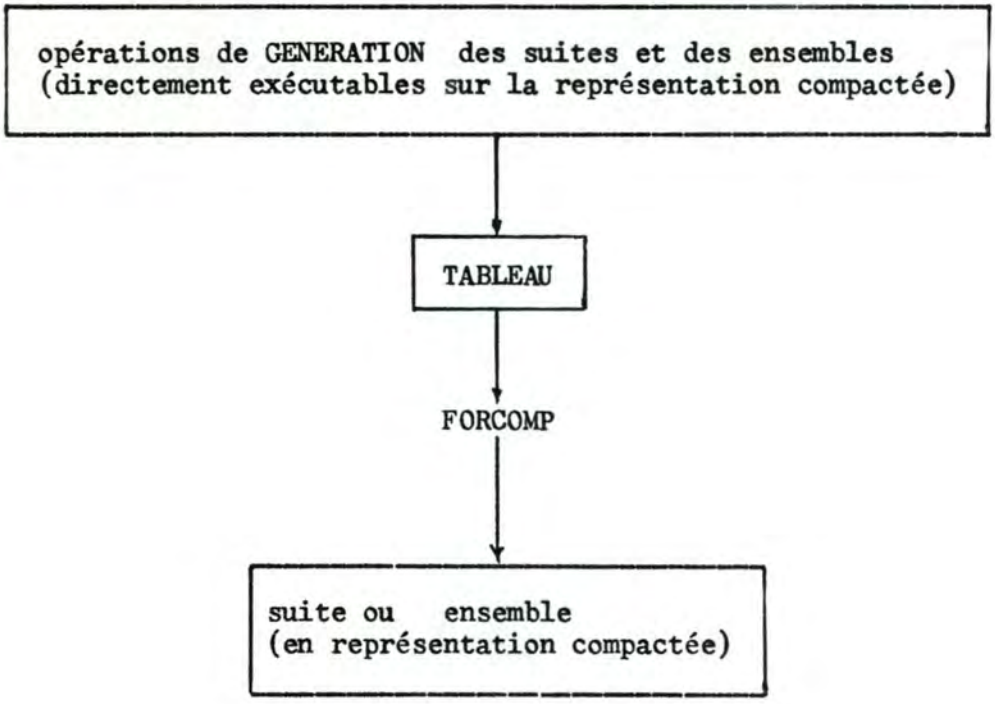
Toutes les opérations de manipulation des suites et des ensembles seront, comme on le verra, directement exécutables sur cette représentation compactée sans structure intermédiaire, sauf pour PERMUT (cfr page 18.(1)) pour les mêmes raisons que dans la représentation étalée.
Les différents éléments qui constituent le résultat d'une opération quelconque de génération seront envoyés à cet algorithme de formation et de compactage des suites et des ensembles (FORCOMP) en vue de parfaire l'opération, et pour cela, nous envisageons deux possibilités :

soit l'envoi direct de ces éléments un par un (au fur et à mesure qu'ils sont fournis par l'opération de génération) sans utilisation d'une structure intermédiaire de stockage de ces éléments,



Ceci exige que FORCOMP puisse être doté de plusieurs points d'entrée et être déclenché ou quitté en des endroits autres que son point d'entrée ou de sortie normal, respectivement;

soit l'utilisation d'une structure intermédiaire de stockage
de ces éléments, à savoir un tableau,



Vu la simplicité de cette deuxième possibilité, nous proposons de l'adopter,
surtout que certaines opérations inhérentes aux représentations des ensembles (TRI, ELIMINATION DES COPIES, ...) pourront immédiatement être accomplies dans ce tableau.
La spécification précise de la fonction FORCOMP (espèce, a, g, d) (cfr page 39) sera alors :

- de créer la représentation compactée des éléments du tableau a dont les indices $\in [g..d]$ et d'en renvoyer le pointeur r si la valeur de la variable "espèce" vaut "t" ou
- de créer la représentation compactée des éléments $\in [g..d]$ et d'en renvoyer le pointeur r si la valeur de la variable "espèce" est "i".

La formation et le compactage des suites et ensembles se feront selon les règles suivantes :

* soit i, un indice du tableau a [g..d] qui est la structure intermédiaire de stockage des éléments des représentations : $g \leq i \leq d$,

(i) si a [i] =a [i+1] alors il s'agit du cas de répétition d'éléments, il faut contrôler si les éléments suivants vérifient cette condition en faisant i ← i + 1, on s'arrêtera et on notera le nombre d'éléments remplissant cette condition quand il n'y a plus d'éléments ou s'il existe un i tel que a [i] ≠ a [i+1],

(ii) si a [i] = a [i+1]+1 alors il s'agit d'un intervalle croissant, il faudra noter a [i] comme borne inférieure de cet intervalle et vérifier successivement si les éléments suivants remplissent cette même condition en faisant i ← i + 1, on s'arrêtera et on notera a [i] comme borne supérieure de cet intervalle quand il n'y a plus d'éléments (i = d + 1) ou s'il existe un i : a [i] ≠ a [i+1] +1,

(iii) si a [i] = a[i+1] -1 alors il s'agit d'un intervalle décroissant, il faudra noter a [i] comme borne inférieure de cet intervalle et vérifier successivement si les éléments suivants remplissent cette même condition en faisant i ← i + 1, on s'arrêtera si i = d + 1 ou s'il existe un i : a [i] ≠ a [i+1] -1 et on notera a [i] comme borne supérieure de cet intervalle décroissant,

(iv) si a [i] ≠ a [i+1] ≠ a [i+1]+1 ≠ a [i+1] -1
alors il s'agit d'un élément a [i] ne présentant aucune des caractéristiques (i), (ii) et (iii),
il suffit alors de vérifier si les éléments suivants remplissent cette condition, auquel cas on les stockera (avec a[i]) dans un ou plusieurs tableaux "elem" (cfr représentation étalée) quand i > d ou s'il existe un i : a [i] = a [i+1] ou
a [i] = a [i+1] +1 ou
a [i] = a [i+1] -1

* soit i, un indice du tableau intermédiaire de stockage des éléments :
i = d,
alors il suffit de mettre cet élément (a [i]) dans un tableau "elem" (cfr représentation étalée).

Ces règles étant données, la fonction FORCOMP formera et compactera toute suite ou tout ensemble d'UNE ET D'UNE SEULE FACON SANS AMBIGUITE. Dès lors, différentes représentations définiront différents ensembles (ou suites) et deux représentations identiques déterminent le même ensemble (ou la même suite).

Soient r, struct et lien des variables de type "ensuite" (cfr page 34), FORCOMP pourrait se présenter comme suit :

```
fonction FORCOMP (espèce, a, g, d) : ^ensuite;
    début
    r ← nil
    case espèce of
    'i' :  si g ≤ d alors | new (struct)
                          | r ← struct
                          | struct^.suiv ← nil
                          | si g < d alors | struct^.nature ← di
                          |                | struct^.binf ← g
                          |                | struct^.bsup ← d                    (°)
                          |          sinon | struct^.nature ← ds
                          |                | struct^.comp ← 1
                          |                | struct^.elem [1] ← g
    't' :  firstime ← vrai
           tant que g ≤ d faire | preced ← a [g]
                                | si  g + 1 > d
                                |     alors  c ← 's'
                                |     sinon  si preced + 1 = a [g + 1]
                                |                 alors  c ← 'c'                  (°°)
                                |                 sinon  si  preced - 1 = a [g + 1]
                                |                             alors  c ← 'd'
                                |                             sinon  si  preced = a [g + 1]
                                |                                         alors  c ← 'r'
                                |                                         sinon  c ← 's'
                                | new (struct)
                                | si firstime alors  r ← struct
                                |                   firstime ← faux               (°°°)
                                |              sinon  lien^.suiv ← struct
                                | lien ← struct
                                | case c of
                                | 'c' : | g ← g + 1
                                |      | borne ← a [g]
                                |      | g ← g + 1
                                |      | tant que  g ≤ d et borne + 1 = a [g] faire  borne ← a [g]
                                |      |                                             g ← g + 1
                                |      | struct^.nature ← di                       (°°°°)
                                |      | struct^.binf ← preced
                                |      | struct^.bsup ← borne
                                |
                                |      .../...
```

```
                    'd' : | g ← g + 1                                                          }
                          | borne ← a [g]                                                     }
                          | g ← g + 1                                                         }
                          | tant que g ≤ d et borne - 1 = a [g] faire | borne ← a [g]        } (-)
                          |                                           | g ← g + 1            }
                          | struct^.nature ← di                                              }
                          | struct^.binf ← preced                                           }
                          | struct^.bsup ← borne                                            }
                    'r' : | g ← g + 2                                                         }
                          | k ← 2                                                             }
                          | tant que g ≤ d et preced = a [g] faire | k ← k + 1              }
                          |                                        | g ← g + 1              } (—)
                          | struct^.nature ← dr                                             }
                          | struct^.val ← preced                                            }
                          | struct^.rep ← k                                                 }
                    's' : cfr représentation étalée (fonction GENERER page 27)        (——)
si r ≠ nil alors struct^.suiv ← nil
forcomp ← r                                                                           (——)
fin
```

(°) : cas où l'on construit une suite ou un ensemble avec les éléments de l'intervalle g..d

Dans le cas où l'on doit construire une suite ou un ensemble avec les éléments a g , a g+1 , ..., a d , il s'agira d'abord de repérer (°°) de quelle succession d'éléments il est question :

* progression par pas de +1 ('c')
* progression par pas de -1 ('d')
* répétition ('r')
* quelconque ('s') et,

ensuite de construire effectivement la structure correspondante

* progression par pas de +1 (°°°°)
* progression par pas de -1 (-)
* répétition (—)
* quelconque (——)

en attachant les différentes structures construites (°°°) pour fournir la structure finale demandée (——)

1. Expressions De Manipulation Des Suites Et Ensembles D'Entiers

a. EMPTY (Sk) et EMPTY (Ek) : cfr page 18.(0)

b. FIRST (Sk) : (précondition : p ≠ nil)

```
fonction FIRST (p) : entier;
    début
    case p^.nature of
         di :  first ← p^.binf
         dr :  first ← p^.val
         ds :  first ← p^.elem [1]
    fin
```

c. LONG (Sk) :

```
fonction LONG (p) : entier;
    début
    l ← 0
    tant que p ≠ nil
         faire  case p^.nature of
                   di :  l ← l + ABS (p^.binf - p^.bsup) + 1
                   dr :  l ← l + p^.rep
                   ds :  l ← l + p^.comp
                p ← p^.suiv
    long ← l
    fin
```

d. CARD (Ek) : cfr LONG (Sk) ci-dessus, mais il est évident qu'ici,
on n'aura ni le cas "dr",
ni le cas "di" avec binf > bsup

e. LAST (Sk) : (précondition : p ≠ nil)

```
fonction LAST (p) : entier;
    début
    tant que p^suiv ≠ nil faire  p ← p^.suiv
    case p^.nature of
        di :  last ← p^.bsup
        dr :  last ← p^.val
        ds :  last ← p^.elem [p^.comp]
    fin
```

f. MIN (Ek) :

```
fonction MIN (p) : entier;
    début
    si p ≠ nil alors  case p^.nature of
                          di : min ← p^.binf
                          ds : min ← p^.elem [1]
                      sinon  min ← zmax
    fin
```

g. MAX (Ek) :

```
fonction MAX (p) : entier;
    début
    si p ≠ nil alors  tant que p^.suiv ≠ nil faire  p ← p^.suiv
                      case p^.nature of
                          di :  max ← p^.binf
                          ds :  max ← p^.elem [p^.comp]
                      sinon  max ← zmin
    fin
```

h. ELT (Ek) : (précondition : p ≠ nil)

```
fonction ELT (p) : entier;
    début
    case p^.nature of
        di :  elt ← p^.binf
        ds :  elt ← p^.elem [1]
    fin
```

i. TRIDEC (Sk) :

```
fonction TRIDEC (p) : booléen;
|début
|mval ← zmax
|trid ← vrai
|tant que p ≠ nil et trid
|      faire | case p^.nature of
|             |    di : | si p^.binf < p^.bsup
|             |         |    alors  trid ← faux
|             |         |    sinon  si mval ≥ p^.binf
|             |         |              alors  si mval > p^.binf
|             |         |                        alors  mval ← p^.bsup
|             |         |              sinon  trid ← faux
|             |    ds : | i ← 1
|             |         | tant que i ≤ p^.comp et trid
|             |         |      faire  si mval ≥ p^.elem [i]
|             |         |                alors  si mval > p^.elem [i]
|             |         |                          alors
|             |         |                          mval ← p^.elem [i]
|             |         |                       i ← i + 1
|             |         |                sinon  trid ← faux
|             |    dr : | si mval ≥ p^.val
|             |         |    alors  si mval > p^.val
|             |         |              alors  mval ← p^.val
|             |         |    sinon  trid ← faux
|             | p ← p^.suiv
|si trid alors   tridec ← vrai
|        sinon   tridec ← faux
|fin
```

j. TRISTRDEC (Sk), TRICROIS (Sk) et TRISTRCROIS (Sk) :
les algorithmes correspondants sont construits sur la même logique que celle de TRIDEC (Sk) ci avant, il suffit d'y apporter les aménégements adéquats
(cfr TRISTRDEC (Sk), TRICROIS (Sk) et TRISTRCROIS (Sk) dans la représentation étalée).

k. PROD FOR <id> IN Sk OF <expr-arith> :

```
fonction PRODFOR (p, id, expr-arith) : entier;
|début
|v ← 1
|tant que p ≠ nil
|      faire | case p^.nature of
|             | di : | si p^.binf < p^.bsup
|             |      |    alors  for id ← p^.binf to p^.bsup
|             |      |                 do  v ← v * expr-arith (id)
|             |      |    sinon  for id ← p^.binf downto p^.bsup
|             |      |                 do  v ← v * expr-arith (id)
|             | ds : | for id ← 1 to p^.comp
|             |      |     do  v ← v * expr-arith (p^.elem [id] )
|             | dr : | k ← expr-arith (p^.val) ** p^.rep
|             |      | v ← v * k
|             | p ← p^.suiv
|prodfor ← v
|fin
```

l. SUM FOR <id> IN Sk OF <expr-arith> :

```
fonction SUMFOR (p, id, expr-arith) : entier;
|début
|s ← 0
|tant que p ≠ nil
|      faire | case p^.nature of
|             | di : | si p^.binf < p^.bsup
|             |      |    alors  for id ← p^.binf to p^.bsup
|             |      |                 do  s ← s + expr-arith (id)
|             |      |    sinon  for id ← p^.binf downto p^.bsup
|             |      |                 do  s ← s + expr-arith (id)
|             | ds : | for id ← 1 to p^.comp
|             |      |     do  s ← s + expr-arith (p^.elem [id] )
|             | dr : | k ← expr-arith (p^.val) * p^.rep
|             |      | s ← s + k
|             | p ← p^.suiv
|sumfor ← s
|fin
```

m. MAX FOR <id> IN Ek OF <expr-arith> :

```
fonction MAXFOR (p, id, expr-arith) : entier;
    début
    max ← zmin
    tant que p ≠ nil
        faire | case p^.nature of
              |     di : | for id ← p^.binf to p^.bsup
              |          |     do | k ← expr-arith (id)
              |          |        | si max < k alors max ← k
              |     ds : | for id ← 1 to p^.comp
              |          |     do | k ← expr-arith (p^.elem [id] )
              |          |        | si max < k alors max ← k
              | p ← p^.suiv
    maxfor ← max
    fin
```

n. MIN FOR <id> IN Ek OF <expr-arith> :

même raisonnement que MAXFOR ci-dessus,
les instructions "max ← zmin",
"si max < k alors max ← k" et
"maxfor ← max" seront respectivement remplacées
par les instructions "min ← zmax",
"si min > k alors min ← k" et
"minfor ← min".

o. FOR ALL <id> IN Ek :<(bexp)>

```
fonction FORALL (p, id, bexp) : booléen;
    début
    f ← vrai
    tant que p ≠ nil et f
        faire | case p^.nature of
              |     di : | id ← p^.binf
              |          | tant que id ≤ p^.bsup et f
              |          |     faire si bexp (id) = faux
              |          |            alors f ← faux
              |          |            sinon id ← id + 1
              |     ds : | id ← 1
              |          | tant que id ≤ p^.comp et f
              |          |     faire si bexp (p^.elem [id]) = faux
              |          |            alors f ← faux
              |          |            sinon id ← id + 1
              | p ← p^.suiv
    si f alors forall ← vrai
         sinon forall ← faux
    fin
```

p. THERE EXISTS id IN Ek : (bexp)

```
fonction THEREXISTS (p, id, bexp) : booléen;
    début
    t ← faux
    tant que p ≠ nil et t = faux
          faire  case p^.nature of
                    di :  id ← p^.binf
                          tant que id ≤ p^.bsup et t = faux
                                faire  si bexp (id) alors  t ← vrai
                                                    sinon  id ← id + 1
                    ds :  id ← 1
                          tant que id ≤ p^.comp et t = faux
                                faire  si bexp (p^.elem [id] )
                                             alors  t ← vrai
                                             sinon  id ← id + 1
                 p ← p^.suiv
    si t alors  therexists ← vrai
          sinon  therexists ← faux
    fin
```

q. EGAL (Sk, Sj) :

```
fonction EGAL (p, q) : booléen;                                   (°)
    début
    eg ← vrai
    tant que p ≠ nil et q ≠ nil et eg
    faire  si p^.nature ≠ q^.nature
                alors  eg ← faux
                sinon  case p^.nature of
                          ds :  si p^.comp ≠ q^.comp
                                alors eg ← faux
                                sinon i ← 1
                                      tant que eg et i ≤ p^.comp
                                      faire  si p^.elem [i] ≠ q^.elem [i]
                                                   alors  eg ← faux
                                                   sinon  i ← i + 1
                          di :  si p^.binf ≠ q^.binf ou
                                   p^.bsup ≠ q^.bsup
                                alors eg ← faux
                          dr :  si p^.val ≠ q^.val  ou
                                   p^.rep ≠ q^.rep
                                alors  eg ← faux
                       p ← p^.suiv
                       q ← q^.suiv
    si p ≠ nil ou q ≠ nil alors eg ← faux
    si eg alors  egal ← vrai
          sinon  egal ← faux
    fin
```

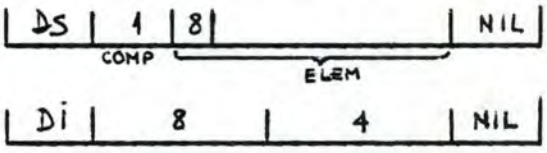
(°) : nous préférons cette version à la version récursive car elle est meilleure au point de vue temps d'exécution !

r. PREFIXE (Sk, Sj) :

```
fonction PREFIXE (p, q) : booléen;                    (°)
    début
    i ← 1;  j ← 1;
    f1 ← vrai;  f2 ← vrai;
    f3 ← vrai;  f4 ← vrai;
    pref ← vrai
    tant que p ≠ nil et q ≠ nil et pref
    faire | case p^.nature of
          |   di : si f1 alors | f1 ← faux
          |                    | k1 ← p^.binf
          |      un ← k1
          |   ds : un ← p^.elem [i]
          |   dr : si f2 alors | f2 ← faux
          |                    | k2 ← 1
          |      un ← p^.val                              } (1)
          | case q^.nature of
          |   di : si f3 alors | f3 ← faux
          |                    | k3 ← q^.binf
          |      deux ← k3
          |   ds : deux ← q^.elem [j]
          |   dr : si f4 alors | f4 ← faux
          |                    | k4 ← 1
          |      deux ← q^.val
          | si un ≠ deux                                 (2)
          | alors pref ← faux
          | sinon | case p^.nature of
          |       |   di : | si p^.binf < p^.bsup
          |       |        | alors | k1 ← k1 + 1
          |       |        |       | si k1 > p^.bsup alors | p ← p^.suiv
          |       |        |       |                       | f1 ← vrai
          |       |        | sinon | k1 ← k1 - 1
          |       |        |       | si k1 < p^.bsup alors | p ← p^.suiv
          |       |        |       |                       | f1 ← vrai
          |       |   ds : | i ← i + 1
          |       |        | si i > p^.comp alors | p ← p^.suiv
          |       |        |                      | i ← 1
          |       |   dr : | k2 ← k2 + 1
          |       |        | si k2 > p^.rep alors | p ← p^.suiv
          |       |        |                      | f2 ← vrai
          |       | case q^.nature of                                  (3)
          |       |   di : | si q^.binf < q^.bsup
          |       |        | alors | k3 ← k3 + 1
          |       |        |       | si k3 > q^.bsup alors | q ← q^.suiv
          |       |        |       |                       | f3 ← vrai
          |       |        | sinon | k3 ← k3 - 1
          |       |        |       | si k3 < q^.bsup alors | q ← q^.suiv
          |       |        |       |                       | f3 ← vrai
          |       |   ds : | j ← j + 1
          |       |        | si j > q^.comp alors | q ← q^.suiv
          |       |        |                      | j ← 1
          |       |   dr : | k4 ← k4 + 1
          |       |        | si k4 > q^.rep alors | q ← q^.suiv
          |       |        |                      | f4 ← vrai
    si p ≠ nil alors pref ← faux
    si pref alors préfixe ← vrai
            sinon préfixe ← faux
    fin
```

(°) : Encore une fois, nous avons dû choisir :
l'algorithmique de PREFIXE proposée ici, bien que longue (car elle fait une comparaison élément par élément) est cependant
- la plus rapide (au point de vue exécution) de tous les algorithmes (de PREFIXE) qui ont été développés,
- claire et très simple,
- très générale; elle n'a, en effet pas besoin de tester les différents cas particuliers que cache cette primitive dans la représentation compactée.

Exemple de cas particulier :
Une suite Sk peut être préfixe d'une autre suite Sj et être de "nature" différente,



est un préfixe de

.

Un programme qui ne comparerait pas tous les éléments un par un serait obligé de tester la possibilité d'avoir ce cas,

(1) : les 2 éléments à comparer sont mis dans les variables un et deux,

(2) : la comparaison

(3) : la détermination des adresses des 2 éléments suivants à comparer.

s. EGAL (Ek, Ej) : même chose que EGAL (Sk, Sj) sauf qu'on n'aura pas ici ni le cas "dr", ni le cas "di" avec binf > bsup.

t. PERMUT (Sk, Sj) : Soit la procédure SORT (p, table, i),

cfr page 18.(1), où la fonction FORMTAB serait enrichie des cas "dr" et "di" (cfr page 44), alors la traduction de cette primitive dans la représentation compactée est la même que celle de la représentation étalée.

u. INCLUS (Ek, Ej) :

```
fonction INCLUS (p, q) : booléen;
début
i ← 1;  j ← 1;  g ← 3
f1 ← vrai;  f2 ← vrai
tant que p ≠ nil et q ≠ nil
faire | si g = 3 alors | case p^.nature of
                       |   di : | si f1 alors | f1 ← faux
                       |        |             | k1 ← p^.binf
                       |        | un ← k1
                       |   ds : un ← p^.elem [i]
      | si g = 2 ou g = 3 alors | case q^.nature of
                               |   di : | si f2 alors | f2 ← faux
                               |        |             | k2 ← q^.binf
                               |        | deux ← k2
                               |   ds : deux ← q^.elem [j]
      | si un ≠ deux alors g ← 2
      |               sinon g ← 3
      | si g = 3 alors | case p^.nature of
      |                |   di : | k1 ← k1 + 1
      |                |        | si k1 > p^.bsup alors | p ← p^.suiv
      |                |        |                       | f1 ← vrai
      |                |   ds : | i ← i + 1
      |                |        | si i > p^.comp alors | p ← p^.suiv
      |                |        |                      | i ← 1
      | si g = 2 ou g = 3 alors | case q^.nature of
      |                         |   di : | k2 ← k2 + 1
      |                         |        | si k2 > q^.bsup
      |                         |        |    alors | q ← q^.suiv
      |                         |        |          | f2 ← vrai
      |                         |   ds : | j ← j + 1
      |                         |        | si j > q^.comp
      |                         |        |    alors | q ← q^.suiv
      |                         |        |          | j ← 1
si p = nil alors  inclus ← vrai
           sinon  inclus ← faux
fin
```

v. SSUITE (Sk, Sj) : en ajoutant les cas "dr" et "di" (binf > bsup) à l'algorithme de INCLUS (ci-dessus), on obtiendra la traduction de SSUITE.

w. SUFFIXE (Sk, Sj) et SEGMENT (Sk, Sj) : en enrichissant les fonctions SUFFIXE et SEGMENT de la représentation étalée des cas "dr" et "di" et en procédant comme dans PREFIXE (rep. compactée), ces 2 fonctions booléennes ne causent aucun problème dans la représentation compactée malgré leur longueur.

