MASTER EN SCIENCES INFORMATIQUES

Étude de la méthode de programmation structurée de Jackson et tentative de formalisation

de Grady, Etienne

Award date:
1988

Awarding institution:
Universite de Namur

FACULTES UNIVERSITAIRES NOTRE-DAME DE LA PAIX - NAMUR

INSTITUT D'INFORMATIQUE

ETUDE DE LA METHODE DE PROGRAMMATION STRUCTUREE DE JACKSON ET TENTATIVE DE FORMALISATION

Etienne de GRADY

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

Promoteur :

Baudouin Le Charlier

Année académique 1987 - 1988

RUE GRANDGAGNAGE,21,B - 5000 NAMUR (BELGIUM)

RESUME

Nous présentons tout d'abord la méthode de programmation structurée de Jackson de manière rigoureuse. Les clés de cette méthode sont les définitions des entrées et des sorties d'un programme libellées sous-forme d'arbres ainsi que la reconnaissance de la correspondance entre les structures de données. Nous présentons ensuite une définition plus formelle des arbres et de la correspondance. Nous reprenons la description donnée par J. Hughes de la construction de la correspondance Entrée->Sortie dans la méthode de Programmation Structurée de Jackson (P.S.J.) et nous la transcrivons sous une forme plus adaptée et plus rigoureuse des **règles de déduction.** Nous espérons en retirer une meilleure compréhension de la méthode et des constructions analogues dans d'autres méthodes de développemement. Nous développons également une application permettant la construction automatique, pour certains énoncés, d'un programme sur base des structures de données.

ABSTRACT

First, we give a rigorous description of Jackson's Program Design Method. The keys to this method are the definition of the inputs and outputs of a program as labelled trees and the recognition of a correspondence between the two data structures. Then, we give a more formal definition of the trees and the correspondence. Starting from the description by J. Hughes of the construction of the Input-Output correspondence in Jackson Structured Programming (J.S.P.), we translate it into the more rigorous and better suited form of **deduction rules.** The expected benefits are a deeper understanding of this method and of similar constructions in other development methods. We also develop an application allowing automatic program construction based on data structures.

En cette fin d'année, je tiens tout d'abord à exprimer toute ma gratitude à Monsieur Baudouin Le Charlier, promoteur de ce mémoire. Sa disponibilité de tous les instants, la pertinence de ses remarques et ses judicieux conseils ont très largement contribué à la confection du présent travail.

Je tiens également à remercier le service M.A.I. de son acceuil lors de mon stage à la C.G.E.R. Je suis très reconnaissant à Mesdames Marie-Claire Verlaine et Reinhilde Van 'T Dack et à Monsieur Christian Graas pour leur collaboration. Leurs conseils ont constitué pour moi un apport unique et fort précieux.

Enfin, j'adresse mes plus vifs remerciements à mes parents qui m'ont permis d'entreprendre mes études et qui m'ont toujours soutenu dans les moments difficiles. Merci aussi à Tamara pour la correction d'une grande partie de ce travail.

Jackson présente les différents concepts de sa théorie de manière "intuitive" en se basant sur des exemples "simples". L'inconvénient de cette description est que les concepts sous-jacents restent assez vagues. Il nous est dès lors difficile de déterminer la portée exacte de la méthode et de délimiter la classe de problèmes auxquels ces concepts s'appliquent.
Pour tenter de remédier à cet inconvénient, nous exposerons de manière plus systématique les différents concepts, ce qui nous permettra de mieux identifier les domaines d'application et les problèmes de compréhension de la méthode lors de son utilisation.

Il nous a paru interessant dans la seconde partie d'étudier la méthode J.S.P. de manière plus formelle. Certaines personnes ont essayé de mieux comprendre les concepts de la méthode J.S.P. en réalisant des tentatives de formalisation. Nous pouvons citer : Hughes ("A formalization and explication of the Michael Jackson method of program design"), Durieux ("Ebauche de formalisation de la construction de Jackson-Hughes"), Enselme ("Automatic Program Synthesis from data structures"). Leurs buts étaient une meilleure compréhension de J.S.P. et une approche de l'automatisation (en tout ou en partie) de cette méthode. Notre étude formelle de la seconde partie fut réalisée en respectant ces deux objectifs.
Nous présenterons et critiquerons, pour débuter cette partie du travail, le formalisme de Hughes qui est une des premières études de ce type portant sur J.S.P. Les formalismes des autres auteurs sont des améliorations ou des compléments de la présentation de Hughes. Cette analyse critique nous permettra de proposer un nouveau formalisme qui se rapprochera de la solution de Durieux ("Transduction Rationnelle") mais de manière plus simple et plus compréhensible.

Enfin, sur base des règles de déduction de correspondance et des règles de construction (automatique) de programmes qui ont été dégagées de la seconde partie, une application a été développée, permettant de mettre en oeuvre et de vérifier les résultats obtenus dans l'analyse formelle de cette seconde partie. Ce programme recherche les correspondances formelles entre deux structures de données de Jackson et sur base d'une des

correspondances, construit le programme Pascal s'y rapportant. Une description de la spécification et de la justification de cette application est présentée dans la troisième partie de ce travail.

PREMIERE PARTIE

PRINCIPES DE LA PROGRAMMATION STRUCTUREE DE JACKSON

PARTIE 1 : PRINCIPES DE LA PROGRAMMATION STRUCTUREE DE JACKSON (P.S.J.)

1.1. Introduction.

L'idée essentielle de la programmation structurée est la décomposition progressive de la résolution d'un problème en sous-problèmes élémentaires. Cette démarche consiste à travailler par raffinements successifs, en partant d'une idée générale du traitement et, à le détailler et le décomposer de plus en plus finement.

La méthode de construction de programmes proposée par JACKSON est fondée sur cette approche progressive. JACKSON expose une classification des problèmes (problèmes de base, problèmes des conflits, problèmes d'identification, etc) et propose pour chacun d'entre-eux une méthode de résolution : méthode de base, méthode des conflits, "backtracking", etc.

Dans ce premier chapitre, nous présenterons tout d'abord les différents concepts de la méthode P.S.J. de base (paragraphe 1.2. structures et composants de base).
Les hypothèses de la méthode de base sont :

- un seul flux de données en entrée et en sortie;
- une correspondance entre les deux structures de données;
- déterminisme des conditions de sélection et d'itération;
- évaluation possible des conditions avec un seul enregistrement (courant) en mémoire centrale.

Ce paragraphe concernant la description de la méthode se réfère à [JACKSON, 75] mais également à d'autres auteurs ayant fait des études sur la méthode de Jackson ou sur certains concepts de la méthode : [MATHIEU, 86], [JAVEY, 86], [SOUQUIERE, 82], [VAN 'T DACK, 86]. Les études réalisées par ces différents auteurs permettront de présenter la méthode P.S.J. de manière la plus générale et la plus stricte.

Après la description du formalisme utilisé, nous nous préoccuperons plus particulièrement dans le paragraphe 1.3. (approche d'une structure de données "appropriée") du choix d'une bonne découpe des données c'est-à-dire, d'une découpe d'un "bon" programme. Pour terminer, nous aborderons dans le paragraphe 1.4. certaines limites et extensions possibles de la méthode de base.

Les limites se réfèrent aux hypothèses de base, et pour chacune d'entres elles, nous proposerons une extension : plusieurs flux d'entrée et de sortie, détermination des conditions de sélection et d'itération par un nombre (borné ou non-borné) de lecture d'enregistrement et le dernier cas envisagé sera celui où une correspondance des structures ne peut être établie (les conflits de structure).

1.2. Structures et composants de base.

1.2.1. Présentation succinte de la méthode.

L'idée essentielle de la méthode proposée par JACKSON repose sur la décomposition hiérarchique des structures de données imposées dans l'énoncé du problème et sur la recherche d'une correspondance entre ces diverses structures. La structure de données est un guide important pour créer la structure du programme et nous analyserons plus en détail dans le paragraphe 1.3.3. (structure logique et structure physique) le concept de structuration des données.

Dans un premier temps, les diverses structures (d'entrée et de sortie) imposées dans l'énoncé du problème, sont représentées de manière arborescente. Cette décomposition est assurée grâce à l'utilisation des trois composants de base : la Concaténation (ou traitement séquentiel), l'Itération et la Sélection.

La phase suivante consiste à décomposer les deux structures définies précédemment et à rechercher des parallèles entre les niveaux correspondants (en procédant de manière descendante).

Le squelette du programme est déterminé à partir de la structure élaborée après la comparaison des deux structures initiales.

1.2.2. Types de programmes considérés.

[MATHIEU, 86, chap 3.1.1.]

De manière générale, la méthode de Programmation Structurée de Jackson (P.S.J) s'applique à un certain type de programmes. Il s'agit de programmes recevant en entrée un ou plusieurs flux de données séquentiels.

Il est également possible de considérer la méthode P.S.J.

dans un environnement "ON LINE" [VAN 'T DACK, 86]. Ce type d'approche ne sera pas abordé dans cette étude.

Dans la présentation des concepts de base de la méthode (structures et composants), nous nous limiterons à un seul flux d'entrée et un seul flux de sortie (l'extension à plusieurs flux d'entrée et/ou de sortie sera exposée au paragraphe 1.4.1.).

1.2.3. Représentation des programmes

[MATHIEU, 86, chap 3.1.2.]

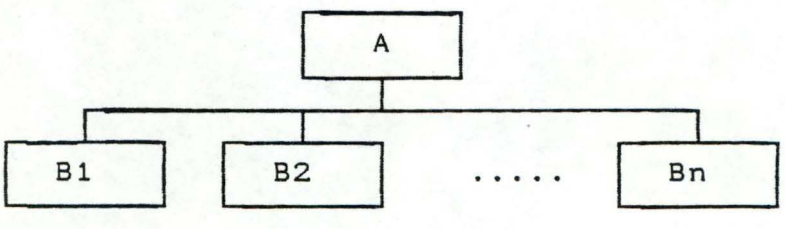
Le squelette du programme sera représenté par un arbre constitué à partir des quatre types de composants suivants :

a. Composant-Séquence :

un composant-séquence représente un morceau de programme dont l'exécution correspond à l'exécution successive d'un ou plusieurs autres morceaux de programme. Ces morceaux de programme seront représentés eux-mêmes par des composants que nous appellerons sous-composants-séquence du composant considéré.

Un composant-séquence est schématisé par un rectangle à l'intérieur duquel est inscrit son nom et en dessous duquel on schématisera les sous-composants.

Le schéma :



signifie que l'exécution du composant A est constituée des exécutions successives des composants B1, B2, ... Bn (dans cet ordre).

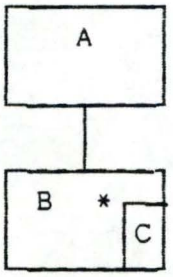
b. Composant-Itération :

Un composant-itération représente un morceau de programme dont l'exécution correspond à 0, une, ou plusieurs exécutions d'un même morceau de programme.

Ce morceau de programme sera représenté par un composant que nous appellerons sous-composant-itération. Un composant-

itération est schématisé par un rectangle à l'intérieur duquel on inscrira son nom et en dessous duquel on schématisera le sous-composant par un rectangle comprenant son nom, un astérisque et une condition d'itération.

Le schéma :



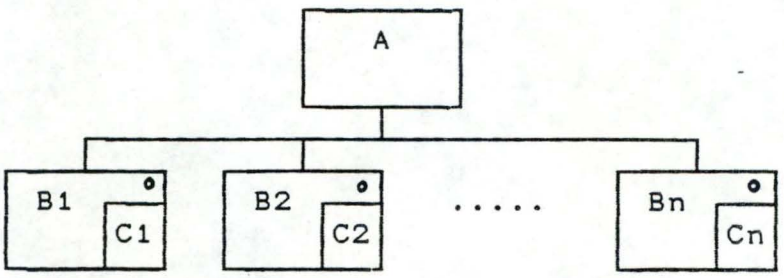
signifie : l'exécution de A consiste à exécuter B tant que la condition C est vraie.

c. Composant-Sélection :

Un composant-sélection représente un morceau de programme dont l'exécution correspond à celle d'un même morceau de programme choisi parmi plusieurs selon la valeur d'une condition.

Les différents morceaux de programme pouvant être choisis seront représentés par un rectangle appelé sous-composant-sélection du composant considéré. Chacun de ces rectangles contiendra le nom du sous-composant considéré. Chacun de ces rectangles contiendra le nom du sous-composant, le signe "O" et la condition d'exécution du sous-composant.

Le schéma :



signifie : l'exécution de A consiste à exécuter le composant B_i tel que la condition C_i est vraie.

(remarque : ceci suppose qu'une et une seule des conditions C_i soit vraie).

d. Action Primitive :

Un composant action primitive représente un morceau de programme "suffisamment simple" pour qu'on ne juge pas utile de le décomposer en parties plus élémentaires. Il est schématisé par une feuille de l'arbre représentant le programme.

Remarque : chaque composant constituant l'arbre d'un programme est sous-composant d'un et un seul autre composant, exception faite de la racine de l'arbre qui représente le programme tout entier.

Les trois premiers composants sont considérés comme des composants composites. Ces types de composants font eux-mêmes l'objet d'une décomposition plus raffinée. Tandis que le composant élémentaire ne sera pas disséqué plus en détail.

La structure hiérarchique est très puisssante parce qu'elle permet de créer des programmes "complexes" sans utiliser de composants "complexes". La complexité et la dimension du programme est reflété seulement dans le nombre de composants et le nombre de niveaux dans la hiérarchie.

1.2.4. Structuration des données.

[MATHIEU, 86, chap 3.1.3.]

1.2.4.1. Introduction

Une des étapes de la méthode P.S.J. consiste à définir la structure des données en entrée et en sortie. C'est sur base de ces structures que l'on déduira celle du programme. Le problème inhérent à cette étape est qu'il existe plusieurs manières de structurer les données. Toutes ces façons ne sont pas équivalentes dans la mesure où l'une peut amener à construire un programme relativement complexe et incompréhensible tandis que l'autre conduira à un programme nettement plus simple et compréhensible.

Parmi les différentes structurations de données pouvant se présenter on cherchera à découvrir celle qui "reflète le mieux la structure du problème à résoudre". Nous illustrerons par un exemple ce type de difficulté après avoir présenté le symbolisme de structuration des données (une étude plus approfondie concernant l'approche d'une "bonne" structuration des données

est faite dans le paragraphe 1.3.).

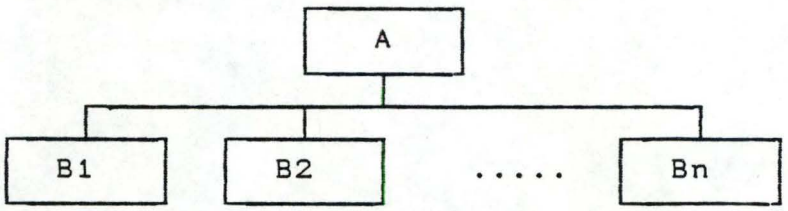
1.2.4.2. types de composants.

Un flux de données sera représenté (tout comme le programme) par un arbre. cet arbre est constitué de composants d'un des quatre types mentionnés ci-dessous.

a. Composant-Séquence :

Un composant-séquence représente une classe de flux de données obtenus par concaténation de flux de données appartenant à plusieurs autres classes. Le symbolisme utilisé est le même que dans le cas des programmes.

Donc, le schéma :



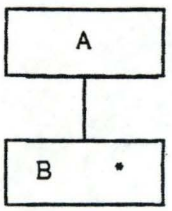
signifie qu'un flux de données de la classe A est formé par concaténation de n flux de données appartenant aux classe B1, B2,... Bn respectivement.

b. Composant-Itération :

Un composant-itération représente une classe de flux de données obtenus par concaténation de 0, 1, ou plusieurs flux de données appartenant à une même classe.

Le symbole utilisé est le même que le cas des programmes.

Donc, le schéma :



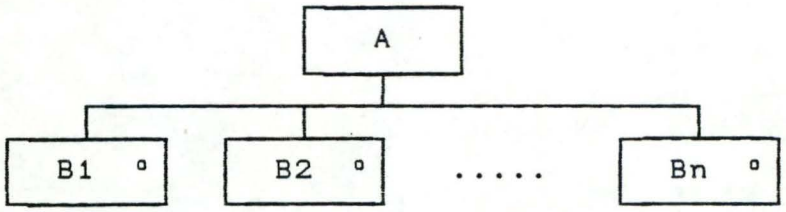
Signifie : un flux de données de la classe A est formé par concaténation de 0, un ou plusieurs flux de données appartenant à la classe B.

c. Composant-Sélection :
.....

Un composant-sélection représente une classe de flux de données égale à la réunion d'un ensemble d'autres classes de flux de données.

Le symbolisme utilisé est le même que dans le cas des programmes.

Donc, le schéma :



signifie : un flux de données de la classe A est un flux de données appartenant à une des classes B1, B2, ... Bn.

d. Composant Elémentaire :
.....

Un composant élémentaire représente une classe de données élémentaires (flux réduits à un élément ou vides). Il est schématisé par une feuille de l'arbre représentant le flux de données.

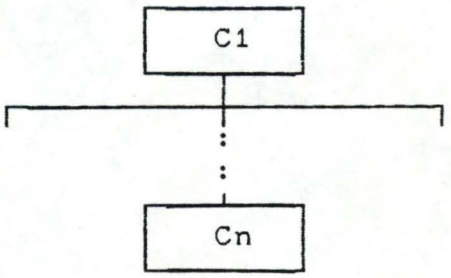
1.2.5. Définition de la notion de correspondance.

[MATHIEU, 86, chap 3.1.4.]

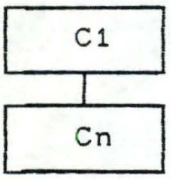
Une des étapes dans la construction d'un programme selon la méthode de Jackson consiste à rechercher une correspondance entre les structures des entrées et des sorties. Nous expliquerons ci-dessous ce qu'il faut entendre par cette notion de correspondance. Au préalable, nous définirons un certain nombre de concepts nécessaires à la définition du terme "correspondance".

- Le concept d'arbre devra être pris dans le sens d'arbre construit à partir des quatre types de composants vus précédemment.

- Un arbre A est un élagage d'un arbre B ssi A peut être obtenu
  1) en supprimant des feuilles ou des sous-arbres de B,
  2) en remplaçant certains chemins de B.



par des arcs :



- Un arbre A peut être mis en correspondance formelle avec un arbre B ssi (par définition) il existe un arbre C tel que A et B sont tous deux des élagages de C.
  On peut choisir C de telle sorte que tout composant de C se retrouve au moins dans A ou B. On choisira toujours C de cette sorte en pratique (On l'appellera fusion de A et B, selon la correspondance choisie).
  Dans une telle correspondance formelle, on dira qu'un composant de A est en correspondance avec un composant de B s'ils sont tous deux obtenus par élagage à partir du même composant de C.

- soient A l'arbre représentant la structure des entrées d'un programme et B l'arbre représentant celle des sorties.
  On appellera CORRESPONDANCE entre A et B, une correspondance formelle entre A et B vérifiant les conditions suivantes :
  1) pour tout couple C_A, C_B de composants de A et B en correspondance, le flux d'entrée contient toujours autant d'occurrences de C_A que le flux de sortie en contient de C_B.

2) si n est ce nombre d'occurrences, $\forall$ i : $1 \leq i \leq n$, la ième occurrence de C_B peut être générée (obtenue) à partir de la ième occurrence de C_A.

1.2.6. Construction du Programme.

[MATHIEU, 86, chap 3.1.5.]

Dans la méthode de Jackson, la conception d'un programme se déroule en 4 étapes:

1. On spécifie le problème à traiter;
2. On donne une description des données en entrée et en sortie; on définit la structure des entrées et des sorties par deux arbres (soient A et B respectivement);
3. On recherche une correspondance entre A et B comme indiqué au paragraphe 1.2.5. Soit C la fusion de A et B pour cette correspondance;
4. On complète l'arbre C du programme
   - en ajoutant des conditions aux sous-composants-itération et aux sous-composants-condition de C;
   - en ajoutant des composants actions primitives.(1)

La définition de cette 4ième étape est particulièrement peu précise. En fait, cette imprécision reflète la difficulté qu'il y a à donner des règles pour déterminer les conditions et actions-primitives à ajouter à la structure de programme.
Nous donnerons au paragraphe 1.2.7. quelques ébauches de telles règles ; malheureusement elles restent encore bien trop vagues.

Néanmoins, quelques précisions au sujet des actions-primitives peuvent être apportées, dans les cas où les flux sont des fichiers séquentiels.

1. Chaque action-primitive se présente dans l'une des classes suivantes :
   - Opérations d'ouverture et de fermeture de fichier;
   - Opérations d'écriture {WRITE(f,e)};
   - Opérations arithmétiques;
   - Opérations de lecture {READ(f,e)};
   - Opérations d'affectation.

---

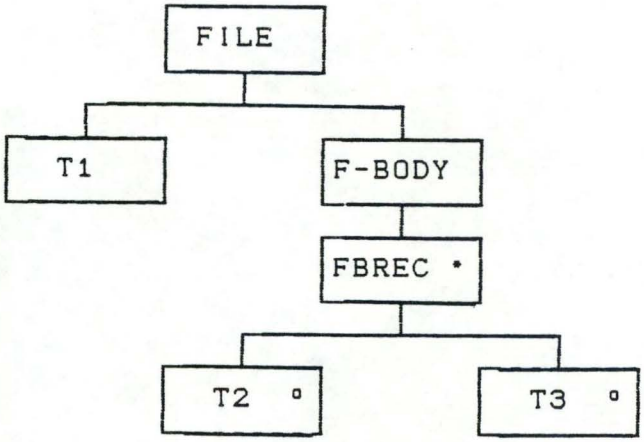
(1). Rem. : L'ajout des actions primitives entraine parfois la création de niveaux supplémentaires dans l'arbre (Composants séquences).

2. Nous utiliserons la technique de "lecture préalable" pour traiter les fichiers séquentiels d'entrée : avant le début de chaque exécution d'un composant du programme, le premier élément du flux à traiter par ce composant a déjà été lu et placé dans un "buffer".

3. La technique de la lecture préalable s'applique mieux au langage Cobol qu'au langage Pascal dû à l'implémentation différente de l'instruction de lecture d'un fichier. L'instruction Cobol "READ nom-fichier RECORD AT END instruction-impérative." a pour effet principal (sauf lorsqu'elle échoue) de placer dans la zône de mémoire tampon accessible au programme un nouvel article logique provenant du fichier désigné. Lorsqu'elle échoue et se termine par une condition d'erreur ou d'exception, le contenu de la mémoire tampon est indéfini. Par contre, en Pascal, l'instruction "READ(file,var)" a pour effet de ranger dans 'var' le contenu du composant de 'file' sur lequel on pointe et avance le pointeur d'un élément. Si ce composant est le dernier du fichier, la fonction EOF(file) prend la valeur "true". Cette implémentation Pascal entraine certains problèmes pour la technique de "la lecture préalable" lorsque nous sommes en présence d'un fichier vide ou du dernier élément du fichier. Néanmoins, préférant la lisibilité, la concision, la clarté de la structure du langage Pascal, nous présenterons les différents exemples dans ce langage. Pour remédier à la différence de l'opération de lecture, nous proposons une nouvelle instruction Pascal se rapprochant des effets du "READ" Cobol :

```
Procedure PREAD(f,e);
if not eof(f) then read(f,e)
              else ff := true;

{ff : fin de fichier}
```

Exemple : Considérons la structure suivante d'un programme :



dont le programme correspondant, en utilisant la technique de "lecture préalable", est :

```
Program FILE(input,output);

var f : file of T;  {fichier d'entrée}
    e : T;
    ff : boolean;

begin
   reset(f);
   ff := false;
   pread(f,e);
   procedure_T1;
   while (not ff) do
   begin
      if (e = TYPE_T2)
        then procedure_T2
        else procedure_T3;
   end;
   close(f)
end.
```

Chaque "procedure_Ti" traite un composant Ti et lit le premier enregistrement suivant si il existe, sinon, la marque de fin de fichier 'ff' est mis à "true";

La technique de "lecture préalable" assure que les données nécessaires sont disponibles en mémoire centrale pour effectuer l'évaluation des conditions de sélection ou d'itération.

Remarque : - nous supposons également que la lecture d'un seul enregistrement est suffisante pour la détermination des conditions de sélection ou d'itération. (nous analyserons au paragraphe 1.4.2. comment cette règle peut être généralisée à plusieurs lectures préalables).

1.2.7. Détermination des conditions et actions primitives. (tentative)

[MATHIEU, 86, chap 3.1.6.]

Soit :
- A : Structure de l'arbre d'entrée;
- B : Structure de l'arbre de sortie;
- C : Structure de la correspondance des arbres A et B (fusion de A et de B) (cfr. paragraphe 1.2.5.)

La détermination des conditions et actions primitives se fait en se basant sur le type des composants de C et sur les correspondances entre les composants de A et de B.

Tout composant de C est (représente) un morceau de programme qui traite en entrée une partie (peut-être vide) du flux d'entrée et génère en sortie une partie (peut-être vide) du flux de sortie. Ceci peut être précisé en se basant sur les correspondances.

Soit C_o le composant considéré de C. Trois cas principaux sont à considérer :

1) C_o correspond à deux composants de A et B mis en correspondance.
2) C_o correspond à un composant de A mais pas de B.
3) C_o correspond à un composant de B mais pas de A.

Remarque :

Dans le premier cas, une exécution de C_o, traitera effectivement une partie du flux d'entrée et générera la partie correspondante du flux de sortie. C'est le cas "normal".

Dans le deuxième cas, il y a lecture sans génération. Cela signifie (en général) qu'une partie de l'information contenue dans le flux d'entrée sera mémorisée pour être générée plus tard (cas particulier : partie inutile du flux d'entrée).

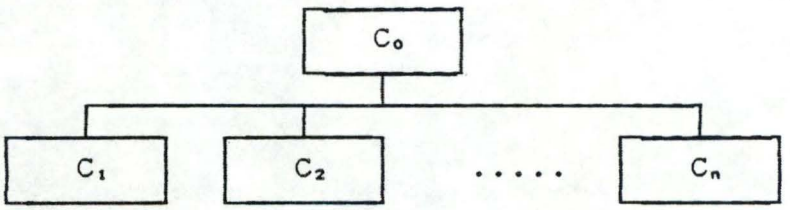
Dans le troisième cas, il y a génération sans lecture, c'est le cas inverse du précédent : une partie du flux de sortie est généré, non à partir du flux d'entrée mais d'une information mémorisée précédemment. (fin de la remarque).

Pour justifier l'applicabilité de la méthode, il faudrait étudier tous les cas pour montrer qu'il est chaque fois possible de construire le programme en complétant l'arbre C par des conditions et des actions primitives. Nous envisagerons

seulement le premier cas pour simplifier. Remarquons cependant que l'abondance des cas possibles montre que la méthode ne peut être appliquée sans une bonne dose d'imagination.

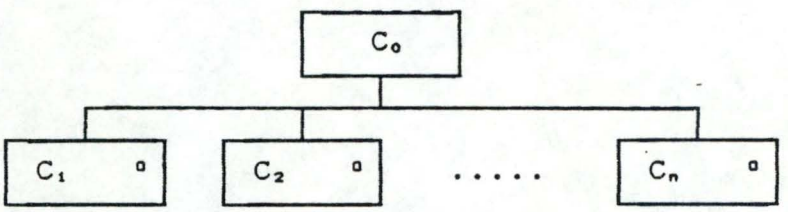
Construction d'un composant de programme correspondant à un composant de flux d'entrée et à un composant du flux de sortie.

Cas 1 : **Le composant est un composant séquence**



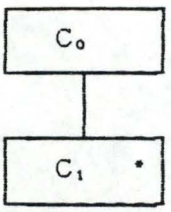
On suppose que C_0, C_1, C_2, ..., C_n correspondent à des composants à la fois des flux d'entrée et de sortie. Alors, une occurrence de C_0 dans le flux d'entrée se compose d'une occurrence de C_1, suivie d'une occurrence de C_2, etc. L'exécution des composants C_1, suivie de celle de C_2, etc, générera l'occurrence correspondante de C_1 (dans le flux de sortie), suivie de celle de C_2, etc. Le tout formera bien l'occurrence de C_0 dans le flux de sortie correspondant à celle de C_0 dans le flux d'entrée. Ceci démontre que le schéma ci-dessus est correct par rapport à sa spécification.

Cas 2 : **Le composant est un composant condition**



Il suffit d'ajouter aux composant C_1, ..., C_2 une condition permettant de savoir si le flux d'entrée appartient à la classe C_1 ou à la classe C_2, ..., ou à la classe C_n. La manière dont ces conditions seront réalisées dépend de la nature du flux d'entrée.

Cas 3 : **Le composant est un composant itération**



Une occurrence du flux d'entrée de la classe C_0 est formé de n occurrences de la classe C_1. Il suffit d'exécuter n fois le composant C_1 pour générer les n occurrences des flux de sortie de la classe C_1 composant le flux de sortie de C_0. Pour cela on cherchera une condition permettant de "détecter" la fin de la dernière occurrence du flux d'entrée de la classe C_1.

Cas 4 : **Le composant est un composant "élémentaire"**

Une exécution du composant C_0 doit traiter un élément du flux d'entrée et générer un élément du flux de sortie. Le composant C_0 contiendra un ensemble d'actions primitives permettant de réaliser ce traitement.

Conclusion

La "méthode" pour déterminer les actions et les conditions vue ci-dessus, est trop vague pour pouvoir être qualifiée de systématique. Il est difficile d'être plus précis parce que selon le problème traité, les flux d'entrée et de sortie peuvent prendre des formes très différentes. La méthode est "relativement" systématique jusqu'à la constitution de l'arbre fusionné des structures d'entrée et de sortie (étape 3 dans la construction du programme, paragraphe 1.2.6.).
Après cela, si l'on veut travailler proprement, il faut en fonction du problème, donner des spécifications précises aux différents composants de l'arbre du programme permettant alors de construire de manière exacte ces composants, selon le principe de la méthode descendante.
L'intérêt principal de la méthode de Jackson sera donc de suggérer une "bonne" découpe du problème basée sur la correspondance entre les entrées et les sorties.

1.2.8. Illustration.

[MATHIEU, 86, chap 3.1.7.]

Pour illustrer les différents concepts exposé dans cette section 1.2. (structures et composants de base), nous avons traité dans l'annexe 1 un exemple. Par cet exemple, nous montrerons que le flux de données en entrée peut être structuré de deux façons différentes (au moins). Nous insisterons sur le fait que, même si ces deux structurations amènerons à des programmes corrects, le choix d'une structuration "appropriée" des données amènera à construire un programme nettement plus simple et compréhensible que l'autre.

1.3. Approche d'une sélection d'une structure de données appropriée

1.3.1. Introduction

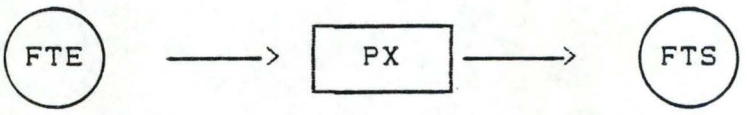
Nous avons présenté précédemment (paragraphe : 1.2.4. Structuration des données) un symbolisme nous permettant de structurer les données de manière arborescante. Nous avons laissé apparaître à travers l'exemple du "Stock Magasin" (Annexe 1) les difficultés relatives aux structures des données.
Nous avons montré que le flux de données en entrée ou en sortie peut être structuré de deux façons différentes (au moins) et qu'une structuration de données amènera à construire un programme nettement plus "simple" et "compréhensible" qu'une autre.

L'approche et les précisions concernant la structuration des données faites dans ce paragraphe sont rarement abordées dans la littérature. On nous présente un formalisme de structuration ainsi que les "bonnes" structures de données d'un problème sans en préciser réellement leur sens. Nous tenterons, dans le paragraphe 1.3.4 (Sélection d'une structure logique), de classer quelques principes dont la compréhension et l'application pourra assister le concepteur dans la sélection d'une structure "appropriée" permettant l'élaboration d'une structure de programme "efficace". Un des intérêts de l'étude formelle est de faire apparaître plus clairement les difficultés de structuration.

Nous ferons tout d'abord une distinction entre la structure physique et la structure logique des données (paragraphe 1.3.3.). Cette distinction nous permettra d'expliquer ce que l'on entend par "différentes manières de structurer les données". Nous présenterons également (paragraphe 1.3.2.) la spécification d'un problème servant de référence aux diverses illustrations de ce paragraphe 1.3.

1.3.2. Enoncé du problème de référence : Fichier Texte

On doit concevoir un programme qui, à partir d'un fichier de caractères (fichier d'entrée) composé d'une séquence de caractères alphabétiques et "blanc" (espaces), doit copier dans un fichier de sortie les caractères "non-blanc" et ignorer les "blanc".



FTE : Fichier Texte d'Entrée;
FTS : Fichier Texte de Sortie.

1.3.3. Structure physique et structure logique

A. Structure physique :

Une structure physique des données représente le format des enregistrements du fichier de données en entrée ou en sortie tel qu'il est défini dans les types du programme.

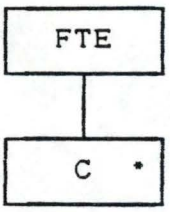
Dans notre exemple du "fichier texte", la structure physique des données en entrée est (dans le formalisme du langage PASCAL) :

```
FTE = file of char;
```

ou

```
FTE = text;
```

Nous pouvons représenter également cette structure physique dans le formalisme de JACKSON :



FTE : Fichier Text d'Entrée;
C : Caractère.

La structure physique pour un flux de données est unique. Elle nous est parfois imposée dans la spécification du problème, c'est-à-dire que cette structure physique des données est alors établie avant l'élaboration du programme (le fichier d'entrée ou de sortie sur lequel le programme opérera, existe déjà, est déjà structuré). C'est à partir de cette structure physique que nous devons trouver une "bonne" structure logique (cfr. point B.).

