DE NANCY I
EMATIQUES

CENTRE DE RECHERCHE EN INFORMATIQUE
de NANCY

CONSTRUCTION ET TRANSFORMATIONS D'ALGORITHMES ITERATIFS

THESE

présentée pour l'obtention du diplôme de
DOCTEUR INGENIEUR EN INFORMATIQUE

par

Jeanine SOUQUIERES

SOUTENUE LE 26 MARS 1982
DEVANT LA COMMISSION D'EXAMEN

M. J.P. FINANCE — Président

Mme F. BELLEGARDE — Examinateurs
M. P.C. SCHOLL
M. A. VAN LANSWERDE

UNIVERSITE DE NANCY I
UER de MATHEMATIQUES

CENTRE DE RECHERCHE EN INFORMATIQUE
de NANCY

CONSTRUCTION ET TRANSFORMATIONS D'ALGORITHMES ITERATIFS

THESE

présentée pour l'obtention du diplôme de
DOCTEUR INGENIEUR EN INFORMATIQUE

par

Jeanine SOUQUIERES

SOUTENUE LE 26 MARS 1982
DEVANT LA COMMISSION D'EXAMEN

M. J.P. FINANCE	Président
Mme F. BELLEGARDE	Examinateurs
M. P.C. SCHOLL	
M. A. VAN LANSWERDE	

82 -T- 023

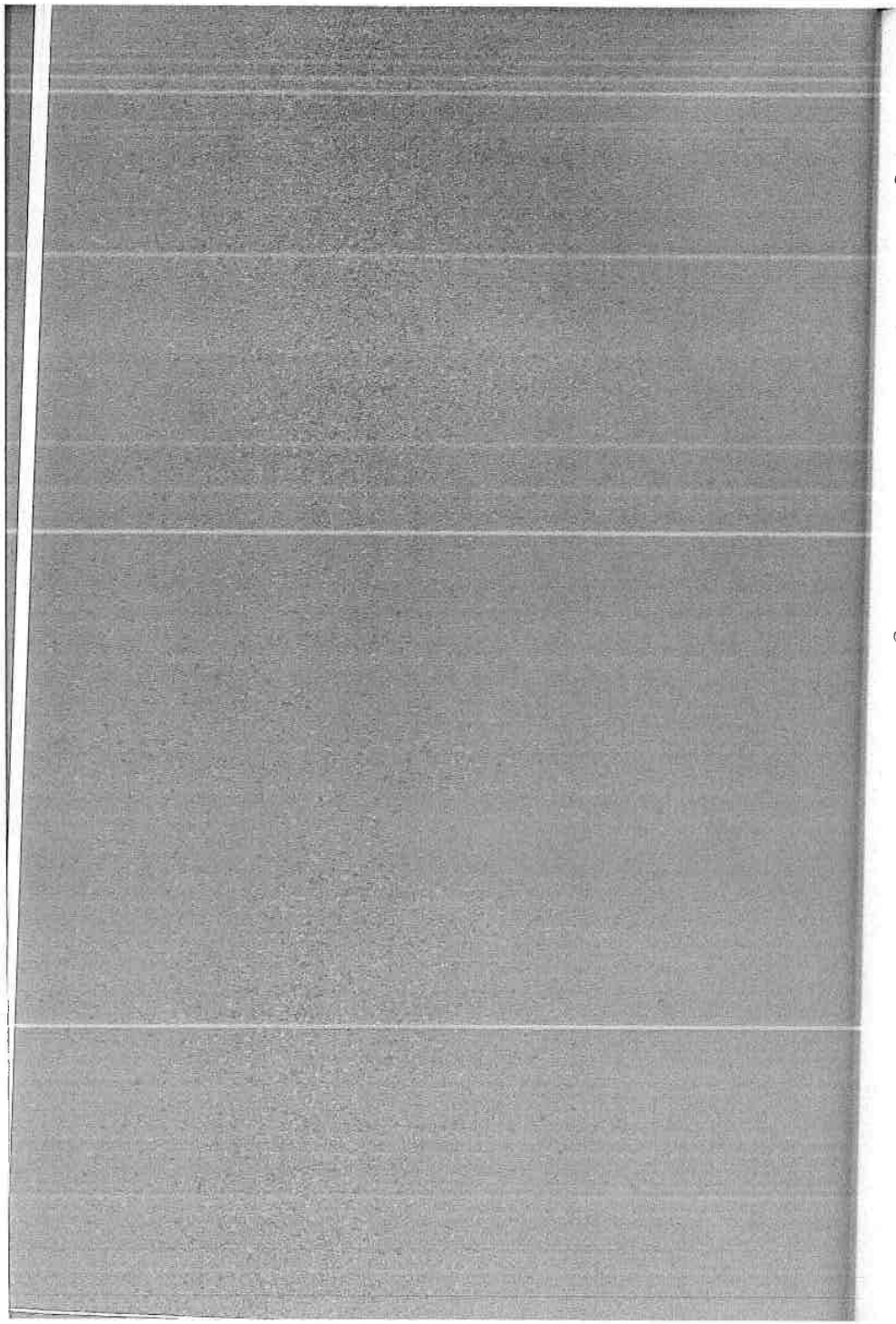
Je tiens à remercier toutes les personnes qui m'ont aidée dans la réalisation de ce travail et en particulier Jean Pierre FINANCE et tous les membres de l'ex-équipe CASTOR.

- RESUME -

Afin de valider la méthode déductive sur des problèmes réels, nous proposons une extension du langage MEDEE existant en lui adjoignant des objets de type SUITE et des opérations associées. La méthode de construction d'algorithmes proposée repose alors sur l'introduction de suites intermédiaires permettant de décomposer le problème à résoudre en deux sous-problèmes indépendants. L'algorithme final est obtenu par transformations successives.

Cette phase de transformations est automatisable: une maquette du système de transformations est actuellement utilisable.

- SOMMAIRE -



SOMMAIRE

Chapitre 4 -SYSTEME ET PROLONGEMENTS

- CHAPITRE 1 -

LA CONSTRUCTION DES PROGRAMMES

CHAPITRE I

LA CONSTRUCTION DES PROGRAMMES

I - "ETAT DE L'ART"

Aux premiers temps de l'informatique, la programmation a été une activité assez aléatoire. Les recherches théoriques ont alors longtemps porté sur l'amélioration des langages existants, sur la création de nouveaux langages mieux adaptés à la spécificité de classes de problèmes bien précises, sur la création de langages "de plus haut niveau". Il s'est révélé que l'activité de programmation (construction de programmes efficaces, maintenance et modification de ces programmes, fiabilité, développement de gros logiciels...) ne dépendait pas seulement de l'utilisation de langages toujours plus sophistiqués mais nécessitait l'utilisation de véritables théories. Le besoin d'une programmation systématique, méthodique s'est très vite fait sentir et a conduit au développement de méthodes de programmation.

Les débuts de ces nouvelles recherches en informatique sont marqués par le développement de la notion de preuves de programmes [FLO, 67] [HOA, 69]. D'une manière générale, ces preuves consistent à vérifier, par un calcul formel, que le programme obtenu correspond bien aux spécifications de l'énoncé de départ.

La décomposition de l'activité de programmation suivant le schéma :

énoncé informel ——————> énoncé formel ——————> algorithme ——————> programme

explique la diversité des axes de recherches développés.

En amont de la programmation, on vient développer des recherches sur la spécification de problèmes [ABR, 78], [LIS, 79]... A partir d'un énoncé de problème, relativement informel et peu précis, elles visent à en donner une spécification précise de laquelle une solution pourra être dérivée. Pour ce faire, elles utilisent un cadre formel, faisant par exemple appel au langage ensembliste (dans les langages SETL [KEN, 75], Z [ABR, 78]) au concept de "type abstrait" (le langage CLU [LIS, 75]), au calcul des prédicats...

Le passage d'un énoncé formel à un algorithme a suscité trois types de recherches complémentaires [PAI, 78] :

- Des recherches axées sur la théorie des calculs et des programmes (cf. les travaux de Nivat, Milner et Scott...). Elles visent à préciser le concept de programme.

- Des recherches axées sur les structures de données manipulées par un programme (travaux de Guttag, Gaudel, Rémy...).

- Des recherches sur la méthodologie de la programmation.

Les diverses approches méthodologiques existantes reposent sur le concept de "bon programme", concept difficile à cerner. Généralement, on associe à "bon programme" une liste de qualificatifs parmi lesquels on peut citer : clair, correct (dont on peut prouver qu'il satisfait aux spécifications du problème initial), structuré, modulaire, efficace, bien documenté, fiable ... Un certain nombre de ces propriétés pouvant être contradictoires (par exemple fiabilité et efficacité, clarté et efficacité, ...) il est difficile de les satisfaire simultanément lors d'une même étape de résolution.

Parmi les diverses approches possibles pour obtenir un premier algorithme (ou premier programme), citons :

1 - La programmation structurée et descendante.

Cette démarche consiste à travailler par raffinements successifs, en partant d'une idée générale du traitement et à le détailler et décomposer de plus en plus finement. A cette démarche sont associés différents types de décomposition de problèmes et outils linguistiques de recomposition des solutions (ex. à l'analyse par cas correspond la conditionnelle).

Présentons quelques méthodes issues de ces idées :

- la programmation structurée ([ARS, 77], [WIR, 76], [DIJ, 76]) dont l'idée essentielle est la décomposition progressive de la résolution d'un problème en sous-problèmes élémentaires. Elle permet d'utiliser des connaissances acquises en se référant à des problèmes déjà traités. Les informations qu'elle utilise ne sont pas toujours déterminées à priori et peuvent aussi être définies par raffinements successifs. Les méthodes proposées en Informatique de Gestion : CORIG, GAP, LCP [WAR, 73] ou langage de construction de programmes, la méthode proposée par Jackson, [JAC, 77] pour des problèmes plus généraux sont considérées comme des applications de la programmation structurée.

- la programmation par décomposition ou structuration proposée par Scholl [SCH, 79] repose sur deux principes simples :

- raffinement de l'information en un ensemble d'informations élémentaires et structuration de cet ensemble en se servant d'une des deux structures connues (à savoir la structure de file ou la structure d'arbre) ;

- analyse du problème posé en utilisant la structure déterminée précédemment.

- la programmation déductive proposée par PAIR [PAI, 77]. Elle aide l'utilisateur à se poser les bonnes questions lors de la résolution d'un problème, questions déduites de l'énoncé. Elle consiste à partir de la spécification des résultats et à les

définir en fonction d'intermédiaires, eux-mêmes considérés comme résultats de sous-problèmes. La même démarche est appliquée à ces intermédiaires jusqu'à n'avoir plus pour intermédiaires que des données.

2 - La programmation ascendante

Avec cette démarche, la solution du problème n'est pas envisagée de façon globale. Elle consiste, en dernier lieu, à composer des morceaux de solutions.

3 - La synthèse automatique de programmes, soit à partir d'exemples [JOU, 78], [KOD, 78], soit à partir de spécifications [BIB, 80], [BID, 79].

Afin d'augmenter leur efficacité, certaines de ces méthodes visent à conserver le plus longtemps possible un certain niveau d'abstraction et utilisent les recherches effectuées sur les structures de données [REM, 79]. Elles ont en commun une approche descendante : le problème à résoudre est envisagé de manière globale et le résultat final obtenu par raffinements successifs.

Ce premier algorithme étant écrit, il peut être soit directement exécuté, soit transformé. Lorsque, lors de la construction du premier algorithme, on ne se soucie pas d'écrire directement un algorithme efficace, une deuxième phase est alors nécessaire : c'est l'étape de transformation de programmes. Avant de transformer un programme, il est nécessaire de connaître les qualités requises du résultat cherché. Des études sur la complexité des programmes [AHO, 74] fournissent un certain nombre de critères d'améliorations.

II - ENVIRONNEMENT NANCEEN

II - 1 - Recherches effectuées

Avec les méthodes de programmation structurée, [WIR, 77], une étape importante a été franchie vers la systématisation de l'activité de programmation. Le but de ces méthodes n'est pas seulement de présenter un algorithme mais de permettre de le découvrir petit à petit. Procéder par raffinements successifs en partant d'une idée générale du traitement, en le détaillant et en le décomposant de plus en plus finement, évite de se noyer, dès l'abord, dans trop de détails. Avec ces méthodes descendantes se sont développés les concepts de modularité, de décomposition d'un problème en sous-problèmes plus simples, de documentation d'un algorithme. Malheureusement, la plupart de ces méthodes adoptent un point de vue dynamique en liant dès la genèse de l'algorithme la résolution d'un problème à son exécution. Or, l'une des difficultés de l'activité de programmation est due au fait que l'ordre de construction des définitions d'un algorithme est la plupart du temps différent de l'ordre de calcul.

Un effort supplémentaire a été fait avec la méthode déductive [PAI, 77] en évitant de se préoccuper de l'ordre d'exécution au moment de l'écriture de l'algorithme. Cette méthode va plus loin que les méthodes structurées classiques : elle permet, dans un premier temps, de faire abstraction de toute notion de calcul pour aboutir à une description axiomatique ou définitionnelle de l'algorithme résolvant

le problème posé. Notons que ce sont les questions posées ("Quel est le résultat cherché ?") plus que les réponses qui sont obtenues de manière déductive.

Ces recherches méthodologiques développées à Nancy depuis 1974 ont eu pour origine essentielle des motivations pédagogiques : elles correspondaient à un soucis dans la structuration et la progressivité de l'activité de programmation et dans son enseignement. Dans un premier temps, l'aspect structures de données n'a pas été approfondi et les seules structures utilisées correspondaient aux structures simples utilisées dans les langages de programmation existants.

Cette méthode s'est accompagnée de la définition d'un langage support, le langage MEDEE, pour lequel un compilateur, appelé SNOOPY, a été implementé [HUC, 77].

Cet outil, le langage MEDEE et le compilateur, est utilisé dans l'enseignement de la programmation et a fait l'objet d'une évaluation [KOL, 79] qui a porté sur deux aspects :

- sur la méthode de programmation, méthode qui présente les notions fondamentales en les décomposant (le traitement séquentiel, le traitement conditionnel et le traitement itératif) ;

- sur la méthode d'apprentissage de la programmation qui propose une progression pédagogique pour l'acquisition des notions.

Parallèlement aux réflexions pédagogiques sur cette méthode de programmation, plusieurs axes de recherche se sont développés dans le but de systématiser la construction des programmes. Une définition sémantique du langage MEDEE a été donnée, [FIN 78], [LES, 78], ainsi qu'une méthode de localisation d'erreurs [BEL, 78] fondée sur le parcours de l'arborescence des modules constituant un programme MEDEE et sur l'utilisation de règles de preuves formelles. La définition d'un cadre permettant d'exprimer les structures de données dans un problème (cadre algébrique ou logique), [FIN, 78], [REM, 78] a apporté une aide importante à la mise en forme d'un problème. Elle a permis d'exprimer clairement les choix de représentation [REM, 78] et ainsi d'améliorer l'efficacité du programme final.

En amont de la programmation, des travaux sur la spécification de problèmes ont été développés [FIN, 79]. Actuellement, des réflexions sont menées sur le passage systématique d'une spécification à un algorithme [FIN, 79].

Parmi les diverses recherches sur la méthode déductive, signalons la construction d'un éditeur syntaxique permettant la manipulation de programmes MEDEE ainsi que la conception d'un système d'aide à la construction déductive d'algorithmes [GUY, 80].

Plus proche de nos préoccupations, citons aussi les travaux de Françoise BELLEGARDE [BEL, 81], dont le but est l'obtention d'un système d'aide à la transformation des programmes. Cette étude diffère de la nôtre par son cadre théorique : elle utilise les techniques de la programmation fonctionnelle. Tout programme est exprimé par une expression fonctionnelle sur laquelle sont effectuées les transformations. La preuve de la correction des transformations est liée à la théorie du type abstrait fonction. Une automatisation partielle des manipulations des expressions fonctionnelles est rendue possible par l'utilisation d'un ensemble de théorèmes considéré comme un système de règles de réécriture.

II - 2 - Présentation succinte de la méthode déductive

II - 2 -1- Concepts

Le langage MEDEE est un langage de définitions se distinguant d'un langage de programmation par ses aspects statiques : il permet une description de la relation entre résultats et données plutôt qu'une description des calculs, et ses aspects méthodologiques : il permet la résolution progressive d'un problème.

Les idées développées : langage adapté à l'expression d'énoncés récurrents, méthode de construction de programmes sous-jacente sont voisines de celles développées par ARSAC [ARS, 77], dans LUCID par ASHCROFT [ASH, 77], par SCHOLL [SCH, 79]. Alors que LUCID est orienté vers les preuves de programmes, MEDEE est plus orienté vers la construction.

Un algorithme ou un programme MEDEE est un ensemble de modules dont la structure est arborescente. Un module se décompose lui-même en un ensemble de définitions. Dans le langage MEDEE initial [HUC, 77], on retrouve les trois schémas classiques de définitions : définitions simples, définitions conditionnelles et définitions itératives. Les définitions itératives ont deux formes possibles :

a) uf : IDEF ; pour i dans 1 ⟶ n répéter DEF

Sémantiquement, une telle définition consiste à spécifier uf comme le dernier terme d'une suite (ui) définie sur un intervalle d'entiers (ici 1 ⟶ n).
(En fait uf = dernier ui).
-DEF désigne le module ou ensemble de définitions qui associe à i appartenant à l'intervalle 1 ⟶ n, l'élément ui.
-IDEF est le module d'initialisation (il définit éventuellement uo et les objets définis par récurrence dans DEF).

b) Une forme d'itération plus générale :

uf : IDEF ; jqa arret répéter DEF

Par opposition avec l'autre forme d'itération, la suite (ui) n'est pas définie à partir d'une suite guide (dans la forme précédente, chaque élément ui de la suite (ui) est défini en bijection avec i, élément de la suite d'entiers 1⟶ n).

Cette forme d'itération conduit l'utilisateur à penser en même temps à la définition de la suite domaine de définition de la suite (ui) et à la manière de définir cette suite.

On devra définir, dans IDEF l'accès au premier élément de cette suite et dans DEF la fonction de succession. L'inconvénient dans cette forme d'itération est qu'elle ne met pas en évidence la suite guide ou domaine de définition du résultat, lorsqu'elle existe.

II - 2 -2- Application sur un exemple

Le drapeau tricolore.

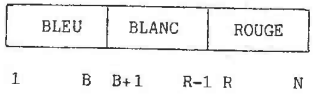
On dispose d'un tableau à N cases. Chaque case du tableau contient

une bille. Ecrire un algorithme qui entraîne une réorganisation du tableau telle que les billes soient dans l'ordre du drapeau français.

Remarque : Le tableau peut contenir des billes de trois couleurs BLEU, BLANC, ROUGE mais rien ne garantit que les trois couleurs sont toutes représentées.

Analyse du problème :

Schématisons la situation finale cherchée :



B et R sont des entiers vérifiant les propriétés suivantes :

(1) $0 \leq B < R \leq N+1$

(2) pour tout i tel que $1 \leq i \leq B$ couleur (i) = BLEU

(3) pour tout i tel que $B < i < R$ couleur (i) = BLANC

(4) pour tout i tel que $R \leq i \leq N$ couleur (i) = ROUGE

La propriété (1) indique qu'il y a au minimum une seule couleur de bille.

A partir des propriétés décrites précédemment, on peut définir un invariant en découplant deux des conditions précédentes grâce à l'introduction d'une variable supplémentaire D. D vérifie la propriété :

$$0 \leq Bi \leq Di < Ri \leq N+1$$

Cet invariant qui caractérise une suite d'indices Bi, Di, Ri et de tableaux ti est facile à vérifier à l'initialisation. Par exemple :

$Bo = 0$

$Do = 0$

$Ro = N+1$

avec pour paramètre to de la suite de tableaux, le tableau donné DTRI.

Résoudre ce problème revient à déterminer Bi, Di, et Ri tels que après i manipulations de billes:

Bi représente le nombre de billes bleues déjà classées

Ri-N le nombre de billes rouges déjà classées

et Di le nombre de billes bleues plus le nombre de billes blanches rencontrées.

Schématiquement, à l'étape i, le tableau des billes se présente sous la forme :

Le résultat est atteint lorsque $Ri = Di+1$

Remarque : La notation ∂ est utilisée pour les valeurs récurrentes (ancienne valeur de).

Algorithme :

* DRAPEAU TRICOLORE

- résultat (édit) impression du drapeau tricolore - DTRI(tableau 1...N) des couleurs - N (entier) nombre de billes - arret (booléen) indiquant si le résultat est atteint	2 1	résultat = écrire DTRI DTRI : IN ; jqa arret répéter PERMUT	- IN initialisation de DTR et des valeurs recurrent de PERMUT - PERMUT détermine Bi, Di, pour comparaison et écha
*PERMUT			
- B (entier) nombre de billes bleues déjà classées - D (entier) nombre de billes bleues plus nombre de billes blanches déjà rencontrées - R (entier) position dans le tableau de la dernière bille rouge classée	2 3 1	DTRI, B, D, R : si ∂DTRI [I] = 'bleu' alors CBLEU si ∂DTRI [I] = 'blanc' alors CBLANC sinon CROUGE arret = (D+1 = R) I = ∂ D+1	- CBLEU traite le cas d'u bille bleue (la positionne) - CBLANC traite le cas d' bille blanche CROUGE traite le cas d' bille rouge
* C BLEU			
	1 2 3 4	B = ∂ B+1 D = ∂ D+1 R = ∂ R DTRI = échange(∂DTRI, B, I)	- échange fonction qui échange dans le tableau DTRI les éléments d'indi B et I : DTRI [B] = ∂DTR [I] DTRI [I] = ∂DTR [B] DTRI [J] = ∂DTR [I] pour tout J ≠ B, J ≠ I
* C BLANC			
	1 2 3 4	B = ∂ B D = ∂ D+1 R = ∂ R DTRI = ∂ DTRI	
* C ROUGE			
	1 2 3 4	B = ∂ B D = ∂ D R = ∂ R-1 DTRI = échange(∂DTRI, R, I)	
* INIT			
		N, DTRI = donnée arret = faux B = 0 D = 0 R = N+1	

Remarques :

La présentation modulaire de l'algorithme fait apparaître de gauche à droite les définitions lexicales des identifications d'objets, la colonne d'ordonnancement des définitions, les définitions algorithmiques et les définitions lexicales des identificateurs de module.

Dans le module *PERMUT, la définition numérotée 3 a été obtenue après fusion des 4 définitions suivantes : ce sont des définitions conditionnelles ayant même domaine de définition.

```
DTRI : si  ∂DTRI [I] = 'bleu' alors DTRI = échange (∂DTRI,B,I)
       si  ∂DTRI [I] = 'blanc' alors DTRI = ∂DTRI
                               sinon DTRI = échange (∂DTRI,R,I)
B : si  ∂DTRI [I] = 'bleu' alors B = ∂B+1 sinon B = ∂B
D : si  ∂DTRI [I] = 'rouge' alors D = ∂D sinon D = ∂D+1
R : si  ∂DTRI [I] = 'rouge' alors R = ∂R-1 sinon R = ∂R
```

III - LIMITES DU LANGAGE MEDEE

III - 1 - Quelques exemples

Proposons ici quelques exemples de problèmes pour lesquels les limitations imposées par le langage MEDEE initial conduisent à une résolution assez lourde.

a) Les ateliers (cas de "rupture" en analyse de gestion)

Considérons le fichier des employés d'une entreprise. Chaque employé est caractérisé par son nom, son numéro d'atelier et son numéro de catégorie. Sachant que ce fichier est trié par ordre croissant d'atelier, puis par numéro croissant de catégorie, au sein d'un même atelier et par nom (le classement est effectué suivant l'ordre alphabétique), on demande l'impression de la liste des ateliers suivie de l'effectif total de l'entreprise. On précise, de plus, que chaque atelier du résultat demandé se compose :

- de son numéro d'atelier
- de la liste des catégories de l'atelier considéré
- de l'effectif de cet atelier

et chaque catégorie de chaque atelier se compose elle-même :

- de son numéro de catégorie
- de la liste des personnes de cette catégorie
- de l'effectif de cette catégorie.

b) Donner le nombre d'occurences d'un texte T1 contenu dans un texte T.

c) Considérons un texte terminé par le caractère ".". On demande d'imprimer le nombre de mots palindromes contenus dans ce texte et les palindromes eux-mêmes (un mot est palindrome s'il est égal à son miroir).

d) Même chose pour les ensembles de mots palindromes.

e) Traitement de chaînes de caractères (FINANCE, 1979).

Compter le nombre de mots de longueur 1, 2, ... 20 d'un texte donné. On supposera que 20 est la longueur maximum d'un mot, que deux mots sont séparés par un caractère blanc et que le texte se termine par le caractère "." précédé d'un blanc.

f) Les télégrammes (dû à HENDERSON et SNOWDON, 1972)

On veut traiter une suite de télégrammes. Le texte donné se présente sous la forme d'une suite de caractères transférés par blocs de longueurs fixes. Chaque bloc contient un nombre entier de mots, les mots étant séparés par le mot spécial ZZZZ et le texte donné se termine par le télégramme vide (ou télégramme sans mot).

On demande d'imprimer la liste des messages issus du texte donné, un message se composant d'un télégramme, du nombre de mots comptables (ZZZZ et STOP ne sont pas pris en compte) et du nombre de mots trop longs.

On demande à ce que, dans chaque télégramme, STOP soit remplacé par ".", les séquences de blancs remplacées par un seul blanc et les mots trop longs (ayant plus de 12 caractères) soient tronqués.

g) Sous-suites

Chercher la longueur maximum d'une sous-suite croissante extraite d'une suite finie donnée.

h) Exemple (dû à ABRIAL)

Soit un fichier f dont chaque enregistrement est formé :

- d'une clé
- d'une longueur lgn (lgn $\geq$ 0, lgn entier)
- d'une suite de lgn valeurs numériques v.

Imprimer pour chaque enregistrement de f :

- une liste de lgn lignes : une ligne est associée à une valeur de l'enregistrement et contient le couple (clé, valeur)
- une ligne finale contenant le couple (clé, somme des lgn valeurs de l'enregistrement)

On demande detenir compte d'un saut de page toutes les 4 lignes.

i) Mise à jour de fichiers

On dispose du fichier des stocks. C'est un fichier séquentiel trié par ordre croissant des numéros de produits (un enregistrement contient le numéro du produit et la quantité en stock). On dispose aussi du fichier majstock des mises à jour à effectuer. C'est un fichier séquentiel trié par ordre croissant des numéros de produit (un enregistrement contient trois champs : le numéro du produit, un code de mise à jour et une quantité Q).

Si code = C, il s'agit d'un nouveau produit et Q désigne la quantité en stock.

Si code = M, il s'agit d'une modification de stock, Q désigne la quantité à rajouter au stock (Q pouvant être négatif).

Si code = S, on supprime le produit et il n'y a pas de quantité Q.

On demande la mise à jour du fichier des stocks, sachant qu'il peut y avoir plusieurs modifications pour un même produit, l'impression de la liste des nouveauxproduits et celle des produits supprimés.

j) Reprendre l'énoncé de la mise à jour précédente en imposant de n'effectuer les modifications que lorsqu'on est sûr que le produit traité n'a pas été supprimé.

k) Etat d'avancement des livraisons sur les commandes en cours

Disposant du fichier des commandes en cours trié par numéro de commande, du fichier de livraison, on demande de mettre à jour le fichier des commandes et d'imprimer l'état des commandes.

Description des fichiers :

. fichier commandes en cours qui contient :

- numéro de commande
- type (renseignement sur la commande)

si type = 1,
- date commande
- numéro du client
- nombre d'articles commandés
- renseignements divers

si type = 2 (correspond au cas où il existe un enregistrement par article commandé, les numéros d'articles sont classés par ordre croissant)
- numéro d'article
- quantité commandée
- référence article

si type = 3 (un enregistrement correspond à une livraison)
- quantité livrée
- date de livraison
- renseignements divers

si type = 4 (récapitulatif)
- quantité restant à livrer
- date limite de livraison

. le fichier de livraison contient :

- numéro de commande
- numéro d'article
- quantité livrée
- date de livraison
- renseignements divers.

Ce fichier est trié sur le numéro de commande, le numéro d'article et la date de livraison.

. fichier commandes mises à jour a la même structure que le fichier des commandes en cours. Il est obtenu à partir des commandes en cours en tenant compte des livraisons (enregistrement de type 3) et en créant ou modifiant l'enregistrement (type 4).

. sortie imprimante : pour chaque commande, on imprime un en-tête et par article :

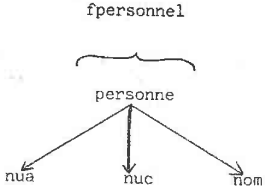
- numéro d'article
- quantité commandée
- total des quantités livrées
- nombre de livraisons
- quantité restant à livrer
- date limite de livraison.

1) A partir d'une suite d'entiers donnée, constituer la suite des nombres pairs et la suite des nombres impairs qui la constituent.

III - 2 - Une caractéristique de ce type de problèmes : "les ruptures"

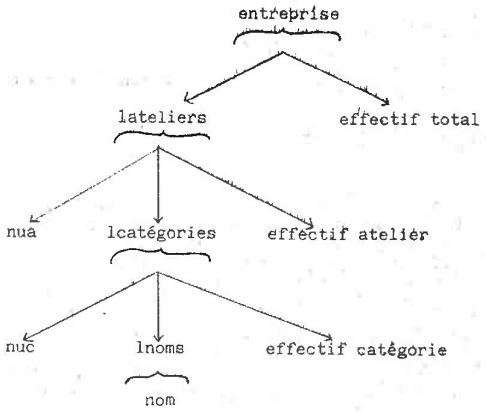
1 - Considérons l'exemple a, les ateliers.

La structure du fichier du personnel peut être schématisée par :



où fpersonnel désigne le fichier du personnel composé d'une suite de personne, chaque personne étant définie par le triplet : nua (pour numéro d'atelier), nuc (numéro de catégorie), nom.

La structure du résultat peut être schématisée par :



La structure du résultat nous invite à introduire trois itérations emboîtées : la plus externe définie sur la suite des ateliers, une autre sur la suite des catégories et la dernière sur la suite des personnes de la catégorie.

Or, dans la suite donnée fpersonnel, tous les éléments sont au même niveau : c'est une suite linéaire "à plat". Cette suite est dite à ruptures :

- rupture sur les ateliers : elle a lieu lorsqu'en passant d'un élément au suivant, on change de numéro d'atelier.

- rupture sur les catégories : elle a lieu lorsque pour un même atelier, le passage d'une personne à la suivante a lieu avec un changement de numéro de catégorie.

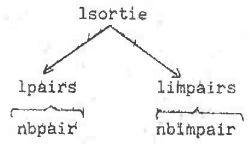
Nous parlerons de critère de rupture :

- le premier critère de rupture consiste en un changement de numéro d'atelier

- le deuxième critère de rupture consiste en un changement de numéro de catégorie.

2 - Considérons l'exemple 1 : constituer la suite des nombres pairs suivie de la suite des nombre impairs.

structure des données



structure des résultats

La suite résultat est une suite à deux niveaux.

Le critère de rupture consiste ici en un changement de parité.

3 - Conclusion

D'une manière générale, les ruptures sont des caractéristiques de la forme des données : la suite donnée est une suite linéaire représentant une suite à plusieurs niveaux.

Les ruptures doivent être prises en compte lorsque un élément de la suite résultat n'est pas défini en bijection avec un élément de la suite donnée mais en bijection d'une sous-suite connexe de la suite donnée.

Aux problèmes de rupture est lié un problème de synchronisation : on doit exécuter simultanément deux ou plusieurs processus qui ne peuvent être synchronisés.

III - 3 - Les limites de MEDEE

Une caractéristique commune aux exemples proposés est que le résultat cherché peut-être défini simplement à partir d'une certaine suite. Dans les cas les plus simples, cette suite sera la suite donnée, mais généralement, elle devra être elle-même définie à partir d'une ou plusieurs suites données:

-Dans l'exemple a, le résultat se définit simplement, si on dispose de la suite des ateliers, cette suite étant constituée de couples (numéro d'atelier, suite des catégories de cet atelier).

-Dans l'exemple e, on peut définir le résultat à partir de la suite des mots du texte.

-Dans l'exemple i, le résultat se déduit simplement de la suite des produits, cette suite étant constituée de couples (enregistrement du fichier stock, suite des mises à jour à effectuer sur le produit correspondant).

L'idée principale retenue ici consiste à introduire des INTERMEDIAIRES de type SUITE bien adaptés à la définition du résultat. Le résultat se définit alors simplement par une itération sur ces suites intermédiaires.

Si la forme classique des définitions itératives introduites initialement dans le langage MEDEE permet d'exprimer tous les algorithmes itératifs, elle est, en contre-partie, de trop "bas niveau" et mal adaptée pour une résolution progressive du type de problème énoncé précédemment.

L'itération, constructeur du type suite, était considérée dans le langage MEDEE comme une fonction associant un objet uf, dernier terme d'une suite (ui) à une suite donnée. De plus, elle ne mettait pas en évidence la suite

guide ou domaine de définition de l'itération, sauf dans le cas où cette suite était un intervalle d'entiers.

Or, de nombreux problèmes informatiques ont pour données des suites linéaires : traitement de textes, traitement de fichiers ... La méthode décrite précédemment ne propose pas de solution simple à ce genre de problèmes essentiellement parce qu'elle utilise des structures de données très pauvres.

IV - OBJECTIFS DE CE TRAVAIL

Notre objectif est de résoudre simplement ce type de problèmes en utilisant la méthode déductive. Nous proposons une extension du langage MEDEE initial en enrichissant les structures de données : le type suite muni des opérations appropriées est considéré comme un type de base.

La définition d'algorithmes à l'aide d'intermédiaires de type SUITE adaptés à la définition du résultat conduit à écrire un premier algorithme peu efficace. Un algorithme final sera obtenu par transformations successives d'algorithmes.

En même temps qu'un langage, nous proposons une méthode de résolution. Le travail présenté ici est un développement et un approfondissement du chapitre 2 - 3 de la thèse de Jean-Pierre FINANCE [FIN, 79]. Il concrétise un certain nombre des idées développées dans la mesure où nous avons réalisé une maquette du système de transformations proposées.

- CHAPITRE 2 -

METHODE ET LANGAGE

CHAPITRE 2

METHODE ET LANGAGE

I - MISE EN OEUVRE DE LA METHODE

Pour résoudre le type de problèmes dont des exemples ont été donnés au Chapitre 1, en restant dans le cadre proposé, nous introduirons des objets intermédiaires de type suite nous permettant de définir simplement le résultat.

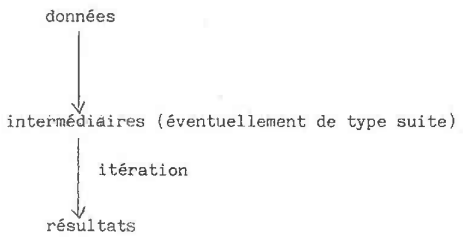
La démarche proposée est composée de deux étapes :

I- 1 - Construction d'un premier algorithme

I- 1 - 1 - Méthode

L'objectif est d'écrire un premier algorithme en utilisant la méthode déductive. Le premier algorithme doit être clair, correct (répondant à la spécification du problème), facile à comprendre et à modifier.

L'une des idées à la base de la méthode déductive consiste à partir de la spécification des résultats. Définir les résultats à partir d'une suite guide comme domaine de définition conduit naturellement à introduire des intermédiaires pouvant éventuellement être de type suite. En effet, il existe très peu de cas où les données du problème puissent être utilisées simplement pour définir le résultat. Dans la plupart des cas, il n'y a pas identité entre les données et les suites nécessaires à la définition du résultat. Schématiquement :



Initialement les intermédiaires (il peut y en avoir plusieurs) n'ont pas d'existence : c'est en ce sens qu'on les appelle aussi objets logiques. Leur caractéristique est qu'ils sont bien adaptés à la définition du résultat. L'intérêt de ces intermédiaires est qu'ils permettent de ramener le problème initial à deux sous-problèmes indépendants :

- l'un définissant les résultats en supposant que l'on dispose des suites intermédiaires adaptées à leur définition. Cette phase correspond à une phase d'abstraction des données : l'algorithme de définition du résultat est développé autour de structures appropriées à sa définition sans tenir compte des structures existantes.