2. Expressions De Génération Des Suites Et Ensembles D'Entiers

Soit la fonction TRIER (table, n) dont la spécification est :
- de trier (en quicksort, par exemple) le tableau "table" de taille n et,
- de renvoyer le pointeur r de la représentation compactée de l'ensemble (triée) formé des éléments de ce tableau,

```
fonction TRIER (table, n) : ^ensuite;
    début
    QSORT (table, 1, n)
    k ← ELIMINER (table, n)
    trier ← FORCOMP ('t', table, 1, k)
    fin
```

(cfr QSORT page 25 et ELIMINER page 27),

alors, les fonctions FORMSUIT (expr) (cfr page 20.(0)) et FORMENS (expr) (cfr page 28.(0)) dont les spécifications sont :

de produire la représentation compactée, respectivement de la suite et de l'ensemble demandés en fonction des différentes expressions "expr" de génération et d'en renvoyer le pointeur r,

(dans lesquelles, pour des raisons de clarté, nous supposerons que toutes les variables ont été correctement initialisées et laisserons de côté tous les contrôles qui ne nous intéressent pas directement) deviendraient :

```
fonction FORMSUIT (expr) : ^ensuite;
  DEBUT
  case expr of
  [tab] : |début
          |formsuit ← FORCOMP ('t', table, 1, n)
          |fin

  [exp] :  cfr  représentation étalée page 20.(0)

  [exp1..exp2] : |début
                 |formsuit ← FORCOMP ('i', ?, évaluation (exp1), évaluation (exp2))
                 |end

  [tab [Sk]] : |début
               |j ← 0
               |tant que p ≠ nil faire |case p^.nature of
               |                       |di : |si p^.binf < p^.bsup alors  for i ← p^.binf to p^.bsup do |j ← j + 1
               |                       |     |                                                            |table [j] ← tab [i]
               |                       |     |                sinon  for i ← p^.binf downto p^.bsup
               |                       |     |                                                     do |j ← j + 1
               |                       |     |                                                        |table [j] ← tab [i]
               |                       |dr : |for i ← 1 to p^.rep do |j ← j + 1
               |                       |     |                       |table [j] ← tab [p^.val]
               |                       |ds : |for i ← 1 to p^.comp do |j ← j + 1
               |                       |     |                        |table [j] ← tab [p^.elem [i]]
               |                       |p   p^.suiv
               |formsuit ← FORCOMP ('t', table, 1, j)
               |fin
```

```
[id of Sk | bexp] : début
                   j ← 0
                   tant que p ≠ nil faire | case p^.nature of
                                          | di : si p^.binf < p^.bsup alors for i ← p^.binf to p^.bsup
                                          |                                 do si bexp (i) alors | j ← j + 1
                                          |                                                      | table [j] ← i
                                          |                            sinon for i ← p^.binf downto p^.bsup
                                          |                                 do si bexp (i) alors | j ← j + 1
                                          |                                                      | table [j] ← i
                                          | ds : for i ← 1 to p^.comp do si bexp (p^.elem [i])
                                          |                                 alors | j ← j + 1
                                          |                                       | table [j] ← p^.elem [i]
                                          | dr : si bexp (p^.val) alors for i ← 1 to p^.rep do | j ← j + 1
                                          |                                                    | table [j] ← p^.val
                                          | p ← p^.suiv
                   formsuit ← FORCOMP ('t', table, 1, j)
                   fin

CONCAT (Sk, Sj) : début
                  j ← 0
                  s ← p
                  cpt ← 1
                  répéter | tant que s ≠ nil faire | case s^.nature of
                          |                        | di : si s^.binf < s^.bsup alors for i ← s^.binf to s^.bsup
                          |                        |                                 do | j ← j + 1
                          |                        |                                    | table [j] ← i
                          |                        |                            sinon for i ← s^.binf downto s^.bsup
                          |                        |                                 do | j ← j + 1
                          |                        |                                    | table [j] ← i
                          |                        | dr : for i ← 1 to s^.rep do | j ← j + 1
                          |                        |                             | table [j] ← s^.val
                          |                        | ds : for i ← 1 to s^.comp do | j ← j + 1
                          |                        |                              | table [j] ← s^.lem [i]
                          |                        | s ← s^.suiv
                          | cpt ← cpt + 1
                          | s ← q
                  jusqu'à ce que cpt > 2
                  formsuit ← FORCOMP ('t', table, 1, j)
                  fin

FIN
```

```
fonction FORMENS (expr) : ^ensuite
    DEBUT
    case expr of
    {exp} :  cfr représentation étalée page 28.(1)

    {exp1..exp2} :  même traduction que  exp1..exp2  (cfr page 43.(0)) où "formsuit" est remplacé par "formens"

    {tab} :  début
             formens ← TRIER (tab, n)          (n étant la taille du tableau tab)
             fin

    {Sk} :   début
             j ← 0
             tant que p ≠ nil faire  case p^.nature of
                                     di : si p^.binf < p^.bsup alors  for i ← p^.binf to p^.bsup do  j ← j + 1
                                                                                                    table [j] ← i
                                          sinon  for i ← p^.binf downto p^.bsup do  j ← j + 1
                                                                                    table [j] ← i
                                     dr : j ← j + 1
                                          table [j] ← p^.val
                                     ds : for i ← 1 to p^.comp do  j ← j + 1
                                                                   table [j] ← p^.elem [i]
                                     p ← p^.suiv
             formens ← TRIER (table, j)
             fin

    {tab [Sk]} : même traduction algorithmique que  tab  Sk   (cfr page 43.(0)) mais l'instruction
                 "formsuit ← FORCOMP ('t', table, 1, j)" sera remplacée par l'instruction "formens ← TRIER (table, j)".
```

```
{id of Ek | bexp} : | début
                    | j — 0
                    | tant que p ≠ nil faire | case p^.nature of
                    |                        | di : for i←p^.binf to p^.bsup do si bexp (i) alors | j ← j + 1
                    |                        |                                                    | table [ j]← i
                    |                        | ds : for i←1 to p^.comp do si bexp (p^.elem [i] ) alors | j←j + 1
                    |                        |                                                         | table[j]←p^.elem [ i]
                    |                        | p←p^.suiv
                    | formens ← FORCOMP ('t', table, 1, j)
                    | fin
```

```
UNION (Ek, Ej) : | début
                 | j←0
                 | s←p
                 | cpt← 1
                 | répéter | tant que s ≠ nil faire | case s^.nature of
                 |        |                        |   di : for i←s^.binf to s^.bsup do | j ←j + 1
                 |        |                        |                                    | table [j] ←i
                 |        |                        |   ds : for i←1 to s^.comp do | j←j + 1
                 |        |                        |                              | table [j] ← s^.elem [i]
                 |        |                        | s ← s^.suiv
                 |        | cpt ← cpt + 1
                 |        | s ← q
                 | jusqu'à ce que cpt > 2
                 | formens ← TRIER (table, j)
                 | fin
```

```
INTER (Ek, Ej) : début
                 i ← 1;    j ← 1;    l ← 0;    g ← 3
                 f1 ← vrai;    f2 ← vrai
                 tant que p ≠ nil et q ≠ nil
                 faire | si g = 1 ou g = 3 alors | case p^.nature of
                       |                         | di : | si f1 alors | f1 ← faux
                       |                         |      |             | k1 ← p^.binf
                       |                         |      | un ← k1
                       |                         | ds : un ← p^.elem [i]
                       |       si g = 2 ou g = 3 alors | case q^.nature of
                       |                               | di : | si f2 alors | f2 ← faux
                       |                               |      |             | k2 ← q^.binf
                       |                               |      | deux ← k2
                       |                               | ds : deux ← q^.elem [j]
                       |       si un = deux alors | g ← 3
                       |                          | l ← l + 1
                       |                          | table [l] ← un
                       |                  sinon | si un < deux alors g ← 1
                       |                                sinon g ← 2
                       |       si g = 1 ou g = 3 alors | case p^.nature of
                       |                               | di : | k1 ← k1 + 1
                       |                               |      | si k1 > p^.bsup alors | p ← p^.suiv
                       |                               |      |                       | f1 ← vrai
                       |                               | ds : | i ← i + 1
                       |                               |      | si i > p^.comp alors | p ← p^.Suiv
                       |                               |      |                      | i ← 1
                       |       si g = 2 ou g = 3 alors | case q^.nature of
                       |                               | di : | k2 ← k2 + 1
                       |                               |      | si k2 > q^.bsup alors | q ← q^.suiv
                       |                               |      |                       | f2 ← vrai
                       |                               | ds : | j ← j + 1
                       |                               |      | si j > q^.comp alors | q ← q^.suiv
                       |                               |      |                      | j ← 1
                 formens ← FORCOMP ('t', table, 1, 1)
                 fin
```

DIFFER (Ek, Ej) : en se basant su INTER ci-dessus et DIFFER de la représentation étalée (cfr page 28.(2)), il n'y a aucun problème à construire le pseudo-algorithme de DIFFER dans la représentation compactée.

FIN

3. Autres Expressions

Soit la fonction FORMTAB (p, table) de même spécification que celle définie page 21 mais enrichie des cas "dr" et "di" :

```
fonction FORMTAB (p, table) : entier;
    |début
    |i ← 0
    |tant que p ≠ nil
    |faire |case p^.nature of
    |      |ds : |for j←1 to p^.comp do |i ← i + 1
    |      |     |                       |table [i] ← p^.elem [j]
    |      |     |                       |j ← j + 1
    |      |di : |si p^.binf < p^.bsup
    |      |     |   alors  for j← p^.binf to p^.bsup do |i ← i + 1
    |      |     |                                       |table [i] ← j
    |      |     |   sinon  for j← p^.binf downto p^.bsup
    |      |     |                                    do |i ← i + 1
    |      |     |                                       |table [i] ← j
    |      |dr : |for j←1 to p^.rep do |i ← i + 1
    |      |     |                     |table [i] ← p^.val
    |      |p ← p^.suiv
    |formtab ← i
    |fin,
```

alors, les fonctions TAIL et HEAD deviendraient :

```
fonction TAIL (p) : ^ensuite;
    |début
    |k ← FORMTAB (p, table)
    |tail ← FORCOMP ('t', table, 2, k)
    |fin
```

```
fonction HEAD (p) : ^ensuite;
    |début
    |k ← FORMTAB (p, table)
    |head ← FORCOMP ('t', table, 1, k - 1)
    |fin
```

B. SUITES DE CARACTERES

Soit l'ensemble C déjà évoqué page 22, les règles développées (en représentation compactée) pour les suites d'entiers restent valables pour les suites de caractères.
Les modifications à y apporter viendraient essentiellement de la nécessité de "numéraliser" les caractères pour pouvoir effectuer certaines opérations (dans les primitives de génération, de manipulation et dans FORCOMP);
les fonctions ORD et CHR sont donc un secours utile. Ces modifications étant évidentes, il n'est pas, par conséquent intéressant d'écrire dans ce mémoire les pseudo-algorithmes modifiés pour ces suites de caractères.

C. ENSEMBLES DE CARACTERES

Il serait plus intelligent de garder la même structure unique que celle utilisée pour les ensembles de caractères en représentation étalée (cfr page 29).
Elle est efficace en temps et, en général, économique en consommation de l'espace mémoire.

D. SUITES ET ENSEMBLES DE BOOLEENS

Les ensembles de booléens, comme on l'a déjà dit, n'ont aucune utilité pratique. Théoriquement, on pourrait adopter la même structure que pour la représentation étalée des ensembles de booléens (cfr page 31).
Les suites de booléens seront implémentées de la même manière que les suites d'entiers en représentation compactée; les modifications à y apporter seront les mêmes que celles apportées aux algorithmes de la représentation étalée des suites de booléens (cfr page 22).
Il serait cependant inefficace d'utiliser le cas "di" dans la représentation compactée des suites et ensembles de booléens car ce cas représentera systématiquement une plus forte consommation de la place mémoire par rapport à la représentation étalée.

Il serait plus clair pour certains pseudo-algorithmes développés d'avoir une primitive d'accès à l'élément suivant comme par exemple :

```
procédure NEXT (p, i);
    début
    g ← faux
    case p^.nature of
       ds : i ← i + 1
            si i > p^.comp alors g ← vrai
       di : si p^.binf < p^.bsup alors i ← i + 1
                                       si i > p^.bsup alors g ← vrai
                                  sinon i ← i - 1
                                       si i < p^.bsup alors g ← vrai
       dr : i ← i + 1
            si i > p^.rep alors g ← vrai
    si g alors p ← p^.suiv
                case p^.nature of
                ds, dr : i ← 1
                di : i ← p^.binf
    fin
```

avec comme précondition p ≠ nil.

IV. GESTION DYNAMIQUE DE LA MEMOIRE

Les instructions de L2 risquant à tout moment de consommer un espace de mémoire aussi grand que l'on veut, on voudrait alors pour tout programme L2 ne conserver à chaque instant que la mémoire (occupée par les suites et les ensembles) strictement nécessaire.
Pour la gestion de l'utilisation dynamique de la mémoire, on utilise généralement deux méthodes :

- le garbage collector ou
- la libération immédiate.

Dans le cadre de ce travail, nous opterons pour le second procédé qui est plus facile car d'une part, nous savons en scrutant les divers programmes testés, les structures qui deviennent inutiles et encombrantes pour la mémoire et d'autre part, nous nous contentons seulement d'évaluer la place strictement nécessaire à tout moment.
Mais, dans un contexte beaucoup plus général, c'est plutôt la première méthode qui correspondrait le mieux à cette gestion.

Il faudrait donc, pour ce qui nous concerne, reconnaître les structures qui sont devenues obsolètes et les libérer pour récupérer de la place mémoire.
Pour chaque programme L2, il pourra être nécessaire, pour mieux se rendre compte de cette activité dynamique de la mémoire, d'en fournir un tracé graphique dont l'abscisse serait les différents moments d'allocation et de désallocation et dont l'ordonné donnerait cette quantité de mémoire (occupée) strictement nécessaire. On tiendrait donc finalement la comptabilité des allocations et des libérations des mots de mémoire utilisés par les suites et les ensembles.

La libération des structures (représentations des suites et des ensembles) se fera soit (juste) après ou avant l'exécution de l'expression E qui a demandé ou qui demande l'allocation, si la suite ou l'ensemble qui a été crée n'a plus aucune raison d'exister,
soit à la fin du programme L2 sinon.

On sait donc, dans certains cas, exécuter automatiquement la libération car on connait les expressions qui laissent des structures vieillies après leur exécution :

* d'une manière générale, toute place mémoire occupée par une suite ou par un ensemble provenant des facteurs :

  "in [exp1..exp2] ", "in {exp1..exp2} ",
  "of [exp1..exp2] ", "of {exp1..exp2} ",
  " [exp1..exp2] ", " {exp1..exp2} ",
  " [exp] ", " { exp } "
  (exp, exp1 et exp2 étant des expressions arithmétiques)

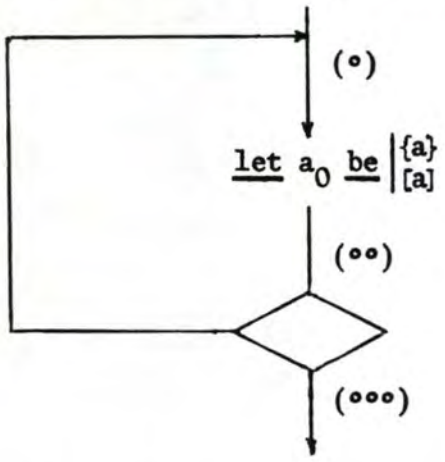
  contenus dans une expression de génération E, sera récupérée après l'exécution de celle-ci.

  Ex : - l'ensemble {exp1..exp2} crée par l'expression
  "for all i in {exp1..exp2} : bexp"
  sera détruit (et sa place récupérée) après l'exécution de celle-ci;
  - la place mémoire occupée par les suites [exp1..exp2] et [exp3..exp4] dans l'expression "..... CONCAT ([exp1..exp2] , [exp3..exp4]) sera libérée après l'exécution de celle-ci;
  - dans l'expression ".... EGAL ({exp1} , {exp2}) ", les ensembles {exp1} et {exp2} n'auront plus aucune raison d'exister après son exécution;
  - l'espace mémoire occupé par la suite [exp1..exp2] dans l'expression " .... id of [exp1..exp2] | bexp "
  sera récupéré après l'exécution de cette expression,

* soit V, tout ensemble ou toute suite crée par une expression de génération E dépourvue du "signe d'assignation", alors la place occupée par V peut être récupérée sans problème après l'exécution de E;

  Ex : - quelles que soient les expressions de type ensemble Ek, Ej, Ei et El, les ensembles Ek \ Ej et Ei ∪ El produits par l'expression " .... EGAL (DIFFER (Ek, Ej), UNION (Ei, El)) ... " peuvent disparaître après l'exécution de celle-ci;
  - quelles que soient les expressions de type suite Si, Sk, Sj, Sl et Sm, les suites Sk | Sj et Sl | Sm produites par l'instruction "Si := CONCAT (CONCAT (Sk, Sj), CONCAT (Sl, Sm))
  peuvent disparaître sans problème après l'exécution de cette instruction, les expressions de génération Sk | Sj et Sl | Sm étant dépourvues du signe d'assignation ":=",

* la pseudo-instruction let est aussi susceptible de produire un nombre considérable de structures impertinentes, ce sera le cas si elle se trouve dans une boucle.
Dans ce cas, la libération et la récupération se feront avant l'exécution de ladite instruction; en effet, dans la boucle



l'ensemble ou la suite a_0 ne peut être libéré que juste avant l'exécution de l'instruction let,
a_0 pouvant être manipulé partout ailleurs en (°), (°°) et (°°°).
Il pourrait alors être très intéressant, dans ce cas, d'inclure la procédure de la désallocation dans la traduction algorithmique de cette instruction.

* les instructions d'assignation Sk := ....., Ek := ..... (Sk et Ek étant des variables de désignation de suite et d'ensemble, respectivement) se trouvant dans une boucle peuvent devenir aussi des gaspilleurs en puissance; on pourra théoriquement tenir le même raisonnement que pour let.

La récupération de l'espace mémoire occupé par les suites et ensembles générés en d'autres situations que celles qui viennent d'être citées ne pourra s'accomplir qu'avec une parfaite connaissance du programme L2 en vue de localiser ceux que l'on n'a pas besoin de garder jusqu'à la fin de celui-ci.

V. PRINCIPES DE COMPARAISON

Pour cette comparaison, il s'agira finalement de se lancer sur un exercice de "Performances et Mesures" et de définir d'une manière précise les critères de la campagne de mesures.
Pour définir ces critères, nous examinerons essentiellement trois domaines :

- l'utilisation de la mémoire,
- le temps d'exécution,
- la programmation des différentes primitives de L2.

Comme D. FISETTE ((1)) l'a déjà souligné, le reproche principal fait à ce type de langage est essentiellement sa complexité en temps pendant l'exécution. Les critères sur le temps d'exécution selon les différentes représentations seront donc nos critères majeurs,
cela revient à dire que lorsque la méthode de choix (simple ou d'analyse multicritère) que l'on proposera ne pourra pas se décider sur la seule base de ces critères, elle fera alors intervenir les autres critères (des autres domaines) de performance, c-à-d ceux de l'utilisation de la mémoire et de la programmation des différentes primitives de L2.

1. UTILISATION DE LA MEMOIRE

L'utilisation de la mémoire par les structures de représentation des suites et des ensembles doit donc, comme on vient de le constater retenir notre attention.
Il est donc utile de connaître le comportement de cette utilisation dans chacune des deux représentations;
celles-ci, par rapport à leur définition consommeront la place mémoire strictement nécessaire pour stocker les éléments des suites et des ensembles qu'elles doivent représenter.
Cette consommation nous fournit principalement trois critère :

- la consommation totale de mots de mémoire utilisés par les représentations des suites et ensembles. Ce sera le critère le plus important dans ce domaine,

- la consommation moyenne (par demande de génération) de mots de mémoire utilisés par les structures de représentation des suites et des ensembles. Ce critère sera nécessaire si notre choix ne peut s'exercer valablement sur la seule consommation totale,

- la consommation maximale de mots de mémoire alloués aux différentes demandes de génération des représentations des suites et ensembles. On se basera sur ce critère lorsque les consommations totale et moyenne ne nous permettront pas d'opter pour une représentation.

2. TEMPS D'EXECUTION

Soient P et P*, respectivement le programme L1 et le programme L2 correspondant,
si t1 est le temps d'exécution de P et
t2 est le temps d'exécution de P*,
alors $\underline{(t2 - t1) / t1}$ sera le critère le plus important dans le choix de la (des) structure(s) d'implémentation des suites et ensembles pour L2.
Lorsqu'il sera difficile d'opter pour une structure d'implémentation sur la seule base de ce critère, on se tournera alors vers le deuxième critère de ce domaine, à savoir $\underline{t2}$.

On s'attend dans la plupart des cas, vu la complexité temporelle théorique des programmes L2 testés à ce que $(t2 - t1) / t1 \gg 1$.

3. PROGRAMMATION DES DIFFERENTES PRIMITIVES DE L2

La programmation en pascal des différentes primitives dans la représentation étalée a été directe et n'a présenté aucune difficulté particulière grâce à sa structure unique et plate.
La principale amélioration que l'on y a apportée était de remplacer la récursivité utilisée dans la plupart des fonctions booléennes (comme EGAL, SUFFIXE, ...) par un bouclage et cela, lorsque la version récursive était moins efficace (en temps d'exécution) que la version par bouclage.
La traduction en pascal de ces mêmes primitives en représentation compactée est presque complètement calquée sur celle des primitives en représentation étalée. Elle était en général 2 ou 3 fois plus longue qu'en étalée
(dans certaines primitives, comme SUFFIXE et SEGMENT, la présence des 3 cas : DR, DI et DS était plutôt embêtante au point de vue amélioration et recherche d'efficacité) mais elle permettait plus de finesse et de richesse dans l'utilisation des instructions pascal (dans la primitive TRISTRCROIS par exemple, dans le cas DI, le seul élément que l'on a besoin de tester est binf;
dans la primitive PRODFOR, pour avoir le produit des éléments, dans le cas DR, il suffit de faire val**rep).
Il nous manque cependant une mesure commune pour la définition et l'appréciation des critères de choix des structures d'implémentation dans ce domaine.
Les différences de programmation n'étant pas trop énormes, nous préférons donc de ne pas définir de critère ici (qui ne manquera pas d'introduire une certaine subjectivité) et de faire plutôt parler les critères mesurables des autres domaines.

En résumé, notre choix de la (des) structure(s) de représentation des suites et des ensembles pouvant nous donner satisfaction sera exercée en fonction des 5 critères suivants :

1. temps d'exécution t2 du programme L2 (programme de base + assertions) ...CR1
2. rapport (t2 - t1) / t1 .....CR2(°)
   t1 étant le temps d'exécution du programme L1
   (programme de base sans assertions),
   t2 étant le temps d'exécution du programme L2
   (programme de base avec assertions)
3. nombre moyen de mots de mémoire par demande de génération
   (des suites et ensembles).....CR3
4. nombre total de mots utilisés par le programme L2 .....CR4
5. nombre maximum de mots de mémoire par suite ou ensemble généré .....CR5.

Le critère CR1 sera exprimé sous forme de mm:ss.cc où

mm est le nombre de minutes,

ss, le nombre de secondes et

cc, le nombre de centièmes de secondes.

Le citère CR2 sera un réel r positif (en général, r > 1).

Si l'on considère que chaque variable (variable simple, élément de tableau, pointeur, ...) prend 1 mot de mémoire de 4 octets,

alors les critères CR3, CR4 et CR5 seront exprimés en nombre de mots de mémoire de 32 bits :

- pour la représentation étalée (cfr page 16), une structure consommera un nombre de mots égal à m + 2,
- pour la représentation compactée (cfr page 32), une structure consommera
  * dans le cas "di", 4 mots de mémoire,
  * dans le cas "dr", 4 mots de mémoire et
  * dans le cas "ds", m + 2 mots de mémoire.

(°) : si t1 = 00:00.00, on posera le rapport (t2 - t1) / t1 = t2

FONCTIONS ET PROCEDURES DE PRISE DE MESURES

Selon ce qui a été signalé dans la partie concernant la Gestion Dynamique de la mémoire, les programmes testés affichent le tracé de la consommation de la mémoire en donnant à chaque moment t (abscisse du graphique) de l'allocation et de la libération la quantité totale tot (ordonné du graphique) de la mémoire utile consommée
(cfr procédure CMEM page 78.(6) et exemple de graphique page 78.(13)).

La fonction GENERER, pour la représentation étalée (cfr fonction ALLOUER page 78.(6)) et la fonction FORCOMP, pour la représentation compactée calculent le nombre (nb) de mots de mémoire utilisés pour la génération d'une suite ou d'un ensemble;
avec nmot = la taille du tableau "elem" et n = un entier ≥ 0 indiquant le nombre de fois que la primitive NEW est appelée

* pour la représentation étalée, nb = n * (nmot + 2),
* pour la représentation compactée,
  nb = n * (nmot + 2) s'il s'agit d'un cas "ds" et
  nb = 4 * n s'il s'agit d'un cas "dr" ou d'un cas "di";

le moment effectif de l'appel de la primitive NEW détermine le moment t de l'allocation.
Une fois que la représentation de la suite ou de l'ensemble est créée, ces 2 fonctions calculent finalement la quantité totale (= tot + nb) et le nombre ef des suites et ensembles déjà crées (= ef + 1).

Connaissant les expressions de génération qui laissent des structures vieillies après leur exécution (cfr gestion dynamique de la mémoire), il a été automatiquement placé dans chaque programme testé, aux endroits adéquats, une instruction de libération. La fin de la procédure de libération (cfr procédure LIBERER page 78.(6)) indiquant le moment t de la libération;
soit h le nombre de mots de mémoire libérés, on fera alors tot $\leftarrow$ tot - h.

La moyenne de nombre de mots de mémoire par suite ou ensemble généré sera, à la fin du programme L2, simplement tot / ef (cfr page 78.(11)).

Le maximum de nombre de mots par suite ou ensemble est, à tout instant, nb* = max {nb} (cfr (°) page 78.(7)).

Pour avoir le temps d'exécution d'un programme donné P, il suffit :

- de relever le temps CPU (en msec), soit T, indiqué par l'horloge CLOCK (cfr procédure INIT page 78.(3)),
- d'exécuter P et
- de relever de nouveau le temps CPU (clock), soit T*;

le temps d'exécution de P est alors (T* - T) msec (cfr page 78.(3)) que nous avons transformé sous forme de "minutes, secondes et centièmes de secondes" (cfr (°) page 78.(3)).

VI. TESTS

A. DEMARCHE

Il a fallu au départ envisager un certain nombre de programmes Pi* (Pi* = Pi $\cup$ Ai où Pi est un programme L1 et Ai, l'ensemble des assertions correspondantes).

Ces Pi* devaient être variés et les plus représentatifs possible de la classe des assertions envisagées (celles du cours de "Séminaire de programmation"),

variés et représentatifs tant du point de vue de la complexité des assertions, des structures de données utilisées - variables, tableaux, suites, ... - que du point de vue des types d'opérations qui y sont effectuées.

Ensuite pour chaque paire (Pi, Pi*), on a calculé les valeurs des différents critères pour chacune des 2 représentations.

Dès cet instant, résultats en main, nous avons considéré que nous nous trouvions devant une activité de choix multicritère, c-à-d devant un problème de classement ((5)), il faudrait en tout cas, quelle que soit la méthode employée, exercer son choix en fonction des différents critères, un choix qui ne devrait pas trop contredire les différentes "importances" attribuées aux différents critères. On a déjà fait remarquer que ces critères n'avaient pas le même poids de décision : notre activité de choix partira des critères CR2 et CR1 et ce, pour la simple raison qu'un palliatif hardware ou software peut être plus facilement trouvé pour combler le manque d'espace mémoire que pour diminuer le temps de réponse.

Avec les risques que de telles démarches empiriques ((6)) comportent, on ne pourrait se fier très fort aux résultats que si les tests sont accomplis sur un nombre beaucoup plus grand de programmes. Nous espérons qu'avec notre échantillon de programmes, la tendance de choix qui se profilera restera valable pour tout autre échantillon de programmes L2 (de type des assertions visées ici).

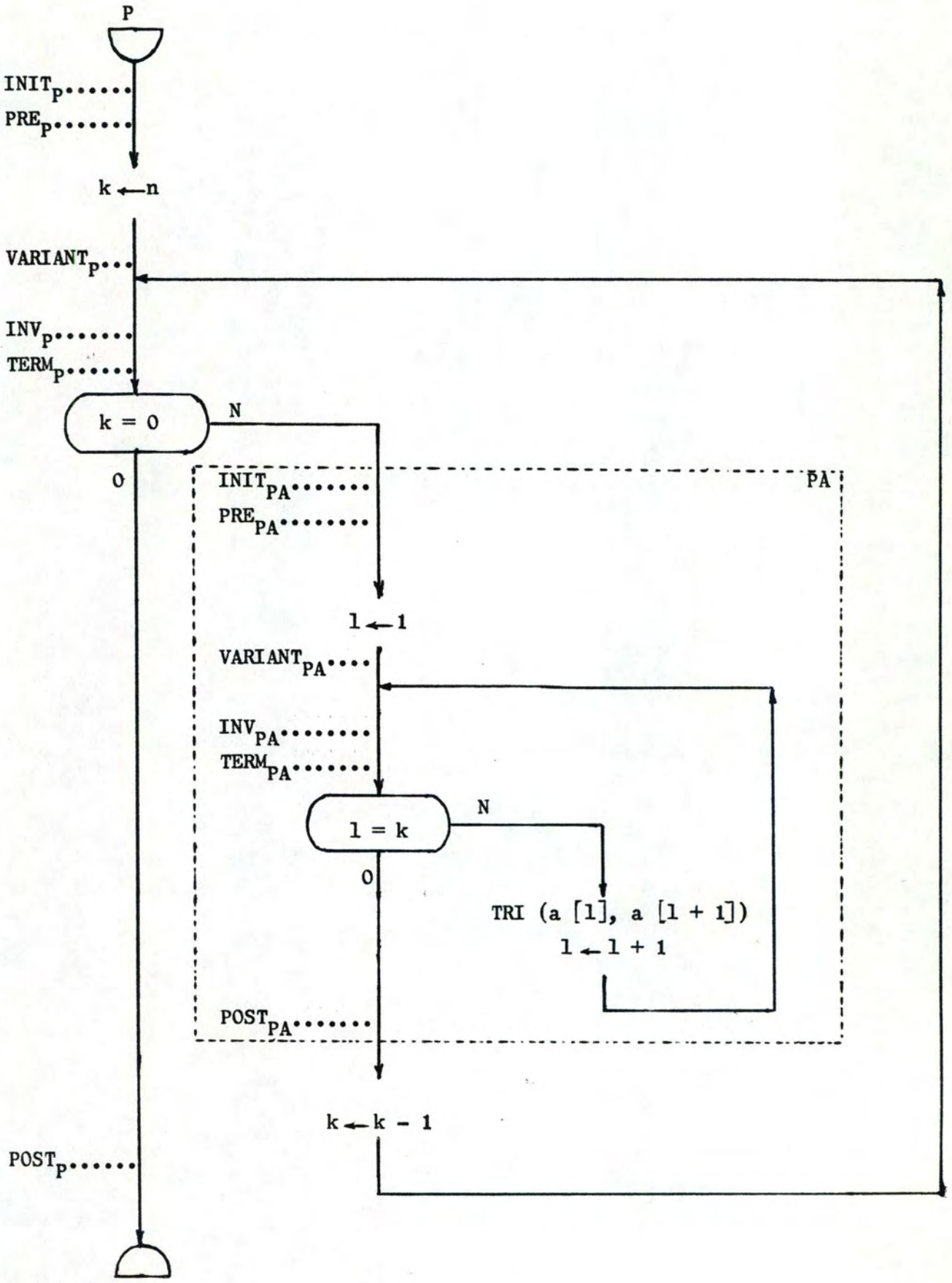
Nous gardons constamment à l'esprit qu'après avoir effectué le(s) choix, il sera toujours possible d'y apporter une certaine amélioration, surtout en temps d'exécution en essayant d'introduire d'autres structures intermédiaires de données (cfr PERMUT) dans les traductions des primitives les moins performantes, ce qui pourrait induire d'autres techniques de programmation plus efficaces.