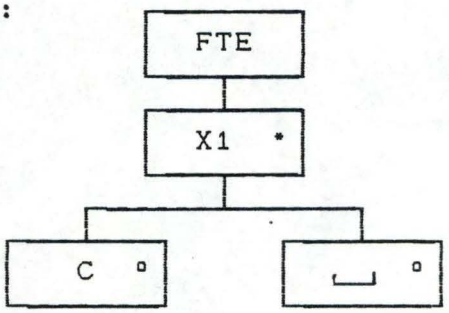
Dans d'autres cas, les structures physiques des fichiers d'entrée et de sortie n'existent pas encore lors de l'étape de structuration (logique) des données. Nous pouvons structurer logiquemment les données indépendamment de l'aspect physique et c'est à partir de la structure logique "appropriée" sélectionnée que l'on déterminera la structure physique la plus adéquate pour l'exécution du programme.

B. Structure logique :

Une structure logique d'un flux de données représente une manière de voir, de spécifier la structure physique d'une classe de données en fonction de certaines de ses propriétés que l'on veut dégager. Cette structure logique doit permettre de générer tous les flux de données de cette classe.

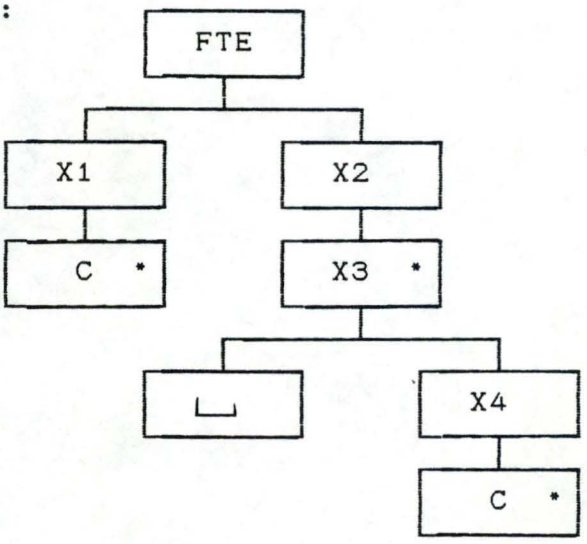
Voici quelques exemples par lesquels le "Fichier Texte d'Entrée" (FTE) peut être vu :

EX 1 :



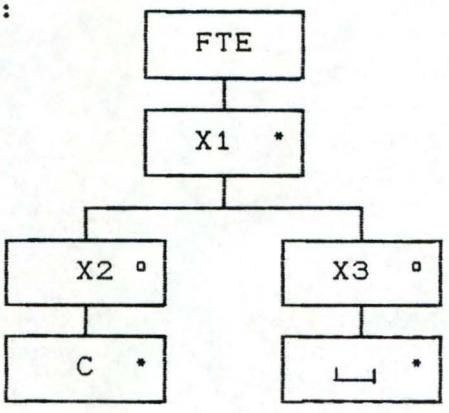
(FTE est composé d'un ensemble de caractères. Chaque caractère est soit un caractère alphabétique soit un caractère "blanc").

EX 2 :



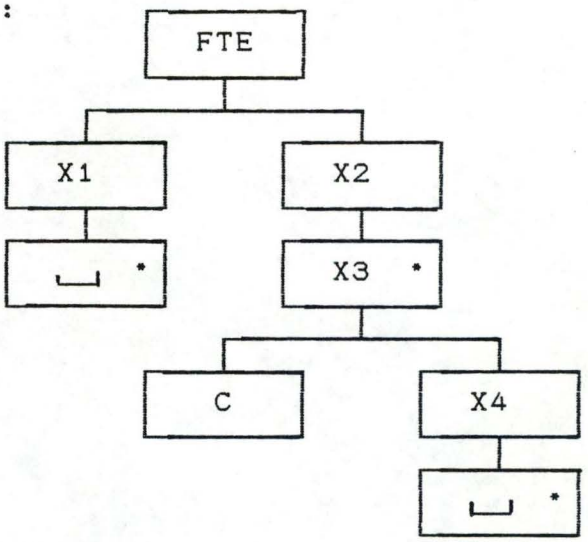
(FTE est composé de deux séquences d'éléments. Le premier élément est une suite de 0, 1 ou plusieurs caractères alphabétiques et le second élément est une suite de 0, 1 ou plusieurs groupes composés d'un "blanc" suivi de 0, 1 ou plusieurs caractères alphabétiques).

EX 3 :



(FTE est composé d'un ensemble (peut être vide) d'éléments. Un élément est composé soit de 0, 1 ou plusieurs caractères alphabétiques, soit de 0, 1 ou plusiers caractères "blanc").

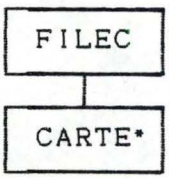
EX 4 :



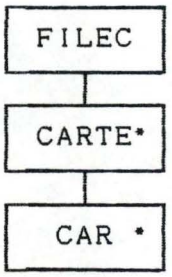
(FTE est composé de deux séquences d'éléments. Le premier élément est une suite de 0, 1 ou de plusieurs caractères "blanc" et le second élément est une suite de 0, 1 ou de plusieurs groupes composés d'un caractère alphabétique suivi de 0, 1 ou de plusieurs caractères "blanc").

Chacune de ces structures d'arbre est syntaxiquement correcte et représente une structure logique possible de la structure physique.

Remarque : la confusion entre la structure logique et la structure physique peut apporter certains problèmes dans l'élaboration d'un programme. Si l'on considère un fichier de CARTES (enregistrements) et chaque CARTE est composée d'une suite de caractères, la structure physique correspondante est :



Tandis que la structure logique peut être vue de la manière suivante :



où le composant "CARTE" n'apparaît plus comme étant l'enregistrement du fichier.
Les opérations de lecture affectées à la structure logique (étape 4 de l'élaboration d'un programme - paragraphe 1.2.6.) seront différentes suivant que l'enregistrement du fichier est une "CARTE" ou un "CARactère".

1.3.4. Sélection d'une structure logique

1.3.4.1. Introduction

Si un flux de données peut être structuré logiquement de différentes manières, ces différentes structures logiques dicterons des structures différentes du programme final.

L'étape de structuration des données représente un choix crucial entre toutes les structures logiques pour l'élaboration d'une structure d'un "bon" programme. Généralement, on associe à "bon programme" une liste de qualificatifs parmis 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.

Nous tenterons dans le paragraphe 1.3.4.2. de déterminer ce que l'on entend par "meilleur" programme et certains principes seront dégagés dans le paragraphe 1.3.4.3. permettant de guider le concepteur dans le choix d'une structure logique la plus "appropriée" et la plus "efficace" pour la conception du programme "simple" et "compréhensible".

1.3.4.2. Programme "simple" et "compréhensible"

Par la notion de programme "simple" et "compréhensible", nous voulons souligner la qualité de "traçabilité" d'un programme [VAN LANSWEERDE, 86], c'est-à-dire l'existence d'une correspondance directe entre la structure du programme et les spécifications fonctionnelles. Un changement dans la structure de données a une répercution sur la structure même du programme et toutes ces structures différentes auront un degré de traçabilité différent avec les spécifications.
Nous nous permettons d'énumérer certaines caractéristiques qui différencient un programme plus "simple" et plus "compréhensible" qu'un autre. Nous pouvons tout d'abord ajouter aux deux qualités "simple" et "compréhensible" d'un programme, la qualité d'un programme "clair" (qui reflète plus la structure du problème). La simplicité d'un programme se répercute sur le coût moindre dans l'utilisation de variables locales (externes aux flux de données du problème), en nombre d'enregistrements nécessaires pour la détermination des conditions d'itération ou de sélection, en nombre d'utilisation de fichiers intermédiaires, etc. La caractéristique suivante est déduite de la structure même du programme (de Jackson) et de la notion de clarté d'un programme : dans un programme "simple", tout flux constitutif peut être caractérisé à partir de ses composants.

Une deuxième qualité d'un programme "simple" et "compréhensible" que nous pouvons relever, est le degré de difficulté d'élaboration de la structure d'un programme à partir des structures des données; c'est-à-dire le degré de difficulté à trouver les correspondances (à l'étape 3) entre les deux structurations des données (élaborées à l'étape 2).

Pour illustrer cette notion de programme "simple", nous nous référons à l'exemple du "stock magasin" de l'annexe 1, où deux structures des données en entrée sont proposées. Ces deux structurations des données mènent à deux programmes différents. Les différences de clarté et de simplicité peuvent être déduites des spécifications des différents composants (TRAIT-GROUPE, TRAIT-CENTRE, TRAIT-ENTETE, TRAIT-ENREG et TRAIT-FIN).

Remarque : Nous ne visons pas dans la présentation d'un

programme "simple" la notion d'"éfficacité" et d'"optimisation" d'un programme, qui peut mener à une diminution du nombre d'opérations et du nombre de variables dans le programme, mais peut également mener à une certaine "incompréhensibilité" du programme.

1.3.4.3. Principes de sélection d'une structure logique

Bien qu'il n'existe aucune règle pour trouver une structure "appropriée" à un flux de données en entrée ou en sortie, nous définirons quelques principes permettant de faire un tel choix. Ces principes seront de deux types, les premiers proviennent de [JAVEY, 86], tandis que les seconds sont issus de [LE CHARLIER, 85].

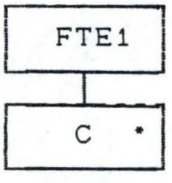
A. Premiers principes

Principe 1 : "La structure sélectionnée doit fournir une description des propriétés principales des séquences d'entrée et de sortie."

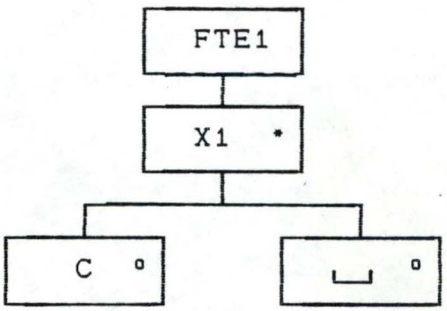
Ce principe souligne le fait que la structure logique sélectionnée ne tient compte que de certaines propriétés qui sont considérées comme principales par rapport aux autres et que le concepteur désire représenter dans sa structure. Cette structure logique ne doit pas représenter toutes les propriétés du flux de données, elle peut ne pas laisser apparaître certaines propriétés qui sont non-nécessaires à la structuration d'un "bon" programme.

Principe 2 : "La sélection d'une structure doit être influencée par le but de l'application."

Si dans notre exemple du fichier de caractères, le but de l'application est de compter le nombre de caractères dans le fichier texte d'entrée (FTE), alors il est préférable de voir le fichier comme une séquence de caractères :

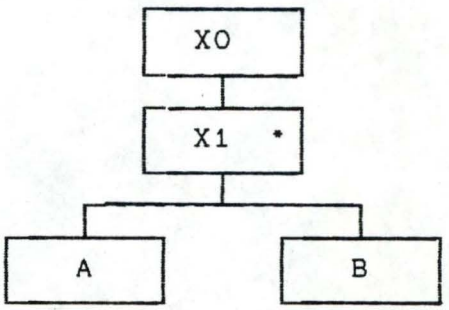


au lieu d'une séquence de caractères alphabétiques et blancs (espace) :



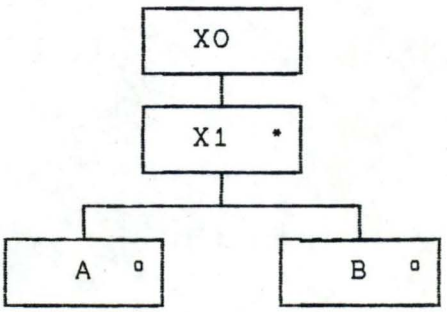
Le but de l'application nous permet de déterminer les propriétés des séquences qui doivent être considérées comme pertinentes et les propriétés qui peuvent être ignorées dans la structure des données.

Considérons une séquence dont la structure est :



(structure 1)

Si dans le but de l'application, la propriété "A doit être suivi par B" du composant-séquence [X1] n'a aucun intérêt, alors la structure logique la plus appropriée est :



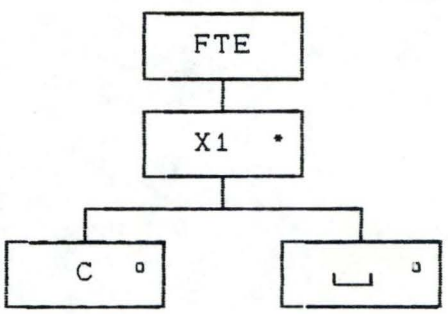
(structure 2)

Nous pouvons dire que la structure 2 est plus "générale" que la structure 1, c'est-à-dire que la classe de flux de données dénotée par la structure 2 (que nous appellerons structure générale) inclut la classe de flux de données dénotée par la structure 1 (que nous appellerons structure particulière).
Le flux de données {A,B,A,B,A,B} fait partie de la classe de flux de données de la structure 1 et de classe de flux de données de la structure 2. Tandis que le flux de données {A,B,B,A,A,A} ne fait partie que de la classe de flux de donées de la structure 2.

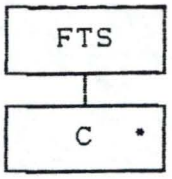
Conclusion :Suivant les deux principes de [JAVEY,86], une sélection d'une structure logique doit se porter sur une structure la plus "générale" et décrivant les propriétés "principales" pour réaliser le programme.

Exemple :

Reprenons l'énoncé du paragraphe 1.3.2. concernant le fichier de caractères. Nous avons vu que nous pouvons structurer logiquement les flux de données en entrée ou en sortie de différentes manières (paragraphe 1.3.3.). Nous montrons par cet exemple que l'application des principes de [JAVEY, 86] ne mène pas toujours à un choix "optimum".
Le but de l'application est d'ignorer les "blanc" (espaces) et de recopier les caractères alphabétiques. La structure logique la plus générale des entrées décrivant les propriétés principales est :

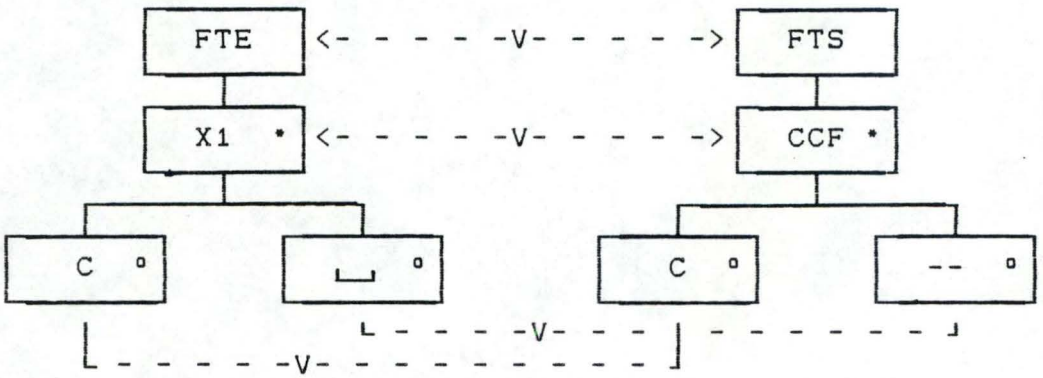


tandis que la structure logique des sorties est :

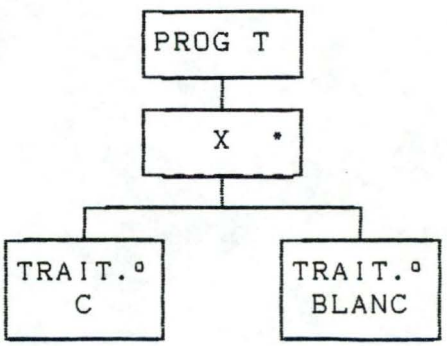


Le choix de la structure logique de données n'est pas "optimum" parce que nous sommes en présence de deux structures que l'on ne peut mettre en correspondance (suivant les règles définies dans le paragraphe 1.2.5). Le composant 'C' de FTE ne peut être mis en correspondance avec le composant 'X1' de FTS parce qu'il n'y a pas autant d'occurrences de 'C' de FTE que de 'X1' de FTS.

Pour résoudre ce problème il suffit de remplacer le composant 'C' de FTS par un composant condition 'fictif' (CCF), ayant le même nombre d'occurrences que le composant X1. CCF se décomposant en deux sous-composants-sélection dont les occurrences sont soit des caractères alphabétiques 'C', soit le composant 'vide'.
On obtient finalement l'arbre de sortie suivant pouvant être mis en correspondance avec l'arbre d'entrée.



En fusionnant les deux arbres, on obtient la structure suivante pour le programme :



Il serait exagéré de dire que cette structure de programme a réellement été obtenue en appliquant la méthode de JACKSON. Elle est obtenue en modifiant quelque peut la structure des sorties en se référant aux fonctions que le programme doit réaliser (Ce -> Cs; Blanc -> --).
Les principes de [JAVEY, 86] ne nous ont pas permis de découvrir la structure la plus adaptée des données pour la construction du programme. En présentant les deuxièmes types de principes, une structure logique plus appropriée sera proposée.

B. Deuxième principe

1. Les seconds types de principes permettant la sélection d'une structure "appropriée" des données pour la méthode de Jackson sont déduits par application de [LE CHARLIER, 85] (chap 5.5 : influence du choix, a priori, d'une méthode de raisonnement, sur la forme des programmes, lorsque ces méthodes sont utilisées au moment de la construction de ceux-ci. (p III/216)).

"Les méthodes de raisonnement (déjà présentées) ne pouront, en général, fournir l'intuition nécessaire pour découvrir le "principe" de la solution. Mais elles procureront un moyen de formuler avec précision les propriétés que devra vérifier le programme réalisant une certaine idée intuitive de solution, et, à partir de là, de construire ce programme sur une base plus sûre. On peut cependant noter que chaque méthode de raisonnement suggère, dans certaines catégories de problèmes, une manière "évidente" de généraliser l'énoncé initial fournissant immédiatement une idée de solution.
Nous appellerons ces généralisations les "heuristiques" associées à ces méthodes de raisonnement. Enfin, nous montrerons que les "heuristiques ascendantes" conduisent à raisonner sur la "structure" des données (au sens propre du terme) du programme et qu'au contraire, les "heuristiques descendantes" conduisent à raisonner sur la "structure" des résultats (ou, plus généralement, sur une "structure intermédiaire"). " [LECHARLIER, 85, p III/217].

Le raisonnement ascendant se base essentiellement sur ce qui a déjà été fait, après un certain nombre d'étapes (invariant), tandis que le raisonnement descendant se base sur ce qui reste à

faire (spécification d'un programme auxiliaire de ce qui reste à faire). [LECHARLIER, 85] a montré que le raisonnement ascendant revient à raisonner sur la structure des données et que le raisonnement descendant revient à raisonner sur la structure de sortie.

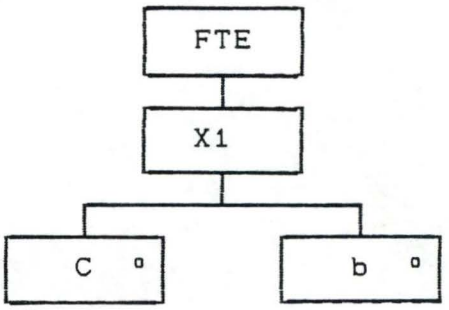
Nous allons montrer que le fait de raisonner de façon ascendante (structure de données en entrée) ou descendante (structure des données en sortie) conduit à construire des programmes de formes tout à fait différentes et que le fait de privilégier la structure des résultats par rapport à la structure des données en entrée (à la deuxième étape de la construction de programme de la méthode de Jackson) nous permettra de construire un programme plus "simple". Dans la méthode de Jackson, le fait de raisonner sur la structure des résultats fournit une certaine heuristique "aveugle" permettant, parfois, d'obtenir "mécaniquement" des idées de solutions. Cette heuristique consiste à structurer les données d'entrée en fonction des correspondances qui seront établies entre les structures d'entrée et de sortie (le fait de réfléchir sur l'arbre de sortie mènera à une correspondance plus naturelle).

2. Premier exemple : le programme d'élimination des caractères 'blanc' d'un fichier de caractères

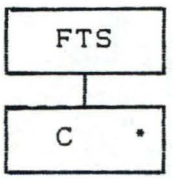
Ce programme présenté au paragraphe 1.2.3. sera tout d'abord construit en raisonnant sur la structure des données en entrée, ensuite, en raisonnant sur la structure des résultats.

a. Raisonnement ascendant

Si l'on considère la structure des données en entrée sans se référer à la structure des résultats, nous avons vu (paragraphe 1.2.4.3.A.) que la structure la plus "générale" est :

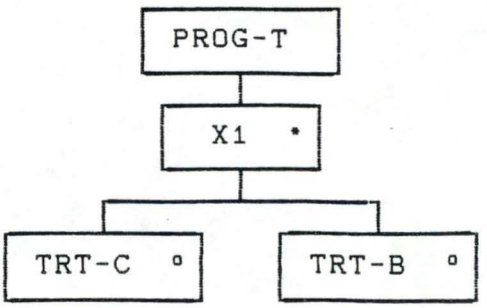


- Le fichier d'entrée est vu comme une suite de caractères et chaque caractère est soit un caractère alphabétique "C", soit un caractère 'blanc' "b" .
- Le fichier de sortie est vu comme une suite de caractères alphabétiques C :



Une correspondance entre ces deux structures ne peut être établie suivant les règles définies dans le paragraphe 1.2.5. Nous avons proposé dans le paragraphe 1.2.4.3.A une modification de la structure de sortie menant à une solution 'simple' mais ne correspondant pas exactement à l'application de la méthode de Jackson. Elle correspond plutôt à l'idée intuitive de la solution.

La structure du programme correspondant est :



Où TRT-C traite les caractères alphabétiques du fichier d'entrée FTE (recopie le caractère alphabétique dans le fichier FTS) et TRT-B traite les caractères 'blancs' du fichier FTE (ne recopie pas le caractère 'blanc' dans le fichier FTS).

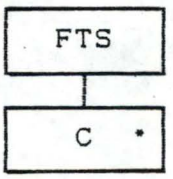
Le programme suivant correspondant en est déduit :

```
Programme TEXTE(input,output);

var  fce  : file of char;
     fcs  : file of char;
     c    : char        ;

begin
   reset(fce);
   rewrite(fcs);
   while (not eof(fce)) do
   begin
      read(fce,c);
      if (c <> ' ')
        then write(fcs,c);
   end;
   close(fce);
   close(fcs)
end.  {fin de TEXTE}
```

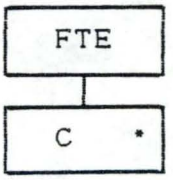
b. Raisonnement descendant :

Nous allons structurer les données en entrée sur base des correspondances à établir avec la structure des résultats. Outre la correspondance entre les deux composants racines des deux structures (FTE <---> FTS), nous devons trouver une correspondance entre les caractères alphabétiques des deux fichiers (C_* <---> C_*).

La structure des données en sortie se présente de la manière suivante :

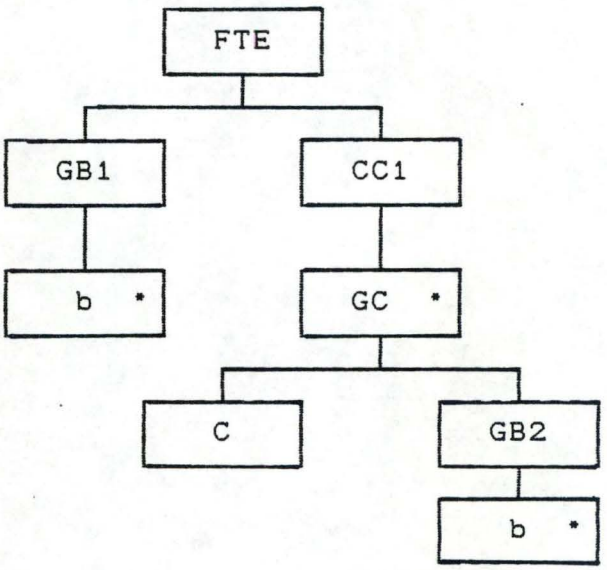


Nous considérons tout d'abord le fichier d'entrée comme ayant avant tout des caractères alphabétiques C. Chacun de ces caractères posséde un caractère correspondant dans le fichier de sortie :

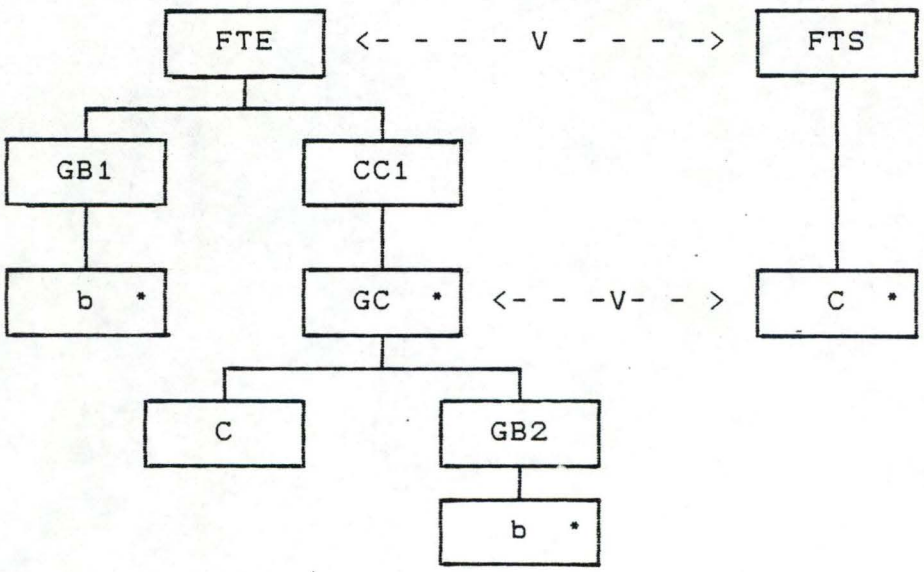


Chaque caractère peut être suivi de 0, 1 ou plusieurs 'blancs'.

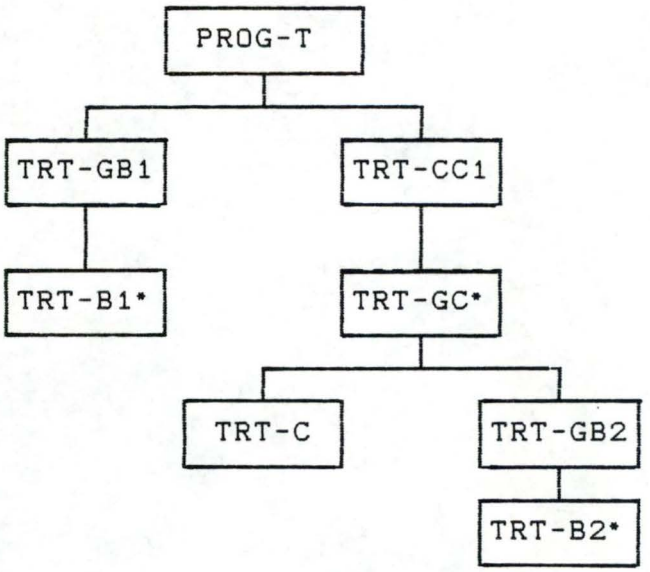
Chaque caractère suivi de ces 'blancs' (composant GC) est toujours en correspondance avec le caractère C de sortie. Le premier caractère alphabétique du fichier d'entrée peut être suivi de 0, 1 ou plusieurs 'blancs'. On obtient la structure des entrées correspondante :



La structure des données en entrée peut dés lors être mise en correspondance formelle (suivant le paragraphe 1.2.5.) avec la structure des sorties :



La structure du programme peut être déduite de cette correspondance de la manière suivante :



Dont le programme correspondant est :

```
Programme TEXTE(input,output);

var  fce  : file of char;
     fcs  : file of char;
     c    : char        ;
     ff   : boolean     ;

begin
   reset(fce);
   rewrite(fcs);
   pread(fce,c);     (1)
   while (not ff) and (c = ' ') do
        pread(fce,c);
   while (not ff) and (c <> ' ') do
   begin
      write(fcs,c);
      pread(fce,c);
      while (not ff) and (c = ' ') do
              pread(fce,c)
   end;
   close(fce);
   close(fcs)
end.  {fin de TEXTE}
```

c. Conclusion :

La seconde structuration des données mène à une correspondance plus naturelle et, la structuration et la construction du programme est rendue plus facile. Le premier raisonnement demande une adaptation des structures de données pour structurer le programme. Par contre le second raisonnement, permet de tenir compte des fonctions du programme lors de la

(1) Définition de l'instruction PREAD : page 13.

structuration des données (cfr. 2ième caractèristique d'un programme "simple"). Cet exemple simple ne nous a pas permis de mettre en évidence la première caractéristique d'un programme "simple".

3. deuxième exemple : enregistrement directeur

Ce second exemple est présenté dans l'annexe 2. Il illustre les caractéristiques de ce paragraphe, ainsi que le problème "d'identification" (paragraphe 1.4.2.). Nous voulons montrer dans cet exemple, que le raisonnement descendant appliqué à la méthode de Jackson a une influence sur les deux caractéristiques d'un programme plus simple (paragraphe 1.3.4.2.). Nous n'avons pu montrer la première caractéristique dans le premier exemple, car le programme élaboré se basait sur des structures de données assez simple.

1.4. Extensions possibles de la méthode de base

1.4.1. Cas de plusieurs flux d'entrée et de sortie

1.4.1.1. Introduction

Jusqu'à présent, nous avons exposé les différents concepts avec un seul flux de données en entrée et un seul flux de données en sortie. Il est possible de généraliser et d'étendre la méthode au cas de plusieurs flux d'entrée et de sortie.

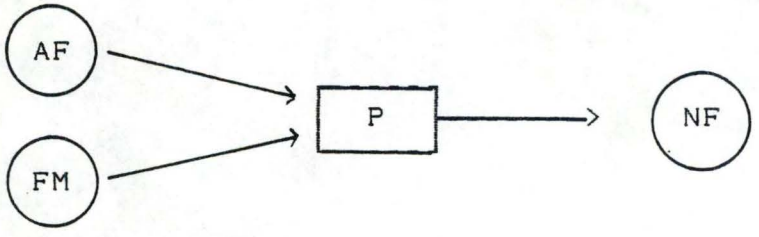
Une certaine classe de problèmes implique la recherche des correspondances de deux ou plusieurs fichiers séquentiels d'entrée qui doivent être traités simultanément suivant la valeur d'une clé. La généralisation de la méthode à plusieurs flux d'entrée consiste à représenter l'ensemble de ces fichiers par une seule structure ("virtuelle").

Remarque 1 : De tels problèmes sont souvent appelés problèmes de "mise en phase" ("Matching" ou "Collating").

Remarque 2 : Lorsque les fichiers d'entrée ou de sortie doivent être successivement traités, on ne parlera pas de problèmes de mise en phase; on obtient simplement dans la structure du programme, une séquence de composants qui, traiteront chacun totalement un des fichiers (ce cas entre aussi dans la catégorie du "flux virtuel").

1.4.1.2. Création d'un flux virtuel

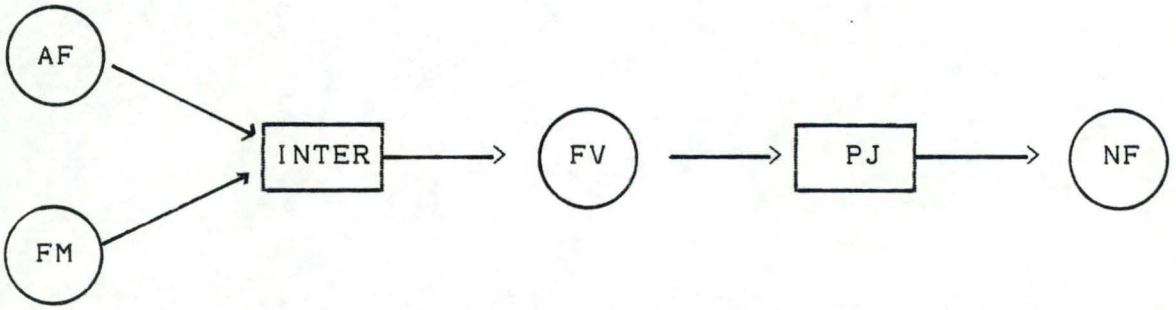
Les programmes possédant plusieurs flux séquentiels d'entrée que nous considérons sont des problèmes de mise-à-jour d'un (ancien) fichier et plus généralement des problèmes d'interclassement de fichiers.



AF : Ancien Fichier à mettre-à-jour;
FM : Fichier des Modifications;
P : programme de traitement.

Les deux fichiers d'entrée (l'ancien fichier et le fichier des modifications) doivent être parcourus "parallèlement" suivant une valeur de clé commune. Une clé permet de déterminer les groupes d'enregistrements portant la même valeur de clé d'un fichier. Une valeur de clé peut apparaître soit dans les deux fichiers d'entrée, soit dans un seul des deux, soit dans aucun des deux.

Le principe de résolution du problème de plusieurs flux d'entrée parcourus séquentiellement suivant les valeurs d'une clé, repose sur la représentation de l'ensemble de ces fichiers par une seule structure. On définit pour l'ensemble des flux d'entrée une structure (unique) d'un flux virtuel d'entrée constituée par un interclassement des différents flux d'entrée (réels).



AF : Ancien Fichier à mettre-à-jour;
FM : Fichier des Modifications;
FV : Fichier Virtuel des entrées;
NF : Nouveau Fichier mis-à-jour;
INTER : programme d'interclassement;
PJ : programme élaboré suivant la méthode de Jackson.

La méthode de Jackson est appliquée au flux virtuel (unique) d'entrée et au flux de sortie (unique). Le traitement des flux réels d'entrée suivant les valeurs de la clé (interclassement des deux fichiers d'entrée) est réalisé par un programme d'interclassement (INTER) entre les flux réels et le flux virtuel.