- l'autre définissant ces suites intermédiaires introduites en fonction des données réelles du problème. Cette phase correspond à une phase de représentation : trouver la fonction de représentation des suites logiques intermédiaires en fonction des suites données (en utilisant la structure des résultats comme guide).

Remarque : Nous n'avons rien apporté de nouveau ici au niveau de la méthode utilisée. Seul l'enrichissement des structures de données avec le type suite permet d'aborder un plus large éventail de problèmes.

Comme nous l'avons signalé au Chapitre 1, un système d'aide à la construction d'un tel algorithme est en cours d'implémentation.

I-1 - 2 - Analogie avec des démarches existantes

La démarche consistant à introduire des intermédiaires lorsque les données ne nous permettent pas d'expliciter simplement le résultat a déjà été proposée par Jackson [JAC, 75] et WARNIER [WAR, 72]. Nous développerons ces idées au Chapitre 3.

I- 2 - Version finale de l'algorithme

I-2 - 1 - Selon le cas (problème traité, contraintes matérielles ...), on peut avoir deux attitudes vis à vis des suites intermédiaires introduites :

- soit les représenter effectivement
- soit les faire disparaître.

Afin de rendre l'algorithme obtenu plus efficace, nous adoptons la deuxième démarche : nous concevons l'algorithme final comme le résultat d'un processus de transformations successives d'algorithmes.

La lourdeur de cette démarche qui conduit à faire des manipulations d'algorithmes à la main nous a conduit à automatiser cette phase de transformations.

I-2 - 2 - Travaux existants sur les transformations de programmes

De nombreuses recherches ont été développées dans le domaine des transformations de programmes. Dans les travaux d'Arsac [ARS, 78], Bauer et All [BAU, 78] avec le projet CIP, Feather [FEA, 79], Burstall et Darlington [BUR, 77], l'idée principale correspond à la mise en oeuvre de stratégies générales permettant de passer d'une version récursive à une version itérative. Dans Bellegarde et Cie, [BEL, 79], l'idée générale développée consiste à accélérer le calcul du dernier terme d'une suite en remplaçant cette suite par une suite extraite ayant même dernier élément.

Certains de ces travaux se sont concrétisés par l'implantation de systèmes permettant d'effectuer des transformations de manière automatique ou semi-automatique.

Parmi les systèmes existants, citons le système semi-automatique de Burstall et Darlington : il applique un nombre fixe restreint de règles pour la synthèse et la transformation d'énoncés récursifs. Le système proposé par Feather suggère à l'utilisateur un certain nombre de transformations possibles à partir d'une description de la forme du résultat.

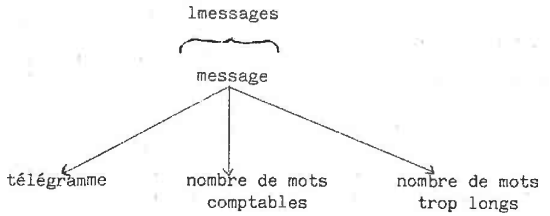
Dans le projet CIP, l'utilisateur guide les transformations en proposant deux schémas de programme : le schéma initial et un schéma final munis des préconditions à satisfaire. L'utilisateur a la possibilité de localiser l'application de certaines transformations et d'introduire ses propres transformations.

II - DEROULEMENT DE LA METHODE SUR UN EXEMPLE:

Prenons l'exemple des télégrammes dont l'énoncé a été donné p.15.

II-1 - Quelles sont les structures nécessaires ?

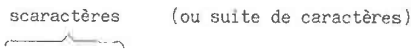
II-1 - 1 - Structure du résultat



lmessages désigne la suite des messages. Chaque message est composé d'un triplet :

- télégramme (le télégramme source dans lequel les mots "STOP" ont été remplacés par ".", le mot final "ZZZZ" est supprimé et les mots trop longs tronqués à partir du 12ème caractère.
- le nombre de mots comptables de ce télégramme.
- le nombre de mots trop longs.

II-1 - 2 - Structure des données



Remarque : Il n'y a pas de compatibilité immédiate entre ces deux structures: lmessage ne peut être déduite simplement de scaractères.

II-2 - Définition du résultat

II-2 - 1 - D'après la spécification du problème, nous pouvons définir le résultat comme l'impression d'une suite de messages ; nommons-la lmessages :

résultat = écrire lmessages

lmessages désigne une suite dont chaque élément est un triplet : télégramme, nombre mots comptables, nombre mots trop longs. Les mots comptables sont tous les mots du texte autres que "STOP" et "ZZZZ" qui sont des séparateurs.

II- 2 - 2 - Comment est définie lmessages ?

Elle peut être définie par une itération dont le domaine de définition est la suite des télégrammes, la suite scaractèrés donnée ne pouvant être directement utilisée. Introduisons la suite intermédiaire de télégrammes notée stels.

lmessage = iter *cmessage sur stels

où : - *cmessage désigne le module associant à chaque élément de stels un élément de lmessage.

- stels désigne la suite intermédiaire introduite dont l'élément courant est le télégramme. La syntaxe de l'itération utilisée fait apparaître une bijection entre la suite lmessage et la suite domaine d'itération (sur).

II- 2 - 3 - Définissons le terme général de lmessage appelé message

D'après la spécification du problème, message se compose d'un télégramme appelé untel, du nombre de mots comptables appelé totmot et du nombre de mots trop longs tlong de ce télégramme.

*cmessage
message = untel, totmot, tlong
totmot, tlong = der (nbmot, nblong)
nbmot, nblong, untel = iter *compte sur tel

Le nombre de mots d'un télégramme est obtenu en comptant les mots du télégramme : il nécessite l'introduction d'un nouvel intermédiaire, le mot. Ceci nous amène à considérer le télégramme comme étant une suite de mots plutôt que comme une suite de caractères.

- untel désigne le télégramme dans lequel on a effectué les modifications nécessaires.

- totmot désigne le nombre total de mots comptables.

- tlong le nombre total de mots trop longs.

- nbmot et nblong les suites de nombre de mots et de mots trop longs du télégramme courant.

- *compte associé à chaque mot de la suite tel le nombre de mots comptables et le nombre de mots trop longs.

II- 2 - 4 - Définition du module *compte.

Il est nécessaire de connaître la longueur d'un mot. Cette constatation nous amène à introduire un autre intermédiaire : le caractère et à considérer un mot comme une suite de caractères.

On doit aussi se préoccuper de savoir si le mot courant est comptable ou non. Plutôt que de comparer chaque mot du télégramme aux mots "STOP" et "ZZZZ", nous n'effectuerons la comparaison que sur les mots de quatre lettres.

*compte
unmot , nbmot : si nbcar = 4 alors si mot = 'STOP' alors | unmot = . | nbmot = ω nbmot
si mot = 'ZZZZ' alors | unmot = <> | nbmot = ω nbmot
sinon | unmot = mott | nbmot = ω nbmot + 1
nblong, mott : si nbcar > 12 alors | mott = mot [1, 12] | nblong = ωnblong + 1
sinon | nblong = ω nblong | mott = mot
nbcar = der iter *ccar sur mot
prem nbmot = 0
prem nblong = 0

- unmot désigne le mot qui sera imprimé (il tient compte des modifications à effectuer sur le télégramme source).

- nbcar désigne le nombre de caractères contenu dans un mot

- mott désigne le mot tronqué à 12 caractères

- *ccar module qui définit le calcul du nombre de caractères.

Remarque : La valeur du prédécesseur du terme général x d'une suite est représentée par ωx. Puisque l'utilisation de cette notation correspond à une définition récurrente, la suite correspondante doit être initialisée à l'aide de l'opérateur pre (premier). Cette initialisation est placée dans le module itéré et non pas dans un module spécial comme dans la syntaxe présentée dans l'exemple du drapeau tricolore (cf. chapitre 1, $ II - 2 -2-).

II-2 - 5 - Définition de *ccar

*ccar
nbcar = ∂nbcar + 1
prem nbcar = 0

La définition du résultat et des résultats intermédiaires est terminée. On doit maintenant se préoccuper des suites intermédiaires.

II-3 - Définition des suites intermédiaires :

La définition globale du résultat nous a suggéré l'introduction de la suite des télégrammes. Une décomposition plus approfondie nous a amené à considérer chaque terme de la suite des télégrammes comme une suite de mots et chaque mot comme une suite de caractères (l'introduction de ces suites intermédiaires ne s'est pas effectuée globalement mais au fur et à mesure de la définition plus approfondie du résultat).

stels — ou suite des télégrammes

tel — ou suite de mots

mot — ou suite de caractères

car — un caractère

La suite donnée est une suite de caractères, scaractères.

II- 3 - 1 - Définissons stels

Elle est caractérisée par son dernier élément, le télégramme vide. Le nombre de télégrammes est à priori inconnu.

stels = jqa telvide iter *ctel avec scaractères

- telvide est une suite de booléens initialisée à faux. Elle prend la valeur vrai lorsque le télégramme vide est rencontré.

- *ctel module définissant tel un télégramme à l'aide de la suite scaractères. Il n'y a pas bijection entre les éléments de stels et les éléments de scaractères (il faut plusieurs caractères pour constituer un télégramme). La façon dont la suite scaractères est utilisée est définie dans *ctel. Ceci est indiqué syntaxiquement par le symbole avec.

Remarque : La lecture du texte par bloc n'apporte pas d'éléments supplémentaires à la résolution du problème.

II-3 - 2 - Le module *ctel

tel a été utilisée comme une suite de mots. On n'en connait pas le nombre, mais on est capable de caractériser le dernier élément de cette suite : le mot est égal au mot spécial "ZZZZ".

*ctel
tel, telvide = jqa fintel iter *cmot avec scaractères

- fintel est la suite de booléens initialisée à faux qui prend la valeur vrai lorsque la fin du télégramme traité est atteint, lorsque mot = "ZZZZ".

- *cmot définit le terme général d'un télégramme soit un mot.

II-3 - 3 - Le module *cmot

L'utilisation de la suite mot dans la définition du résultat nous a conduit à la considérer comme une suite de caractères. mot peut donc être défini à partir de la suite donnée scaractères. L'énoncé précise que deux mots sont séparés par des séquences de blancs : un mot peut être défini relativement à son premier et son dernier caractère.

*cmot
mot = scaractères de d à f exclu
d = prem car dans scaractères depuis f telque car # ' '
f = prem car dans scaractères depuis d telque car = ' '
telvide = fintel et ∂fintel
fintel = mot = 'ZZZZ'
scaractères = si finbloc alors donnée
prem fintel = faux
prem f = tête (scaractères)

- scaractères correspond au texte donné à traiter.

- d désigne le premier caractère du mot courant et f son dernier caractère.

- finbloc est un booléen représentant la contrainte imposée par la lecture des télégrammes sources.

II- 4 - Première version de l'alogorithme

Elle est obtenue en rassemblant les définitions des résultats et des suites intermédiaires dans une même table. La présentation est modulaire et se divise en deux parties (pour chaque module): la partie lexique contenant les définitions de types et des commentaires de tous les objets utilisés et la partie définitions algorithmiques.

*Les télégrammes

- lmessage suite de triplets - *cmessage associe à chaque tel de stels un triplet (untel, totmot, tlong) - stels suite de télégrammes - telvide suite de booléens - *ctel définit tel - scaractères suite de caractères donnée	résultat = écrire lmessage lmessage = iter *cmessage sur stels stels = jqa telvide iter *ctel avec scaractères
*cmessage	
- message triplet associé à un télégramme - totmot entier nombre total de mots comptables - tlong entier nombre total de mots trop longs - untel suite des mots du télégramme après les modifications nécessaires - nbmot suite des nombres de mots comptables - nblong suite des nombres de mots trop longs - *compte associe à un télégramme tel le nombre de mots et transforme le mot si nécessaire	message = untel, totmot, tlong totmot, tlong = der(nbmot, nblong) nbmot, nblong, untel = iter *compte sur tel
*compte	
- nbcar entier nombre de caractères du mot courant - mot suite de caractères, élément courant de tel - *ccar associe à un mot son nombre de caractères - unmot suite de caractères, élément courant de untel obtenu à partir de mot - mott suite de caractères, mot tronqué à 12 caractères	nbmot, unmot : si nbcar=4 alors si mot='STOP' alors | unmot=. | nbmot=∂nbmot si mot='ZZZZ' alors | unmot=( ) | nbmot=∂nbmot sinon | unmot=mott | nbmot=∂nbmot+1 nblong, mott : si nbcar 12 alors | mott = mot | nblong =∂nblong sinon | mott = mott [1, 12] | nblong =∂nblong+1 nbcar = der iter *ccar sur mot prem nbmot = 0 prem nblong = 0

..../....

..../....

*ccar	
	nbcar =∂nbcar + 1 prem nbcar = 0
*ctel	
- fintel suite de booléens - *cmot définit un mot à partir de scaractères	tel, telvide = jqa fintel iter *cmot avec scaractères
*cmot	
- car caractère - d caractère marquant le début du mot - f caractère marquant la fin du mot - tête fonction d'accès au premier caractère de la suite - finbloc booléen lié à la contrainte imposée par la lecture des télégrammes sources	mot = scaractères de d à f exclu d = prem car dans scaractères depuis ∂f telque car # ' ' f = prem car dans scaractères depuis d telque car = ' ' telvide = fintel et ∂fintel scaractères = si finbloc alors donnée prem fintel = faux prem f = tête (scaractères) fintel= mot = 'ZZZZ'

Les telegrammes: algorithme initial

II- 5 - Conclusion

Dans ce premier exemple, la première suite intermédiaire introduite ne s'exprime pas directement en fonction des données du problème. Au fur et à mesure de l'explicitation des résultats, cette suite se décompose en télégrammes, puis le télégramme en suite de mots et enfin les mots en suite de caractères (l'élément de base de cette suite est identique à l'élément de base de la suite donnée).

Cette décomposition s'écrit facilement par une succession d'itérations simples.

III - PRESENTATION DU LANGAGE UTILISE

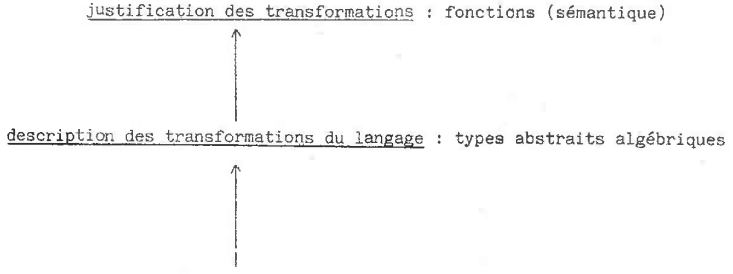
Dans ce paragraphe, nous définissons formellement le langage utilisé que nous appelons MEDEE.

Nous distinguerons trois niveaux de langages, de niveaux d'abstractions différents permettant de décrire, formaliser et justifier les manipulations proposées:

- Le premier niveau de langage est le langage MEDEE lui-même, langage de description d'algorithmes permettant de définir des objets.

- Au-dessus de ce langage, nous utiliserons un méta-langage permettant de décrire notre langage : les définitions et les objets de MEDEE seront définis en termes de types abstraits algébriques.

- Le niveau fonctionnel nous permettra de justifier les transformations proposées sur notre langage.



Notation utilisée : Les identificateurs

x, y, z désignent des noms de résultat
s, u, v, w des suites
d des données
exp des expressions
g, h des fonctions
f des fichiers

III- 1 - Le langage MEDEE

Nous distinguons les définitions et les objets du langage.

III - 1 - 1 - Les définitions

Nous disposons des trois sortes de définitions habituelles.

a) Les définitions simples :

de la forme x = exp

ou x = f (y, d)

avec f une fonction primitive du langage.

b) Les définitions conditionnelles :

x = si condition alors *cx1 sinon *cx2

où condition est de type booléen

*cx1 désigne le module (ou ensemble de définitions) définissant x lorsque condition est vrai

*cx2 module définissant x lorsque condition est faux.

c) Les définitions itératives :

Elles définissent des suites.

c1/ Définition itérative sur une suite l

u = iter *cu sur l

définit la suite u de terme général ui en bijection avec l'élément courant de la suite l.

*cu désigne le module qui associe à chaque eli de l le terme ui de la suite u.

l est une suite quelconque.

Remarque : Cette forme d'itération est d'un emploi très commode car on n'a pas à se soucier du parcours de la suite l en même temps que de la définition de u. Elle a été utilisée pour définir lmessage dans l'exemple des télégrammes:

lmessage = iter *ctel avec scaractères

c2/ Définition itérative sur une suite l avec condition d'arret

u = jqa arret iter *cu sur l

permet de définir des itérations dont le domaine de définition est déterminé par la suite l et dépend aussi d'une certaine condition d'arret (cette condition peut porter sur l'élément courant de la suite l ou sur l'élément courant de la suite u).

c3/ Définition itérative sans domaine de définition

u = jqa arret iter *cu avec l.

Contrairement à la forme précédente, au niveau de cette itération, on ne sait pas comment intervient la suite l dans la définition de la suite u : ui n'est pas défini en bijection avec eli, terme général de la suite l.

- avec l signifie que cette suite l sera utilisée pour définir ui. Le parcours de cette suite sera défini dans le module *cu.

Remarque : Cette forme d'itération correspond au "jusqu'à ..." classique dans lequel, pour des raisons méthodologiques, on met en évidence les suites dont elle dépend (on peut avoir avec l1, l2).

Reprenons l'exemple des télégrammes. Nous avons été amené à définir la suite des télégrammes notée stels par :

stels = jqa telvide iter *ctel avec scaractères

la définition du résultat de manière déductive nous a suggéré de définir stels comme la suite des télégrammes et se terminant par le télégramme vide. Pour définir cette suite, nous disposons d'une suite donnée scaractères qui est la suite des caractères du texte source. stels est définie à partir de cette suite mais pas par une itération sur cette suite: cela reviendrait à définir un télégramme en bijection avec un caractère.

c4/ Sous-suite

u = l de début (el) à fin (el)

permet de définir une sous-suite ou "tranche" au sens algol 68 du terme.

III -1 - 2 - Les objets du langage MEDEE

Nous serons amené à distinguer le type suite et le type fichier.

a) Les suites

L'objet suite est un objet fondamental en informatique qui contient la notion de fonction. On lui associe deux modes de représentations :

- l'une spatiale (notion de tableaux, de fichiers séquentiels ...)
- l'autre temporelle (notion de variable), lié à un calcul.

b) Les fichiers

Le type suite introduit ici est utilisé dans le sens d'objet logique : on ne lui associe pas de fonction de représentation. C'est un type abstrait, mathématique.

Dans la plupart des problèmes traités, un certain nombre de contraintes physiques doivent être prises en compte dès le début de l'analyse (exemple : présentation des états de sortie, utilisation du fichier des livraisons en attente stocké sur bande magnétique ...). Afin de prendre en compte ces contraintes dans l'algorithme sans pour autant se préoccuper de la machine, nous utiliserons le type fichier.

III - 2 - Description algébrique du langage

Remarque : Tous les concepts utilisés dans ce paragraphe ne seront pas toujours définis complétement de manière formelle. Nous avons jugé qu'ils n'apporteraient rien de plus aux démonstrations si ce n'est une certaine lourdeur.

Ce niveau, plus abstrait que le langage MEDEE décrit au paragraphe III - 1 nous permet de définir formellement les manipulations effectuées sur le langage.

Nous expliciterons les types DEFINITION, SUITE et MODULE.

1 - Les définitions

Le type DEFINITION noté DEF dépend des types de base SIDENT, DEFORM (partie droite d'une définition), EXP (expression), MOD (module défini au paragraphe suivant) et TYPEDEF où :

SIDENT désigne le type des suites finies d'identificateurs
TYPEDEF = (defsimple, defcond, defiter)
désigne le type de la définition.

1 - 1 - Opérations sur le type définition :

defvide :	——>	DEF	
partie gauche : DEF	——>	SIDENT	% nom des résultats de la définition %
partie droite : DEF	——>	DEFORM	
iddroite : MOD X DEF	——>	SIDENT	% liste des identificateurs utilisés en partie droite d'un module %
fusdef : DEF X DEF	——>	DEF	% fusion de deux définitions %
subs : DEF X SIDENT X EXP	——>	DEF	% substitution d'un identificateur par une expression %

Remarque : Les identificateurs précédés de ∂ (ancienne valeur de) sont considérés comme des identificateurs.

1 - 2 - Préconditions

pré fusdef (d1, d2) = partiegauche (d1) ∩ iddroite (d2) = vide
et partiegauche (d2) ∩ iddroite (d1) = vide
et iddroite (d1, m) ∩ iddroite (d2, m) = vide
et typedef (d1) = typedef (d2)

pré subs (d1, x, exp) = x ∈ iddroite (d1, m)

Notation : pré subs (d1, x, exp) = x ∈ iddroite (d1, m) signifie :
subs (d1, x, exp) est défini si la condition x ∈ iddroite (d1, m) est vérifiée.

1 - 3 - Axiomes

⊥ note l'élément indéfini de SIDENT.

∀ m ∈ MOD

| partiegauche (defvide) = ⊥
| partiedroite (defvide) = ⊥
| iddroite (m, defvide) = ⊥

| partiegauche (x = exp) = { x }
| partiedroite (x = exp) = exp

Ces deux définitions sont valables quel que soit exp .

Pour définir iddroite, il est nécessaire d'envisager tous les cas possibles de définitions.

a) définitions simples : x = exp

iddroite (m, x = exp) = { id ∈ exp }

b) définitions conditionnelles : x = si c alors *cx1 sinon *cx2

iddroite (m, x = si c alors *cx2 sinon *cx2) = idroitem (*cx1) U idroitem (*cx2)

Remarque : idroitem sera défini dans le type Module. Il désigne l'ensemble des identificateurs utilisés dans le module.

c) définitions itératives

c1 - iddroite (m, u = iter *cu sur 1) = idroitem (*cu) U { 1 }

c2 - iddroite (m, u = jqa arret iter *cu sur 1) = idroitem (*cu) U { 1 }

c3 - iddroite (m, u = jqa arret iter *cu avec 1) = idroitem (*cu)

ici 1 est défini et utilisé dans *cu.

d) fusion de deux définitions : Pour tout d1, d2 ∈ DEF

partiegauche (fusdef (d1, d2)) = partie gauche (d1) U partiegauche (d2)

Remarque : Pour définir partiedroite (fusdef (d1, d2)), nous sommes amenés à considérer le type des définitions d1 et d2

d1/ Conditionnelles

partiedroite (fusdef (d1, d2)) = si c1 = c2 alors
partiedroite (x, y = si c1 alors fusm(*cx1, *cy1) sinon fusm(*cx2, *cy2))

d2/ Itératives

partiedroite (fusdef (u = iter *cu sur 1, v = iter *cv sur 1))
= partiedroite (u, v = iter fusm (*cu, *cv) sur 1))

d3/ u = jqa arret iter *cu sur 1

partiedroite (fusdef (u = jqa arret iter *cu sur 1, v = jqa arret iter *cv sur 1)
= partiedroite (u, v = jqa arret iter fusm (*cu, *cv) sur 1)

2 - Les suites

Le type suite noté SUITE dépend des types de base VAL, ENTIER.

Remarque : Dans le type VAL, ensemble des valeurs, nous ne ferons pas de distinctions entre les valeurs réelles, booléennes ....

2 - 1 - Opérations sur le type suite

svide : ⟶ SUITE
cons : SUITE X VAL ⟶ SUITE
prem : SUITE ⟶ VAL % accès au premier élément %
der : SUITE ⟶ VAL % accès au dernier élément %
début : SUITE ⟶ SUITE % suite privée de son dernier élément %
queue : SUITE ⟶ SUITE % suite privée de son premier élément %
jqa : SUITE X VAL ⟶ SUITE % suite avec condition d'arrêt %
lg : SUITE ⟶ ENTIER % longueur de la suite %
aplat : SUITE [SUITE [VAL]] ⟶ SUITE [VAL]
concat : SUITE X SUITE ⟶ SUITE % concaténation de deux suites %
estsousuite : SUITE X SUITE ⟶ BOOLEEN % prédicat sous-suite de %
dec : SUITE [VAL, VAL] ⟶ SUITE [VAL] X SUITE [VAL]
% associe à une suite de couples un couple de suites %

cons et svide sont les deux constructeurs du type SUITE. La règle de récurrence associée peut s'écrire :

| pour toute propriété $\mathcal{P}$ sur SUITE telle que
| $\mathcal{P}$ (svide) vrai
| et ∀ v ∈ VAL, ∀ s ∈ SUITE $\mathcal{P}(s) \Rightarrow \mathcal{P}$ (cons(s, e))
| alors $\mathcal{P}(s)$ est vrai pour tout s ∈ SUITE.

Remarque : la concaténation de deux suites sera notée concat ou, plus souvent, *.

2 - 2 - Axiomes

pour tout v1, v2 $\in$ VAL
s1, s2 $\in$ SUITE

Ω note l'élément indéfini associé au type VAL. Dans certains cas, il sera commode que Ω représente le plus grand élément possible (noté ∞).

a/ prem (svide) = Ω
prem (cons(s, v)) = prem (s)

b/ der (svide) = Ω
der (cons (s, v)) = v

c/ début (svide) = svide
début (cons (s, v)) = s

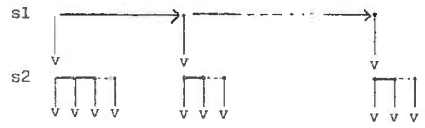
d/ queue (svide) = svide
queue (cons (s, v)) = cons (queue (s), v)

e/ jqa (svide, vrai) = svide
jqa (svide, faux) = svide
jqa (cons (s, v), vrai) = s
jqa (cons (s, v), faux) = cons (s, v)

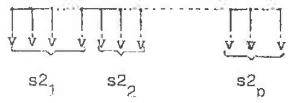
f/ lg (svide) = 0
lg (cons (s, v)) = lg (s) + 1

g/ aplat (svide) = svide

schématiquement, s1 (s2(v)) se représente par :



aplat (s1 (s2(v))



Remarque : Si s est une suite linéaire simple, l'opération aplat appliquée à s représente la fonction identité.

Définissons cet opérateur à l'aide de prem et queue :

prem (aplat (s1)) = prem (prem (s1))

queue (aplat (s1)) = concat (queue (prem (s1)), aplat (queue (s1)))

j/ dec (svide (s1, s2)) = svide, svide
dec (cons ((s1, s2), (u1, u2)) = cons (s1, u1), cons (s2, u2)

h/ prem (concat(s1, s2)) = prem (s1)
queue (concat (s1, s2)) = concat (queue (s1), s2)

i/ estsousuite (svide, s1) = faux
estsousuite (cons (s1, v), s2) =
si prem (s1) = prem (s2) alors estsousuite (cons (queue (s1), v), queue(s2
sinon estsousuite (cons (queue (s1), v), s2)

2 - 3 - Opérations composées sur les suites

Ces opérations s'appliquent dans le cas où le type VAL représente un ensemble totalement ordonné.

a - Suite définie par interclassement de deux suites

intercl : SUITE X SUITE ——————> SUITE

prem (intercl (s1, s2)) = si prem (s1) $\leq$ prem (s2) alors prem (s1)
sinon prem (s2)

queue (intercl (s1, s2)) = si prem (s1)$\leq$ prem (s2) alors intercl (queue (s1), s2)
sinon intercl (s1, queue (s2))

Remarque : Ces opérations sont valables quelles que soient les suites s1 et s2, en particulier lorsque l'une des suites est vide grâce à l'axiome

prem (svide) = Ω = + ∞ (le plus grand élément)

b - Insertion d'une sous-suite connexe

insersuite : SUITE X SUITE ——————> SUITE

prem (insersuite (s, s1)) = si prem (s) < prem (s1)
alors prem (s)
sinon prem (s1)

queue (insersuite (s, s1)) = si prem (s) < prem (s1)
alors insersuite (queue (s), s1)
sinon insersuite (s, queue (s1))

c - Suppression d'une sous-suite

supsuite : SUITE X SUITE ——————> SUITE

précondition :

pré supsuite (s, s1) = estsousuite (s, s1)

prem (supsuite (s, s1) = si prem (s) = prem (s1)
alors prem (supsuite (queue (s), queue(s1)))
sinon prem (s)

queue (supsuite (s, s1)) = si prem (s) = prem (s1)
alors queue (supsuite (queue (s), queue (s1)))
sinon supsuite (queue (s), s1)

d - Mise à jour d'une suite par une autre noté ⊠

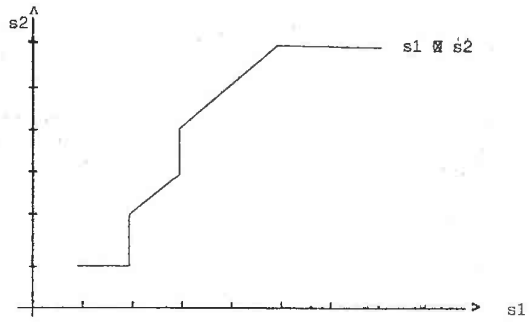
Notre but est d'aboutir à une méthode systématique pour aborder ce type de problèmes. Pour cela, nous introduisons une nouvelle suite dont chaque terme est un couple constitué d'éléments des deux suites :

⊠ : SUITE X SUITE ⟶ SUITE

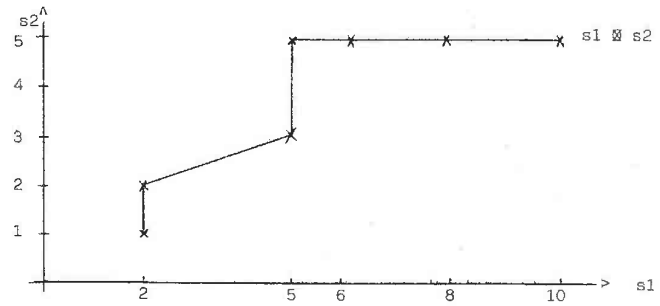
prem (s1 ⊠ s2) = (prem (s1), prem (s2))
(couple d'éléments)

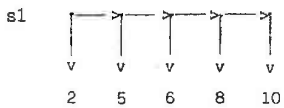
queue (s1 ⊠ s2) = si prem (s1) < prem (s2) alors queue (s1) ⊠ s2
si prem (s1) = prem (s2) alors queue (s1) ⊠ queue (s2)
si prem (s1) > prem (s2) alors s1 ⊠ queue (s2)

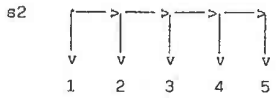
Ce qui se traduit graphiquement par :

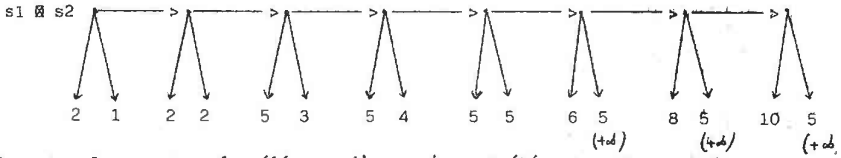


Prenons un exemple : s1 et s2 sont deux suites d'entiers :









Remarque: Lorsque tous les éléments d'une suite ont été parcourus, on note son élément courant (qui n'a pas d'existence) +∞: cela évite de tester la fin de la suite à partir du moment où elle a été rencontrée.

Remarque : Cette opération peut être généralisée au cas de plus de deux suites

s1 ⊠ s2 ⊠ ... ⊠ sk est définie par :

prem (s1 ⊠ s2 ⊠... ⊠ sk) = (prem (s1), prem (s2), ..., prem (sk))
queue (s1 ⊠ s2 ⊠ ... ⊠ sk) = (l1, l2, ..., lk)

li est défini par :

$\forall\, 1 \leq i \leq k$, li = si m = prem (si) alors queue (si) sinon ∂li
$m = \min_{1 \leq i \leq k} (\text{prem}(si))$

Cas particulier de suite :

L'intervalle p ⟶ q peut être considéré comme un cas particulier de suite défini par :

prem [p ...q] = p
queue [p ...q] = si p< q alors [p+1 ...q]
sinon svide

3 - Le type fichier

Le type fichier noté FICHIER est défini à partir des types SUP (pour support physique), SUITE et VAL.

3 - 1 - Opérations sur le type FICHIER

fvide : ⟶ FICHIER

écrire : FICHIER X SUP X SUITE ⟶ FICHIER % adjonction %

lire : FICHIER X SUP ⟶ VAL % accès à un élément %

tête : FICHIER X SUP ⟶ VAL % accès au premier élément %

concatf : FICHIER X FICHIER ⟶ FICHIER % concaténation %

iterf : SUITE X SUP ⟶ FICHIER % adjonction par itération %

3 - 2 - Préconditions (sup1 et sup2 désignent les supports de fichiers)

pré concatf (f1, f2) = sup1 (f1) = sup2 (f2)
(les deux fichiers ont même support)

3 - 3 - Axiomes

fvide et écrire sont deux constructeurs du type FICHIER.

lire (fvide) = Ω

lire (écrire (f, sup1, v), sup2) = si sup1 = sup2 alors v
sinon lire (f, sup2)

tête (fvide) = Ω

tête (écrire (f, sup1, u), sup2) = si sup1 = sup2 alors tête (f, sup1)
sinon tête (f, sup2)

concatf (fvide, fvide) = fvide

concatf (écrire (f1, sup, u1), écrire (f2, sup, u2))=écrire (f1, sup, u1 * u2)

iterf (svide, sup) = fvide

iterf (cons (s, v), sup) = écrire (iterf (s, sup), sup, v)

3 - 4 - Propriétés de l'opérateur écrire

3 - 4 -1- Par rapport à la concaténation de deux suites

écrire (f, sup, u1 * u2) = concatf (écrire (f, sup, u1), écrire (f, sup, u2))

(il s'agit de l'axiome sur l'opérateur concatf lorsque les deux fichiers sont les mêmes).

3 - 4 -2- Par rapport à l'itération

écrire (f, sup, iter *cu sur 1) = (iterf *cu'sur 1, sup)

où *cu est défini par *cu [u' = écrire (f, sup, u)]

4 - Le type MODULE

Un module est un ensemble de définitions auquel est associé un ou plusieurs résultats. Il existe deux types de modules :

a) Les modules itérés: Ils peuvent être considérés comme la réunion de deux ensembles de définitions :

- l'un définissant l'initialisation des objets récurrents (cet ensemble peut, éventuellement, être vide).
- l'autre définissant l'élément courant.

b) Les modules introduits par les définitions conditionnelles dans lesquels la partie initialisation n'existe pas.

Le type module noté MOD, dépend des types DEF, SIDENT et BOOLEEN.

4 - 1 - Opérations sur les modules

modvide : ⟶ MOD

ajoutinit : MOD X DEF ⟶ MOD % ajouter une définition d'initialisation

ajoutres : MOD X DEF ⟶ MOD % ajouter une définition résultat %

ajoutrec : MOD X DEF ⟶ MOD % ajouter une définition quelconque %

résultat : MOD ⟶ SIDENT % ensemble des résultats %

idroitm : MOD ⟶ SIDENT % ensemble des identificateurs utilisés en partie droite des définitions du module %

init : MOD ⟶ MOD % module d'initialisation %

rec : MOD ⟶ MOD % module définissant le terme général de la suite résultat %

retirem : MOD X DEF ⟶ MOD % enlever une définition %

fusm : MOD X MOD ⟶ MOD % fusion de deux modules notée ∧ %

caplat : MOD ⟶ MOD % introduction d'un nouveau module %

estdéfini : MOD X SIDENT ⟶ BOOLEEN % est-ce que l'identificateur x est défini dans ce module?%

appartient : MOD X DEF ⟶ BOOLEEN % est-ce que la définition d appartient à ce module?%

estutilisé : MOD X SIDENT ⟶ BOOLEEN % est-ce que l'identificateur x est utilisé dans ce module?%

dépend : MOD X SIDENT X SIDENT ⟶ BOOLEEN % exprime la non circularité %

Remarque : Nous avons introduit deux opérateurs ajoutres (ajouter une définition résultat) et ajoutrec (ajouter toute autre définition) car savoir si une définition est un résultat est un problème purement syntaxique.

modvide, ajoutinit, ajoutres et ajoutrec sont les quatre constructeurs du type MOD.
La règle de récurrence associée peut s'écrire :

> Pour toute propriété $\emptyset$ sur MOD,
> si $\emptyset$ (modvide) vrai, et si pour tout m $\in$ MOD, pour tout d $\in$ DEF
> $\emptyset$ (m) vrai et ajoutrec (m, d)
> alors $\emptyset$ (ajoutrec(m, d)) vrai

4 - 2 - Préconditions :

pré ajoutinit (m, d) = appartient (m, d) = faux
pré ajoutres (m, d) = appartient (m, d) = faux
pré ajoutrec (m, d) = appartient (m, d) = faux
pré retirem (m, d) = appartient (m, d) = vrai
pré fusm (m1, m2) = estdéfini (m1, x) et ⌝ estdéfini (m2, x)
et estdéfini (m2, y) et ⌝ estdéfini (m1, y)
pré caplat (d) = typedef (d) = defiter % d est une définition itérativ
pré dépend (m, x, y) = ⌝ dépend (m, y, x) et estdéfini (m, x) et
y $\in$ iddroite (m, x) et estdéfini (m, y).