1. Origine Des Résultats

Après avoir étudié dans tous leurs détails tous les programmes du cours de séminaire de programmation, nous avons retenu, sur base de la complexité (essentiellement, longueur de l'ensemble des assertions, difficulté de leur rédaction et, variétés de primitives L2 utilisés) pour notre campagne de mesures les programmes suivants et croyons qu'ils constituent un noyau représentatif des assertions développées dans ce cours :

(i). TRIVECT : tri simple d'un tableau a d'entiers en ordre croissant.
Nous le choisissons essentiellement pour son grand nombre varié d'assertions (cfr cours de séminaire de programmation)

Organigramme



Assertions exprimées en L2

PROGRAMME P :

- $INIT_P$: def (a);
- PRE_P : let a_o be [a];
- $VARIANT_P$: variant (x, "k", 1);
- INV_P : tricrois ([0, k, n]) and permut ([a] , a_o) and
  tricrois ([a [k+1..n]]) and
  for all i in {1..k} :
  (for all j in {k+1..n } : (a [i] ≤ a [j]));
- $TERM_P$: term (x)
- $POST_P$: permut ([a] , a_o) and tricrois ([a])

PROGRAMME PA :

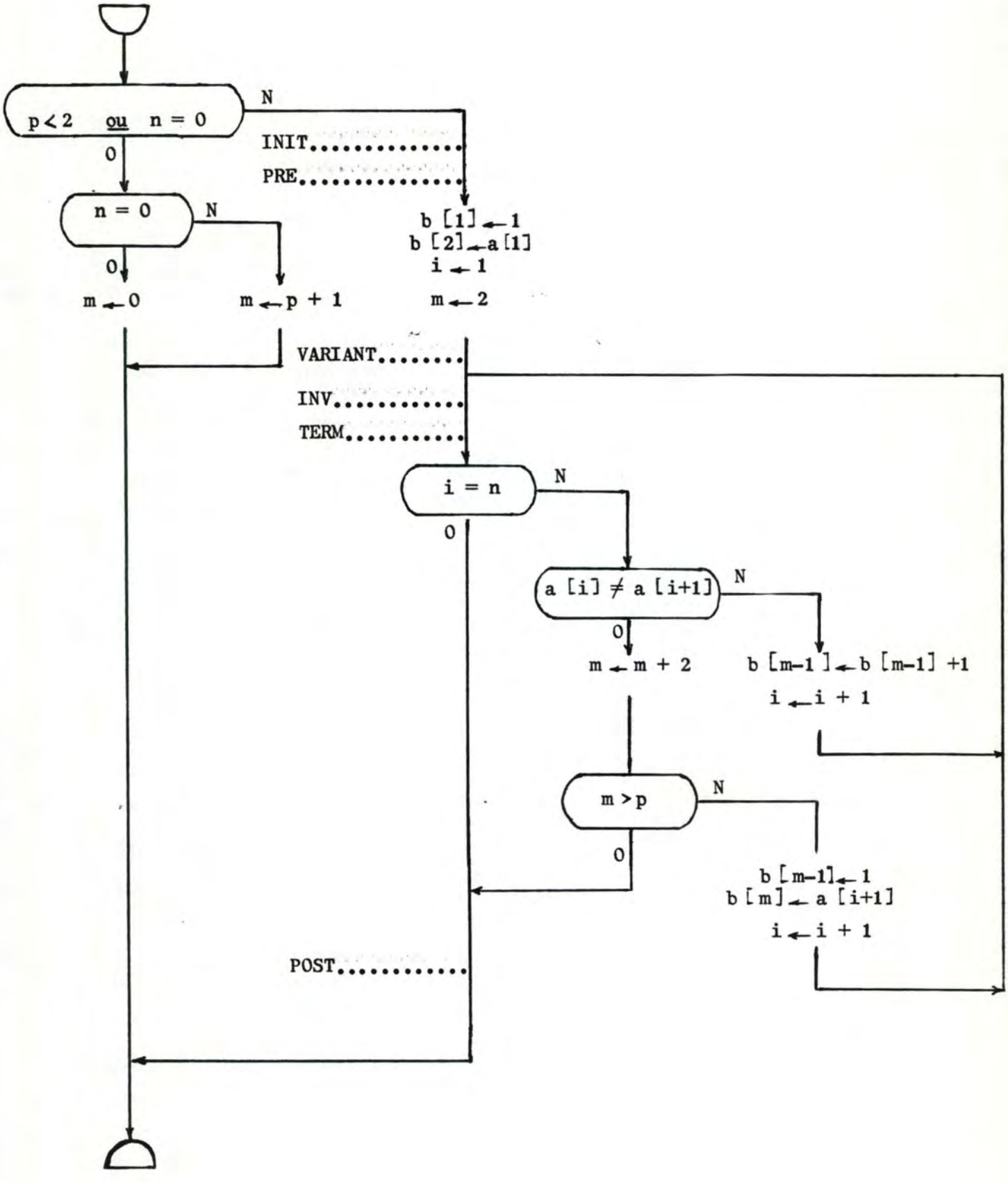
- $INIT_{PA}$: déf (a, k);
- PRE_{PA} : let a_1 be [a] ; let k_1 be k; tricrois ([1, k, n])
- $VARIANT_{PA}$: variant (y, "k-1", 1);
- INV_{PA} : tricrois ([1, 1, k]) and permut ([a] , a_1) and
  a [1] = max ({a [1..1]}) and suffixe ([a [1+1..n]] , a_1)
  and k = k_1;
- $TERM_{PA}$: term (y);
- $POST_{PA}$: permut ([a], a_1) and a [k] = max ({a [1..k]}) and
  suffixe ([a [k+1..n]], a_1) and k = k1

(ii). COMPACT : Etant donné 2 tableaux a [1:n] , b [1:p] de type entier et m, une variable entière, l'exécution du programme aura comme effet de modifier les contenus du tableau b et de la variable m de sorte que :

- m > p, si la compactée de la suite des valeurs des éléments de a possède plus de p éléments;
- $0 \leq m \leq p$, sinon; dans ce cas, la suite des valeurs de b [1..m] = compactée de la suite des valeurs de a.

Soit la suite S = (v1,v1,...,v2,...,v2,.....,
vm,....,vm),
alors la compactée de S où
v1 ≠ v2 ≠ ..... ≠ vm est la suite
C = (l1,v1,l2,v2,.....,lm,vm) où
li = longueur du segment (vi, vi, ..., vi).
(cfr examen séminaire de programmation, août 1982).
Ce programme illustre bien le principe que l'activité de vérification des assertions (pendant l'exécution) sera généralement une activité de programmation et qu'il s'agira souvent, en vue de permettre une vérification aisée de trouver des opérateurs beaucoup plus puissants que ceux dont on dispose dans la définition de L2. Ces puissants opérateurs seront en général construits sous forme des fonctions L2 à partir des expressions de base de L2 et de L1.

Organigramme



Assertions exprimées en L2

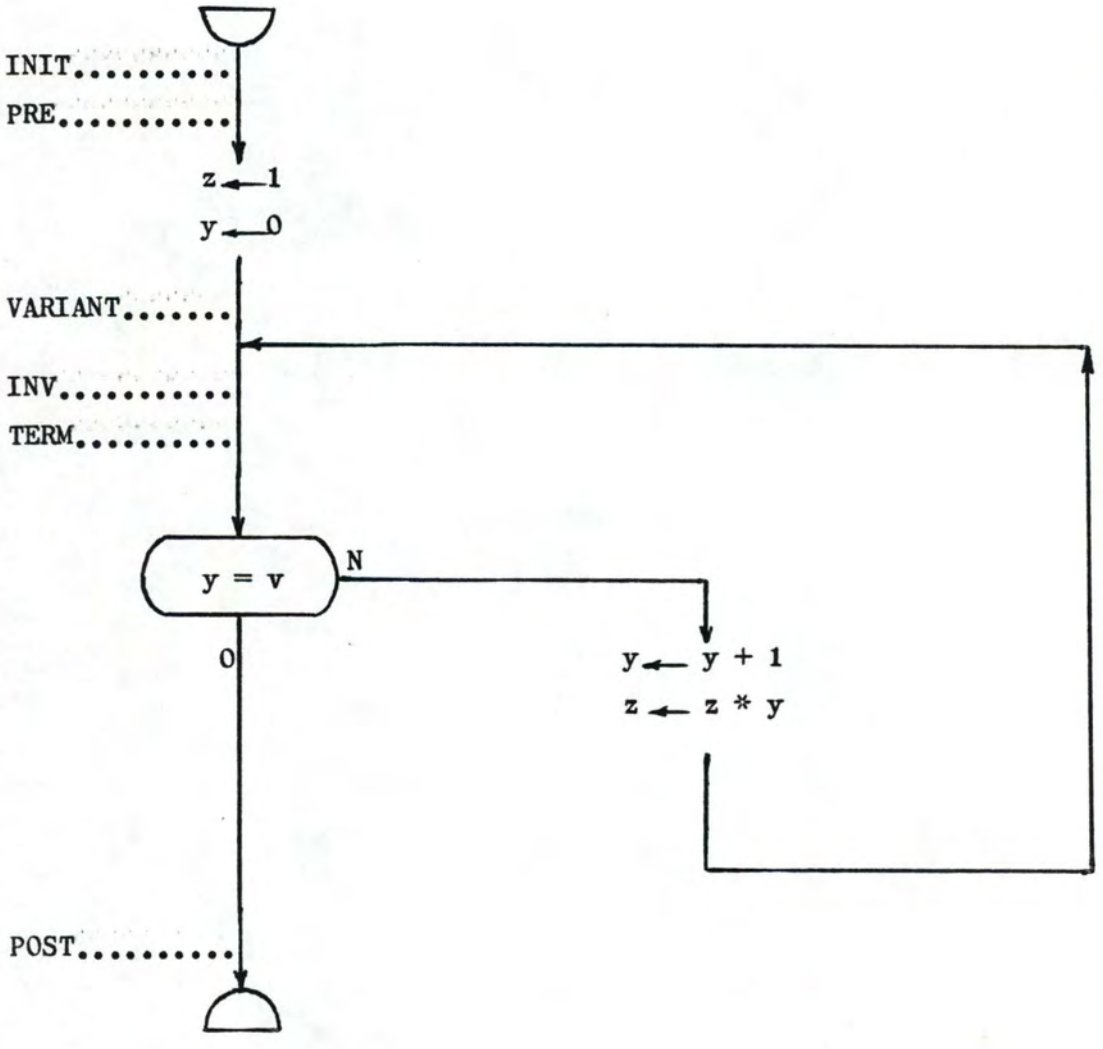
- INIT : déf (a);
- PRE : let a_0 be [a] ; $n > 0$; $p \geq 2$;
- VARIANT : variant (boucle, "n-i", 1);
- INV : $1 \leq i \leq n$ and $1 \leq m \leq p$ and
  égal ([b [1..m]], [compact (a [1..i])]);
- TERM : term (boucle);
- POST : ((égal (a_0, [a])) and
  (((m = p + 1) and (long (compact (a [1..n])) > p)) or
  ((m ≤ p) and (égal ([b [1..m]] , [compact (a [1..n])])))));

où compact (a [i..j]) est une fonction L2 qui renvoie la compactée de la suite des éléments a [i] , a [i+1] , ... , a [j] et qui pourrait être définie comme suit :

```
fonction COMPACT ( a, i, j) : ^ensuite;
    début
    c [1] ← 1
    c [2] ← a [i]
    m ← 2
    tant que i ≠ j faire | si a [i] = a [i + 1]
                         |    alors | c [m - 1] ← c [m - 1] + 1
                         |           | i ← i + 1
                         |    sinon | m ← m + 2
                         |          | c [m - 1] ← 1
                         |          | c [m] ← a  i + 1
                         |          | i ← i + 1
    compact ← FORCOMP ('t', c, 1, m) ou GENERER (c, 1, m)
    fin
```

(iii). FACT : renvoi de la valeur v ! étant donné une valeur entière v positive.
Ce programme (fonction) illustre l'utilisation d'un opérateur itératif.

Organigramme



Assertions traduites en L2

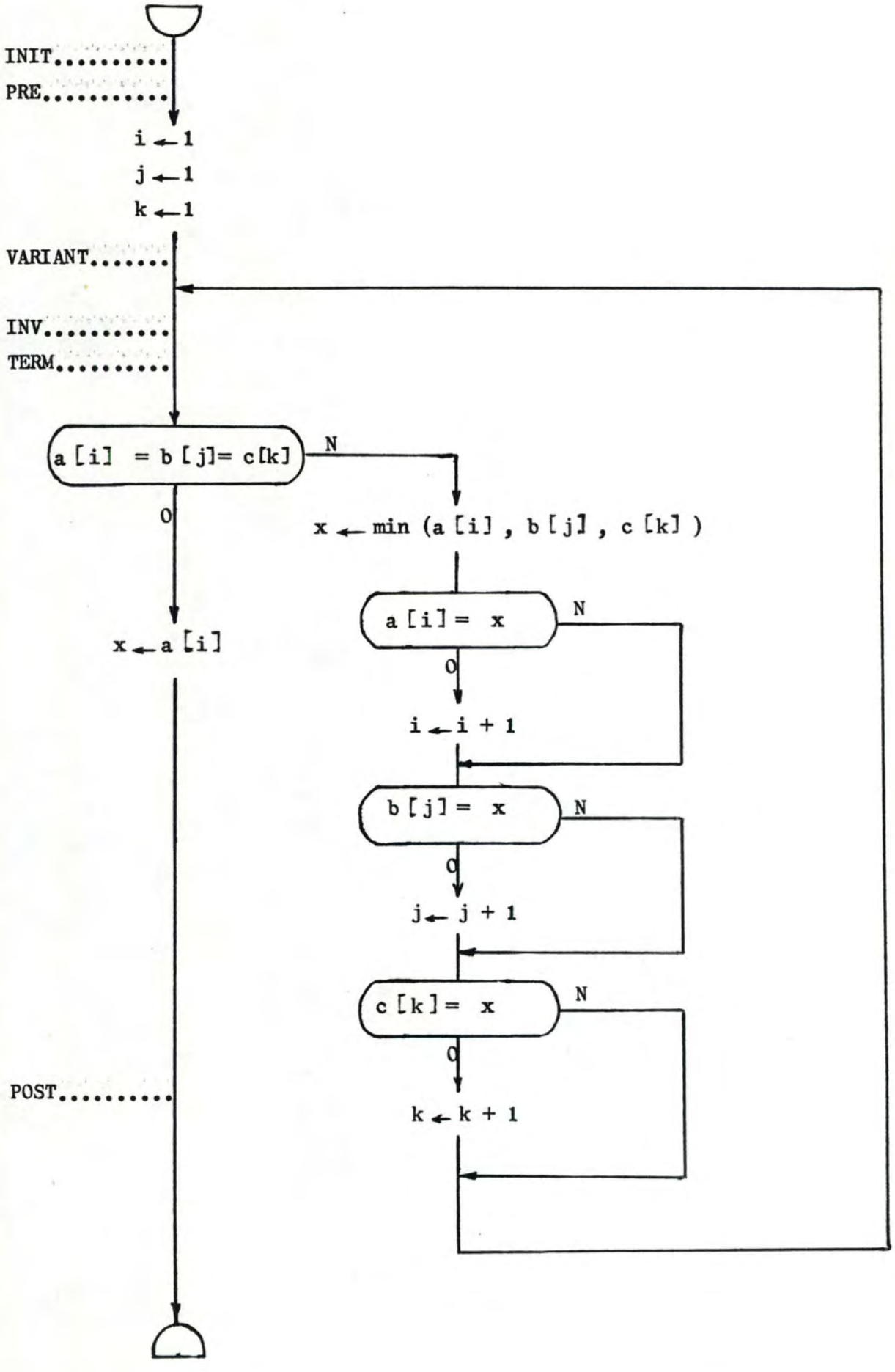
- INIT : déf (v);
- PRE : let v_0 be v; v ⩾ 0;
- VARIANT : variant (boucle, "v - y", 1);
- INV : $v = v_0$ and tricrois ([0, y, v]) and z = fact2 (y);
- TERM : term (boucle);
- POST : $v = v_0$ and z = fact2 (v);

où fact2 (v) est une fonction L2 qui pourrait être définie de la façon suivante :

```
fonction FACT2 (v) : entier;
    | début
    | fact2 ← prod for i in [ 1..v] of (i)
    | fin
```

(iv). COMMUNE : affectation à la variable entière x du plus petit élément commun à 3 tableaux d'entiers a, b et c (de tailles respectives m, n et p) en ordre strictement croissant.
Nous choisissons ce programme à cause de sa quantité variée d'assertions et de son input (3 tableaux).

Organigramme

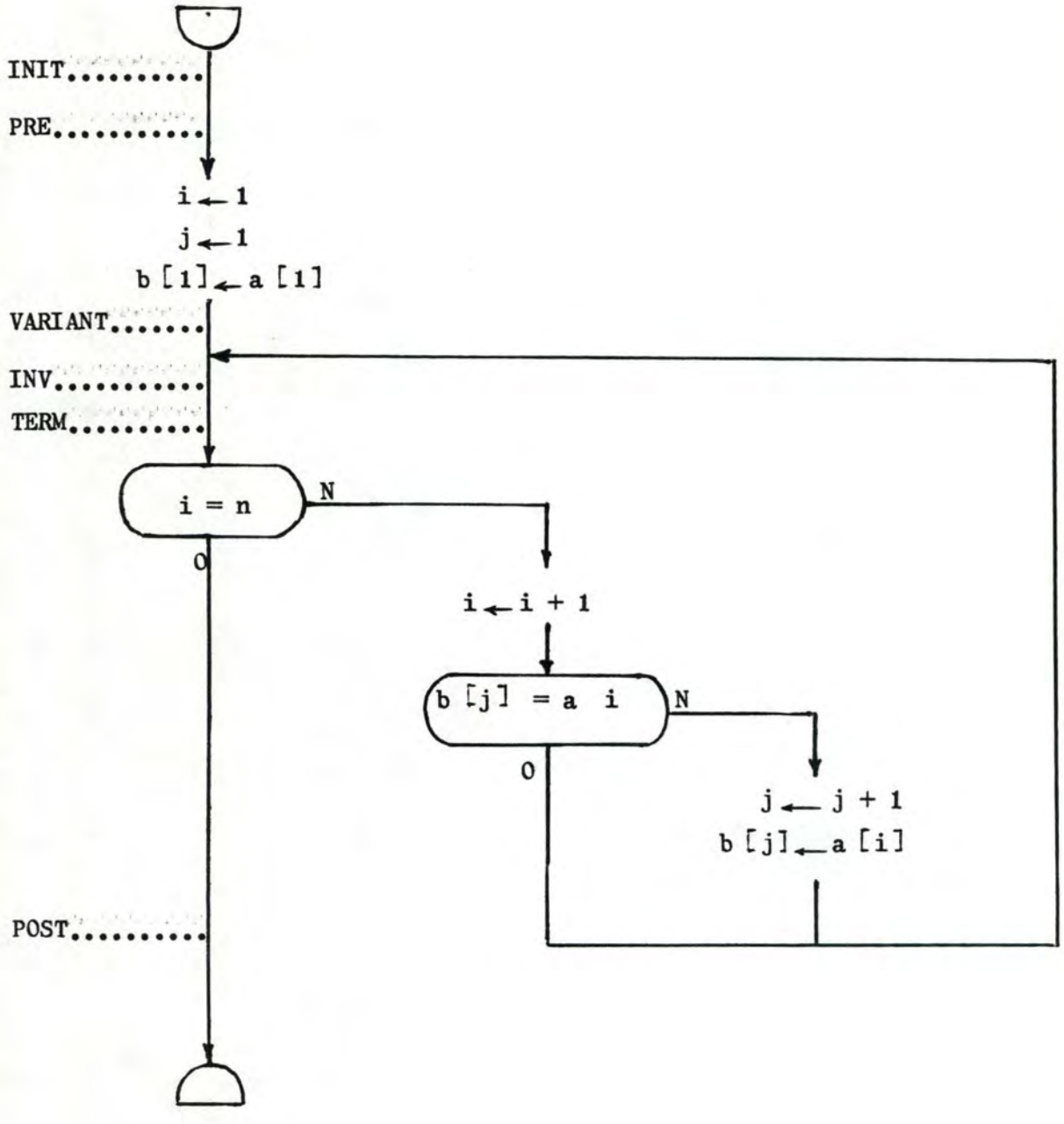


Assertions traduites en L2

- INIT : déf (a, b, c);
- PRE : let a_o be {a} ; let b_o be {b} ; let c_o be {c} ;
  tristrcrois ([a]); tristrcrois ([b]); tristrcrois ([c]);
  inter (a_o, inter (b_o, c_o)) ≠ ∅;
- VARIANT : variant (boucle, "m + n + p - (i + j + k)", 1);
- INV : 1 ≤ i ≤ m and 1 ≤ j ≤ n and 1 ≤ k ≤ p and
  inter ({a [1..i-1]}, inter ({b [1..j-1]}, {c [1..k-1]})) = ∅
  and
  max ({a [1..i-1] , b [1..j-1] , c [1. .k-1] })
  min ({a [i] , b [j] , c [k] })
- TERM : term (boucle);
- POST : x = min (inter (a_o , inter (b_o , c_o));

(v). CONTRACT : soit un tableau a 1:n d'entiers en ordre croissant, il s'agit de modifier le contenu du tableau b [1:n] de telle manière que b 1..j ($1 \leq j \leq n$) soit trié en ordre strictement croissant et contienne, sans répétitions, la suite des valeurs a [1] , a [2] , ..., a [n] .

Organigramme

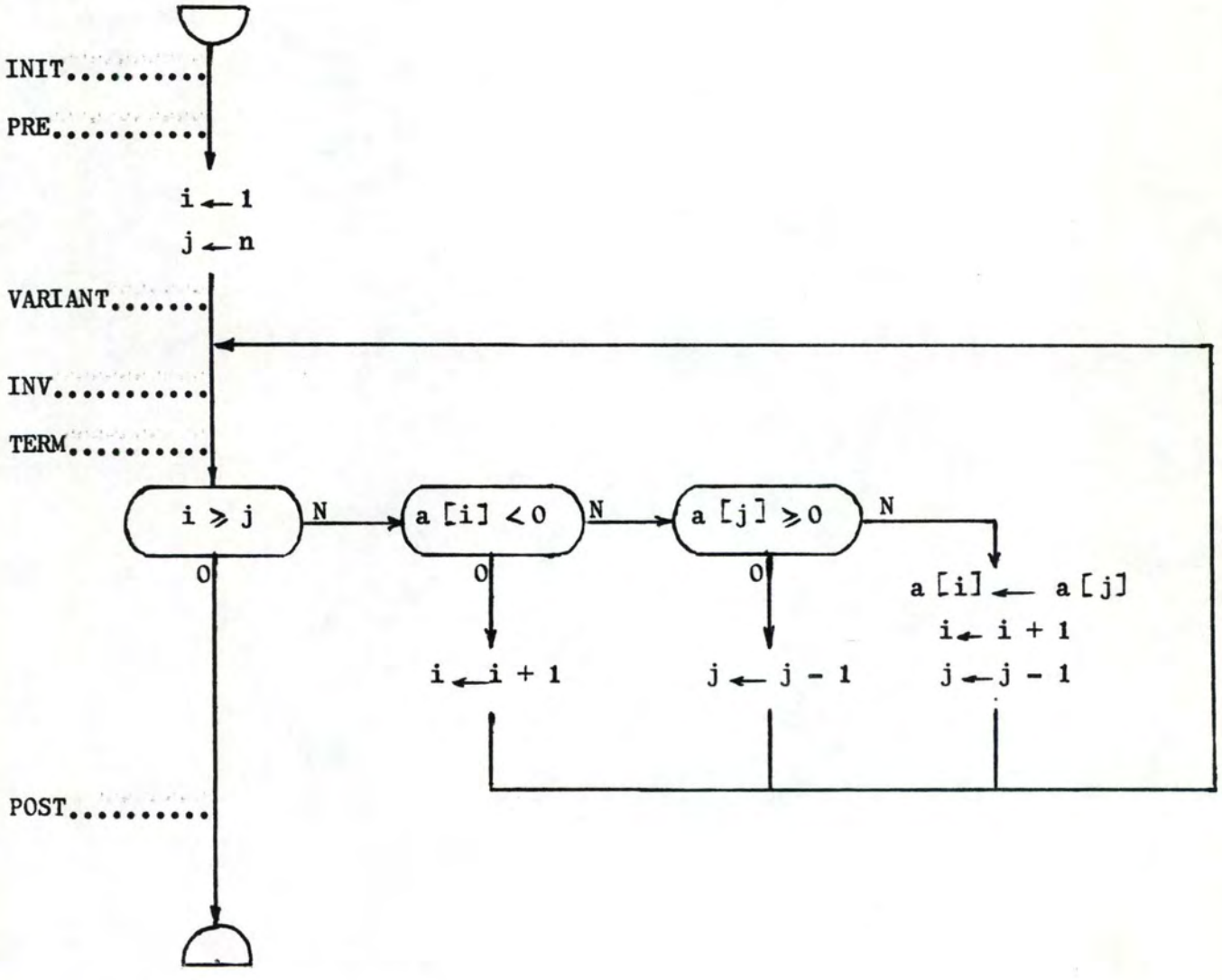


Assertions traduites en L2

- INIT : déf (a);
- PRE : let a_o be [a] ; $n \geq 1$;
- VARIANT : variant (boucle, "n - i", 1);
- INV : $1 \leq j \leq i \leq n$ and tristrcrois ([b [1..j]]) and
  égal ({b [1..j]} , {a [1..i]}) and
  égal ([a] , a_o);
- TERM : term (boucle);
- POST : $1 \leq j \leq n$ and tristrcrois ([b [1..j]]) and
  égal ({b [1..j]} , {a}) and
  égal ([a] , a_o);

(vi). PERMUTAT : permutation d'un tabeau a [1:n] d'entiers de telle façon que ses éléments <0 soient à gauche et ses éléments $\geqslant 0$ soient à droite.