Quelques remarques peuvent être précisées à propos de la structure du fichier virtuel :

1- La structure de ce fichier doit être influencée par les structures physiques et logiques des fichiers réels d'entrée, mais elle doit également être influencée par la structure logique du fichier de sortie, afin d'élaborer une

"bonne" structure de programme entre le fichier virtuel et le fichier de sortie (cfr. paragraphe 1.3.4.3.).

2- La structure logique du fichier virtuel est créée avant sa structure physique. Nous pouvons dés lors structurer physiquement ces données virtuelles de manière à rendre le programme plus "simple" (cfr. paragraphe 1.3.3.).

Remarque 1 : une amélioration de cette méthode consisterait :

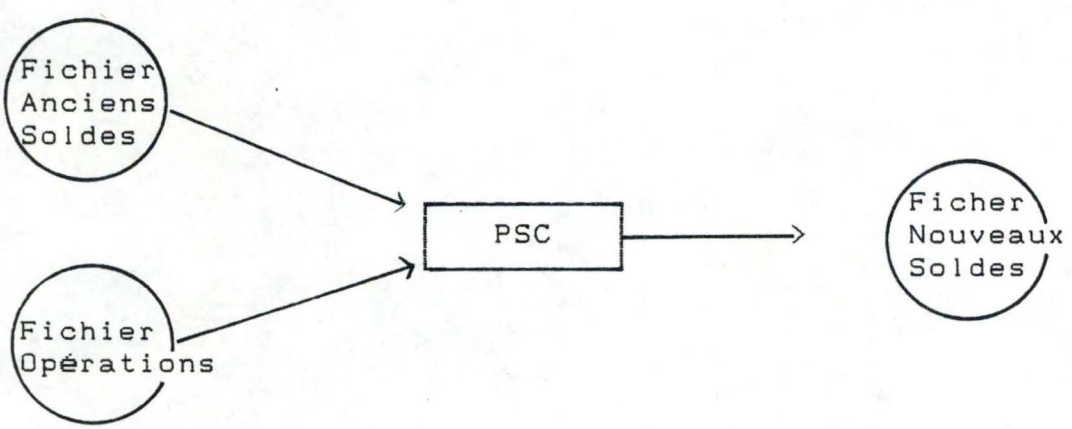
- soit à utiliser des opérations de lecture virtuelle que l'on intègre à la structure virtuelle du programme et qui opèrent directement sur les fichiers physiques sans utiliser le programme d'INTERclassement,

- soit à utiliser la technique de "l'inversion de programme" qui est présentée au paragraphe 1.4.4. et qui supprimerait le fichier virtuel (intermédiaire entre les fichiers d'entrée et de sortie). Le programme INTER serait inversé par rapport au fichier virtuel, c'est-à-dire : au lieu de lire directement un enregistrement du fichier virtuel, le programme PJ appellera le programme INTER qui lui fournira cet enregistrement. Le problème de plusieurs flux d'entrée est transparent au programme PJ, qui considère les données en entrée comme un seul flux séquentiel. L'opération de lecture (READ(F,E)) de l'enregistrement suivant (E) est remplacée par un appel à une procédure (INTER).

Remarque 2 : une autre méthode traitant plusieurs flux de données en entrée ou en sortie est présentée dans [JACKSON, 75, chap 4.2] et dans [VAN 'T DACK, 86, chap 2].

1.4.1.3. Exemple : Mise-à-jour des soldes_clients

Enoncé du problème :



"Dans une institution financière, on tient à jour un fichier des soldes clients : pour chaque client, il existe un enregistrement sous la forme suivante :

```
COMPTE_CLIENT = RECORD
                    NO_SOLDE : integer;
                    SOLDE    : integer
                END;
```

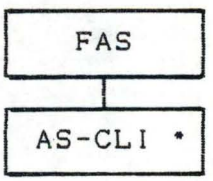
On demande d'élaborer un programme pour mettre ces soldes à jour via un fichier des opérations (FO) : les anciens soldes sont repris de leur fichier des anciens soldes (FAS), après quoi ils sont mis-à-jour si besoin est et transférés sur le fichier des nouveaux soldes (FNS). Les enregistrements de FO ont la forme suivante :

```
OPER_COMPTE = RECORD
                  TYPE_OPER : char; (C = crédit
                                     D = débit)
                  NO_CLIENT : integer;
                  MONTANT   : integer
              END;
```

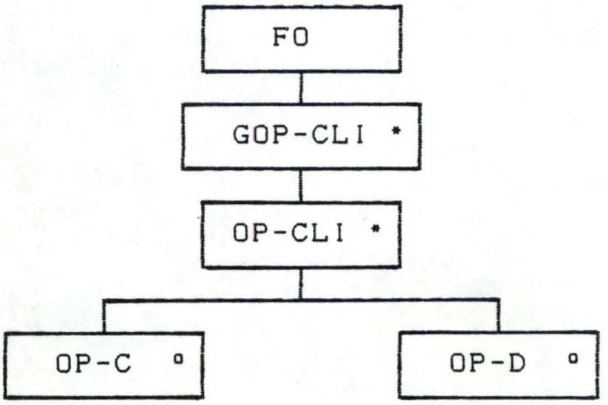
On obtient les nouveaux soldes en additionnant le montant de chaque opération Crédit (C) à l'ancien solde du client et en soustrayant le montant de l'opération Débit (D). Aussi bien FO que FAS sont classés dans l'ordre croissant des numéros des clients. Il n'existe pas d'enregistrement en double dans FAS. Tous les numéros de client dans FO existent dans le fichier FAS".

Les structures réelles des données en entrée et en sortie se présentent de la manière suivante :

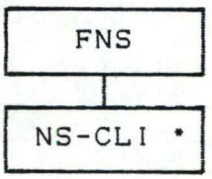
1) Fichier-Anciens-Soldes :



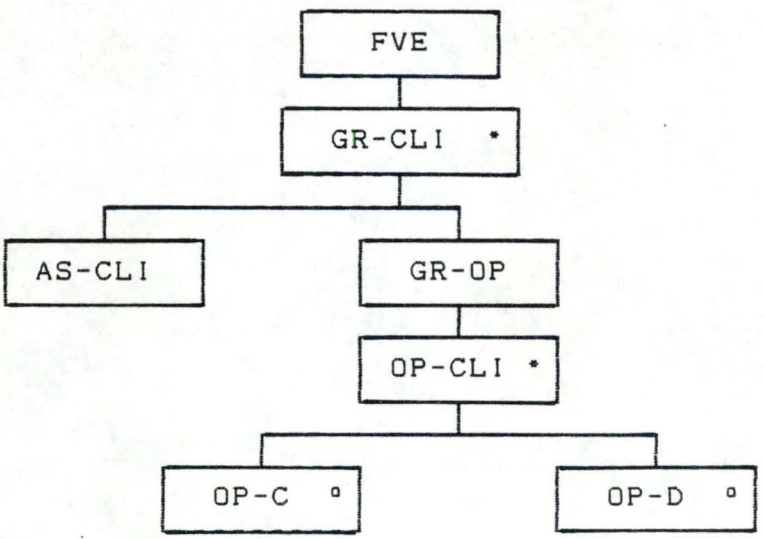
2) Fichier-Opérations :



3) Fichier-Nouveaux-Soldes :



Le fichier virtuel des entrées (FVE) doit faire correspondre les deux fichiers d'entrée FAS et FO sur base de la clé "NO-CLIENT".
La structure du fichier virtuel prenant en compte l'interclassement des fichiers réels FAS et FO sur base de la clé "nr_client", est présentée de la manière suivante :



Pour rendre le programme SOLDE_CLIENT le plus "simple" possible, on décide de représenter les enregistrements d'interclassement du fichier virtuel FVE de la manière

suivante :

```
ENR_VIRT = RECORD
                NO_CLIENT : integer;
                SOLDE     : integer;
                TYPE_OP   : char   ;  (C ou D)
                MONTANT   : integer;
           END;
```

Le programme SOLDE_CLIENT traitant le flux virtuel d'entrée FVE et le flux de sortie FNS est construit sur base de la correspondance entre les composants (FVE <--> FNS) et les composants (GR-CLI <--> NS-CLI) :

```
Program SOLDE_CLIENT(input,output);

var  fns : file of compte_client;
     fve : file of enr_virt;
     enrv : enr_virt;
     cc   : compte_client;
     noc  : integer;
     s    : integer;
     ff   : boolean;

begin
     reset(fve);
     rewrite(fns);
     pread(fve,enrv);
     while (not ff) do
     begin
          noc := enrv.no_client;
          s := enrv.solde;
          pread(fve,enrv);
          while ((not ff) and
                 (enrv.no_client = noc)) do
          begin
            if enrv.type_op = 'C'
              then s := s + montant
              else s := s - montant;
            pread(fve,enrv)
          end;
          cc.no_client := noc;
          cc.solde := s;
          write(fns,cc);
          if (not eof(fve))
            then noc := enrv.no_client
     end;
     close(fns);
     close(fve)
end.  {fin SOLDE_CLIENT}
```

Le programme d'interclassement des fichiers réels en un fichier virtuel se base sur la clé "numéro de client". Les deux

fichiers (FAS et FO) sont lus en parallèle et sont classés en ordre croissant suivant cette clé. Le fichier virtuel sera rangé suivant cette même clé "numéro de client" avec pour un même numéro de client l'enregistrement "ancien_solde" précédant les enregistrements "opération".

```
Programme INTER(input,output);

var  fas : file of compte_client;
     fo  : file of oper_compte;
     fve : file of enr_virt;
     elem1 : compte_client;
     elem2 : oper_compte;
     ev    : enr_virt;

begin
   reset(fas);
   reset(fo);
   rewrite(fve);
   read(fas,elem1);
   read(fo,elem2);
   while(not eof(fas)) do
   begin
      ev.no_client := elem1.no_client;
      ev.solde := elem1.solde;
      ev.type_op := ' ';
      ev.montant := 0;
      write(fve,ev);
      while ((not eof(fo)) and
          (elem1.no_client = elem2.no_client)) do
      begin
         ev.no_client := elem2.no_client;
         ev.solde := 0;
         ev.type_op := elem2.type_oper;
         ev.montant := elem2.montant;
         write(fve,ev);
         read(fo,elem2)
      end;
      read(fas,elem1)
   end;
   close(fas);
   close(fo);
   close(fve)
end.  {fin de INTER}
```

1.4.2. Difficultés d'identification et "Backtracking"

1.4.2.1. Introduction

Nous avons utilisé jusqu'à présent des formes d'itération et de sélection dans lesquelles la condition de test précède l'exécution de la partie de programme correspondante. Pour traiter les fichiers séquentiels, nous utiliserons la technique "d'une lecture à l'avance" (déjà utilisée précédemment et présentée dans le paragraphe 1.2.6).

Cette technique d'une lecture préalable nous assure que les données nécessaires sont disponibles en mémoire centrale afin que les conditions du test soient évaluées.

Pour certains types de problèmes, il n'est pas suffisant de lire un seul enregistrement parce qu'on ne dispose pas avec cet enregistrement de données nécessaires à la vérification immédiate des conditions de sélection ou d'itération.

L'information nécessaire pour l'évaluation de ces conditions est disponible par une lecture de plusieurs enregistrements (il est nécessaire que le flux de données en entrée soit "non-ambigu", c'est-à-dire qu'au maximum une lecture de toutes les données du flux doit permettre de déterminer les conditions de sélection).

Ces difficultés d'identification sont résolues dans la méthode J.S.P. grâce à deux techniques : la pré-lecture multiple et le "Backtracking" [JACKSON, 75, chap 6].

1.4.2.2. Classification des difficultés d'identification

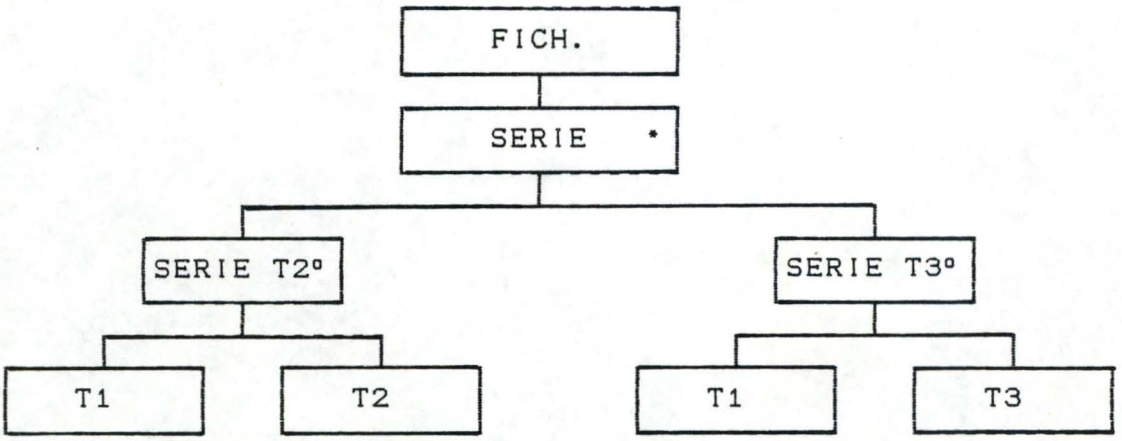
A. Définition

Une difficulté d'identification peut survenir lors d'une sélection d'un composant, c'est-à-dire : dans les deux cas ou les variables locales du programme et/ou l'enregistrement du flux de données en entrée suivant sont insuffisants pour décrire avec précision la condition de sélection.

B. Exemple

B1. Exemple 1 :

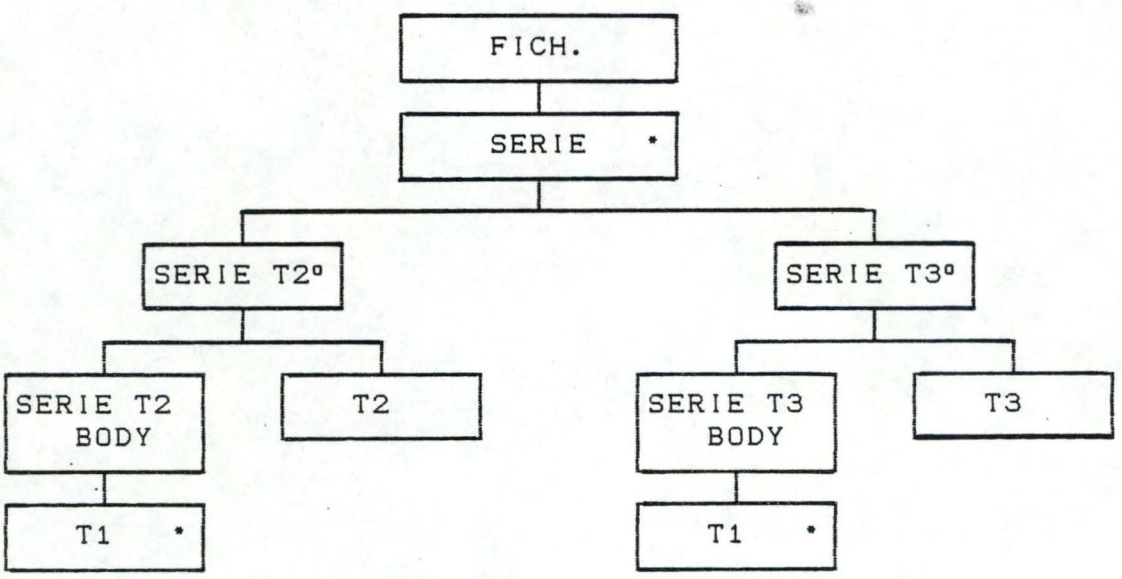
Considérons le fichier en entrée suivant :



En effectuant une seule lecture préalable, nous n'aurons comme seule donnée l'enregistrement T1. Cette information est insuffisante pour sélectionner et traiter l'un des composants "SERIE T2" ou "SERIE T3".
Le problème est totalement résolu si nous disposons toujours de deux enregistrements consécutifs, c'est-à-dire un "T1" suivi d'un "T2" ou d'un "T3". L'information du second enregistrement détermine toujours la série à traiter.

B2. Exemple 2 :
- - - - -

Considérons le fichier en entrée suivant :



Remarque : D'autres structurations de ce flux de donnée sont possibles pour résoudre le problème sans difficultés d'identification mais, comme nous l'avons vu dans le paragraphe 1.3.4.3.B., ces autres structurations peuvent être imcompatibles avec la structure de sortie.

Nous ne pouvons traiter ce fichier avec une prélecture d'un nombre fixe d'enregistrements. Dans quel cas, nous devrions effectuer la pré-lecture d'un nombre variable d'enregistrements avant de savoir si la série de "T1" est suivie de l'enregistrement final "T2" ou "T3".

C. Classification

Les difficultés d'identification peuvent être classées en deux catégories :

Catégorie 1 : Lecture multiple fixe

Les conditions sont évaluées en éffectuant la pré-lecture d'un nombre fixe d'enregistrements. C'est le cas de l'exemple 1.

Catégorie 2 : Lecture multiple variable

Les conditions sont évaluées en effectuant une pré-lecture d'un nombre indéterminé et variable d'enregistrements. C'est le cas de l'exemple 2.

1.4.2.3. L'histoire du parfais programmeur, et de la fée

Avant de présenter les différentes techniques de résolution des problèmes d'identification survenant lors d'une sélection ou d'une itération, vous pouvez lire dans [JACKSON, 75] au chap 6.4 (consacré aux problèmes du Backtracking), une "fable" présentant intuitivement le mécanisme de résolution adopté par le principe du Backtracking.

1.4.2.4. Pré-lecture multiple

La pré-lecture multiple ne peut être utilisée que pour la difficulté d'identification de catégorie 1, c'est-à-dire qu'un nombre fixe de lecture préalable permet d'évaluer les conditions de sélection ou d'itération.

Remarque : Le principe du placement des opérations de lecture est virtuellement inchangé par rapport à la technique d'une lecture préalable : nous lirons un nombre fixe d'enregistrements à l'ouverture du fichier et ultérieurement, une lecture sera faite, chaque fois qu'un enregistrement sera traité (tout en tenant compte de la marque de fin de fichier à

chaque lecture d'un enregistrement).

Exemple : programme correspondant à l'exemple 1 :

```
Program FILE(input,output);

var   f : file of T;
      e1, e2 : T   ;
      ff : boolean ;

begin
   reset(f);
   pread(f,e1);   (1)
   pread(f,e2);
   while (not ff) do
   begin
      if (e2 = TYPE_T2)
        then begin
                procedure_T1;
                procedure_T2
             end
        else begin
                procedure_T1;
                procedure_T3
             end
   end;
   close(f)
end.  {fin de FILE}
```

Chaque "procedure_Ti" traite un composant Ti et lit un enregistrement du groupe suivant s'il existe sinon, la marque de fin de fichier est mise à "true".

1.4.2.5. Backtracking

A. Introduction.

Le BACKTRACKING peut être utilisé pour résoudre toutes les catégories de difficultés d'identification, et notamment la deuxième catégorie, où la pré-lecture d'un nombre fixe d'enregistrements ne fournit pas toujours une solution.

B. Principe du BACKTRACKING.

L'enseignement de l'histoire "du parfait programmeur et de la fée" est de se fier à la bonne fée, c'est-à-dire que nous pouvons choisir le premier chemin (sous-composant) et n'ayant pas toutes les informations nécessaires, nous supposons que ce choix est correct. La fée nous permettra d'annuler ce choix s'il s'avère, qu'en collectant des informations supplémentaires, nous

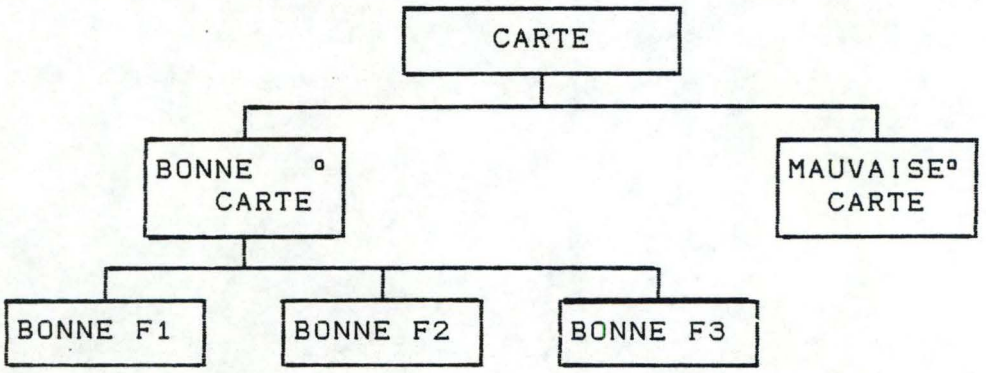
(1) Définition de l'instruction "pread" : page 13.

avons fait le mauvais choix du chemin et elle nous ramènera à la bifurcation (composant sélection) où nous emprunterons le chemin suivant (sous-composant suivant).

C. Procédure de Backtracking.

Prenons un exemple simple pour expliquer la procédure. Un fichier est composé d'un seul enregistrement "CARTE". Cette CARTE est soit un bon enregistrement, soit un mauvais enregistrement. Si c'est une mauvaise carte, nous ne sommes pas intéressés par son contenu (nous exécutons seulement Proc_mauv_carte). Si c'est une bonne carte, elle est composée d'une bonne_F1, d'une bonne_F2 et d'une bonne_F3.

La structure que nous désirons utiliser est définie de la manière suivante :



La difficulté est que la condition de sélection ne peut être évaluée avec une seule analyse de "BONNE_F1" (premier élément d'une bonne carte). Nous ne savons pas s'il faut prendre le chemin de gauche (bonne carte) ou le chemin de droite (mauvaise carte).

Le problème de Backtracking peut être expliqué en trois étapes :

Etape 1 : Cette étape consiste simplement à ignorer le problème d'identification (la fée nous indique le chemin à emprunter). Le problème est dès lors une simple sélection et le programme correspondant peut s'écrire directement :

```
Program CARTE(input,output);

var  fcart : file of carte;
     elem  : carte        ;
     bonne_carte : boolean ;

begin
     reset(fcart);
     read(fcart,elem);
     if (bonne_carte)
     then begin
               proc_F1;
               proc_F2;
               proc_F3
          end
     else proc_mauv_carte
end. {Fin de CARTE}
```

(programme C.1)

Etape 2 : La deuxième étape consiste à pouvoir vaincre le démon qui nous a mis sur un mauvais chemin et nous permettre de faire demi-tour, c'est-à-dire : transformer le choix (composant-sélection) en une hypothèse sur un des chemins (sous-composants-sélection) (bonne_carte := true) et ajouter les ordres de retour aux endroits où la supposition peut être réfutée.

Nous devons dés lors adopter une attitude plus sceptique pour exécuter une "bonne carte", en se demandant à chaque opportunité si après tout, il ne s'avère pas que la carte est mauvaise.

Pour réaliser ce type d'opération de retour, nous créons une nouvelle instruction Pascal "WHILETRUE" dont la sémantique est :

"on exécute séquentiellement les instructions 'ins_i' tant que l'expression "booléenne" 'exp' est vérifiée. Si après la terminaison de 'WHILETRUE' (c-à-d après 'ins_n') l'expression 'exp' est toujours vérifiée, alors on n'exécute pas 'WHENFALSE" sinon, on exécute l'instruction 'ins_{n+1}'." :

```
WHILETRUE exp DO
BEGIN
    ins1 ;
    ins2 ;
    ...
    insn
END
WHENFALSE DO
    insn+1
```

(l'implémentation de cette instruction est laissé comme exercice au lecteur)

La solution de l'étape 1 devient :

```
Program CARTE(input,output);

var  fcart : file of carte;
     elem  : carte        ;
     bonne_carte : boolean;

begin
     reset(fcart);
     read(fcart,elem);
     bonne_carte := true;
     whiletrue bonne_carte do
     begin
          if (erreur_cart_1)
            then bonne_carte := false;
          proc_F1;
          if (erreur_cart_2)
            then bonne_carte := false;
          proc_F2;
          if (erreur_cart_3);
            then bonne_carte := false;
          proc_F3
     end
     whenfalse do proc_mauv_carte;
end. {Fin de CARTE}
```

Nous avons fait deux changements,

1- dans la première étape, nous ne nous préoccupions pas de savoir si la carte était bonne ou fausse (bonne_carte n'était pas initialisé). Le premier changement est de faire l'hypothèse d'être en présence d'un bonne carte (bonne_carte := true) et s'il s'avère que notre hypothèse est fausse, nous admettrons que la carte est une fausse carte et nous traiterons donc une mauvaise carte (whenfalse bonne_carte do).

2- nous avons introduit des appels de procédure (erreur_carte_1(elem), etc). Ce sont des opérations

éxécutables, placées à chaque point dans la structure de données d'une bonne carte où l'hypothèse d'une "bonne carte" serait falsifiée par les faits. Donc "if erreur_carte_1(elem) then bonne_carte := false" signifie "si ce n'est pas une bonne F1, alors la carte ne peut être une bonne carte, et nous devons admettre que notre hypothèse est fausse".

Etape 3 : Ce type d'opération va amener des effets secondaires en cas de retour de chemin (et qu'il faudra gerer).

Ces effets secondaires sont de trois types :

1. Les effets secondaires intolérables qui devront être supprimés.

   ex : - une variable qui a été augmentée devra être ramenée à sa valeur de départ;

   - il faut éffacer la ligne qui aurait été écrite dans le fichier résultat.

   - il faut faire un "backspace" du lecteur de carte s'il a déjà lu une carte.

2. Les effets secondaires neutres qui peuvent mais ce n'est pas obligatoire, être annulés.

   ex : - une variable mise à jour dont on n'aura plus d'utilité par la suite.

3. Les effets secondaires avantageux qui doivent être conservés. Si nous les annulons, nous devrons les exécuter à nouveau sur un autre chemin.

   ex : - dans l'exemple 1 de la section 1.4.2.2., il ne faut plus traiter à nouveau dans le deuxième chemin possible (T1-T3) les T1 que nous avons déjà traités dans le premier chemin (T1-T2).

D. Traitement des effets secondaires.

D1. Identification des problèmes.

Les effets secondaires inacceptables et avantageux nous obligent à apporter quelques modifications au programme texte. Pour illustrer ces modifications, nous utiliserons l'exemple suivant :

Enoncé du problème "string délimité" :

Un composant de programme doit être conçu pour analyser des

caractères d'un string (S) inclus dans un string T et imprimer deux sous-strings (S1 et S2) inclus dans S. S1 est terminé par le caractère '@', et S2 par le caractère '&'; le string complet S est terminé par le caractère '%'.
A l'entrée de ce composant de programme, deux paramètres sont disponibles :

- le string complet T;
- La position SS du premier caractère du string S dans le string T.

S1 est défini comme étant le sous-string de S dont le premier caractère est le caractère positionné par le paramètre SS et dont le dernier caractère est le caractère de terminaison '@'; S2 a comme premier caractère le caractère suivant la terminaison '@' de S1, et son dernier caractère est la terminaison '&' (il peut exister des caractères non-terminaisons résiduels entre le caractère '&' et le caractère terminal du string S '%'). L'un ou l'autre string peut être dépourvu de caractères non-terminaisons. La longueur maximale d'un string complet S (% inclu) est de 100 caractères à partir de la variable localisation SS d' entrée.
Ce composant de programme doit vérifier si les deux string S1 et S2 sont présents et corrects, et un rapport sera imprimé dans la forme suivante :

```
STRING CORRECT

S1 = xxxxxxx@
S2 = yyyy&
```

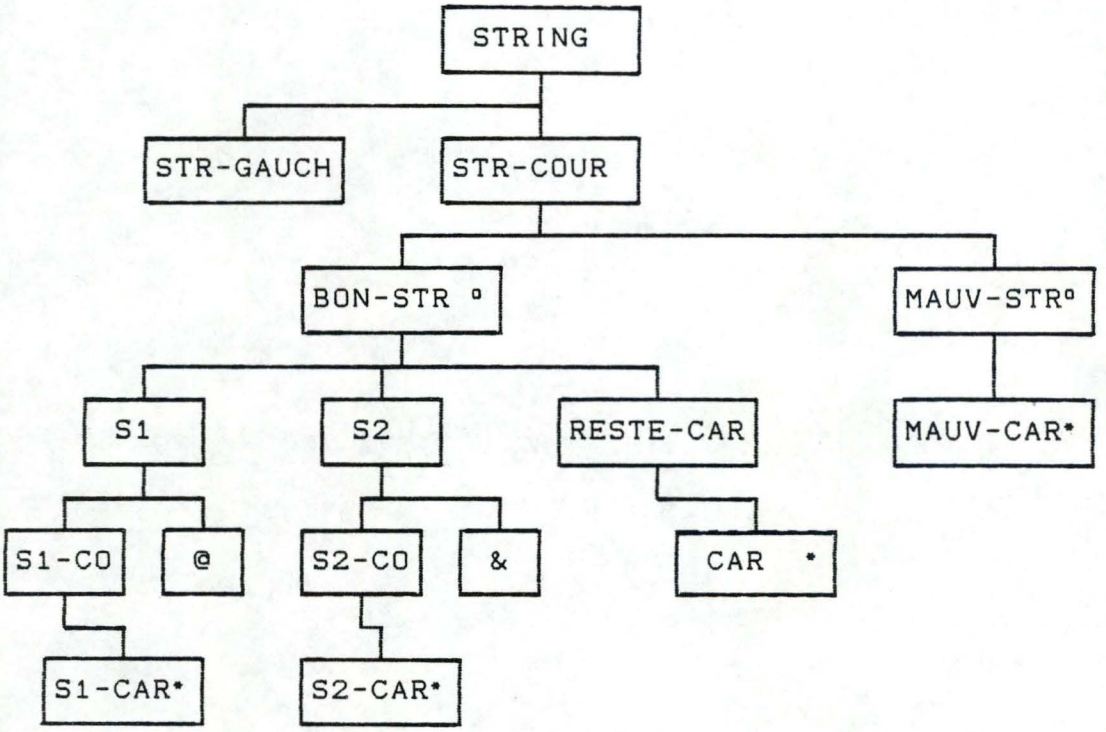
Dans le cas où le string serait incorrect, il devra être imprimé à partir de la localisation courante SS jusqu'au caractère de terminaison '%', de la manière suivante :

```
STRING INCORRECT :

CHAR-001 = f
CHAR-002 = g
CHAR-003 = e
...
CHAR-nnn = %
```

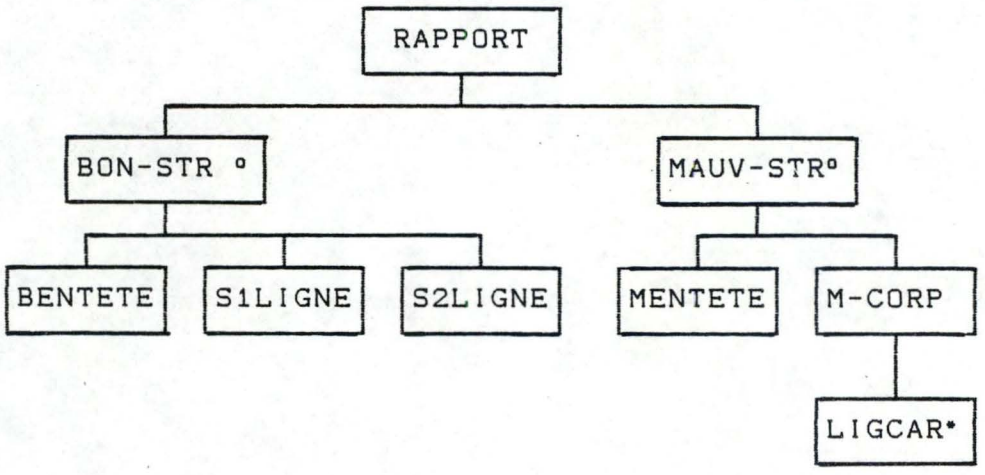
Dans chaque cas, la variable position du string S 'ss' sera placée, à la fin du programme après le caractère de terminaison

'%'. Les deux structures de données peuvent être représentées de la manière suivante :

String d'entrée S :



Rapport de sortie :



La structure du programme peut être facilement construite en utilisant les étapes suivantes de construction de programme.
Nous présentons tout de suite ce programme dans la version de la première étape du problème de Backtracking (point E) :

```
Procedure STRING(var ss : integer;
                 var t : stringt);

var  pstring : stringt;
     ps      : integer;
     ch      : integer;
     bon_str : boolean;

begin
   if (bon_str = true)
   then begin
          writeln('STRING CORRECT');

          pstring := '';                          ┐
          ps := 1;                                │
          while (t[ss] <> '@') do                 │
          begin                                   │
             pstring[ps] := t[ss];                │
             ps := ps + 1;                        │  Proc_S1
             ss := ss + 1                         │
          end;                                    │
          pstring[ps] := t[ss];                   │
          ps := ps + 1;                           │
          ss := ss + 1;                           │
          writeln('S1 = ',pstring);               ┘

          pstring := '';                          ┐
          ps := 1;                                │
          while (t[ss] <> '&') do                 │
          begin                                   │
             pstring[ps] := t[ss];                │
             ps := ps + 1;                        │  Proc_S2
             ss := ss + 1                         │
          end;                                    │
          pstring[ps] := t[ss];                   │
          ps := ps + 1;                           │
          ss := ss + 1;                           │
          writeln('S2 = ',pstring);               ┘

          while (t[ss] <> '%') do                 ┐
                ss := ss + 1;                     │  Proc_FIN
          ss := ss + 1;                           ┘

        end
   else begin  {bon_str = false}

          writeln('STRING INCORRECT');            ┐
          ch := 001;                              │
          while (t[ss] <> '%') do                 │
          begin                                   │
             writeln('CHAR-',ch,' = ',t[ss]);     │
             ss := ss + 1;                        │  Proc_False
             ch := ch + 1                         │
          end;                                    │
          writeln('CHAR-',ch,' = ',t[ss]);        │
          ss := ss + 1;                           │
          ch := ch + 1;                           ┘

        end {else}
end; {procedure STRING}          (programme S1)
```

La variable 'ss' représente la position courante du caractère à traiter dans le string S par rapport à T. La variable 'ps' est utilisée comme variable de la position courante dans le string d'impression S1 ou S2.