4 - 3 - Axiomes

Pour tout d, d1, d2 $\in$ DEF,
m, m1, m2 $\in$ MOD,

ω désigne le module non défini ($\omega \in$ MOD) et
$\perp$ l'élément non défini de SIDENT.

a) résultat (modvide) = ω
résultat (ajoutinit (m, d)) = résultat (m)
résultat (ajoutres (m, d)) = résultat (m) U partiegauche (d)
résultat (ajoutrec (m, d)) = résultat (m)

b) idroitm (modvide) = $\perp$
idroitm (ajoutinit (m, d)) = idroitm (m) U iddroite (m, d)
idroitm (ajoutres (m, d)) = idroitm (m) U iddroite (m, d)
idroitm (ajoutrec (m, d)) = idroitm (m) U iddroite (m, d)

c) init (modvide) = ω
init (ajoutinit (m, d)) = ajoutinit (init (m), d)
init (ajoutrec (m, d)) = init (m)
init (ajoutres (m, d)) = init (m)

d) rec (modvide) = ω
rec (ajoutinit (m, d)) = rec (m)
rec (ajoutrec (m, d)) = ajoutrec (rec (m), d)
rec (ajoutres (m, d)) = ajoutres (rec (m), d)

e) retirem (modvide, d) = ω
retirem (ajoutinit (m, d1), d2) = si d1=d2 alors m
sinon ajoutinit (retirem (m, d1), d2)
retirem (ajoutrec (m, d1), d2) = si d1=d2 alors m
sinon ajoutrec (retirem (m, d1), d2)
retirem (ajoutres (m, d1), d2) = si d1=d2 alors m
sinon ajoutres (retirem (m, d1), d2)

f) fusm (modvide, m) = m
fusm (m1, ajoutinit (m2, d)) = ajoutinit (fusm (m1, m2), d)
fusm (m1, ajoutrec (m2, d)) = ajoutrec (fusm (m1, m2), d)
fusm (m1, ajoutres (m2, d)) = ajoutres (fusm (m1, m2), d)

g) estdéfini (modvide, x) = faux
estdéfini (ajoutinit (m, d), x) = x $\in$ partiegauche (d) ou
estdéfini (m, x)
estdéfini (ajoutrec (m, d), x) = x $\in$ partiegauche (d) ou
estdéfini (m, x)
estdéfini (ajoutres (m, d), x) = x $\in$ partiegauche (d) ou
estdéfini (m, x)

h) appartient (modvide, d) = faux
appartient (ajoutinit (m, d1), d2) = (d1=d2) ou appartient (m, d2)
appartient (ajoutrec (m, d1), d2) = (d1=d2) ou appartient (m, d2)
appartient (ajoutres (m, d1), d2) = (d1=d2) ou appartient (m, d2)

i) estutilisé (modvide, x) = faux
estutilisé (ajoutinit (m, d), x) = x $\in$ partiedroite (d) ou
estutilisé (m, x)
estutilisé (ajoutrec (m, d), x) = x $\in$ partiedroite (d) ou
estutilisé (m, x)
estutilisé (ajoutres (m, d), x) = x $\in$ partiedroite (d) ou
estutilisé (m, x)

j) dépend (modvide, x, y) = faux
dépend (ajoutinit (m, d), x, y) = x $\in$ partiegauche (d) et
y $\in$ iddroite (m, d) ou dépend (m, x, y)

dépend (ajoutrec (m, d), x, y) = x ∈ partiegauche (d) et
y ∈ iddroite (m, d) ou dépend (m, x, y)

dépend (ajoutres (m, d), x, y) = x ∈ partiegauche (d) et
y ∈ iddroite (m, d) ou dépend (m, x, y)

k) caplat (modvide) = modvide

caplat (ajoutinit (m, d)) = ajoutinit (caplat (m), d)

caplat (ajoutrec (m, d)) = ajoutrec (caplat (m), d)

caplat (ajoutres (m, d)) = ajoutres (caplat (m), d).

III - 3 - La sémantique

III - 3 -1- Introduction

Afin de justifier les règles de transformations introduites, associons à tout programme écrit en MEDEE une fonction qui est sa valeur sémantique. A tout programme P, on associera une fonction définie dans l'ensemble des données à valeur dans l'ensemble des résultats.

Pour définir la sémantique, nous avons besoin du type fonction noté FONC :

FONC : DEP^n ——— ARR^p

Ce type admettre comme paramètres effectifs des objets de type :

IDENT

SIDENT : IDENT X ... X IDENT

BOOLEEN

EXP (expression)

VAL (ensemble des valeurs).

1 - Les opérations

λ. : SIDENT X EXP ——> FONC

cond : BOOLEEN X FONC X FONC ——> FONC % opérateur conditionnel habituel %

comp : IDENT X FONC X FONC ——> FONC % composition de deux fonctions : remplacement du premier paramètre par sa définition dans le deuxième %

par : FONC X FONC ——> FONC % définition de plusieurs résultats %

decf : FONC ——> FONC X FONC % opérateur de découplage : restitue un couple de résultats %

eval : EXP ——> VAL % évaluation d'une expression %

arité : FONC ——> ENTIER % associe à une fonction le nombre de paramètres résultats %

Remarque : Si on voulait être plus précis, il faudrait donner dans les profils des opérateurs, les paramètres de FONC.

2 - Préconditions :

pour tout f, g ∈ FONC,
x, y ∈ IDENT,
exp ∈ EXP

Remarque : arité est un opérateur qui concerne le domaine d'arrivée de la fonction.

Par définition arité (EXP) = 1

pré cond (c, f, g) = arité (f) = arité (g)

pré comp (xi, f, g) = arité (g) = 1

pré par (f, g) = arité (par (f, g)) ≠ 1

pré decf (f) = arité (f) ≠ 1

3 - Définitions

a) λ. x1 ... xn . exp = f

f (x1 ... xn) = eval (exp (x1/u1, x2/u2, ... xn/un))

b) cond (vrai, f, g) = f

cond (faux, f, g) = g

c) comp (xi, f,g) =

λ x1. x2...xi-1 y1...yp. xi+1...xn . f(x1,...xi-1, g(y1,...yp), xi+1...xn)

Remarque : comp (x, f, g) permet de remplacer la variable x définie par l'intermédiaire de la fonction g par sa définition dans f.

d) par (f, g) = λ.x1 x2 ..xn.y1 .. yp . (f(x1..xn), g(y1..yp))

e) Propriété de decf par rapport à l'opérateur cond :

decf (cond(c, f, g)) = cond (c, decf(f), decf(g))

f) Propriété de decf par rapport à l'opérateur par :

decf (g(par(f1, f2)) = par (g(f1), g(f2))

4 - Propriétés

4 - 1 - Distributivité de l'opérateur par par rapport à l'opérateur cond

PROPRIETE :

par (cond(c, f1, g1), cond(c, f2, g2)) = cond (c, par(f1, f2), par(g1, g2))

PRECONDITIONS :

arité (f1) = arité (g1)
arité (f2) = arité (g2)

DEMONSTRATION

premier cas : c = vrai

utilisons la définition de l'opérateur cond :
par (cond(vrai, f1, g1), cond(vrai, f2, g2)) = par (f1, f2)

deuxième cas : c = faux

par (cond(faux, f1, g1), cond(faux, f2, g2)) = par (g1, g2)

conclusion :

par (cond(c, f1, g1), cond(c, f2, g2)) = cond(c, par(f1, f2), par(g1, g2))

4 - 2 - Distributivité de l'opérateur par par rapport à l'opérateur comp

PROPRIETE :

par (comp(x,f1, g), comp(x, f2, g)) = comp (x, par(f1, f2), g)

PRECONDITION :

arite (g) = 1

DEMONSTRATION :

Pour faciliter la lisibilité, prenons le cas particulier où les fonctions f et g ont une seule variable :

P1 = par (comp(x, f1, g), comp(x, f2, g))
= par (comp(x, f1(x), g(y)), comp(x, f2(x), g(y)))
= par (λ y. f1(g(y)), λ y. f2(g(y)))
= λ y. (f1(g(y)), f2(g(y)))

P2 = comp (x, par(f1, f2), g)
= comp (x, par(f1(x), f2(x)), g(y))
= λ y. par(f1(g(y)), f2(g(y)))
= λ y. (f1(g(y)), f2(g(y)))

4 - 3 - Distributivité de l'opérateur comp par rapport à l'opérateur co

PROPRIETE :

comp(x, cond(c, f1, g1), cond(c,f2, g2)) = cond(c, comp(x, f1, f2), comp(x, g1, g2))

PRECONDITIONS :

arité (f1) = arité (g1)
arité (f2) = arité (g2) = 1

DEMONSTRATION :

premier cas : c = vrai

comp(x, cond(vrai, f1, g1), cond(vrai, f2, g2)) = comp(x, f1, f2)

deuxième cas : c = faux

comp(x, cond(faux, f1, g1), cond(faux, f2, g2)) = comp(x, g1, g2)

conclusion :

comp(x, cond(c, f1, g1), cond(c, f2, g2)) = cond(c, comp(x, f1, f2), comp(x, g1, g2))

III - 3 -2- Sémantique d'une définition

D'une manière générale, la sémantique d'une définition d s'exprime par :

Sem [d] = λ iddroite (m, d) . partiedroite (d)

1 - Définition simple

Sem [x = exp] = λ iddroite (m, x = exp) . exp

2 - Définition conditionnelle

Sem [x = si c alors *cx1 sinon *cx2] = cond (c, Sem [*cx1], Sem [*cx2])

Remarque : La sémantique d'un module, tel que *cx1, notée Sem [*cx1] sera définie au paragraphe III - 3, p. 69.

3 - Définitions itératives :

3 - 1 - De types u = iter *cu sur l

Sem [u = iter *cu sur l] = it (*cu) (uo, l)

Remarque : it définit la fonction sémantique de l'itération. C'est une fonction récursive dépendant du module *cu, dont le premier appel fait intervenir l'initialisation de la suite u et la suite l.

Définitions :

uo = Sem [init (*cu)]
it (*cu) (u, ll) = cond (ll = svide, u , concat(u , it(*cu)(u ',queue(ll)))
u' = Sem [rec (*cu)] (u, prem(ll))

Remarque : Nous avons fait une utilisation abusive de l'opérateur de concaténation sur les suites :

concat : SUITE X SUITE ——→ SUITE

Ici, les deux paramètres de l'opérateur concat ne sont pas des suites, mais des fonctions à valeur dans les suites.

Pour être complétement rigoureux, nous devrions introduire un nouvel opérateur de concaténation sur les fonctions.

3 - 2 - De type u = jqa arret iter *cu sur l :

Sem [u = jqa arret iter *cu sur l] = it (*cu) (uo, l)

uo, arreto = Sem [init (*cu)]
it(*cu)(u,arret,ll) = cond(arret ou ll=svide,u,concat(u,it(*cu)(u',queue(ll))))
u', arret = Sem [rec (*cu)] (u, prem (ll))

3 - 3 - De type u = jqa arret iter *cu avec l

Sem [u = jqa arret iter *cu avec l] = it (*cu) (uo,l)

uo, arreto, lo = Sem [init (*cu)]
it (*cu) (u, ll) = cond (arret, u , concat (u, it (*cu) (u',ll")))
u', arret , ll" = Sem [rec (*cu)] (u, ll')
ll = concat (l', ll")
estsousuite (l, ll') et estsousuite (l, ll")

La différence avec les deux autres types d'itérations est dûe au fait qu'ici la suite l fait partie des objets définis dans *cu (c'est un intermédiaire défini dans *cu : l appartient à idroitem (*cu)).

III - 3 -3- Sémantique d'un module

L'élément indéfini du type module sera noté ω.
Définissons la sémantique des constructeurs du type module.

Remarque : Nous considérons les modules comme des ensembles ordonnés de définitions (ordre total). Cet ordre correspond à l'ordre inverse d'exécution: c'est l'ordre déductif (algorithme construit en partant du résultat).

DEFINITIONS :

1 - Sem (modvide) = ω

2 - Sem [ajoutinit(m, d)] = si estutilisé (m, partiegauche(d))
alors par(comp(partiegauche(d), Sem [m], Sem [d]), Sem[d])
sinon par(Sem [m], Sem [d])

3 - Sem [ajoutres(m, d)] = par(Sem [m], Sem [d])

4 - Sem [ajoutrec(m, d)] = comp(partiegauche(d), Sem [m], Sem [d])

5 - Sem [fusm(m1, m2)] = par(Sem [m1], Sem [m2])

6 - Propriété de l'opérateur comp par rapport à la sémantique d'un module

comp(x, Sem [m1], Sem [m2]) = Sem [fusm(m1, m2)]

III - 3 -4- Sémantique d'un algorithme

Un algorithme A est constitué d'un module principal MP utilisant des modules secondaires qui à leur tour utilisent des sous-modules suivant une décomposition hiérarchique.
Nous définissons la sémantique d'un algorithme par :

Sem [A] = Sem [MP]

MP désigne le module principal
A désigne l'algorithme.

IV - LES TRANSFORMATIONS

IV - 1 - Introduction

Lors de la construction du premier algorithme (cf. exemple des télégrammes), aucune hypothèse n'a été faite sur l'existence physique, réelle des objets lors de l'exécution, ni sur la forme de leur représentation. C'est en ce sens que l'on peut parler d'objets logiques.

Quelle est la signification de l'introduction de la suite intermédiaire stels dans l'exemple des télégrammes ? En fait, nous avons décomposé le problème à résoudre en deux sous-problèmes indépendants. L'implantation immédiate de l'algorithme obtenu entraîne la mise en oeuvre de deux processus indépendants au niveau fonctionnement. L'un des processus est chargé de construire la suite stels (suite des télégrammes) à partir des données du problème (la suite scaractères) : cette suite a donc une représentation physique en mémoire. Le deuxième processus construit les résultats à partir de cette suite stels.

Dans la plupart des cas, il est tout à fait inutile de conserver les suites intermédiaires en mémoire : dans l'exemple précédent, la suite stels n'a pas besoin d'être connue entièrement pour définir un message. A partir du moment où un télégramme est fourni, on peut constituer un message.

Un premier algorithme étant construit, notre but est maintenant d'obtenir un algorithme plus efficace, au sens d'une réduction de l'occupation mémoire et du temps de calcul.

Une première technique consisterait à utiliser des mécanismes de genre coroutines [CON , 63] , [CLI , 73] . Cette technique met en oeuvre deux processus : l'un appelé le producteur (ici le producteur de la suite intermédiaire) et l'autre le consommateur (ou l'utilisateur de cette suite intermédiaire fournissant le résultat),ces deux processus n'opérant pas simultanément.

La technique que nous proposons ici consiste à supprimer le maximum de suites intermédiaires introduites : ce choix est justifié par la taille des fichiers utilisés en analyse de gestion. Nous disposons d'un nombre restreint de règles de transformation portant sur les définitions du langage. Ces transformations sont sémantiques : elles altèrent l'histoire des calculs de l'alogorithme. Une stratégie d'utilisation de ces règles a été mise au point permettant de minimiser l'algorithme initial dans un nombre minimum d'étapes.

IV - 2 - Régles de transformations

Chaque règle sera décrite intuitivement, on en donnera ensuite une définition formelle, les conditions d'utilisation et une justification sémantique.

Remarque : Les notations utilisées sont celles introduites au début du paragraphe II de ce chapitre.

IV - 2 -1- Règle du pliage et dépliage

La règle du dépliage consiste à remplacer dans une expression un intermédiaire par sa définition. La règle du pliage est la règle inverse.

a) Premier cas :

1 - Définition :

```
res = x, y                 ↑ pliage de y = exp2 dans l'expression
  x = exp1 (y)               définissant x
  y = exp2
  ↕
res = x, y                 ↓ dépliage dans exp1(y)
  x = exp1 (exp2)            de y = exp2
  y = exp2
```


Commentaires sur l'écriture des règles en général :

Lorsque x est défini par une expression en fonction de y (ce qui est noté x = exp1 (y)) et y est lui-même défini par une expression (y = exp2), on peut remplacer ces deux définitions par deux autres définitions :

- l'une définissant x dans laquelle y a été remplacé par sa définition (x = exp1 (exp2) qui est un abrégé de Subs (x = exp1, y, exp2) où on fait apparaître l'élément qui a été remplacé).

- l'autre correspondant à la définition de y qui est inchangée par rapport à l'algorithme initial.

La flèche ↕ signifie que cette transformation peut s'utiliser dans les deux sens.

2 - Conditions d'utilisation :

- règle du dépliage (↓) : il n'y a pas de condition d'utilisation dans ce cas de figure.

- règle du dépliage (↑)

Le nouvel intermédiaire introduit (y) n'appartient pas au lexique existant.

$$id \in SIDENT$$

$$\| \forall\ id \text{ défini dans } A \Rightarrow y \neq id$$

A désigne l'algorithme.

3 - Justification sémantique :

Remarque : Nous noterons Sem1 la sémantique de l'algorithme avant transformation et Sem2 la sémantique de l'algorithme transformé à l'aide de la règle que l'on cherche à justifier. Notre but est de prouver Sem1 = Sem2.

Sem1 = Sem [x, y]

= par (Sem [x = exp1(y)], Sem [y = exp2])

= par (comp(y, Sem [x = exp1(y)], Sem [y = exp2]), Sem [y = exp2])
par définition

= par (λ y. exp1(exp2), exp2)

= par (Sem [x = exp1(exp2)], Sem [y = exp2])

= Sem2

b) Deuxième cas :

1 - Définition :

```
res = x
  x = exp1 (y)
  y = exp2
  ↓
res = x
  x = exp1 (exp2)
```

Seul x définit un résultat : y est un identificateur intermédiaire.

2 - Condition d'utilisation :

L'utilisation de cette règle entraîne la suppression de la définition de y seulement si y n'apparaît pas en partie droite d'une définition autre que x dans le module où y est défini ainsi que dans ses modules fils.

$d \in DEF$

$m \in MODULE$

$\| \forall\, d \in m,\ d \neq (x = expl(y)) \implies$ partiegauche $(y = exp2) \notin$ partiedroite (d)

m désigne le module dans lequel y est défini, ainsi que tous ses modules fils.

IV - 2 -2- Fusion de deux définitions

IV - 2 -2-1- Définitions conditionnelles

1 - Définition :

```
res = x, y
  x = si c alors *cx1  sinon *cx2
  y = si c alors *cy1  sinon *cy2
-------------------------------------------------
x, y = si c alors fusm (*cx1, cy1) sinon fusm (*cx2, cy2)
```

Cette règle évite de répéter deux fois le même test.

2 - Conditions d'utilisation :

L'application de cette règle entraîne la fusion de modules. Elle ne pourra s'effectuer que si la précondition sur la fusion des modules est vérifiée :

pré fusm (*cx1,*cy1) = estdéfini (*cx1, x) et ¬ estdéfini (*cy1, x)
et estdéfini (*cx1, y) et ¬ estdéfini (*cx1, y)

3 - Justification sémantique :

Sem1 = par (Sem [x = si c alors *cx1 sinon *cx2], Sem[y=si c alors *cy1 sinon*cy2]
= par (cond(c, Sem [*cx1], Sem[*cx2]), cond(c, Sem[*cy1], Sem[*cy2])

utilisons la distributivité de l'opérateur par par rapport à l'opérateur cond ; Sem1 se réécrit :

Sem1 = cond(c, par (Sem[*cx1], Sem[*cy1]), par (Sem[*cx2], Sem[*cy2]))

Les modules *cx1 et *cy1 sont deux modules indépendants (de même pour *cx2 et *cy2). Utilisons les propriétés de l'opérateur par par rapport à la sémantique d'un module :

Sem1 = cond(c, Sem[fusm (*cx1,*cy1)], Sem[fusm (*cx2,*cy2)])
= Sem [x, y = si c alors fusm (*cx1, *cy1) sinon fusm (*cx2,*cy2)]
= Sem2

IV - 2 -2-2- Définitions itératives

Dans la démarche d'explicitation d'algorithmes proposée, il arrive fréquemment que deux suites résultats aient le même domaine de définition. La règle de fusion d'itérations permet de n'utiliser qu'une seule fois le domaine de définition considéré.

Premier cas : Cas des suites

1 - Définition

```
res = u, v
  u = iter *cu sur l
  v = iter *cv sur l
-------------------------------------------------
u, v = decf iter fusm (*cu, *cv) sur l
```

Avant transformation, l'algorithme définit un couple de suites (u, v) et après transformation, une suite de couples (ui, vi) : l'opérateur decf permet de conserver la structure du résultat.

2 - Conditions d'utilisation :

u et v sont de type SUITE : on n'a aucune contrainte sur u et v ; dans ce cas, la fusion est possible (les objets manipulés sont des objets abstraits indépendants de toute représentation).

3 - Justification sémantique

Sem1 = par (it (*cu) (uo, l), it (*cv) (vo, l))
Sem2 = decf (it (fusm (*cu, *cv)) ((uo, vo), l))

Premier cas : la suite l = svide

par définition de la fonction sémantique it,

Sem1 = par (uo, vo) = (uo, vo)
Sem2 = decf (uo, vo) = (uo, vo)

Deuxième cas : l ≠ svide

u = concat (uo, u')
v = concat (vo, v')
(u, v) = decf (concat (par(uo, vo), par(u', v')))
= par (concat (uo, u'), concat (vo, v'))
= par (u, v)

Cette propriété découle directement de l'application des propriétés de decf par rapport à l'opérateur par.

Deuxième cas : les fichiers

```
res = f1, f2
f1 = iterf *cf1 sur l, sup1
f2 = iterf *cf2 sur l, sup2
```

↓

```
f1, f2 = decf iterf fusm (*cf1, *cf2) sur l, (sup1, sup2)
```

Conditions d'utilisation :

Elles sont liées aux contraintes imposées par le problème à résoudre, ici au support des fichiers.

si sup1 ≠ sup2, on peut effectuer la fusion et l'opérateur decf qui fait passer d'une suite de couples à un couple de suites, maintient la correction du résultat.

si sup1 = sup2, la fusion n'est pas possible.

En effet, par définition du type fichier :

f1 = (u, sup) et f2 = (v, sup) et il existe f3

tel que : | f3 = (w, sup)
| u ⊂ w
| v ⊂ w

avant transformation, res = f1, f2 = (concat (u, v), sup))
après transformation, res'= (concat ((u', v'), (ui, vi)), sup)

les deux suites sont interclassées dans le fichier de support sup.

IV - 2 -3- Composition d'itérations

La composition va dépendre de l'itération définissant la suite intermédiaire.

IV - 2 -3-1- Cas d'une suite complète

Le domaine de définition de l'itération définissant le résultat est une suite elle-même définie par une itération de même type.

1 - Définition :

```
res = u
  u = iter *cu sur l
  l = iter *cl sur ld
```

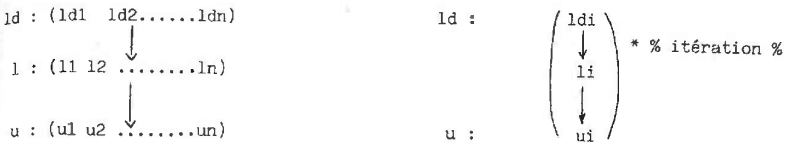
↓

```
  u = iter fusm (*cu, *cl) sur ld
```

Avant transformation, la suite u est définie à partir de la suite l, elle-même définie par ailleurs. On remarque que définition et utilisation de la suite l sont effectuées de manière complétement indépendantes.

La transformation a pour but de supprimer la suite l et de n'en conserver que la définition de son terme général. En effet, chaque terme de la suite u est maintenant défini à partir du terme de même rang de la suite l, mais non à partir de la suite globale l.

Schématiquement, on est passé de deux processus indépendants : production de la suite l, consommation à l'itération du processus : production d'un élément de l, consommation immédiate de cet élément.



Avant transformation

Après transformation

Cette transformation entraîne la suppression de la suite l.

2 - Conditions d'utilisation

- la suppression de la suite l nécessite que cette suite ne soit pas utilisée par ailleurs :

∀ d∈m, d ≠ (u=iter *cu sur l) ⇒ partiegauche (l=iter *cl sur ld) ∉ partiedroite (d)

- la fusion des modules *cu et *cl suppose que la précondition suivante est vérifiée :

fusm (*cu, *cl) estdéfini ⇔ estdéfini (*cu, x) et ⌉ estdéfini (*cl, x)
et estdéfini (*cl, y) et ⌉ estdéfini (lcu, y)

3 - Justification sémantique :

Sem1 = comp (l, it (*cu) (uo, l), it (*cl) (lo, ld)

avec | uo = Sem ⌈init (*cu)⌉
| lo = Sem ⌈init (*cl)⌉
| lg (u) = lg (l)
| lg (l) = lg (ld)

Sem2 = it (fusm(*cu, *cl)) (u'o, ld)

avec | u'o = Sem ⌈init (fusm (*cu, *cl))⌉

Premier cas : ld = svide ⇒ l = svide

Sem1 = comp (l = svide, Sem ⌈init (*cu)⌉, Sem ⌈init (*cl)⌉)
= Sem ⌈fusm (init (*cu), init (*cl))⌉
= Sem2

grâce à la propriété de l'opérateur comp par rapport à la sémantique d'un module.

Deuxième cas : ld ≠ svide ⇒ l ≠ svide

Sem2 = it (fusm (*cu , *cl)) (uo, ld)
= concat (uo, it (fusm (*cu, *cl)) (u'1, queue (ld)))

avec u'1 = Sem [rec (fusm (*cu, *cl))] (uo, prem (ld))

L'utilisation du lemme1 conduit à écrire :

Sem2 = concat (uo, it (fusm(*cu, *cl)) (comp (prem(l), Sem [rec(*cu)] (uo,prem(l)) ,
Sem [rec(*cl)] (lo, prem(ld))), queue(ld)))

Utilisons la distributivité de it par rapport à comp (lemme2) :

Sem2 = concat (uo, comp (prem(l), it (fusm (*cu, *cl)) (Sem [rec(*cu)] (uo, prem(l)),
queue(ld), it (fusm (*cu, *cl)) (Sem [rec(*cl)] (lo,prem(ld)) , queue(ld))))
= concat (uo, comp (prem(l), it(*cu) (uo, queue(l)), it(*cv) (lo, queue(ld))))
= concat (uo, u')
= Sem1

a) Lemme1

Sem [rec (fusm (*cu, *cl))] (uo, prem (ld)) =
comp (prem(l), Sem [rec(*cu)] (uo, prem(l)), Sem [rec(*cl)] (lo, prem(ld)))

Préconditions : | estutilisé (*cu, résultat (*cl))
| résultat (*cl) ∉ résultat (*cu)

Démonstration :

u'1 = Sem [rec (fusm (*cu, *cl)] (uo, prem (ld))

résultat (*cu) = u1

résultat (*cl) = prem (l)

Les préconditions étant vérifiées, nous pouvons utiliser la propriété de l'opérateur comp par rapport à la sémantique d'un module :

u'1 = comp (prem(l), Sem [rec (*cu)] (uo, prem(ld)) , Sem [rec (*cu)] (uo,prem(ld)))
= comp (prem(l), Sem [rec (*cu)] (uo, prem(l)), Sem [rec (*cl)] (lo , prem(ld)))

b) Lemme2 : Distributivité de it par rapport à l'opérateur comp :

it (*cu) (comp (prem(l), u', d), queue(l)) = comp (prem(l), it(*cu) (u', queue(l)),d)

Démonstration :

Utilisons la définition de l'opérateur comp :

it (*cu) (comp (prem(l), u', d), queue(l))
= it (*cu) (λ(iddroite(m, u') U iddroite (m, d)).(u' [prem(l)/d], queue(l)))
= λ (iddroite (m, u') U iddroite (m, d)).it (*cu) (u'[prem(l)/d], queue(l))
= comp (prem(l), it (*cu) (u', queue(l)), d)

IV - 2 -3-2- Cas d'une itération avec condition d'arrêt

1 - Définition :

res = u
u = iter *cu sur l
l = jqa arret iter *cl sur ld
↓
u = jqa arret iter fusm (*cu, *cl) sur ld

2 - Conditions d'utilisation

Elles sont identiques à celles du cas précédent.

3 - Justification sémantique :

La démonstration s'effectuerait de la même manière que pour le cas d'une itération sur une suite complète.

IV - 2 -3-3- Cas du avec

Définition :

res = u
u = jqa arret1 iter *cu avec l
l = jqa arret2 iter *cl avec ld
↓
u = jqa fin iter fusm (*cu, *cl) avec ld

avec fin = arret1 et arret2

Les conditions d'utilisation sont identiques au cas précédent et la justification sémantique s'effectue de la même manière.

IV - 2 -4- Passage d'une itération sur une suite complète à une itération générale

Il ne s'agit pas réellement d'une règle de transformation mais d'une explicitation de la définition itérative:

u = iter * cu sur l sous la forme d'une définition itérative dans laquelle le parcours n'est pas explicite : il est défini au niveau du module résultat.

1 - Définition :

```
res = u
  u = iter *cu sur l
  ------------------------------
  u = jqa arret iter *cu' avec l'
```

Définissons *cu' à partir de *cu :

Nous devons distinguer deux cas :

1 - a - cas où arrêt inclus (correspondant au traitement du dernier élément de la suite l : on n'effectue pas de lecture à l'avance).

```
Init (*cu') = ajounit (init (*cu), l' = l)
Rec (*cu') = ajoutrec (ajoutrec(rec(*cu), l'=queue(l')), arret=l'=svide)
```

2 - a - cas où arret exclu (le dernier élément de la suite l n'est pas traité : on procéde alors à une lecture à l'avance , dans *cu, el est remplacé par Ω(el).

```
Init (*cu') = ajounit (ajounit(init(*cu), l'=queue(l)), arret=l'=svide)
Rec (*cu') = ajoutrec (ajoutrec (rec(*cu), l'=queue(l')),arret=l'=svide)
```

Remarque : Cette transformation d'itération sera utile lorsque la suite l ne sera pas définie par l'utilisitateur mais possédera une définition générale (ex: l est définie par interclassement ou à l'aide de l'opérateur ⊠ : l = l1 ⊠ l2). Elle nous permettra de remplacer, au moment des transformations, cette suite n'ayant pas d'existence physique par le parcours des suites ou fichiers existants qui la composent.

2 - Condition d'utilisation

Cette transformation se fait sans condition.

3 - Justification sémantique (cas où arrêt inclus)

Sem1 = Sem [u = iter *cu sur l]
= it (*cu) (uo, l)

Sem2 = it (*cu') (u'o, l)

Premier cas : l = svide

Sem1 = uo = Sem [init (*cu)]

Sem2 = u'o = Sem [init (*cu')]
= Sem [ajoutinit ((*cu), l = svide)]
= Sem [init (*cu)]
= Sem1

Deuxième cas : l ≠ svide

Sem1 = concat (uo, u1)
= concat (uo, it(*cu) (Sem [rec(*cu)] (uo, prem(l)), queue(l)))

Sem2 = concat (u'o, u'1)
= concat (uo, it(*cu') (Sem [rec(*cu')] (uo, l') , l"))

utilisons la définition de *cu'. Dans ce cas particulier,

```
l" = queue (l)
l' = prem (l)
arret = l = svide
```

Sem2 = concat (uo, it(*cu) (Sem [rec(*cu)] (uo, prem(l)), queue(l)))
= Sem1.

IV - 2 -5- Eclatement d'itération

La suite résultat est définie par une itération dont le domaine de définition est une suite définie par concaténation. Plutôt que de construire la suite intermédiaire, résultat de la concaténation de deux suites, nous définissons le résultat par la concaténation de deux sous-suites résultats définies par itérations sur les deux sous-suites domaines.

1 - Définition :

```
res = u
  u = iter *cu sur l
  l = l1 * l2                    % l = concat (l1, l2) %
  ------------------------------
res = u
  u = ul1 * ul2                  % u = concat (ul1, ul2) %
ul1 = iter *cu1 sur l1
ul2 = iter *cu2 sur l2
```

l est définie par 'morceaux". Eviter sa définition revient à faire éclater le résultat u en "sous-résultats" ul1 et ul2 définis respectivement à partir des sous-suites l1 et l2 constituantes de l.

2 - Conditions d'utilisation :

Après transformation, la suite l n'existe plus : elle ne doit pas être utilisée par ailleurs.

$\forall d \in m,\ d \neq (u = \underline{iter}\ *cu\ \underline{sur}\ l) \Rightarrow partiegauche\ (l = l1 * l2) \not\in partiedroite\ (d)$

Comment sont définis *cu1 et *cu2

*cu1 ‖ Init (*cu1) = Init (*cu [prem(l) remplacé par prem(l1)])
‖ rec (*cu1) = rec (*cu [el remplacé par el1)]

el élément courant de la suite l.

*cu2 ‖ Init (*cu2) = ajoutinit(init(*cu [prem(l) remplacé par prem(l2)]), prem (ul2) = der (ul1))
(cette précaution est nécessaire pour préserver la correction du résultat).
‖ rec (*cu2 = rec (*cu [el remplacé par el2])

Remarque : Le module *cu est dupliqué moyennant certaines substitutions. Dans un cas, tout terme de la suite l est remplacé par un terme de la suite l1 et dans l'autre par un terme de la suite l2.

3 - Justification sémantique :

Sem1 = $\underline{it}$ (*cu) (uo, l1 * l2)

Sem2 = Sem [u = ul1 * ul2]
= concat (Sem [ul1 = $\underline{iter}$ *cu1 $\underline{sur}$ l1], Sem [ul2 = $\underline{iter}$ *cu2 $\underline{sur}$ l2])
= concat ($\underline{it}$ (*cu1) (ul1o, l1), $\underline{it}$ (*cu2) (ul2o, l2))

a) l = l1 * l2 = svide :

Sem1 = $\underline{it}$ (*cu) (uo, svide) = Sem [init (*cu)]

Sem2 = concat ($\underline{it}$ (*cu1) (ul1o, svide), $\underline{it}$ (*cu2) (ul2o, svide))
= par (Sem [init (*cu1)], Sem [init (*cu2)])
= Sem [fusm (init (*cu1), init (*cu2))]

en utilisant la propriété de la sémantique par rapport à l'opérateur par :

Sem2 = Sem [fusm (init (*cu [prem(l) ⟶ prem(l1)]), init (*cu [prem(l) ⟶ prem(l2)]))]
= Sem [init (*cu)] car l1 = l2 = svide
= Sem1

b) l1 * l2 ≠ svide

Sem1 = concat (uo, u')
= concat (uo, $\underline{it}$ (*cu) (Sem [rec(*cu)] (u1, prem (queue(l1))), concat (queue (l1), l2)))
= concat (uo, concat (u1, $\underline{it}$(*cu) (Sem[rec(*cu)] (u1, prem (queue(l1))), concat (queue (queue(l1)), l2)))

avec u1 = Sem [rec(*cu)] (uo, prem(l1))

Sem1 = concat (uo, concat (u1, concat (u2...., concat (ulg (l1), $\underline{it}$ (*cu) (Sem[rec(*cu)] (ulg (l1), prem (l2)) , queue (l2))...)