Organigramme



Assertions traduites en L2

- INIT : déf (a);
- PRE : let a_o be [a];
- VARIANT : variant (boucle, "j - i + 1", 1);
- INV : tricrois ([1, i, j + 1, n + 1]) and
  permut ([a], a_o) and
  for all k in {1..i-1} : (a [k] < 0) and
  for all k in {j+1..n} : (a [k] ⩾ 0);
- TERM : term (boucle);
- POST : permut ([a], a_o) and
  there exists q in {0..n} :
  (for all k in {1..q} : (a [k] < 0) and
  (for all k in {q+1..n} : (a [k] ⩾ 0));

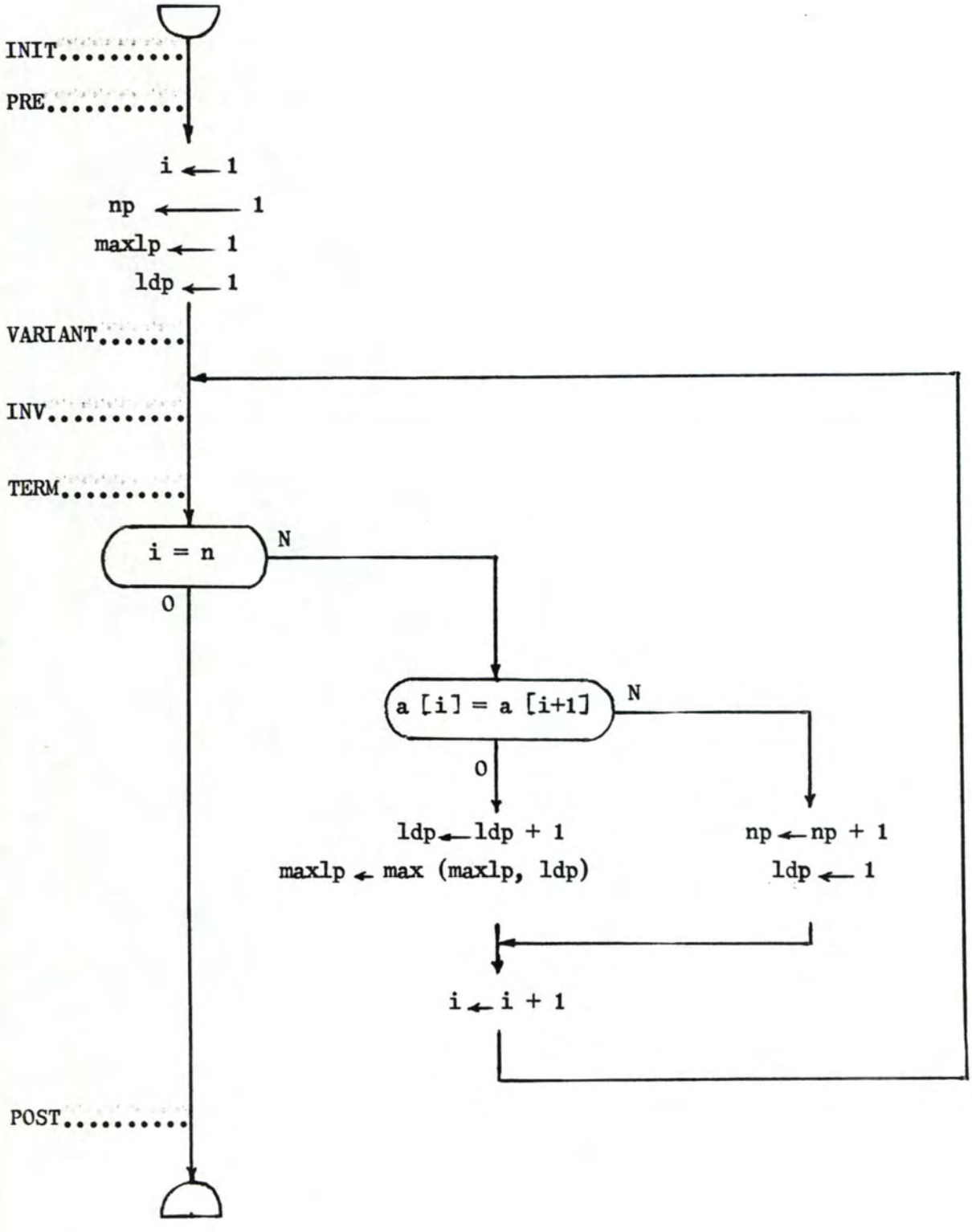
(vii). PLATEAUX : manipulation des plateaux d'un tableau a [1:n] d'entiers et
affectation :
- du nombre de plateaux de a [1:n] à la variable np,
- de la longueur du plus grand plateau de a [1:n] à la variable maxlp

sans modifier le tableau a [1:n]

Un plateau est un intervalle $[i:j] \subseteq [1:n]$
tel que :
- $i \leq j$,
- $a[i] = a[i+1] = \ldots\ldots\ldots = a[j]$,
- $\nexists$ intervalle $[i_o:j_o]$ tel que
  $[i:j] \subset [i_o:j_o]$ ($[i:j] \neq [i_o:j_o]$) et
  $a[i_o] = a[i_o+1] = \ldots\ldots\ldots\ldots = a[j_o]$.

En vue de rendre leur vérification plus aisée, ce programme n'utilise quasiment que des assertions de haut niveau construites au moyen des fonctions L2 utilisant des primitives disponibles dans les langages L1 et L2.

Organigramme



Assertions traduites en L2

- INIT : déf (a);
- PRE : let a_0 be [a] ; n > 0;
- VARIANT : variant (boucle, "n - i", 1);
- INV : tricrois ([1, i, n]) and np = nbplat (a, i) and
  maxlp = maxplat (a, i) and ldp = ldplat (a, i)
- TERM : term (boucle);
- POST : np = nbplat (a, n) and
  maxlp = maxplat (a, n) and égal ([a] , a_0)

où nbplat (a, i), maxplat (a, i) et ldplat (a, i) sont des fonctions respectivement définies comme suit :

```
fonction NBPLAT (a, i) : entier;
    | début
    | nbplat ← card ( {j in {1..i} | j ≠ 1 ⟹ a [j-1] ≠ a [j]} )
    | fin,
```

calcul du nombre de plateaux de a [1..i]

```
fonction LDPLAT (a, i) : entier;
    | début
    | ens ← {j in {1..i} | j ≠ 1 ⟹ a [j-1] ≠ a [j]}
    | maxplat    max for j in ens of
                 (max ({k in {j..i} |
                  for all l in {j..k} : (a [1] = a [j] )}) - j + 1)
    fin
```

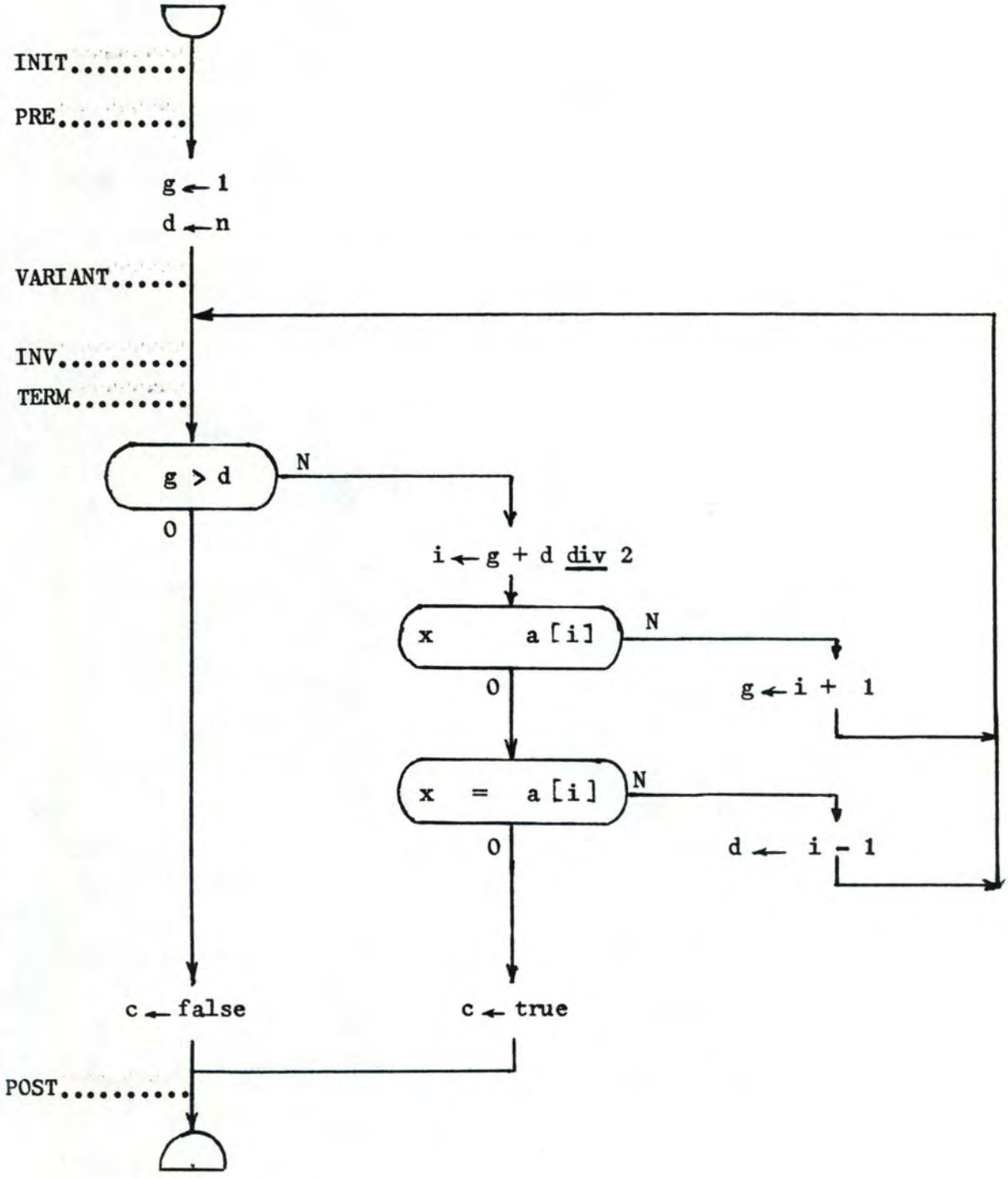
calcul de la longueur du plus grand plateau de a [1..i]

```
fonction LDPLAT (a, i) : entier;
    | début
    | ldplat ← i - max ( {j in {1..i} | j ≠ 1 ⟹ a [j-1] ≠ a [j]}) + 1
    | fin
```

calcul de la longueur du dernier plateau de a 1..i

(viii). DYCHO : étant donné un tableau a [1:n] d'entiers trié en ordre croissant et x, un entier qui est la clé de recherche; il s'agit de faire une recherche dychotomique de x dans a et d'affecter à la variable c : - la valeur "true" si x ∈ a [1:n] ou
- la valeur "false" sinon.

Organigramme



Assertions traduites en L2

- INIT : déf (a);
- PRE : let a_o be [a] ; tristrcrois (a_o);
- VARIANT : variant (boucle, "d - g + 1", 1);
- INV : $1 \leq g \leq n + 1$ and $0 \leq d \leq n$ and
  for all j in {1..g-1} : (a [j] < x) and
  for all j in {d+1..n} : (a [j] > x);
- TERM : term (boucle);
- POST : égal (a_o, [a]) and c = inclus ({x}, {a});

2. Quelques Exemples de Programmes Traduits En Pascal

REMARQUES GENERALES :

a. Dans un environnement où un interpréteur de L2 existe, ce qui suppose que l'on ait déjà analysé syntaxiquement et sémantiquement le langage L2, la traduction en pascal des différents programmes testés ne causerait aucun problème.
Dans notre environnement où un tel interpréteur est absent, le problème principal était celui de la diversité des formes de paramètres associés à une même fonction (ou à une même procédure) au sein d'un programme déterminé.
Dans ce cas, nous avons simplement considéré qu'une telle fonction (ou procédure) n'a qu'un seul paramètre, et qui est une chaîne de caractères; nous avons alors défini une procédure d'analyse (locale) de la chaîne.
Comme il était IMPORTANT pour nous de CONSERVER dans la programmation en pascal l'écriture "NATURELLE" des paramètres des primitives des différentes assertions dans la traduction L2, cette manoeuvre a souvent été nécessaire, sinon il aurait été long et difficile à exprimer autrement.

A part ce problème de la diversité des formes des paramètres, nous ne croyons pas avoir eu une difficulté quelconque dans cette traduction :
chaque programme étant littéralement formé d'un ensemble des pseudo-algorithmes adéquats.
Comme ces pseudo-algorithmes sont dans un langage proche de pascal, il n'y a vraiment eu aucun autre problème;
ainsi, nous espérons avoir mis dans les explications des programmes qui suivent tout raisonnement qui ne nous paraissait pas clair. Ces programmes ont été choisis arbitrairement.

b. Pour la traduction de la pseudo-instruction LET et de la primitive DEF, nous avons défini une fonction booléenne unique (DEFLETT pour les tableaux et DEFLETV pour les variables simples), ce faisant, nous évitons de définir deux fonctions différentes, surtout que DEF est nécessairement suivi de LET :

- pour un tableau a de taille n, DEFLETT (a, n, a_o) est vrai et aura crée la suite ou l'ensemble a_o si tous les éléments de a ont été initialisés; il est faux et n'aura crée ni suite ni ensemble, sinon;
- pour un élément simple quelconque k de type entier, caractère ou booléen, DEFLETV (k, k_1) est vrai et aura déclaré la variable k_1 initialisée à la valeur de k, si celui-ci a été initialisé; il est faux et n'aura déclaré aucune variable sinon.

c. La terminaison des programmes à l'aide de la fonction TERM et de la pseudo-instruction VARIANT s'accomplit comme suit (k étant la valeur de la fonction de terminaison avant d'entrer dans la boucle et l, la quantité dont doit décroître k à chaque passage dans la boucle) :

- la procédure VARIANT (k, l :entier; var exp1, exp2 : entier)
  * initialisera exp1 à k + l et
  * initialisera exp2 à l;
- la fonction TERM (k : entier; var exp1, exp2 : entier) : booléen vérifiera :
  si k + exp1 > exp1, auquel cas, elle renvoit "faux" car la valeur actuelle k de la fonction de terminaison est au moins égale à la valeur précédente exp1 de la même fonction;
  si k + exp2 $\leq$ exp1, auquel cas elle renvoit "vrai" car la valeur actuelle k de la fonction de

terminaison est différente de la valeur
précédente exp1 de la même fonction
au moins de "exp2" unités;
dans ce cas, elle assignera à exp1 la
valeur de k.

Remarquons que la valeur k de la fonction de terminaison est toujours connue avant l'entrée de la boucle; donc dans VARIANT, au lieu d'initialiser exp1 à zmax comme le préconise D. FISETTE, il suffit de l'initialiser à k + la quantité dont décroitra k à chaque passage dans la boucle.

```
DIANE>
DIANE>
DIANE>type time.pas
[environment('time.env')]
module time(output) ;

type    type_date = packed array [1..11] of char;
                { exemple :'14:20:25.98' }
var     start_date: type_date ;
                { temps reel quand le programmme demarre }
        start_time:integer;
                { en millisecondes                       }
                { temps consomme depuis le login quand   }
                { on demarre le programme                }
{-------------------------------------------------------------------------}
function CONVERT (a_date:type_date):integer ;
                { convertit une date a la quantite cor-  }
                { respondante en millisecondes           }
var     t:integer;
begin
        t:=ord(a_date[11]) - 48 ;
        t:=t + (ord(a_date[10])- 48) * 10;
        t:=t + (ord(a_date[8]) - 48) * 100;
        t:=t + (ord(a_date[7]) - 48) * 1000;
        t:=t + (ord(a_date[5]) - 48) * 6000;
        t:=t + (ord(a_date[4]) - 48) * 60000;
        t:=t + (ord(a_date[2]) - 48) * 360000;
        t:=t + (ord(a_date[1]) - 48) * 3600000;
        convert := t * 10 ;
end;
{-------------------------------------------------------------------------}
[global]procedure INIT_TIME;
                { memorisation du temps reel et du       }
                { temps cpu quand le programme demarre   }
begin
        TIME (start_date);
        start_time:=clock
end;
{-------------------------------------------------------------------------}
[global]procedure SHOW_TIME;
                { ecrit le temps cpu consomme par le     }
                { processus courant et le temps reel     }
                { (elapsed time) ecoule depuis le begin  }
                { du processus                           }.
var     cpu_time:integer;
                { temps cpu consomme par le processus    }
                { courant en millisecondes               }
        elapsed_time: integer;
                { temps reel ecoule depuis  start_date   }
        now_date:type_date;
                { temps reel maintenant                  }
begin
        cpu_time:=clock - start_time ;
        TIME (now_date);
        elapsed_time :=CONVERT (now_date) -CONVERT (start_date);
        write('le temps cpu est : ');
        write(cpu_time div 60000 :2,':');
        if (cpu_time mod 60000)/1000 < 10 then write('0');
        write((cpu_time mod 60000)/1000  :2:2,' consomme en ');
        write(elapsed_time div 60000 :2,':');
        if (elapsed_time mod 60000)/1000 < 10 then write('0');
        write((elapsed_time mod 60000)/1000 :2:2)
                { le  Load etant egal a                  }
                {                  elapsed_time/cpu_time }
end;
end.
```

(°)

EXPLICATION DU PROGRAMME COMMUNE (ETALEE)

- l'instruction TRISTRCROIS (' [x] ') vérifie donc si la suite que l'on peut générer à partir du tableau x est en tri strictement croissant.
  On aurait pu définir cette fonction comme :
  fonction TRISTRCROIS (c1 : char; var t : tableau; c2 : char):booléen,
  auquel cas, l'appel se ferait comme suit :
  TRISTRCOIS ('[', a, ']') ou
  fonction TRISTRCROIS (var t : tableau; n : entier) : booléen;
  (n étant la taille du tableau t) auquel cas, l'appel se ferait comme suit :
  TRISTRCROIS (a, n).
  Dans le 1er cas, nous trouvons que l'appel n'est pas commode pour l'utilisateur.
  Dans tous les cas, on s'éloigne de l'écriture utilisée dans la traduction des assertions en L2.
  On a alors eu recours à la manoeuvre citée dans les remarques générales. Ce qui a permis une notation plus claire et plus proche de la traduction des assertions en L2.

- Dans la fonction de type pointeur INTER (' $\{e_o\}$, $\{e_1\}$, $\{e_2\}$ ')
  où e_o, e_1 et e_2 sont des expressions d'ensembles,
  une autre fonction de type pointeur INTERSEC (p, q) est déclarée (p et q étant des pointeurs sur des représentations des ensembles).
  Soient r le pointeur de la représentation de l'ensemble issu de l'expression e_o,
  s le pointeur de la représentation de l'ensemble issu de l'expression e_1,
  t le pointeur de la représentation de l'ensemble issu de l'expression e_2
  alors, INTERSEC (r, INTERSEC (s, t)) est le pointeur de la représentation de l'ensemble $e_o \cap e_1 \cap e_2$.
  Nous pouvons voir que INTER ('$\{e_o\}$, $\{e_1\}$, $\{e_2\}$ ') ne sert qu'à analyser la chaine de caractères qui constitue son argument, qui peut être de deux formes différentes,

et à passer à la fonction INTERSEC des arguments de type pointeur issus des résultats de l'analyse de la chaîne.
Nous pensons qu'il aurait été compliqué de vouloir respecter la notation utilisée dans la traduction des assertions en L2 et de définir directement une fonction INTERSEC à deux arguments de type pointeur.

- La fonction MIN accepte plusieurs formes d'arguments, alors l'analyse dans le cas MIN (' {a [i] , b [j] , c [k] }'),
  * reconnaîtra la chaîne,
  * formera un tableau avec les éléments a [i], b [j] , c [k],
  * triera ce tableau,
  * éliminera toute copie de ce tableau et
  * générera l'ensemble demandé (dont le pointeur est p) à partir de ce tableau;

  la fonction MIN proprement dite n'aura plus qu'à trouver le minimum de la représentation de l'ensemble dont le pointeur est p,

  dans le cas MIN ('INTER (a_o, b_o, c_o) '),
  * reconnaîtra la chaîne,
  * fera appel à la primitive INTER;

  soit p, le pointeur de la représentation de l'ensemble intersection, la fonction MIN n'aura plus qu'à trouver le minimum de la représentation dont le pointeur est p.

```
DIANE>type ecommune.pas
[INHERIT('TIME.ENV')]
program COMMUNE  (input, output);
{                                                                              }
{       donne la plus petite valeur commune a 3 vecteurs en etalee             }
{                                                                              }
label 1, 2, 3, 5, 6, 7, 10, 40;
const nmax=60; tmax=1000; ch=37; zmin=-999999999; zmax=999999999;
      noval=999999999;
type tableau = array [1..nmax] of integer;
     packar = packed array [1..ch] of char;
     vers = ^ensuite;
     ensuite = record
                 comp : integer;
                 elem : array [1..tmax] of integer;
                 suiv : vers
                 end;
var a, b, c : tableau;
    nmot, nb, ef, nal, addit, qmax, aexp1, aexp2, aexp3, aexp4, x,
    i, j, k, l, m, n, p : integer;
    struct, a0, b0, c0, lp, lq : vers;
{------------------------------------------------------------------------------}
procedure CMEM (nb : integer);
var i, v : integer;
begin
writeln; write('*'); v:=nb div 3;
if v>0 then begin for i:=1 to v do write(' '); write('*',' ') end
end;
{------------------------------------------------------------------------------}
procedure ALLOUER (var nb : integer);
begin
new (struct);
nb:=nb+addit; nal:=nal+addit;
CMEM(nb)
end;
{------------------------------------------------------------------------------}
procedure LIBERER(debut : vers; var nb : integer);
var i : integer; deb : vers;
begin
i:=0;
while debut <> nil do begin
                      i:=i+1; deb:=debut;
                      debut:=debut^.suiv; dispose(deb)
                      end;
if i>0 then begin
            nb := nb-(i*addit);
            CMEM(nb)
            end;
end;
{------------------------------------------------------------------------------}
procedure LIRE (var a, b, c : tableau);
var i : integer;
begin
lp:=nil; lq:=nil;
m:=10; n:=10; p:=10;
for i:=1 to m do if i=10 then a[i]:=20 else a[i]:=i;
for i:=1 to n do b[i]:=i*2;
for i:=1 to p do if i=10 then c[i]:=20 else c[i]:=i*2+1;
repeat
   write('donnez la taille des suites et ensembles '); readln(nmot); writeln;
   if nmot > tmax then writeln('taille trop grande.....')
```

```
until  nmot <= tmax;
addit:=nmot+2; writeln;
writeln('en ordonne, * egale environ 3 mots memoire');   writeln;
for i:=1 to 75 do write('*');
nb:=0; ef:=0; nal:=0; qmax:=0
end;
{-------------------------------------------------------------------------}
procedure ECRIRE;
var i : integer;
begin
LIBERER (a0,nb);
LIBERER (b0,nb);
LIBERER (c0,nb);
LIBERER(lp,nb); LIBERER(lq,nb);
writeln; writeln; writeln;
writeln ('l''element minimum commun (en etalee) aux 3 vecteurs est : ',x)
end;
{-------------------------------------------------------------------------}
function GENERER (exp : char; var t : tableau; g, d : integer) : vers;
var j, qm : integer; r, lien : vers;
begin
r:=nil; qm:=0;
if g <= d then begin
               ALLOUER(nb); qm:=qm+addit; r:=struct; lien:=struct;
               repeat j:=1;
                      while (j<=nmot) and (g<=d) do
                            begin
                            if exp='i' then struct^.elem[j]:=g
                                       else struct^.elem[j]:=t[g];
                            g:=g+1; j:=j+1
                            end;
                      struct^.comp:=j-1;
                      if g<=d then begin
                                   ALLOUER(nb); qm:=qm+addit;
                                   lien^.suiv:=struct;
                                   lien:=struct
                                   end
                              else struct^.suiv := nil
               until g>d
               end;
generer:=r;
if r <> nil then begin ef:=ef + 1; if qm>qmax then qmax:=qm end (°)
end;
{-------------------------------------------------------------------------}
function CHIFFRE (chn : packar; var v : integer) : integer;
var chif : integer;
begin
chif := 0;
while (v<=ch) and (chn[v]<>' ') and (chn[v]<>',') and (chn[v]<>']') and
      (chn[v]<>'.') and (chn[v]<>'}')
      do begin
         chif:=(chif*10) + (ord(chn[v]) - 48);
         v:=v+1
         end;
chiffre := chif
end;
{-------------------------------------------------------------------------}
procedure PARSEUR (chn : packar; var v,w : integer; var u : tableau);
var ip : integer;
begin
while (v<=ch) and (chn[v]<>']') and (chn[v]<>' ') and (chn[v]<>'}')
      do case chn[v] of
         '0','1','2','3','4','5','6','7','8','9' : begin
                                                    w := w + 1;
                                                    u[w] := CHIFFRE(chn, v)
                                                    end;
```

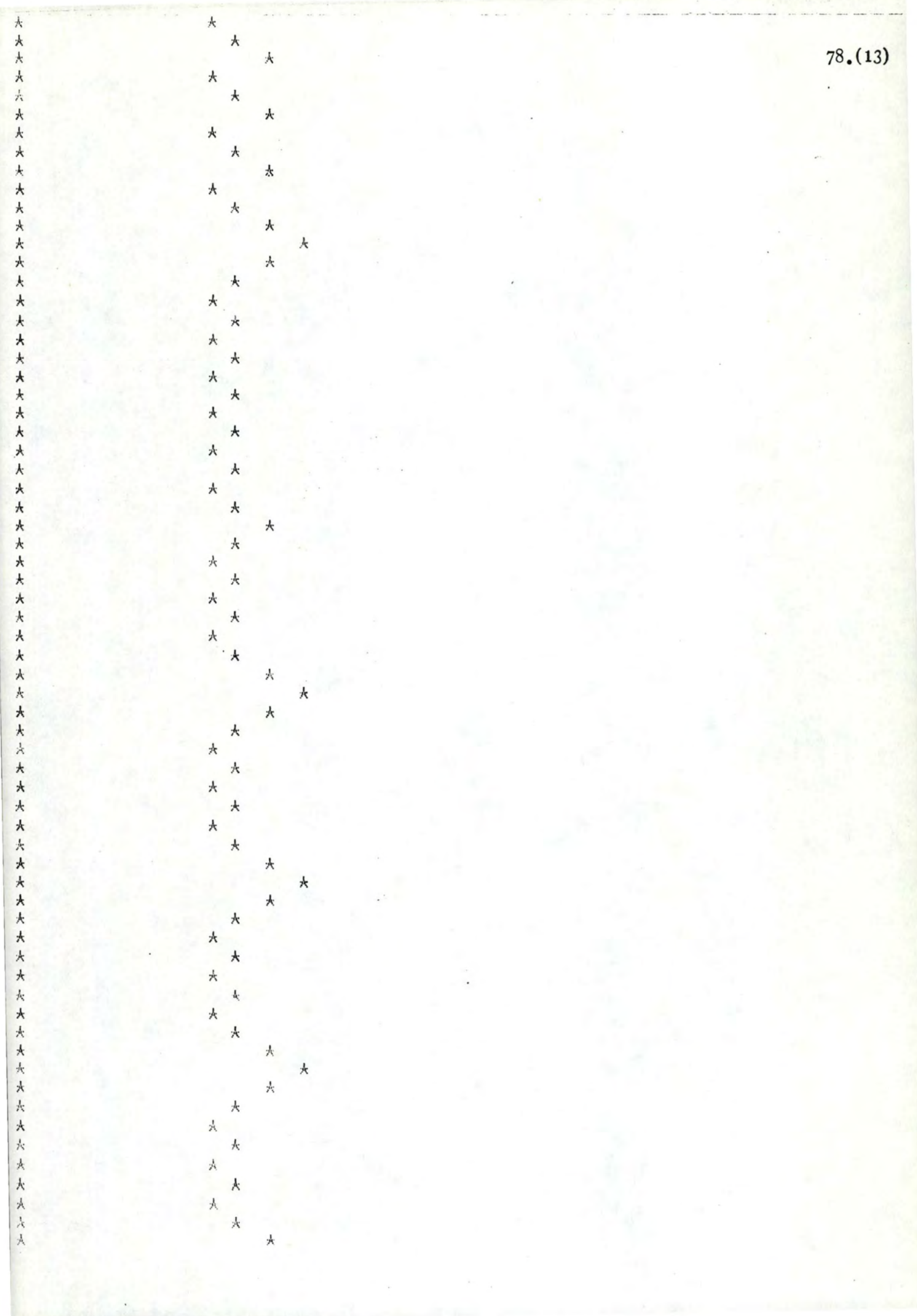
```
        'i','j','k','l','m','n','p' : begin
                                      w:=w+1;
                                      case chn[v] of
                                      'i': ip := i;
                                      'j': ip := j;
                                      'k': ip := k;
                                      'l': ip := l;
                                      'm': ip := m;
                                      'n': ip := n;
                                      'p': ip := p
                                      end;
                                      v:=v+1;
                                      if (chn[v]='+') or (chn[v]='-')
                                         then case chn[v] of
                                              '+' : begin
                                                    v:=v+1;
                                                    u[w]:=ip + CHIFFRE (chn,v
                                                    end;
                                              '-' : begin
                                                    v:=v+1;
                                                    u[w]:=ip - CHIFFRE (chn,v
                                                    end
                                              end
                                         else u[w]:=ip
                                      end;
        '.',',': v:=v+1
        end
end;
{-------------------------------------------------------------------------------}
function DEFLETT (var a : tableau; n : integer; var debut : vers) : boolean;
var i : integer; def : boolean;
begin
def:=true;
i:=1;
while (i<=n) and (def) do if a[i]=noval then def:=false
                                        else i:=i+1;
if def then begin deflett:=true; debut := GENERER ('t',a,1,n) end
       else begin deflett:=false; debut:= nil end
end;
{-------------------------------------------------------------------------------}
procedure VARIANT(k, l : integer; var exp1, exp2 : integer);
begin exp1:=k + 1; exp2:=1 end;
{-------------------------------------------------------------------------------}
function TERM(k : integer; var exp1, exp2 : integer) : boolean;
begin
if k+exp2 > exp1 then term:=false else begin exp1:=k; term:=true end
end;
{-------------------------------------------------------------------------------}
procedure ELIMINER(var t : tableau; var k : integer);
var i, j : integer;
begin
i:=1;
j:=1;
while i<=k do begin
              if t[j] <> t[i] then begin j:=j+1; t[j]:=t[i] end;
              i:=i+1
              end;
if k>0 then k:=j
end;
{-------------------------------------------------------------------------------}
procedure PARTIT(u, v : integer; var t : tableau; var p : integer);
label 1, 2, 3;
var i, j, x : integer;
begin
   i:=u;
   j:=v;
```