A cette étape, nous ne nous sommes pas préoccupés de savoir si nous étions en présence d'un string correct ou incorrect. Nous considérons tout d'abord que nous sommes en présence d'un bon string (if (bon_str = true) then ..), c'est-à-dire que nous rencontrons les caractères '@' et '&' (dans cet ordre) avant d'atteindre la fin du string '%'. Nous envisageons ensuite le cas du mauvais string (if (bon_string = true) .. else ..).
A la deuxième étape, nous adoptons une attitude plus critique dans notre hypothèse d'un bon_string. Notre hypothèse peut être réfutée seulement quand nous rencontrons un '%' avant que S1 et S2 aient été complétement traités (c'est le seul cas d'erreur que nous envisageons de traiter). Nous devons, dés lors, insérer des opérations d'interruption du traitement d'un bon_string en cas de détection d'une erreur dans le string (if schar(ss) = '%') then bon_str := false) (on vérifie la présence ou non du caractère '%' avant les caractères '@' et '&' dans le string).
La solution de l'étape deux se présente dés lors de la manière suivante :

```
Procedure STRING(var ss : integer;
                 var t : stringt);

var  pstring : stringp;
     ps      : integer;
     ch      : integer;
     bon_str : boolean;

begin
   bon_str := true;
   whiletrue bon_str do
   begin
       writeln('STRING CORRECT');
       Proc_S1;
       Proc_S2;
       Proc_FIN
   end
   whenfalse do Proc_False
end; {procedure STRING}
```

(programme S2)

et où Proc_S1 et Proc_S2 sont modifiés de la manière suivante (on teste les cas d'erreur, c-à-d la présence du caractère '%' dans les strings S1 et S2) :

```
while (t[ss] <> '@') do
if t[ss] <> '%'
then begin
        pstring[ps] := t[ss];
        ps := ps + 1;
        ss := ss + 1
     end
else bon_str := false;
```

Dans la troisième étape, nous allons nous occuper des effets secondaires qui peuvent se présenter en cas de détection d'un mauvais string pendant le traitement d'un bon string. Avant d'aborder cette étape, nous pouvons remarquer que les variables 'pstring' et 'ps' sont locales à la partie (bon_string = true), contrairement à la variable 'ss'. Dans la présentation des différentes méthodes, nous ferons la différence entre les effets secondaires réversibles et les effets secondaires irréversibles. Une opération réversible est une opération dont les effets peuvent être supprimés par une opération inverse (ex : x := x+1; opération inverse : x := x-1). Les effets d'une opération irréversible ne peuvent être supprimés par une autre opération (ex : les opérations d'écriture sur listing).

D2. 'Inversion' des effets d'un programme.

Le problème de manipulation des effets secondaire est le problème de la restauration de l'assertion originelle. Si l'assertion {P} définit l'ensemble des valeurs des variables au point d'entrée d'un ensemble d'opération S et l'assertion {Q}, l'ensemble des valeur des variables après exécution des opérations S : {P} S {Q}, alors la gestion des effets secondaires consiste à retrouver les valeurs originelles {P} des variables à partir des valeurs {Q} par une suite d'opération S' : {Q} S' {P}. S' doit défaire les effets d'un programme S afin de retrouver l'assertion originelle {P}.

D3. Méthodes d'élimination des effets inacceptables.

Nous présenterons tout d'abord une méthode très générale, tandis que les méthodes suivantes seront des améliorations de cette dernière en essayant de pallier à certains désavantages. Les quatre premières méthodes permettent la gestion que des effets secondaires réversibles, tandis que la dernière méthode tiendra compte des effets irréversibles.

Première méthode :

Cette première méthode consiste à ne pas créer d'effets secondaires en déterminant les conditions d'entrée d'une sélection. Avant d'exécuter une branche de la sélection, on détermine par appel à une fonction l'évaluation des conditions. Cette fonction effectue un certain nombre d'opérations nécessaires sur les données pour évaluer les conditions de sélection, et restaure l'état initial de ces données à la fin de son exécution. Dans l'exemple du string, la fonction appelée lit une première fois le string pour évaluer l'ordre correct des caractères spéciaux (@,&,%), et renvoie comme résultat la valeur du boolean 'bon_str'.
On reprend la version 1 du programme STRING (programme S1), auquel on ajoute l'appel à la fonction BON_STR avant l'opération 'if bon_str then'.

```
Program STRING(input,output);

var  pstring : stringp;
     ps      : integer;
     ch      : integer;
     bon_str : boolean;


function BON_STR

var s : integer;

begin
   bon_str := true;
   s := ss;
   while (bon_str) and
         (t[s] <> '@') do
   begin
      if t[s] = '%'
        then bon_str := false
        else s := s+1
   end;
   while (bon_str) and
         (t[s] <> '&') do
   begin
      if t[s] = '%'
        then bon_str := false
        else s := s+1
   end
end;  {fin de la fonction BON_STR}


begin
   if (BON_STR)
   then begin
           writeln('STRING CORRECT');
           Proc_S1;
           Proc_S2;
           Proc_FIN
        end
   else Proc_False
end; {Fin programme STRING}
```

(programme S3)

■ L'avantage de cette méthode est de ne pas créer d'effets secondaires, c'est-à-dire que l'on ne traite une branche d'un composant sélection que si on est certain de son choix. Ce choix ne sera pas réfuté par la suite.

■ Le désavantage de cette solution est que l'on fait plusieurs fois certaines opérations. Une première fois dans la fonction appelée (BON_STR) pour l'évaluation de la condition, et une seconde fois dans la procédure appelante (STRING) qui

considère que rien n'a encore été fait.

Les méthodes suivantes tenterons d'améliorer cette méthode, en opérant un choix d'une branche du composant sélection et en permettant un retour en cas d'erreur (cfr. l'étape 2 de la section C)

Seconde méthode :

Cette méthode consiste, en cas de détection d'un mauvais choix d'une branche de sélection, à exécuter l'inverse des opérations effectuées jusqu'à présent (exécution partiel ou complète d'un des sous-composants-sélection).

$\{P\}\ Si\ \{Qi\}$

$\{Qi\}\ Si^{-1}\ \{P\}$

S : ensemble d'opérations;
Si : une exécution partielle ou complète de S;
{P} : ensemble des valeurs des variables en entrée de S;
{Qi} : ensemble des valeurs des variables après exécution de Si;
Si^{-1} : exécution des opérations inverses de Si.

On détermine à chaque possibilité d'erreur (if schar(ss) = '%' then ..), l'ensemble des opérations Si^{-1} permettant de retrouver les valeurs originelles des variables {P}.

Exemple : Cette seconde méthode appliquée à la seconde version du programme STRING (programme S2), modifie les Proc_S1 et Proc_S2 de la manière suivante :

```
if t[ss] <> '%'
then begin
        pstring[ps] := t[ss];
        ps := ps + 1;
        ss := ss + 1
     end
else begin
        while (t[ss] <> '%') or
              (ss > 0) do
                 ss := ss - 1;
        EFFACER_BON_STRING
        bon_str := false;
```

$=S^{-1}$

Nous supposons que la procédure EFFACER_BON_STRING est capable d'inverser les effets de l'opération 'writeln()'. La cinquième méthode permetttra de gérer de telles opérations irréversibles.

Cette technique palliant le désavantage de la première méthode

entraine d'autres désavantages tels que par exemple l'inversion de l'opération 'writeln()'.

■ Désavantages :

1. Cette méthode dépend de la réversibilité des effets secondaires individuels.

2. Cette méthode produit un texte programme relativement long, dû à la répétition des opérations inverses à différents endroits du programme.

Le second désavantage peut être surmonté par une optimisation. Le premier désavantage est plus sérieux et peut parfois être résolu par l'utilisation de facilités "hardware" telles que le "backspace" d'une imprimante ou d'un lecteur de cartes; le "rewind" d'une bande, etc .., mais celles-ci ne sont pas toujours présentes. C'est pour ces raisons que nous préférerons les méthodes suivantes.

La troisième et la quatrième méthodes utilisent des techniques permettant de surmonter le second désavantage en travaillant sur des variables locales aux sous-composants-sélection.

Troisième méthode :

■ Cette méthode plus générale nous permet d'éviter l'examen détaillé du contexte individuel des opérations de retour. Cette méthode consiste à se souvenir des états des variables globales (G) en entrée du premier sous-composant-sélection et de restaurer ces états en entrée du second sous-composant-sélection (si le premier échoue). On enregistre dans des variables locales (L) l'ensemble des valeurs des variables globales avant l'exécution du sous-composant-sélection, et on réinisialise en cas de retour, les variables globales en utilisant les variables locales :

```
P<g>  ::= l:=g;
          P<g>;
          if KO then g:=l;
```

■ Exemple : La procédure STRING de la seconde étape (page 55) devient :

```
Procedure STRING(var ss : integer;
                 var t  : stringt);

var  pstring : stringp;
     ps      : integer;
     ch      : integer;
     bon_str : boolean;
     s       : integer;

begin
   bon_str := true;
   whiletrue bon_str do
   begin
       s := ss;
       writeln('STRING CORRECT');
       Proc_S1<ss>;
       Proc_S2<ss>;
       Proc_FIN<ss>
   end
   whenfalse do
   begin
       ss := s;
       Proc_False
   end
end; {procedure STRING}
```

(programme S5)

(les Proc_Si et Proc_FIN travaillent sur les variables globales 'SS')

Dans cet exemple, la partie des états des variables qui nous concerne est la valeur de "ss"; nous enregistrons dés lors cet état par l'opération "s := ss;" en entrée du premier sous-composant-sélection (whiletrue bon_str do), et nous restaurons cet état par l'opération "ss := s;" en entrée du second sous-composant-sélection (whenfalse do).

Remarque : nous ne nous préoccupons toujours pas du problème de l'écriture du rapport qui sera traité dans la méthode 5.

Quatrième méthode :

Cette quatrième méthode consiste à éviter la création d'effets secondaires dans le premier sous-composant-sélection. A la place d'opérer sur les variables globales (G) qui sont connues en dehors de ce premier sous-composant-sélection (que nous appellerons A), nous imposons la restriction que ce composant A opère seulement sur des variables locales (L). Nous devons initialiser les variables locales appropriées en entrée de A, avec l'ensemble des variables non-locales. Si le composant A termine avec succès sa séquence d'opérations, on assigne les nouvelles valeurs des variables locales aux variables connues en

dehors de A :

```
P<g> ::= l:=g;
         P<l>;
         if OK then g:=l;
```

■ Exemple : La procédure STRING de la seconde étape (page 55) devient :

```
Procedure STRING(var ss : integer;
                 var t  : stringt);

var  pstring : stringp;
     ps      : integer;
     ch      : integer;
     bon_str : boolean;
     s       : integer;

begin
   bon_str := true;
   whiletrue bon_str do
   begin
       s := ss;
       writeln('STRING CORRECT');
       Proc_S1<s>;
       Proc_S2<s>;
       Proc_FIN<s>;
       ss := s
   end
   whenfalse do Proc_False
end; {procedure STRING}
```

(programme S6)

(les Proc_Si et Proc_FIN travaillent sur les variables locales 'S')

La variable 'ss' est enregistrée dans la variable locale 's' dans le composant "whiletrue bon_str do" par l'opération "s := ss;", et en cas de réussite de ce choix (bon_str = true) la variable globale 'ss' est mise à jour par l'opération "ss := s" à la fin du sous-composant-sélection "whiletrue bon_str do".

Cette quatrième méthode paraît plus difficile dans l'élaboration du programme car il faut, outre les opérations d'affectation des variables en début et fin du premier sous-composant, remplacer toutes les variables globales par des variables locales dans ce premier sous-composant. Tandis que la méthode précédente, il suffit d'ajouter des instructions d'affectation au début des deux sous-composants-sélection.

Cinquième méthode :

Cette méthode généralise la quatrième méthode de résolution des effets secondaires réversibles aux effets secondaires non-réversibles. Le problème séquentiel d'entrée-sortie peut être traité de manière analogue par utilisation de fichiers virtuels (considérés comme des variables locales) sur lesquels d'autres opérations sont possibles. En cas de retour, on retrouve les fichiers originels (globaux) non-modifiés et en cas de réussite, on modifie ces fichiers originaux suivant les fichiers virtuels (de manière analogue à la quatrième méthode présentée pour les effets réversibles).

■ Exemple :

Nous utiliserons pour l'écriture du rapport le fichier virtuel FRAPPORT. L'impression du rapport ne se fera qu'à la fin d'un des sous-composants-sélection, c'est-à-dire au moment où l'on sera certain d'être en présence d'un bon string ou non. En cas de détection d'un mauvais string lors du traitement d'un bon_string, il suffira de supprimer le fichier virtuel du rapport FRAPPORT et de traiter le composant "mauvais string". Par l'utilisation d'un fichier virtuel, on évitera de créer des effets secondaires irréversibles.

La procédure STRING (S6) de la quatrième méthode (page 62) devient :

```
Procedure STRING(var ss : integer;
                 var t  : stringt);

var  frapport : file of stringp;
     elemr    : stringp        ;
     pstring  : stringp        ;
     ps       : integer        ;
     ch       : integer        ;
     bon_str  : boolean        ;
```

```
begin
   rewrite(frapport);
   bon_str := true;
   whiletrue bon_str do
   begin
       s := ss;
       Proc_S1;
       Proc_S2;
       Proc_FIN;
       ss := s;
       close(frapport);
       reset(frapport);
       read(frapport,elemr);
       while(not eof(frapport)) do
       begin
           writeln(elemr);
           read(frapport,elemr);
       end
   end
   whenfalse do Proc_False
end; {procedure STRING}
```

(programme S7)

et où les instructions d'écriture des Proc_S1 et Proc_S2 'writeln('Si = ',pstring)' sont remplacées par les instructions 'writeln(frapport,'Si = ',pstring)'.

1.4.3. Conflits de structure

1.4.3.1. Introduction

La structure d'un programme doit être basée sur les structures de tous les flux de données d'entrée et de sortie, c'est-à-dire que la structure d'un programme doit refléter en même temps toutes les structures de données, en identifiant les correspondances 1-1 entre elles. Ceci n'est possible que si la structure du programme peut correspondre aux différentes structures de données à la fois.

Un conflit de structure est un problème dans lequel des structures de données ne peuvent être mises en correspondance. Dans cette section, nous exposerons les catégories de conflit, la nature des difficultés causées par un conflit de structure, et nous expliciterons ces conflits par quelques exemples.

1.4.3.2. Types de conflits de structure

A. Pour simplifier la présentation des différents conflits, nous supposons pour chacun d'entre eux, un problème n'ayant qu'un seul flux de données en entrée et un seul en sortie. Cette restriction ne change rien aux caractéristiques de ces conflits.

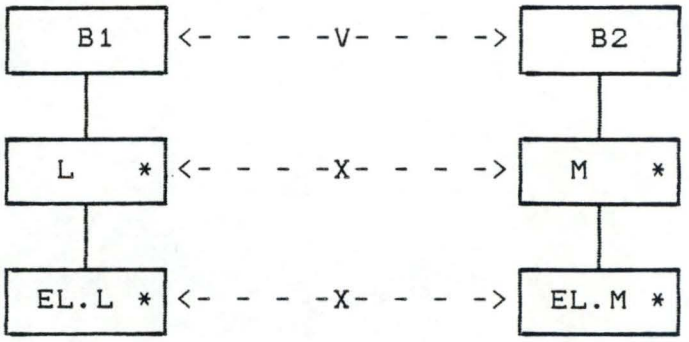
   Supposons qu'on ait défini pour un problème donné, un arbre A représentant une "bonne" structure des entrées et un arbre B représentant une "bonne" structure des sorties.
   On dit alors qu'il y a conflit de structure entre A et B si on ne peut pas établir une correspondance entre A et B selon les règles définies précédemment dans le paragraphe 1.2.5.

B. On distingue plusieurs formes de conflits de structure (classification non exhaustive) :

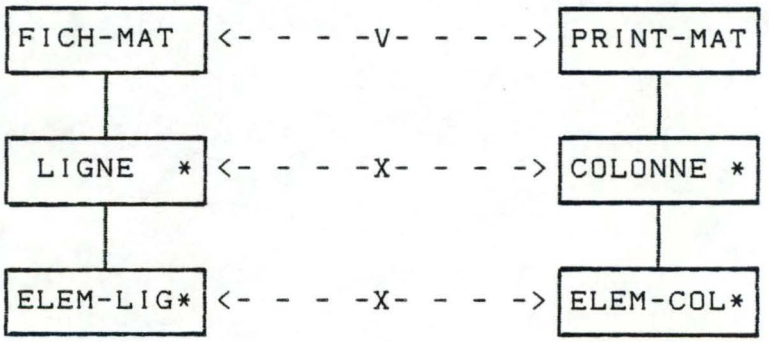
   B.1. Conflit d'ordre :

   Ce type de conflit apparaît lorsque les composants des deux structures de données ne se présentent pas dans le même ordre (Van 't dack, 86);

Schéma d'identification :



Exemple : nous avons un fichier d'entrée contenant les rangées d'une matrice. Nous souhaitons écrire un programme qui imprimerait les colonnes de la matrice.
Les structures de données sont :



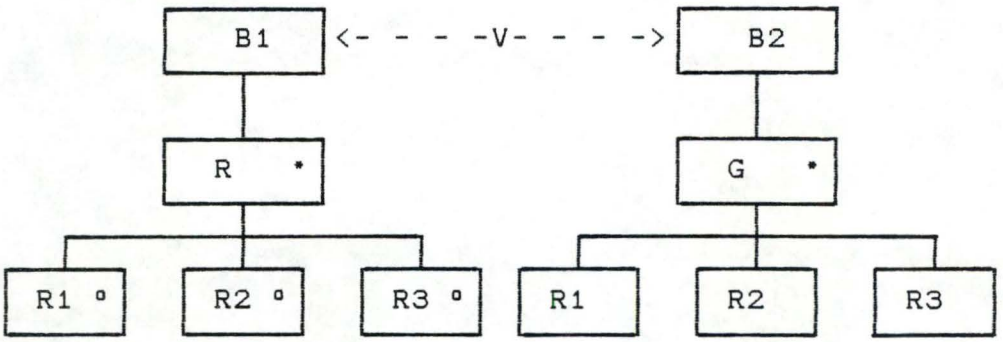
(exemple 1.1)

Il y a correspondance entre les composants supérieurs "FICH-MAT" et "PRINT-MAT" puisque "PRINT-MAT" est dérivée de "FICH-MAT". Mais les "LIGNE" ne correspondent pas aux "COLONNE". L'impression d'une "COLONNE" n'est pas dérivé d'une lecture d'une "LIGNE" et en général, il n'y a pas le même nombre de rangées que de colonnes. Au niveau des "ELEMENT", il n'y a pas non plus de correspondance, même si il y a le même nombre d'"ELEMENT", car il ne respecte pas la règle 2 de correspondance définie plus tôt.

B.2. Conflit d'entrelacement :

Ce type de conflit se présente lorsque deux composants de A et B ne peuvent être mis en correspondance parce que les données qui constituent les occurrences de ces composants sont regroupées selon des critères différents;

Schéma d'identification :



Dans ce cas de conflit, deux (ou plusieurs) groupes différents sont entrelacés l'un dans l'autre mais l'ordre est respecté au sein de chaque groupe. En B1 (entrée), les données relatives à chaque entité ($R1^{o}$-$R2^{o}$-$R3^{o}$) sont entrelacées avec les données des autres entités. B2 contient les entités non entrelacées (R1-R2-R3).

Exemple : System Log (exemple 1.2)

Un système "Time-sharing" collecte des informations sur l'usage d'un système. Ces informations consistent en des enregistrements sur les "Log-on", les "Log-off", les "Program-load" et les "Program-unload". Le système accorde un numéro de travail pour la session "Log-off -- Log-on". Un utilisateur recevra deux numéros de travail différents s'il se "log-on" en deux occasions différentes. Le système assure à ce qu'un utilisateur ne puisse lancer qu'un seul "log-on" si le terminal est libre (c'est-à-dire dans le cas où aucune autre session n'est occupée sur ce terminal) et qu'il ne puisse faire aucun "log-off" sauf s'il a préalablement fait un "log-on".

Chacune de ces sessions contient un nombre d'exécutions de programmes. La mise en marche d'un programme a lieu par un "load", l'arrêt par un "unload". A un moment choisi de la session, il ne peut y avoir qu'un seul programme actif : l'utilisateur doit d'abord arrêter ("unload") son programme avant de mettre en marche ("load") un autre (ou à nouveau le même).

Les informations concernant l'utilisation du système sont enregistrées sur une bande magnétique :

- un enregistrement "log-on" avec données : code = 'N', numéro travail, moment "log-on";

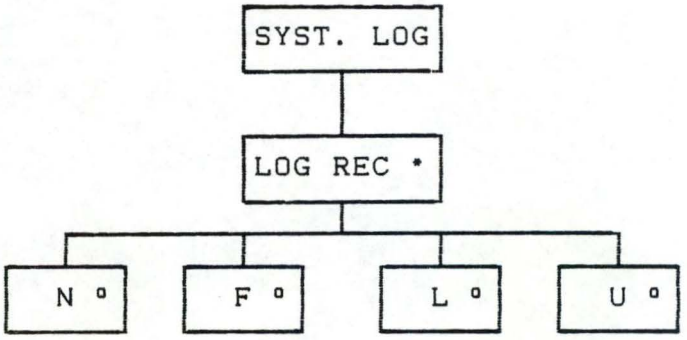
- un enregistrement "log-off" avec données : code = 'F', numéro de travail, moment "log-off";

- un enregistrement "program-load" avec données : code = 'L', numéro de travail, program-id, moment "load";

- un enregistrement "program-unload" avec donnée : code = 'U', numéro de travail, program-id, moment "unload".

Les enregistrements sont naturellement écrits en ordre chronologique.

La structure du fichier d'entrée "System-log" est :

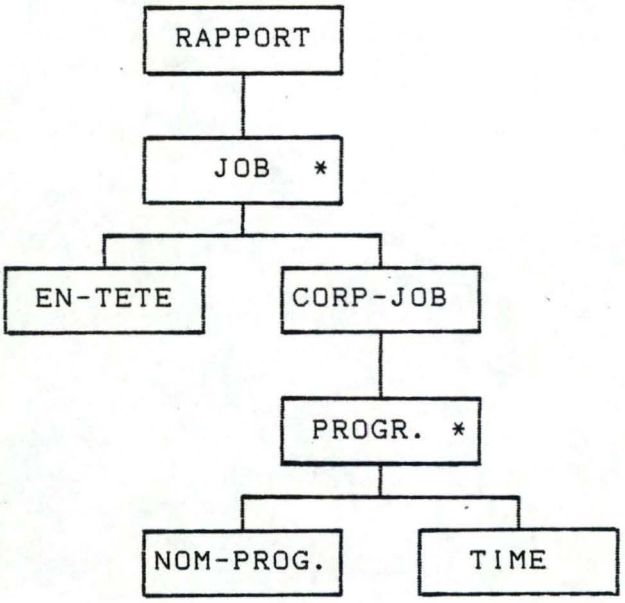


A partir de ces informations, on demande de produire un rapport présentant un aperçu des sessions sous le format suivant :

```
JOB 1456
        PGM     X0453R86        TIME = ssss
        PGM     X0453R87        TIME = dddd
        PGM     X0453R88        TIME = eeee
        ....
                                JOB TIME = tttt

JOB 1457
        ....
```

La rubrique TIME donne la durée d'un travail ou d'un programme.

La structure du rapport est :



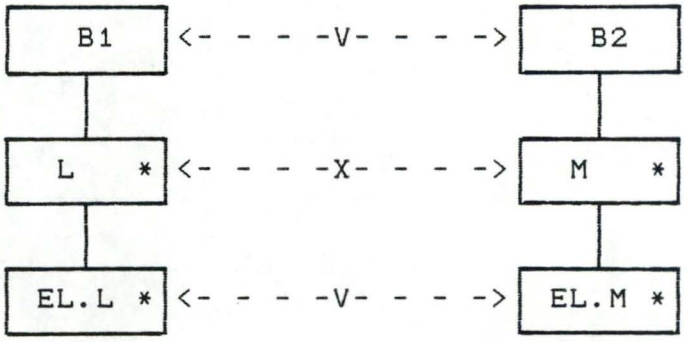
On retrouve sur le rapport tous les enregistrements concernant le même travail dans le même ordre que dans le fichier d'entrée. Tandis que l'ordre entre enregistrements de travaux différents peut être différent sur le rapport, c'est-à-dire que dans l'entrée, des enregistrements du travail 1456 peuvent apparaître après un enregistrement du travail 1457 ou même 1458. Le "load-record" du programme X0453R88 viendra toutefois après le "unload-record" du programme X0453R87. Les enregistrements des différents travaux sont donc entrelacés les uns avec les autres, mais ont conservé leur ordre au sein du travail.

B.3. Conflit de frontière :

Ce type de conflit survient lorsque les deux structures de données possèdent chacune deux ou plusieurs niveaux d'itération et que :

1. les composants d'itération au niveau supérieur correspondent bien,
2. les parties itérées au niveau inférieur correspondent bien, mais
3. il n'existe aucune correspondance entre les composants au niveau intermédiaire.

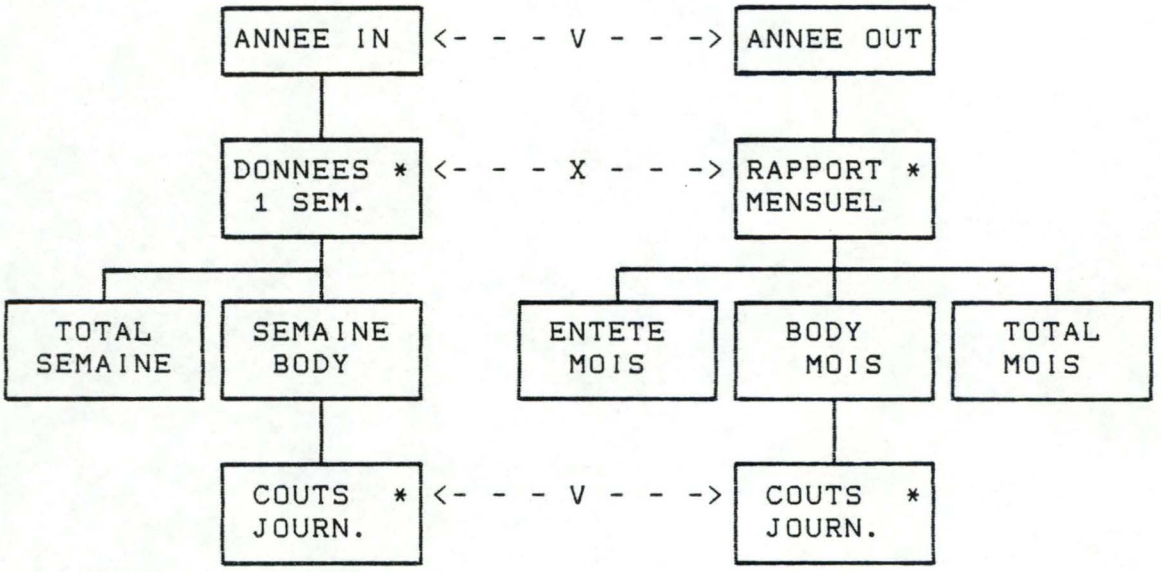
Schéma d'identification :



Exemple : rapport annuel. (Van 't dack, 86)

Une firme désire sortie un rapport portant sur l'année écoulée. Elle désire donner un aperçu des coûts journaliers pour chaque mois ainsi qu'un total mensuel des coûts.
A cet effet, la firme dispose d'un fichier groupant et totalisant par semaine les coûts journaliers.

Les structures de données, avec les correspondances indiquées, se présentent comme suit :



(exemple 1.3)

Il existe une correspondance au niveau le plus haut entre les composants "ANNEE-IN" et "ANNEE-OUT", et au niveau le plus bas entre les composants "COUTS-JOURN-IN" et "COUTS-JOURN-OUT" (contrôler à l'aide des règles de correspondances). Ces règles ne permettent pas d'identifier des correspondances entre composants de niveaux intermédiaires. Nous ne pouvons pas dire que "MOIS" est une

itération de "SEMAINE". Ce qui signifierait en effet que chaque "MOIS" comprend un nombre entier de "SEMAINES" alors que nous savons que la plupart des mois comprennent 4 "SEMAINES" plus une partie d'une "SEMAINE", ou 3 "SEMAINES" et deux parties de deux autres "SEMAINES". Nous pouvons encore moins considérer "SEMAINE comme une itération de "MOIS". Nous ne pouvons non plus trouver un composant "X" qui serait une séquence de "SEMAINE" et de "MOIS" ou une sélection de "SEMAINE" et de "MOIS". Les deux composants "DONNEES POUR 1 SEMAINE" et "RAPPORT POUR 1 MOIS" sont clairement en conflit.

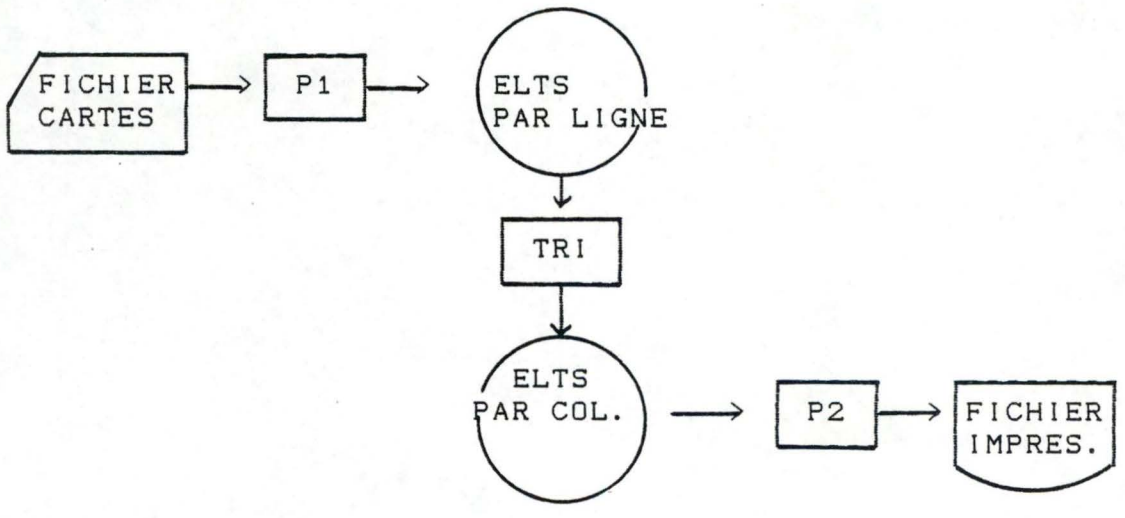
1.4.3.3. Solutions aux conflits de structure

A. Conflit d'ordre

Il existe deux manières de procéder pour résoudre les conflits d'ordre. Le principe des solutions de ce problème repose sur la construction d'un programme ayant deux structures, au lieu d'une. Ces deux solutions sont : l'utilisation d'un programme de tri et l'utilisation d'un fichier à accès direct.

1. Programme de tri

Cette première méthode consiste à introduire un programme de tri entre les deux structures de données initiales. Nous pouvons représenter cette solution par le diagramme de flux du système, incluant le programme de tri entre le fichier d'entrée et le fichier de sortie.

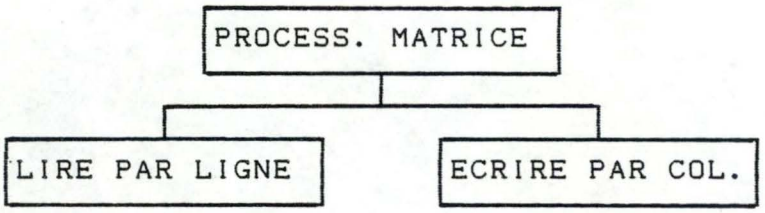


Si nous prenons l'exemple 1.1 de la "Matrice", le programme P1 écrit un fichier d'enregistrements "Elément"; chaque enregistrement porte le numéro de rangée et le numéro de

colonne de l'"Elément". Le programme de tri réordonne ce fichier selon une séquence ascendante de numéro colonne, tout en retenant le numéro de la rangée de l'"Elément". Le programme P2 lit le fichier de sortie du programme de tri, et imprime la matrice dans le fichier de sortie par colonne.

2. Fichier à accès direct

La seconde manière peut être utilisée s'il y a assez de place en mémoire centrale pour tenir la matrice (pour l'exemple 1.1) entière. Nous pouvons structurer le programme comme suivant :



Le programme est une séquence de deux composants : premièrement la matrice est lue par ligne et placée en mémoire centrale, ensuite elle est imprimée par colonne. Le fait que l'accès à la mémoire centrale est direct et non séquentiel rend l'étape de tri non-nécessaire.

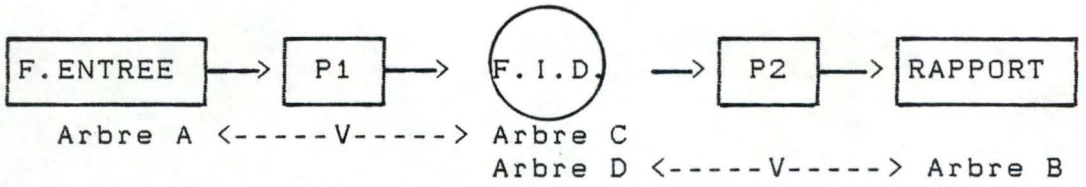
Remarque : cette méthode, tout comme la première, doit lire entièrement la matrice avant de commencer à l'imprimer. La lecture et l'impression sont manipulées par des composants de programme séparés.