En utilisant le lemme1 , Sem1 se réécrit :

Sem1 = concat ($\underline{it}$ (*cu1) (uo, l1), $\underline{it}$ (*cu) (Sem[rec(*cu)] (ulg(l1), prem(l2)) , queue(l2)))

Remarque : Le deuxième terme de concat, ulg(l1) désigne l'élément défini à partir du dernier élément de l1. Par définition de init (*cu2), cet élément désigne aussi l'élément initial de la suite ul2 soit ul2o.

donc : $\underline{it}$(*cu) (Sem[rec(*cu)] (ul2o, prem(l2)) , queue(l2)) définit la suite u à partir du module *cu et de la suite domaine de définition l2. Ceci correspond à :

$\underline{it}$ (*cu2) (ul2o, l2)

Sem1 = concat (ul1, $\underline{it}$(*cu2) (ul2o, l2))
= concat (ul1, ul2)
= Sem2

Lemme1 :

$\underline{it}$ (*cu1) (uo, l1) = uo, concat(u1, concat(u2.... ulg(l1))...))

avec u1 = Sem [rec(*cu)] (uo, prem(l1))
u2 = Sem [rec(*cu)] (u1, prem (queue(l1)))
ulg(l1) = Sem [rec(*cu)] (ulg(l1 -1), queue(l1))

Ce lemme découle directement de la définition de la fonction sémantique $\underline{it}$ appliquée au module *cu et à la suite l1, cé qui correspond par définition au module *cu1.

IV - 2 -6- Règle de décomposition

Le domaine de définition de l'itération n'est pas une suite linéaire à un seul niveau : elle est définie à l'aide de l'opérateur aplat.

1 - Définition

u = iter *cu sur aplat (l)

↓

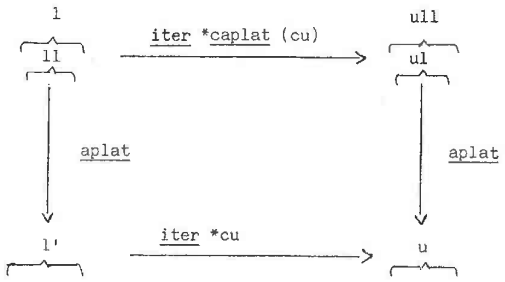
u = aplat iter *caplat (cu) sur l

l est une suite de suites d'éléments. Par exemple, l peut être définie par :

| l = iter *cl sur ll
| ll = donnée

ll est une suite simple.

aplat (l) ramène tous les éléments au même niveau et définit une suite linéaire d'éléments. Exprimons graphiquement la relation entre les diverses suites en présence :



relations entre les suites u et l

Remarque : La règle de décomposition exprime la commutativité des opérateurs iter et aplat mise en évidence sur le diagramme.

Relations entre les suites :

lg (l) = lg (ull) = q
lg (lli) = lg (uli) = r_i
lg (l') = lg (u) = p

aplat (l)k = lli de lj avec $k = \sum_{n=1}^{j-1} r_{n+i}$

Remarque : Cette transformation évite la construction effective de la suite définie par aplat (l).

ull = iter *caplat (cu) sur l désigne une suite résultat définie en bijection avec la suite l : à chaque terme de l (en fait, une suite d'éléments) correspond une suite d'éléments résultat. L'opérateur aplat appliqué à ull (u = aplat ull) permet de conserver la correction du résultat.

2 - Conditions d'utilisation

- La suite aplat (l) disparait après transformation : elle ne doit pas être utilisée par ailleurs.

$\forall d \in m,\ d \neq (u = \text{iter } *cu \text{ sur aplat } (l)) \Rightarrow \text{aplat } (l) \notin \text{partiedroite } (d)$

- Définition du module *caplat (cu)

Correspond au module définissant un terme de la suite ull, soit en fait une suite, en bijection avec un élément de l.

Comment le définir ?

Intuitivement, on va introduire une itération pour chaque sous-suite de la suite l, donc pour chaque ll.

*caplat (cu)
ul = iter *cu sur ll

Le module *cu a été "descendu d'un niveau" : il s'applique maintenant à chaque terme de la suite ll. L'itération englobante introduite permet de traiter chaque terme de la suite l, soit ll.

3 - Justification sémantique

Sem1 = it (uo, aplat (l))
= it (Sem [init (*cu)], aplat (l))

Sem2 = aplat (it (u'o, l))
= aplat (it (Sem [init (*caplat (cu))], l)

Premier cas : l = svide

Sem2 = aplat (u'o)
= aplat (Sem [init (*caplat (cu))])
= aplat (Sem [init (ul = iter *cu sur ll)])

= aplat (Sem [init (*cu)]) % car ll = svide %
= uo
= Sem1

Deuxième cas : l ≠ svide

Sem1 = concat (uo, u')

avec :

u' = it (*cu) (Sem [rec(*cu)] (uo, prem(aplat(l)), queue(aplat (l)))
prem (aplat(l)) = prem (prem(ll))
queue (aplat(l)) = concat (queue (prem(ll), queue(ll)))

Sem2 = aplat (concat (u'o, u'1))
= concat (aplat (u'o), aplat (u'1)) % propriété de l'opérateur aplat %
= concat (uo, aplat u'1) % dans le cas où l = svide, nous avons démontré aplat (u'o) = uo %
= concat (uo, u') % en utilisant le lemme 3 %
= Sem1

lemme3 : u' = aplat (u'1)

avec :

u' = it (*cu) (Sem [rec(*cu)] (uo, prem(aplat(l))), queue (aplat(l)))
u'1 = it (*caplat (cu)) (Sem [rec(*caplat (cu))] (uo, prem(l)), queue (l)))

Démonstration :

*caplat (cu) se réduit à la seule définition itérative :

ul = iter *cu sur ll

Remplaçons ce module par sa définition dans u'1.

u'1 = it ((it (*cu) (Sem [rec(*cu)] (uo, prem(ll)), queue(ll))) (uo,prem(l)),queue(l))
aplat (u'1) = it(*cu) (Sem[rec(*cu)] (uo, prem (aplat (l))), queue (aplat (l)))
= u' % en utilisant le lemme 4 sur la commutativité de aplat par rapport à it %

lemme 4 : Commutativité de l'opérateur aplat par rapport à it

aplat (it (it (*cu) (uo, ll)) (uo, l)) = it (*cu) (uo, aplat (l))

Démonstration :

Premier cas : l = svide ⇒ ll = svide

aplat (it (it (*cu) (uo, svide), (uo, svide))
= aplat (it (Sem [init (*cu)], (uo, svide))
= aplat (uo)
= it (*cu) (uo, svide)
= it (*cu) (uo, aplat (l))

Deuxième cas : l ≠ svide

aplat (it (it (*cu) (uo, ll)) (uo, l))
= aplat (it (concat(uo, it(*cu) (Sem [rec(*cu)] (uo, prem(ll)))), (uo, l)))
= aplat (concat (uo, it(it(*cu) (Sem [rec(*cu)] (uo, prem(ll))), (uo, l))))
% par définition %
= concat (aplat (uo), aplat (it(it(*cu) (Sem[rec(*cu)] (uo, prem(ll))), (uo,l))))
% propriété de l'opérateur aplat par rapport à concat %
= concat (uo, aplat (it(it(*cu) (Sem[rec(*cu)] (uo, prem(ll))), (uo, l))))

Utilisons à nouveau la définition de it et la propriété de aplat par rapport à concat.

= concat (uo, concat (u1, ...(un, aplat(it(*cu)Sem[rec(*cu)]) (un, der (ll))) ...)
= concat (uo, it(*cu) (Sem[rec(*cu)] (uo, prem(aplat(l))), queue(l))
= it (*cu) (uo, aplat (l))

IV - 3 - Stratégie d'utilisation des règles de transformations

IV - 3 -1- Objectif

Notre objectif consiste à minimiser le nombre d'intermédiaires introduits (principalement de type suite) en appliquant les règles de transformations proposées, ceci dans un nombre d'étapes minimum.

Pour cela, nous disposons d'un algorithme écrit déductivement selon la méthode proposée et de l'ensemble des règles de transformations T portant sur les définitions du langage.

T se compose des règles suivantes :

- dépliage et pliage
- fusion de définitions conditionnelles
- fusion de définitions itératives
- composition d'itération
- éclatement d'itération
- décomposition d'itération
- passage d'une itération sur une suite complète à une itération générale

et des propriétés de la fonction écrire :

- par rapport à la concaténation de deux suites
- par rapport à l'itération

qui seront considérées comme des règles de transformation.

Problème : Y a-t-il une stratégie permettant d'obtenir un algorithme "réduit" en un minimum d'étapes ?

Cette stratégie est-elle unique ?

IV - 3 -2- Explication de la stratégie

IV - 3 -2-1- Idées générales

Considérons le graphe G = (T, Γ) formé :

- de l'ensemble fini T des règles de transformations
- de la relation binaire Γ dans T définie par :

  x Γ y $\Leftrightarrow$ la transformation y succède la transformation x.

L'utilisation de ces règles sur des exemples nous a amené à constater l'existence d'une relation de dépendance entre :

- le dépliage et l'éclatement d'itération
- le dépliage et la décomposition d'itération
- le dépliage et les propriétés de l'opération écrire
- le pliage et la fusion d'itérations.

Remarque : L'élaboration de la stratégie proposée est empirique. Nous n'avons pas de justifications formelles à fournir.

IV - 3 -2-2- A partir d'un exemple

Reprenons l'algorithme initial concernant l'exemple des télégrammes (p. 28, 29). L'application des règles s'effectue de manière descendante en commençant par le module principal.

L'application de la règle du dépliage à la définition de résultat (en remplaçant lmessage par sa définition) conduit à la nouvelle définition de résultat (Remarque : comme il n'y a qu'un seul fichier de sortie, son nom et le support n'ont pas été noté dans la définition).

résultat = écrire (iter *cmessage sur stels)

Ce qui nous entraîne à utiliser les propriétés de l'opération écrire par rapport à l'itération. (cf. paragraphe III - 2 -3-).

résultat = iter *cmessage'sur stels.

Quelle transformation peut-on encore effectuer dans le module principal ?

résultat est défini par itération sur la suite stels

stels est elle-même définie par itération.

Nous pouvons composer ces deux itérations et définir le résultat par :

résultat = jqa telvide iter*(cmessage', ctel) avec scaractères.

Remarque : Dans cet exemple, on aurait pu commencer par appliquer la composition des itérations définissant lmessage et stels :

lmessage = jqa telvide iter *(cmessage, ctel) avec scaractères.

Puis, dans un deuxième temps déplier la définition de résultat et appliquer les propriétés de écrire par rapport à l'itération.

Dans les deux cas, le résultat final est identique.

Les transformations possibles dans le module principal ont été effectuées. Elles nous permettent de tenir compte :

- d'une relation de dépendance entre l'utilisation des propriétés de l'opération écrire et l'application du dépliage
- de l'autonomie de la règle de composition.

L'illustration des autres relations de dépendance sera mise en valeur sur des exemples proposés au paragraphe suivant (V).

IV - 3 -2-3- Stratégie

Les règles d'éclatement et de décomposition d'itération ont pour but de transformer une itération dont le domaine de définition est une suite

définie à l'aide d'opérateurs différents de l'itération en une itération dont le domaine de définition est une suite définie par une itération : ces règles de transformations sont une préparation de l'algorithme à l'application de la règle de composition d'itération.

Règles à appliquer en fonction de la définition de la suite l :

u = iter *cu sur l

a) si l = aplat (l1) on applique la décomposition

b) si l = l1 * l2 on applique l'éclatement d'itération

c) si l = iter *cl sur ld on applique la composition d'itération.

Remarque : Ces règles ne sont appliquées que si les préconditions nécessaires sont vérifiées.

1 - Description du graphe G1 = (T1, ⌜)

T1 représente l'ensemble des transformations dépliage, propriétés de l'opération écrire, éclatement d'itération et composition d'itération.

La première règle à appliquer est celle du dépliage. En fonction de la nouvelle définition obtenue après transformation, plusieurs cas sont possibles. Soit x = f(l) cette nouvelle définition.

1 - 1 - C'est une définition simple

a) f est l'opération écrire : on applique alors les propriétés de écrire.

b) f est quelconque : la transformation est terminée.

1 - 2 - C'est une définition itérative de la forme :

x = iter *cx sur l

La transformation à appliquer dépend de la définition de la suite l.

a) l = l1 * l2 : on applique alors la règle d'éclatement d'itération

b) l = aplat (ld) : on applique la règle de décomposition d'itération.

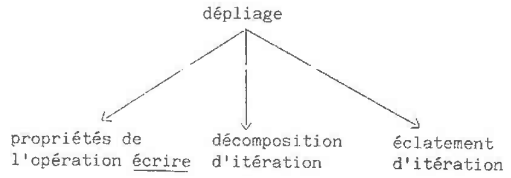


fig. 1 : représentation graphique du graphe G1

2 - Description du graphe G2 =(T2, ⌜)

T2 représente les transformations de pliage et de fusion d'itérations. La règle de pliage est essentiellement utilisée dans notre cadre en vue d'appliquer la règle de fusion d'itérations:

ex : u = der iter *cu sur l

v = iter *cv sur l

Pour respecter des contraintes énoncées dans le problème, il se peut que la suite l ne puisse être parcourue plusieurs fois, auquel cas, on est obligé de fusionner les deux itérations et donc de définir une nouvelle suite u' telle que :

u = der u'

u'= iter *cu sur l

L'application de la règle du pliage précède celle de fusion.

3 - Description du graphe G3 = (T3, ⌜)

T3 représente la règle de composition d'itérations.

4 - Détermination du graphe minimal G à partir de G1, G2, G3

Enumérons les diverses possibilités :

G1 G2 G3

G1 G3 G2

G2 G1 G3

G2 G3 G1 G3 (après éclatement ou décomposition, la composition d'itérations est nécessaire)

G3 G1 G2 G3

G3 G2 G1 F3

Les cas de figure correspondant à un coût minimal du point de vue du nombre de règles appliquées impliquent le parcours du graphe G1 avant G3. Deux cas sont possibles :

G1 G2 G3 et G2 G1 G3.

Remarque : L'application du groupe G2 de règles de transformation doit-il précéder ou non l'application du groupe de règles G1 ?

Appliquer la fusion de définitionsavant l'éclatement d'une itération, donc d'un module, évite d'appliquer cette règle deux fois (l'éclatement d'un module équivaut à sa duplication) : ce qui privilégie la position de G2 par rapport à G1.

Le chemin de coût minimum correspond à G2 G1 G3.

La stratégie de transformation, au niveau d'un module peut être schématisée par :

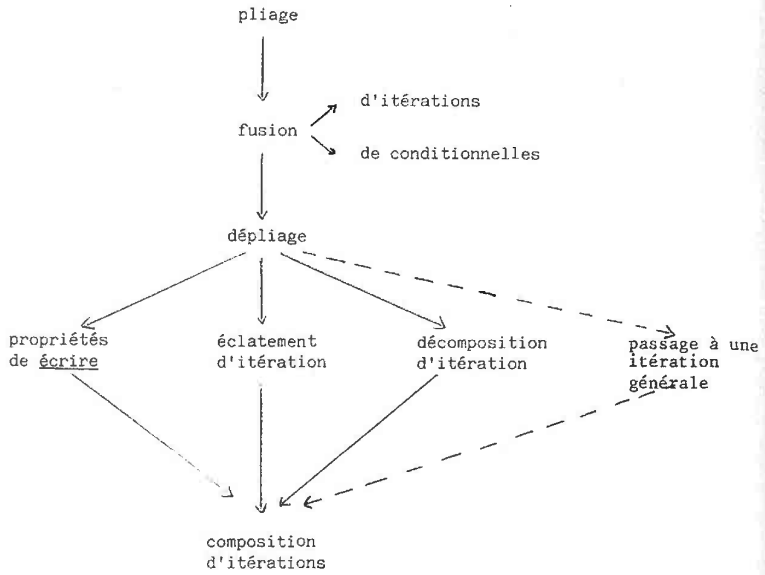


fig. 2 : stratégie de transformation d'un module

Cette stratégie s'applique de manière descendante dans un algorithme.

Au niveau d'un module, on essaie d'appliquer récursivement chacune des transformations dans l'ordre établi. Dès que toutes les transformations possibles ont été effectuées sur ce module, le processus est itéré sur les modules fils.

IV - 3 -3- Application sur l'exemple des télégrammes

Le point de départ est l'algorithme initial présenté p. 28-29.

1 - Application de la stratégie dans le module principal

La première règle applicable consiste à déplier résultat soit à remplacer dans sa définition la suite lmessage par sa définition, les pré-conditions nécessaires étant vérifiées.

résultat = écrire (iter *cmessage sur stels)

Nous pouvons alors appliquer les propriétés de la fonction écrire par rapport à l'itération :

résultat = iter *cmessage' sur stels

avec *cmessage' = ajoutrec (*cmessage, {message' = écrire message})

Cette transformation effectuée, résultat est maintenant défini par une itération sur la suite stels et la suite stels est elle-même définie par une itération. D'autre part, la suite stels n'est pas utilisée par ailleurs : les préconditions nécessaires à l'application de la règle de composition d'itérations sont vérifiées.

Le module principal devient :

résultat = jqa telvide iter *(cmessage', ctel) avec scaractères

Les transformations possibles dans le module principal ont été effectuées. Le gain est caractérisé par la suppression des deux suites lmessage et stels, dont l'existence n'était pas nécessaire pour l'obtention du résultat cherché.

Remarque : *(cmessage', ctel) est une notation abrégée de
fusm (*cmessage', *ctel) : fusion des deux modules *cmessage' et *ctel.

2 - Le module *(cmessage', ctel)

Ce module contient les définitions :

message' = écrire message	
message = ...	Définitions de *cmessage
totmot = ...	
nbmot, ...	
tel, telvide = ...	Définition de *ctel

Appliquons la règle de dépliage à la définition de message'

message' = écrire (untel, totmot, tlong)

Cette règle peut à nouveau être appliquée à cette définition en remplaçant totmot, tlong par l'expression der (nbmot, nblong)

message' = écrire (untel, der (nbmot, nblong))

Dans cette même définition, on peut encore appliquer les propriétés de l'opération écrire par rapport à la concaténation de deux suites (les suites étant réduites ici à un seul élément).

message' = écrire untel, écrire der (nbmot, nblong)

untel est une suite définie par itération : écrire untel correspond à l'écriture globale de cette suite. Améliorer l'efficacité de cet algorithme peut être réalisé par l'écriture terme à terme des éléments de cette suite. Pour cela, nous appliquons la règle de pliage à message' relativement au premier terme de l'expression.

message' = suntel, écrire der (nbmot, nblong)
suntel = écrire untel
untel, nbmot, nblong = iter *compte sur tel
tel, telvide = jqa fintel iter *cmot avec scaractères

Nous pouvons maintenant déplier suntel et appliquer la propriété de l'opération écrire par rapport à l'itération :

suntel , nbmot, nblong = iter *compte' sur tel

avec *compte' = ajoutrec (*compte, {unmot' = écrire unmot})

La dernière transformation applicable dans ce module consiste à composer les deux itérations. Ce module s'écrit alors :

message = suntel, écrire der (nbmot, nblong)
suntel, nbmot, nblong, telvide = jqa fintel iter *(compte', cmot) avec scaractères

Remarque : Les notations ' ont disparu : après transformations, les objets initiaux ont été éliminés. Pour être rigoureux, nous devrions conserver une notation différente car le type des objets a été modifié.

Les transformations ont été effectuées et ont abouti à la suppression de deux suites intermédiaires :

- untel qui correspondait à la suite des mots du télégramme courant après modification
- tel qui correspondait à la suite des mots du télégramme source.

3 - Le module *(compte', cmot)

unmot' = écrire unmot
{définitions de *compte}
{définitions de *cmot }

Appliquons la règle de fusion à mot et f

mot, f = scaractères de d à car = ' ' exclu

On peut déplier unmot' :

unmot', nbmot : si nbcar=4 alors si mot='STOP' alors | unmot = écrire
nbmot = ω nbmot

si mot='ZZZZ' alors | unmot = < >
nbmot = ω nbmot

sinon | unmot = écrire mot
nbmot = ω nbmot + 1

Autres transformations possibles :

Peut-on composer les itérations définissant nbcar et mot (et supprimer la suite mot) ?

Non, cette transformation n'est pas réalisable car les préconditions nécessaires à son application ne sont pas réalisées à savoir la suite mot est utilisée dans une autre définition du module. On est amené à comparer mot avec 'ZZZZ' et 'STOP'.

Remarque : Parmi les suites intermédiaires introduites, la suite mot est la plus grande composante nécessaire au traitement demandé. La composante de base donnée : le caractère n'est pas adaptée.

4 - Le module *ccar

Aucune transformation n'est possible.

5 - Algorithme final :

- telvide suite de booléens - *(cmessage, ctel) définit l'impression d'un triplet correspondant au télégramme courant - scaractères suite de caractères donnée	résultat = jqa telvide iter *(cmessage, ctel) avec scaractères

..../....

..../....

*(cmessage, ctel)	
- message élément courant de la suite résultat - nbmot suite des nombres de mots du télégramme traité - nblong suite des nombres de mots trop longs - suntel suite des mots imprimés - fintel suite de booléens déterminant la fin d'un télégramme - *(compte, cmot) détermine les nombres de mots et imprime le mot traité	message = suntel, écrire der (nbmot, nblong) suntel, nbmot, nblong, telvide = jqa fintel iter *(compte, mot) avec scaractères
*(compte, cmot)	
- unmot mot imprimé - nbcar entier nombre de caractères de mot - mot suite de caractères - mott suite de caractères, mot tronqué à 12 caractères - d caractère début de mot - f caractère fin de mot - tête fonction d'accès au premier caractère de la suite - finbloc booléen	nbmot, unmot : si nbcar = 4 alors si mot = 'STOP' alors | unmot = écrire nbmot = ω nbmot si mot = 'ZZZZ' alors | unmot = < > nbmot = ω nbmot sinon | unmot = écrire mott nbmot = ω nbmot + 1 nblong, mott : si nbcar < 12 alors | nblong = ω nblong mott = mot sinon | nblong = ω nblong + 1 mott = mot [1, 12] fintel = mot = 'ZZZZ' nbcar = der iter *ccar sur mot mot, f = scaractères de d à car = ' ' exclu d = prem car dans scaractères depuis ω f telque car # ' ' telvide = fintel et ω fintel scaractères = si finbloc alors donnée prem fintel = faux prem f = tête (scaractères) prem nbmot = 0 prem nblong = 0
*ccar	
	nbcar = ω nbcar + 1 prem nbcar = 0

algorithme final: les télégrammes

IV - 4 - Conclusion

La stratégie proposée a été appliquée manuellement sur l'exemple des télégrammes. On remarque que la démarche suivie est lourde, fastidieuse et peut être source d'erreurs : on a été conduit à réécrire plusieurs fois des mêmes fragments d'algorithmes.

Pour appliquer les transformations en utilisant la stratégie proposée, on doit regarder la forme des définitions de l'algorithme, les comparer à un nombre fini de schémas donnés, et, en fonction de la comparaison et de la vérification de préconditions, générer une ou plusieurs nouvelles définitions.

Cette démarche est automatisable. Elle nécessite d':

- implémenter les diverses règles de transformations
- implémenter la vérification des préconditions
- implémenter la reconnaissance des schémas susceptibles d'être transformés dans un algorithme
- implémenter les modifications de définitions dues aux transformations.

Une maquette de ce système est présentée au chapitre 4.

V - EXEMPLES D'APPLICATION DE LA METHODE

Nous traiterons deux exemples pour lesquels les suites intermédiaires introduites ne sont pas simplement définies par itération mais à l'aide des opérateurs sur les suites.

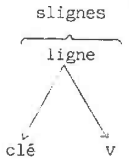
V - 1 - Exemple 1 (dû à ABRIAL)

L'énoncé est donné dans le chapitre 1, paragraphe III (exemple h)

V - 1 -1- Construction de l' algorithme initial

V - 1 -1-1- Structures imposées par l'énoncé

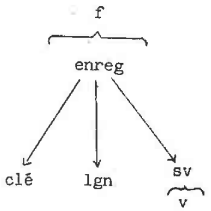
La structure du résultat est schématisée par :



slignes est une suite de ligne (avec des lignes vides pour tenir compte des sauts de page). Une ligne est soit vide, soit constituée par le couple (clé, v) avec v défini de deux manières possibles :

- c'est la valeur d'un article ou
- c'est la somme des lgn valeurs de l'enregistrement.

La structure des données peut être représentée par :



f est une suite d'enregistrements noté enreg, chaque enreg se compose d'un triplet (clé, lgn, sv) où :

- clé désigne la clé de l'enregistrement
- lgn la longueur
- sv une suite de valeurs v telle que lg(sv) = lgn.

Remarque : Ces deux structures ne sont pas immédiatement compatibles, elles ne possèdent pas le même nombre d'éléments. Pour définir le résultat, nous distinguerons la définition des lignes de la contrainte imposée par les sauts de page.

V - 1 -1-2- Définition du résultat

1 - Le résultat est défini comme l'impression d'une suite de lignes. Nommons slignes cette suite :

résultat = écrire slignes

2 - Définition de slignes : Elle désigne la suite dont chaque élément appelé ligne est un couple (clé, v).
slignes peut se définir par une itération sur la suite des couples (clé, v), nommée lcouples, constituée à partir des valeurs de chaque élément de l'enregistrement traité.

slignes = iter *clignes sur lcouples

*clignes représente le module définissant une ligne à partir d'un élément de lcouples.

3 - Le module *clignes
ligne, terme général de slignes est défini en fonction du nombre de lignes déjà constituées :

| ligne = si ∂n = 3 alors ligne = (couple, saut)
| sinon ligne = (couple, < >)
|
| n = si ∂n = 3 alors n = 0 sinon n = ∂n + 1
|
| prem n = 0

- n représente le nombre de lignes constituées depuis le dernier saut de page
- couple désigne l'élément courant (clé, v) de lcouples
- saut indique qu'un saut de page doit être prévu dans ce cas là.

V - 1 -1-3- Définition de la suite intermédiaire introduite

1 - lcouples est une suite de couples(clé, v), avec v défini par :

- soit la valeur d'un élément d'un enregistrement
- soit la somme des valeurs d'un enregistrement

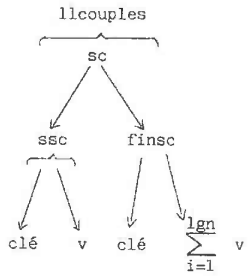
ex : (c1, v1), (c1, v2)··· (c1, v lgn), (c1, $\sum_{i=1}^{lgn}$ vi), (c2, v1), (c2, v'2), (c2, v'lgn'), (c2, $\sum_{i=1}^{lgn'}$ vi'), ...

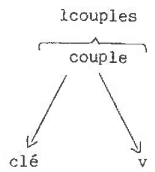
Remarque : nous pouvons à chaque enregistrement associer la suite des couples (clé, v), suite de longueur (lgn + 1): soit llcouples cette suite.
A partir du fichier f, nous pouvons constituer une suite de suites de couples (clé, v), nommée lcouples

| llcouples = iter *ccouple sur f
| f = donnée

Quel est le lien entre la suite llcouples et la suite lcouples ?

Donnons une représentation graphique de ces deux suites :





Nous pouvons définir lcouples à partir de llcouples avec l'opérateur aplat :
aplat :SUITE X SUITE ——→ SUITE

| lcouples = aplat (llcouples)
| llcouples = iter *ccouple sur f
| f = donnée

2 - Le module *ccouple

Chaque élément de llcouples, noté sc, est une suite définie par la concaténation de deux suites, dont la deuxième est réduite à un seul élément.

*ccouple définit cet élément en bijection avec un enregistrement de f.

| sc = ssc *finsc

Comment définir ssc ?

A chaque valeur de l'enregistrement courant, nous associerons le couple (clé, v). Un enregistrement comporte lgnvaleurs :

| ssc = iter *cenreg sur 1 ——> lgn

*cenreg associe à la ième valeur de l'enregistrement le couple (clé, v)

Comment définir finsc ?

C'est un couplé (cle, stv) où stv représente ici la somme des valeurs de l'enregistrement traité

| finsc = (clé, stv)
| stv = der iter *som sur 1 ——> lgn

3 - Le module *cenreg

*cenreg
essc = (clé, v)

4 - Le module *som

*som
stv = ω stv + v prem stv = 0

V - 1 -1-4- Première version de l'algorithme

Dans cet exemple, la suite intermédiaire n'est pas directement déduite par itération sur la suite donnée. Elle nécessite un traitement supplémentaire.

L'algorithme s'écrit :

slignes suite de ligne slignes définit une ligne en bijection avec un élément de lcouples lcouples suite de couples (clé, v) llcouples suite de suite de couples *couples associe à chaque enregistrement une suite de couples définie par concaténation f suite d'enregistrements de la forme (clé, lgn suite de v)	résultat = écrire slignes slignes = iter *clignes sur lcouples lcouples = aplat (llcouples) llcouples = iter *ccouples sur f f = donnée
*clignes	
ligne élément courant de slignes n entier nombre de lignes constituées depuis le dernier saut de page couple élément courant de lcouples	ligne = si ωn=3 alors ligne = (couple, saut) sinon ligne = (couple, < >) n = si ωn=3 alors n=0 sinon n = ωn + 1 prem n = 0
*ccouples	
sc suite de couples définie par concaténation ssc suite de couples (clé, v) de l'enregistrement courant finsc couple dont la valeur représente la somme des valeurs de l'enregistrement *cenreg module constituant le couple courant de l'enregistrement lgn entier longueur de l'enregistrement stv entier somme des v de l'enregistrement *som calcule stv	sc = ssc *finsc ssc = iter *cenreg sur 1 ——> lgn finsc = (clé, stv) stv = der iter *som sur 1——> lgn
*cenreg	
clé entier de l'enregistrement essc couple v entier valeur de l'enregistrement	essc = (clé, v)
*som	
	stv = ω stv + 1 prem stv = 0

algorithme initial

V - 1-2- Obtention de l'algorithme final

Appliquons la stratégie proposée au paragraphe IV à l'algorithme initial.

1 - Application dans le module principal

Le dépliage du résultat et l'application des propriétés de écrire par rapport à l'itération conduisent à écrire :

```
|résultat = iter *clignes' sur lcouples
```

avec *clignes' = ajoutrec (*clignes, { ligne' = écrire ligne })

L'étape de transformation suivante consiste à déplier résultat (en remplaçant lcouples par sa définition), les préconditions nécessaires étant vérifiées:

```
|résultat = iter *clignes' sur aplat (llcouples)
```

Nous pouvons appliquer les propriétés de aplat soit la règle de décomposition d'itération. Par définition, le résultat devient :

```
|résultat = aplat iter *caplat (clignes') sur llcouples
```

avec

```
*caplat (clignes')
slignes' = iter *clignes' sur sc
```

Nous pouvons maintenant composer les deux itérations du module principal :

```
|résultat = aplat iter *(caplat(clignes'), ccouple) sur f
|f = donnée
```

Toutes les transformations possibles ont été effectuées dans ce module et ont conduit à la suppression de trois suites intermédiaires:slignes, lcouples et llcouples.

Nous devons noter l'introduction d'une nouvelle suite intermédiaire notée slignes' correspondant à une sous-suite du résultat final.

2 - Le module *(caplat (clignes'), ccouple)

```
|slignes' = iter *clignes' sur sc
|{ définitions de *ccouple }
```

Appliquons la règle du pliage à la définition de stv:

```
|stv = der st
|st = iter *som sur 1 ——> lgn
```

qui nous permet de fusionner les définitions de ssc et st:

```
|ssc,st = iter *(cenreg, som) sur 1——>lgn
```

La règle du dépliage appliquée à la définition de slignes' entraîne :

```
|slignes' = iter *clignes' sur (ssc *finsc)
```


Le domaine de définition de cette itération est une suite définie par concaténation : appliquons la règle d'éclatement d'itération.

```
|slignes' = slignel  *ligne2
|slignel' = iter  *clignes1' sur ssc
|ligne2   = iter  *cligne2' sur finsc
```

avec *clignes1' = *clignes' [couple est remplacé par essc]
*cligne2' = *clignes' [couple est remplacé par finsc]

Remarque : La suite finsc est réduite à un seul élément. L'itération consiste alors à expliciter *cligne2'

Récapitulons les définitions de *(caplat(clignes'), ccouple):

```
slignes' = slignel' *ligne2'
slignel' = iter *clignes1' sur ssc
ligne2', n = si ω̂n=3 alors |ligne2' = (écrire finsc, saut)
                           |n = 0
           sinon           |ligne2' = (écrire finsc, < >)
                           |n = ω̂n + 1
stv = der st
ssc, st = iter *(cenreg, som) sur 1 ———> lgn
finsc = (clé, stv)
```

Nous pouvons déplier ligne2'(remplacer finsc par sa définition) et composer les deux itérations :

```
|slignel', st = iter *(clignes1', cenreg, som) sur 1 ——>lgn
```

Remarque : Lorsque nous avons décidé de dupliquer le module *clignes', nous avons appliqué la stratégie de transformation dans ce module là avant de le dupliquer pour éviter d'avoir à effectuer deux fois les mêmes transformations.

Les transformations possibles dans ce module étaient la fusion des deux définitions conditionnelles :

```
|ligne' = écrire ligne
|ligne, n = si ω̂n=3 alors |ligne = (couple, saut)
|                         |n = 0
|            sinon        |ligne = (couple, < >)
|                         |n = ω̂n + 1
```

et l'application de la règle du dépliage à ligne' :

```
|ligne', n = si ω̂n=3 alors |ligne' = (écrire couple, saut)
|                          |n = 0
|             sinon        |ligne' = (écrire couple, < >)
|                          |n = ω̂n + 1
```

3 - Le module *(clignes'1, cenreg, som)

ligne1', n = si ω n=3 alors | ligne1' = (écrire essc, saut)
                           | n = 0
            sinon | ligne1' = (écrire essc, < >)
                  | n = ω n + 1

essc = (clé, v)

st = ⍵st + 1

prem st = 0

La seule transformation possible consiste à remplacer essc par sa définition.

4 - Algorithme final

Remarques: Les transformations effectuées ont modifié la manière de définir le résultat partiel. De manière globale, il n'a pas été modifié, grâce à l'opérateur aplat.Les propriétées de cet opérateur permettent de reporter la définition du résultat (initialement le module *clignes) au niveau de chaque constituant de base définissant la suite intermédiaire introduite (soit llcouples) . Il évite la construction effective de cette suite intermédiaire en utilisant chacun de ses termes lors de sa définition.

- résultat suite des lignes imprimées *(clignes, ccouple) associe à un enregistrement une suite de lignes - f suite donnée	résultat = aplat iter *(clignes, ccouple) sur f f = donnée
*caplat (lignes, ccouple)	
- slignes suite définie par concaténation - slignel suite correspondant à l'impression de chaque valeur de l'enregistrement - lgn entier nombre de valeur dans l'enregistrement *(clignel, cenreg, som) associe à une valeur l'impression correspondante - st entier somme des v de l'enregistrement - stv suite des sommes des v - ligne2 impression d'une ligne comportant la somme de tous les v de l'enregistrement - n entier nombre de lignes constituées depuis le dernier saut de page	slignes = slignel * ligne2 slignel, stv= iter *(clignel, cenreg, som) sur 1 ——→lgn st = der stv ligne2, n = si ω n=3 alors | ligne2 = (écrire(clé,st)saut) | n = 0 sinon | ligne2 = (écrire(clé,st) < >) | n = ω n + 1 prem n = 0
*(clignel, cenreg, som)	
- ligne1 élément de slignel - clé entier de l'enregistrement - v entier valeur de l'enregistrement	ligne1, n = si ω n=3 alors | ligne1 = (écrire(clé,v) saut) | n = 0 sinon | lignes = (écrire(clé,v) < >) | n = ω + 1 stv = stv + 1 prem stv = 0

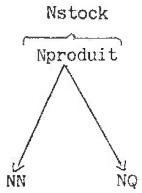
Exemple1 : algorithme final

V - 2 - Exemple 2 : Les mises à jour

L'énoncé est donné dans le chapitre 1 , paragraphe III (exemple i).

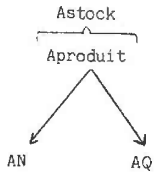
V- 2 -1- Construction d'un premier algorithme

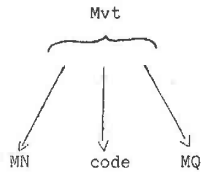
V - 2 -1-1- Structures des données



structure des résultats

-Nstock est une suite (ou fichier) des stocks mis à jour. Chaque élément Nproduit se compose du couple (NN, NQ) où NN désigne le numéro du produit et NQ la quantité en stock. Elle est classée par ordre croissant des numéros de produit.





structure des données

- Astock est la suite des anciens stocks, chaque élément Aproduit est un couple (AN, AQ) où AN est le numéro du produit et AQ la quantité en stock, classée par ordre croissant des numéros de produit.