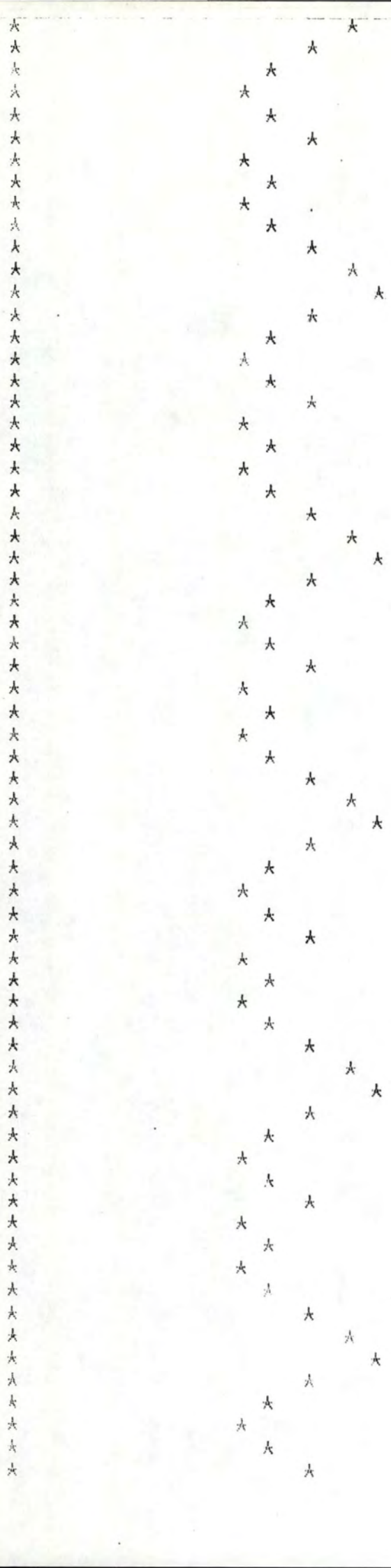
```
   x:=t[i];
1: if i=j then begin t[i]:=x; p:=i; goto 3 end
          else if t[j]>=x then begin j:=j-1; goto 1 end
                          else begin t[i]:=t[j]; i:=i+1; goto 2 end;
2: if i=j then begin t[i]:=x; p:=i; goto 3 end
          else if t[i]<=x then begin i:=i+1; goto 2 end
                          else begin t[j]:=t[i]; j:=j-1; goto 1 end;
3: end;
{-------------------------------------------------------------------------------}
procedure TRIQ(u, v : integer; var t : tableau);
var p : integer;
begin if u<v then begin PARTIT(u,v,t,p); TRIQ(u,p-1,t); TRIQ(p+1,v,t) end end;
{-------------------------------------------------------------------------------}
procedure QSORT(var t : tableau; li, ls : integer);
var u, v : integer;
begin
u:=li;
v:=ls;
TRIQ(u,v,t)
end;
{-------------------------------------------------------------------------------}
function INTER(chn : packar) : vers;
var zz, v, w, h, g, d, lib : integer; debut : array [1..3] of vers;
    abc : char; u : tableau;
{- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -}
function INTERSEC(dp, dq : vers) : vers;
var v, w, z : integer; r, lien : vers; louer, firstime : boolean;
begin
v:=1;  w:=1;  z:=1;  lib:=lib+1;
r:=nil; louer:=true; firstime:=true;
while (dp<>nil) and (dq<>nil)
      do begin
         while (v<=dp^.comp) and (w<=dq^.comp) and (z<=nmot)
               do if dp^.elem[v] = dq^.elem[w]
                     then begin
                          if louer then begin
                                        ALLOUER (nb);
                                        if firstime then begin
                                                         r:=struct;
                                                         firstime:=false
                                                         end
                                                    else lien^.suiv:=struct;
                                        lien:=struct;
                                        louer:=false
                                        end;
                          struct^.elem[z]:=dp^.elem[v];
                          z:=z+1; v:=v+1; w:=w+1
                          end
                     else if dp^.elem[v] < dq^.elem[w] then v:=v+1 else w:=w+1;
         if v > dp^.comp then begin v:=1; dp:=dp^.suiv end;
         if w > dq^.comp then begin w:=1; dq:=dq^.suiv end;
         if z > nmot then begin louer:=true; struct^.comp:=z-1; z:=1 end
         end;
if r<> nil then begin
                if z <> 1 then struct^.comp :=z-1;
                struct^.suiv :=nil
                end;
if lib=1 then lp:=r else lq:=r;
intersec :=r
end;
{- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -}
begin
zz:=1;  v:=3; lib:=0;
while (v<=ch) and (chn[v]<> ' ') do
case chn[v] of
'}': v:=v+1;
```

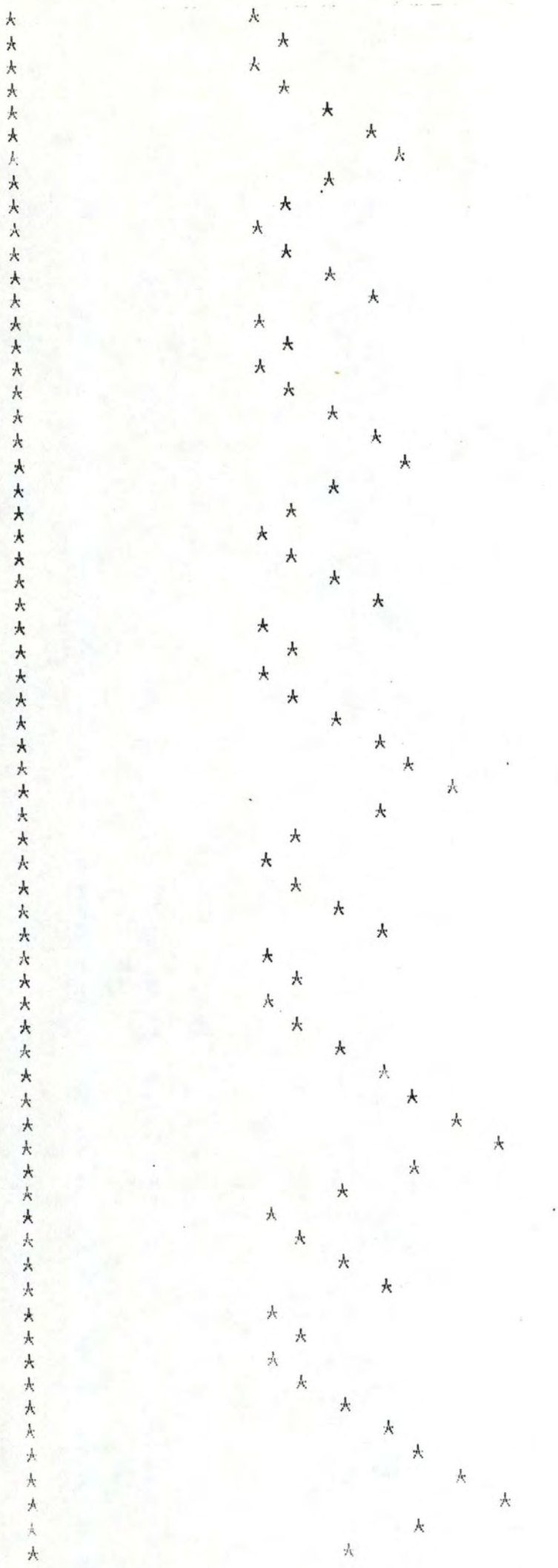
```
'0': begin
     case chn[v-1] of
             'a': debut[zz]:=a0;
             'b': debut[zz]:=b0;
             'c': debut[zz]:=c0
     end;
     v:=v+3; zz:=zz+1
     end;
'[': begin
     abc := chn[v-1];
     v:=v+1;  w:=0; h:=0;
     PARSEUR (chn,v,w,u);
     g:= u[1];
     d:= u[2];
     case abc of
     'a': for w:= g to d do begin h:= h+1; u[h]:= a[w] end;
     'b': for w:= g to d do begin h:= h+1; u[h]:= b[w] end;
     'c': for w:= g to d do begin h:= h+1; u[h]:= c[w] end
     end;
     debut[zz]:= GENERER ('t',u,1,h)
     end;
',': begin v:=v+3; zz:=zz+1 end;
']': v:= v+1
end;
LIBERER(lp,nb); LIBERER(lq,nb);
inter:=INTERSEC (debut[1], INTERSEC (debut[2],debut[3]));
if (debut[1]<>a0) and (debut[1]<>b0) and (debut[1]<>c0)
   then LIBERER (debut[1],nb);
if (debut[2]<>a0) and (debut[2]<>b0) and (debut[2]<>c0)
   then LIBERER (debut[2],nb);
if (debut[3]<>a0) and (debut[3]<>b0) and (debut[3]<>c0)
   then LIBERER (debut[3],nb)
end;
{-------------------------------------------------------------------------}
function TRISTRCROIS(chn : packar) : boolean;
var debut,deb : vers; trst : boolean; v, mval : integer;
begin
case chn[2] of
'a': debut:=GENERER ('t',a,1,m);
'b': debut:=GENERER ('t',b,1,n);
'c': debut:=GENERER ('t',c,1,p)
end;
deb:=debut;
trst:=true;
if debut <> nil then begin
                     mval:=debut^.elem[1]; v:=2;
                     while (debut <> nil) and (trst) do
                           begin
                           while (v<=debut^.comp) and (trst) do
                              if mval < debut^.elem[v]
                                 then begin
                                      mval:=debut^.elem[v];
                                      v:=v+1
                                      end
                                 else trst := false;
                           debut:=debut^.suiv; v:=1
                           end
                     end;
if trst then tristrcrois:=true else tristrcrois:=false;
LIBERER(deb,nb)
end;
{-------------------------------------------------------------------------}
function MAX(chn : packar) : integer;
var debut, deb : vers; u, uu : tableau; nbre, v, w, h : integer; abc : char;
begin
h:=0;
```

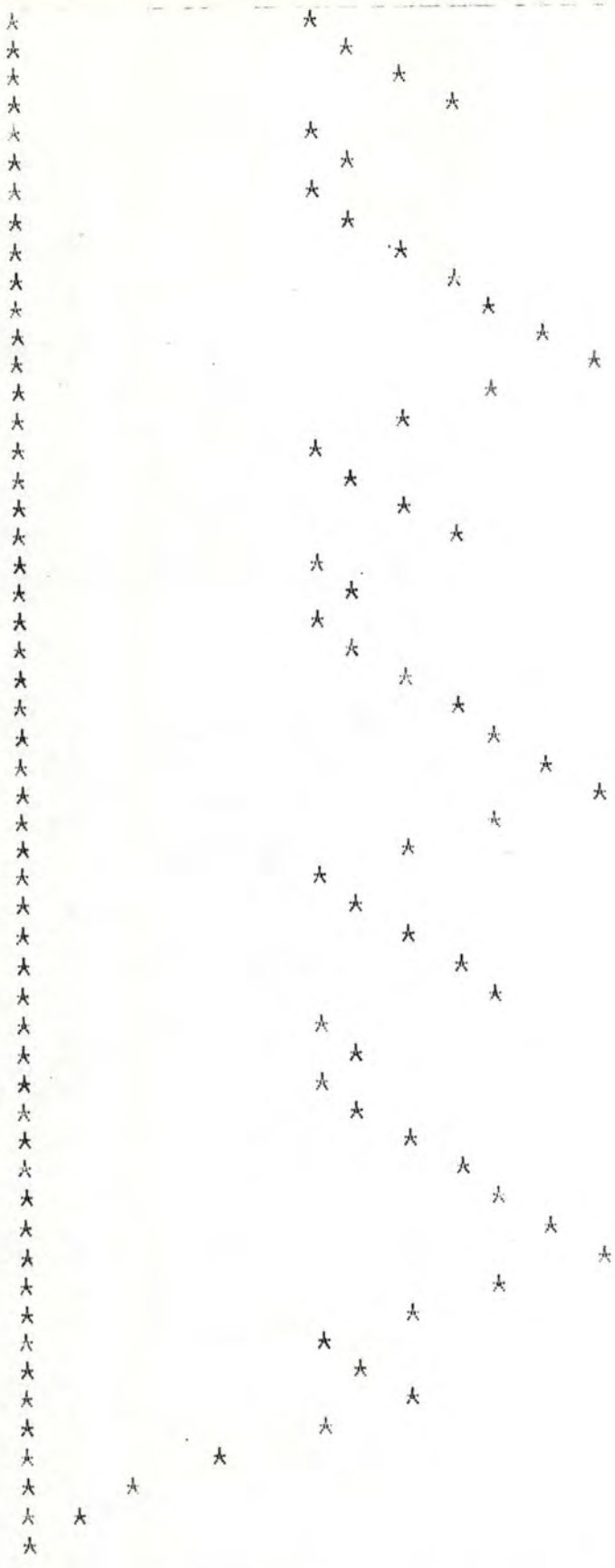
```
for v:=1 to i-1 do begin h:= h+1; uu[h]:= a[v] end;
for v:=1 to j-1 do begin h:= h+1; uu[h]:= b[v] end;
for v:=1 to k-1 do begin h:= h+1; uu[h]:= c[v] end;
QSORT (uu,1,h);
ELIMINER (uu,h);
debut := GENERER ('t',uu,1,h);
deb := debut;
max := zmin;
if debut<>nil then begin
                   while debut^.suiv<>nil do debut:=debut^.suiv;
                   max:=debut^.elem[debut^.comp]
                   end;
LIBERER(deb,nb)
end;
{-------------------------------------------------------------------------------}
function TROUVEMIN(g, d, h : integer) : integer;
begin
if d>h then begin d:= d+h; h:= d-h; d:= d-h end;
if g>d then begin g:= g+d; d:= g-d; g:= g-d end;
trouvemin :=g
end;
{-------------------------------------------------------------------------------}
function MIN(chn : packar) : integer;
var v, w : integer; p : vers; u, uu : tableau; tbool:boolean;
begin
if chn[1]='{' then begin
                   tbool:=true;
                   v:=3;
                   w :=0;
                   while chn[v]<> '}' do case chn[v] of
                                         '[': begin
                                              v:=v+1; PARSEUR (chn,v,w,u)
                                              end;
                                         ']': v:=v+1;
                                         ',': v:=v+2
                                          end;
                   uu[1]:= a[u[1]];
                   uu[2]:= b[u[2]];
                   uu[3]:= c[u[3]];
                   QSORT (uu,1,w);
                   ELIMINER (uu,w);
                   p :=GENERER ('t',uu,1,w)
                   end
              else begin
                   p:=INTER(' a0,b0,c0                                    ');
                   tbool:=false
                   end;
if p<> nil then min := p^.elem[1]
           else min := zmax;
if tbool then LIBERER(p,nb)
end;
{-------------------------------------------------------------------------------}
procedure FINALE;
var moy : integer;
begin
if ef=0 then moy:=0 else moy:=nal div ef; writeln; writeln;
writeln('avec ',nmot,' comme taille du tableau elem,');
writeln('la moyenne  de mots alloues pour les suites et ensembles est ',moy);
writeln('le nombre total de mots alloues pour suites et ensembles est ',nal);
writeln('le nbre max de mots alloues d''1 coup pour suites et ens. est ',qmax);
SHOW_TIME;
writeln; writeln; writeln;
writeln('fin du programme')
end;
{-------------------------------------------------------------------------------}
begin
```

```
            LIRE (a,b,c);
            INIT_TIME;
  { ass-1 } if not ((m>0) and (n>0) and (p>0) and
                    (DEFLETT(a,m,a0)) and
                    (DEFLETT(b,n,b0)) and
                    (DEFLETT(c,p,c0))) then goto 1;
  { ass-2 } if not ((TRISTRCROIS('[a]                                    ')) an
                    (TRISTRCROIS('[b]                                    ')) an
                    (TRISTRCROIS('[c]                                    ')))
                    then goto 2;
  { ass-3 } if not (INTER(' a0,b0,c0                              ')<>nil)
                   then goto 3;
            i:=1;
            j:=1;
            k:=1;
  { ass-4 } VARIANT(m-i+n-j+p-k,1,aexp1,aexp2);
10:{ ass_5 } if not ((1<=i) and (i<=m) and (1<=j) and (j<=n) and
                    (1<=k) and (k<=p) and
                    (INTER('{a[1..i-1]},{b[1..j-1]},{c[1..k-1]}  ')=nil) and
                    (MAX('{a[1..i-1],b[1..j-1],c[1..k-1]}       ') <
                     MIN('{a[i],b[j],c[k]}                      ')))
                    then goto 5;
  { ass-6 } if not (TERM(m-i+n-j+p-k,aexp1,aexp2)) then goto 6;
            if not ((a[i]=b[j]) and
                       (b[j]=c[k])) then begin
                                         x :=TROUVEMIN (a[i],b[j],c[k]);
                                         if a[i]=x then i:=i+1;
                                         if b[j]=x then j:=j+1;
                                         if c[k]=x then k:=k+1;
                                         goto 10
                                         end;
            x:= a[i];
  { ass-7 } if not (x=MIN ('INTER( a0,b0,c0)                      '))
               then goto 7;
            ECRIRE;
            FINALE;
            goto 40;
1 :         writeln; writeln('1''assertion 1 n''est pas verifiee'); goto 40;
2 :         writeln; writeln('1''assertion 2 n''est pas verifiee'); goto 40;
3 :         writeln; writeln('1''assertion 3 n''est pas verifiee'); goto 40;
5 :         writeln; writeln('1''assertion 5 n''est pas verifiee'); goto 40;
6 :         writeln; writeln('1''assertion 6 n''est pas verifiee'); goto 40;
7 :         writeln; writeln('1''assertion 7 n''est pas verifiee'); goto 40;
40:         end.
{-------------------------------------------------------------------------}
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>run ecommune
donnez la taille des suites et ensembles 6

en ordonne, * egale environ 3 mots memoire

**************************************************************************
*  *
*     *
*        *
*           *
*              *
*                 *
```









l'element minimum commun (en etalee) aux 3 vecteurs est : 20

avec 6 comme taille du tableau elem,
la moyenne de mots alloues pour les suites et ensembles est 12
le nombre total de mots alloues pour suites et ensembles est 1232
le nbre max de mots allouas d'1 coup pour suites et ens. est 32
le temps cpu est : 0:01.31 consomme en 1:02.92

```
fin du programme
```

EXPLICATION DU PROGRAMME CONTRACT (ETALEE)

- L'argument de la fonction TRICROIS étant unique et simple, nous avons pu directement définir la fonction TRICROIS (p) avec p, un argument de type pointeur, et garder la notation utilisée lors de la traduction des assertions en L2.

- Comme l'argument de la fonction TRISTRCROIS n'a qu'une forme, on aurait pu, tout en voulant garder la notation originale de l'utilisateur, définir cette fonction comme suit :

```
fonction TRISTRCROIS (c1 : char; var t : tableau; c2 : char;
                      i1 : integer; c3 : packed array [1..2] ;
                      i2 : integer; c4 : packed array [1..2] )
```

ou si on n'attachait aucune importance à cette notation, on aurait pu la définir comme suit :

```
fonction TRISTRCROIS (var t : tableau; i, j : integer).
```

- La fonction EGAL acceptant diverses formes d'arguments, il était impératif, selon notre point de vue de recourir à la manoeuvre citée dans les remarques générales;
de cette façon, au moment de l'appel :