B. Conflit de frontière

Les solutions aux conflits de frontière sont de deux types. La première repose sur la création d'un fichier intermédiaire entre les deux fichiers présentant un conflit. La deuxième solution consiste à améliorer cette première version en utilisant le principe de l'inversion.

1. Fichier intermédiaire

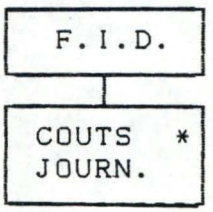
Il s'agit tout d'abord de définir un flux intermédiaire entre le flux des entrées (représenté par l'arbre A) et le flux des sorties (représenté par l'arbre B). Ce flux intermédiaire sera réalisé sous forme d'un fichier appelé fichier intermédiaire des données (F.I.D.).



Sur ce flux intermédiaire, on doit pouvoir placer deux structures différentes (représentées par les arbres) C et D telles que les structures A, C (resp. D, B) soient en correspondance. Ensuite, il ne restera plus qu'à construire deux programmes P1, P2 en se basant sur chacune des correspondances.

P1 : créera le fichier intermédiaire à partir du fichier de données en entrée;
P2 : produira le rapport à partir du fichier intermédiaire.

Cette solution qui consiste à construire deux programmes simples reliés entre eux à l'aide d'un FICHIER séquentiel n'est cependant pas très efficace. En effet, les données du flux intermédiaire ne peuvent être exploitées (par le programme P2) que lorsque le programme P1 a produit l'entièreté du FICHIER intermédiaire.
Pour l'exemple du rapport annuel (exemple 1.3 du paragraphe 1.4.3.2.), la structure du fichier intermédiaire (F.I.D.) serait :



Cette structure peut être mise en correspondance avec, d'une part, le fichier d'entrée FENTREE et d'autre part, le fichier de sortie RAPPORT.

2. Inversion

Une manière d'améliorer cette première solution, c'est de supprimer le fichier intermédiaire et rendre accessible au programme P2 une donnée du flux intermédiaire dès qu'elle est produite.

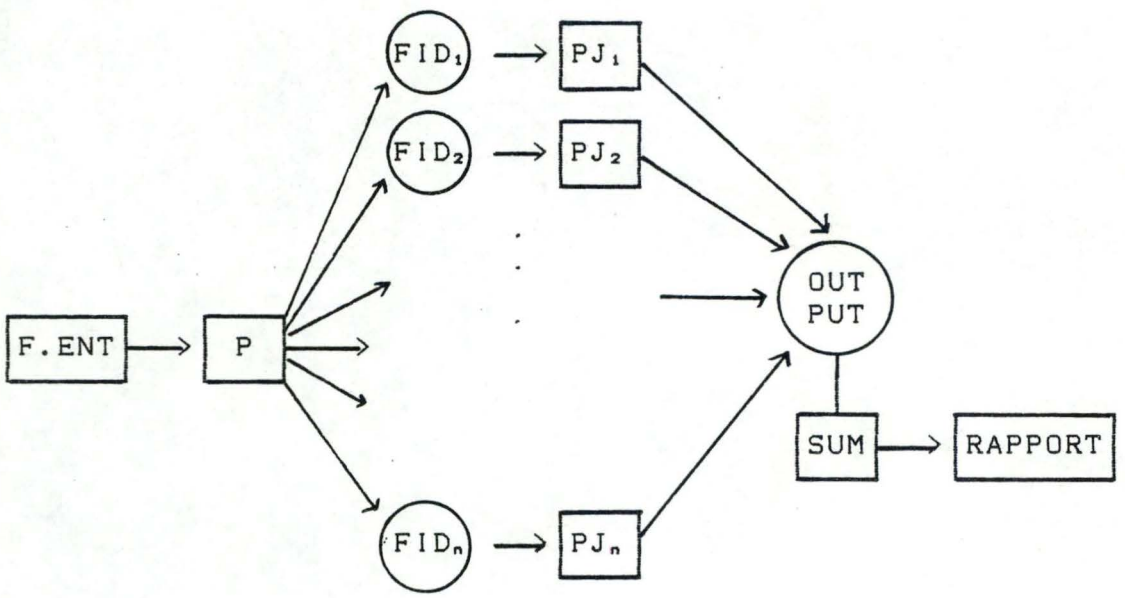
Ainsi on voudrait pouvoir exécuter P1 jusqu'au moment où il aura produit un enregistrement du flux de données intermédiaires, ensuite l'ajourner, exécuter P2 jusqu'au moment où il aura besoin de l'élément suivant du flux intermédiaire, puis ajourner P2 et remettre P1 en marche, etc..

Ceci est possible en appliquant aux programmes P1 et P2 des modifications, utilisant une technique appelée inversion (qui sera exposé au paragraphe 1.4.4.).

C. Conflit d'entrelacement

La résolution de ce type de conflit peut être effectuée de deux manières différentes.

1) la première solution consiste à trier le fichier des données en entrées en regroupant les enregistrements selon leur appartenance à un travail.
2) la deuxième possibilité consiste à diviser Le fichier d'entrée en n flux intermédiaires de données (F.I.D.), un F.I.D. par entité entrelacée. Pour chaque entité, le FID_j est traité séparément dans un programme PE_j.



Le programme P lit le fichier d'entrée et écrit dans des fichiers distincts pour tous travaux différents. Les n programmes PJ_1, PJ_2, ... PJ_n sont alors exécutés : chacun lit son fichier intermédiaire FID_i et écrit les enregistrements concernant son travail dans le fichier OUTPUT. Le fichier OUTPUT

est finalement traité par le programme SUM qui présente le rapport final avec les "entête" et "fin de rapport".

1.4.4. Inversion de programme

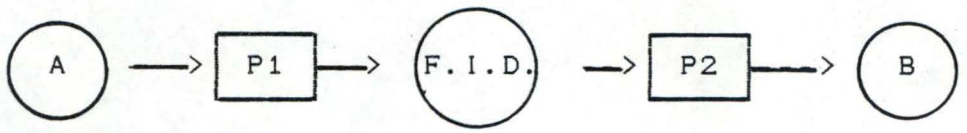
A. Introduction

L'inversion de programme est une méthode d'exécution de programmation et non directement liée à la méthode P.S.J. Les solutions apportées aux conflits de structure grâce à des fichiers intermédiaires restent valables, mais la technique de l'inversion de programme permet de pallier les désavantages de ces fichiers intermédiaires.
L'inversion de programme repose sur le principe de la programmation "parallèle" (utilisation de "coroutines"), c'est-à-dire :

- P1 est exécuté jusqu'au moment où il a produit un enregistrement du flux de données intermédiaire;
- P1 est ajourné et P2 est mis en marche;
- P2 est exécuté jusqu'au moment où il a traité l'enregistrement;
- ensuite, P2 est ajourné et P1 à nouveau mis en marche.

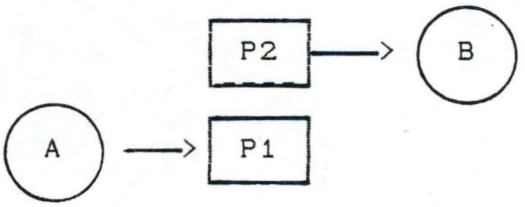
B. Principe

Supposons que pour résoudre le conflit de structure entre les flux A et B, nous avons été amenés à construire deux programmes P1, P2 reliés par un flux de données intermédiaire.

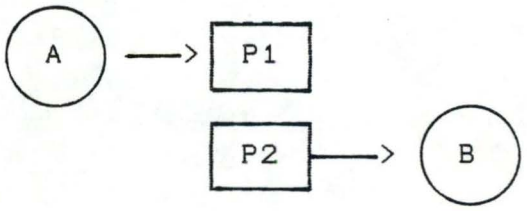


Nous pouvons amélioré cette solution en supprimant le fichier intermédiaire F.I.D. :

- soit en inversant P1 par rapport à la sortie F.I.D., dans ce cas P1 devient un sous-programme de P2 : au lieu de lire directement un enregistrement de F.I.D., P2 appellera le programme P1 qui lui fournira cet enregistrement (amélioration 1);



- soit en inversant P2 par rapport à l'entrée F.I.D., dans ce cas P2 devient un sous-programme de P1 : au lieu d'écrire un enregistrement de F.I.D., P1 appellera le programme P2 auquel il transfèrera cet enregistrement (amélioration 2).



Nous allons montrer et justifier les modifications qu'il faut apporter aux programmes P1 et P2 utilisant le fichier intermédiaire si on opte pour l'amélioration 1 (la démonstration se fait de manière plus ou moins analogue pour l'amélioration 2, il ne nous a pas semblé nécessaire de la reprendre ici).

C. Règles d'inversion et justification

Pour justifier les règles d'inversion appliquées à P1 et P2 il faut donner une "spécification" précise d'un appel de P1 inversé (effectué par P2).

Supposons que l'exécution de P1 non inversé produise un fichier intermédiaire F.I.D. comportant N enregistrements. Alors le sous-programme P1 inversé ($P1^{-1}$) sera appelé N + 1 fois par P2 et le ième appel de $P1^{-1}$ aura l'effet suivant :

- pour i tel que $1 \leq i \leq N$: un indicateur EOF est renvoyé avec la valeur "0"; le ième enregistrement de F.I.D. est fourni au programme P2;
- pour i = N + 1 : l'indicateur EOF est renvoyé avec la valeur "1".

La démonstration de la règle d'inversion est réalisée dans [MATHIEU, 86, chap 3.2.4.2. p72].

DEUXIEME PARTIE

FORMALISME DE JACKSON-HUGHES

PARTIE 2 : FORMALISME DE JACKSON-HUGHES

2.1. Introduction

Nous avons vu dans la première partie que la méthode de programmation structurée de Jackson cherche à présenter une manière systématique de résoudre certaines classes de problèmes : problèmes de base (correspondance des structures), conflits d'identification, conflits de structure, etc. Il est intéressant de se demander pourquoi il a choisi une telle classification des problèmes et de telles méthodes. Une étude plus formelle de la méthode J.S.P. nous permettra d'apporter une réponse à cette question. Nous baserons cette étude sur le formalisme de Hughes. Il explique, en terme de théorie des langages formels, pourquoi les méthodes de base, de backtracking et de conflits de structure sont appliquées.
La présentation critique de ce formalisme (paragraphe 2.3.) nous permettra dans les deux paragraphes suivants, une révision de l'étude théorique de la méthode J.S.P. entreprise par Hughes. Nous tenterons dans le paragraphe 2.4. de poser les limites de ce formalisme à partir desquelles, nous proposerons dans le paragraphe 2.5. certaines extensions (non-exhaustives) possibles. Sur base de cette étude et du formalisme de recherche de correspondance de Hughes, nous chercherons à formaliser le mécanisme de la construction lui-même, plutôt que ses résultats (paragraphe 2.6. Recherche d'un nouveau formalisme). Sur base de ce nouveau formalisme, nous tenterons de répondre aux diverses critiques formulées à l'encontre du formalisme de Hughes et nous exposerons dans le paragraphe 2.7. un certain nombre de règles permettant la construction automatique d'un programme (code pseudo-Pascal) sur base de la correspondance de deux structures de données (structure d'un programme).

Nous débuterons cet exposé par une brève description des machines d'états-finis, ainsi que par leurs relations aux expressions régulières (paragraphe 2.2.) (ces notions seront nécessaires pour la présentation du formalisme de Hughes) et pour terminer cette seconde partie, nous présenterons brièvement des études réalisées par d'autres auteurs portant sur la

formalisation de la méthode J.S.P. (paragraphe 2.8.).

2.2. Machine d'états-finis et expression régulière

[MINSKY, 67]

2.2.1. Introduction

Les Machines d'Etats-Finis (M.E.F.) sont des machines qui procèdent par des étapes "discrètes" (1) séparées clairement d'une étape à l'autre et possède un nombre fini de configurations ou d'états. Ces machines (idéalisées) ont des intérêts spéciaux pour un certain nombre de raisons. De leurs particularités limitées, de nature finie, la structure et le comportement de ces machines sont faciles à décrire complètement, sans aucune ambiguïté ni approximation. La caractéristique marquante de ces machines d'états finis est la simple relation entre leur structure et leur comportement.
Nous présenterons dans ce paragraphe ces machines de manière succinte, ainsi que leurs relations aux expressions régulières (théorèmes de Kleene). Les lecteurs désirant plus de renseignements sur ces machines peuvent consulter [MINSKY, 67, chap 2 et 4].

2.2.2. Description succinte d'une Machine d'Etats-Finis

Du point de vue de l'utilisateur (ou de l'environnement) une machine peut être vue comme une boîte fermée avec un point d'entrée et un point de sortie. Nous appellerons le point d'entrée S (stimulus) et le point de sortie R (réponse). L'entrée S peut se trouver dans n états différents S1, S2, ... Sn et les états possibles de R sont R1, R2, ...Rm. A chaque moment t, l'environnement E détermine un certain état d'entrée S, le signal choisi sera dénoté par S(t). La machine M sélectionne ensuite un certain état de R qui affectera l'environnement E d'une certaine manière. Le choix de la réponse

(1) Le comportement d'une M.E.F. est décrite comme une séquence simple et linéaire d'évènement dans le temps. Ces évènements se déroulent seulement à des "moment" discrets (il ne se passe rien entre deux évènements).

au temps t sera appelé R(t).

Admettons que la Machine M ait reçu un signal de l'environnement E et qu'elle réponde à ce signal, et que, H(t) dénote l'histoire de ce processus au temps t (c'est-à-dire que H(t) décrit d'une certaine manière l'enregistrement de tous les stimulis qui ont influencés H depuis le début). La réponse rj de M au temps (t+1) dépendra, bien sûr, du signal Si mais également des états précédents au temps (t). Ces états précédents sont déterminés par l'histoire de H(t). Nous pouvons dès lors établir la relation F, de la manière suivante :

$$R(t+1) = F(H(t),S(H))$$

Nous supposons que la machine peut distinguer (dans ces comportements présents et futurs) seulement un certain nombre fini de classes d'histoires possibles. Ces classes seront appelées "ETATS INTERNES" de la machine. Nous désignerons l'état interne de la machine, au temps t, par le symbol Q(t), et nous appellerons ces états eux-mêmes q_1, ...q_p.

2.2.3. Définition des fonctions F et G

Les deux fonctions ou relations F et G que nous décrivons dans ce paragraphe, nous permettent de donner une description complète de la machine, tout en restant discret sur ce qui se passe à l'intérieur de la machine.

(1) fonction F

Par notre définition des états internes, la réponse de la machine R(t+1) au stimulus S(t) dépend également de Q(t) c'est-à-dire, de l'état interne distinct au temp t. Nous pouvons formaliser cette dépendance par :

$$R(t+1) = F(Q(t),S(t)) \qquad (F)$$

L'état d'entrée ou le signal S(t) dépend, naturellement, de l'environnement. Mais de quoi dépend l'état interne Q(t) ? :

(2) fonction G

L'état interne au temps t+1 de la machine Q(t+1) dépend seulement de son entrée précédente S(t) et de son état interne précédent Q(t). Q(t+1) ne dépend de rien de plus. Nous pouvons

exprimer cette fontion de la manière suivante :

$$Q(t+1) = G(Q(t),S(t)) \qquad (G)$$

Les machines caractérisées par un nombre fini d'états internes seront appelées "Machines d'Etats-Finis".

2.2.3. Expressions régulières

Les réseaux de [McCULLOCH, 43] décrivent des classes de séquences d'éléments reconnues par des M.E.F. Devant la complexité de ces réseaux, la formulation de Kleene décrit de telles classes de manière plus élégante en utilisant des formules qu'il appelle "expressions régulières" et un nombre de notions associées. Chaque expression régulière servira à représenter un certain ensemble de séquences de signaux. De tels ensembles sont appelés ensemble régulier de séquences.

La définition récursive d'une expression régulière est :

soit K : une classe d'expressions régulières;

BASE : chaque symbole a, b, c, .. est une expression régulière de K;

RECURRENCE :
- si E et F sont des expressions de K, alors (E.F) l'est aussi;
- si E1, E2, ..,En sont des expressions de K, alors (E1 | E2 | .. En) l'est aussi;
- si E est une expression de K, alors E* l'est aussi;

RESTRICTION : seules les expressions générées par les règles ci-dessus sont des expressions de K.

2.2.4. Théorèmes de KLEENE

Nous considérons une machine M qui commence dans un certain état Q_{init} et on se demande "Quelles séquences de symboles d'entrée mènent la machine M à terminer dans un certain ensemble d'états $\{Q_{fin}\}$?" Chaque séquence possédant cette propriété est dite "reconnaissable" par la machine. L'ensemble de toutes ces séquences est l'ensemble des séquences reconnaissables par la machine M. Les théorèmes de Kleene montrent que les ensembles reconnaissables par des machines d'états-finis sont des ensembles réguliers de séquences.

Nous divisons la preuve en deux parties :

Théorème 1 : "Une machine d'états-finis peut reconnaître seulement des ensembles réguliers de séquences."

La preuve est basée sur l'induction mathématique du nombre d'états de la machine. On montre premièrement que si le théorème est vérifié pour toutes les machines ayant n états, alors il est vérifié pour toutes machines possédant n+1 états. On montre ensuite que le théorème est correct pour toutes machines possédant un seul état.

Théorème 2 : "Tout ensemble régulier peut être reconnu par une machine d'états-finis."

Ces deux théorèmes montrent que les ensembles reconnus par des machines et des ensembles représentés par des expressions régulières sont équivalents (équivalence entre la notion d'ensemble régulier et ensemble reconnaissable par des machines d'états-finis).

2.3. Formalisme de HUGHES

2.3.1. Introduction

Nous présentons dans ce paragraphe le formalisme de Hughes concernant la méthode de Jackson auquel nous apportons une analyse critique. Ce formalisme présenté par Hughes est basé sur la théorie des langages formels. Nous définirons dans le paragraphe 2.3.3. la relation entre les arbres de Jackson et les expressions régulières. Dans le paragraphe 2.3.4. nous exposerons le concept de 'correspondance' formalisé par Hughes entre deux expressions régulières représentant des arbres de Jackson.

Le but de ce formalisme est de permettre la distinction entre la méthode de base (cfr. paragraphe 1.1.) et les autres méthodes. Pour rappel, la méthode de base s'identifie aux problèmes où les structures des données peuvent être mises en correspondance, tandis que les autres méthodes (que nous appellerons méthodes de résolution des conflits) correspondent

aux problèmes de conflits de structure (d'ordre, d'entrelacement et de frontière) et aux problèmes d'identification des conditions de sélection ou d'itération par une lecture d'une seule donnée (courante).

Sur base du paragraphe 2.2., nous montrerons que la correspondance entre deux expressions régulières peut s'identifier à une fonction exécutable par une machine d'états-finis. A partir des caractéristiques de ces machines (paragraphe 2.3.5.), nous pourrons expliquer la différence entre la méthode de base (paragraphe 2.3.6.) et les méthodes de conflits (paragrahe 2.3.7.).

Les différents concepts du formalisme de Hughes sont développés dans le cadre de la méthode de base dont nous rappelons les hypothèses :

- un seul fichier séquentiel de données en entrée;
- un seul fichier séquentiel de données en sortie;
- déterminisme de la structure d'arbre d'un programme;
- correspondance des deux structures d'arbre.

Remarque : Le caractère déterministe d'une structure d'arbre représente la sélection d'une seule possibilité de choix d'un sous-somposant, ainsi que la possibilité d'identification des conditions de sélection ou d'itération d'un sous-composant par l'information d'un seul enregistrement (courant).

Les illustrations des différents concepts se baseront sur l'exemple donné dans [HUGHES, 79]. L'énoncé de cet exemple sera repris dans le paragraphe 2.3.2.

2.3.2. Enoncé de l'exemple de référence :

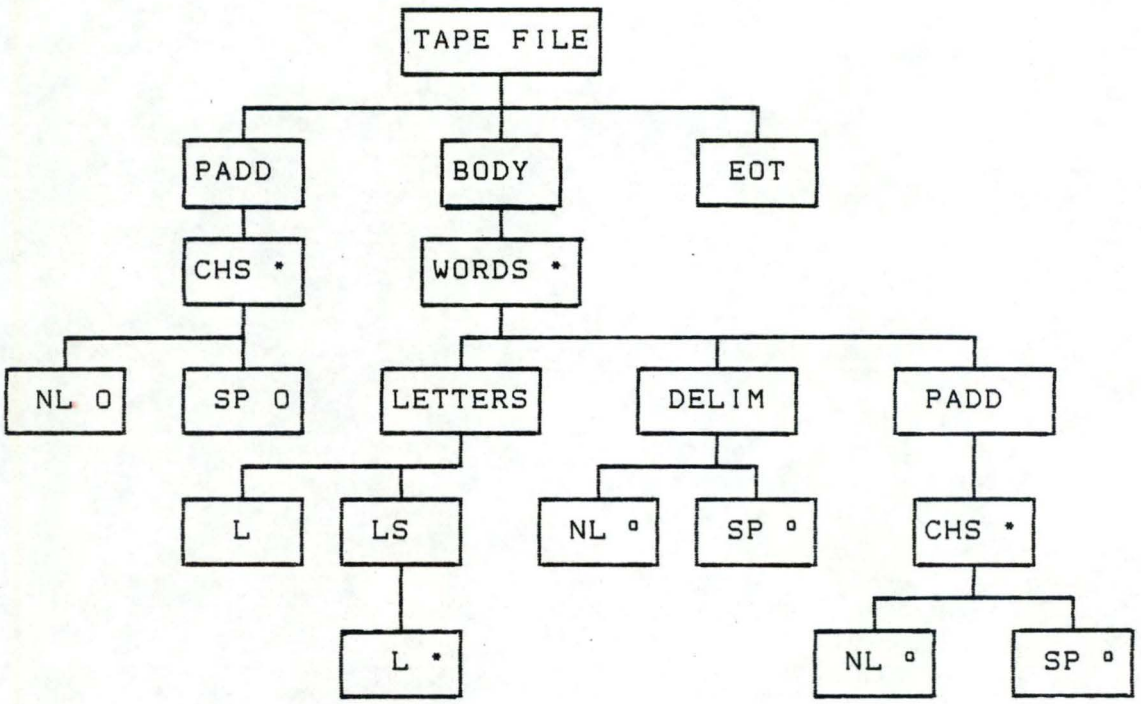
"Traitement de texte" [HUGHES, 79]

"On demande d'élaborer un programme qui lit un fichier de caractères et le recopie dans un fichier de sortie suivant certaines modifications. Le fichier texte d'entrée "TAPEFILE" est composé de caractères alphabétiques, de caractères "blanc" (espaces) et des caractères de "nouvelle ligne". Le fichier de sortie "OUTPUTFILE" est composé de caractères alphabétiques et de caractères de "fin de mot" (espace). La différence entre les

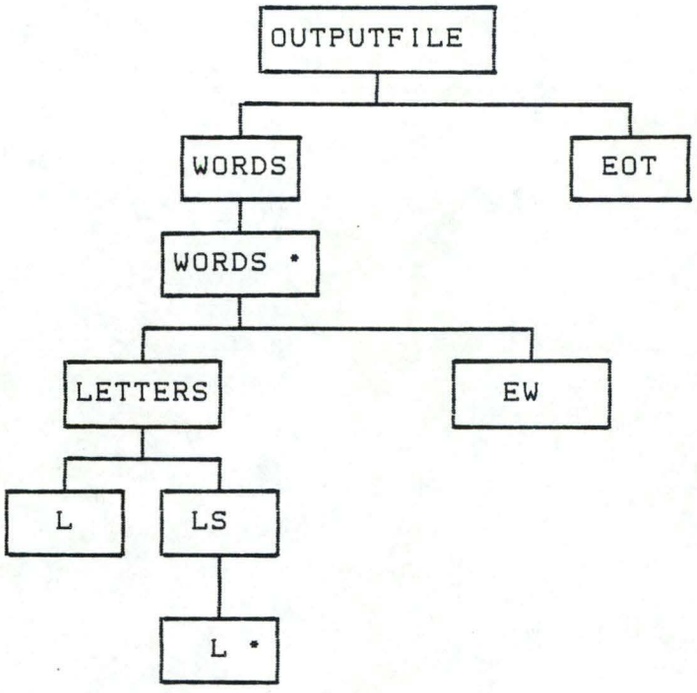
deux fichiers est que les mots (ensembles continus de caractères alphabétiques) ne sont séparés que d'un seul blanc (caractère de fin de mot) dans le fichier de sortie, tandis que les mots dans le fichier d'entrée peuvent être séparés par un ou plusieurs caractères non-alphabétiques (blanc et/ou nouvelle-ligne)".

Les structures d'arbre (de Jackson) les plus appropriées sont :

- pour les données d'entrée :



- pour les données de sortie :



Abréviation :

SP : espace;
NL : nouvelle ligne;
L : lettre;
EOT : fin de texte;
EOF : fin de fichier;
EW : fin de mot.

2.3.3. Définition des arbres de Jackson

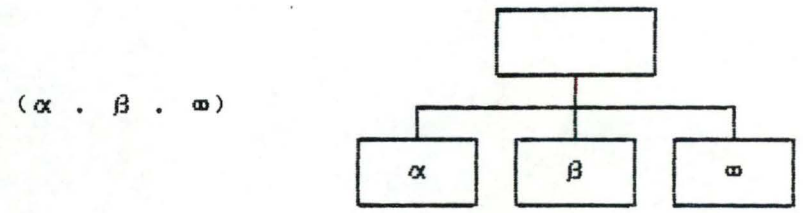
Les ensembles des données en entrée et en sortie peuvent être vus comme des langages (éventuellement infinis). Dans la méthode de Jackson, nous allons montrer qu'un arbre donne une représentation finie d'un langage et qu'il est capable de représenter seulement des expressions regulières. Les arbres de Jackson ne sont, dès lors, qu'une notation alternative pour des expressions régulières. En fait, il n'est capable de représenter que des formes restrictives d'expressions régulières, dans lesquelles, il n'y a pas de précédence des opérateurs ("." : concaténation; "/" : sélection; "*" : itération). L'ambiguïté est supprimée par l'utilisation de parenthèses. Avec cette restriction, l'ensemble de tous les arbres de Jackson peut être mis en correspondance (1 à 1) avec l'ensemble de telles expressions régulières appropriées de la manière suivante :

1. Un symbole terminal simple est représenté par un arbre composé d'un noeud (composant-élémentaire), libellé par un symbole (fig. 1) :

symbole terminal arbre de Jackson

(figure 1)

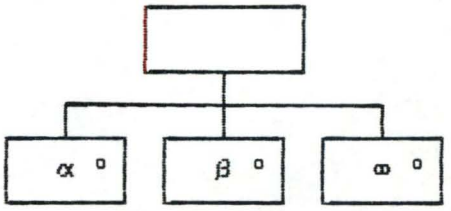
2. Une expression qui est une concaténation de symboles ou expressions entre parenthèses est représentée par un noeud (composant-séquence) avec une séquence ordonnée de fils. Le Nième fils est la racine d'un arbre représentant le ième symbole ou expression entre parenthèses concaténé. Dans les figures suivantes, les lettres grecques représentent soit des symboles terminaux, soit des expressions régulières.



expression régulière arbre de Jackson

(figure 2)

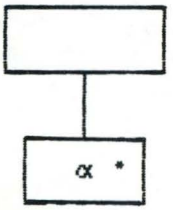
3. Une expression qui est une union de symboles ou de sous-expressions entre parenthèses est représentée par un noeud (composant-sélection) avec un ensemble non-ordonné de fils dans lequel, chaque fils est la racine d'un arbre représentant un symbole ou une sous-expression entre parenthèses, dans l'union, et il y a un fils représentant chaque alternative. Chaque fils de l'alternative est marqué avec le symbole ° (fig. 3)



expression régulière — arbre de Jackson

(figure 3)

4. Une expression qui est une itération d'un symbole ou d'une sous-expression entre parenthèses est représentée par un noeud (composant-itération) avec un fils libellé par "*". Le fils est la racine d'un arbre représentant un symbole ou une sous-expression entre parenthèses itéré (fig. 4), seul les arbres de cette forme sont utilisés.



expression régulière — arbre de Jackson

(figure 4)

Exemple : Transformation d'un arbre de Jackson en expression régulière : "Traitement de texte"
(Le développement de cet exemple est présenté dans l'annexe 3.)

2.3.4. Définition de la 'correspondance' entre les arbres d'entrée et de sortie

2.3.4.1. Introduction

Hughes exige que la correspondance soit identifiée entre une paire d'arbres pour baser sa structure de programme sur celle des données en entrée. Cette correspondance signifie l'existence d'une correspondance à tous les niveaux, de la racine jusqu'au niveau des feuilles des deux arbres, en passant par tous les

composants de niveaux intermédiaires.

En terme d'expressions régulières, ceci implique des correspondances entre symboles ou sous-expressions entre parenthèses dans les expressions régulières d'entrée et de sortie (les symboles d'une expression régulière correspondent aux feuilles des arbres et les sous-expressions entre parenthèses correspondent aux composants des niveaux supérieurs aux feuilles).

Nous pouvons distinguer deux types de correspondance. La première, nous l'appellerons "correspondance formelle". Elle est obtenue par application des règles de Hughes définies au paragraphe 2.3.4.3. Ces règles permettrons de définir plusieurs correspondances formelles entre deux expressions régulières. Nous entendrons, dés lors, par "correspondance effective", la correspondance formelle qui parmi toutes les autres, satisfait aux spécifications du programme.

Nous présenterons dans les paragraphes 2.3.4.2. et 2.3.4.3. le concept de 'correspondance' utilisé dans le formalisme de Hughes et nous expliquerons plus spécialement le concept de "traduction" dans le paragraphe 2.3.4.4. Nous nous permetterons dans le paragraphe suivant (2.3.4.5.) de porter un jugement critique sur la définition formelle proposée par Hughes.

2.3.4.2. Présentation informelle de la définition de la correspondance de Hughes

Hughes définit la correspondance comme étant une "traduction" des noeuds des arbres d'entrée vers des noeuds des arbres de sortie. Pour la classe de problèmes envisagée (correspondance des structures), la "traduction" d'un noeud d'entrée vers un noeud de sortie dépend seulement de ses descendants; ainsi, la génération de la correspondance de deux arbres procéde de manière récursive ou bottom-up c'est-à-dire, pour que des noeuds correspondent, leurs descendants doivent correspondre.

Si on peut démontrer, pour cette classe de problèmes, la correspondance au niveau des feuilles des arbres (c'est-à-dire, en terme d'expression régulière, au niveau du vocabulaire des deux expressions), on aura démontré la correspondance générale des deux arbres.

Remarque : nous rappelons que les problèmes ayant un conflit de

frontière ne font pas partie de la classe de problèmes considérée.

2.3.4.3. Définition formelle de la correspondance proposée de Hughes :

Nous avons déjà montré que les arbres de Jackson sont des alternatives des expressions régulières. Pour montrer la correspondance formelle des arbres, Hughes montre la correspondance formelle entre les deux expressions régulières respectives des arbres d'entrée et de sortie.

soit : - Σ : l'ensemble constituant le vocabulaire d'entrée (composant-élémentaire ou symbole terminal).
- Ω : l'ensemble constituant le vocabulaire de sortie (composant-élémentaire ou symbole terminal).
- I : l'expression régulière correspondant à l'arbre d'entrée.
- O : l'expression régulière correspondant à l'arbre de sortie.

Hughes définit la correspondance formelle entre deux expressions régulières de la manière suivante :

"Les deux expressions I et O correspondent si et seulement si il existe une fonction "output" telle que :

O = output(I)

L'existence de cette fonction "output" peut être dérivée par application des règles suivantes si R et Q sont des expressions régulières :

(i) $R \in \Sigma \Rightarrow output(R) \in \Omega \cup \{E\}$, où E représente l'ouput "vide";
(ii) $output(R.Q) = output(R).output(Q)$
(iii) $output(R|Q) = output(R)|output(Q)$
(iv) $output(R^*) = (output(R))^*$
(v) $R.E = E.R = R$
(vi) $R|Q = Q|R$
(vii) $R|R = R$
(viii) $(R^*)^* = R^*$
(ix) $E^* = E$ "

La correspondance formelle O = output(I) signifie qu'à chaque feuille de l'arbre des entrées (appartenant à Σ) on associe par la fonction output un élément de (Ω U {E}) (un composant-élémentaire de l'arbre des sorties ou l'élément vide (E)). Cette fonction "output" doit être vérifiée pour tous flux de données appartenant à l'expression d'entrée I c'est-à-dire, que de n'importe quelles représentations de I, on peut obtenir une représentation de O, en effectuant le programme de l'arbre aux feuilles duquel on a attaché les opérations correspondantes à la fonction "output".

2.3.4.4. "Traduction"

Le concept de "traduction" est basé sur la fonction Output définie précédemment. La traduction d'un noeud d'un arbre de Jackson vers un autre représente une fonction d'une sous-expression de l'expression d'entrée vers une sous-expression de l'expression de sortie.

Soit R = symbole terminal ou sous-expression entre parenthèses d'entrée;
S = symbole terminal ou sous-expression entre parenthèses de sortie;

l'image de la fonction Output de tout flux de données appartenant à l'expression R appartient à l'expression S. Les règles de traduction seront définies au niveau du vocabulaire des deux arbres, c'est-à-dire au niveau des composants-élémentaires :

pour tout i : $Output(a_i) = b_i$ avec : $a_i \in \Sigma$;
$b_i \in \Omega^* \cup \{E\}$.

2.3.4.5. Analyse du formalisme de Hughes

La critique majeure que l'on peut formuler à l'encontre de ce formalisme est le manque de précision de la définition formelle concernant la fonction de "traduction" OUTPUT. Nous constatons un manque d'explication nous permettant d'interpréter de manière formelle cette définition. Nous pouvons détailler cette critique en quatre points :

A. Manque de précision de la sémantique de la fonction "output qui est à la base de la définition de la correspondance et des propriétés (de dérivation) de cette fonction (règles

(ii), (iii) et (iv)). Cette fonction 'output' s'applique parfois au langage entier, parfois au sous-langage ou encore à d'autres éléments (cfr. point 2.3.6.).