- Mvt est la suite des mouvements ou modifications à effectuer sur la suite Astock. Chaque élément est un triplet (MN, code, MQ) avec MN le numéro du produit à modifier, code le type de modification et MQ une quantité. Cette suite est classée par ordre croissant des numéros de produits et peut comporter plusieurs éléments de même numéro.

V - 2 -1-2- Définition du résultat

Le résultat, c'est le nouveau fichier des stocks Nstock. Il possède la même structure que Astock, mais il ne possède pas le même nombre d'éléments. Pour le définir, on doit prendre en compte la suite Mvt.

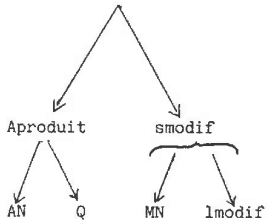
Il est défini simplement à partir de la suite dont chaque élément est le couple (Aproduit, smodif) où Aproduit désigne un produit de Astock et smodif la suite des modificiations à effectuer sur un même produit. Cette suite, nommée sastmvt, sera déterminée à partir des deux suites données et de l'opérateur ⊠ de mise à jour de deux suites.

Nstock = iter *Maj sur sastmvt

*Maj définit un élément de Nstock soit Nproduit.

2 - Le module *Maj

Pour définir Nproduit, nous disposons d'un élément de sastmvt, dont la structure est représentée par :



smodif est un couple (MN, lmodif) où lmodif est la suite des modifications à effectuer sur le produit MN.

D'après la définition de l'opérateur ⊠, AN et MN ne sont pas forcément les mêmes. Donc, pour déterminer Nproduit, nous allons être amenés à comparer ces deux éléments:

Nproduit = si AN < MN alors *Recopie
si AN = MN alors *Modif
si AN > MN alors *Nouveau

3 - Le module *Recopie

Correspond au cas où le produit dans Astock n'a pas été modifié :

Nproduit = Aproduit

4 - Le module *Modif

Le produit de numéro AN a été modifié : trois types de modifications sont possibles, et elles ne sont pas exclusives.

Nproduit = si supprimé alors < > sinon (AN, NQ)

où :

- supprimé est un booléen indiquant si le produit a été supprimé du fichier des stocks
- NQ désigne la nouvelle quantité en stock, la suite des modifications sur ce produit ayant été prise en compte.

. Comment définir NQ et supprimé ?

Nous disposons de smodif, donc de la suite des modifications à effectuer sur le produit traité.

```
| NQ,  supprimé= der iter *unemodif sur lmodif de smodif
| AN = tête (Aproduit)
```

Remarque : Nous avons directement fusionné les définitions de NQ et supprimé.

. Comment définir *unemodif ?

Nous sommes dans le cas où il s'agit de modifications sur un produit existant. Si nous ne tenons pas compte des erreurs possibles, les seules modifications possibles sont des suppressions ou des mises à jour simples. NQ est initialisé à la quantité contenue dans l'ancien stock soit AQ.

```
| supprimé, NQ : si code = M alors | supprimé = ω supprimé
|                                  | NQ = ω NQ + MQ
|                          sinon   | supprimé = vrai
|                                  | NQ = 0
| prem   NQ = queue (Aproduit)
| prem supprimé = faux
```

Remarque : Dans le cas où le produit est supprimé, la poursuite du parcours de la liste des modifications n'est pas très astucieuse.

5 - Le module *Nouveau

Il s'agit d'insérer un nouveau produit n'existant pas dans l'ancien fichier des stocks.

Remarque : Ce nouveau produit a pu être créé, puis supprimé entre la dernière mise à jour effectuée et la suivante. Bien que cette possibilité semble peu raisonnable, nous devons en tenir compte dans l'algorithme.

```
| Nproduit = si supprimé alors <  > sinon (MN, NQ)
| MN = tête (smodif)
```

Comme dans le module *Modif,

- supprimé est un booléen indiquant, toutes modifications effectuées, si le produit est conservé
- NQ la nouvelle quantité.

Remarque : Nous aurions pu supposer qu'il n'était pas possible de supprimer un nouveau produit.

. Définition de NQ et supprimé

Nous disposons de smodif, donc de la suite des modifications à effectuer sur le produit MN :

```
| NQ, supprimé = der iter *creprod sur lmodif (de smodif)
```


. Le module *créprod :

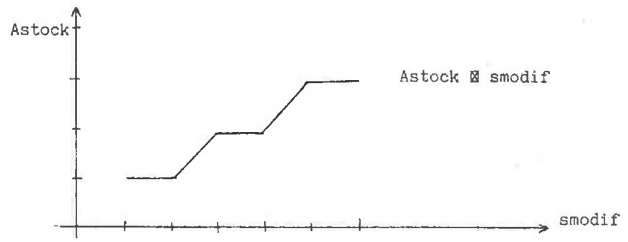
Au niveau de ce module, les modifications possibles sont des mises à jour ou une suppression du produit considéré :

```
| NQ, supprimé = si code = M alors | supprimé = ω supprimé  sinon | supprimé=vrai
|                                  | NQ = ω NQ + MQ               | NQ = 0
| prem NQ = MQ
| prem supprimé = faux
```

V - 2 -1-3- Définition de la suite intermédiaire sastmvt

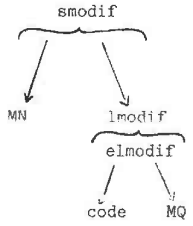
1 - | sastmvt = Astock ⊠ smodif

où smodif est la suite dont chaque élément contient pour un numéro de produit MN toutes les modifications à effectuer sur ce produit, sastmvt est définie de manière canonique, quelles que soient les deux suites Astock et Mvt (cf. la définition de ⊠ au paragraphe III - 2 de ce chapitre).



2 - smodif ne correspond pas exactement à la suite Mvt donnée.

Sa structure est représentée par :



Cette suite peut se déduire simplement de la suite Mvt, en utilisant une itération avec condition d'arret.

```
| smodif = jqa arret iter *csmod avec Mvt
```

Il s'agit d'une suite à "rupture" dont le critère correspond à un changement de produit.

*csmod associe à un numéro de produit, la suite de ses modifications.

Comment le définir ?

esmodif = MN, lmodif
lmodif = jqa finmodif iter *parmot sur Mvt
prem (MN, code, MQ) = donnée
arret = fin (Mvt)

Le module *parmvt

elmodif = code, MQ
MN, code, MQ = donnée
finmodif = MN ≠ ω MN

V - 2 -1-4- Première version de l'algorithme

- Nstock suite des stocks mise à jour *Maj définissant un élément de Nstock - sasmvt suite de couples (ancien produit, modification sur un produit) - Astock suite des anciens stocks - smodif couple numéro de produit, suite des modifications sur ce produit - arret suite de booléens *csmvt définit smodif - Mvt suite des modifications	Nstock = iter *Maj sur sastmvt sastmvt = Astock @ smodif smodif = jqa arret iter *csmvt avec Mvt Astock, Mvt = donnée
*Maj	
- Nproduit élément de Nstock - AN entier numéro dans Astock - MN entier numéro produit dans Mvt *Recopie le produit non modifié *Modif modifie le produit existant *Nouveau créé un nouveau produit	Nproduit si AN < MN alors *Recopie si AN = MN alors *Modif si AN > MN alors *Nouveau
*Recopie	
Aproduit (AN, NQ) de Astock	Nproduit = Aproduit

..../....

.../....

*Modif	
- supprimé booléen précisant si le produit est conservé après modification - NQ réel quantité après modifications *unemodif prend en compte une modification - lmodif suite des modifications à effectuer sur le produit traité	Nproduit = si supprimé alors < > sinon (AN,NQ) AN = tête (Aproduit) NQ, supprime = der iter *unemodif sur lmodif (de smodif)
*Unemodif	
- MQ réel quantité à ajouter ou supprimer provenant de Mvt NQ réel quantité dans l'ancien stock	supprimé, NQ = si code = M alors | supprimé = ω supprimé NQ = ω NQ + MQ sinon | supprimé = vrai NQ = 0 prem NQ = queue (Aproduit) prem supprimé = faux
*Nouveau	
- MN entier numéro du produit créé *creprod effectue la modification	Nproduit = si supprimé alors < > sinon (MN, NQ) MN = tête (smodif) NQ, supprimé = der iter *créprod sur lmodif (de smodif)
*creprod	
	NQ, supprimé = si code = M alors | supprimé = ω supprimé NQ = ω NQ +MQ sinon | supprimé = vrai NQ = 0 prem NQ = MQ prem supprimé = faux
*csmvt	
- esmodif élément de smodif - lmodif suite des modifications à effectuer sur un produit - finmodif booléen indiquant la fin des modifications sur le produit traité	esmodif = MN, lmodif lmodif = jqa finmodif iter *parmvt sur Mvt prem Mproduit = si fin (Mvt) alors + ∞ sinon donnée arret = (Mproduit = + ∞)
*parmvt	
- code caractère type de la modification - Mproduit triplet (MN, code, MQ) du fichier Mvt	elmodif = code, MQ Mproduit = si fin (Mvt) alors + ∞ sinon donnée finmodif = MN ≠ ω MN

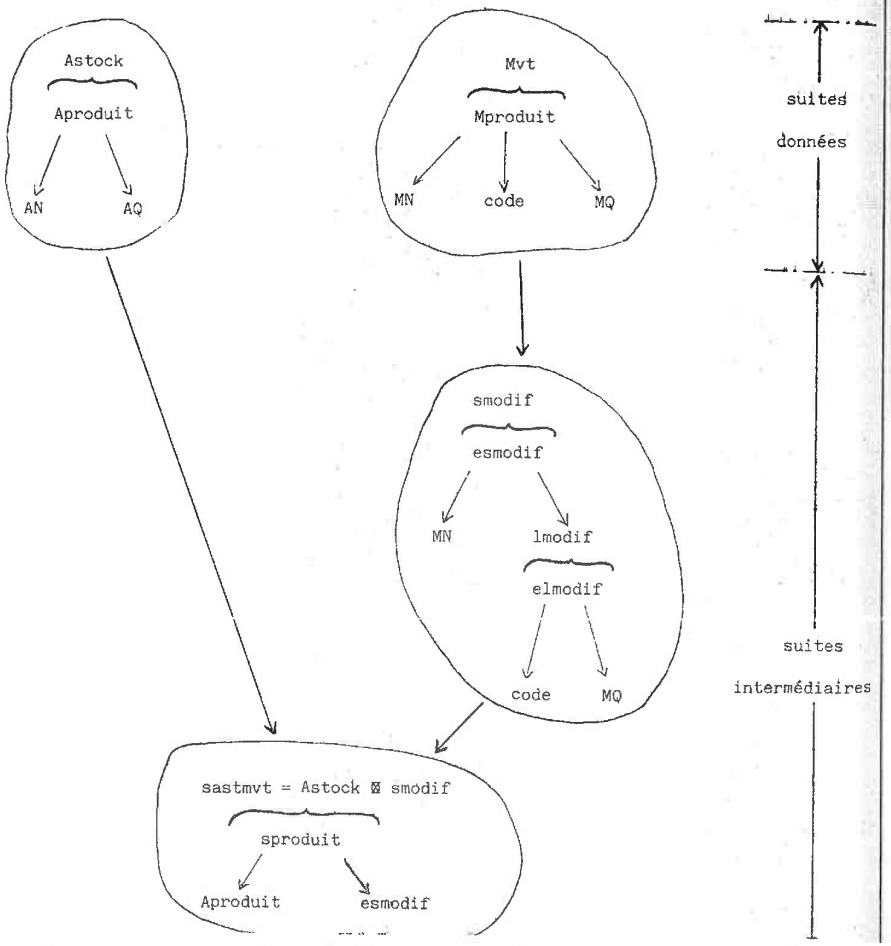
Remarque: L'élément +∞ permet de tenir compte des fins de fichiers.

V - 2 -1-5- Commentaires

- Pour mettre à jour la suite Astock avec la suite Mvt (des modifications), nous avons utilisé l'opération ⊠ sur les suites, ces suites n'étant pas toutes les deux des suites données.

- La suite Mvt peut contenir plusieurs éléments concernant le même produit : elle peut être considérée comme suite à "ruptures" dont le critère correspond à un changement de numéro de produit. On introduit alors la suite smodif à deux niveaux permettant de regrouper toutes les modifications portant sur un même produit.

L'ensemble des structures introduites est schématisé par :



V - 2 -2- Obtention de l'algorithme final

Appliquons la stratégie de transformation au :

1 - Module principal

Le dépliage de Nstock conduit à :

Nstock = iter *Maj sur Astock ⊠ smodif

Nstock est défini par itération sur une suite définie à l'aide de l'opérateur ⊠. Cette suite n'est pas définie par l'utilisateur : sa définition est générale. Elle s'écrit :

Astock ⊠ grmvt = jqa arret2 iter *couple avec (Astock, smodif)

*couple

sproduit = si AN < MN alors | (ω Aproduit, ω esmodif)
Aproduit = donnée
esmodif = ω esmodif

si AN = MN alors | (ω Aproduit, ω esmodif)
Aproduit = donnée
esmodif = suc_{smodif} (ω esmodif)

si AN > MN alors | (ω produit, ω esmodif)
esmodif = suc_{smodif} (ω esmodif)
Aproduit = ω Aproduit

arret2 = fin (Astock) et fin (smodif)

prem Aproduit = donnée

prem esmodif = tête (smodif)

Commentaires :

Dans cet algorithme, nous n'avons pas tenu compte du cas des fins de liste. Lors de la définition formelle du type suite, nous avions rajouté un axiome permettant de généraliser les opérations sur les suites en introduisant un élément supplémentaire plus grand que tout élément de la suite traitée noté + ∞ .

Pour être complétement rigoureux, nous aurions dû définir *couple de la façon suivante :

*couple

```
sproduit = si AN < MN   alors |( ∂ Aproduit, ∂ esmodif)
                              |Aproduit = si fin (Astock) alors + ∞
                              |                           sinon donnée
                              |esmodif = ∂ esmodif

         si AN = MN   alors |( ∂ Aproduit, ∂ esmodif)
                            |Aproduit = si fin (Astodk) alors + ∞
                            |                           sinon donnée
                            |esmodif = suc_smodif (∂esmodif)

         si AN > MN   alors |( ∂ Aproduit, ∂ esmodif)
                            |Aproduit = ∂ Aproduit
                            |esmodif = suc_smodif (∂esmodif)

arret2 = (Aproduit = + ∞ ) et fin (smodif)
prem (Aproduit) = si fin (Astock) alors + ∞ sinon donnée
```

Remarques :

- La suite Astock est un fichier physique existant. La fonction de succession dans cette suite a été exprimée à l'aide de donnée.

- La suite smodif est une suite logique introduite par commodité pour définir le résultat. Pour l'instant, elle n'a pas d'existence concrète : nous avons appelé suc la fonction de succession dans cette suite et tête la fonction d'accès au premier élément. Ces deux fonctions sont définies par l'utilisateur.

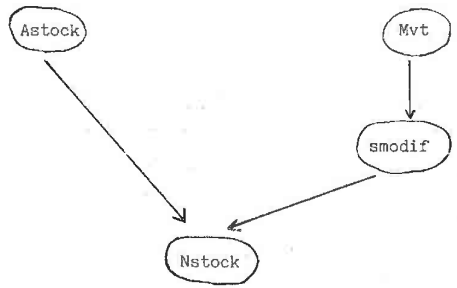
Notre but est d'éviter la construction de la suite intermédiaire sastmvt.
Pour cela, explicitons l'itération définissant Nstock en une itération générale, où le parcours de la suite sera géré dans le module *Maj (cf. chapitre 2, §. IV-2-4 : passage d'une itération sur une suite complète à une itération générale).


```
Nstock = jqa arret2 iter *Maj' avec Astock, smodif

*Maj'
Nproduit = si AN < MN alors *Recopie
           si AN = MN alors *Modif
           si AN > MN alors *Nouveau

arret2 = (Aproduit = + ∞ ) et fin (smodif)

sproduit = si AN   MN alors |( ∂ Aproduit, ∂ esmodif)
                            |Aproduit = si fin (Astock) alors + ∞  sinon donnée
                            |esmodif = ∂ esmodif

         si AN = MN alors |(∂ produit, ∂ esmodif)
                          |Aproduit = si fin (Astock) alors + ∞  sinon donnée
                          |esmodif = suc_smodif (∂esmodif)

         si AN   MN alors |(∂ produit, ∂ esmodif)
                          |esmodif = suc_smodif (∂esmodif)
                          |Aproduit = ∂ Aproduit
```

Remarques :

- Avec la nouvelle définition itérative introduite, il n'y a plus de suite guide : à chaque fois que l'on définit un élément de la suite résultat on ne parcourt pas forcément un élément de la suite Astock et de la suite smodif.

- Avec cette transformation, nous devons remplacer dans *Maj chaque élément sproduit par ∂ sproduit (donc Aproduit par ∂ Aproduit et smodif par ∂ smodif)
(la nouvelle définition avec condition d'arret nécessite une lecture à l'avance).



suites restantes après cette première transformation

Quelles transformations sont-elles encore possibles dans le module principal ?

```
|Nstock = jqa arret2 iter *Maj' avec Astock, smodif
|smodif = jqa arret  iter *csmvt avec Mvt
```

Utilisons la règle de composition d'itérations dans le cas d'une itération contenant avec :

```
|Nstock = jqa fin iter fusm (*Maj', *csmvt) avec Astock, Mvt.
```

Quel est le rôle d'une telle transformation ?

- disparition de la suite intermédiaire smodif
- remplacement de la fonction de succession dans cette suite par sa définition.

fusm (*Maj', *csmvt) est le module contenant les définitions de *Maj' et de *csmvt.

2 - Le module fusm (*Maj, *csmvt)

Nproduit et (Aproduit, esmodif) sont définis de manière conditionnelle. De plus, les conditions sont identiques : nous pouvons fusionner ces deux définitions :

*fusm (Maj', csmvt)

```
Nproduit, sproduit = si AN < MN alors *Recopie'
                     si AN = MN alors *Modif'
                     si AN > MN alors *Nouveau'

fin = (Aproduit = + ∝ ) et fin ( Mvt)
esmodif = MN, lmodif
lmodif = jqa finmodif iter *parmvt sur Mvt
prem (MN, code, MQ) = donnée
prem Aproduit = si fin (Aproduit) alors + ∝ sinon donnée
```

Les modules *Recopie', *Modif', *Nouveau' correspondent respectivement aux modules *Recopie, *Modif, *Nouveau auxquels on a adjoint les définitions des objets sproduit, Aproduit et esmodif (définitions issues du module *couple).

Ces nouveaux modules peuvent être définis formellement à l'aide de l'opérateur ajoutrec (sur le type MODULE).

L'élément esmodif a-t-il besoin d'être construit complétement pour être utilisé ? Non, de plus cet élément n'est utilisé qu'au niveau des modules *Modif et *Nouveau. Lorsqu'il s'agit d'une recopie simple, sa définition n'intervient pas : la transformation suivante va consister à déplier esmodif en tenant compte de cette remarque.

*fusm (Maj', csmvt)

```
Nproduit, (Aproduit, MN, lmodif) = si AN < MN alors *Recopie'
                                   si AN = MN alors *NModif
                                   si AN > MN alors *NNouveau
fin = (Aproduit = + ∝ ) et fin (Mvt)
prem (MN, code, MQ) = donnée
prem Aproduit = si fin (Astock) alors + ∝ sinon donnée
```

*NModif est le module *Modif' auquel on a rajouté la définition de lmodif
*NNouveau est le module *Nouveau' auquel on a rajouté lmodif.

Remarque : La condition d'arrêt portant sur la suite smodif a disparu avec cette suite.

3 - Le module *Recopie'

Aucune modification n'est possible.

4 - Le module NModif :

Il contient les définitions suivantes :

```
Nproduit = si supprimé alors < > sinon (AN,NQ)
AN = tête ( ω Aproduit)
NQ, supprimé = der iter *unemodif sur lmodif
lmodif = jqa finmodif iter *parmvt sur Mvt
Aproduit = si fin (Astock) alors + ∝ sinon donnée
esmodif = ω esmodif
smodif = ( ω Aproduit, ω esmodif)
NQ, supprimé = der jqa finmodif iter fusm (*unemodif, *parmvt) sur Mvt
```

Remarque : La définition de sproduit n'est plus utile donc esmodif disparaît aussi.

5 - Le module fusm (*unemodif, *parmvt)

```
supprimé, NQ = si code = M alors |supprimé = ω supprimé
                                 |NQ = ω NQ + MQ
                      sinon      |supprimé = vrai
                                 |NQ = 0
finmodif = MN ≠ ω MN
MN, code, MQ = donnée
prem NQ = queue (Aproduit)
prem supprimé = faux
```

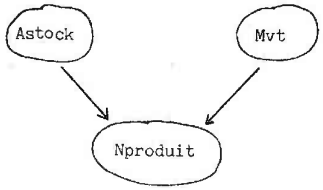
Remarque : l'élément elmodif n'a plus de raison d'être.

6 - Le module NNouveau

On procède de la même façon que pour le module NModif.

V - 2 -3- Algorithme final

Nous arrivons, après transformations successives, à une définition de la suite résultat Nstock à partir des deux suites données soient :



suites restantes toutes transformations effectuées

- Nstock suite des stocks mise à jour - fin suite de booléns *Maj définit un élément de Nstock - Astock suite des produits en stocks - Mvt suite des modifications à effectuer	Nstock = jqa fin iter *Maj avec Astock, Mvt Astock, Mvt = suites données
*Maj	
- Nproduit (NN, NQ) élément de Nstock - AN entier numéro produit dans Astock - MN entier numéro produit dans Mvt - Aproduit élément de Astock (AN, AQ) - Mproduit élément de Mvt (MN, MQ) *Recopie le produit non modifié *Nmodif modifie le produit existant *NNouveau créé un nouveau produit	Nproduit = si AN < MN alors *Recopie si AN = MN alors *NModif si AN > MN alors *NNouveau fin = (Aproduit = + ∞) et (Mproduit=+ ∞) prem Mproduit = si fin(mvt) alors + ∞ sinon donnée prem Aproduit = si fin (Aproduit) alors + ∞ sinon donnée
*Recopie	
	Nproduit = ω̂Aproduit Aproduit = si fin (Astock) alors + ∞ sinon donnée
*NModif	
- supprimé booléen précisant si le produit est conservé après modification - NQ réel quantité en stock mise à jour - MN entier numéro de produit dans Mvt fusm (*unemodif, *parmvt) effectue une modification	Nproduit = si supprimé alors < > sinon (AN, NQ) AN = tête (ω̂Aproduit) NQ, supprimé = der jqa finmodif iter fusm(*unemodif,*parmvt) sur Mvt Aproduit = si fin (Astock) alors + ∞ sinon donnée
*fusm (*unemodif, *parmvt)	
- code élément de Mproduit - MQ entier quantité de Mvt	supprimé, QT = si ω̂ code= M alors | supprimé = supprimé NQ = ω̂NQ + MQ sinon | supprimé = vrai NQ = 0 finmodif = MN ≠ ω̂MN Mproduit = si fin(Mvt) alors + ∞ sinon donnée prem supprimé = faux prem NQ = queue (ω̂produit)
*NNouveau	
	Nproduit = si supprimé alors < > sinon (MN, NQ) MN = tête (Mproduit) NQ,supprimé= der jqa MN ≠ MN iter fusm(*creprod,*parmvt) sur Mv
	..../....

..../....	
*fusm (*créprod,*parmvt)	
	supprimé, NQ = si ∂code = M alors | supprimé= ∂supprimé NQ = ∂NQ + MQ sinon | supprimé = vrai NQ = 0 prem NQ = ∂MQ prem supprimé = faux Mproduit = si fin (mvt) alors + ∞ sinon donnée

algorithme final de la mise à jour des stocks

V - 2 -4- Modification de l'énoncé

Supposons que l'on ajoute à l'énoncé de ce problème la contrainte suivante : les modifications ne seront effectuées que lorsqu'on est sûr que le produit n'a pas été supprimé.

Au niveau des transformations :

- dans l'algorithme initial, nous avions fusionné la définition de supprimé et de NQ : cela signifiait qu'en même temps que nous déterminions supprimé, nous effectuions la modification correspondante.

Pour répondre à cette deuxième spécification, nous devrons définir séparément NQ et supprimé :

Nproduit = si supprimé alors < > sinon *supr supprimé = der jqa MN ≠ ∂MN iter *parcourlmodif sur lmodif
*Supr
Nproduit = AN, NQ NQ = der iter *Unmodif sur lmodif
*parcourlmodif
supprimé = si code = M alors ∂supprimé sinon vrai prem supprimé = faux

La suite lmodif devra être parcourue une première fois pour déterminer supprimé puis, en fonction du résultat obtenu, elle sera à nouveau utilisée pour définir les modifications.Au niveau des transformations, cette suite ne vérifiera pas les préconditions nécessaires à l'application de la règle de composition d'itérations.

Finallement, le résultat ne sera plus défini à partir des deux suites de départ Astock et Mvt, mais nécessitera l'introduction de la suite intermédiaire lmodif (suite qui à un numéro de produit associe la suite des modifications à effectuer sur ce produit).

VI - Conclusion

L'application stricte et rigoureuse de la méthode de construction d'algorithmesen plusieurs étapes proposée ici, s'avère lourde et fastidieuse, quel que soit le problème traité.

Le nombre de règles limité et l'application récursive de la stratégie proposée rendent possible l'automatisation de cette partie du travail.

CHAPITRE 3 -

CHAPITRE 3

PARALLELE AVEC DES METHODES EXISTANTES

Un certain nombre de concepts utilisés dans ce travail ont été développés depuis les années 1970. La notion de structuration des données et des résultats, l'introduction d'intermédiaires, la décomposition des structures sont des notions fondamentales dans les travaux de WARNIER et de JACKSON.

I - LA METHODE LCP (langage de construction de programmes)

C'est une méthode descendante qui privilegie les entrées [WAR, 76], essentiellement utilisée en informatique de gestion.

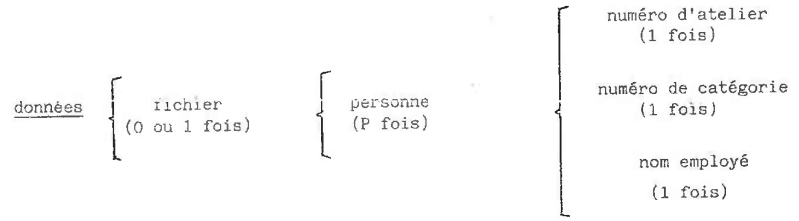
"Toute information est une donnée car elle ne présente d'intérêt que si elle fait l'objet d'un traitement. A ce titre, tout résultat est une donnée pour le traitement suivant", cf. [WAR, 71]. Ainsi, l'ensemble des traitements est considéré comme un fichier d'informations au même titre que ceux des données et des résultats. Les divers ensembles d'informations sont décomposés en sous-ensembles élémentaires à l'aide des deux structures de base : la structure alternative et la structure répétitive.

I - 1 - Aperçu de la méthode sur un exemple

Reprenons l'exemple des ateliers (énoncé p. 14, exemple a).

I - 1 -1- Définition de la structure hiérarchique des données à l'entrée :

Le fichier du personnel est un ensemble de triplets (nuatelier, nucatégorie, nom). Ceci est représenté par :



On note entre parenthèses le nombre de fois que l'élément concerné devra être traité. Dans l'exemple, fichier (0 ou 1 fois) signifie que le fichier peut être vide (0) auquel cas, on n'effectuera pas de traitement ou que le fichier contient des enregistrements consécutifs, chaque enregistrement concernant une personne étant lui-même constitué de trois champs : un numéro d'atelier, un numéro de catégorie et un nom.

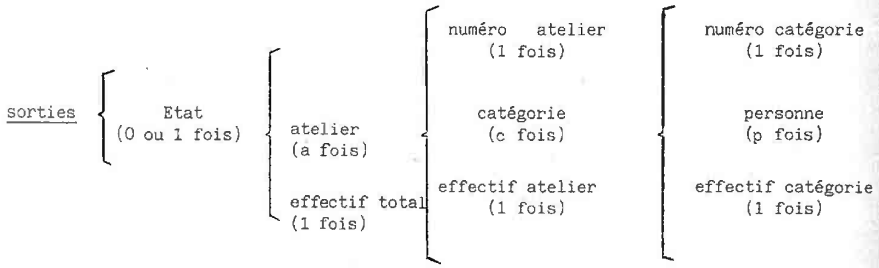
Remarque : Cette notation est insuffisante : elle n'apporte aucune précision sur la façon dont sont rangées les personnes (le fait que les personnes sont classées par numéro croissant d'atelier n'apparaît pas).

La description d'un fichier dont les personnes ne sont pas classées par ordre croissant serait la même et pourtant la résolution du problème en serait modifiée.

Comment traiter ce problème en utilisant la méthode L.C.P. ? Les travaux cités ne mentionne pas ce type de problème.

I - 1 -2- Définition de la structure hiérarchique des données à la sortie :

La résolution du problème posé nécessite une décomposition par atelier puis par catégorie.



I - 1 -3- Organisation du programme

A partir de la structure hiérarchique des données à l'entrée, notre but est de décrire un "squelette de programme". Nous disposons de deux modes de composition d'algorithme : la structure répétitive et la structure alternative.

- Définition d'une structure répétitive de programme

Un sous-ensemble du programme de structure répétitive comporte obligatoirement une séquence "début" et une séquence "fin" exécutée une fois dans l'ensemble et un sous-ensemble répétitif exécuté N fois dans l'ensemble. La dernière instruction du sous-ensemble répétitif est un branchement conditionnel.

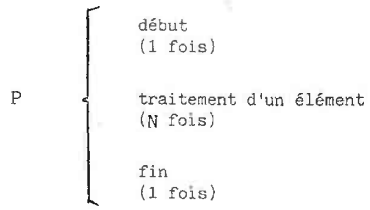


figure 1 : Représentation d'une structure répétitive

- Définition d'une structure alternative de programme

Un ensemble de programme de structure alternative est obtenu en appliquant la règle : à structure alternative de données, structure alternative du programme. C'est un ensemble comportant au moins quatre séquences logiques. Il comporte toujours un branchement conditionnel qui termine la séquence début et autant de branchements systématiques qu'il y a de branches à l'alternative moins une.

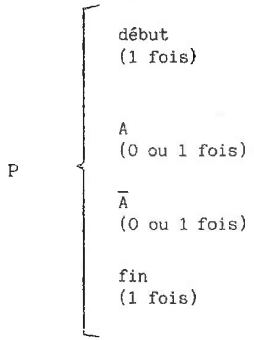


figure 2 : représentation d'une structure alternative

I - 1 -3-1- Organisation hiérarchique ou décomposition en séquences

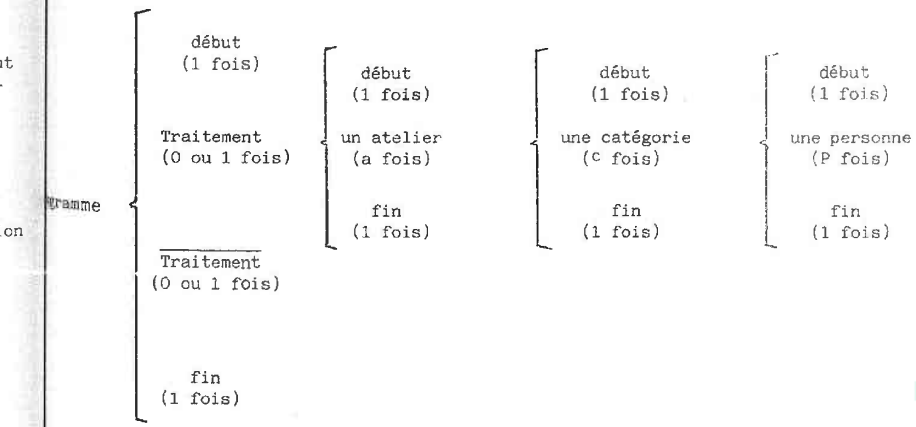


figure 3 : organisation hiérarchique (les ateliers)

On a fait apparaître les séquences logiques avec leurs nombres d'exécutions.

A partir du squelette de programme présenté figure 3, établissons un organigramme des séquences logiques (cf. figure 4). En fait, l'organigramme n'est qu'une représentation isomorphe de la figure 3. La numérotation introduite au niveau de l'organigramme aurait pu être indiquée directement sur la figure 3. Elle sert pour l'écriture du programme détaillé : introduction des diverses instructions.

I - 1 -3-1- Organisation détaillée du programme

Cette phase se décompose en deux parties :

- WARNIER propose d'abord d'établir les listes d'instructions par catégories (les lectures, les sorties, les calculs), puis ensuite d'établir la liste donnée des instructions.

Remarque : Cette façon de procéder est assez peu méthodique. Faire des énumérations par catégorie semble source d'erreur et fastidieux à réaliser.

listes d'instructions par catégories :

sorties : 110 éditer eftotal
100 éditer efatelier
090 éditer efcatégorie
040 éditer NUA
050 éditer NUC
070 éditer NOM
030, 080 lecture

Branchements et préparation des branchements :

010	Si non fin de fichier	030
020		120
080	Si n° de catégorie lu = n° de catégorie précédent	060
090	Si n° atelier lu = n° atelier précédent	050
100	Si n° atelier ≠ 0	040

Calculs

030	eftotal = 0
040	efatelier = 0
050	efcatégorie = 0
100	eftotal = ∂ eftotal + efatelier
090	efatelier = ∂ efatelier + efcatégorie
070	efcatégorie = ∂ efcatégorie + 1

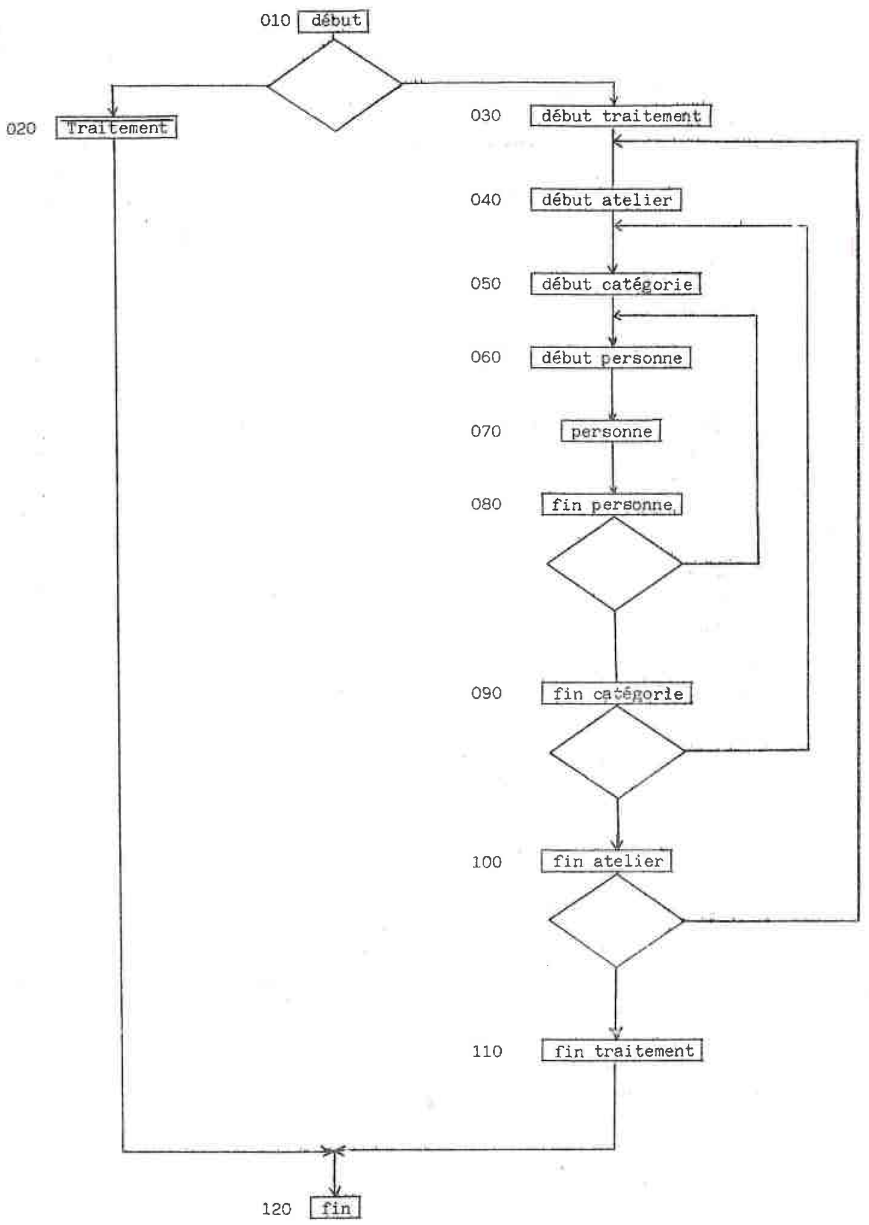


figure 4: organigramme des séquences logiques

Liste des instructions :

010	lire NUA, NUC, NOM si NUA ≠ 0 et NUC ≠ 0	030
020		120
030	eftotal = 0	
040	efatelier = 0 éditer NUA	
050	efcatégorie = 0 éditer NUC	
060		
070	éditer NOM efcatégorie = ω efcatégorie + 1	
080	lire NUA, NUC, NOM si NUC ≠ ω NUC	060
090	éditer efcatégorie si NUA ≠ ω NUA	050
100	éditer efcatégorie eftotal = ω eftotal + efatelier si NUA ≠ 0	040
110	éditer eftotal	
120	fin	

La phase d'organisation détaillée étant terminée, WARNIER effectue un contrôle du programme par les sorties et en dernier lieu le codage dans le langage de programmation choisi.