  * elle analyse d'abord la chaîne de caractères pour découvrir à quelle forme d'argument elle a affaire,
  * ensuite, s'il s'agit des arguments de type ensemble, alors elle détermine les 2 ensembles après avoir trié leurs représentations et y avoir éliminé les copies éventuelles (soient p et q, les pointeurs de leurs représentations respectives) et
    s'il s'agit des arguments de type suite, alors elle détermine simplement les 2 suites (soient p et q les pointeurs de leurs représentations respectives,
  * enfin, elle passera les 2 arguments p et q à la fonction EGALITE (p, q) (qui est définie dans la fonction EGAL) qui détermine finalement si la suite ou l'ensemble (dont la représentation est pointée par p) est égal à la suite ou à l'ensemble (dont la représentation est pointée par q).

```
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>type econtract.pas
[INHERIT('TIME.ENV')]
program CONTRACT  (input, output);
{                                                                    }
{                contraction d'un vecteur en etalee                  }
{                                                                    }
label 1, 3, 4, 5, 20, 30, 40;
const nmax=60; tmax=1000; ch=20; noval=999999999;
type tableau = array [1..nmax] of integer;
     packar = packed array [1..ch] of char;
     vers = ^ensuite;
     ensuite = record
                comp : integer;
                elem : array [1..tmax] of integer;
                suiv : vers
                end;
var a, b : tableau;
    nmot, nb, ef, nal, addit, qmax, aexp1, aexp2, aexp3, aexp4,
    i, j, k, l, m, n, p : integer;
    struct, a0 : vers;
{--------------------------------------------------------------------}
procedure CMEM (nb : integer);
var i, v : integer;
begin
writeln; write('*'); v:=nb div 3;
if v>0 then begin for i:=1 to v do write(' '); write('*',' ') end
end;
{--------------------------------------------------------------------}
procedure ALLOUER (var nb : integer);
begin
new (struct);
nb:=nb+addit; nal:=nal+addit;
CMEM(nb)
end;
{--------------------------------------------------------------------}
procedure LIBERER(debut : vers; var nb : integer);
```

```
var i : integer; deb : vers;
begin
i:=0;
while debut <> nil do begin
                      i:=i+1; deb:=debut;
                      debut:=debut^.suiv; dispose(deb)
                      end;
if i>0 then begin
            nb := nb-(i*addit);
            CMEM(nb)
            end
end;
{-------------------------------------------------------------------------}
procedure LIRE (var a : tableau);
var i : integer;
begin
writeln; write('introduisez la taille du vecteur a contracter '); readln(n);
for i:=1 to n do a[i]:=n;
repeat
   write('donnez la taille des suites et ensembles '); readln(nmot); writeln;
   if nmot > tmax then writeln('taille trop grande.....')
until  nmot <= tmax;
addit:=nmot+2; writeln;
writeln('en ordonne, * egale environ 3 mots memoire');  writeln;
for i:=1 to 75 do write('*');
nb:=0; ef:=0; nal:=0; qmax:=0
end;
{-------------------------------------------------------------------------}
procedure ECRIRE (var b : tableau; n : integer);
var i : integer;
begin
LIBERER (a0,nb);
writeln; writeln; writeln; writeln ('le vecteur contracte (en etalee) est : ');
for i:=1 to n do write(' ',b[i])
end;
{-------------------------------------------------------------------------}
function GENERER (exp : char; var t : tableau; g, d : integer) : vers;
var j, qm : integer; r, lien : vers;
begin
r:=nil; qm:=0;
if g <= d then begin
               ALLOUER(nb); qm:=qm+addit; r:=struct; lien:=struct;
               repeat j:=1;
                      while (j<=nmot) and (g<=d) do
                            begin
                            if exp='i' then struct^.elem[j]:=g
                                       else struct^.elem[j]:=t[g];
                            g:=g+1; j:=j+1
                            end;
                      struct^.comp:=j-1;
                      if g<=d then begin
                                   ALLOUER(nb); qm:=qm+addit;
                                   lien^.suiv:=struct;
                                   lien:=struct
                                   end
                               else struct^.suiv := nil
               until g>d
               end;
generer:=r;
if r <> nil then begin ef:=ef + 1; if qm>qmax then qmax:=qm end
end;
{-------------------------------------------------------------------------}
function CHIFFRE (chn : packar; var v : integer) : integer;
var chif : integer;
begin
chif := 0;
```

```
while (v<=ch) and (chn[v]<>' ') and (chn[v]<>',') and (chn[v]<>']') and
      (chn[v]<>'.') and (chn[v]<>'}')
      do begin
         chif:=(chif*10) + (ord(chn[v]) - 48);
         v:=v+1
         end;
chiffre := chif
end;
{-------------------------------------------------------------------------}
procedure PARSEUR (chn : packar; var v,w : integer; var u : tableau);
var ip : integer;
begin
while (v<=ch) and (chn[v]<>']') and (chn[v]<>' ') and (chn[v]<>'}')
      do case chn[v] of
         '0','1','2','3','4','5','6','7','8','9' : begin
                                                    u[w] := CHIFFRE(chn, v);
                                                    w := w + 1
                                                    end;
         'i','j','k','l','m','n','p' : begin
                                       case chn[v] of
                                       'i': ip := i;
                                       'j': ip := j;
                                       'k': ip := k;
                                       'l': ip := l;
                                       'm': ip := m;
                                       'n': ip := n;
                                       'p': ip := p
                                       end;
                                       v:=v+1;
                                       if (chn[v]='+') or (chn[v]='-')
                                          then case chn[v] of
                                               '+' : begin
                                                     v:=v+1;
                                                     u[w]:=ip + CHIFFRE (chn,v)
                                                     end;
                                               '-' : begin
                                                     v:=v+1;
                                                     u[w]:=ip - CHIFFRE (chn,v)
                                                     end
                                               end
                                          else u[w]:=ip;
                                       w:=w+1
                                       end;
          '.',',': v:=v+1
          end
end;
{-------------------------------------------------------------------------}
function DEFLETT (var a : tableau; n : integer; var debut : vers) : boolean;
var i : integer; def : boolean;
begin
def:=true;
i:=1;
while (i<=n) and (def) do if a[i]=noval then def:=false
                                        else i:=i+1;
if def then begin deflett:=true; debut := GENERER ('t',a,1,n) end
       else begin deflett:=false; debut:= nil end
end;
{-------------------------------------------------------------------------}
procedure VARIANT(k, l : integer; var exp1, exp2 : integer);
begin exp1:=k + 1; exp2:=1 end;
{-------------------------------------------------------------------------}
function TERM(k : integer; var exp1, exp2 : integer) : boolean;
begin
if k+exp2 > exp1 then term:=false else begin exp1:=k; term:=true end
end;
{-------------------------------------------------------------------------}
```

```
function TRICROIS(debut:vers) : boolean;
var  tr : boolean; mval, v : integer;
begin
tr := true;
if debut <> nil then begin
                     mval:=debut^.elem[1]; v:=2;
                     while (debut <> nil) and (tr) do
                           begin
                           while (v<=debut^.comp) and (tr) do
                              if mval > debut^.elem[v]
                                 then tr := false
                                 else begin
                                      if mval < debut^.elem[v]
                                         then mval:=debut^.elem[v];
                                      v:=v+1
                                      end;
                           debut:=debut^.suiv; v:=1
                           end
                     end;
if tr then tricrois:=true else tricrois:=false
end;
{-------------------------------------------------------------------------}
function TRISTRCROIS(chn : packar) : boolean;
var mval, v, w : integer; trst : boolean; debut, deb : vers; u : tableau;
begin
v:=4;
w:=1;
PARSEUR (chn,v,w,u);
debut := GENERER ('t',b,u[1],u[2]);
deb:=debut;
trst:=true;
if debut <> nil then begin
                     mval:=debut^.elem[1]; v:=2;
                     while (debut <> nil) and (trst) do
                           begin
                           while (v<=debut^.comp) and (trst) do
                              if mval < debut^.elem[v]
                                 then begin
                                      mval:=debut^.elem[v];
                                      v:=v+1
                                      end
                                 else trst := false;
                           debut:=debut^.suiv; v:=1
                           end
                     end;
if trst then tristrcrois:=true else tristrcrois:=false;
LIBERER (deb,nb)
end;
{-------------------------------------------------------------------------}
procedure ELIMINER(var t : tableau; var k : integer);
var i, j : integer;
begin
i:=1;
j:=1;
while i<=k do begin
              if t[j] <> t[i] then begin j:=j+1; t[j]:=t[i] end;
              i:=i+1
              end;
if k>0 then k:=j
end;
{-------------------------------------------------------------------------}
procedure PARTIT(u, v : integer; var t : tableau; var p : integer);
label 1, 2, 3;
var i, j, x : integer;
begin
   i:=u;
```

```
    j:=v;
    x:=t[i];
1: if i=j then begin t[i]:=x; p:=i; goto 3 end
          else if t[j]>=x then begin j:=j-1; goto 1 end
                          else begin t[i]:=t[j]; i:=i+1; goto 2 end;
2: if i=j then begin t[i]:=x; p:=i; goto 3 end
          else if t[i]<=x then begin i:=i+1; goto 2 end
                          else begin t[j]:=t[i]; j:=j-1; goto 1 end;
3: end;
{-------------------------------------------------------------------------}
procedure TRIQ(u, v : integer; var t : tableau);
var p : integer;
begin if u<v then begin PARTIT(u,v,t,p); TRIQ(u,p-1,t); TRIQ(p+1,v,t) end end;
{-------------------------------------------------------------------------}
procedure QSORT(var t : tableau; li, ls : integer);
var u, v : integer;
begin
u:=li;
v:=ls;
TRIQ(u,v,t)
end;
{-------------------------------------------------------------------------}
function EGAL(chn : packar) : boolean;
var u : tableau; w, g, v, h, d : integer; virgule : boolean;
    ab : char; d0, d1 : vers;
{- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -}
function EGALITE(d0, d1 : vers) : boolean;
var d : integer; eg : boolean;
begin
eg := true;
while (d0<>nil) and (d1<>nil) and (eg)
do begin
   d:=1;
   while (d<=d0^.comp) and (eg) do if d0^.elem[d]<>d1^.elem[d]
                                   then eg:=false
                                   else d:=d+1;
   if d>d0^.comp then d0:=d0^.suiv;
   if d>d1^.comp then d1:=d1^.suiv
   end;
if (d0<>nil) or (d1<>nil) then eg:=false;
if eg then egalite:=true else egalite:=false
end;
{- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -}
begin
virgule:=false;
v:=3;
while (v<=ch) and (chn[v]<>' ') do
case chn[v] of
']': begin
     if (chn[v-1]='a') or (chn[v-1]='b')
        then if not(virgule)
                then if chn[v-1]='a' then d0 := GENERER ('t',a,1,n)
                                    else d0 := GENERER ('t',b,1,n)
                else if chn[v-1]='b' then d1 := GENERER ('t',a,1,n)
                                    else d1 := GENERER ('t',b,1,n);
     v:=v+1
     end;
'[': begin
     ab:=chn[v-1];
     v:=v+1;
     w:=1;
     PARSEUR (chn,v,w,u);
     case chn[1] of
     '<': begin
          g:=u[1];
          d:=u[2];
```

```
          h:=1;
          case ab of
          'a': for w:=g to d do begin u[h]:=a[w]; h:=h+1 end;
          'b': for w:=g to d do begin u[h]:=b[w]; h:=h+1 end
          end;
          h:=h-1;
          QSORT (u,1,h);
          ELIMINER (u,h);
          if not(virgule) then d0 := GENERER ('t',u,1,h)
                          else d1 := GENERER ('t',u,1,h)
          end;
     '[': if not(virgule)
             then if ab='a' then d0 := GENERER ('t',a,u[1],u[2])
                            else d0 := GENERER ('t',b,u[1],u[2])
             else if ab='a' then d1 := GENERER ('t',a,u[1],u[2])
                            else d1 := GENERER ('t',b,u[1],u[2])
     end
     end;
',': begin v:=v+2; virgule:=true end;
'a','b': v:=v+1;
'0': begin d1:=a0; v:=v+1 end;
'}': begin
     if (chn[v-1]='a') or (chn[v-1]='b')
        then begin
             case chn[v-1] of
             'a': for h:=1 to n do u[h]:=a[h];
             'b': for h:=1 to n do u[h]:=b[h]
             end;
             QSORT (u,1,h);
             ELIMINER (u,h);
             if not(virgule) then d0 := GENERER ('t',u,1,h)
                             else d1 := GENERER ('t',u,1,h)
             end;
     v:=v+1
     end
end;
egal:= EGALITE (d0, d1);
if d0 <> a0 then LIBERER (d0,nb);
if d1 <> a0 then LIBERER (d1,nb)
end;
{-------------------------------------------------------------------------}
procedure FINALE;
var moy : integer;
begin
if ef=0 then moy:=0 else moy:=nal div ef; writeln; writeln;
writeln('avec ',nmot,' comme taille du tableau elem,');
writeln('la  moyenne de mots alloues pour les suites et ensembles est ',moy);
writeln('le nombre total de mots alloues pour suites et ensembles est ',nal);
writeln('le nbre max de mots alloues d''1 coup pour suites et ens. est ',qmax);
SHOW_TIME;
writeln; writeln; writeln;
writeln('fin du programme')
end;
{-------------------------------------------------------------------------}
begin
          LIRE (a);
          INIT_TIME;
  { ass-1 } if not ((n>=1) and (DEELETT(a,n,a0)) and
                   (TRICROIS(a0))) then goto 1;
          i:=1;
          j:=1;
          b[1]:=a[1];
  { ass-2 } VARIANT(n-i,1,aexp1,aexp2);
30:{ ass_3 } if not ((1<=j) and (j<=i) and (i<=n) and
                   (TRISTRCROIS('[b[1..j]]          ')) and
                   (EGAL('<b[1..j]>,<a[1..i]> ')) and
```

```
                    (EGAL('[a],a0                    '))) then goto 3;
   { ass-4 } if not (TERM(n-i,aexpl,aexp2)) then goto 4;
             if i=n then goto 20;
             i:=i+1;
             if b[j]=a[i] then goto 30;
             j:=j+1;
             b[j]:=a[i];
             goto 30;
20:{ ass-5 } if not ((1<=j) and (j<=n) and
                    (TRISTRCROIS('[b[1..j]]          )) and
                    (EGAL('{b[1..j]},{a}       )) and
                    (EGAL('[a],a0                    '))) then goto 5;
             ECRIRE (b,j);
             FINALE;
             goto 40;
1 :          writeln; writeln('1''assertion 1 n''est pas verifiee'); goto 40;
3 :          writeln; writeln('1''assertion 3 n''est pas verifiee'); goto 40;
4 :          writeln; writeln('1''assertion 4 n''est pas verifiee'); goto 40;
5 :          writeln; writeln('1''assertion 5 n''est pas verifiee');
40 :         end.
{-------------------------------------------------------------------------------}
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>run econtract

introduisez la taille du vecteur a contracter 10
donnez la taille des suites et ensembles 5

en ordonne, * egale environ 3 mots memoire

**********************************************************************
*  *
*     *
*        *
*     *
*        *
*           *
*        *
*     *
*        *
*           *
*     *
*        *
*     *
*        *
*           *
*        *
*     *
*        *
*           *
*     *
*        *
*     *
*        *
*           *
*        *
*     *
```

le vecteur contracte (en etalee) est :
10

avec 5 comme taille du tableau elem,
la moyenne de mots alloues pour les suites et ensembles est 8
le nombre total de mots alloues pour suites et ensembles est 399
le nbre max de mots alloues d'1 coup pour suites et ens. est 14
le temps cpu est : 0:00.39 consomme en 0:19.01

fin du programme
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>

EXPLICATION DU PROGRAMME COMPACT (COMPACTEE)

- Les arguments de la fonction EGAL étant exprimées sous formes diverses dans la traduction des assertions en L2, il était plus facile de faire appel à la manoeuvre expliquée dans les remarques générales.
  Cette fonction, dans laquelle la fonction booléenne EGALITE est déclarée
  * analyse la chaine pour découvrir s'il s'agit du cas
    EGAL (a_0, [a]) ou du cas
    EGAL ([b [i_1..i_2]], [COMPACT (a [i_3..i_4])]);
    cette analyse fournira 2 pointeurs p et q qui seront le début de la représentation de la 1ère suite et celui de la 2ème suite respectivement,
  * et passera les 2 pointeurs p et q à la fonction EGALITE qui vérifiera effectivement si les 2 représentations de suites sont égales.

- Pour la fonction LONG, nous avons préféré recourir à la même manoeuvre (celle citée dans les remarques générales).
  Cette fonction
  * fait appel à la fonction COMPACT (x [i_1..i_2]) qui renvoit un pointeur p représentant la suite qui est la compactée de x [i_1..i_2] et
  * calcule la longueur de la suite représenté par p

```
DIANE>type kcompact.pas
[INHERIT('TIME.ENV')]
program COMPACT  (input, output);
{                                                                    }
{                  compactage d'un vecteur en compactee              }
{                                                                    }
label 1, 2, 4, 5, 7, 8, 10, 20, 30, 40, 50;
const nmax=60; tmax=1000; ch=28; quat=4; noval=999999999;
type tableau = array [1..nmax] of integer;
     packar = packed array [1..ch] of char;
     nat = (ds, dr, di);
     vers = ^ensuite;
     ensuite = record
               suiv : vers;
               case nature : nat of
                 di : (bi, bs : integer);
                 dr : (val, rep : integer);
                 ds : (comp : integer; elem : array [1..tmax] of integer)
               end;
var a, b : tableau;
    nmot, nb, ef, nal, addit, aexp1, qmax, aexp2, i, j,k, l, m, n, p : integer
    struct, a0 : vers;
{--------------------------------------------------------------------------}
procedure LIRE (var a : tableau; var p : integer);
var i : integer;
begin
writeln; write('introduisez la taille du vecteur ''a'' a compacter ');
readln(n);
for i:=1 to n do a[i]:=n;
writeln;
write('entrez la taille du vecteur ''b'' de compactage '); readln(p);
for i:=1 to nmax do b[i]:= noval;
repeat
   write('donnez la taille des suites et ensembles '); readln(nmot); writeln;
   if nmot > tmax then writeln('taille trop grande.........')
until  nmot <= tmax;
addit:=nmot+2; writeln;
writeln('en ordonne, * egale environ 3 mots memoire');   writeln;
for i:=1 to 75 do write('*');
nb:=0; ef:=0; nal:=0; qmax:=0
end;
{--------------------------------------------------------------------------}
procedure CMEM (nb : integer);
var i, v : integer;
begin
writeln; write('*'); v:=nb div 3;
if v>0 then begin for i:=1 to v do write(' '); write('*',' ') end
end;
{--------------------------------------------------------------------------}
function FORCOMP (exp : char; var a : tableau; g, d : integer) : vers;
var c : char;
    struct, lien, r : vers;
    firstime, louer, ok : boolean;
    preced, j, k, borne, qm : integer;
begin
r:=nil; qm:=0;
case exp of
'i': if g<=d then begin
                  new(struct); r:=struct; struct^.suiv:=nil;
                  if g<d then begin
                              struct^.nature:=di;
```

```
                        struct^.bi:= g;
                        struct^.bs:= d;
                        nb:=nb+quat; nal:=nal+quat;qm:=qm+quat;
                        CMEM (nb)
                        end
                    else begin
                        struct^.nature:=ds;
                        struct^.comp:= 1;
                        struct^.elem[1]:= g;
                        nb:=nb+addit; nal:=nal+addit; qm:=qm+addit;
                        CMEM (nb)
                        end
                end;
't': begin
    firstime := true;
    while g <= d
       do begin
          preced := a[g];
          if g+1 > d
             then c := 's'
             else if preced+1 = a[g+1]
                     then c := 'c'
                     else if preced-1 = a[g+1]
                             then c := 'd'
                             else if preced = a[g+1]
                                     then c := 'r'
                                     else c := 's';
          new(struct);
          if firstime then begin r := struct; firstime := false end
                      else lien^.suiv := struct;
          lien := struct;
          case c of
          'c' : begin
                g:=g+1;
                borne := a[g];
                g:=g+1;
                while (g<=d) and (borne+1=a[g]) do begin
                                                   borne:=a[g];
                                                   g:=g+1
                                                   end;
                struct^.nature := di;
                struct^.bi := preced;
                struct^.bs := borne;
                nb:=nb+quat; nal:=nal+quat; qm:=qm+quat; CMEM (nb)
                end;
          'd' : begin
                g:=g+1;
                borne := a[g];
                g:=g+1;
                while (g<=d) and (borne-1=a[g]) do begin
                                                   borne:=a[g];
                                                   g:=g+1
                                                   end;
                struct^.nature := di;
                struct^.bi := preced;
                struct^.bs := borne;
                nb:=nb+quat; nal:=nal+quat; qm:=qm+quat; CMEM (nb)
                end;
          'r' : begin
                g:=g+2;
                k:=2;
                while (g<=d) and (preced=a[g]) do begin k:=k+1; g:=g+1 end;
                struct^.nature := dr;
                struct^.val := preced;
                struct^.rep := k;
                nb:=nb+quat; nal:=nal+quat; qm:=qm+quat; CMEM (nb)
```

```
                end;
          's' : begin
                ok := true; j := 1; struct^.nature:=ds;
                louer := false;
                nb:=nb+addit; nal:=nal+addit; qm:=qm+addit; CMEM (nb);
                while (g<=d) and (ok)
                  do if g=d
                        then begin
                             if louer then begin
                                           new (struct);
                                           lien^.suiv :=struct;
                                           lien:=struct;
                                           struct^.nature:=ds;
                                           louer := false;
                                           nb:=nb+addit; nal:=nal+addit;
                                           CMEM (nb); qm:=qm+addit
                                           end;
                             struct^.elem[j] := a[g];
                             g:=g+1; j := j+1
                             end
                        else while (g<d) and (ok)
                             do begin
                                ok:= (a[g]<>a[g+1]+1) and (a[g]<>a[g+1]-1) and
                                     (a[g]<>a[g+1]);
                                if ok then begin
                                           if louer then begin
                                                         new (struct);
                                                         lien^.suiv :=struct;
                                                         lien :=struct;
                                                         struct^.nature:=ds;
                                                         louer := false;
                                                         nb:=nb+addit;
                                                         nal:=nal+addit;
                                                         qm:=qm+addit;
                                                         CMEM (nb)
                                                         end;
                                           struct^.elem[j] := a[g];
                                           g:=g+1;
                                           j:=j+1;
                                           if j>nmot then begin
                                                          louer := true;
                                                          struct^.comp:= j-1;
                                                          j := 1
                                                          end
                                           end
                                end;
                if j<>1 then struct^.comp := j-1
                end
          end
          end
     end
end;
if r <> nil then begin
                 struct^.suiv := nil;
                 ef:=ef+1; if qm>qmax then qmax:=qm
                 end;
forcomp := r
end;
{-------------------------------------------------------------------------}
procedure LIBERER(debut : vers; var nb : integer);
var i : integer; deb : vers;
begin
i:=0;
while debut <> nil do begin
                      deb:=debut;
                      debut:=debut^.suiv;
```

```
                    case deb^.nature of
                     di,dr: i:=i+quat;
                     ds: i:=i+addit
                    end;
                    dispose(deb)
                    end;
if i>0 then begin
            nb := nb-i;
            CMEM(nb)
            end
end;
{-------------------------------------------------------------------------}
function CHIFFRE (chn : packar; var v : integer) : integer;
var chif : integer;
begin
chif := 0;
while (v<=ch) and (chn[v]<>' ') and (chn[v]<>',') and (chn[v]<>']') and
      (chn[v]<>'.') and (chn[v]<>'}')
      do begin
         chif:=(chif*10) + (ord(chn[v]) - 48);
         v:=v+1
         end;
chiffre := chif
end;
{-------------------------------------------------------------------------}
procedure PARSEUR (chn : packar; var v,w : integer; var u : tableau);
var ip : integer;
begin
while (v<=ch) and (chn[v]<>']') and (chn[v]<>' ') and (chn[v]<>'}')
      do case chn[v] of
         '0','1','2','3','4','5','6','7','8','9' : begin
                                                    u[w] := CHIFFRE(chn, v);
                                                    w := w + 1
                                                    end;
         'i','j','k','l','m','n','p' : begin
                                       case chn[v] of
                                       'i': ip := i;
                                       'j': ip := j;
                                       'k': ip := k;
                                       'l': ip := l;
                                       'm': ip := m;
                                       'n': ip := n;
                                       'p': ip := p
                                       end;
                                       v:=v+1;
                                       if (chn[v]='+') or (chn[v]='-')
                                          then case chn[v] of
                                               '+' : begin
                                                     v:=v+1;
                                                     u[w]:=ip + CHIFFRE (chn,v
                                                     end;
                                               '-' : begin
                                                     v:=v+1;
                                                     u[w]:=ip - CHIFFRE (chn,v
                                                     end
                                               end
                                          else u[w]:=ip;
                                       w:=w+1
                                       end;
         '.',',': v:=v+1
         end
end;
{-------------------------------------------------------------------------}
function DEFLETT (var a : tableau; n : integer; var debut : vers) : boolean;
var i : integer; def : boolean;
begin
```

```
def:=true;
i:=1;
while (i<=n) and (def) do if a[i]=noval then def:=false
                                        else i:=i+1;
if def then begin deflett:=true; debut := FORCOMP ('t',a,1,n) end
       else begin deflett:=false; debut:= nil end
end;
{-------------------------------------------------------------------------------}
procedure VARIANT(k, l : integer; var exp1, exp2 : integer);
begin exp1:=k + 1; exp2:=1 end;
{-------------------------------------------------------------------------------}
function TERM(k : integer; var exp1, exp2 : integer) : boolean;
begin
if k+exp2 > exp1 then term:=false else begin exp1:=k; term:=true end
end;
{-------------------------------------------------------------------------------}
function COMPACT(var a,uu:tableau; i,j:integer):vers;
var h:integer;
begin
uu[1]:=1;
uu[2]:=a[i];
h:=2;
while i<j do if a[i]=a[i+1] then begin
                                 uu[h-1]:=uu[h-1]+1;
                                 i:=i+1
                                 end
                           else begin
                                 h:=h+2;
                                 uu[h-1]:=1;
                                 uu[h]:=a[i+1];
                                 i:=i+1
                                 end;
compact:=FORCOMP('t',uu,1,h)
end;
{-------------------------------------------------------------------------------}
function LONG (chn:packar):integer;
var l,v,w:integer; debut,qq:vers; u,uu:tableau;
begin
v:=2; w:=1;
while chn[v]<>'E' do v:=v+1; v:=v+1; PARSEUR (chn,v,w,u);
debut:=COMPACT(a,uu,u[1],u[2]); qq:=debut;
l:=0;
while debut<>nil do begin
                    case debut^.nature of
                    di: begin
                        i:=debut^.bi-debut^.bs;
                        if i<0 then i:=0-i;l:=l+i+1
                        end;
                    dr: l:=l+debut^.rep;
                    ds: l:=l+debut^.comp
                    end;
                    debut:=debut^.suiv
                    end;
long:=l;
LIBERER(qq,nb)
end;
{-------------------------------------------------------------------------------}
procedure ECRIRE (var b : tableau; var n, m, p : integer);
var i, j, cl : integer;
begin
LIBERER (a0,nb); writeln; writeln;
if n=0 then writeln('rien a compacter ..... m    = ',m)
       else begin
            if m>p then writeln('trop peu de place dans b, m    =',m);
            j:= m;
            writeln('voici (en compactee) le compactage : '); writeln; cl:=1;
```

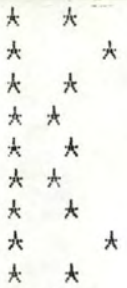
```
                                do if d0^.elem[v]<>d1^.elem[v] then eg:=false
                                                               else v:=v+1
                           end;
                di: if (d0^.bi<>d1^.bi) or (d0^.bs<>d1^.bs) then eg:=false;
```

```
            for i:=1 to j do if b[i]=noval then write(' pas de val ')
                                           else begin
                                                write(b[i],' ');
                                                c1:=c1+1;
                                                if c1=6 then begin
                                                             writeln;
                                                             c1:=1
                                                             end
                                                end;
            writeln
            end
end;
{-------------------------------------------------------------------------
function EGAL(chn : packar) : boolean;
var g, d, h, v, w : integer;
    d0, d1 : vers; u, uu : tableau;
{- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
function EGALITE(d0, d1 : vers) : boolean;
var v: integer; eg:boolean;
begin
eg:= true;
while (d0<>nil) and (d1<>nil) and (eg)
do if d0^.nature<>d1^.nature
      then eg:=false
      else begin
           case d0^.nature of
           ds: if d0^.comp<>d1^.comp
                  then eg:= false
                  else begin
                       v:=1;
                       while (eg) and (v<=d0^.comp)
                           do if d0^.elem[v]<>d1^.elem[v] then eg:=false
                                                          else v:=v+1
                       end;
           di: if (d0^.bi<>d1^.bi) or (d0^.bs<>d1^.bs) then eg:=false;
           dr: if (d0^.val<>d1^.val) or (d0^.rep<>d1^.rep) then eg:=false
           end;
           case d0^.nature of
           ds: begin
               d0:=d0^.suiv;
               if v>d1^.comp then d1:=d1^.suiv
               end;
           di,dr: begin d0:=d0^.suiv; d1:=d1^.suiv end
           end
           end;
if (d0<>nil) or (d1<>nil) then eg:=false;
if eg then egalite:=true else egalite:=false
end;
{- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
begin
v:=4;
if chn[v]='[' then begin
                   d0:=a0;
                   d1 :=FORCOMP ('t',a, 1, n)
                   end
              else begin
                   w:=1;
                   PARSEUR (chn,v,w,u);
                   d0:=FORCOMP ('t',b,u[1],u[2]);
                   while chn[v]<>'C' do v:=v+1;
                   while chn[v]<>'[' do v:=v+1;
                   v:= v+1; w:=1;
                   PARSEUR (chn,v,w,u);
                   d1:= COMPACT (a,uu,u[1],u[2])
                   end;
egal:= EGALITE (d0, d1);
```

```
if d0 <> a0 then LIBERER (d0,nb);
if d1 <> a0 then LIBERER (d1,nb)
end;
{-------------------------------------------------------------------------}
procedure FINALE;                                                  78.(35)
var moy : integer;
begin
if ef=0 then moy:=0 else moy:=nal div ef; writeln; writeln;
writeln('avec ',nmot,' comme taille du tableau elem,');
writeln('la moyenne  de mots alloues pour les suites et ensembles est ',moy);
writeln('le nombre total de mots alloues pour suites et ensembles est ',nal);
writeln('le nbre max de mots alloues d''1 coup pour suites et ens. est ',qmax)
SHOW_TIME;
writeln; writeln; writeln;
writeln('fin du programme')
end;
{-------------------------------------------------------------------------}
begin
              LIRE (a,p);
              INIT_TIME;
   { ass-1 } if not ((n>=0) and (p>=0)) then goto 1;
              if (n>0) and (p>=2)
                 then begin
   { ass-2 } if not (DEFLETT(a,n,a0)) then goto 2;
                       b[1]:=1;
                       b[2]:= a[1];
                       i:= 1;
                       m:= 2;
   { ass-3 } VARIANT(n-i,1,aexp1,aexp2);
10:{ ass_4 } if not ((1<=i) and (i<=n) and (1<=m) and (m<=p) and
                       (EGAL('[b[1..m]],[COMPACT(a[1..i])]'))) then goto 4;
   { ass-5 } if not (TERM(n-i,aexp1,aexp2)) then goto 5;
                       if i<n then
                              if a[i]=a[i+1]
                                 then begin
                                      b[m-1]:= b[m-1]+1;
                                      i:=i+1;
                                      goto 10
                                      end
                                 else begin
                                      m:= m+2;
                                      if m<=p then begin
                                                   b[m-1]:= 1;
                                                   b[m]:= a[i+1];
                                                   i:=i+1;
                                                   goto 10
                                                   end
                                              else goto 20
                                      end
                              else goto 30
                       end
                 else begin
                       if n=0 then m:=0
                              else m:=p+1;
                       goto 50
                       end;
20:{ ass-7 } if not ((m>=p+1) and
                       (LONG ('[COMPACT(a[1..n])]          ')>p) and
                       (EGAL('a0,[a]                      '))) then goto 7;
                       goto 50;
30:{ ass-8 } if not ((m<=p) and
                       (EGAL('[b[1..m]],[COMPACT(a[1..n])]')) and
                       (EGAL('a0,[a]                      '))) then goto 8;
50:           ECRIRE (b,n,m,p);
              FINALE;
              goto 40;
```

```
1 :           writeln; writeln('l''assertion 1 n''est pas verifiee'); goto 40;
2 :           writeln; writeln('l''assertion 2 n''est pas verifiee'); goto 40;
4 :           writeln; writeln('l''assertion 4 n''est pas verifiee'); goto 40;
5 :           writeln; writeln('l''assertion 5 n''est pas verifiee'); goto 40;
7 :           writeln; writeln('l''assertion 7 n''est pas verifiee'); goto 40;
8 :           writeln; writeln('l''assertion 8 n''est pas verifiee');
40:           end.                                                    78.(36)
{-------------------------------------------------------------------------}
```

```
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>
DIANE>run kcompact

introduisez la taille du vecteur 'a' a compacter 10
entrez la taille du vecteur 'b' de compactage 2
donnez la taille des suites et ensembles 4

en ordonne, * egale environ 3 mots memoire

**********************************************************************
* *
*   *
*     *
*   *
* *
*   *
*     *
*   *
* *
*   *
*     *
*   *
* *
*   *
*     *
*   *
* *
*   *
*     *
*   *
* *
*   *
*     *
*   *
* *
*   *
*     *
*   *
* *
*   *
*     *
*   *
* *
*  *
*    *
*  *
* *
```



voici (en compactee) le compactage :

10 10

avec 4 comme taille du tableau elem,
la moyenne de mots alloues pour les suites et ensembles est 5
le nombre total de mots alloues pour suites et ensembles est 128
le nbre max de mots alloues d'1 coup pour suites et ens. est 6
le temps cpu est : 0:00.18 consomme en 0:08.56

fin du programme

Le paramètre principal dans les 2 représentations (étalée et compactée) est la longueur nmot du tableau elem.
Notre campagne de mesures s'est déroulée comme suit :

- pour les programmes dont l'input est constitué d'un ou de plusieurs tableaux :
  1. nous avons fixé la longueur moyenne de ces tableaux à K (un entier),
  2. pour chacun de ces 8 programmes, nous avons testé quelques permutations remarquables de ces tableaux;
     ces permutations remarquables sont intuitivement celles qui risquent surtout de causer des temps d'exécution moins intéressants, vu qu'elles susceptibles, si on étudie les différents programmes testés, soit :
     * de générer le plus grand nombre de suites ou d'ensembles (cfr par exemple COMMUNE (page 80), M1 utilise la fonction INTER le plus grand nombre de fois) et / ou
     * de générer les plus longues suites ou les plus grands ensembles (cfr par exemple CONTRACT (page 80), K1 générera toujours les plus longues suites dans la fonction TRISTRCROIS) et / ou
     * de passer un nombre maximum de fois dans les boucles principales en exécutant toutes les instructions (ou presque toutes) de ces boubles (cfr par exemple DYCHO (page 80), D1 et D2 passeront au point de bouclage un nombre maximum de fois qui est d'environ $\lceil \log_2 (K + 1) \rceil$),
  3. pour chacune de ces permutations remarquables, nous avons considéré pour nmot les valeurs : 1, 4, 7, 10, ..., K, K+3, 2K, 5K, 10K, 50K, 100K.
     Le pas de 3 au départ de ces valeurs n'est pas un hasard; nous avons, en effet, remarqué qu'en général le temps d'exécution principalement, n'accusait pas de différences sensibles entre l et l + 2 si l est une valeur de nmot, et que les différences devenaient sensibles au-delà de l + 2;
     nous avons considéré que :
     "une variation $<$ 10 centièmes de secondes n'est pas sensible"
     (cfr tableau 0),
  4. nous avons enfin retenu les plus mauvaises valeurs des différents critères, elles se trouvent dans les tableaux I et II.