B. La règle (i) est également peu précise : est-il suffisant d'avoir toutes les égalités S = output(R) avec S ∈ AS et R ∈ AE pour déterminer la correspondance ? (AS, AE : ensembles des composants élémentaires des deux structures).

C. Si l'on concidère que la règle (i) définit la correspondance au niveau du vocabulaire des deux expressions régulières, la fonction de "traduction" d'un langage d'entrée vers un langage de sortie ne permet pas de définir (formellement) les correspondances aux niveaux des composants intermédiaires.

D. Le formalisme de Hughes ne propose aucune règle permettant de prendre en compte la sémantique du problème traité, afin de déterminer la correspondance "effective". L'ensemble des règles {(i), .., (ix)} ne permet pas de trouver la "bonne" correspondance.

2.3.5. Machine d'états-finis et le formalisme de Hughes

A. Nous avons donné dans le paragraphe 2.2. les caractéristiques d'une machine d'états-finis ainsi que leur relation avec les expressions régulières. Kleene a montré qu'il existe pour chaque ensemble régulier de séquences, une machine d'états-finis qui reconnaît précisément cet ensemble. Il a ainsi démontrer que les ensembles reconnus par ces machines et les ensembles représentants des expressions régulières sont équivalents.

Avant de montrer que la méthode de base est applicable seulement aux fonctions d'une machine d'états-finis, Hughes présente de manière informelle comment une machine d'états-finis peut, à partir d'une expression régulière représentant un arbre de Jackson, engendrer une autre expression régulière.

Une machine d'états-finis est définie par un sextuplet $(S, \Sigma, \Omega, \xi, \tau, q1)$ où :

- S est un ensemble (d'états) fini et non-vide;
- Σ est un alphabet d'entrée (impulsion d'entrée);
- Ω est un alphabet de sortie (impulsion de sortie);
- δ est une fonction de S X Σ vers S (fonction d'état suivant);
- τ est une fonction de S X Σ vers Ω^* (fonction de sortie);
- q1 est un élément distinct de S (l'état de départ).

La machine est capable de "traduire" des strings d'un langage d'entrée (un sous-ensemble de Σ^*) en des strings d'un langage de sortie (sous-ensemble de Ω^*). Elle commence dans un état q1 et examine le premier symbole du string d'entrée. τ détermine le symbole de sortie pour cette entrée et δ l'état suivant. Dans ce nouvel état, le symbole d'entrée suivant est analysé, pour ce faire, δ et τ sont appliqués à nouveau. Cette procédure s'arrête uniquement lorsque toutes les entrées sont traitées ou une erreur détectée. Ces machines d'états-finis peuvent être représentées par des diagrammes de transition dont les noeuds représentent l'ensemble S et les flèches les fonctions δ et τ.

Exemple : considérons l'arbre d'entrée de Jackson "TAPE FILE" et l'expression régulière correspondante :

((NL/SP)*.((L.(L)*).(NL/SP).(NL/SP)*)*.EOT)

ainsi que l'arbre de sortie "OUTPUT FILE" :

((L.L*).EW)*.EOF

Nous pouvons, sur base des deux expressions d'entrée et de sortie caractérisant les impulsions d'entrée et de sortie, représenter la machine d'états-finis P "traduisant" le string d'entrée en string de sortie.

P = {S, Σ, Ω, δ, τ, q1}

S = {1, 2, 3}

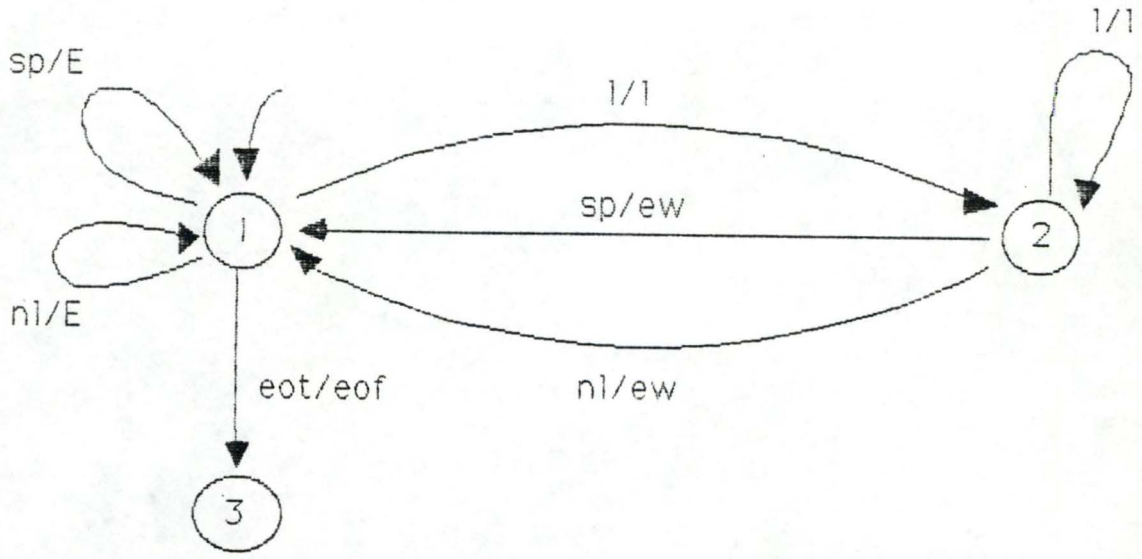
Σ = {nl, sp, l, eot}

Ω = {(1, sp, l), (1, nl, 1), (1, l, 2),
(1, eot, 3), (2, l, 2), (2, sp, 1),
(2, nl, 1)}

τ = {(1, sp, s), (1, nl, s), (1, l, l),
(1, eot, eof), (2,l,l), (2, sp, ew),
(2,nl,ew)}

q1 = 1

Le diagramme de transition correspondant est :



Dans l'état 1, le composant PADD (cfr arbre de Jackson : p. 83 et 175) est transformé en un composant vide "E"; la pemière lettre du composant "WORD" modifie l'état de la machine P (état 1 -> état 2) pour être en position d'écriture de "lettres" : "L". Dans l'état 2, l'itération de lettres restantes (L*) est acceptée. L'occurrence d'un espace "SP" ou d'une nouvelle ligne "NL" modifie l'état 2 en état 1 et "traduit" le caractère de sortie correspondant "EW". Le caractère "EOT" dans l'état 1 fait passer la machine dans l'état 3 et "traduit" le caractère de sortie "EOF".
Nous pouvons remarquer que cette machine d'états-finis "traduit" un langage d'entrée défini par l'expression régulière ((NL/SP)*.((L.(L)*).(NL/SP).(NL/SP)*)*.EOT) en un langage de sortie défini par (L.L*).EW)*.EOF .

Cette machine P exécute les mêmes fonctions que le programme correspondant à la méthode de base de Jackson. Les libellés sur les flèches du diagramme de transition montrent les correspondances au niveau des feuilles de l'arbre de Jackson, tandis que l'état courant sert de "marqueur de position" des composants-noeuds dans l'arbre d'entrée (l'état 1 (resp. 2 et 3) mentionne l'arbre engendré par le sous-composant PADD (resp. BODY et EOF) (cfr. structure de l'arbre TAPEFILE p 175)).

B. Critique de la présentation de la M.E.F.

Si Hughes définit la M.E.F. relative à la structure de correspondance de l'exemple "Traitement de Teste" par le sextuplet $(S, \Sigma, \Omega, \xi, \Gamma, q1)$, il n'apporte aucune précision sur la manière dont il la construit à partir de la correspondance des expressions régulières. Il nous est dès lors difficile de retrouver la correspondance entre cette machine et la méthode de Jackson.

C. [Ginsberg, 65] donne les caractéristiques des Machines d'Etats-Finis, à partir desquelles, Hughes peut montrer, (cfr. paragraphe 2.3.6.), que tout problème résolu par la méthode de base de Jackson est une fonction exécutable par une Machine d'Etats-Finis.

Ces caractéristiques sont :

$F : \Sigma^* \rightarrow \Omega^*$ est une fonction exécutable par une M.E.F. si et seulement si :

1. F préserve le sous-mot initial (si u est un sous-mot de v, où u, v $\in \Sigma^*$ alors F(u) est un sous-mot de F(v).
2. F à des sorties limitées ($\exists$ un entier M tel que $|f(wa)| - |f(w)| \leq M$ pour tout $w \in \Sigma^*$ et $a \in \Sigma$)
3. f(E) = E (le mot-vide ne génère aucune sortie)
4. $f^{-1}(R)$ est régulier pour tout ensemble régulier R (si la sortie est régulière, alors chaque entrée qui génère la sortie régulière R doit aussi être régulière).

2.3.6. La méthode de base est applicable seulement à des fonctions exécutables par des Machines d'Etats-Finis

En utilisant les quatre propriétés caractéristiques d'une Machine d'Etats-Finis (M.E.F.), Hughes montre que si un problème est résolvable par la méthode de base de Jackson alors, il représente une fonction exécutable par une M.E.F. entre un ensemble régulier déterministe et un ensemble régulier.

Soit : F : L(I) -> L(O) représente un programme dérivé de la méthode de base de Jackson où I est une expression régulière déterministe telle que :

O = Output(I)

Pour chaque expression régulière R définie par un vocabulaire, nous définissons un ensemble first(R) :

$$\begin{aligned} first(R) = \{a \mid\ & R = a\ v \\ & (R = A^* \wedge a \in first(A))\ v \\ & (R = R_1 \mid R_2 \mid .. \mid R_w \wedge a \in \bigcup_{i=1..\omega} first(R_i))\ v \\ & (R = R_1 . R_2 .. R_w \wedge a \in first(R_1))\ \} \end{aligned}$$

1. Dans l'idée de prouver la première propriété "préservation des sous-strings initiaux", on prouve par induction le lemme suivant :

   " $\forall$ expressions d'entrée, I, first (output(I)) = output(first(I)) "

Considérons quatre cas :

(a) I est un symbole terminal :

$$\begin{aligned} & I \in \Sigma : first(I) = \{I\} \\ & \quad output(I) \in \Omega \cup \{E\} \\ & \quad \Rightarrow first(output(I)) = \{output(I)\} \\ & \qquad\qquad\qquad\qquad = \{output(first(I))\} \end{aligned}$$

(b) I est une expression séquence :

$$\begin{aligned} & I = I1.I2...Iw \\ & first(I) = first(I1) \\ & output(I) = output(I1).output(I2) \ldots output(Iw) \\ & first(output(I)) = first(output(I1)) \\ & \qquad = \{output(first(I1))\} \\ & \qquad\qquad\qquad\qquad \text{par induction} \\ & \qquad = \{output(first(I)\} \end{aligned}$$

(c) I est une expression sélection :

$$\begin{aligned} & I = I1 \mid I2 \mid .. \mid Iw \\ & first(I) = \bigcup_{i=1..\omega} first(Ii) \\ & output(I) = output(I1) \mid .. \mid output(Iw) \\ & first(output(I)) = \bigcup_{i=1..\omega} first(output(Ii)) \\ & \qquad = \bigcup_{i=1..\omega} \{output(first(Ii))\} \\ & \qquad\qquad\qquad\qquad \text{par induction} \\ & \qquad = \{output(first(I)\} \end{aligned}$$

(d) I est une expression itération :

$I = A^*$ first(I) = first(A)
output(I) = output(A)*
first(output(I)) = {first(output(A))}
= {output(first(A)}
par induction
= {output(first(I)}

Donc, pour $a \in \Sigma$ et $v = a.\alpha \in \Sigma^*$, $f(v) = f(a).\beta$.
Par induction si u est un sous-string initial de v => f(u) est un sous-string initial de f(v).

2. La fonction f à des sorties limitées.
Du point (1) ci-dessus, f(w.a) = f(w).output(a).
Mais si $a \notin \Sigma$, alors output(a) $\in \Omega \cup \{E\}$ et

$$|f(w.a)| - |f(w)| \leq 1.$$

3. f(E) = E
Par définition $E \notin \Sigma$ ainsi f(E) peut être définie trivialement par E. Cette règle empêche la génération spontanée de sortie sans entrée correspondante.

4. Les arbres d'entrée et de sortie peuvent seulement représenter des expressions régulières, ainsi, chaque programme utilisant la méthode de base préserve les ensembles réguliers.

Sur base de ces quatre caractéristiques, Hughes conclut que, par la formalisation donnée pour les définitions des arbres de Jackson et de la correspondance, les problèmes résolvables par la méthode de base de Jackson représentent une fonction exécutable par une M.E.F. entre ensembles réguliers.
Remarque : nous pouvons formuler la même critique concernant la fonction 'output' faite dans le paragraphe 2.3.4.5.

2.3.7. Machines d'états-finis et conflits de structure

2.3.7.1. Introduction

Nous avons présenté dans la première partie, un nombre de problèmes qui ne sont pas soumis à la méthode de base. Jackson les identifie comme des conflits de structure et les classe en problèmes d'identification des conditions d'itération ou de sélection (backtracking), conflits d'ordre, conflits

d'entrelacement et conflits de frontière.
Le résultat du paragraphe précédent est la correspondance entre la méthode de base et les M.E.F. Nous présenterons brièvement dans ce paragraphe les différentes classes de problèmes en analysant leur correspondance à une fonction exécutable par une M.E.F.

2.3.7.2. Problèmes d'identification et backtracking

Un problème de Backtracking est identifié lorsque l'expression régulière d'entrée est non-déterministe.

Exemple : Un programme doit valider une liste de caractères suivant le dernier caractère. Si la liste se termine par B, la liste est valide, si elle se termine par M, la liste est invalide. Ces opérations de sortie sont différentes suivant le résultat :

$$P : liste \rightarrow RESULT$$

$$\text{où} \quad liste = (N^*.b) \mid (N^*.m)$$
$$RESULT = (b/N^*)$$

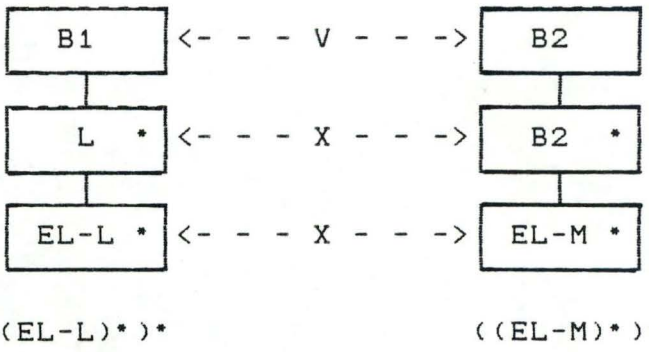
$$\text{et} \quad P(N^*.b) = b$$
$$P(N^*.m) = N^*$$

La première propriété du sous-string initial d'une M.E.F. est violée : la fonction P(N) dépend de ce qui suit. La méthode de base n'est théoriquement pas capable de résoudre ce type de problème.

2.3.7.3. Problèmes de conflits d'ordre et d'entrelacement

A nouveau ces problèmes ne sont pas des fonctions exécutables par des M.E.F., car la propriété du sous-string initial est violée.

Exemple : transposition de matrices

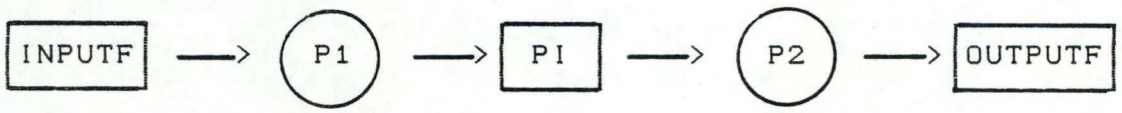


Le sous-string (EL-L)* initial d'entrée n'est pas préservé par la fonction de "traduction" F. (EL-L)* est un sous-string initial de ((EL-L)*)* mais F[((EL-L)*)] n'est pas un sous-string de F[((EL-L)*)*] = ((EL-M)*)*.

Nous avons vu que ces types de conflits peuvent être résolus soit par un programme de tri, soit par l'utilisation d'un fichier intermédiaire. (paragraphe 1.4.3.)

2.3.7.4. Conflits de frontière

Le conflit de frontière se caractérise par une correspondance au niveau des feuilles et un conflit entre des composants de niveaux intermédiaires. De tels problèmes peuvent être résolus par une composition fonctionnelle des deux processus.



Chacun des processus P1 et P2 est résolu par la méthode de base (paragraphe 1.4.3.4.).

L'application des règles de déduction de Hughes ne nous permet pas de détecter de tels conflits. En outre, la propriété d'un sous-string initial paraît être respectée. Le formalisme de Hughes ne nous permet donc pas de différencier ce conflit avec la méthode de base. Cependant Hughes montre que théoriquement, si un problème de conflits de frontière peut être résolu en utilisant sa méthode, il peut être résolu par la méthode de base seule. La preuve utilise le fait que la composition de plusieurs

M.E.F. est une M.E.F. Nous avons donné deux programmes P1 : I1 -> O1 et P2 : I2 -> O2 générés par la méthode de base et on souhaite prouver que le programme P2∘P1 est aussi résolu par la méthode de base.

On sait que :

- I1, O1, I2, O2, sont réguliers;
- I1 et I2 sont non-déterministes;
- P2 et P1 sont des fonctions exécutables par une M.E.F.

Et on doit montrer que :

1. I1 sont des données déterministes et régulières;
2. H∘G : I1 -> X (où X = Range(H | O1 ∩ I2)⊆ O2 est une fonction M.E.F.
   Démarche de la preuve : H∘G est une fonction M.E.F. puisque G et H sont chacune une fonction M.E.F. H reçoit I2 mais seulement après réception de O1, dés lors H "traduit" seulement les données I2 ∩ O1 donnant X⊆ O2.
3. X est régulière.
   Démarche de la preuve : H∘G est une fonction M.E.F.; les M.E.F. préserve les ensembles réguliers; I1 est régulier, dés lors X est régulier.

Remarque : La démonstration proposée par Hughes ne nous paraît pas convaincante car s'il a démontré qu'un programme réalisable par la méthode de base de Jackson est une fonction exécutable par une M.E.F., il n'a pas montré la réciproque c'est-à-dire, qu'une fonction exécutable par une M.E.F. n'est pas nécessairement réalisable par la méthode de base de Jackson. La composition H∘G de deux M.E.F. représentant des fonctions réalisables par la méthode de base, est une nouvelle M.E.F., mais cette nouvelle M.E.F. n'est pas nécessairement réalisable par la méthode de base de Jackson.

2.3.8. Conclusion

La méthode de Jackson concernant la conception de programmes a été examinée en relation avec les Machines d'Etats-Finis. Hughes a montré que la méthode de base est applicable seulement aux fonctions exécutables par une M.E.F. (les langages d'entrée sont définis par des expressions régulières déterministes dans lesquelles des parenthèses sont utilisées pour dénoter la précédence des opérateurs). Cette méthode de résolution correspond à la décomposition d'une fonction exécutable en une

fonction exécutable par une M.E.F. Les problèmes ayant un conflit de frontière sont également des fonctions exécutable par une M.E.F. et sont donc théoriquement résolvables en utilisant la méthode de base. Par contre, les problèmes d'identification, de conflits d'ordre et de conflits d'entrelacement ne le sont pas car ils ne respectent pas les propriétés caractéristiques d'une M.E.F.

2.4. Limites du formalisme de Hughes

2.4.1. Le vocabulaire

Une des caractéristiques du formalisme de Hughes est la notion de vocabulaire. cette notion se retrouve tant dans la définition de la correspondance que dans la définition de la machine d'états finis. Du côté des expressions régulières, on caractérise les données en entrée et en sortie chacune par un vocabulaire définissant également leur propre langage. Cette notion de vocabulaire est aussi présente dans les définitions d'une machine d'états-finis (le sextuplet [S, Σ, Ω, δ, τ, q_i]) [Hughes, 79, p 196].

Hughes ne reconnaît qu'un certain nombre d'opérations applicables à ce vocabulaire.

Ces opérations sont :

- la lecture : Read(S) avec S $\in \Sigma$
- l'écriture : Write(E) avec E $\in \Omega^*$
- la traduction : f : S -> E avec S $\in \Sigma$, E $\in \Omega^*$

La correspondance est uniquement vérifiée au niveau du vocabulaire (on définit les règles de traduction formelle au niveau du vocabulaire). Tandis que la correspondance des niveaux supérieurs est induite de ce dernier niveau par récurrence (paragraphe 2.3.4.). Par contre, la définition de la correspondance dans la méthode de Jackson (paragraphe 1.2.5.) permet de déterminer des correspondances entre composants de niveaux intermédiaires sans se référer à leurs descendants. Ce dernier point permet de comprendre intuitivement pourquoi il n'est pas possible de détecter les conflits de frontière par le formalisme des expressions régulières. (Conflit de frontière :

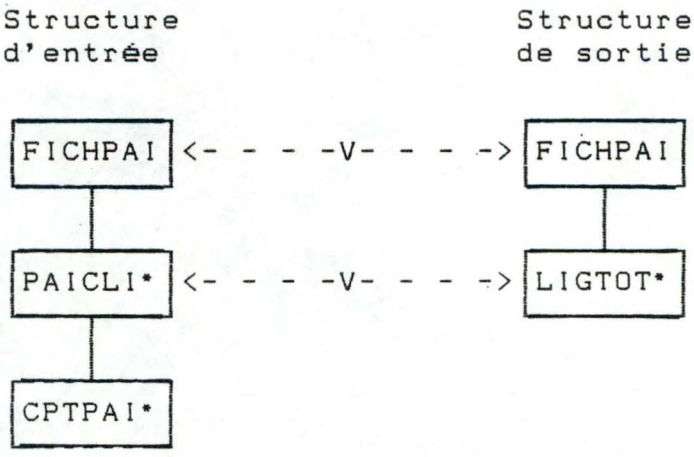
Les correspondances aux niveaux supérieurs des arbres et aux niveaux inférieurs sont vérifiées, tandis que les niveaux intermédiaires ne correspondent pas).

2.4.2. Niveau de langage

Le formalisme de Hughes ne reconnaît qu'un niveau de langage du côté des entrées et du côté des sorties. Ce niveau de langage correspond au vocabulaire d'entrée et de sortie (aux composants-élémentaires des deux structures). Tous les programmes construits selon la méthode de base et traitant les fichiers séquentiels lirons un élément à la fois. Les données d'un fichier sont considérées comme une suite d'éléments appartenant au vocabulaire d'entrée Σ ou de sortie Ω.

Or, pour un certain nombre de problèmes, un seul niveau de langage n'est pas suffisant. Les données des fichiers doivent être considérées comme des listes de données. Chaque liste de données peut être décomposée en plusieurs informations ou en plusieurs données du fichiers (vocabulaire). Nous verrons dans le paragraphe 2.6. consacré aux extensions du formalisme, que pour ces types de problèmes, nous devons considérer des niveaux supérieurs de langage.

Exemple : soit l'exemple de "TOTALISATION DE PAIEMENTS" (analysé dans le paragraphe 2.5.2.C.), dont les structures des données se présentent de la manière suivante (rem : on ne tient pas compte dans la stucture de sortie du début de rapport, ni de la fin de rapport) :



Si nous considérons les deux fichiers sous forme de suite

d'éléments du vocabulaire Σ et Ω (un seul niveau de langage) :

- Σ = {CPTPAI};
- Ω = {LIGTOT};

sur base de ces deux vocabulaires, la méthode de Hughes ne peut établir la correspondance entre PAICLI et TOTCLI. On devrait avoir comme vocabulaire d'entrée Σ' = {PAICLI}, et comme règle de traduction : Output(PAICLI) = TOTCLI.

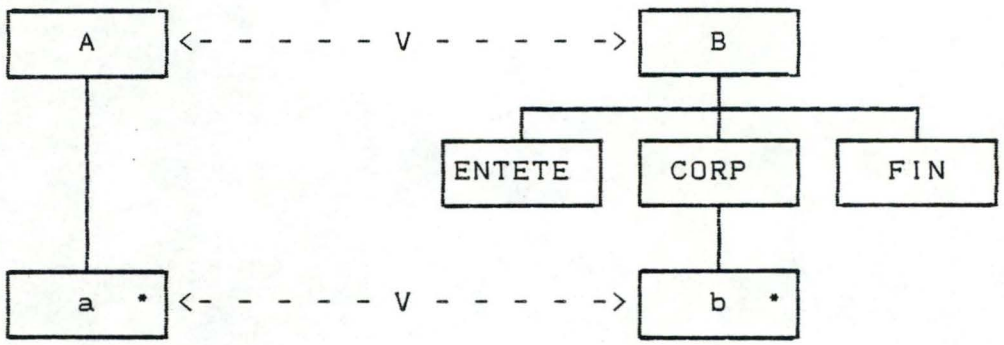
2.4.3. Caractéristique de Ginsberg

Par la troisième caractéristique d'une Machine d'Etats-Finis : F(E) = E

où E = élément vide;

un composant-vide ne peut générer aucune sortie.

Dans certains problèmes de la méthode de Jackson, il peut apparaître une génération spontanée d'une sortie tel qu'un en-tête de rapport, sans composant correspondant dans la structure d'entrée. Ce type de problèmes n'empêche pas la correspondance (de Jackson) entre les deux structures.

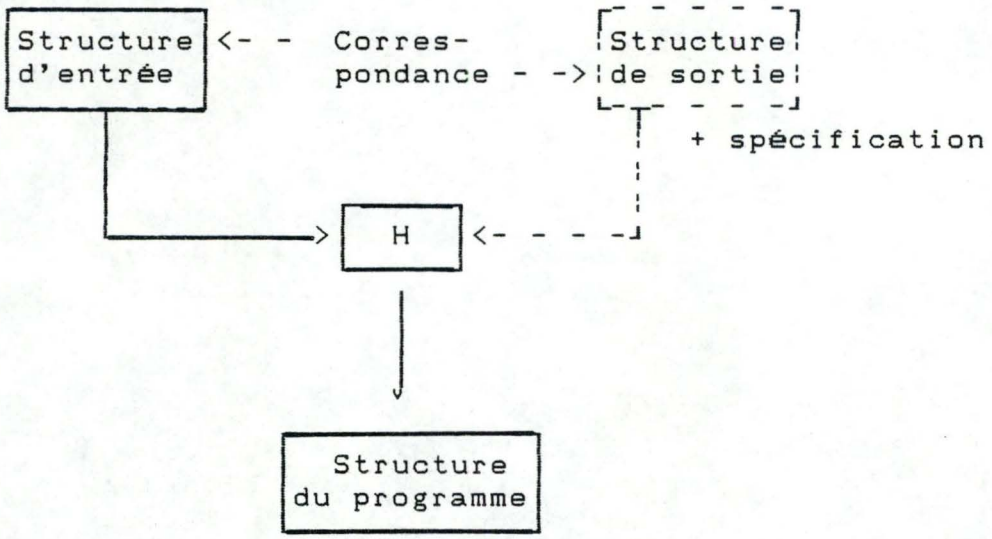


2.4.4. Structure du programme

Dans la méthode de base de Hughes, on élabore la structure du programme sur la structure des données en entrée. Le programme est obtenu en ajoutant à cette structure des instructions de lecture du fichier d'entrée et d'écriture sur le fichier de sortie suivant certaines règles de "traduction". Le programme "Traitement de texte" (2.3.1.) a été construit de cette manière (cfr. annexe 4). Le fait de baser la structure de programme uniquement sur la structure des données en entrée, peut apparaître comme une restriction dans l'élaboration d'un

programme "simple" et "efficace" (cfr paragraphe 1.3.).

Toutefois, on peut retrouver dans le formalisme de Hughes et la construction de la structure d'un programme les deux principes que l'on a défini au paragraphe 1.3.4.3. : le but de l'application (principe 1) se retrouve dans les fonctions informelles "Output" au niveau des composants-élémentaires (règle de traduction) et les propriétés des résutats (principe 2 : raisonnement descendant) se retrouvent dans la fonction formelle générale "Output" entre l'arbre de sortie et l'arbre d'entrée (dans l'exemple du "Traitement de Texte", les deux structures de données correspondent suivant les règles de traductions "O" : {O(nl)=E; O(sp)=E; O(l)=l; O(nl)=ew; O(sp)=ew); O(eot)=eof}).



2.4.5. Fonction 'Output'

Ce dernier point n'est pas véritablement une limite du formalisme, mais plutôt une précision apportée aux fonctions de "traductions" :

$$O = Output(I)$$

où - $I \in \Sigma$;
- $O \in \Omega^* \cup \{E\}$;

Nous avons vu dans les machines d'états-finis que l'élément de sortie (O) est dépendant de l'élément d'entrée (I), mais également de l'état (S_i) de la machine, c'est-à-dire qu'un élément (I) peut avoir deux sorties différentes (O_j et O_k) suivant son état (S_i). La fonction 'Output' définie par Hughes

ne peut faire une telle distinction. Dans l'exemple du "Traitement de texte", nous avons pour certains caractères de "TAPEFILE" deux types de caractères de sortie : ew=Output(nl); E=Output(nl); ew=Output(sp); E=Output(sp).

Pour tenir compte de l'état des élements d'entrée, nous modifions les fonctions de traductions de la manière suivante :

Output : $S \times \Sigma \rightarrow \Omega^* \cup \{E\}$;

où : S = ensemble fini d'états (S_1, .., S_n);
Σ = alphabet d'entrée;
Ω = alphabet de sortie;
E = élément vide;

Le couple (S,Σ) définit un élément de l'ensemble Σ dans un état S. Cette fonction Output détermine toutes les traductions d'un arbre vers l'autre;

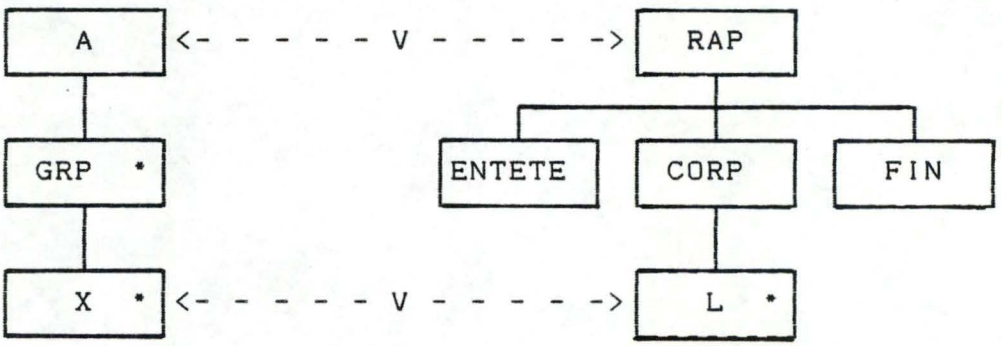
ex : ew = Output(1,nl);
E = Output(2,nl);

(cfr diagramme de transition du paragraphe 2.3.5.)

2.5. Extensions du formalisme de Hughes

2.5.1. Composant fictif

Une des contraintes des machines d'états-finis est l'interdiction de génération d'éléments de sortie sans entrées initiales : f(E) = E.
Pourtant, bon nombre de problèmes auraient besoin d'une règle permettant de contourner cette restriction, sans pour autant nuire à la correspondance (de Jackson) entre deux structures d'arbre.
Un des multiples exemples est la notion "d'en-tête" et de "fin" de rapport :



(fig 6.1.)

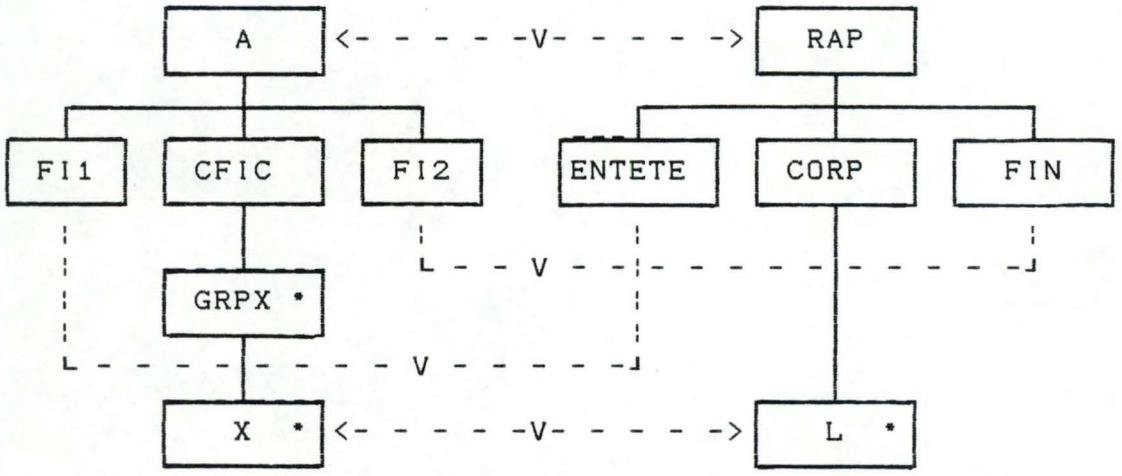
C'est à la troisième étape de la construction de programme (recherche de correspondances), que l'on constate que des éléments du vocabulaire de sortie ne peuvent être produits, c'est-à-dire que le formalisme de Hughes ne peut déterminer une correspondance entre les deux expressions régulières :

$$\exists\, S' \in \Omega \mid \forall\, X \in \Sigma : f(X) \neq S' \;;$$

(au moment où l'on définit les règles de traduction d'une structure vers une autre, on peut déterminer si certains éléments de Ω ne sont pas produits à partir de Σ.)

S'il y a effectivement au sens de Jackson une correspondance entre les deux structures, il est possible d'intégrer des composants fictifs (composants-élémentaires) à la structure d'entrée pouvant être mis en correspondance avec les composants S' de la structure de sortie (qui ne possédaient pas de correspondants).