Remarque : Supposons que les données ne soient pas organisées comme prévu. Que fait-on dans ce cas ? WARNIER ne propose pas de réponse.

I - 2 - Récapitulatif de la méthode L.C.P.

La mise en oeuvre de la méthode s'effectue en trois phases.

-2 - 1 - définition de la structure hiérarchique des données à l'entrée

Lorsqu'il y a plusieurs fichiers physiques en entrée, on s'attache à la notion de fichier logique. Pour un fichier de traitement, il existe un seul et unique fichier logique à traiter (ceci quel que soit le nombre de fichiers physiques de données à l'entrée).

fichiers physiques de données à l'entrée ⟶ fichier logique de données à l'entrée

-2 - 2 - définition de la structure hiérarchique des données à la sortie

de la même manière :

fichier logique de données à la sortie ⟶ fichiers physiques de données à la sortie

I - 2 -3- Organisation du programme se déroule en 3 étapes :

I - 2 -3-1- Organisation en séquences logiques

C'est la définition de la structure hiérarchique du programme à partir de la structure hiérarchique des données à l'entrée. Le programme obtenu est constitué d'un ensemble ordonné de séquences logiques (une séquence logique est un ensemble ordonné d'instructions exécutées un même nombre de fois sous les mêmes conditions).

I - 2 -3-2- Organisation détaillée du programme

On numérote les séquences et on établit les listes d'instructions par catégorie. L'ordre d'établissement des listes d'instructions est la suivante :

- lectures
- branchements
- préparation des branchements (tenir compte des données disponibles)
- calculs, initialisations
- sorties.

On établit ensuite la liste ordonnée des instructions et son codage dans le langage approprié. Cette liste est rédigée séquence par séquence dans l'ordre croissant des numéros de séquence.

I - 2 -3-3- Contrôle du programme par les sorties

Cette étape consiste à vérifier que chaque sortie est programmée dans la séquence appropriée.

I - 3 - Application de la méthode L.C.P. à l'exemple des télégrammes

Remarque préliminaire : Nous ne prétendons pas que la mise en oeuvre de la méthode L.C.P. pour la résolution de ce problème soit la seule possible, mais c'est ce qui nous a semblé le plus naturellement déductible des textes de WARNIER.

I - 3 -1- Définition de la structure hiérarchique des données à l'entrée

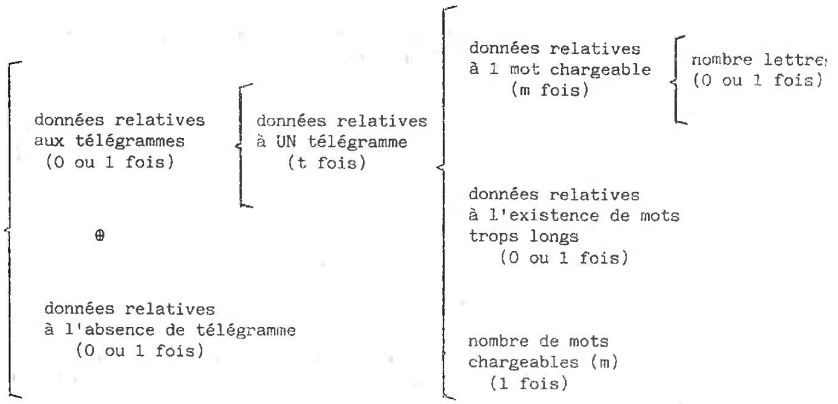
Le fichier physique des données à l'entrée est constitué d'une suite de caractères.

fichier physique des données à l'entrée { caractères (n fois)

Cette structure d'entrée doit aussi faire intervenir la notion de bloc : en effet, la succession des caractères obéit à une certaine logique.

I - 3-2- Définition de la structure hiérarchique des données à la sortie

On retient l'hypothèse de la non existence de télégrammes.



⊕ symbolise la réunion des deux sous-ensembles dans l'ensemble sorties.

Lors de l'élaboration de la structure du programme, il sera tenu compte des précisions supplémentaires n'apparaissant pas dans la structure hiérarchique des sorties :

- les mots de plus de 12 caractères seront tronqués à 12
- on éditera le nombre de mots comptables et le nombre de mots trop longs
- ZZZZ et STOP ne sont pas des mots chargeables
- Dans chaque télégramme, les mots STOP seront remplacés par des points (.) et les mots ZZZZ seront supprimés.

Pour pouvoir organiser le programme à partir de ces structures, on doit faire apparaître de nouvelles structures : des structures dites logiques permettant de mieux prendre en compte la forme du résultat.

I - 3 -3- Introduction de fichiers logiques à l'entrée

Si on se contente d'examiner le fichier donnée à l'entrée, il est constitué comme cela est présenté au paragraphe I - 3 -1- de caractères et on ne peut rien dire de plus.

La structure de fichier physique ne fait pas apparaître la notion de télégrammes utile à l'élaboration de la structure du programme (fait partie de la définition du résultat).

Introduisons la notion de fichier logique (qui n'a pas de support physique): aux enregistrements physiques, on va associer la notion logique de télégrammes.

En fait, pour introduire ce fichier logique, on est guidé par les résultats et pas uniquement par les données.

fichier logique des données à l'entrée { UN télégramme (t fois)

Cette structure est incomplète et ne nous permet pas encore d'élaborer la structure du programme. A la structure de télégramme, associons la notion de mot :

un télégramme { UN mot (m fois)

Cette description est incomplète : un mot peut être considéré comme un ensemble de caractères précédés de 0 à C caractères blancs.

un mot {
- caractère à éliminer (c1) { blanc (c1-1) (0, 1) { fin bloc (0 ou 1 fois) ; ⊕ ; fin bloc (0 ou 1 fois) } ; ⊕ ; blanc (1) (0, 1) % correspond au début d'un mot %
- caractère significatif (c2) { blanc (1) (0, 1) % correspond à la fin d'un mot % ; ⊕ ; blanc (c2-1) (0, 1) { fin bloc (0 ou 1 fois) ; ⊕ ; fin bloc (0 ou 1 fois) }
}

Progressivement, nous enrichissons la structure du fichier logique introduit en prenant en compte de plus en plus de détails sur la structure du résultat.

Le découpage de la chaîne de caractères en mots permet de supprimer les caractères blancs inutiles séparant deux mots. On doit maintenant se préoccuper de savoir si le mot courant est comptable ou non. Une façon de faire consiste à comparer les mots de 4 lettres "ZZZZ" et "STOP" aux mots spéciaux.

un mot {
- mot de 4 caractères (0,1) { NC ⩽ 12 (0,1) ; ⊕ ; NC > 12 (0,1) }
- ⊕
- mot de 4 caractères (0,1) { mot ≠ STOP (0,1) (m2) { mot ≠ ZZZZ (0,1) ; ⊕ ; mot = ZZZZ (fin du (0,1) télégramme) } ; ⊕ ; mot = STOP (0,1) (m1) }

figure 6 : structure hiérarchique de mot

Pour compléter la description du fichier logique d'entrée, nous devons étudier la valeur du premier mot nécessaire à la détermination de l'existence de télégrammes. Une description est donnée figure 7.

L'organisation détaillée de la structure du fichier logique à l'entrée est terminée : toutes les entités nécessaires à l'obtention du résultat ont été introduites. Nous pouvons en déduire la structure du programme (cf. figure 8).

Remarque : La façon d'introduire ce fichier logique et de le définir n'est pas très méthodique : l'utilisateur de la méthode L.C.P. doit souvent faire appel à l'intuition. Il ne semble pas y avoir de guide, ni d'indications permettant de lui apporter une aide efficace lorsqu'il se trouve dans une impasse.

I - 3 -4- Organisation du programme

Nous nous arrêterons là sur cet exemple, les étapes suivantes n'apportant aucun élément nouveau.

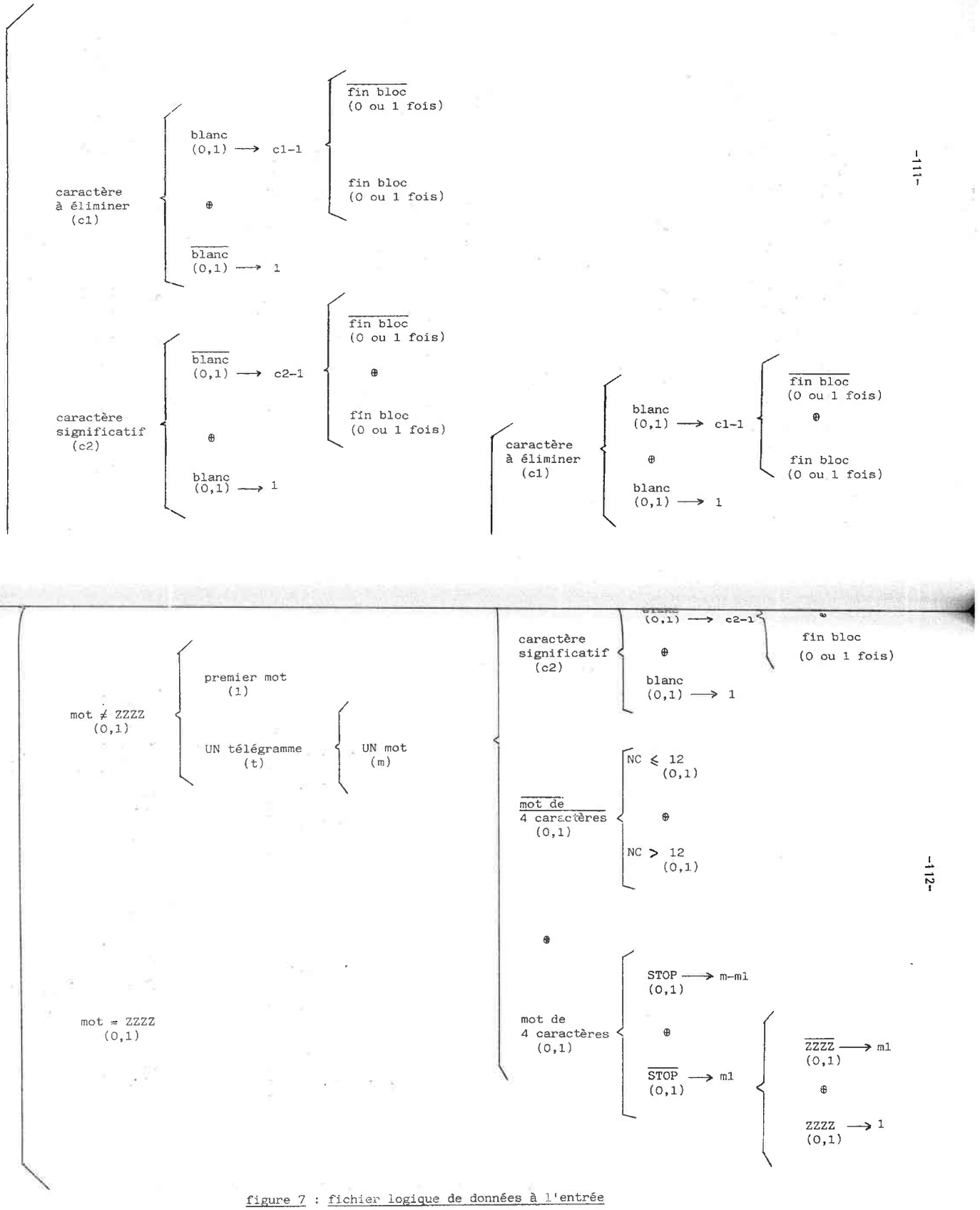
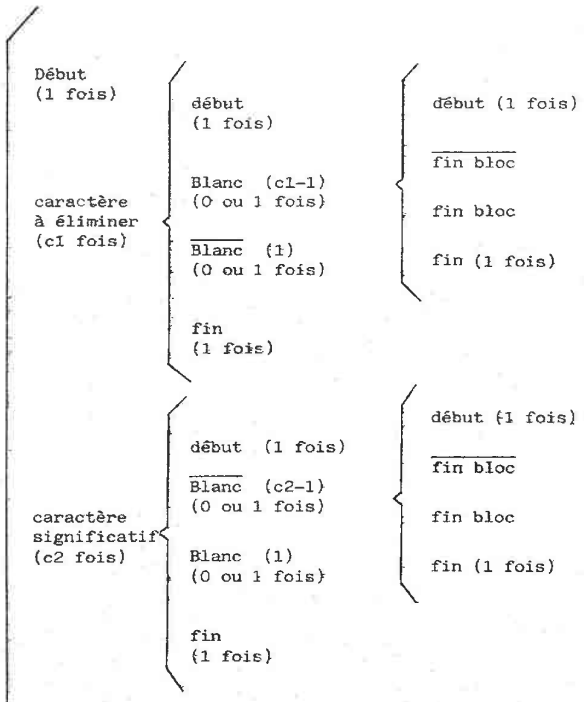
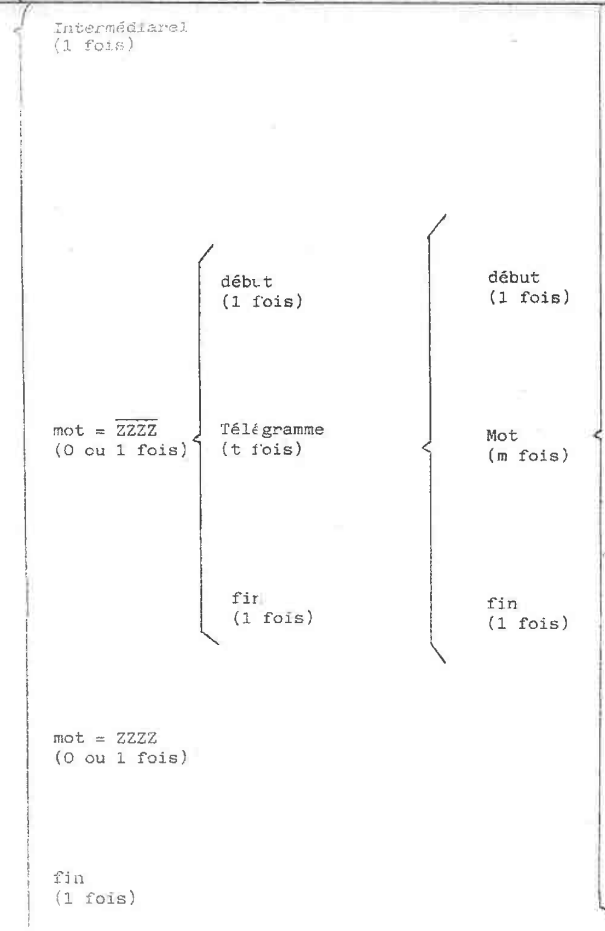
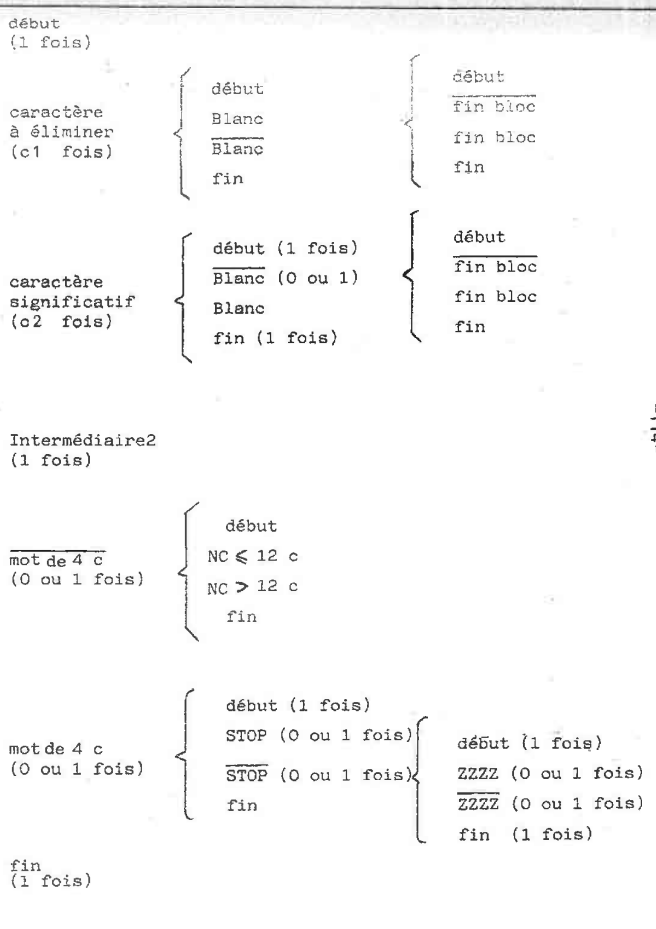


figure 7 : fichier logique de données à l'entrée

figure 8 : structure hiérarchique du programme



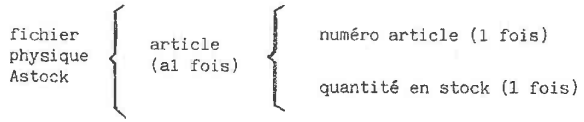


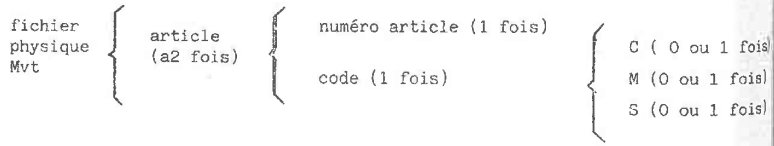


I - 4 - Application de la méthode L.C.P. à l'exemple des mises à jour.

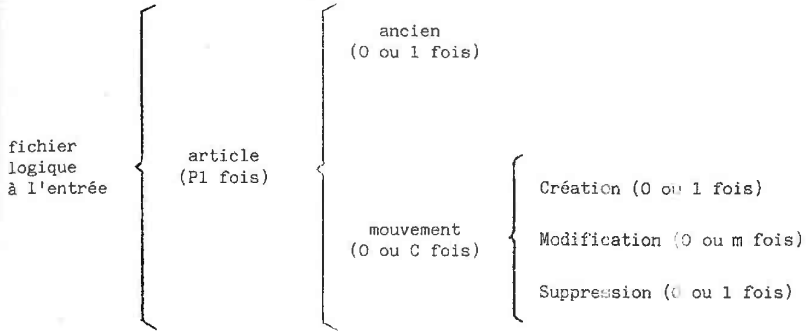
Chaque article du fichier Mvt porte un code (C, M ou S). Ces codes ne sont pas exclusifs. Par contre, pour un article, on peut avoir un article dans le fichierAstock sans maj, un article dans le fichierAstock avec maj, un article dans Mvt sans correspondant dans le fichier des stocks (cas d'une création). On peut avoir une création suivie de modifications, des modifications suivies d'une suppression (mais pas de modification après une suppression).

I - 4 -1- Description des structures hiérarchiques à l'entrée





A partir de ces deux structures physiques, on doit créer une nouvelle structure logique tenant compte de ces deux structures : la structure logique des données à l'entrée :



Remarque : le fichier logique des données à l'entrée est construit en tenant compte :

- des fichiers physiques à l'entrée (fichier des anciens stocks et des modifications)
- des actions de traitement et de sortie qui portent directement sur les données à l'entrée.

I - 4 -2- Description des structures hiérarchiques de sorties :

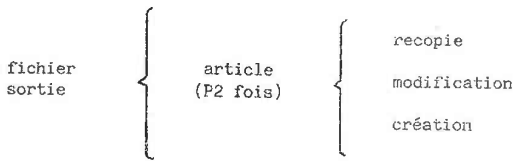
Chaque article du fichier sortie est obtenu par recopie d'un article de l'ancien stock, par modification ou par création.

I - 4 -3- Description du programme :

L'examen des structures d'entrée et de sortie met en évidence l'apparition de article P1 fois dans le fichier logique d'entrée et P2 fois dans le fichier de sortie ($P2 \leqslant P1$)

L'ensemble des articles figurant dans le fichier de sortie doit être éclaté en trois sous-ensembles en raison des divers traitements qui sont à l'origine du fichier de sortie :

- articles recopiés (pas de modification)
- articles créés
- articles modifiés



Remarque : le sous-ensemble des articles supprimés ne figure pas dans le fichier de sortie. Ce sous-ensemble correspond à l'action de suppression, non répertoriée en sortie.

Pour déterminer la décomposition des sous-ensembles de traitements, établissons une table de vérité pour l'ensemble des articles, en tenant compte de leur appartenance aux fichiers physiques d'entrée.

Notons :-A la rubrique correspondant à un article existant (contenu dans le fichier Astock)

-M la rubrique correspondant à un article à modifier (contenu dans le fichier Mvt)

$\bar{A}M$ correspond à la création d'un nouvel article (il n'appartient pas au fichier Astock, mais au fichier Mvt).

	recopie	création	modification	suppression
$\bar{A}\ \bar{M}$	Ø	Ø	Ø	Ø
$\bar{A}\ M$		1	0 ou m	0 ou 1
$A\ \bar{M}$	1			
$A\ M$			0 ou m	0 ou 1

Remarque : Dans l'énoncé du problème, nous avions supposé qu'un nouvel article pouvait être modifié ou supprimé (ce qui correspond dans la table précédente à $\bar{A}M$).

Le tableau de décomposition en séquence est présenté figure 9.

La déduction du programme final à partir d'un tel schéma, n'est pas immédiate : en effet, l'utilisateur devra se préoccuper, lors de l'écriture du code, du parcours des deux fichiers, celui-ci ne s'effectuant pas de la même manière en fonction des actions.

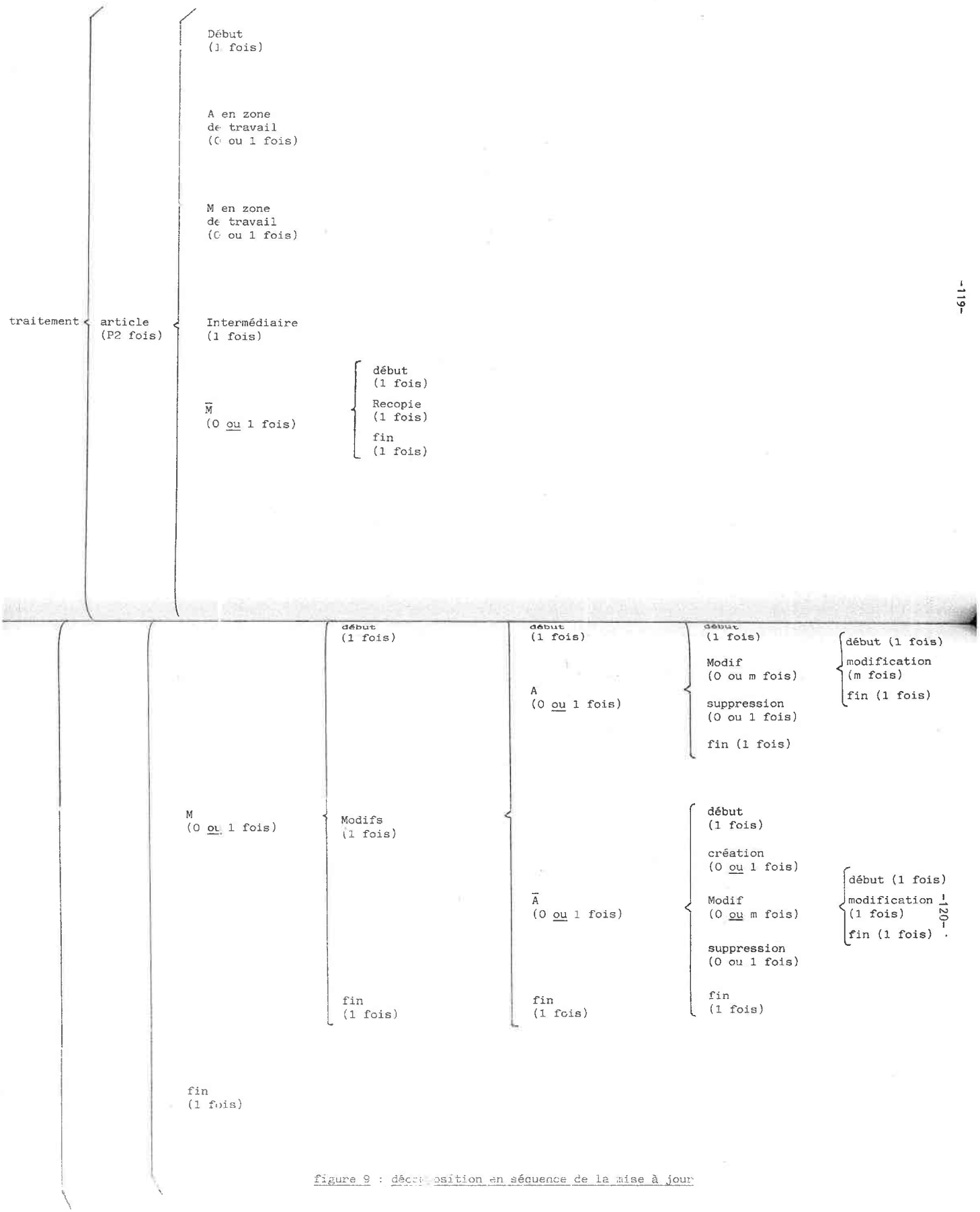


figure 9 : décomposition en séquence de la mise à jour

I - 4 -4- Conclusion

Avec cet exemple de mise à jour du fichier des stocks, nous avons traité le cas où plusieurs modifications pouvaient être effectuées sur un même article. Le programme proposé prend en compte chaque modification sans se préoccuper de la suivante.

Dans l'énoncé de ce problème, nous avions prévu le cas où un article pouvait être supprimé après modifications : construire un programme qui n'effectuera les modifications que si l'article n'a pas été supprimé (ceci nécessite un premier parcours de la liste des modifications et, en fonction du résultat, un deuxième parcours prenant en compte chaque modification).

La méthode L.C.P. ne permet pas de traiter ce problème : les modifications sont effectuées une à une (le traitement est directement associé à la lecture d'un enregistrement). Elle n'apporte pas de méthode permettant de choisir entre l'introduction effective ou non d'un intermédiaire.

I - 5 - Conclusion sur la méthode L.C.P.

Cette méthode proposée par WARNIER possède un certain nombre de qualités parmi lesquelles on peut citer :

- la structuration des données à l'entrée et à la sortie qui permet d'obtenir des programmes structurés.

- l'introduction de fichiers logiques lorsque les fichiers physiques ne permettent pas de résoudre le problème posé.

On peut aussi émettre des critiques qui portent surtout sur le manque de guide et de méthode pour introduire et définir les fichiers logiques. L'utilisateur sait qu'il doit introduire un fichier logique quand il ne peut facilement passer des données aux résultats, mais il est laissé à son intuition quant à la manière de l'introduire. Il semble, qu'en fait, il est fortement guidé par les sorties mais ceci n'apparaît pas explicitement (les sorties sont utilisées en fin d'analyse pour contrôler le programme).

Si on veut suivre la méthode jusqu'au bout, cela pose un certain nombre de problèmes. En effet, l'établissement des listes d'instructions par catégorie est fastidieuse et souvent source d'erreur. De plus, elle ne permet pas d'avoir une vision globale du problème. Dès le début de l'analyse, la résolution du problème est directement liée à son exécution : on doit se préoccuper de l'ordre des instructions. De plus, cette méthode s'applique à des types de problèmes restreints.

I - 6 - Comparaison avec notre méthode

La structuration des données et des résultats et l'introduction des suites intermédiaires sont des idées communes aux deux méthodes.

Notre étape de transformations successives correspond chez WARNIER à sa définition progressive du fichier logique introduit.

Notre méthode propose en plus :

- un guide pour introduire les intermédiaires
- un guide pour les définir
- des règles de transformation permettant d'améliorer l'efficacité du premier algorithme (et un système générant automatiquement le programme transformé).

II - La METHODE JACKSON

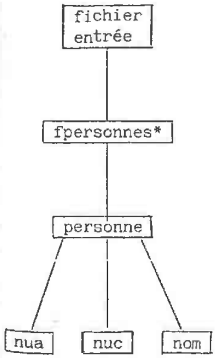
C'est une méthode rationnelle de construction de programmes fondée sur une approche progressive et sur les concepts de la programmation structurée. JACKSON a établi une classification de problèmes de manière intuitive [JAC, 75] et propose une méthode de résolution pour chaque type : algorithmes dans le cas de structures de données multiples, avec retour arrière, dans le cas des ruptures, de conflits de structures. On trouvera un essai de formalisation de cette méthode et la justification de son adéquation dans [HUG, 79].

Ici encore, l'analyse d'un problème repose sur la confrontation des structures d'entrées et des structures de sorties.

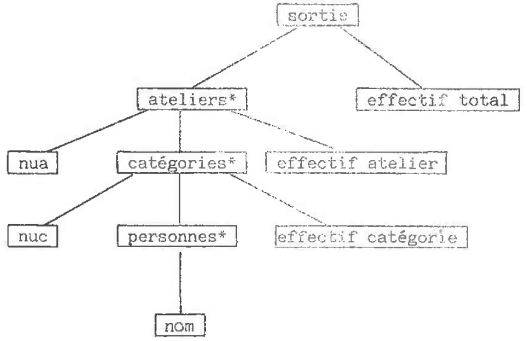
II - 1 - Aperçu de la méthode sur un exemple : les ateliers

II - 1 -1- Définition des structures d'entrée et de sortie

Donnons une représentation arborescente des deux structures imposées par l'énoncé du problème.



structure d'entrée



structure de sortie

Afin d'établir un programme qui, à partir du fichier d'entrée décrit précédemment,produise le fichier de sortie, nous cherchons à expliciter les correspondances entre ces deux structures arborescentes.

II - 1 -2- Correspondance entre les deux structures

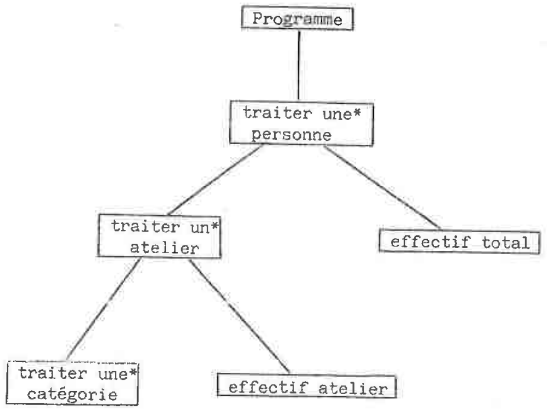
. Il y a une correspondance au premier niveau : au fichier entrée, on peut faire correspondre le fichier sortie. Lors de l'exécution, nous souhaitons qu'un fichier entrée fournisse un fichier sortie.

. Peut-on faire correspondre fpersonnes avec ateliers ? ateliers est bien défini à partir de fpersonnes, mais il n'y a pas bijection entre ces deux ensembles. Il n'y a pas conflit de structures entre fpersonnes et ateliers : elles ont en commun un plus petit élément : une personne. Un atelier représente un ensemble de personnes ayant certaines caractéristiques : ces personnes ont le même numéro d'atelier. fpersonnes est un ensemble de personnes, de toutes les personnes de l'usine, sachant qu'elles sont classées par numéro d'atelier. Il sera relativement facile de passer d'un élément de fpersonnes à un élément de ateliers moyennant certaines comparaisons.

II - 1 -3- Construction du programme

Le programme va être construit à partir de l'élément commun aux deux structures. On va être amené à introduire une instruction correspondant à la lecture d'une personne dans le fichier entrée ; cette instruction sera exécutée pour chaque personne du fichier. A une personne sera associé un traitement permettant de définir le fichier de sortie. Ce traitement va être défini en fonction du numéro d'atelier de la personne, puis de son numéro de catégorie.

La structure du programme est représentée par :



JACKSON propose ensuite de définir le travail à effectuer en fonction des opérations élémentaires disponibles dans le langage de programmation choisi et d'allouer chaque opération à un composant adapté à la structure du programme.

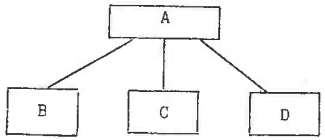
II - 2 - Présentation de la méthode

L'idée essentielle de la méthode proposée par JACKSON repose sur la décomposition hiérarchique des structures imposées dans l'énoncé du problème et sur la recherche de correspondances entre ces diverses structures. La structure de données est un guide important pour créer la structure du programme.

II - 2 -1- Représentation des structures d'entrées et de sorties sous forme d'arbres.

Dans un premier temps, on se contente de représenter de manière arborescente les diverses structures (d'entrée et de sortie) imposées dans l'énoncé du problème. Cette décomposition est assurée grâce à l'utilisation des trois composantes de base : la concaténation (ou traitement séquentiel), l'itération et la sélection. JACKSON représente ces composantes par des diagrammes :

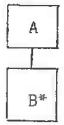
- La séquence est représentée par :



dans lequel A est le résultat de B suivi de C, suivi de D.

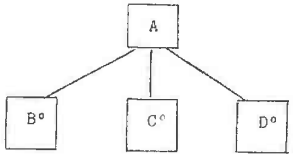
Remarque : Ces diagrammes seront utilisés aussi bien pour représenter des structures de données que des programmes (auquel cas, A, B, C, D seront des algorithmes).

- L'itération :



* désigne les multiples occurences de B.

- La sélection :



est indiquée par °. Cela signifie que A est défini soit par B, soit par C, soit par D.

Remarquons que la représentation proposée peut être considérée comme une notation d'expressions régulières (confère les travaux de Hugues, [HUG, 79]).

Notre but est d'établir la structure du programme permettant, à partir de la structure de données décrite, d'obtenir le résultat. Cherchons à expliciter des correspondances entre ces deux structures.

II - 2 -2- Recherche de correspondances entre arbre d'entrée et arbre de sortie

Cette phase consiste à comparer les deux structures définies précédemment et à rechercher des parallèles entre les niveaux correspondants (en procédant de manière descendante).

Dans le cas où il n'y a pas de correspondances immédiates (exemple vu précédemment), on est amené à faire des décompositions supplémentaires sur les structures en introduisant des intermédiaires communs aux entrées et aux sorties. Cette phase consiste à décomposer le problème initial trop complexe pour être résolu tel quel en sous-problèmes plus simples et indépendants.

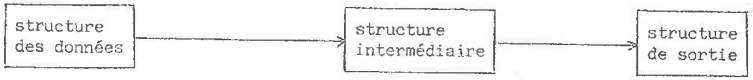
Remarque : Pour qu'il y ait correspondance à un niveau donné, il faut qu'il y ait correspondance au niveau inférieur. La génération de la correspondance se fait de manière récursive.

Cette nouvelle décomposition va nous permettre d'élaborer la structure du programme : les diagrammes utilisés seront les mêmes que pour les structures de données.

II - 2 -3- Construction du programme

1 - Le squelette du programme est déterminé à partir de la structure élaborée après comparaison des deux structures initiales. Il tient compte de l'introduction des intermédiaires.