Le tableau I donne ces valeurs par critère et par test et, le tableau II les fournit par critère seulement.

Les permutations remarquables que l'on a testées sont :

* pour DYCHO :
  - la clé de recherche x se trouve soit en 1ère position, soit en dernière position du tableau a [1..K] ..... D1 (°)
  - la clé de recherche x ne se trouve pas dans le tableau a [1..K] D2

* pour PLATEAUX :
  - tableau a [1..K] tel que a [i] = a [j] pour tout i et tout j L1
  - tableau a [1..K] tel que a [i] ≠ a [i + 1] pour tout i : $1 \leqslant i < K$ L2

* pour TRIVECT :
  - tableau en ordre strictement croissant T1
  - tableau totalement non trié T2
  - tableau en ordre strictement décroissant T3

* pour COMPACT :
  - tableau a [1..K] totalement non trié C1
  - tableau a [1..K] tel que a [i] = a [j] pour tous i, j C2
  - tableau en ordre strictement croissant C3
  - tableau en ordre strictement décroissant C4

* pour COMMUNE :
  - l'élément minimum commun est le dernier dans les 3 tableaux de l'input M1

* pour CONTRACT :
  - tableau en ordre strictement croissant K1
  - tableau a [1..K] tel que a [i] = a [j] pour tous i et j K2

* pour PERMUTAT :
  - tableau a [1..K] : pour tout i : $1 \leqslant i \leqslant K - 1$, on a : a [i] $\geqslant$ 0 et a [i + 1] < 0 (ou a [i] < 0 et a [i + 1] $\geqslant$ 0) P1

(°) : code des tests

- pour les programmes dont l'input est constitué des entiers, nous avons simplement testé les valeurs nmot = 1, 4, 7, 10, ... sur les combinaisons (de ces entiers) susceptibles de produire principalement des temps d'exécution moins bons :

  * pour FACT : - nous avons considéré l'entier i le plus grand tel que sa factorielle soit acceptable par la machine ..... F1.

Cela a donc été fait pour les deux représentations.
Nous pensons avoir accordé un maximum de temps et d'intérêt à l'amélioration et à la simplification des algorithmes des différentes primitives, par conséquent, nous croyons que toute amélioration que l'on pourrait ultérieurement apporter à ces algorithmes (tout en respectant les particularités et les exigences des différentes représentations, sans introduction d'autres structures intermédiaires de données) ne modifiera pas nos conclusions d'une manière sensible.

		PERMUTAT (étalée)					PERMUTAT (compactée)					PERMUTAT (L1)
CODES TESTS	NMOT	CR1	CR2	CR3	CR4	CR5	CR1	CR2	CR3	CR4	CR5	TEMPS D'EXECUTION
P1	1	0:00.30	30	12	411	30	0:00.16	16	6	206	30	
	4	0:00.13	13	10	333	18	0:00.08	8	6	221	22	
	7	0:00.11	11	8	270	18	0:00.08	8	5	188	18	
	10	0:00.10	10	12	396	12	0:00.08	8	7	248	28	
	13	0:00.10	10	15	495	15	0:00.08	8	8	284	34	
	20	0:00.10	10	22	726	22	0:00.08	8	10	346	48	0:00.00
	50	0:00.10	10	52	1716	52	0:00.08	8	22	728	108	
	100	0:00.10	10	102	3366	102	0:00.08	8	40	1328	208	
	500	0:00.10	10	502	16566	502	0:00.09	9	185	6128	1008	
	1000	0:00.10	10	1002	33066	1002	0:00.08	8	367	12128	2008	

tableau 0 : tableau principal des résultats de la campagne de mesures où K = 10.

		COMMUNE (étalée)					COMMUNE (compactée)					COMMUNE (L1)
CODES TESTS	NMOT	CR1	CR2	CR3	CR4	CR5	CR1	CR2	CR3	CR4	CR5	TEMPS D'EXECUTION
M1	1	0:00.97	97	19	1935	54	0:00.64	64	9	1095	30	
	4	0:00.41	41	12	1254	30	0:00.33	33	7	834	18	
	7	0:00.35	35	14	1404	27	0:00.24	24	8	975	22	
	10	0:00.27	27	15	1488	24	0:00.24	24	9	1080	16	
	13	0:00.27	27	18	1815	30	0:00.24	24	11	1293	19	0:00.00
	20	0:00.27	27	26	2552	22	0:00.24	24	15	1790	26	
	50	0:00.27	27	62	6032	52	0:00.24	24	33	3920	56	
	100	0:00.27	27	121	11628	102	0:00.24	24	64	7470	106	
	500	0:00.27	27	600	58232	502	0:00.24	24	309	35870	506	
	1000	0:00.26	26	1198	116232	1002	0:00.25	25	615	71370	1006	

tableau 0 : tableau principal des résultats de la campagne de mesures où K = 10.

		CONTRACT (étalée)					CONTRACT (compactée)					CONTRACT (L1)
CODES TESTS	NMOT	CR1	CR2	CR3	CR4	CR5	CR1	CR2	CR3	CR4	CR5	TEMPS D'EXECUTION
K1	1	0:00.50	50	21	945	30	0:00.12	12	3	177	4	
	4	0:00.19	19	9	594	18	0:00.10	10	4	186	6	
	7	0:00.14	14	13	621	18	0:00.10	10	4	195	9	
	10	0:00.10	10	13	585	13	0:00.10	10	4	204	12	
	13	0:00.10	10	15	675	15	0:00.10	10	4	213	15	0:00.00
	20	0:00.10	10	22	990	22	0:00.10	10	5	234	22	
	50	0:00.11	11	52	2340	52	0:00.10	10	7	324	52	
	100	0:00.11	11	102	4590	102	0:00.10	10	10	474	102	
	500	0:00.10	10	502	22590	502	0:00.10	10	37	1674	502	
	1000	0:00.10	10	1002	45090	1002	0:00.10	10	70	3174	1002	
K2	1	0:00.27	27	10	459	30	0:00.09	9	3	147	4	
	4	0:00.14	14	9	414	18	0:00.09	9	5	246	6	
	7	0:00.11	11	11	513	18	0:00.09	9	7	345	9	
	10	0:00.10	10	12	540	12	0:00.09	9	9	444	12	
	13	0:00.10	10	15	675	15	0:00.09	9	12	543	15	
	20	0:00.10	10	22	990	22	0:00.09	9	17	774	22	0:00.00
	50	0:00.10	10	52	2340	52	0:00.10	10	39	1764	52	
	100	0:00.10	10	102	4590	102	0:00.10	10	75	3414	102	
	500	0:00.10	10	502	22590	502	0:00.12	12	369	16614	502	
	1000	0:00.11	11	1002	45090	1002	0:00.11	11	735	33114	1002	

tableau 0 :tableau principal des résultats de la campagne de mesures où K = 10.

		PLATEAUX (étalée)					PLATEAUX (compactée)					PLATEAUX (L1)
CODES TESTS	NMOT	CR1	CR2	CR3	CR4	CR5	CR1	CR2	CR3	CR4	CR5	TEMPS D'EXECUTION
L1	1	0:00.14	14	10	228	30	0:00.05	5	3	76	4	
	4	0:00.09	9	8	192	18	0:00.05	5	5	112	6	
	7	0:00.07	7	10	234	18	0:00.05	5	6	148	9	
	10	0:00.06	6	12	264	12	0:00.05	5	8	184	12	
	13	0:00.06	6	15	330	15	0:00.05	5	10	220	15	
	20	0:00.06	6	22	484	22	0:00.05	5	13	304	22	0:00.00
	50	0:00.06	6	52	1144	52	0:00.05	5	30	664	52	
	100	0:00.06	6	102	2244	102	0:00.05	5	57	1264	102	
	500	0:00.06	6	502	11044	502	0:00.05	5	275	6064	502	
	1000	0:00.06	6	1002	22044	1002	0:00.05	5	548	12064	1002	
L2	1	0:00.25	25	3	66	3	0:00.11	11	3	66	3	
	4	0:00.11	11	6	132	6	0:00.08	8	6	132	6	
	7	0:00.09	9	9	198	9	0:00.05	5	9	198	9	
	10	0:00.07	7	12	264	12	0:00.06	6	12	264	12	
	13	0:00.06	6	15	330	15	0:00.06	6	15	330	15	0:00.00
	20	0:00.06	6	22	484	22	0:00.05	5	22	484	22	
	50	0:00.07	7	52	1144	52	0:00.06	6	52	1144	52	
	100	0:00.06	6	102	2244	102	0:00.06	6	102	2244	102	
	500	0:00.06	6	502	11044	502	0:00.05	5	502	11044	502	
	1000	0:00.06	6	1002	22044	1002	0:00.05	5	1002	22044	1002	

tableau 0 : tableau principal des résultats de la campagne de mesures où K = 10.

		DYCHO (étalée)					DYCHO (compactée)					DYCHO (L1)
CODES TESTS	NMOT	CR1	CR2	CR3	CR4	CR5	CR1	CR2	CR3	CR4	CR5	TEMPS D'EXECUTION
D1	1	0:00.16	16	21	129	30	0:00.03	3	3	23	4	
	4	0:00.05	5	14	84	18	0:00.03	3	4	26	6	
	7	0:00.05	5	15	90	18	0:00.03	3	4	29	9	
	10	0:00.03	3	12	72	12	0:00.03	3	5	32	12	
	13	0:00.04	4	15	90	15	0:00.03	3	5	35	15	
	20	0:00.04	4	22	132	22	0:00.03	3	7	42	22	0:00.00
	50	0:00.04	4	52	312	52	0:00.03	3	12	72	52	
	100	0:00.03	3	102	612	102	0:00.04	4	20	122	102	
	500	0:00.03	3	502	3012	502	0:00.03	3	87	522	502	
	1000	0:00.04	4	1002	6012	1002	0:00.05	5	170	1022	1002	
D2	1	0:00.14	14	26	156	30	0:00.03	3	4	24	4	
	4	0:00.06	6	16	96	18	0:00.04	4	4	24	4	
	7	0:00.05	5	16	99	18	0:00.04	4	4	24	4	
	10	0:00.04	4	12	72	12	0:00.04	4	4	24	4	
	13	0:00.04	4	15	90	15	0:00.04	4	4	24	4	
	20	0:00.04	4	22	132	22	0:00.04	4	4	24	4	0:00.00
	50	0:00.04	4	52	312	52	0:00.04	4	4	24	4	
	100	0:00.04	4	102	612	102	0:00.04	4	4	24	4	
	500	0:00.04	4	502	3012	502	0:00.04	4	4	24	4	
	1000	0:00.04	4	1002	6012	1002	0:00.04	4	4	24	4	

tableau 0 : tableau principal des résultats de la campagne de mesures où K = 10.

		COMPACT (étalée)					COPACT (compactée)					COMPACT (L1)
CODES TESTS	NMOT	CR1	CR2	CR3	CR4	CR5	CR1	CR2	CR3	CR4	CR5	TEMPS D'EXECUTION
C1	1	0:00.52	51	35	840	60	0:00.51	50	31	722	51	
	4	0:00.15	14	19	456	30	0:00.18	17	24	596	42	
	7	0:00.12	11	18	450	27	0:00.11	10	29	700	48	
	10	0:00.10	9	19	432	26	0:00.09	8	35	856	60	
	13	0:00.06	5	21	510	30	0:00.06	5	42	1030	72	0:00.01
	20	0:00.06	5	22	528	22	0:00.06	5	59	1436	100	
	50	0:00.06	5	52	1248	52	0:00.06	5	132	3176	220	
	100	0:00.06	5	102	2448	102	0:00.06	5	253	6076	420	
	500	0:00.06	5	502	12048	502	0:00.06	5	1219	29276	2020	
	1000	0:00.06	5	1002	24048	1002	0:00.06	5	2428	58276	4020	
C2	1	0:00.15	15	8	192	30	0:00.06	6	5	132	6	
	4	0:00.08	8	7	168	18	0:00.07	7	5	132	6	
	7	0:00.07	7	9	234	18	0:00.06	6	7	196	9	
	10	0:00.06	6	12	288	12	0:00.07	7	10	240	12	
	13	0:00.06	6	15	360	15	0:00.06	6	12	294	15	0:00.00
	20	0:00.06	6	22	528	22	0:00.06	6	17	420	22	
	50	0:00.07	7	52	1248	52	0:00.07	7	40	960	52	
	100	0:00.06	6	102	2448	102	0:00.07	7	77	1860	102	
	500	0:00.06	6	502	12048	502	0:00.07	7	377	9060	502	
	1000	0:00.06	6	1002	24048	1002	0:00.06	6	752	18060	1002	

tableau 0 : tableau principal des résultats de la campagne de mesures où K = 10.

		COMPACT (étalée)					COMPACT (compactée)					COMPACT (L1)
CODES TESTS	NMOT	CR1	CR2	CR3	CR4	CR5	CR1	CR2	CR3	CR4	CR5	TEMPS D'EXECUTION
C3	1	0:00.54	54	35	840	60	0:00.36	36	25	618	51	
	4	0:00.16	16	19	456	30	0:00.11	11	20	492	36	
	7	0:00.11	11	18	450	27	0:00.11	11	19	474	30	
	10	0:00.05	5	18	432	24	0:00.11	11	22	528	36	
	13	0:00.05	5	21	510	30	0:00.11	11	21	510	27	0:00.00
	20	0:00.07	7	22	528	22	0:00.11	11	26	636	34	
	50	0:00.05	5	52	1248	52	0:00.12	12	49	1176	64	
	100	0:00.07	7	102	2448	102	0:00.12	12	86	2076	114	
	500	0:00.07	7	502	12048	502	0:00.12	12	386	9276	514	
	1000	0:00.08	8	1002	24048	1002	0:00.12	12	761	18276	1014	
C4	1	0:00.55	55	35	840	60	0:00.43	43	31	748	53	
	4	0:00.15	15	19	456	30	0:00.18	18	19	428	38	
	7	0:00.11	11	18	450	27	0:00.11	11	18	420	35	
	10	0:00.07	7	18	432	24	0:00.11	11	21	424	44	
	13	0:00.09	9	21	510	30	0:00.13	13	25	472	53	0:00.00
	20	0:00.08	8	22	528	22	0:00.11	11	26	520	52	
	50	0:00.08	8	52	1248	52	0:00.11	11	59	1416	112	
	100	0:00.08	8	102	2448	102	0:00.12	12	113	2316	212	
	500	0:00.08	8	502	12048	502	0:00.11	11	546	14116	1012	
	1000	0:00.08	8	1002	24048	1002	0:00.11	11	1088	26116	2012	

tableau 0 : tableau principal des résultats de la campagne de mesures où K = 10.

		TRIVECT (étalée)					TRIVECT (compactée)					TRIVECT (L1)
CODES TESTS	NMOT	CR1	CR2	CR3	CR4	CR5	CR1	CR2	CR3	CR4	CR5	TEMPS D'EXECUTION
T1	1	0:01.76	175	14	3489	30	0:01.20	119	5	1257	9	
	4	0:00.85	84	9	2292	18	0:00.62	61	5	1326	10	
	7	0:00.73	72	10	2673	18	0:00.62	61	6	1599	13	
	10	0:00.65	64	12	2952	12	0:00.62	61	7	1872	16	
	13	0:00.66	65	15	3690	15	0:00.62	61	8	2145	19	
	20	0:00.66	65	22	5412	22	0:00.62	61	11	2782	26	0:00.01
	50	0:00.66	65	52	12792	52	0:00.62	61	22	5512	56	
	100	0:oo.67	66	102	25092	102	0:00.62	61	40	10062	106	
	500	0:00.66	65	502	123492	502	0:00.62	61	188	46462	506	
	1000	0:00.66	65	1002	246492	1002	0:00.62	61	373	91962	1006	
T2	1	0:01.69	169	14	3489	30	0:01.01	101	7	1867	30	
	4	0:00.80	80	9	2292	18	0:00.83	83	7	1858	20	
	7	0:00.70	70	10	2673	18	0:00.70	70	9	2308	26	
	10	0:00.63	63	12	2952	12	0:00.61	61	11	2776	32	
	13	0:00.62	62	15	3690	15	0:00.61	61	13	3289	38	
	20	0:00.62	62	22	5412	22	0:00.61	61	18	4486	52	0:00.00
	50	0:00.62	62	52	12792	52	0:00.61	61	39	9616	112	
	100	0:00.63	63	102	25092	102	0:00.61	61	70	17346	212	
	500	0:00.62	62	502	123492	502	0:00.61	61	351	86566	1012	
	1000	0:00.62	62	1002	246492	1002	0:00.61	61	699	172066	2012	

tableau 0 : tableau principal des résultats de la campagne de mesures où K = 10.

		TRIVECT (étalée)					TRIVECT (compactée)					TRIVECT (L1)
CODES TESTS	NMOT	CR1	CR2	CR3	CR4	CR5	CR1	CR2	CR3	CR4	CR5	TEMPS D'EXECUTION
T3	1	0:01.77	176	14	3489	30	0:01.18	117	5	1416	9	
	4	0:00.83	82	9	2292	18	0:oo.66	65	6	1530	10	
	7	0:00.71	70	10	2678	18	0:00.66	65	7	1851	13	
	10	0:00.69	68	12	2952	12	0:00.66	65	8	2172	16	
	13	0:00.68	67	15	3690	15	0:00.67	66	10	2493	19	0:00.01
	20	0:00.68	67	22	5412	22	0:00.66	65	13	3242	26	
	50	0:00.69	68	52	12792	52	0:00.67	66	26	6452	56	
	100	0:00.69	68	102	25092	102	0:00.66	65	47	11802	106	
	500	0:00.68	67	502	123492	502	0:00.66	65	221	54602	506	
	1000	0:00.68	67	1002	246492	1002	0:00.66	65	439	108102	1006	

tableau 0 : tableau principal des résultats de la campagne de mesures où K = 10.

		FACT (étalée)					FACT (compactée)					FACT (L1)
CODES TESTS	NMOT	CR1	CR2	CR3	CR4	CR5	CR1	CR2	CR3	CR4	CR5	FACT (L1)
F1	1	0:00.29	29	17	609	48	0:00.24	24	6	212	9	
	4	0:00.11	11	10	366	24	0:00.13	13	5	188	10	
	7	0:00.09	9	12	423	27	0:00.07	7	7	242	13	
	10	0:00.06	6	14	492	24	0:00.07	7	8	296	16	
	13	0:00.06	6	16	570	30	0:00.07	7	10	350	19	0:00.00
	20	0:oo.05	5	22	748	22	0:00.07	7	14	476	26	
	50	0:00.06	6	52	1768	52	0:00.06	6	29	1016	56	
	100	0:00.06	6	102	3468	102	0:00.07	7	56	1916	106	
	500	0:00.06	6	502	17068	502	0:00.06	6	268	9116	506	
	1000	0:00.05	5	1002	34068	1002	0:00.07	7	532	18116	1006	

tableau 0 : tableau principal des résultats de la campagne de mesures.

PROGRAMMES ET CODES TESTS	ETALEE					COMPACTEE				
	CR1	CR2	CR3	CR4	CR5	CR1	CR2	CR3	CR4	CR5
TRIVECT : T1	0:01.76	175	1002	246492	1002	0:01.20	119	373	91962	1006
T2	0:01.69	169	1002	246492	1002	0:01.01	101	699	172066	2012
T3	0:01.77	176	1002	246492	1002	0:01.18	117	439	108102	1006
COMPACT : C1	0:00.52	51	1002	24048	1002	0:00.51	50	2428	58276	4020
C2	0:00.15	15	1002	24048	1002	0:00.09	9	752	18060	1002
C3	0:00.54	54	1002	24048	1002	0:00.36	36	761	18276	1014
C4	0:00.55	55	1002	24048	1002	0:00.43	43	1088	22116	2012
COMMUNE : M1	0:00.97	97	1198	116232	1002	0:00.64	64	615	71370	1006
CONTRACT : K1	0:00.50	50	1002	45090	1002	0:00.12	12	70	3174	1002
K2	0:00.27	27	1002	45090	1002	0:00.12	12	735	33114	1002
PERMUTAT : P1	0:00.30	30	1002	33066	1002	0:00.16	16	367	12128	2008
PLATEAUX : L1	0:00.14	14	1002	22044	1002	0:00.05	5	548	12064	1002
L2	0:00.25	25	1002	22044	1002	0:00.11	11	1002	22044	1002
FACT : F1	0:00.29	29	1002	34068	1002	0:00.24	24	532	18116	1006
DYCHO : D1	0:00.16	16	1002	6012	1002	0:00.05	5	170	1022	1002
D2	0:00.14	14	1002	6012	1002	0:00.06	6	4	24	4

tableau I : tableau des résultats - les valeurs les plus mauvaises du tableau 0, par critère et par test.

		ETALEE	COMPACTEE
PERMUTAT	CR1	0:00.30	0:00.16
	CR2	30	16
	CR3	1002	367
	CR4	33066	12128
	CR5	1002	2008
COMMUNE	CR1	0:00.97	0:00.64
	CR2	97	64
	CR3	1198	615
	CR4	116232	71370
	CR5	1002	1006
FACT	CR1	0:00.29	0:00.24
	CR2	29	24
	CR3	1002	532
	CR4	34068	18116
	CR5	1002	1006
TRIVECT	CR1	0:01.77	0:01.01
	CR2	176	101
	CR3	1002	699
	CR4	246492	172066
	CR5	1002	2012
COMPACT	CR1	0:00.55	0:00.51
	CR2	55	50
	CR3	1002	2428
	CR4	24048	58276
	CR5	1002	4020
CONTRACT	CR1	0:00.50	0:00.12
	CR2	50	12
	CR3	1002	735
	CR4	45090	33114
	CR5	1002	1002
PLATEAUX	CR1	0:00.25	0:00.11
	CR2	25	11
	CR3	1002	1002
	CR4	22044	22044
	CR5	1002	1002
DYCHO	CR1	0:00.16	0:00.06
	CR2	16	6
	CR3	1002	170
	CR4	6012	1022
	CR5	1002	1002

tableau II : tableau des résultats - les valeurs les plus mauvaises du tableau 0, par critère.

B. CONTROLE DE LA TAILLE "NMOT" DU TABLEAU "ELEM"

Il a été évidemment très intéressant de connaître les valeurs de nmot autour desquelles étaient produits les plus petits temps d'exécution (tableau III) et les plus petites consommations de la place mémoire par les suites et les ensembles (tableau IV).

PROGRAMMES L2	ETALEE	COMPACTEE
TRIVECT : T1	10	4
T2	13	10
T3	10	4
COMPACT : C1	13	13
C2	10	1
C3	10	4
C4	10	4
COMMUNE : M1	10	7
CONTRACT : K1	10	4
K2	7	1
PERMUTAT : P1	10	4
PLATEAUX : L1	10	1
L2	13	7
FACT : F1	10	7
DYCHO : D1	10	1
D2	10	1

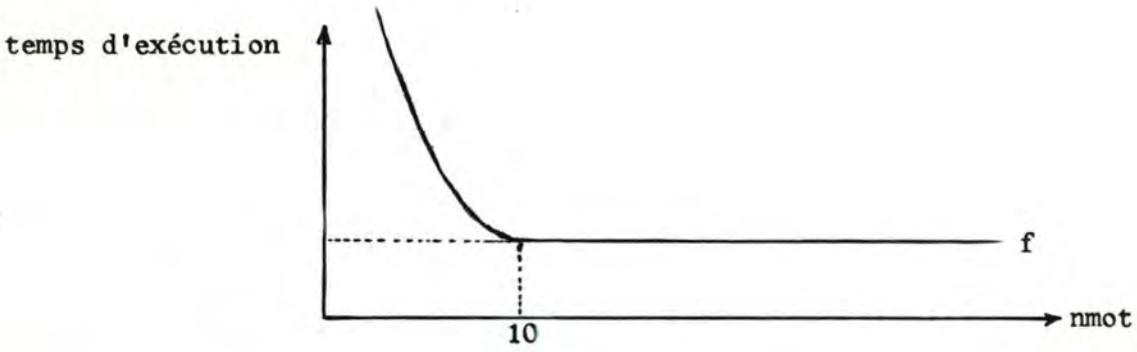
tableau III : les différentes valeurs de "nmot" à partir desquelles le temps d'exécution (CR1 ou CR2) devie t minimum (K étant à 10 pour les programmes dont l'input est constitué d'un ou de plusieurs tableaux).

Durant notre campagne de mesures, nous avons testé d'autres valeurs de la longueur K des tableaux, à savoir : 5, 15, 20, 30, 50, 100, 500 et 1000 et, il en est toujours ressorti deux faits importants :

1. A l'intérieur de chaque test, pour K = 10 par exemple et pour la représentation étalée, l'allure du temps d'exécution en fonction des valeurs de nmot, quand on n'imprime pas le graphique de l'activité de la mémoire, est en général de la forme suivante :

PROGRAMMES L2	ETALEE	COMPACTEE
TRIVECT : T1	4	1
T2	4	4
T3	4	1
COMPACT : C1	10	4
C2	4	1
C3	10	7
C4	10	7
COMMUNE : M1	4	4
CONTRACT : K1	10	1
K2	4	1
PERMUTAT : P1	7	7
PLATEAUX : L1	1	1
L2	4	1
FACT : F1	4	4
DYCHO : D1	4	1
D2	4	1

tableau IV : les différentes valeurs de "nmot" à partir desquelles la consommation totale (CR4) de la place mémoire est minimale (K étant 10 pour les programmes dont l'input est constitué d'un ou de plusieurs tableaux).



le minimum se présentant au plus tôt en moyenne :

3 fois sur 10 à la valeur nmot = 13,
8 fois sur 10 à la valeur nmot = 10,
2 fois sur 10 à la valeur nmot = 7,
1 fois sur 10 à la valeur nmot = 4,
0 fois sur 10 à la valeur nmot = 1
(f (nmot) = ∞, si nmot = 0 évidemment);

on remarque donc que le minimum du temps d'exécution est souvent atteint à partir de nmot = 10 (cfr tableau III).