Exemple : (modification des structures de la fig 6.1.)



(fig 6.2.)

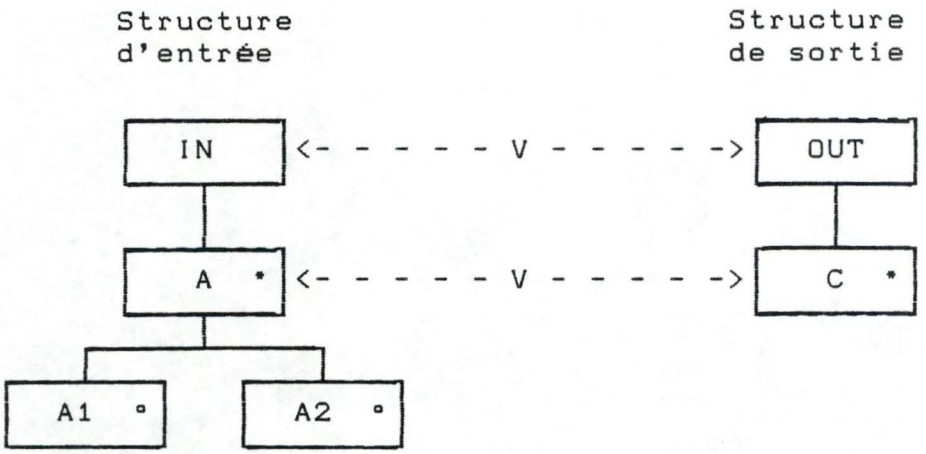
Les composants fictifs (FI1 et FI2) représentent des nouveaux éléments du vocabulaire d'entrée,

soit $\Sigma' = \Sigma \cup \{FI1, FI2\}$.

2.5.2. Niveau de langage

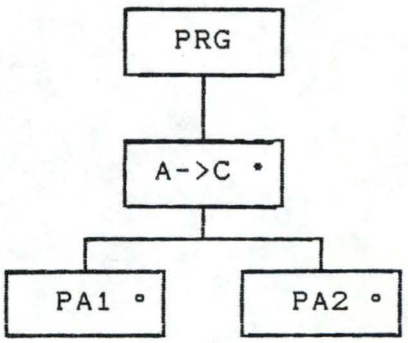
A. Le formalisme de base ne découvre des correspondances qu'au niveau des composants-élémentaires. Par contre, dans la méthode de Jackson, le fait de trouver une correspondance entre des composants-élémentaires d'une des structures et des composants de niveau intermédiaire de l'autre structure, permet de trouver une structure de programme basée sur ces deux structures (et de construire le programme). Ce type de correspondance, le formalisme de Hughes ne le permet pas, car il n'accepte qu'un niveau de langage (l'alphabet d'entrée ou composants-élémentaires d'entrée pouvant être mis en correspondance uniquement avec l'alphabet de sortie) (cfr. paragraphe 2.4.1.).

Exemple 1 : niveau de langage supérieur dans la structure d'entrée :



(fig 6.3.1)

Structure du programme correspondant :



(fig 6.3.2)

L'extension du formalisme que nous proposons est la définition d'un nouveau niveau de langage, soit du côté de la stucture d'entrée (exemple 1), soit du côté des sorties, soit du côté des deux structures. La définition d'un niveau de langage supérieur d'une structure consiste à créer un nouveau vocabulaire (supérieur) basé sur le vocabulaire de base c'est-à-dire, que l'on considère une sous-expression entre parenthèses (un composant intermédiaire d'une structure) de l'expression régulière comme faisant partie du nouveau vocabulaire. L'ensemble du vocabulaire formant cette sous-expression régulière est supprimé de l'ensemble du vocabulaire de base (ensemble des composants-élémentaires de la structure) et est remplacé par la sous-expression entre parenthèses considérée comme atomique, représentant un nouveau élément du vocabulaire.

Le vocabulaire supérieur est obtenu par :

- suppression de la liste du vocabulaire courant des composants-élémentaires-descendants CEi du composant

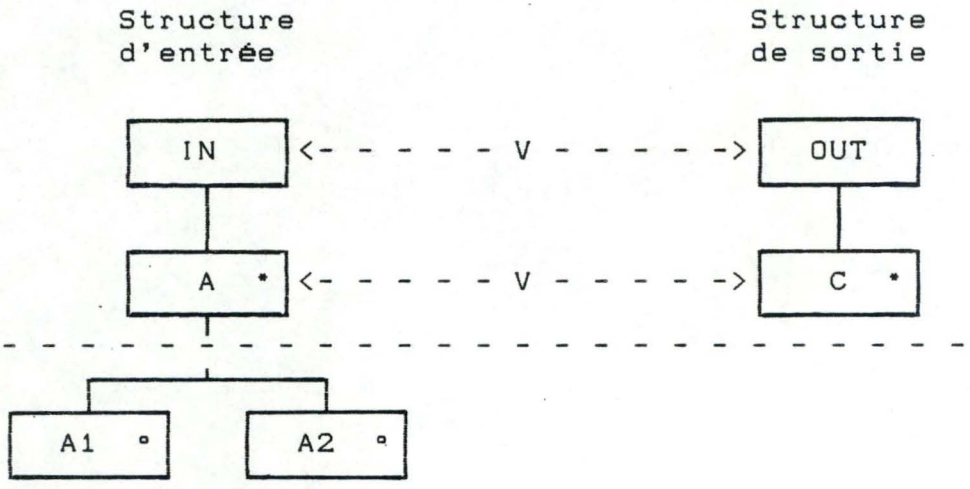
intermédiaire CI considéré comme faisant partie du nouveau vocabulaire;

- ajout, dans la liste du vocabulaire, du composant intermédiaire CI.

Le composant intermédiaire CI devient un composant élémentaire (fictif) de l'arbre et défini un nouvel élément du vocabulaire. La méthode de Hughes est alors appliquée à ce nouveau niveau de langage.

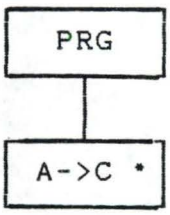
Dans l'exemple 1, l'alphabet de base du côté des entrées est : Σ = {A1, A2}, et du côté des sorties : Ω = {C}.
Si l'on veut déterminer un nouveau niveau de langage dans la structure des entrées pour permettre, en appliquant la méthode de Hughes, de trouver la correspondance des deux structures, on considère le composant A comme faisant partie du vocabulaire. Les composants-élémentaires descendant du composant A sont : {A1, A2}. Si on les supprime de la liste et on les remplace par le composant A, on obtient un nouvel alphabet Σ' = {A}.

Les structures des deux flux de données, en tenant compte du nouveau niveau de langage, deviennent :



(fig 6.3.3.)

Structure du programme correspondant :

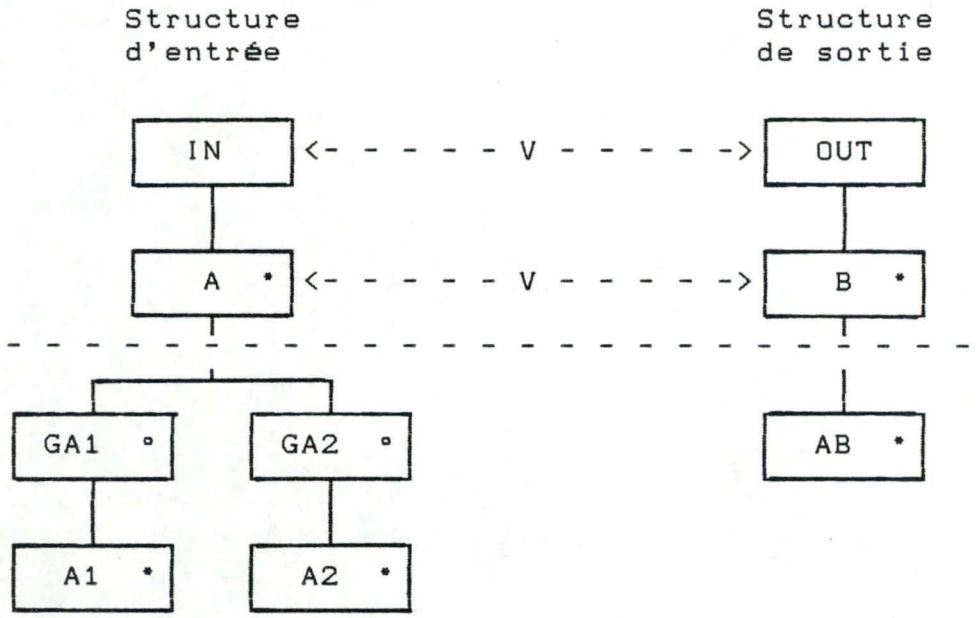


(fig 6.3.4)

La création de nouveaux niveaux de langage dans la structure d'arbre, entraine la création de nouvelles opérations (différentes de la lecture et de l'écriture simple d'un élément d'un fichier) (cfr. paragraphe 2.5.2.). Ces nouvelles opérations liées aux composants-élémentaires fictifs seront appelées Macro-Opérations et constitueront souvent un appel à une procédure traitant les composants-élémentaires effectifs d'un arbre. Les Macro-Opérations seront déterminées dans la dernière étape (opération) de l'élaboration du programme.

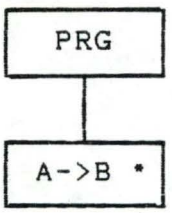
Le but de cette extension, est de permettre sur base de nouveaux niveaux de langage definis sur les deux structures, de trouver des correspondances entre niveaux intermédiaires. Ces niveaux intermédiaires sont considérés comme composants-élémentaires et le formalisme de Hughes peut être utilisé. On créera des Macro-Opérations (de lecture pour les entrées et d'écriture pour les sorties) permettant la construction de la structure du programme sur base de la correspondance (intermédiaire) de Hughes.

Exemple 2 : niveau de langage supérieur dans les deux structures :



(fig 6.4.1)

La structure du programme correspondante, tenant compte des nouveaux niveaux de langage : $\Sigma' = \{A\}$ et $\Omega' = \{B\}$ et des macro-opérations liées à A et à B, est :



B. Cette extension entraine certaines modifications dans la construction de programme. A la seconde étape, on définit pour les structures d'entrée et/ou de sortie un nouveau langage avec un alphabet plus puissant. A la quatrième étape (opération), lors de l'établissement de la liste des opérations, on définit les macro-opérations de lecture ou d'écriture. Ces macro-opérations seront ajoutées à la structure du programme de la même façon que les opérations "élémentaires".

C. Exemple : "TOTALISATION DE PAIEMENTS"
[VAN 'T DACK, 86]

Etape 1 : Énoncé

"Concevez un programme qui totalise un fichier avec des

enregitrements de paiements. Le format de chaque enregistrement de paiements se présente comme suit :

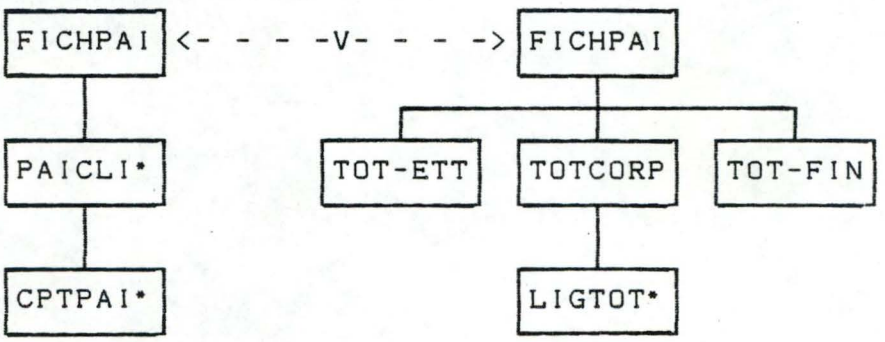
```
COMPTE-PAIEMENTS = RECORD
                       NO_CLIENT : array[1..6] of char;
                       PAIEMENTS : integer
                   END;
```

Le fichier des enregistrements de paiements (FICHPAI) est trié de telle sorte que tous les paiements d'un client se suivent et forment un groupe. L'information a déjà été controlée par un autre programme, et le fichier ne contient pas d'erreurs.
La LISTE TOTAL (fichier de sortie) doit contenir une seule ligne de 80 positions pour chaque client ayant un COMPTE-PAIEMENTS sur le FICHPAI. Pour ce client, la ligne doit contenir le NO-CLIENT et le total des PAIEMENTS de tous les COMPTE-PAIEMENTS de ce client. Le format de la LIGNE TOTAL est le suivant :

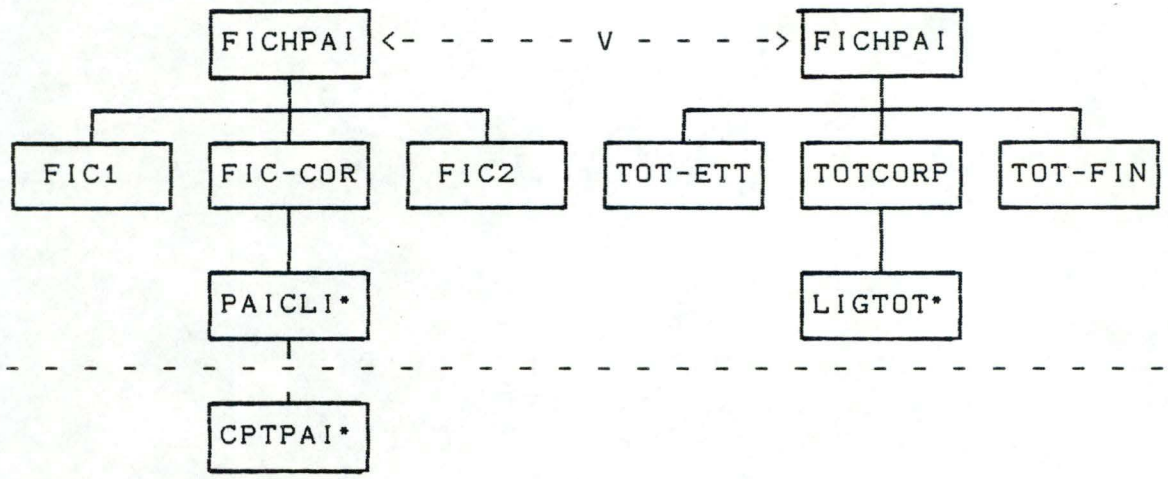
```
LIGNE_TOTAL = RECORD
                  NO_CLIENT : array[1..6] of char;
                  TOT_CLIENT : integer
              end;
```

La LISTE-TOTAUX doit aussi contenir une ligne en-tête LISTE TOTAL PAIEMENTS et une ligne de fin (FIN LISTE TOTAL). IL n'y a pas lieu de tenir compte du saut de page."

Etape 2 : Structure des données en entrées et en sorties sont :



En tenant compte des correspondances des composants TOT-ETT et TOT-FIN avec des composants fictifs, on obtient de nouvelles structures des données où chaque composant-élémentaire peut être mis en correspondance :



Sur base des composants fictifs et d'un nouveau niveau de langage de l'arbre d'entrée FICHPAI défini à partir du composant PAICLI, le vocabulaire de chaque structure est :

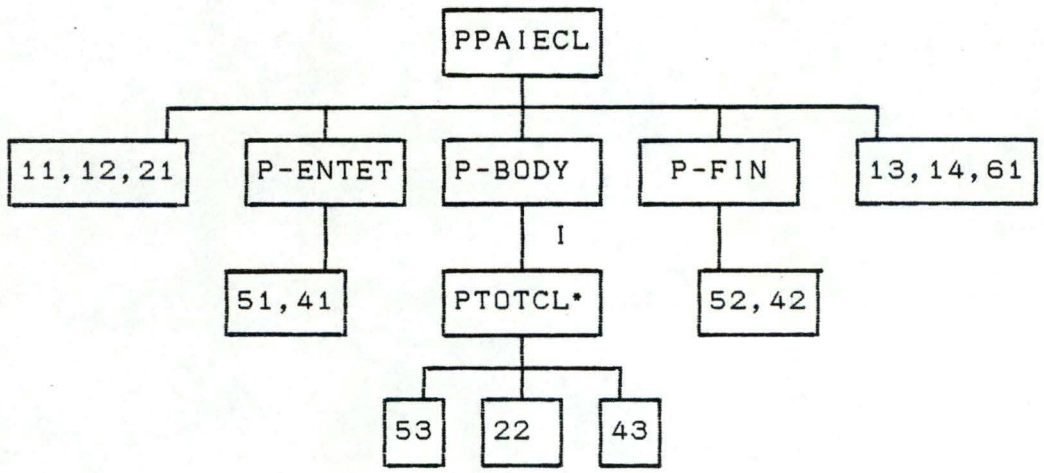
- Σ = {FIC1, PAICLI, FIC2};
- Ω = {TOT-ETT, LIGTOT, TOT-FIN}.

Les expressions régulières correspondantes sont :

- FIC1.(PAICLI)*.FIC2;
- TOT-ETT.(LIGTOT)*.TOT-FIN

Etape 3 : La structure du programme se présente de la manière suivante, sur base des règles de traduction :

Output(FIC1) = TOT-ETT;
Output(PAICLI) = LIGTOT;
Output(FIC2) = TOT-FIN.



Etape 4 : Opérations :

1. 11 reset(fichpaie)
   12 rewrite(listetotaux)
   13 close(fichpaie)
   14 close(listetotaux)

2. 21 read(enr,fpaie)
   22 LIREGRCLI {Maro-Opération}

3. 31 w-tot := w-tot + paiements

4. 41 write(listetotaux,tete)
   42 write(listetotaux,fin)
   43 write(listetotaux,lignetotal)

5. 51 tete := 'LISTE ..'
   52 fin := 'FIN ..'
   53 client-courant := w_no_client

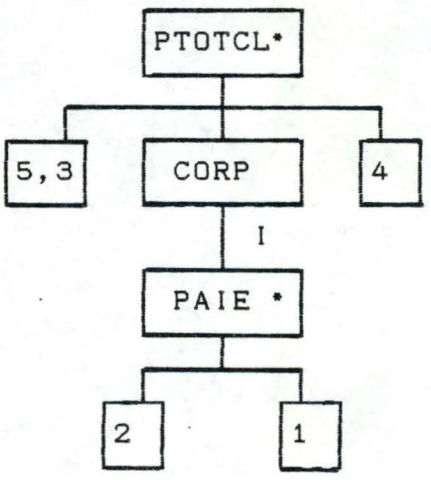
6. 61 stop run

* Spécification des conditions d'itération :

I : while (not eof(fichpaie)) do

* Spécification de "LIREGRCLI" :

Liregrcli(nr-cli-courant, enr, ligne-total) établit le total des paiements d'un client (nr-cli-courant) et initialise la variable 'ligne-total' qui figurera sur le rapport. Liregrcli reçoit également le premier enregistrement d'un groupe client (gr-cli).



opérations : (1) read(fichpaie,enr)
(2) w-tot := w-tot + paiement
(3) ligne-tot.no-client := client-courant
(4) ligne-total.tot-client := w-tot
(5) w-tot := 0

```
condition d'itération :
            (I) while ((not eof(fichpaie) and
                        (no-client <> client-courant)) do
```

2.6. Recherche d'un nouveau formalisme

2.6.1. Introduction

Sur base des critiques formulées dans le paragraphe 2.3. (formalisme de Hughes) et de l'analyse réalisée dans les paragraphes 2.4. et 2.5. (limites et extensions), nous proposons un nouveau formalisme nous permettant de mieux définir le concept de correspondance formelle et, la relation entre la méthode de base de Jackson et les fonctions réalisables par des Machines d'Etats-Finis.

Un des avantages du formalisme que nous proposons, est qu'il se base directement sur les structures d'arbres et non plus sur des expressions régulières. Il nous permettra de définir formellement des correspondances tant au niveau des composants élémentaires qu'aux niveaux des composants intermédiaires. Un second avantage est qu'il nous permet de créer des règles de construction systématique d'un progamme à partir de la correspondance des structures (cfr. paragraphe 2.7.).

2.6.2. Définition de la démonstration de la correspondance formelle entre deux expressions régulières

A. Définition :

A partir de la définition de la correspondance formelle de Hughes, nous classerons un certain nombre de règles d'inférence, à partir desquelles, nous baserons notre définition de la correspondance (formel'e) entre deux structures d'arbre.

1. Règles de déduction :

Nous classons les règles de déduction en quatre catégories :

1.1. Règles de déduction entre deux niveaux d'arbres

(passage des entrées (I) vers les sorties (O)).

soient : - pour tout i : Ai (Bi) est un sous-composant

du composant A (B).

- O est une fonction de "traduction" d'un arbre d'entrée vers un arbre de sortie.
- E est le composant "vide" d'un arbre de Jackson.

- Règle IO1 : (séquence)

si : $B_i = O(A_i)$;
$A = A_1 \ldots A_n$;
$B = B_1 \ldots B_n$;
---
alors : $O(A) = B$

- Règle IO2 : (sélection)

si : $B_i = O(A_i)$;
$A = A_1 / \ldots / A_n$;
$B = B_1 / \ldots / B_n$;
---
alors : $O(A) = B$

- Règle IO3 : (itération)

si : $B_i = O(A_i)$;
$A = A_1^*$;
$B = B_1^*$;
---
alors : $O(A) = B$

- Règle FAB :

si A et B sont des composants élémentaires;
alors, $O(A) = B$

- Règle FAE :

si A est un composant élémentaire;
alors, $O(A) = E$

- Règle FEB :

si B est un composant élémentaire;
alors, $O(E) = B$

1.2. Règles de transformation d'un arbre d'entrée

Soient : - pour tout i : Ai est sous-composant de A;
- E représente le composant "vide".

- Règle TAE :

si $A = A_1 \ldots A_n$;

alors $A = A_1 .. A_i.E.A_{i+1} .. A_n$

pour $0 \leq i \leq n$

- Règle T/ : (permutation des sous-composants-sélection)

si $A = A_1 / .. / A_n$;

alors $A = A_{i1} / .. / A_{in}$

où $i_1, .. , i_n$ perm 1, .., n

1.3. Règles de tansformation d'un arbre de sortie

Soient : - pour tout i : B_i est sous-composant de B;
- E représente le composant "vide".

- Règle TBE :

si $B = B_1 \ldots B_n$;

alors $B = B_1 .. B_i.E.B_{i+1} .. B_n$

pour $0 \leq i \leq n$

- Règle T/ : (permutation des sous-composants-sélection)

si $B = B_1 / .. / B_n$;

alors $B = B_{i1} / .. / B_{in}$

où $i_1, .. , i_n$ perm 1, .., n

1.4. Règles particulières :

- Règle T*1 :

si $A = A_1^*$;
$A_1 = A_2^*$;

alors $A = A_2^*$

- Règle E* :

si $A = E^*$;

alors $A = E$

- Règles B/ et E/ :

B = B / B;
E = E / E.

2. Définition : correspondance des arbres d'entrée et de sortie

"Il existe une correspondance formelle entre l'arbre d'entrée (AE) et l'arbre de sortie (AS) si et seulement si il existe une preuve formelle de O(RE) = RS (Racine de AE et AS) utilisant les règles de déduction (décrites ci-dessus)".

La preuve contient un certain nombre d'égalités de la forme $O(A_i) = B_i$. Chacune d'entre-elle définit une correspondance entre A_i et B_i. La correspondance est dite "effective" si elle répond à la sémantique du problème c'est-à-dire qu'elle répond au but défini dans la spécification du problème. Les contraintes sémantiques sont de deux types. La première contrainte porte au niveau des composants élémentaires : les relations fonctionnelles entre données élémentaires doivent respecter la précédence des opérations. La seconde contrainte s'applique aux composants de niveaux intermédiaires : on dit qu'il y a une correspondance n-m entre deux composants A et B, si pour chaque sous-séquence de n éléments de A, le programme construit génère une sous-séquence de m éléments de B (condition nécessaire, mais non suffisante).

B. Remarque : La preuve contenant un certain nombre d'égalités entre composants intermédaires de la forme $O(A_i) = B_i$, il nous est permis de définir des contraintes (sémantiques) sur la correspondance des deux structures d'arbres (à tous les niveaux). Ces contraintes peuvent être formulées sous la forme d'égalités entre deux composants (élémentaires ou intermédiaires) O(CEi) = CSi, et la démonstration de la correspondance vérifiera aisément ces égalités lors de l'application des règles d'inférences (exemple : nous voulons que la démonstration de la correspondance formelle vérifie la fonction sémantique suivante : O(body)=words).

C. Exemple :

Reprenons les deux structures de données "TAPE FILE" et "OUTPUT FILE" de l'exemple du "Traitement de texte" de Hughes

(paragraphe 2.3.2.).

Nous présentons la démonstration d'une des correspondances formelles suivant les règles de déduction.

(Notation : Chaque ligne est divisée en quatre colonnes et représente une étape de la déduction. La première colonne identifie la ligne, la seconde contient une égalité qui, soit doit être prouvée par d'autres égalités (colonne 3 = '?'), soit correspond à une des règles de déduction ou à une définition d'un noeud (règle AE ou règle ES) (col 3 = ' '). La dernière colonne mentionne les règles et/ou les autres égalités nécessaires à la preuve).

Démonstration :

1 .	O(tapefile) = outputfile	?	I01,2,4,5,6,7
2 .	tapefile = padd.body.eot		Règle AE
3 .	outputfile = words.eof		Règle AS
4 .	= E.words.eof		TBE,3
5 .	O(padd) = E	?	I03,8,9,10
6 .	O(body) = words	?	
7 .	O(eot) = eof	?	TAB
	---		
5 .	O(padd) = E	?	I03,8,9,10
8 .	padd = chs*	?	AX-AE
9 .	E = E*	?	E*
10.	O(nl) = E	?	I03,11,12,13,14
11.	chs = nl/sp		Règle AE
12.	E = E/E		E+
13.	O(nl) = E		FAE
14.	O(sp) = E		FAE
	---		
6 .	O(body) = words	?	I03,15,16,17
15.	body = words*		Règle AE
16.	words = word*		Règle AS
17.	O(words) = word	?	

17.	O(words) = word	?	
18.	words = letters.delim.padd		Règle AE
19.	word = letters.ew		Règle AS
20.	= letters.ew.E		TBE, 19
21.	O(letters) = letters	?	
22.	O(delim) = ew	?	
23.	O(padd) = E		5

21.	O(letters) = letters	?	I01,24,25,26,27
24.	letters = l.l*		Règle AE
25.	letters = l.l*		Règle AS
26.	O(l) = l		FAB
27.	O(ls) = ls	?	

```
   ----------
   27.   O(ls) = ls                    ?  I03,26,27,28
   28.   ls = l*                          Règle AE
   29.   ls = l*                          Règle AS
   30.   O(l) = l                        FAB
   ----------
   22.   O(delim) = ew                  ?  I02,31,32,33,34
   31.   delim = nl/sp                    Règle AE
   32.   ew = ew/ew                      B+
   33.   O(nl) = ew                      FAB
   34.   O(sp) = ew                      FAB
-------------------------------------------------------------
```

Si les fonctions de traduction au niveau du vocabulaire {O(nl)=E; O(sp)=E; O(l)=l; O(nl)=ew; O(sp)=ew; O(eot)=eof} s'identifient à la spécification du problème; la correspondance formelle que nous avons démontrée est appelée correspondance effective.

2.6.3. Construction d'une M.E.F. sur base des règles d'inférence

2.6.3.1. Introduction

Sur base des règles d'inférence définient dans le paragraphe 2.6.2., nous expliquerons de manière générale comment on peut construire une Machine d'Etats-Finis correspondant à une structure de programme de la méthode de base.

Nous rappelons certaines caractéristiques d'une M.E.F. (définie dans le paragraphe 2.2.) :

1. une M.E.F. est caractérisée par un état initial $\{e_i\}$ et par des états finaux $\{e_{f1}, .., e_{fn}\}$ $(n \geq 1)$;
2. une M.E.F. est définie par la fonction des états internes "G" (cfr. paragraphe 2.2.3.); une M.E.F. étendue permet la génération d'un langage (de sortie) et la définition est alors complètée par la fonction de sortie "F";
3. une M.E.F. débute dans l'état initial "e_i" et termine dans des états finaux "$\{e_{fj}\}$" en lisant un langage d'entrée qu'elle 'reconnait';
4. un langage d'entrée est dit "reconnaissable", si il mène la machine de l'état initial (e_i) à un certain ensemble d'états finaux $\{e_{fj}\}$.

Notation : lors de la construction des M.E.F. de ce chapitre, les flèches des différents schémas représentant les transitions d'états seront étiquetées par les éléments des langages

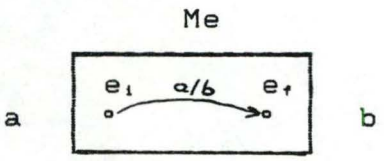
d'entrée et de sortie (e/s) correspondants.

La construction de la M.E.F., qui représente les fonctions s'identifiant à la structure de programme de la méthode de base (correspondance), s'élabore sur base des quatre types de composants d'une structure d'arbre : composant élémentaire, composant séquence, composant sélection, composant itération. Nous montrerons qu'un composant élémentaire représente une fonction exécutable par une M.E.F., ensuite, qu'une composition (séquence, sélection ou itération) de plusieurs M.E.F. est également une M.E.F.

2.6.3.2. Composant élémentaire

soit - Me : une Machine d'Etats-Finis;
- e_i : état initial de Me;
- e_f : état final de Me;
- CE : Composant élémentaire de la structure d'un programme mettant en correspondance les éléments 'a' et 'b'.

Nous pouvons d'écrire la machine Me correspondant au composant CE de la manière suivante :

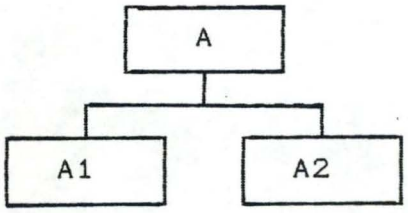


Le seul langage que la machine Me reconnaisse est la chaîne de 'a' : {a} et le seul langage que la machine Me puisse générer est la chaîne de 'b' : {b}.
'Me' passe de l'état initial e_i à l'état final e_f en lisant 'a', le seul élément du langage d'entrée.

2.6.3.3. Séquence de M.E.F.

Soit la machine M = M1 . M2 (séquence de deux M.E.F.).
Nous allons montrer que M est également une M.E.F.

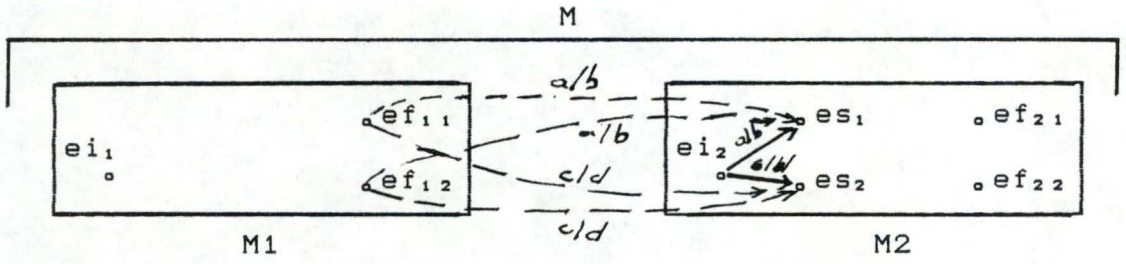
soit : -



où Ai représente le langage reconnu par la machine Mi.

- ei_1, ef_{11}, ef_{12} : l'état initial et les états finaux de M1;
- ei_2, ef_{21}, ef_{22} : l'état initial et les états finaux de M2;

Nous construisons la machine d'états finis M de la manière suivante :



où es_1, es_2 : états successeurs des états initiaux ei_1 et ei_2.

La machine M est obtenue en fusionnant les deux machines M1 et M2 (fusion des états finaux de M1 et de l'état initial de M2) de la manière suivante :

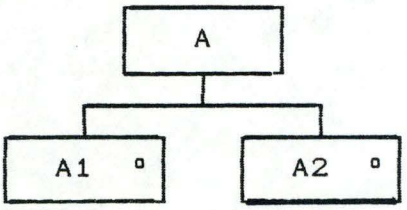
1) rajouter les arcs de transition des états finaux ef_{11} et ef_{12} vers les états successeurs es_1 et es_2 de l'état initial de M2;
2) supprimer l'état initial ei_2;
3) l'état initial de M est l'état initial de M1 : ei1;
   les états finaux de M sont les états finaux de la machine M2 : $\{ef_{21}, ef_{22}\}$.

Le langage d'entrée reconnaissable est l'union des langages reconnaissables par M1 et M2 : A = {A1 U A2}
Les états ef_{11} et ef_{12} ne sont plus des états finaux, mais des états intermédiaires de M qui débutent les états de M2.
Remarque : une des hypothèses de la méthode de base est le déterminisme des structures d'arbre (d'entrée). La lecture

d'un seul élément du langage d'entrée permet de déterminer de manière unique la transition d'un état à un autre ou la détection d'un état final de la machine.

2.6.3.4. Sélection de M.E.F.

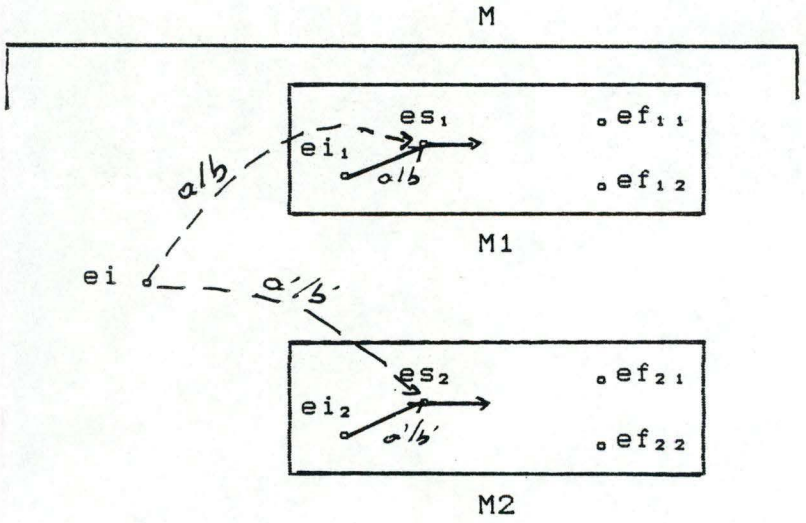
Soit la machine M = M1 + M2 (sélection de deux M.E.F.).
Nous allons montrer que M est également une M.E.F.
soit : -



où Ai représente le langage reconnu par la machine Mi.