2 - L'étape d'optimisation est très importante dans le cas où, avec l'introduction des intermédiaires, on a été amené à décomposer le problème :



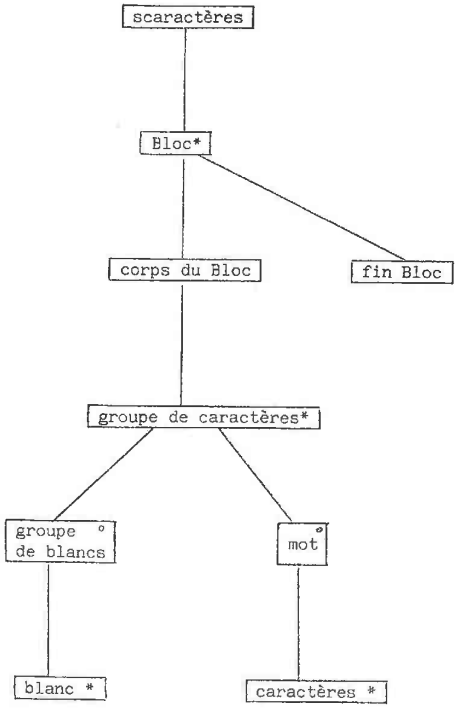
Ceci s'est traduit au niveau du programme par la définition de deux programmes indépendants. Pour rendre le programme final efficace, une étape de composition des divers programmes obtenus est parfois nécessaire (nous le verrons sur l'exemple des télégrammes).

3 - Pour finir, on définit le travail à effectuer en fonctions des opérations élémentaires disponibles dans le langage de programmation utilisé. Chaque opération élémentaire est affectée à une composante du programme schématisé en 2.

II - 3 - Application de la méthode JACKSON à l'exemple des télégrammes

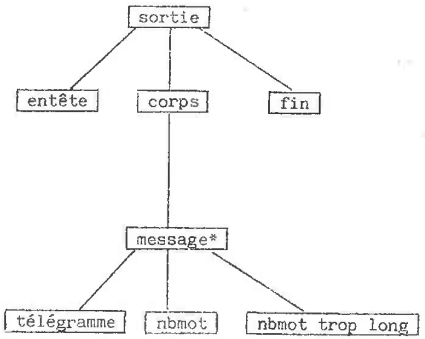
Nous voulons obtenir une suite de messages à partir d'une suite de caractères lus par bloc de longueur fixée, les caractères pouvant être des blancs.

II - 3 -1- Elaboration des structures d'entrée et de sortie



structure d'entrée

Remarque : Les éléments utilisés dans les diagrammes représentent ici des objets.



structure de sortie

Remarques : Le télégramme, au niveau sortie, n'a pas besoin d'être décomposé. Toutefois, on se souviendra de certaines modifications apportées sur le texte source :

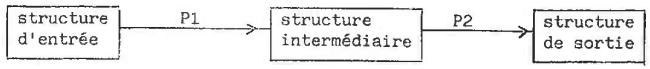
- Les mots "STOP" seront remplacés par des "." .
- Les mots de plus de 12 caractères seront tronqués (contiendront les 12 premiers caractères).
- Les mots "ZZZZ" seront supprimés.

II - 3 -2- Recherche de correspondances entre les deux structures

Une correspondance existe entre scaractères et sortie. Les deux structures ne sont pas compatibles : il y a un conflit entre un bloc et message. Les mots d'un bloc sont aussi les mots d'un message et ils apparaissent dans le même ordre, mais il n'y a pas de bijection entre ces deux ensembles. Un bloc ne représente pas un nombre entier de messages ; de la même façon, un message ne contient pas forcément un nombre entier de blocs. Et on ne peut trouver une troisième composante qui soit, soit un bloc, soit un message.

- Comment résoudre ce conflit ?

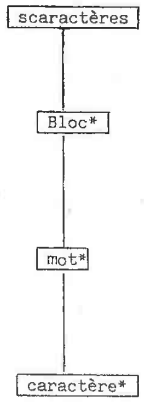
Lors de la construction du programme, on sera amené à utiliser une instruction correspondant à la lecture d'un bloc (qui a une existence physique), instruction qui sera exécutée pour chaque bloc donné. De même, on sera amené à définir un certain traitement pour chaque télégramme (induit par la nature du résultat). La structure du programme devra contenir une composante qui sera exécutée pour chaque bloc et une composante qui sera exécutée pour chaque télégramme, sachant que ces deux composantes ne sont pas compatibles. La solution consiste ici à introduire une structure intermédiaire permettant de résoudre le conflit.

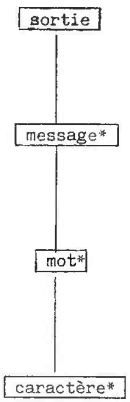


P1 représente le traitement permettant d'obtenir une structure intermédiaire à partir de la structure d'entrée et P2 la structure de sortie à partir de l'intermédiaire introduit.

De plus, pour résoudre le conflit existant, seul P1 contient une composante exécutée pour chaque bloc et P2 seul contient une composante exécutée pour chaque télégramme. Comment définir la structure intermédiaire en tenant compte de ces contraintes ? Appelons SI cette structure.

SI est construite sans aucune connaissance sur la notion de télégramme (P1 ignorant cette entité). SI est utilisée par un programme ne tenant pas compte du concept de bloc. Essayons de simplifier les deux structures précédemment définies en introduisant une nouvelle entité (le mot) qui peut être utilisé au niveau du télégramme et au niveau du bloc.





On peut établir une correspondance entre chaque paire de composants de même niveau excepté pour le groupe (bloc, message) (voir remarques précédentes). Le conflit de structures peut être résolu en introduisant comme SI une des trois entités : un caractère, un mot ou la suite lcaractères.

Choix de la structure intermédiaire la mieux adaptée :

Envisageons toutes les possibilités :

a) La suite scaractères

Choisir scaractères comme SI signifie que la sortie sera imprimée dans son ensemble (il n'y aura pas d'impression intermédiaire) et que la suite donnée scaractères sera elle aussi lue dans son ensemble et devra donc être entièrement stockée en mémoire, chaque caractère de fin de bloc remplacé par un caractère blanc.

Cette solution est inefficace: elle entraîne le stockage en mémoire des fichiers complets d'entrée et de sortie.

b) Le mot

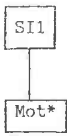
Choisir le mot comme SI signifie qu'à chaque mot lu, correspondra une sortie (il faudra tenir compte des mots spéciaux). Le mot représente la plus grande entité commune aux deux structures. Son stockage est peu coûteux.

c) Le caractère

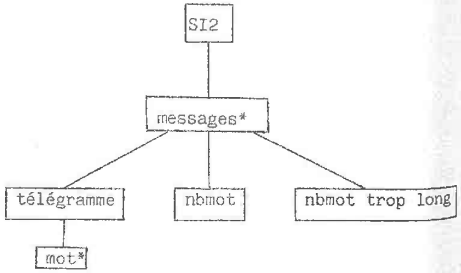
Choisir cette solution revient à décomposer un mot en une suite de caractères pour le reconstituer immédiatement. Son stockage est peu coûteux, mais il entraîne des traitements supplémentaires et oblige à stocker le mot.

Nous prendrons donc comme structure intermédiaire le mot. Ce choix est bien adapté à la solution du problème car on sera amené à comparer les mots de 4 lettres aux mots spéciaux "STOP" et "ZZZZ".

Comme l'on doit compter les mots trops longs de chaque télégramme, on peut compter le nombre de caractères du mot en le constituant. Donc P1 générera les couples (mot, nombre de caractères du mot).



structure intermédiaire vue de P1



structure intermédiaire vue de P2

II - 3 -3- Recherche du programme

Le problème est ramené à deux sous-problèmes indépendants :

- P1 : création de la suite de mots à partir de lcaractères
- P2 : obtention du résultat à partir de la suite de mots.

Cette solution n'est pas très efficace car elle entraîne la construction complète de la suite de mots avant son utilisation. Cette contrainte est imposée par la méthode proposée, elle n'est pas liée au type de problème à résoudre.

En effet, l'impression des mots d'un télégramme peut être effectuée à la lecture même de ces mots : le stockage d'un télégramme complet n'est pas nécessaire.

Comment éviter cette inéficacité ?

1 - Utilisation de processus parallèles :

Le processeur P2 est synchronisé sur le processeur P1, générateur de mots, par l'intermédiaire d'une zone tampon contenant un mot.

2 - Autre solution : composition de programmes ('Program Inversion')

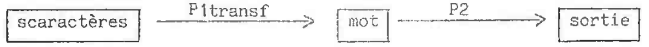
L'analyse du problème nous a conduit au diagramme suivant :

où P1 contient un sous-programme écrivant un mot dans le fichier de mots et P2 contient un sous-programme lisant un mot dans le fichier de mots.

Plutôt que de construire effectivement la suite de mots à l'aide du programme P1 et de l'utiliser en parallèle avec le programme P2, il serait plus judicieux de transformer l'un de ces deux programmes de manière à ce qu'il devienne sous-programme de l'autre (plutôt que de fournir l'ensemble des mots, il produirait un mot qui serait immédiatement traité).

2 - 1 - Transformons P1

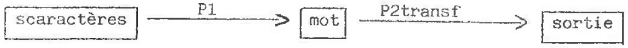
Cela signifie qu'il sera utilisé par P2 comme un remplacement du sous-programme initial de P2 qui lisait un mot.



Quand P2 a besoin d'un mot, il fait appel au sous-programme P1transf. P1transf lui fournit le mot suivant (mot qui dans le diagramme initial aurait été écrit par P1 dans le fichier des mots).

2 - 2 - Autre cas : transformons P2

P2transf remplace le sous-programme de P1 qui écrivait un mot dans le fichier des mots.



Lorsque P1 veut écrire un mot sur le fichier des mots, il fait appel au sous-programme P2transf. P2transf dispose du mot, l'utilise pour définir le résultat requis et se met à nouveau à la disposition de P1.

Pour écrire complétement le programme, on doit choisir maintenant une des deux solutions présentées.

Remarque : La deuxième solution correspond à la solution proposée en utilisant notre méthode de construction d'algorithme.

II - 3 -4- Conclusion

Les premières difficultés rencontrées dans la résolution de ce problème sont apparues avec le recherche de correspondances entre les structures données. Une structure intermédiaire s'impose. Le problème est de trouver cette structure ; nous sommes arrivés à plusieurs possibilités par tâtonnements. Rien ne garantit que nous puissions trouver dans chaque cas la structure appropriée. C'est l'intuition de l'analyste qui est mise en jeu.

Une structure intermédiaire ayant été mise en valeur, le problème consiste ensuite à écrire un programme utilisant cette structure et à rendre ce programme efficace.

II - 4 - Conclusion

Tout comme la méthode L.C.P., cette méthode repose sur les concepts importants de structuration des données et des résultats, représentés par des arborescences. L'idée fondamentale est, après construction des structures, d'établir des correspondances entre elles , ces correspondances permettant d'élaborer une structure de programme.

Lorsqu'il n'y a pas de correspondance immédiate, JACKSON propose de chercher des structures intermédiaires communes aux entrées et aux sorties par décomposition hiérarchique. Le programme final est alors obtenu par transformation des programmes intermédiaires introduits pour satisfaire à l'introduction des intermédiaires.

Ce qui est moins clair dans la méthode proposée, c'est la façon dont on s'y prend pour trouver ces intermédiaires : la recherche se fait "à tâtons". Qu'est-ce qui prouve que celui choisi est le plus adéquat ? Le manque de guide et de critères de choix ici fait défaut. Arrivé à l'étape finale d'amélioration de l'efficacité, se pose le problème : comment effectuer les transformations ?

III - AUTRES DEMARCHES EXISTANTES

III - 1 - Les méthodes syntaxiques

Une autre façon d'aborder ce type de problèmes consiste à être guidé uniquement par les données, à utiliser des algorithmes voisins de ceux de l'analyse syntaxique. Lorsqu'il y a des conflits de structures, plusieurs fichiers en entrée dont aucun ne peut être considéré comme guide (exemple des mises à jour), cette méthode n'est pas évidente à mettre en oeuvre. On est amené à penser à plusieurs choses à la fois :

- comment définir le résultat
- comment utiliser les données (gérer le parcours des fichiers en analyse de gestion)

Explicitons cette démarche sur l'exemple des ateliers (cf. chapitre 1, paragraphe III), en utilisant la syntaxe de MEDEE. L'analyse repose sur la notion d'automate d'état fini. Elle consiste à spécifier la grammaire du fichier en entrée et à construire l'analyseur correspondant. Cet analyseur est ensuite "garni" en associant les divers traitements imposés par la spécification du problème aux reconnaissances des termes du fichier.

- satelier suite des ateliers - eftotal entiereffectif total de l'entreprise - eft suite des effectifs *catelier définit un atelier - lpersonnes suite des personnes de l'entreprise	résultat = écrire satelier, eftotal eftotal = der eft satelier, eft = iter *catelier avec lpersonnes lpersonnes = suite donnée
*catelier	
- efatelier entier effectif d'un atelier - esatelier élément de satelier - efat suite des effectifs de l'atelier - nua entier numéro de l'atelier - atelier suite des catégories *catégories définit une catégorie d'un atelier - nuc entier numéro de catégorie - nom chaine d'une personne	eft = $\mathfrak{Q}$ eft + efatelier esatelier = $\mathfrak{Q}$ nua, atelier efatelier = der efat atelier, efat = jqa nua ≠ $\mathfrak{Q}$ nua iter *catégories avec lpersonnes prem (nua, nuc, nom) = tête (lpersonnes) prem eft = 0
*catégories	
- efcatégorie entier effectif d'1 catégorie - eatelier élément d'un atelier - efcat suite d'effectifs de la catégorie - spers suite des personnes de la catégorie *unecat définit une catégorie	efat = $\mathfrak{Q}$ efat + efcatégorie eatelier = $\mathfrak{Q}$ nuc, spers efcatégorie = der efcat spers, efcat = jqa nuc ≠ $\mathfrak{Q}$ nuc iter *unecat sur lpersonnes prem efat = 0
*unecat	
- espers chaine nom d'une personne	espers = nom efcat = $\mathfrak{Q}$ efcat + 1 (nua, nuc, nom) = $\text{suc}_{\text{lpersonnes}}$($\mathfrak{Q}$nua, $\mathfrak{Q}$nuc, $\mathfrak{Q}$nom) prem efcat = 0

Les ateliers

III - 2 - Approche machines abstraites [SCH, 78], [SCH, 79]

Elle répond à un souci de systématiser ce qui semble être la méthode de programmation la plus couramment admise : l'identification de problèmes déjà connus, d'où la nécessité de définir des schémas généraux permettant de définir des classes de problèmes et de les reconnaître au cours d'une analyse.

Ceci conduit à développer des méthodes d'analyses et des outils adéquats à l'utilisation des solutions de problèmes déjà traités.

La démarche méthodologique de programmation proposée se décompose en deux étapes :

1 - Organisation logique de l'information :

Elle consiste à introduire différents niveaux d'abstractions (relativement aux données et aux résultats). L'information est décomposée en un ensemble d'informations élémentaires.

Cet ensemble d'informations élémentaires est ensuite structuré suivant un des deux modèles : structure de file ou structure d'arbre. Ces deux modèles caractérisent les deux méthodes présentées dans [SCH, 78] :

-le traitement séquentiel associé aux files dans lequel on est amené à définir un ordre total sur l'ensemble considéré.

- le traitement arborescent sur lequel on définit un ordre partiel.

2 - Analyse du problème posé à partir de la structure d'information déduite précédemment

On essaie d'appliquer ici les nombreux algorithmes traitant les files ou les arbres suivant la structure choisie.

Pour aider à l'application systématique de ces principes, des outils de mise en oeuvre sont proposés concernant des modèles d'analyse (machines abstraites) et des schémas d'algorithmes. Les schémas d'algorithmes correspondent à trois problèmes fondamentaux :

- "parcours" : le traitement de l'ensemble est ramené à l'application d'une même action élémentaire à chacun des éléments de l'ensemble.
- "énumération partielle" : le traitement de l'ensemble est ramené à l'application d'une même action élémentaire à certains éléments de l'ensemble dotés d'une propriété caractéristique.
- "recherche associative" : le traitement de l'ensemble est ramené à l'application d'une action élémentaire à l'un des éléments de l'ensemble.

Comparaison avec notre méthode :

La comparaison avec cette démarche est celle que nous proposons porte sur plusieurs points :

1 - L'introduction des problèmes types peut être comparée aux différents types de définitions de MEDEE :

- ce sont des modèles dynamiques

- bien qu'il y ait davantage de problèmes types que de définitions MEDEE, l'éventail des problèmes que l'on peut résoudre avec cette démarche est incomplet.

2 - L'introduction des différents niveaux d'abstraction et le raffinement des structures est comparable aux travaux effectués en MEDEE sur les structures de données, avec l'introduction des intermédiaires adaptés à la définition des résultats, en moins systématique. En effet, le but ici est d'aboutir à une des deux structures connues : quant à la manière d'y arriver, elle est laissée à l'intuition de l'utilisateur.

IV - Conclusion

Nous n'avons pas effectué une liste exhaustive des travaux existants dans le domaine. Des études sur l'organisation des données [HER, 74] et leurs conséquences sur la structure des programmes, plus précisément dans le cadre de problèmes de gestion de fichiers (où intervient fréquemment la notion de rupture) ont abouti à l'élaboration d'un métalangage dans Cobol permettant de libérer le programmeur de la gestion des fichiers et réalisant "la cinématique automatique des fichiers".

- CHAPITRE 4 -

SYSTEME ET PROLONGEMENTS

CHAPITRE 4

SYSTEME ET PROLONGEMENTS

I - PRESENTATION DU SYSTEME :

Nous nous sommes attachés ici à transformer un algorithme MEDEE (écrit à l'aide de la syntaxe décrite en ANNEXE 1) supposé syntaxiquement correct.

Le système proposé a été écrit en PASCAL (IRIS 80). L'arbre des modules et les diverses listes introduites ont été implementés par des listes chaînées (correspondant aux pointeurs en PASCAL). C'est un système interactif.

I - 1 - Fonctions du système :

I - 1 -1- Fonctions locales

1.- Acquisition d'un algorithme MEDEE

Elle consiste à lire un algorithme MEDEE syntaxiquement correct sous forme linéaire et à générer une représentation abstraite de cet algorithme. Le rôle de ce processeur est de reconnaître les diverses formes syntaxiques autorisées et de les stocker en conséquence.

2.- Edition d'un algorithme à partir de sa représentation abstraite

Ce processeur a pour rôle de restituer à l'utilisateur un algorithme MEDEE sous forme linéaire à partir de l'arborescence ou représentation abstraite de l'algorithme en machine.

3.- La règle du pliage

Le système transforme la définition à plier en deux définitions consécutives dans la hiérarchie des définitions du module considéré.

4.- La règle du dépliage

Après vérification des préconditions nécessaires à l'utilisation de cette règle, le système remplace dans une définition un identificateur par sa définition et supprime celle-ci de la liste considérée.

5.- Fusion de deux définitions

Nous avons implementé la fusion de deux définitions itératives de même type et la fusion de deux conditionnelles (en envisageant tous les cas possibles pour chaque branche : cas des définitions simples et cas des modules) : les deux définitions sont supprimées de la liste considérée et remplacées par la définition résultat.

Dans le cas où les définitions faisaient intervenir des modules, ceux-ci sont supprimés et remplacés par le module résultat de la fusion de leurs définitions. La hiérarchie initiale des modules est alors modifiée : des modules qui n'étaient pas reliés entre eux ont maintenant des liens de parenté.

6.- Composition de deux itérations

Cette transformation a été implementée pour les divers types d'itérations possibles et s'accompagne de la vérification de la précondition nécessaire à savoir la suite considérée n'est-elle pas utilisée par ailleurs ?

7.- Décomposition d'itération

Elle correspond à l'implémentation des propriétés de l'opérateur aplat. une définition est modifiée ainsi que la suite des modules qui lui est associée.

8.- L'éclatement d'itération

Cette transformation est en cours d'implémentation.

Remarque : Les propriétés de l'opérateur écrire ne sont pas encore implémentées.

I - 1 -2- Mise en oeuvre d'une stratégie

Elle correspond à l'implémentation de la stratégie proposée au chapitre 3.

Après une présentation du module traité, le système demande à l'utilisateur la liste des définitions pour lesquelles la règle du pliage peut être utilisée. Après réponse, le système s'assure que les transformations des définitions proposées sont possibles auquel cas, il effectue les modifications nécessaires.

Dans un deuxième temps, le système recherche les fusions de définitions possibles dans ce module (recherche de définitions de même type, comparaison des domaines de définition) et en fournit la liste à l'utilisateur. Celui-ci a la possibilité de demander la réalisation ou non de ces transformations.

Pour la règle du dépliage, c'est l'utilisateur qui, après interrogation du système, fournit la liste des définitions concernées.

Le système propose l'application des règles d'éclatement ou de décomposition d'itération lorsque ces modifications sont possibles et effectue ou non ces modifications en fonction de la réponse de l'utilisateur.

Dans un dernier temps, le système recherche les définitions correspondant à une éventuelle composition d'itérations, vérifie les préconditions nécessaires et propose la liste des compositions possibles. Après transformation, la version du module transformé est fournie et le système passe au module suivant (conformément à la hiérarchie proposée).

Lorsque l'algorithme final est obtenu, la version complète est fournie.

Remarques : Dans le système actuel, une fois une transformation effectuée, nous n'avons pas la possibilité de revenir à l'étape précédente. Il serait intéressant de pouvoir conserver, dans une construction interactive d'algorithmes, les diverses étapes effectuées afin d'envisager diverses possibilités de transformations.

I - 2 - Représentation des objets

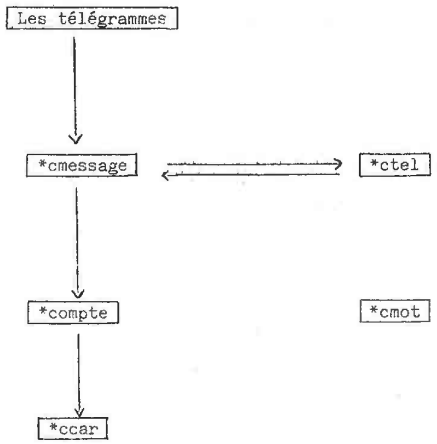
I - 2 -1- Un énoncé

Un énoncé MEDEE est constitué par un ensemble de modules. Le choix de représentation effectué consiste à associer à un module :

- son contenu (nom du module et liste des définitions)
- un module fils
- un module frère gauche
- un module frère droit.

Que représentent ces trois modules ? Pour les besoins des transformations à implémenter, nous avons été amenés à introduire la notion de hiérarchie ou de niveaux entre les divers modules constituant un algorithme : le module principal n'a pas de module frère gauche ou frère droit. Son module fils correspond au premier module introduit dans le module principal, soit celui introduit dans la première définition (ce choix est purement arbitraire).

Prenons l'exemple des télégrammes :



Hiérarchie des modules dans l'exemple des télégrammes

Remarque : Des modules peuvent apparaître hiérarchiquement au même niveau et ne pas avoir de relations entre eux (par exemple *compte et *cmot) : cela correspond en fait, à deux algorithmes différents, l'un de définition des résultats et l'autre celui de définitions des suites intermédiaires.

I - 2 -2- Un module

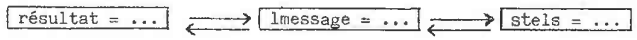
Il est constitué d'un nom de module et d'une liste de définitions. Celle-ci est une structure de même type que celle utilisée au niveau des modules. On lui associe :

- son contenu (il s'agit du contenu de la première définition de la liste)
- la définition suivante
- la définition précédente.

Remarque : La hiérarchie introduite au niveau des définitions correspond à leur ordre d'introduction dans la construction du module.

exemple : Prenons le module principal de l'exemple des télégrammes.

*Les télégrammes



I - 2 -3- Une définition

Se décompose en deux parties :

- le lexique
- la définition formelle.

a) le lexique

comprend:

- le contenu (définition informelle du premier objet avec le nom de l'objet, son type et un commentaire ou description informelle).
- liste des autres objets définis.

b) La définition formelle

Elle se compose :

- d'un identificateur caractérisant le type de la définition
- de la définition proprement dite.

Cette définition peut être de plusieurs types. En fonction du type, elle contiendra :

- une expression (expression simple)
- une expression booléenne et deux entités pouvant être soient des expressions, soient des modules (dans le cas d'une définition conditionnelle)
- un nom de module, un pointeur sur un module, une ou plusieurs expressions dans le cas des itératives.

I - 3 - Réalisation des fonctions

Les transformations de pliage et de dépliage sont locales au module dans lequel elles sont effectuées : elles n'entraînent pas de répercussion sur les autres modules de l'algorithme.

Il n'en est pas de même des autres transformations :

- la fusion de deux définitions peut entraîner la fusion de deux modules
- la composition de deux itération entraîne la fusion de deux modules
- la décomposition d'une itération entraîne la création d'un nouveau module et d'un niveau de plus dans la hiérarchie initiale des modules.

I - 3 -1- Acquisition d'un algorithme MEDEE

Le but est de construire, à partir d'un algorithme MEDEE syntaxiquement correct, l'arbre des modules conformément à la hiérarchie proposée.

Algorithme :

```
% construction de l'arbre des modules appelé arbre %

arbre0 = nil

arbre = tant que  ¬ fin de l'algorithme

        faire  % constituer un module %

            Lire nom du module
            listedef0 = nil    % liste des définitions du module courant %
            Lire une définition
            tant que  définition ≠ nil
                    faire  insérer cette définition dans listedef
                           Si définition introduit un module alors créer module
                           Lire une définition

                    ffaire
            Affecter listedef au module courant
            Affecter ce module au noeud de l'arbre des modules

        ffaire

% fin construction de l'arbre des modules %
```

Remarques : La constitution de la liste des définitions entraîne la constitution de la liste des modules fils du module courant. Le choix de représentation adopté entraîne qu'un module possède un seul module fils, les autres modules étant déclarés comme modules frères de ce module fils. Ce choix impose de regarder, lors de la constitution des modules si le nouveau module introduit est le premier par rapport au module courant (ce qui correspond ici à la première définition du module courant introduisant un module).

plémentation :

L'implémentation de cet algorithme est décomposée en un certain nombre procédures parmi lesquelles on peut citer :

- lecture d'une définition globale
- lecture d'une définition informelle
- lecture des divers types de définitions
- insertion d'une définition dans une liste de définitions
- insertion d'un module dans l'arbre des modules
- recherche d'un module dans l'arbre
- ....

I - 3 -2- Edition d'un algorithme à partir de sa représentation abstraite

lgorithme :

```
% Ecriture d'un algorithme MEDEE %
Ecrire module principal
M = fils du module principal
Pile = vide
Boucle % écriture de la liste des modules issue de M %
Y = dernier M ≠ nil
empiler y  % conserve le dernier module non vide de la liste %
M = fils de Y
Si M ≠ nil alors  Boucle
          sinon tant que pile <> vide

                faire
                   dépiler Y
                   M = frère gauche de Y
                   Si M ≠ nil alors
                         faire
                            empiler M
                            Boucle

                         ffaire

                ffaire

% fin écriture de l'algorithme %
```

```
Boucle
    tant que   M ≠ nil
        faire  Ecrire M
               M = frère droit de M
        ffaire
Fin Boucle
```

I - 3 -3- Les transformations

1. - Stratégie

a) Algorithme général

Il décrit l'application récursive de la stratégie de transformation d'un module à l'arbre des modules.

```
% algorithme de transformation d'un algorithme MEDEE %

Pile = vide
Transformer module principal
M = module fils
Parcours de M
tant que  pile ≠ nil
      faire   dépiler Y
              M = frère gauche de Y
              Si M ≠ nil alors Parcours de M
      ffaire

% fin de l'algorithme général de transformations %
```

Cet algorithme fait appel à deux sous-algorithmes :

- Transformer qui correspond à l'application de la stratégie de transformation au niveau d'un module.
- Parcours de M qui correspond à l'application de Transformer à la liste des modules associée à M ainsi qu'à la liste des modules associée au dernier module non vide de cette liste.

b) Transformations au niveau d'un module

Elles s'effectuent en déroulant séquentiellement et dans l'ordre retenu les divers algorithmes correspondant aux reconnaissances de schémas syntaxiques et aux règles de transformations.


```
Transformer (M)  % méta-algorithme de transformation d'un module %
        Ecrire (M)
        Plirequete          % requête concernant la règle du pliage %
        Tablefusion (ltfusion)  % recherche et constitution de la liste
                                  ltfusion des définitions susceptibles
                                  d'être fusionnées dans M %
        si ltfusion ≠ nil  alors  Fusionrequête
        Décomposition requête   % recherche des définitions susceptibles d'être
                                  décomposées et propositions à l'utilisateur %
        Eclatement requête      % recherche des définitions susceptibles d'être
                                  éclatées et propositions à l'utilisateur %
        Tablecomposition (ltcomp)
                                % recherche et constitution de la liste ltcomp
                                  des définitions susceptibles d'être composées,
                                  avec vérification des préconditions nécessaires %
        si ltcomp ≠ nil alors compositionrequête
        Ecrire (M)
finTransformer
```

c) Transformation de la liste des modules associée à un module

```
Parcours (M)
    tantque  M ≠ nil
        faire  Transform (M)
               M = module frère droit de M
        ffaire

    Y = dernier M ≠ nil
    empiler Y
    X = module fils de Y
    Si X ≠ nil  faire
                    Parcours (X)
                ffaire
finparcours
```

2.- Une transformation particulière : la composition

Elle correspond à Tablecomposition et compositionrequête dans Transformer.

a) Compositionrequête consiste à proposer à l'utilisateur la liste des définitions du module pouvant être composées et en fonction de la réponse à effectuer ou non la composition.

Compdéfinition (M, D1, D2, ND) % composition de deux définitions itératives D1 et D2 . ND est la définition résultat %

- D1 définition du résultat (informelle, type, formelle) -D2 définition de la suite intermédiaire - D définition du résultat après transformation - M module dans lequel s'effectue la composition * RINTERM ND est de même type que D2 * RAUTRE correspond au cas où D2 est une itération avec condition d'arrêt et domaine d'itération - Suppression des deux définitions D1 et D2 dans M et remplacement par ND	ND (informelle) = D1 (informelle) ND (formelle, type) = si type de D1 = ITGENER alors *RINTERM sinon *RAUTRE M = Suppression (ω M, D1, D2, ND)
*RINTERM	
Creedef définition formelle de ND à partir de celles de D1 et D2. Envisage tous les cas possibles pour D2	Type de ND = Type de D2 formelle de ND = Creedef (formelle de D1, formelle de D2)
*RAUTRE	
	Type de ND = si type de D2 = ITARRET1 alors ITARRET1 sinon ITARRET2 formelle de ND = Creedef (formelle de D1, formelle de D2)

Remarque: ITGENER désigne une définition itérative de la forme:

x = iter *cx sur 1

ITARRET1 : x = jqa arret iter *cx avec 1

ITARRET2 : x = jqa arret iter *cx sur 1

b) Tablecomposition s'effectue en deux passes.

1 - Parcours des définitions du module

Le choix effectué consiste à pointer sur toutes les définitions itératives du module et à conserver ces définitions dans Tcomp.

Remarque : Cet algorithme est commun à la fusion et à la composition.

2 - Parcours de cette liste

avec d'une part :

- recherche des itérations avec domaine de définition=1
- et, d'autre part, l est-elle définie par itération ?

```
ltcomp    % constitution de la liste des définitions à composer %
          Tcomp1 = Tcomp
          tant que Tcomp 1 ≠ nil
                faire
                   si   définition de Tcomp1 avec domaine de définition
                      alors faire
                         si domaine défini dans Tcomp  faire
                                                       Vérification
                                                       insére (ltcomp)
                                                       ffaire
                            ffaire
                      sinon définition suivante dans Tcomp&
                ffaire
fin ltcomp
```

Vérification consiste à s'assurer que la suite peut effectivement disparaître. Pour cela, on doit vérifier que cette suite n'est pas utilisée dans le module traité ou dans l'un des modules frères droit de ce module.

```
Vérification (arrêt : Booléen, M)
   arrêt = faux
   Pile = nil
   Vérifchemin (M)
   tant que Pile ≠ nil ou arret = faux
      faire
         dépiler (Y)
         X = frère gauche de Y
```

```
Si X ≠ nil   faire  empiler (X)
                    X = fils de X
                    Verifchemin (X)

             ffaire

      ffaire

finvérification
```

Vérifchemin est une procédure effectuant la vérification sur la liste des modules issues de M.

```
Verifchemin ( M:MODULE, arret: Booléen )
    tantque M ≠ nil ou arret=faux
        faire
            Verifmodule (M)
            M = frère droit de M
        ffaire

    Y = dernier M ≠ nil
    Si arret = faux  alors empiler Y
    M = fils de M
    Verifchemin (M)
fin Verifchemin
```

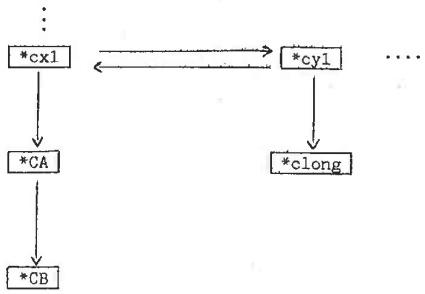
Verifmodule (M: MODULE)

arret booléen *Verifdef vérifie si la définition courante utilise la suite l	arret = iter *Verifdef dans ldef de M prem arret = faux
*Verifdef	
l identificateur de suite	arret = si l est utilisé dans def alors arret=vrai alors arret=ωarret

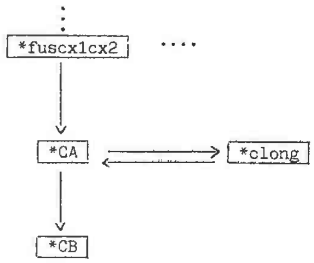
Remarque : La composition de deux définitions itératives s'accompagne de la fusion de deux modules (opération qui est définie dans Creedef).

Cette fusion ne s'effectue que sur des modules frères (ayant des liens entre eux) et entraîne un bouleversement au niveau de la hiérarchie initiale des modules.

ex:

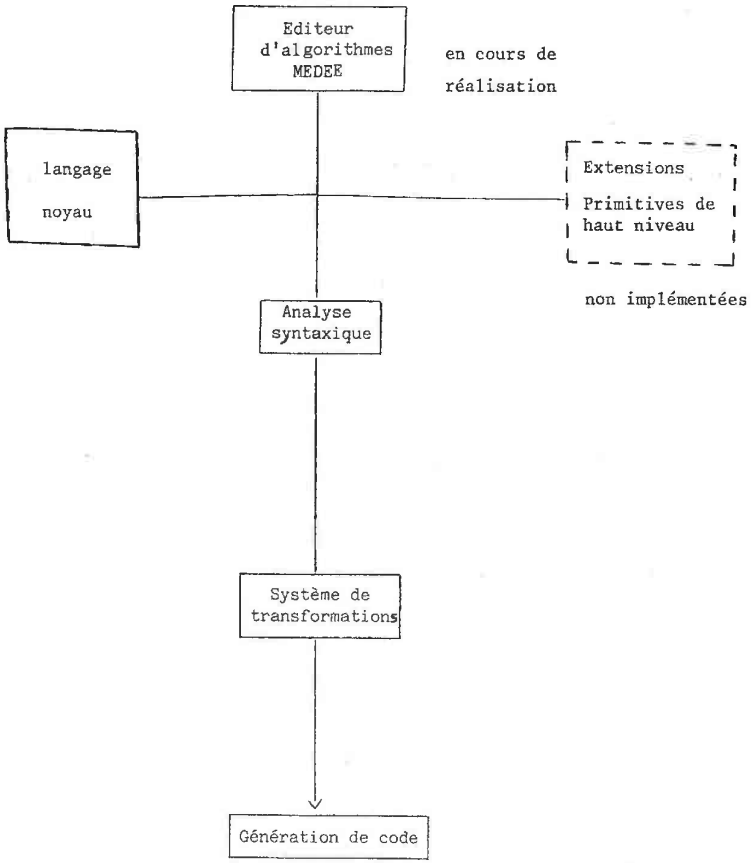


La fusion des modules *cx1 et cy1 entraîne la nouvelle structure où *CA et *clong sont des modules frères :



I - 4 - Conclusion

Le système de transformation en cours de réalisation est actuellement déconnecté au niveau utilisation des applications en cours sur MEDEE (compilateur, éditeur ....). La raison essentielle est due à la nécessité d'implémenter les extensions du langage introduites ici. La réalisation d'un système d'aide à la construction d'algorithmes MEDEE (schématisé p. 149) est en cours.