Comme il est évident que le temps d'exécution dépend entr'autres de la longueur de la chaine des structures "ensuite", le plus petit temps d'exécution ne se produira presque jamais à la valeur nmot = 1 (pour K > 1) vu que la chaine est de longueur maximale pour cette valeur.

Toujours en représentation étalée,
pour K = 20, les valeurs de nmot fournissant les temps d'exécution les plus intéressants commencent à apparaître autour de 20,
pour K = 30, les plus petites valeurs de nmot donnant les temps d'exécution les plus petits sont fort concentrées autour de 30 et,
pour K = 50, c'est la valeur nmot = 50 qui produit souvent les plus petits temps d'exécution.

Nous pouvons donc nous appuyer sur cette expérience et affirmer d'une manière générale que le minimum du temps d'exécution, dans la représentation étalée est presque à coup sûr atteint quand nmot = K, auquel cas, la longueur de la chaîne des structures "ensuite" est minimale;

il s'agit certainement d'une RECHERCHE INVARIABLE du nombre minimum de chaînages par rapport au nombre d'éléments (d'une manière générale) dont on doit générer une représentation (étalée) de suite ou d'ensemble.
Donc, en vue de permettre un accroissement des performances (temps d'exécution) de cette représentation, nmot devrait être fixé en fonction de la longueur K des tableaux (si, étant bien entendu, des suites ou des ensembles doivent être constitués à l'aide des structures "ensuite" avec les K données de ces tableaux lors de la vérification des assertions).

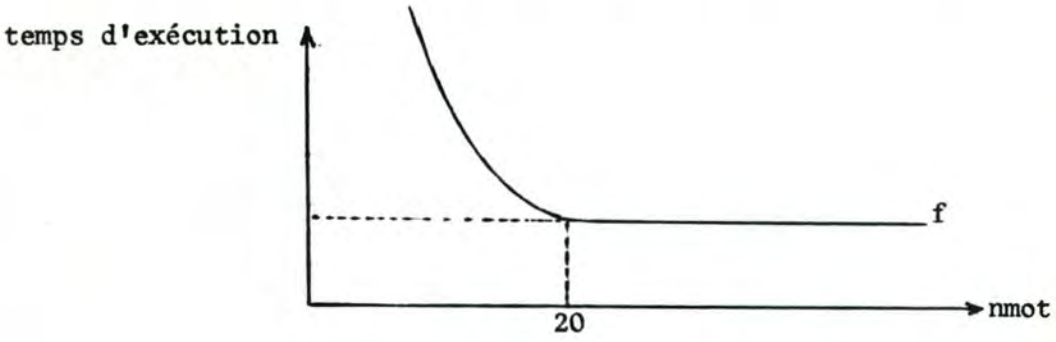
La représentation compactée étant un peu plus vigilante sur la succession des éléments à représenter dans une suite ou un ensemble présente quasiment la même fonction (que ci-avant) mais avec les valeurs de nmot (à partir desquelles sont produits les plus petits temps d'exécution) glissant de plus en plus vers la gauche en fonction :

- du nombre d'éléments se succédant par pas de "+ 1" et,
- du nombre d'éléments se succédant par pas de "- 1" et,
- du nombre d'éléments répétés

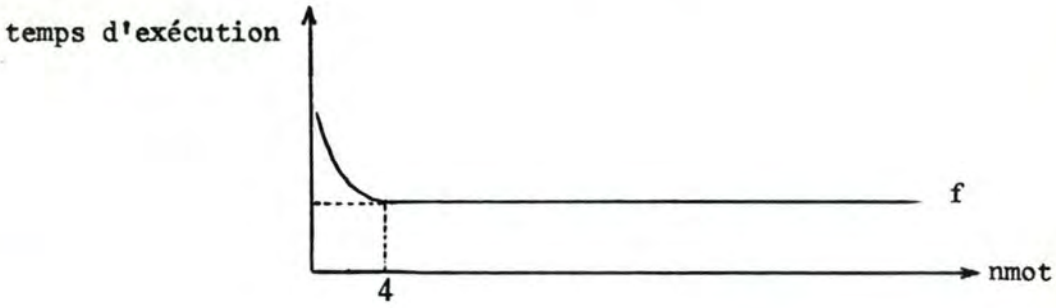
dans le tableau (d'où naîtra une représentation de suite ou d'ensemble).
Appelons succession remarquable une sucession d'éléments par pas de + 1 ou - 1, ou d'éléments répétés;

- pour K = 10, les plus petites valeurs de nmot fournissant les plus petits temps d'exécution sont fort concentrées autour de 4, si tous les éléments du tableau forment une succession remarquable, autour de 10, sinon;
- pour K = 20, le minimum des temps d'exécution est souvent visible à partir de nmot = 4, si tous les éléments du tableau forment une succession remarquable,
  à partir de nmot = 20, sinon;
- pour K = 50, ce même minimum apparaît en général à partir de nmot = 4, si tous les éléments du tableau formant une succession remarquable,
  à partir de nmot = 50, sinon.

Pour K = 20 par exemple, les deux allures extrêmes sont :



si tous les éléments du tableau ne forment pas une succession remarquable.



si tous les éléments du tableau forment une succession remarquable.

Que les différents éléments du tableau se présentent dans un ordre intéressant pour la représentation compactée ou non, nous remarquons encore que les plus petits temps d'exécution sont à attendre autour de nmot déterminant les chaînages les moins longs, c-à-d
nmot =
K - (nombre d'éléments formant une succession remarquable).

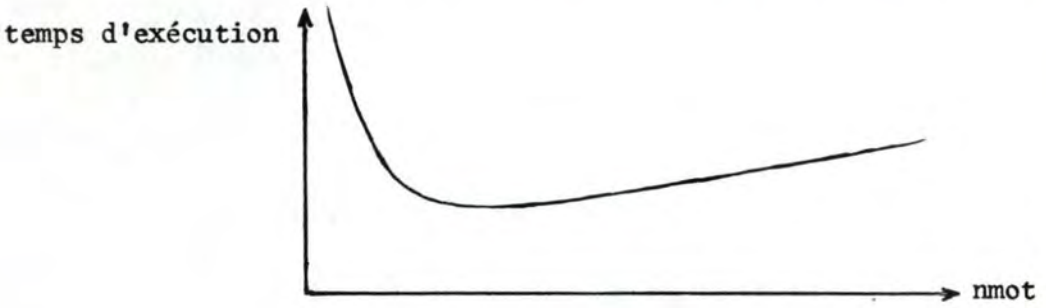
Par exemple :

- le test C2 (cfr page 80), comme on peut le remarquer, n'a pas très souvent utilisé dans la génération des structures de représentation de ses suites et ensembles le cas "ds", il est donc clair que la chaîne de ses structures "ensuite" soit déjà minimum à nmot = 1 et qu'elle le reste quel que soit nmot
(cfr tableau 0 page 87);

- on peut remarquer que le test C1 (cfr page 80) n'a presque utilisé que des cas ds dans la génération des représentations de ses suites et ensembles;
  ici, les seules valeurs de nmot déterminant une chaîne minimale sont celles qui sont telles que nmot 10 (cfr tableau III page 94).

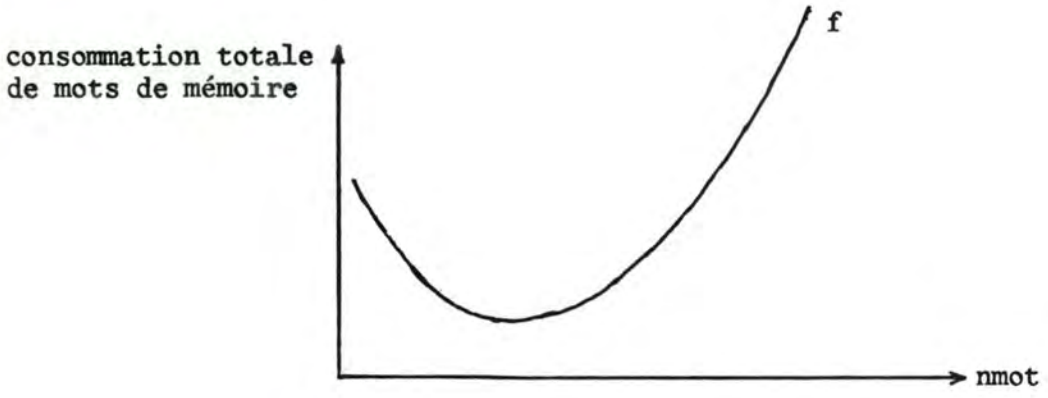
On voit donc que si les autres facteurs dont dépend le temps de réponse ne sont pas très prépondérants, alors le temps d'exécution est quasiment fonction de la longueur de la chaîne des structures "ensuite".

Si l'on désire visualiser le graphique de l'activité dynamique de la mémoire, quelle que soit la représentation et surtout quand nmot >> K, la courbe du temps d'exécution prend la forme suivante :



car comme l'impression de la valeur correspondante à chaque valeur de l'abscisse dans ce graphique, qui est en général très long, dépend de nmot, l'exécution de la procédure imprimant ce graphique se fait sentir doucement mais sûrement dans le temps de réponse final.

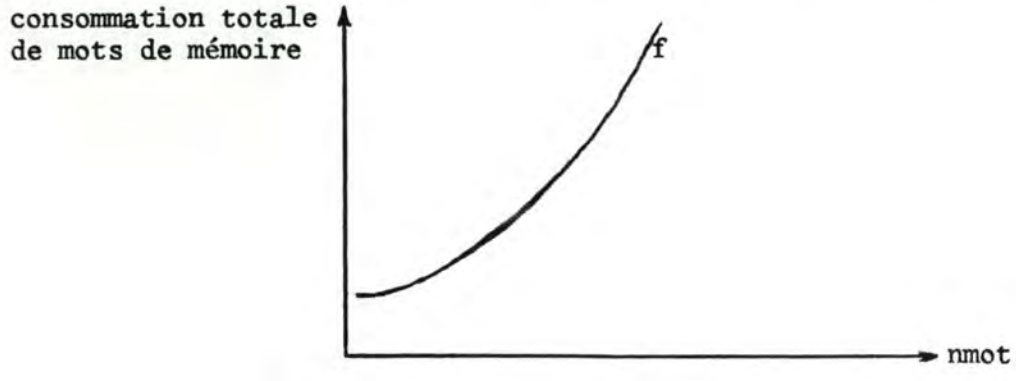
2. Pour la <u>représentation étalée</u>, la consommation de l'<u>espace mémoire</u>, surtout la consommation totale (CR4) (cfr tableau IV page 95) suit la fonction suivante :



avec le minimum de la consommation totale de mots de mémoire apparaissant en général

- pour les valeurs de nmot $\in [4, 10]$ avec K = 10,
- pour les valeurs de nmot $\in [8, 15]$ avec K = 15,
- pour les valeurs de nmot $\in [11, 20]$ avec K = 20,
- pour les valeurs de nmot $\in [17, 30]$ avec K = 30,
- pour les valeurs de nmot $\in [30, 50]$ avec K = 50.

Pour la <u>représentation compactée</u>, si les K éléments du tableau ne forment pas une succession remarquable, alors la consommation totale <u>minimale</u> de l'espace mémoire est atteinte à peu près aux mêmes valeurs de nmot que pour la représentation étalée.
Si les K éléments du tableau constituent tous une succession remarquable, la consommation de l'espace mémoire suit la fonction :



avec la consommation totale minimale apparaissant, pour K = 5, 10, 15, 20, 30, 50, 100, 500 et 1000,
à la valeur de nmot = 1.

Nous pouvons constater, pour la représentation étalée, qu'étant donné K :
- une très petite valeur de nmot, soit n1, entraîne un accroissement du nombre de structures "ensuite" dans la chaîne;
  de ce fait, le nombre de mots de gestion de la structure, à savoir COMP et SUIV augmente, ce qui accroît finalement le nombre total de mots de mémoire consommés;
- une très grande valeur de nmot, soit n2, entraîne malgré le petit nombre de structures "ensuite", un accroissement du nombre de mots inutiles dans le tableau "elem", ce qui augmente finalement le nombre total de mots de mémoire consommés.

Donc, la consommation totale minimale sera atteinte à une valeur v de nmot telle que $v \in [n1, n2]$.

Pour la représentation compactée, on voit que quand tous les éléments du tableau (à partir desquels on devra générer des suites ou des ensembles en vue de la vérification des assertions) se succèdent à son avantage, ce minimum apparaît toujours à nmot = 1, ce qui est évident car pour nmot > 1, elle fait sûrement une consommation inutile des mots étant donné qu'elle n'a presque pas besoin de structures de nature "ds".

Un autre aspect du problème est de savoir dans quelle mesure K est un paramètre adéquat pour caractériser la complexité des programmes au point de vue temps d'exécution.
On pourrati ainsi découvrir éventuellement au-delà de quelle taille K des tableaux, il devient carrément trop coûteux d'utiliser le langage L2 pour la vérification des assertions pendant l'exécution.
Les résultats obtenus avec nmot fixé (à 10) et une variation de K sont repris dans les tableaux V et VI pour les 2 représentations, dans le tableau VII pour les programmes sans assertions.

K / PROGRAMMES L2	5	10	15	20	30	50	100	500	1000
TRIVECT : T1	0:00.50	0:00.65	0:01.00	0:02.79	0:09.30	0:49.75	9:53.44	»40:52.28 *	
T2	0:00.47	0:00.63	0:00.89	0:02.49	0:08.11	0:41.59	8:18.31	»39:49.76 *	
T3	0:00.52	0:00.69	0:01.01	0:03.57	0:10.53	0:48.58	8:38.74	»41:53.17 *	
COMPACT : C1	0:00.04	0:00.10	0:00.13	0:00.19	0:00.27	0:00.47	0:02.82	0:28.61	1:52.00
C2	0:00.04	0:00.06	0:00.08	0:00.13	0:00.13	0:00.17	0:00.34	0:03.14	0:10.42
C3	0:00.03	0:00.05	0:00.09	0:00.17	0:00.26	0:00.48	0:01.42	0:28.56	1:51.59
C4	0:00.03	0:00.07	0:00.10	0:00.18	0:00.26	0:00.46	0:01.42	0:28.57	1:51.59
COMMUNE : M1	0:00.15	0:00.27	0:00.29	0:00.35	0:00.89	0:02.60	0:11.26	5:24.74	24:06.06
CONTRACT : K1	0:00.10	0:00.10	0:00.11	0:00.14	0:00.28	0:01.00	0:05.21	6:22.18	20:08.00
K2	0:00.10	0:00.10	0:00.11	0:00.11	0:00.19	0:00.56	0:02.58	3:08.12	14:47.41
PERMUTAT : P1	0:00.05	0:00.10	0:00.15	0:00.15	0:00.24	0:00.61	0:00.67	1:17.77	5:30.31
PLATEAUX : L1	0:00.02	0:00.06	0:00.06	0:00.20	0:00.30	0:00.50	0:02.52	3:08.90	9:51.78
L2	0:00.02	0:00.07	0:00.07	0:00.12	0:00.24	0:00.50	0:02.53	3:07.59	9:49.11
DYCHO : D1	0:00.01	0:00.03	0:00.06	0:00.09	0:00.10	0:00.09	0:00.11	0:00.30	0:00.64
D2	0:00.02	0:00.04	0:00.06	0:00.09	0:00.09	0:00.09	0:00.11	0:00.32	0:00.68

les temps marqués par * dans TRIVECT sont ceux de K = 150

tableau V : temps d'exécution avec nmot fixé (à 10) et une variation de K pour la représentation étalée.

PROGRAMMES L2 \ K	5	10	15	20	30	50	100	500	1000
TRIVECT : T1	0:00.34	0:00.62	0:01.15	0:02.67	0:09.22	0:49.45	9:45.32	>>44:05.46 *	
T2	0:00.23	0:00.61	0:01.02	0:02.40	0:08.57	0:43.48	7:35.05	>> 37:01.08 *	
T3	0:00.36	0:00.66	0:01.20	0:02.71	0:08.74	0:44.32	8:15.67	>>40:04.66 *	
COMPACT : C1	0:00.07	0:00.08	0:00.08	0:00.08	0:00.09	0:00.40	0:01.78	0:41.74	2:45.01
C2	0:00.03	0:00.07	0:00.05	0:00.07	0:00.07	0:00.14	0:00.26	0:03.02	0:10.39
C3	0:00.10	0:00.11	0:00.11	0:00.12	0:00.11	0:00.45	0:01.80	0:41.66	2:44.05
C4	0:00.10	0:00.11	0:00.12	0:00.12	0:00.16	0:00.50	0:01.84	0:41.13	2:43.08
COMMUNE : M1	0:00.14	0:00.24	0:00.25	0:00.37	0:00.92	0:02.65	0:10.78	5:17.20	23:05.08
CONTRACT : K1	0:00.06	0:00.10	0:00.12	0:00.17	0:00.39	0:01.00	0:04.68	6:00.29	19:41.70
K2	0:00.08	0:00.09	0:00.11	0:00.15	0:00.25	0:00.61	0:02.58	2:59.77	11:44.01
PERMUTAT : P1	0:00.08	0:00.08	0:00.08	0:00.08	0:00.16	0:00.57	0:02.38	1:10.30	5:06.45
PLATEAUX : L1	0:00.01	0:00.05	0:00.05	0:00.05	0:00.05	0:00.07	0:00.23	0:03.81	1:02.02
L2	0:00.03	0:00.06	0:00.06	0:00.06	0:00.07	0:00.09	0:00.23	0:04.03	1:59.78
DYCHO : D1	0:00.02	0:00.03	0:00.03	0:00.03	0:00.02	0:00.03	0:00.05	0:00.10	0:00.18
D2	0:00.02	0:00.03	0:00.04	0:00.04	0:00.05	0:00.05	0:00.05	0:00.06	0:00.16

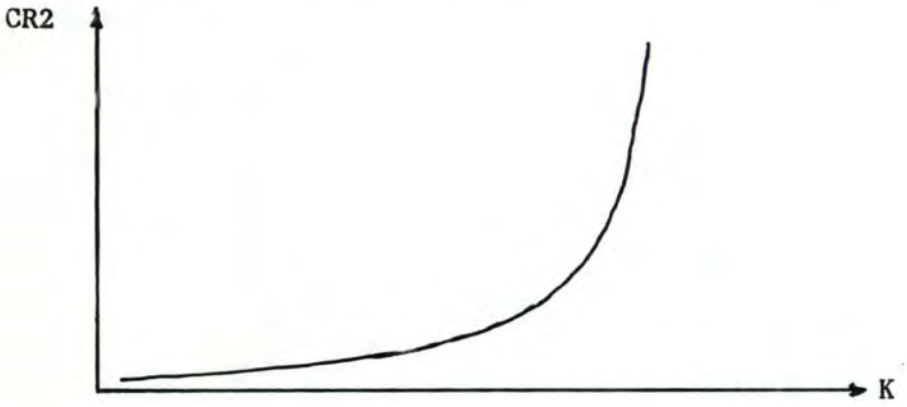
les temps marqués par * dans TRIVECT sont ceux de K = 150

tableau VI : temps d'exécution avec nmot fixé (à 10) et une variation de K pour la représentation compactée.

PROGRAMMES L1 \ K	5	10	15	20	30	50	100	500	1000
TRIVECT : T1	0:00.00	0:00.01	0:00.01	0:00.01	0:00.01	0:00.01	0:00.05	0:02.00	0:07.01
T2	0:00.00	0:00.00	0:00.00	0:00.00	0:00.00	0:00.00	0:00.03	0:02.00	0:08.12
T3	0:00.00	0:00.01	0:00.01	0:00.01	0:00.01	0:00.00	0:00.05	0:02.20	0:08.19
COMPACT : C1	0:00.01	0:00.01	0:00.01	0:00.01	0:00.02	0:00.02	0:00.10	0:00.60	0:01.28
C2	0:00.00	0:00.00	0:00.00	0:00.00	0:00.00	0:00.00	0:00.00	0:00.01	0:00.04
C3	0:00.00	0:00.00	0:00.01	0:00.01	0:00.01	0:00.03	0:00.10	0:00.66	0:01.26
C4	0:00.00	0:00.00	0:00.00	0:00.03	0:00.03	0:00.04	0:00.13	0:00.62	0:01.40
COMMUNE : M1	0:00.00	0:00.00	0:00.00	0:00.01	0:00.02	0:00.02	0:00.02	0:00.05	0:00.11
CONTRACT : K1	0:00.00	0:00.00	0:00.01	0:00.01	0:00.02	0:00.02	0:00.02	0:00.15	0:00.30
K2	0:00.00	0:00.00	0:00.00	0:00.00	0:00.00	0:00.00	0:00.01	0:00.01	0:00.05
PERMUTAT : P1	0:00.00	0:00.00	0:00.00	0:00.00	0:00.00	0:00.01	0:00.06	0:00.30	0:00.50
PLATEAUX : L1	0:00.00	0:00.00	0:00.00	0:00.00	0:00.00	0:00.00	0:00.00	0:00.00	0:00.03
L2	0:00.00	0:00.00	0:00.00	0:00.00	0:00.00	0:00.00	0:00.01	0:00.01	0:00.02
DYCHO : D1	0:00.00	0:00.00	0:00.00	0:00.00	0:00.00	0:00.00	0:00.00	0:00.00	0:00.02
D2	0:00.00	0:00.00	0:00.01	0:00.00	0:00.01	0:00.00	0:00.01	0:00.01	0:00.01

tableau VII : temps d'exécution des programmes sans assertions avec une variation de K.

On voit dans les tableaux V, VI et VII que le rapport (t2 - t1) / t1 devient de plus en plus important et accroît d'une manière généralement exponentielle pour un petit accroissement de K :



Par exemple :

- pour T3, en compactée, pour K = 10, CR2 était 65;
  pour K = 100, CR2 = (8:15.67 - 0:00.05) / 0:00.05 = 9912.4
- pour M1, en étalée, pour K = 10, CR2 était 27;
  pour K = 1000, CR2 = (24:06.06 - 0:00.11) / 0:00.11 = 13145.

Dans les deux structures de représentation :

* la plupart des programmes qui construisent un grand nombre de suites et / ou d'ensembles (cfr COMMUNE, CONTRACT, PERMUTAT) accusent des temps d'exécution énormes aux alentours de K = 500 (K = 50 nmot);

* les programmes qui construisent un très grand nombre de structures "ensuite" accusent des temps d'exécution mauvais déjà autour de K = 100 (K = 10 nmot).
  C'est le cas de TRIVECT (surtout qu'il s'agit en plus d'un tri simple);

* les programmes qui construisent relativement peu de suites et / ou d'ensembles (cfr DYCHO, COMPACT) donnent encore des temps de réponse acceptables jusqu'à K = 1000 ou 2000 (K = 100 nmot ou 200 nmot);

D'une manière générale, on peut donc dire que les temps de réponse commencent à devenir mauvais (1 minute) au delà de K = 10 nmot.

Il n'a pas été utile d'ajouter dans ce mémoire tous les autres résultats provenant des autres valeurs de K (15, 20, 30, 50, 100, 500 et 1000). Cela aurait été long et fastidieux à lire.
Tous les traits essentiels et généraux ont été, nous l'espérons, mis en évidence avec la publication des résultats de K = 10.
Rappelons finalement que plus de 6000 tests ont été faits durant notre campagne de mesures,
nous croyons donc que nous pouvons accorder une certaine confiance à à ces chiffres et aux conclusions qui en résulteront car aucune représentation n'a été privilégiée dans un sens ou dans un autre.

C. METHODE DE CHOIX

Puisque notre but est de trouver la (les) représentation(s) pouvant nous donner satisfaction selon les 5 critères cités et qui n'ont pas tous le même poids de décision, on aurait pu utiliser une des méthodes d'analyse multicritère, ELECTRE I par exemple ((5)), mais avec les chiffres que nous avons, nous pouvons remarquer que cela n'en vaut pas la peine.

Ca ne serait pas nécessaire de se braquer sur chaque petite différence dans nos tableaux des chiffres.
Ce qui est intéressant, c'est de cerner la tendance générale, celle-ci nous est fournie par le tableau II (page 93), tableau qui résume d'une manière éclatante tout notre travail de test :

- La représentation compactée est à coup sûr celle qui consomme le moins de place mémoire.
  On peut, en effet remarquer que même si l'on fournit des tableaux où il n'y a pas de succession remarquable à un programme L2 (dont l'input est constitué des tableaux), la représentation compactée consomme en général moins de mémoire car souvent indépendamment des tableaux à l'entrée, dans la plupart des programmes, on trouve beaucoup d'assertions générant des représentations des suites et des ensembles au moyen des expressions [bi..bs] et {bi..bs} (bi et bs étant des expressions arithmétiques). Et si ces intervalles sont (très) longs, la consommation de la représentation compactée en mots de mémoire sera toujours la plus petite.

  Ce qui s'est passé avec COMPACT (cfr C1 et C4 pages 87 et 88) en ce qui concerne cette consommation est très spécial (consommation en compactée > consommation en étalée); en effet, si on regarde ce programme (L2), on ne manquera pas de voir que dans la boucle principale, quel que soit le tableau fourni à l'input, la représentation compactée n'utilise presque pas ses instruments d'économie de la mémoire ("di" et "dr") :

par exemple,
la compactée de (1, 1, 1, 1, 1) est la suite (5, 1), qui ne constitue pas une succession remarquable,
la compactée de la suite (5, 6, 7, 8) est la suite
(1, 5, 1, 6, 1, 7, 1, 8), ce qui ne constitue pas non plus une succession remarque; dans ces deux cas, la consommation en compactée est la même que celle de la représentation étalée;
mais la compactée de la suite (1, 2, 3, ...) consommera presque toujours plus de mots mémoire en représentation compactée qu'en représentation étalée.

- Avec les test publiés dans ce mémoire (K = 10), les différences dans les temps d'exécution ne sont pas perceptibles d'une manière franche;
bien que l'écart-type des valeurs des critères CR1 et CR2 de la représentation étalée soit plus important que celui des valeurs des mêmes critères dans la représentation compactée, on peut dire que ces 2 représentations sont équivalentes.
Toutefois, leurs plus mauvaises valeurs (des critères CR1 et CR2) se trouveront systématiquement dans la représentation étalée.
Mais, avec l'augmentation de K, on peut remarquer que ces différences deviennent perceptibles (à part le phénomène du programme COMPACT, cfr les colonnes K = 1000 des tableaux V et VI) et la représentation compactée affiche systématiquement le temps de réponse le moins mauvais.

Finalement, nous croyons que ce qui fait la puissance de la représentation compactée c'est la capacité quasi-permanente de pouvoir traiter TOUTE ou PARTIE de la représentation de la suite ou de l'ensemble sans faire d'accès supplémentaires à la mémoire, cela représente, si les autres paramètres dont dépend le temps de réponse restent stables, un gain de temps appréciable par rapport à la représentation étalée, surtout si les successions remarquables sont longues;
alors que pour la représentation étalée, on doit, pour pouvoir disposer des éléments de la structure faire autant d'accès que la longueur du tableau "elem".

BIBLIOGRAPHIE

((1)) : DEFINITION D'UN LANGAGE DE PROGRAMMATION PERMETTANT L'EXPRESSION D'ASSERTIONS, D. FISETTE (Mémoire et Annexes), 1985

((2)) : Notes du Cours "THEORIE DE LA CALCULABILITE", H. LEROY, 1985

((3)) : Notes du Cours "QUESTIONS SPECIALES DE COMPILATION", H. LEROY, 1985

((4)) : DESIGN AND ANALYSIS OF ALGORITHM, Ano, Hopcroft, Ullmann, Addison Wesley, 1974

((5)) : Notes du Cours "ANALYSE MULTICRITERE", Jean FICHEFET, 1984

((6)) : Notes du Cours "PERFORMANCES ET MESURES", M. NOIRHOMME, 1984

((7)) : STRUCTURED PROGRAMMING, O. J. Dahl, E. W. Dijkstra, Hoare, Academic Press, London 1972

((8)) : DEFINITION DU LANGAGE LSD80, Baudouin LECHARLIER

((-)) : COMPUTER DB ORGANIZATION, J. MARTIN