Système d'aide à la construction d'algorithmes MEDEE

I - 5 - Exemple d'utilisation

Utilisons notre maquette du système de transformations sur l'exemple des télégrammes dans lequel l'énoncé a été légérement modifié: les mots comptables sont tous les mots différents de "STOP" et "ZZZZ", les mots trop longs ceux de plus de 12 caractères et les messages imprimés correspondent aux messages sources dans lesquels les mots "STOP" et "ZZZZ" ont été supprimés et les mots trop longs tronqués à partir du 12ème caractère.

La mise en oeuvre de la maquette est présentée (p. 150 à 157).

```
!S-RUN
CORE 745
LOADM 7S-SAM2O
DONNEE ?DONN
- SFER/PASCAL-SYSTEM,VERS. 1/11/79-U4
?
```

ALGORITHME INITIAL:

```
* TELEGRAM

RESULTAT = ECRIRE LMESSAGE
LMESSAGE = ITGENER  *CMESSAGE SUR  STELS
STELS = ITARRET1 TELVIDE  *CTEL

* CMESSAGE

MESSAGE = UNTEL,TOTMOT,TLONG
TOTMOT = DER NBMOT
TLONG = DER NBLONG
UNTEL = ITGENER  *CMES SUR  TEL
NBMOT, NBLONG = ITGENER  *CNBRLONG SUR  TEL

* CTEL

TEL, TELVIDE = ITARRET1 FINTEL  *CMOT

* CMOT

MOT = ITARRET2 FINMOT  *CPARCOUR SUR  SCARS
SCARS = SI FINBLOC  ALORS DONNEE  SINON <>
TELVIDE = FINTEL ET @FINTEL
PREM FINTEL =FAUX

* CPARCOUR

MOT = @MOT /CAR
FINMOT = (CAR=' ')
PREM MOT =VIDE

* CMES

UNMOT = SI MOT ='STOP' OU MOT ='ZZZZ'  ALORS VIDE  SINON  *CMOTSTOP

* CNBRLONG

NBMOT = SI MOT ='STOP' OU MOT ='ZZZZ'  ALORS @NBMOT  SINON @NBMOT +1
NBLONG = SI NBCAR <12  ALORS @NBLONG  SINON @NBLONG +1
NBCAR = DER SNBCAR
SNBCAR = ITGENER  *CCAR SUR  MOT
PREM NBMOT =0
PREM NBLONG =0

* CCAR

NBCAR = @NBCAR +1
PREM NBCAR =0

* CMOTSTOP

UNMOT = SI NBCAR <12  ALORS MOT  SINON MOT[1,12]
```


REMARQUES:

PLIAGE: DONNER LA LISTE DES IDENTIFICATEURS A PLIER TERMINEE PAR "."
EXEMPLE: SOIT X=F(Z) UNE DEFINITION DU MODULE TRAITE
DONNER X ENTRAINERA LA TRANSFORMATION X=F(Y) ET Y=Z
FUSION: LA LISTE DES DEFINITONS SUCCEPTIBLES D'ETRE FUSIONNEES EST FOURNIE.SI VOUS ETES D'ACCORD POUR FUSIONNER REPONDRE "OUI" SINON DONNER LA LISTE DES COUPLES DE DEFINITIONS A FUSIONNER TERMINEE PAR "."
DEPLIAGE: DONNER LA LISTE DES COUPLES D'IDENTIFICATEURS A DEPLIER TERMINEE PAR "."
EXEMPLE: SOIENT X=F(Y) ET Y=Z DEUX DEFINITIONS DU MODULE TRAITE
DONNER X ET Y ENTRAINERA LA TRANSFORMATION: X=F(Z)
COMPOSITION: MEMES REMARQUES QUE POUR LA FUSION

```
* TELEGRAM

RESULTAT = ECRIRE LMESSAGE
LMESSAGE = ITGENER  *CMESSAGE SUR  STELS
STELS = ITARRET1 TELVIDE  *CTEL
```

TRANSFORMATIONS A EFFECTUER DANS TELEGRAM

```
PLIAGE?
?.
DEPLIAGE?
?.
LISTE DES COMPOSITIONS POSSIBLES :  LMESSAGE  AVEC  STELS
COMPOSITION ?
?OUI
```

MODULE TRANSFORME:

```
* TELEGRAM

RESULTAT = ECRIRE LMESSAGE
LMESSAGE = ITARRET1 TELVIDE  *FUMESTEL
```

```
* FUMESTEL  % RESULTAT DE LA COMPOSITION DE CMESSAGE AVEC CTEL      %

MESSAGE = UNTEL,TOTMOT,TLONG
TOTMOT = DER NBMOT
TLONG = DER NBLONG
UNTEL = ITGENER  *CMES SUR  TEL
NBMOT, NBLONG = ITGENER  *CNBRLONG SUR  TEL
TEL, TELVIDE = ITARRET1 FINTEL  *CMOT
```

TRANSFORMATIONS A EFFECTUER DANS FUMESTEL

```
PLIAGE?
?.
LISTE DES FUSIONS POSSIBLES :
UNTEL    ET NBMOT
FUSION ?
?OUI
DEPLIAGE?
?MESSAGE TOTMOT
?MESSAGE TLONG .
LISTE DES COMPOSITIONS POSSIBLES :  UNTEL      AVEC  TEL
COMPOSITION ?
?OUI
```

MODULE TRANSFORME:

```
* FUMESTEL  % RESULTAT DE LA COMPOSITION DE CMESSAGE AVEC CTEL      %

MESSAGE = UNTEL,DER NBMOT ,DER NBLONG
UNTEL, NBMOT, NBLONG, TELVIDE = ITARRET1 FINTEL  *FUMESMOT
```



```
* FUMESMOT  % RESULTAT DE LA COMPOSITION DE FUMESNBR AVEC CMOT      %

UNMOT = SI MOT =´STOP´ OU MOT =´ZZZZ´  ALORS VIDE  SINON  *CMOTSTOP
NBMOT = SI MOT =´STOP´ OU MOT =´ZZZZ´  ALORS @NBMOT  SINON @NBMOT +1
NBLONG = SI NBCAR <12  ALORS @NBLONG  SINON @NBLONG +1
NBCAR = DER SNBCAR
SNBCAR = ITGENER  *CCAR SUR  MOT
PREM NBMOT =0
PREM NBLONG =0
MOT = ITARRET2 FINMOT  *CPARCOUR SUR  SCARS
SCARS = SI FINBLOC  ALORS DONNEE  SINON <>
TELVIDE = FINTEL ET @FINTEL
PREM FINTEL =FAUX
```

TRANSFORMATIONS A EFFECTUER DANS FUMESMOT

```
PLIAGE?
?MESS
MESS     N EST PAS DEFINI DANS CE MODULE
?.
LISTE DES FUSIONS POSSIBLES :
UNMOT    ET NBMOT
FUSION ?
?OUI
DEPLIAGE?
?NB NBL
DEPLIAGE IMPOSSIBLE DANS CE MODULE
?.
```

MODULE TRANSFORME:

```
* FUMESMOT  % RESULTAT DE LA COMPOSITION DE FUMESNBR AVEC CMOT      %

UNMOT, NBMOT = SI MOT =´STOP´ OU MOT =´ZZZZ´  ALORS VIDE,@NBMOT  SINON  *NOUCMOTS
NBLONG = SI NBCAR <12  ALORS @NBLONG  SINON @NBLONG +1
NBCAR = DER SNBCAR
SNBCAR = ITGENER  *CCAR SUR  MOT
PREM NBMOT =0
PREM NBLONG =0
MOT = ITARRET2 FINMOT  *CPARCOUR SUR  SCARS
SCARS = SI FINBLOC  ALORS DONNEE  SINON <>
TELVIDE = FINTEL ET @FINTEL
PREM FINTEL =FAUX
```

```
 * NOUCMOTS  % RESULTAT DE LA FUSION DE      NBMOT    AVEC CMOTSTOP  %

 NBMOT = @NBMOT +1
 UNMOT = SI NBCAR <12  ALORS MOT  SINON MOT[1,12]
```

TRANSFORMATIONS A EFFECTUER DANS NOUCMOTS

```
PLIAGE?
?.
DEPLIAGE?
?.
```

MODULE TRANSFORME:

```
* NOUCMOTS  % RESULTAT DE LA FUSION DE      NBMOT    AVEC CMOTSTOP  %

NBMOT = @NBMOT +1
UNMOT = SI NBCAR <12  ALORS MOT  SINON MOT[1,12]



* CCAR

NBCAR = @NBCAR +1
PREM NBCAR =0
```

TRANSFORMATIONS A EFFECTUER DANS CCAR

```
PLIAGE?
?.
DEPLIAGE?
?.
```

MODULE TRANSFORME:

```
* CCAR

NBCAR = @NBCAR +1
PREM NBCAR =0
```



```
* CPARCOUR

MOT = @MOT /CAR
FINMOT = (CAR=' ')
PREM MOT =VIDE
```

TRANSFORMATIONS A EFFECTUER DANS CPARCOUR

```
PLIAGE?
?.
DEPLIAGE?
?.
```

MODULE TRANSFORME:

```
* CPARCOUR

MOT = @MOT /CAR
FINMOT = (CAR=' ')
PREM MOT =VIDE
```

ALGORITHME TRANSFORME:

```
* TELEGRAM

RESULTAT = ECRIRE LMESSAGE
LMESSAGE = ITARRET1 TELVIDE  *FUMESTEL

* FUMESTEL  % RESULTAT DE LA COMPOSITION DE CMESSAGE AVEC CTEL      %

MESSAGE = UNTEL,DER NBMOT ,DER NBLONG
UNTEL, NBMOT, NBLONG, TELVIDE = ITARRET1 FINTEL  *FUMESMOT

* FUMESMOT  % RESULTAT DE LA COMPOSITION DE FUMESNBR AVEC CMOT      %

UNMOT, NBMOT = SI MOT =´STOP´ OU MOT =´ZZZZ´  ALORS VIDE,@NBMOT  SINON  *NOUCMOTS
NBLONG = SI NBCAR <12  ALORS @NBLONG  SINON @NBLONG +1
NBCAR = DER SNBCAR
SNBCAR = ITGENER  *CCAR SUR  MOT
PREM NBMOT =0
PREM NBLONG =0
MOT = ITARRET2 FINMOT  *CPARCOUR SUR  SCARS
SCARS = SI FINBLOC  ALORS DONNEE  SINON <>
TELVIDE = FINTEL ET @FINTEL
PREM FINTEL =FAUX

* NOUCMOTS  % RESULTAT DE LA FUSION DE      NBMOT    AVEC CMOTSTOP  %

NBMOT = @NBMOT +1
UNMOT = SI NBCAR <12  ALORS MOT  SINON MOT[1,12]

* CCAR

NBCAR = @NBCAR +1
PREM NBCAR =0

* CPARCOUR

MOT = @MOT /CAR
FINMOT = (CAR=´ ´)
PREM MOT =VIDE

- END PASCAL
!
```


Remarque:

Dans l'algorithme transformé, nous constatons que la suite MOT n'a pas été supprimée après composition d'itérations définissant SNBCAR et MOT. De plus, le système n'a pas proposé cette modification, les préconditions nécessaires à sa réalisation n'étant pas vérifiées (en effet, MOT est utilisée par ailleurs dans une comparaison avec les mots spéciaux "STOP" et "ZZZZ").

II - PROLONGEMENTS DE CE TRAVAIL

II - 1 - Critiques

Le travail théorique présenté ici ne peut être considéré comme un travail fini. Nous sommes conscients d'un certain nombre d'imperfections, notamment en ce qui concerne la justification sémantique. Le travail fourni sur ce point doit être considéré comme une ébauche. Une amélioration peut être envisagée en développant plus systématiquement l'aspect fonctionnel, nous rapprochant ainsi des travaux actuels effectués par Françoise BELLEGARDE. L'aspect complétude n'a pas été abordé.

Un effort a été fait pour essayer d'améliorer les notations algorithmiques. Les choix retenus, tendant à nous rapprocher d'une notation fonctionnelle ne sont pas encore satisfaisants.

II - 2 - Prolongements au niveau réalisations pratiques

Au niveau réalisation pratique, notre but est d'obtenir un outil permettant de construire des algorithmes déductifs. Actuellement, il existe un compilateur admettant comme source un algorithme MEDEE, et générant un programme PASCAL. La syntaxe de ce langage est plus pauvre que celle introduite ici, elle n'admet pas, en particulier, des objets de type suite. Afin d'intégrer notre système de transformation dans cet embryon de système, il est nécessaire d'implémenter les extensions du langage que nous avons introduites. Il serait intéressant d'implémenter divers outils concernant les suites, par exemple :

- la gestion du parcours de deux ou plusieurs suites (opération que nous avons noté ⊠)
- la génération automatique de suites intermédiaires à partir de la description des ruptures.

Remarque: L'implémentation du type SUITE et des opérations associées pose le problème de la représentation de ces objets en liaison avec les supports physiques existants.

II-3- Divers prolongements théoriques et méthodologiques possibles

1 - Les ruptures

Il est important d'introduire dans la syntaxe des possibilités permettant de décrire une "suite à ruptures" sans avoir à la définir complétement dans l'algorithme.

Proposition (exemple des ateliers)

Afin d'éviter l'algorithme de construction des suites intermédiaires nécessaires à la définition du résultat, nous suggérons de décrire ces suites de la manière suivante :

Description des suites intermédiaires:

lateliers = suite (nua, lcatégories)

rupture lateliers = nua = 0

lcatégories = suite (nuc, lpersonnes)

rupture lcatégories = nua $\neq \omega$ nua

lpersonnes = suite (nom)

rupture lpersonnes = nuc $\neq \omega$ nuc

Description de la suite donnée:

ld = suite donnée (nua, nuc, nom)

2 - Les transformations proposées visent à améliorer l'efficacité de l'algorithme initial par réduction de l'espace mémoire occupé en supprimant le maximum de suites intermédiaires introduites.

Une autre démarche consisterait, à partir de la construction d'algorithmes proposée, d'améliorer l'efficacité par réduction du temps de calcul en utilisant des processus parallèles.(processus de types coroutines avec un producteur d'intermédiaires et un consommateur fournissant les résultats) ou en définissant un interprète ad hoc.

3 - Les réflexions émises ont essentiellement porté sur l'objet "SUITE" qui caractérise un ensemble totalement ordonné d'éléments. Un travail analogue pourrait être mené sur les "ENSEMBLES" sur lesquels on n'a pas la notion d'ordre.

4 - Dans la construction d'algorithmes proposée, nous introduisons des suites intermédiaires permettant de définir le résultat et dans un deuxième temps, nous définissons ces suites intermédiaires. Une autre façon de procéder consisterait à utiliser les théories de l'unification afin de générer automatiquement l'algorithme à partir d'une description précise des structures d'entrée et des structures de sorties.

III - CONCLUSION

L'extension du langage MEDEE existant nous a permis de généraliser l'utilisation de la méthode déductive à des problèmes plus concrets que ceux sur lesquels elle avait été validée dans les travaux précédents. La définition du résultat à partir d'une suite guide comme domaine nous a conduit naturellement à introduire des intermédiaires de type suite, eux-même spécifiés et définis à l'aide de la méthode déductive.

La réflexion méthodologique a consisté à :

- cerner le type de problèmes à résoudre. Nous nous sommes intéressés à des problèmes concrets, pas uniquement à la cinématique des fichiers mais à des problèmes plus généraux.
- proposer une solution qui comporte la définition d'un langage et d'une méthode.

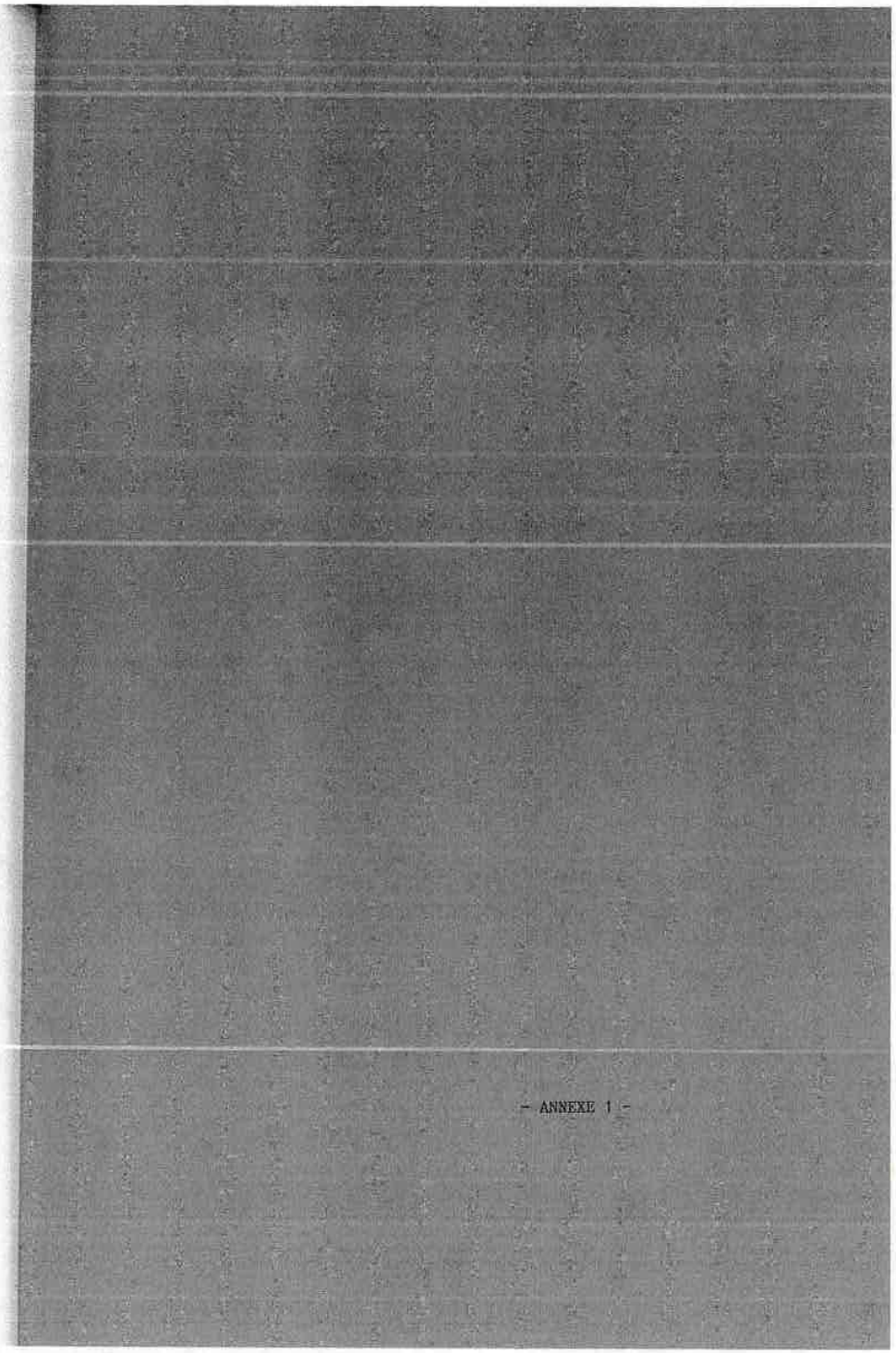
Ce langage, extension de MEDEE initial, est un langage à assignation unique dans lequel les suites et les opérations associées ont une place prépondérante.

La méthode de construction se décompose en deux étapes. La première consiste à définir un premier algorithme en utilisant la méthode déductive et en introduisant les intermédiaires adéquats au fur et à mesure de la construction, ces intermédiaires étant à leur tour définis de la même manière. Mais la mise en oeuvre de tels algorithmes ne se contente pas d'une traduction canonique dans un langage de programmation conventionnel : pour certains algorithmes, les intermédiaires de type suite introduits lors de l'analyse peuvent s'éliminer en appliquant certaines règles de transformations. La deuxième étape consiste donc à obtenir un algorithme final plus efficace par transformations successives.

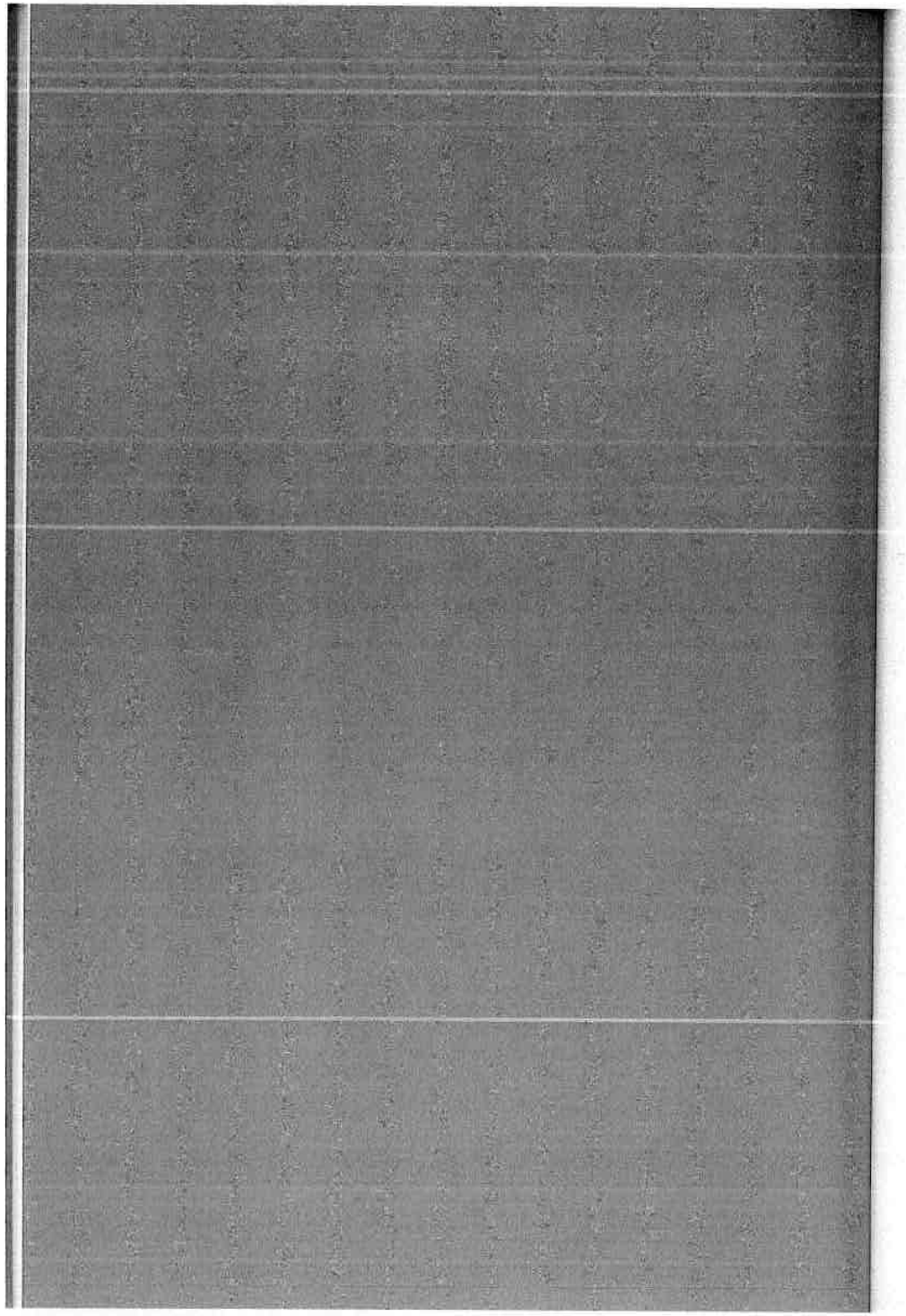
La méthode proposée nous a conduit à définir un ensemble de règles de transformations liées aux opérateursdu langage et à élaborer une stratégie d'application de ces règles. La définition formelle du langage à l'aide des types abstraits algébriques nous a permis de justifier ces règles.

La mise en oeuvre manuelle des règles de transformations est fastidieuse et source d'erreurs et nous a conduit à réaliser un système intéractif permettant une mise en oeuvre automatique de ces règles. Une maquette est actuellement utilisable.

Nous proposons dans ce travail, une généralisation de la méthode déductive et un outil de transformations d'algorithmes itératifs.



- ANNEXE 1 -



ANNEXE 1

Annexe 1 : grammaire décrite à l'aide de la notation de BACKUS :
(convention: ( ) *= nombre quelconque, éventuellement nul)

Syntaxe abstraite de langage MEDEE utilisé :

énoncé :: = (module)*

module :: = idm, (définition)*

définition :: = definfor, defformelle

definfor :: = (ident, idtype, commentaire)*

defformelle :: = f-espr | c-expr | i-expr

f-expr :: = expr | idm

c-expr :: = cond, f-expr, f-expr

i-expr :: = domaine, f-expr, [cond]

cond : = expr

domaine :: = indice, intervalle

intervalle :: = idsuite | constante, constante

idm :: = ident

indice :: = ident

idsuite :: = ident

ident :: = lettre, (chiffre | lettre)*

idtype :: = entier | réel | caractère | chaine | fichier | tableau | suite

commentaire :: = (ident, blanc)*

expr :: = (ident | séparateur)*

séparateur :: = + | - | * | / | = | < | > | : | (|) | ≠ | , | ' | ␣

blanc :: = ( )*

- BIBLIOGRAPHIE -

BIBLIOGRAPHIE

[ABR, 78] J.R. ABRIAL, Z : a specification language ,IFIP, Tokyo, 1978.

[AHO, 74] A.V. AHO, J.E. HOPCROFT, J.D. ULLMAN, the design and analysis of computer Algorithms, Addison - Wesley, 1974.

[ARS, 77] J. ARSAC, la construction des programmes structurés, Dunod 1977.

[ARS, 79] J. ARSAC, Y. KODRATOFF, Some methods for transformation of Recursives Procedures into Iterative ones, Rapport 79 - 2, LITP, Paris, 1979.

[ASH, 76] E.A. ASHCROFT, W.W. WADGE, Lucid : A Non procedural language with iterations, Comm. A.C.M., 20, 1977, p. 519 - 526.

[BAU, 78] F.L. BAUER, P.PEPPER, H. WOSSNER, Notes on the projet CIP : outline of a transformation system, in Program Construction, F.L. BAUER et M. BROY (cd.), Lecture notes in Computer Science 69, Springer Verlag, 1979.

[BEL, 78] F. BELLEGARDE, J.P. FINANCE, B. HUC, J. JARAY, P. LESCANNE, J. MAROLD, C. PAIR, A. QUERE, J.L. REMY, MEDEE : a type of language for the deductive programming method, conference on Reliable Software, German ACM chapter, Bonn, 1978.

[BEL, 78] F. BELLEGARDE, J.P. FINANCE, B. HUC, J.M. PIERREL, A. QUERE, J.L. REMY, Initiation à une construction méthodique de programmes Rapport CRIN 78 - E - 81.

[BEL, 78] F. BELLEGARDE, Recherche sémantique d'erreurs dans un programme construit déductivement, Journées d'études sur la fiabilité des programmes dans les applications industrielles, IRIA - EDEF - BIGRE - Chapitre français de l'ACM, Clamart, 1978.

[BEL, 79] F. BELLEGARDE, A. QUERE, J.L. REMY, Construction et transformation systématiques de programmes, Rapport CRIN 79 - R - 045, 1979.

[BEL, 81] F. BELLEGARDE, Un exemple d'application des techniques de programmation fonctionnelle à la transformation des programmes, Rapport CRIN, 81 - R - 009, Nancy.

[BUR, 77] R.M. BURSTALL, J. DARLINGTON, A transformation system for developing recursive programs, ACM, vol. 24, n°1, 1977.

[CLI, 73] M. CLINT, Program proving : Coroutines, Acta Informatica 2, 1973, p. 50 - 63.

[CON, 63] CONSWAY, Design of a separable transition diagram compiler, comm. A.C.M. 6, p; 396-400, 1963.

[DAR, 78] J. DARLINGTON, M. FEATHER,a transformational approach to modification, Internal Report, 1979.

[DIJ, 76] E.W. DIJKSTRA,A discipline of programming, Prentice Hall, 1976.

[FEA, 79] M.S. FEATHER, A system for developing programs by transformation, Ph. D. Theses, University of Edinburgh, 1979.

[FIN, 76] J.P. FINANCE, Une formalisation de la sémantique des langages de programmation, RAIRO Informatique Théorique, Vol. 10, n°8, p. 5 - 32 et Vol. 10, n°10, p. 5- 21, 1976.

[FIN, 78] J.P. FINANCE, De la spécification Abstraite d'une donnée à sa représentation en mémoire : les états successifs d'une Information, Actes du congrès AFCET TTI, éditions Hommes et Techniques, Tome 1, 1978.

[FIN, 79] J.P. FINANCE, B. HUC, P. LESCANNE, C.PAIR, M. QUERE, J.P. REMY, Programmation déductive et structures de données, Rapport CRIN, 79 - E - 051, 1979.

[FIN, 79] J.P. FINANCE, A. QUERE, Vers une systématisation raisonnable de la construction d'un programme.

[FIN, 79] J.P. FINANCE, Etude de la construction des programmes : méthodes et langages de spécification et de résolution de problèmes, Thèse d'Etat, Nancy, 1979.

[FLO, 67] R.W. FLOYD, Assigning Meaning to programs, Proceedings of the Symposium in Applied Mathematics, Mathematical Aspect of Computer Science 19, J.T. SCHARZ (ed.), American Mathematical Society, 1967, p. 19 - 32.

[GAU, 78] M.C. GAUDEL, C. PAIR, Structures de données et algorithmes fondamentaux, IRIA, 1978.

[GUI, 80] G. GUIHO, C. GRESSE, M. BIDOIT, Conception et certification de programmes à partir d'une décomposition par les données, RAIRO Informatique, Revue bleue, 1980.

[GUT, 77] J.V. GUTTAG, Abstract data types and the developpment of data structures, Comm. ACM, 20, 1977.

[GUY, 80] J. GUYARD, P. LESCANNE, L'utilisation d'un système d'édition et de manipulation comme support d'une méthode de programmation, Actes du congrès AFCET TTI, Nancy, nov. 1980.

[HEN, 72] P. HENDERSON, R. SNOWDON, An experiment in structured programming, BIT 12 p.38-53, 1972 .

[HER, 74] N. HERTSCHUM, L'analyse dans le projet CIVA, Thèse de spécialité de mathématiques appliquées, Univ. de Nancy I, 1974.

[HER, 76] N. HERTSCHUH, La cinématique automatique des fichiers, RAIRO Informatique, Revue bleue, 1976.

[HOA, 69] C.A.R. HOARE, An axiomatic basis of computer programming, Comm. ACM 12, 10, 1969, p. 576 - 580 et 583.

[HUC, 77] B. HUC, Mise en oeuvre de la méthode de programmation déductive, Thèse de docteur Ingénieur, Univ. de Nancy I, 1977.

[HUG, 79] J.W. HUGHES, A formalisation and explication of the Michael Jackson method of program design, Software - Practice and experience, Vol. 9, 1979, p. 191 - 202.

[JAC, 75] M.A. JACKSON, Principles of program Design, Academic press, 1975.

[JAC, 78] M.A. JACKSON, Information systems : modelling, sequencing and transformations, 3d conference on Software engeenering, 1978.

[JOU, 78] J.P. JOUANNAUD, Y. KODRATOFF, Synthèse Automatique de programmes à partir d'exemples, Journées IRIA - SESORI sur la synthèse, la manipulation et la transformation de programmes, St Rémy de Provence, 1978.

[KEN, 75] K. KENNEDY, J. SCHWARTZ, An introduction to the set theoritic language SELT, computers & mathematics with application, 1, 97, 1975.

[KOL, 79] E. KOLMAYER, Developpement et évaluation d'une méthode de programmation : deux évaluations de la méthode déductive de programmation, Rapport CRIN 79 - R - 074, 1979.

[LIS, 75] B.H. LISKOV, An introduction to CLU, in New Directions in algorithmic languages, S.A. Schuman (ed.), IRIA, 1975.

[LIV, 78] C. LIVERCY, Théorie des programmes, DUNOD, 1978.

[LUC, 77] M. LUCAS, P.C. SCHOLL, J. VOIRON, Apprentissage et utilisation du traitement séquentiel pour la construction de programmes, Rapport de recherche RR 74, laboratoire IMAG - USMG, Grenoble, 1977.

[MAN, 77] Z. MANNA, R. WALDINGER, Synthesis : dreams ⟶ programs, Technical notes, 156, SRI, 1977.

[NIV, 75] M. NIVAT, On the interpretation of polyadic recursive schemes, Symp. Mathematica, 15, academic press, 1975.

[PAI, 77] C. PAIR, mise en évidence de l'ensemble de départ dans les itérations en programmation déductive, A tout CRIN 6, bulletin de liaison du CRIN, 1977.

[PAI, 77] C. PAIR, La construction des programmes, Rapport interne CRIN, 77 - R - 019, 1977.

[PAI, 78] C. PAIR, La programmation, de l'énoncé au programme, congrès AFCET TTI, éditions Hommes et Techniques, 1978.

[REM, 74] J.L. REMY, Structure d'information, formalisation des notions d'accès et de modification d'une donnée, thèse de spécialité, Univ. de Nancy I, 1974

[REM, 80] J.L. REMY, Construction, évaluation et amélioration systématique des structures de données, RAIRO Informatique théorique, 1980, Vol. 1.

[SCH, 78] P.C. SCHOLL, Le traitement séquentiel: une classe de problèmes et une méthode de construction de programmes, Actes du congrès AFCET TTI Informatique, Tome 1, nov. 1978.

[SCH, 79] P.C. SCHOLL, Vers une programmation systématique : étude de quelques méthodes, techniques et outils, thèse d'Etat, USMG et INPG, Grenoble, 1979.

[SOU, 80] J. SOUQUIERES, Méthode pour la construction d'algorithmes dans le cas de conflits de structures, Actes du congrès AFCET TTI, Nancy, Nov. 1980.

[SOU, 81] J. SOUQUIERES, Construction d'algorithmes itératifs par suppression de suites intermédiaires, Comm. école d'été AFCET, Thiès, Sénégal, juil. 1981.

[VUI, 74] J. VUILLEMIN, Syntaxe, sémantique et axiomatique d'un langage de programmation simple, Thèse d'Etat, Univ. de Paris VII, 1974.

[WAR, 71] J.D. WARNIER, Entrainement à la programmation, Tomes 1 et 2, ed. d'organisation, 1971.

[WAR, 76] J.D. WARNIER, Pratique de la construction d'un ensemble des données, ed. d'organisation, 1976.

[WAR, 76] J.D. WARNIER, La transformation des programmes, ed. d'organisation, 1976.

[WIR, 73] N. WIRTH, Systematic programming : an introduction, Prentice Hall, Series in Automatic Computation, 1973.

[WIR, 76] N. WIRTH, Algorithms + Data structures = Programs, Prentice Hall, series in Automatic computation, 1976.

[WIR, 77] N. WIRTH, Introduction à la programmation systématique, MASSON, 1977.

NOM DE L'ETUDIANT : *SOUQUIERES Jeanine*

NATURE DE LA THESE : *Doctorat Ingénieur en Informatique*

VU, APPROUVE

ET PERMIS D'IMPRIMER

NANCY, le - 1 MARS 1982

LE PRESIDENT DE L'UNIVERSITE DE NANCY I,
l' Administrateur Provisoire

UNIVERSITÉ DE NANCY I
R. MAINARD

- RESUME -

Afin de valider la méthode déductive sur des problèmes réels, nous proposons une extension du langage MEDEE existant en lui adjoignant des objets de type SUITE et des opérations associées. La méthode de construction d'algorithmes proposée repose alors sur l'introduction de suites intermédiaires permettant de décomposer le problème à résoudre en deux sous-problèmes indépendants. L'algorithme final est obtenu par transformations successives.

Cette phase de transformations est automatisable: une maquette du système de transformations est actuellement utilisable.

mots clés : algorithme - construction - itération - Médée - méthode de programmation déductive - sémantique - suite - système automatique - transformations d'algorithmes -