- ei_1, ef_{11}, ef_{12} : l'état initial et les états finaux de M1;
- ei_2, ef_{21}, ef_{22} : l'état initial et les états finaux de M2;

Nous construisons la machine d'états finis M de la manière suivante :



où es_1, es_2 : états successeurs des états initiaux ei_1 et ei_2

La machine M est obtenue en fusionnant les deux machines M1 et M2 (fusion de l'état initial de M1 et de l'état initial de M2) de la manière suivante :

1) rajouter l'état initial ei de M, et les arcs de transition de

l'état initial ei vers les états successeurs es_1 et es_2 des états initiaux de M1 et M2;

2) supprimer les états initiaux ei_1 et ei_2;

3) l'état initial de M est ei;

les états finaux de M sont les états finaux des deux machines M1 et M2 : $\{ef_{1_j} \cup ef_{2_k}\}$.

Le langage d'entrée reconnaissable est l'union des langages reconnaissables par M1 et M2 : {A1 U A2}.

Les états ei_1 et ei_2 ne sont plus des états initiaux, mais peuvent être des états intermédiaires de M.

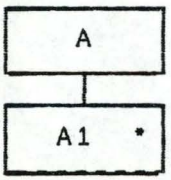
Remarque : le choix d'une seule (sous-)machine Mi est possible car le langage est déterministe. Une fois qu'une des sous-machines est sélectionnée, on reste dans les états de celle-ci.

2.6.3.5. Itération de M.E.F.

Soit la machine M = M1* (itération d'une M.E.F.).

Nous allons montrer que M est également une M.E.F.

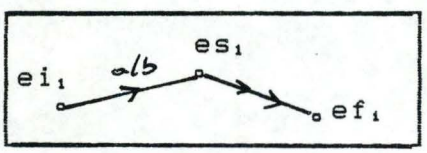
soit : -



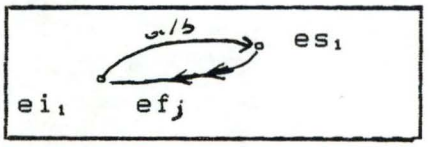
où A1 représente le langage reconnu par la machine M1.

- ei_1, ef_j : l'état initial et les états finaux de M1;

Nous construisons la machine d'états finis M de la manière suivante :



M1



M

où es_i : état successeur de l'état initial ei_i

La machine M est obtenue en "fusionnant" la machine M1 (fusion des états finaux et de l'état initial de M1) de la manière suivante :

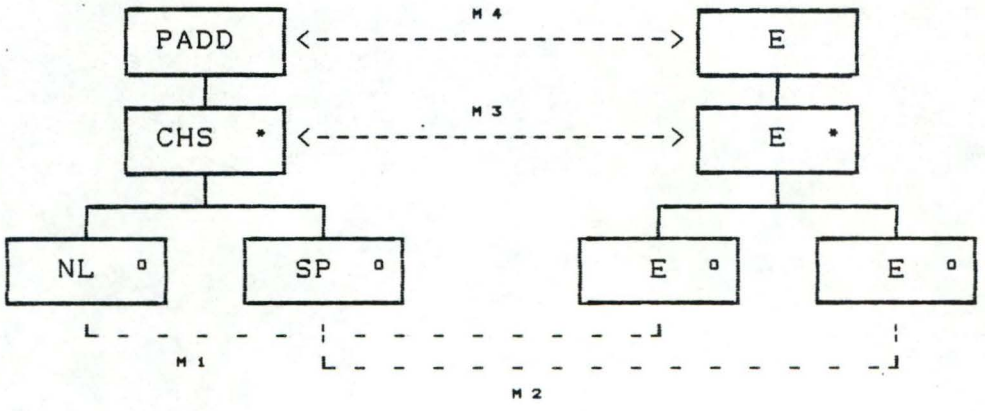
1) ei1 devient un état final ef1;
2) rajouter un arc de transition de l'état ei1 vers es1.

justification :

- soit la machine M s'arrête à l'état final ef_j, soit elle recommence un nouveau cycle par un passage à l'état successeur de ei1;
- ei_i devient un état final car il y a possibilité de "0" itération.

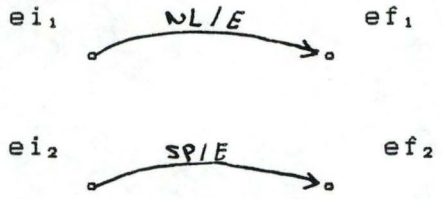
2.6.3.6. Exemple

Reprenons la branche "PADD->E" de la structure d'arbre du programme "Traitement de Texte" (p 175) :



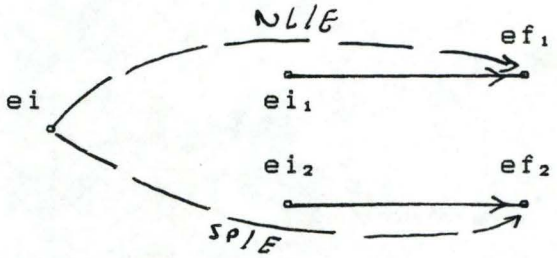
Nous construisons progressivement la M.E.F. correspondant à (PADD-E) sur base des M.E.F. relatives aux composants élémentaires (NL-E) et (SP-E) :

1) Machine élémentaire M1 et M2 :



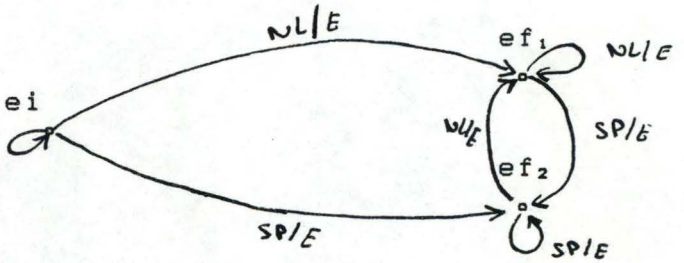
2) Machine sélection M3 (CHS-E) :

On fusionne les deux machines M1 et M2 de la manière suivante (cfr. pt. 2.6.3.4.) :



3) Machine itération M4 (PADD-E) :

On fait correspondre aux états finaux l'état initial de la manière suivante (cfr. pt. 2.6.3.5.) :



(l'unification de l'état initial et des états finaux est représentée par flèches pointillées)

On peut encore créer des règles simplificatrices qui donneraient le diagramme (simplifié) correspondant :

M4' :

Le reste de l'arbre (TAPEFILE-OUTPUTFILE) peut être traité de la même manière pour construire la M.E.F. correspondante. En utilisant des règles simplificatrices, on créera le diagramme de transition d'états de la page 92 (la M.E.F. M4' correspond à l'état 1 de ce diagramme).

2.7. Règles de construction de programmes

La construction d'un programme en pseudo-pascal à partir de la structure d'arbre peut se faire de manière systématique en utilisant les égalités de la démonstration de la correspondance basée sur les règles d'inférence. Nous rappelons que les opérations de connexions de bases entre composants d'un arbre de Jackson sont :

- la concaténation : ".";
- l'alternative : "/";
- l'itération : "*";

Dans les structures d'arbre d'un programme de Jackson, nous considérons également deux types de composants :

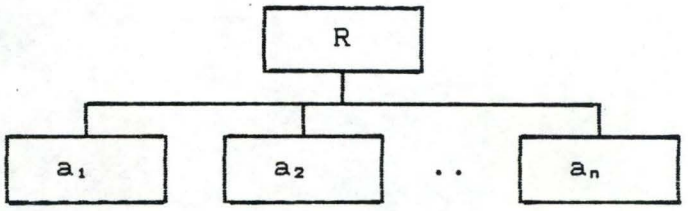
- Les composants feuilles (C.F.)
- Les composants intermédiaires (C.I.)

Remarque : Les composants feuilles sont les composants-actions-primitives correspondants aux opérations de base (lecture, écriture, opération arithmétique, etc...) qui ont été ajoutées à la structure d'arbre du programme (à l'étape précédente de construction de programme (paragraphe 1.2.6.)).

Sur base de ces opérations, nous pouvons définir des règles simples de construction de programme, qui appliquées de manière systématique à la structure d'arbre de Jackson du programme, donnent le pseudo-code Pascal du programme approprié. Nous les regroupons en deux catégories. D'une part, nous avons les règles de base (1), d'autre part, nous présentons un ensemble non-exhaustif de règles simplificatrices (2).

1. Règles de base de construction de programmes

Règle 1 : Si une structure de programme de Jackson a la forme suivante :



où | R ∈ C.I.
| a_i ∈ (C.I. U C.F.)
| i ≥ 2

qui est la représentation de la règle d'inférence I01 (paragraphe 2.6.) :

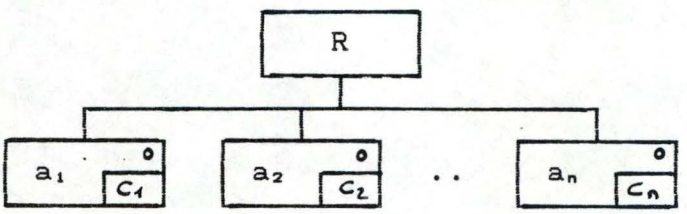
si : Bi = O(Ai);
A = A1 ... An;
B = B1 ... Bn;
---
alors : (A) = B

Alors, la version programme correspondante est :

```
Proc R ::=

begin
      Proc a1 ;
      Proc a2 ;
       ...
      Proc an
end
```

où Proc ai traduit tous string de Ai en un string de Bi.

Règle 2 : Si une structure de Jackson a la forme suivante :

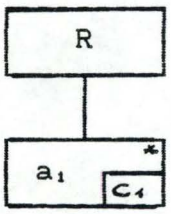


où | P ∈ C.I.
| a_i ∈ (C.I. U C.F) pour tout i
| C_i : condition de sélection du composant a_i

Alors, la version programme correspondante est :

```
Proc R  ::=

        if   C1    then  Proc a1
  else if   C2    then  Proc a2
   ....
  else if   Cn    then Proc an
  else error
```

Règle 3 : Si une structure de Jackson a la forme suivante :



où | $R \in$ C.I.
| $a_i \in$ (C.I. U C.F.)
| C_i = condition d'itération du composant a_i

Alors la version programme correspondante est :

```
Proc R  ::=

While (Ci) do
              Proc ai
```

Règle 4 : Si "a" est un composant feuille, alors "Proc a" est une instruction de base du programme

```
Proc a  ::=  instr_de_base
```

Pour définir le programme correspondant à une structure, nous pouvons citer les deux règles suivantes :

Règle 5 : Le composant de départ pour la construction d'un programme est la racine de la structure du programme. La première ligne du pseudo-code est également générée : "Program RACINE(input,output);" et Proc RAC est terminé par un "." :

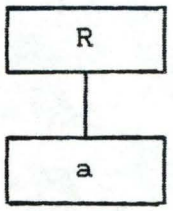
```
Proc RAC ::= Proc RAC.
```

Règle 6 : Si "a" est un composant intermédiaire, alors "Proc a" est un appel à une nouvelle règle de construction.

2. Règles particulières :

Cet ensemble de règles simplificatrices permet d'améliorer le code pseudo-pascal du programme. Il n'est toutefois pas exhaustif.

Règle 1' : Si une structure de structure de Jackson a la forme suivante :



où | R ∈ C.I.
| a ∈ C.F.

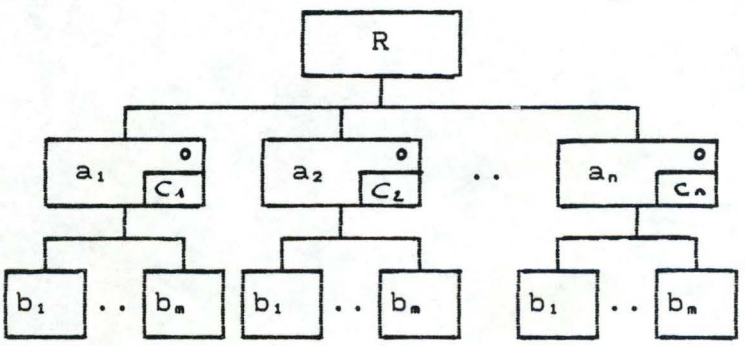
(C'est-à-dire : un composant intermédiaire n'est composé que d'un seul composant feuille (une seule instruction))

Alors, la version programme est :

Proc R ::= Proc a

(où, "Proc a" correspond à une instruction (cfr règle 4)

Règle 2' : Si à tous les sous-composants-sélection (a_i) d'un composant-sélection (R), on attache les mêmes composants feuilles (instructions) dans le même ordre :



où | $R, a_1, \ldots, a_n \in C.I.$
| $b_1, \ldots, b_m \in C.F.$

Alors le "Proc R" :

Proc R ::=

if C_1 then Proc a_1
else if C_2 then Proc a_2
else .. ..

else Proc a_n

avec "Proc a_i" égal à Pour i = 1 ...n

Proc a_i ::=

begin
Proc b_1 ;
Proc b_2 ;
...

Proc b_m
end

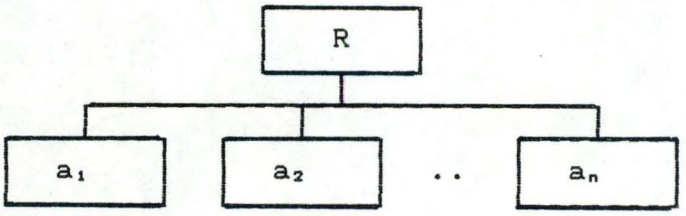
peut être transformé en :

Proc R ::=

begin
Proc b_1 ;
Proc b_2 ;
...

Proc b_m
end

Règle 3' : Soit R un composant-séquence et a_i ces sous-composants-séquence.

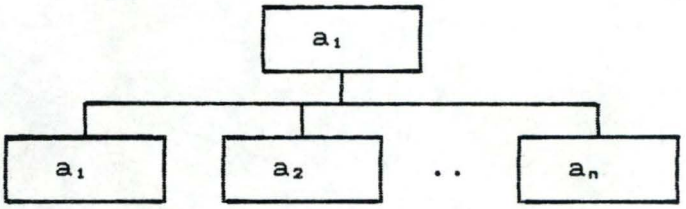


où : | $n \leq 2$
| $\exists\ i \in [1..n] : a_i \in C.I.$
| $\forall\ j = i : a_i \in (C.I. \cup C.F.)$
| $R \in (C.I. \cup C.F.)$

dont la version programme est :

```
Proc R  ::=

begin
    Proc a1 ;
    Proc a2 ;
     ...

    Proc an
end
```

Soit i = 1 => $a_1 \in$ C.I. tel que sa structure ait la forme suivante :



où | bi ∈ (C.I. U C.F.);
| m ≥ 2,

c'est-à-dire a_1 est lui aussi un composant-séquence, dont la version programme est :

```
Proc R
begin
    Proc b1 ;
    Proc b2 ;
     ...

    Proc bm
end
```

Alors la procédure R peut être transformée en :

```
Proc R
begin
    Proc b1 ;
    Proc b2 ;
     ...

    Proc bm
    Proc a2 ;
     ...

    Proc an
end
```

3. Exemple : "TRAITEMENT DE TEXTE"

Le développement de cet exemple est présenté dans l'annexe 4, où nous reprenons la structure du programme, correspondant à l'énoncé du paragraphe 2.3.2., obtenue après la quatrième étape (structure du programme complètée par les conditions d'itération et de sélection, ainsi que les actions primitives).

2.8. Complément au formalisme de Jackson-Hughes

2.8.1. Introduction

Nous présentons sommairement dans ce paragraphe certaines études postérieures à l'article de Hughes. Certaines présentent un formalisme basé sur celui de Hughes, tandis que d'autres s'orientent vers un tout autre formalisme. Nous présenterons le lien entre ces études et le niveau de développement auquel nous sommes arrivés.

2.8.2. Ebauche de formalisation de JAKSON-HUGHES

[DURIEUX, 86]

Cette étude se rapproche de la démarche que nous avons adoptée dans le paragraphe 2.6. pour reformuler les concepts de Hughes en utilisant une notation mathématique plus complexe. Durieux reprend la description donnée par J. Hughes de la construction de la correspondance Entrée-Sortie dans la méthode J.S.P., et il la transcrit sous la forme plus adaptée et plus rigoureuse de **règles d'inférence** exprimant les propriétés utiles de l'espace de recherche : celui des **transductions rationnelles.** Pour assurer la transcription d'un arbre vers un autre, on est amené à déployer un arsenal algèbrique pas totalement négligable sur les expressions algèbriques et les transductions rationnelles. Nous présentons les points particuliers de ce formalisme :

A. Algèbre des Transductions Rationnelles

Définition : Les transductions rationnelles de X^* dans Y^* sont définies usuellement comme les parties rationnelles du produit cartésien $X^* \times Y^*$, avec l'opération

induite :

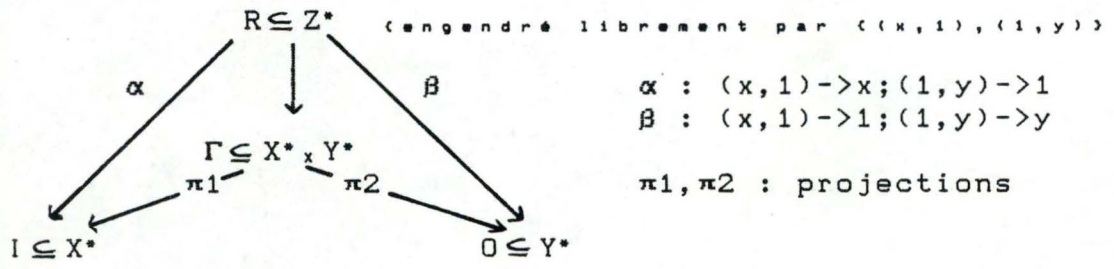
$$(u,u').(v,v') = (uv,u'v')$$

(rem : X^* et Y^* sont les monoïdes libres engendrés par l'alphabet X et Y.)

Théorème (Nivat,68) : Une partie Γ de $X^* \times Y^*$ est une transduction rationnelle si et seulement si il existe un langage R sur Z et deux morphismes alphabétiques α et β tels que :

$$\Gamma = \{(\alpha w, \beta w) \mid w \text{ dans } R\}$$

Schématiquement, on peut représenter ce résultat par :



B. Méthode de recherche

J. Hughes essaie en fait d'exprimer des propriétés de composition des transductions rationnelles, qui s'expriment plus rigoureusement sous la forme de règles d'inférence dans le style de Gentzen (cfr. [SINTZOFF, 84]).
Posant par définition Γ : I -> O si Γ(I) = O, de sorte que (x,1).(1,y) : x->y et (x,1) : x->1, on a les règles :

(a) Séquence : "+"

$$\frac{I=(I1+I2)\ O=(O1+O2)\quad \Gamma1:I1\to O1\quad \Gamma2:I2\to O2}{(\Gamma1+\Gamma2):(I1+I2)\to(O1+O2)}$$

(b) Sélection : "."

$$\frac{I=(I1.I2)\ O=(O1.O2)\quad \Gamma1:I1\to O1\quad \Gamma2:I2\to O2}{(\Gamma1.\Gamma2):(I1.I2)\to(O1+O2)}$$

(c) Itération : "*"

$$\frac{I=(I1)\ O=(O1)\quad \Gamma1:I1\to O1}{(\Gamma1)^*:(I1)^*\to(O1)^*}$$

La construction de Jackson-Hughes se présente alors comme

une preuve d'existence d'une transduction rationnelle Γ : input -> output à partir des règles exprimant les propriétés des expressions régulières, des systèmes réguliers et des transductions rationnelles. Les transductions rationnelles élémentaires, de forme x->y ou x->1 jouent le rôle d'axiomes spécifiques de l'application. Ils sont trouvés et fournis par le développeur. Le mécanisme de recherche de cette preuve peut être un des mécanismes usuels en démonstration automatique : chaînage avant ou chaînage arrière.

2.8.3. Automatic Program Synthésis From Data Structures

[ENSELME, 84]

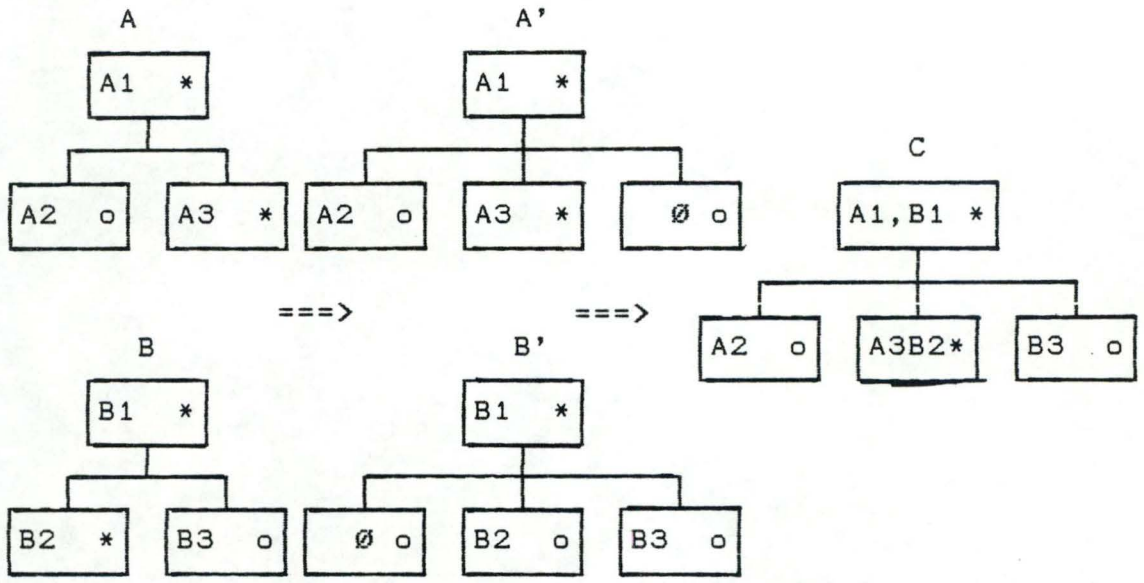
Cette étude s'oriente vers une autre formalisation de la méthode JSP que celle de Hughes. Elle propose un **système de synthèse de programme** à partir de la spécification des structures de données en entrée et en sortie, des relations fonctionnelles entre données élémentaires et des correspondances entre données structurées. La structure du programme synthétisé résulte d'une **unification contrainte des structures de données.**

A. La méthode de synthèse

La structure du programme est la résultante d'une contrainte d'unification des arbres correspondants aux données d'entrée et de sortie. Cette contrainte d'unification d'arbres peut être exprimée de la manière suivante : "étant donné deux arbres libellés A et B, il faut trouver deux séquences minimales de noeuds vides (Ø) pour changer A en A' et B en B' tel que :

(i) B est superposable à A';
(ii) A est superposable à B'.

On dit que T est superposable à T' si et seulement si la structure T peut être obtenue à partir de T' en supprimant seulement des noeuds vides. Les noeuds vides sont insérés simultanément dans les structures A et B en utilisant la technique "en largeur d'abord". La paire (A',B') est appelée la paire d'arbres "uniformisés".



L'arbre C unifié est purement syntaxique et ne tient pas compte de la sémantique du programme.

B. Contraintes sémantiques

Elles sont de deux types :

- Sur les feuilles : les relations fonctionnelles entre données élémentaires imposent de réarranger les noeuds dans l'arbre C unifié pour respecter la précédence dans les traitements.
- Sur les noeuds : la correspondance entre sous-structures doit aussi influencer l'unification (on dit qu'il y a correspondance (n-m) entre deux noeuds A2 et B2 si pour le programme construit, chaque sous-séquence de n éléments de A2 génère une sous-séquence de m éléments de B2).

C. Conclusion

Cette méthode d'unification semble mener à la construction d'arbre plus compliqué c'est-à-dire, des arbres constitués par plus de noeuds intermédiaires par rapport aux autres formalismes.

Par contre un des avantages de ce formalisme est la prise en compte de manière plus riche de l'aspect sémantique du problème traité.

2.8.3. Un outil d'aide à la conception de programmes supportant la méthode de Jackson

[VIVARES, 88]

Le but de cette étude est la réalisation d'un outil semi-

automatique de génération de programmes fonctionnant d'après la construction de Jackson-Hughes. Elle s'est basé pour cela sur la formalisation de la construction faite par J.L. Durieux. Aussi l'objet de cet article est de définir un environnement de programmation interactif supportant la totalité de la méthode et apportant une aide complète au développement de systèmes informatiques. La méthode de Jackson qui est présentée dans cet article est la méthode de conception de système dite J.S.D. (1) qui englobe la méthode J.S.P. (2)

La méthode J.S.D. comporte 6 étapes :

- les trois premières sont des étapes de modélisation de l'existant;
- les deux suivantes concernent la conception du système à implémenter;
- La dernière concerne l'implémentation de ce système.

La construction de Jackson-Hughes va permettre d'automatiser partiellement les trois premières étapes (modéliser les flots de données circulant entre les processus).

(1) Jackson Structured Development.
(2) Jackson Structured Programing.

TROISIEME PARTIE

APPLICATION

PARTIE 3 : APPLICATION

3.1. Introduction

Cette troisième partie consacrée au développement d'une application, se base sur le formalisme exposé dans la seconde partie. Cette application se divise en deux programmes. Le premier (programme "CORJACK") est lié au problème de recherche de correspondances formelles entre deux structures de données et repose sur les règles de déduction dégagées dans le paragraphe 2.6. Le second programme ("CONVPROG") se préoccupera de la génération d'un programme PASCAL à partir de la structure de correspondance établie par le premier programme et, sur base des règles de construction de programmes exposées dans le paragraphe 2.7. D'autres règles ont dû être créées, afin de respecter la syntaxe correcte du langage PASCAL, notamment pour déterminer les conditions des instructions conditionnelles (if .. then .. else) et itératives (while .. do ..). Les illustrations de cette partie se basent sur le problème du "Traitement de Texte" développé dans [HUGHES, 79] et dont l'énoncé est défini dans le paragraphe 2.3.2.

Remarque : Les types de variables des deux programmes ainsi que les différents écrans auquels nous faisons référence dans cette partie sont décrits dans l'annexe 5. Nous y avons également ajouté le programme "TEST" qui est le résultat du programme "CONVPROG" appliqué à l'exemple du "Traitement de Texte" (défini dans le paragraphe 2.3.2.).

3.2. Description du programme "CORJACK"

3.2.1. Introdution

Le programme "CORJACK" recherche les correspondances formelles de deux structures d'arbre sur base des règles de déduction définies dans le paragraphe 2.6. Nous utiliserons pour la représentation des arbres d'entrée et sortie à l'utilisateur les expressions B.N.F. (une amélioration de ce programme pourra être l'utilisation d'une représentation graphique de ces structures).

La description et l'utilisation de ce programme se divisent en deux parties. Nous décrirons dans la première, le principe de fonctionnement du programme (mode d'utilisation) et dans la seconde, nous expliquerons plus particulièrement la technique de recherche de correspondance entre deux composants par rapport aux règles de déduction.

3.2.2. Principe de fonctionnement

3.2.2.1. Création des arbres d'entrée et de sortie

A. L'introduction de la représentation des arbres de JACKSON peut se faire de deux manières (cfr. écran 1) :

1. l'arbre de JACKSON peut être introduit par l'utilisateur à partir du clavier (cfr. point B). Cette opération terminée, l'arbre de JACKSON est présenté sur l'écran à l'utilisateur sous forme d'expression B.N.F (cfr. écrans 9 et 11). On propose ensuite à l'utilisateur de sauver cette représentation sous forme d'expression B.N.F. dans un fichier (cfr. écrans 7 et 8).
2. Si l'arbre de JACKSON existe déjà sur un fichier sous forme d'expression B.N.F., l'utilisateur peut demander de transformer cette représentation d'expressions B.N.F. (cfr. écrans 10 et 11) en une représentation interne de l'arbre de JACKSON (cfr. type de la structure d'arbre : PCMPA).

B. La demande des composants d'un arbre se fait niveau par niveau et de gauche à droite :

- NIV 0 : Demande de la racine (cfr. écran 2)
- NIV i : Pour tout composant de niveau i-1 (i > 0) :
  * demande du type du ou des fils du composant traité;
  * demande du ou des fils de ce composant, suivant le type de fils.

  (cfr. écrans 3, 4, 5 et 6)

(remarque : - les types de fils sont pour l'utilisateur :

1. Concaténation;
2. Sélection;
3. Itération;
4. [pas de fils];

- dans le programme, ceux-ci sont augmentés de 1 :

2. Concaténation;
3. Sélection;
4. Itération;
5. [pas de fils]

Le chiffre "1" est réservé au composant ayant à l'origine un type de fils concaténation ou sélection mais n'ayant qu'un seul fils. Cette particularité permettra de mettre en correspondance un fils unique concaténation (respectivement sélection) avec des composants fils d'un autre arbre sélection (respectivement concaténation).

C. Après l'introduction des arbres d'entrée et de sortie, ceux-ci sont présentés sous forme d'expressions B.N.F. (cfr. écrans 9 et 11).

3.2.2.2. Recherche des correspondances formelles

A. Nous proposons deux possibilités de recherche (écran 12) :
   1. L'utilisateur désire introduire toutes les règles de "traduction" au niveau des feuilles (composants élémentaires).

Les feuilles de l'arbre de sortie sont présentées à l'écran avec leur numéro identifiant et leur niveau dans l'arbre. Les feuilles de l'arbre d'entrée sont présentées de la même manière (identifiant - niveau) une à une. On demande si une feuille de l'arbre de sortie peut être mise en correspondance avec une feuille de l'arbre d'entrée. Si oui, on demande cette feuille par son identifiant (cf. écrans 13 et 14).

Une fois ces règles introduites et présentées à l'utilisateur (écran 15), le programme recherche la correspondance générale entre les deux arbres. Si il démontre cette correspondance, il la présente (cfr. écran 16). Cette structure de correspondance peut être enregistrée dans un fichier (cfr. écran 17).

   2. L'utilisateur ne désire introduire aucune règle de transformation.

Le programme recherche la première correspondance générale formelle entre les deux arbres en déterminant les règles de

transformation au niveau des feuilles. Si cette correspondance convient à la spécification de l'utilisateur, on la retient. Si non le programme recherche la solution suivante, etc. (les écrans 16, 18 et 19 sont quelques exemples de correspondances formelles entre OUTPUTFILE et TAPEFILE, qui comptent 21 possibilités).

Si le programme ne trouve pas ou plus de correspondance, il affiche l'écran 20.

B. Les deux types de recherche de correspondance sont expliqués dans le paragraphe 3.2.3.

C. Les composants d'une structure d'arbre de correspondance formelle sont présentés niveau par niveau. Un composant de l'arbre de correspondance est composé d'un composant de l'arbre d'entrée et d'un composant de l'arbre de sortie (composant réel ou fictif "E") (cfr. écrans 16, 18 et 19).

D. L'arbre de correspondance sélectionné par l'utilisateur peut être sauvé sur un fichier (cfr. écran 17).

Cet arbre de correspondance pourra faire l'objet d'une traduction dans un programme PASCAL équivalent par éxécution du programme "CONVPROG".

3.2.3. Technique de recherche des correspondances formelles

3.2.3.1. Règles d'établissement de correspondances

Nous divisons l'exposé du principe d'établissement des correspondances formelles en deux parties suivant les deux possibilités de recherche définies dans le paragraphe 3.2.2.2.A. soient :

- PPR : Première Possibilité de Recherche;
- PSPR : Première solution de la Seconde Possibilité de Recherche;
- SSPR : solution Suivante de la Seconde Possibilité de Recherche.

La première partie concernera l'établissement de la solution de la PPR ainsi que l'établissement de la PSPR; tandis que la seconde partie, se préoccupera de l'établissement de la SSPR.

Partie 1 :

- La démonstration débute par les deux racines. Si celles-ci ont même type de fils, l'arbre de correspondance peut être créé et le premier niveau (niv 0) est établi (et doit encore être prouvé par ses descendants).
- Ensuite, de manière générale, on vérifie la correspondance du ou des fils d'entrée et de sortie du dernier composant de correspondance établi en examinant les types de leurs propres fils (remarque : la correspondance entre deux composants d'entrée et de sortie est dite "établie" lorsqu'ils ont même type de fils (cfr. 3.2.2.1.B.). La correspondance entre deux composants est dite "prouvée" quand leurs descendants ont été prouvés).
  Si le (dernier) composant établi (de niv i-1) a un fils d'entrée et de sortie de type 4 (selection), il existe au maximum une seule solution possible (à établir) de correspondance entre les deux fils qui sera ajoutée au niveau inférieur (niv i) de l'arbre de correspondance.
  Si le (dernier) composant établi (de niv i-1) a des fils d'entrée et de sortie du type 2 (concaténation) ou 3 (sélection), il peut exister plusieurs solutions possibles (à établir) de correspondance entre les différents fils. La première possibilité sera ajoutée au niveau inférieur (niv i) de l'arbre de correspondance.
  Le ou les nouveaux composants devront être vérifiés par analyse de leurs descendants.
  La suite de la démonstration continue avec le premier des fils.

  (N.B. : le numéro de possibilité de correspondance traité entre les fils concaténations ou sélections est enregistré dans le composant de correspondance père de ceux-ci, cfr. type des variables du proramme : STRCOR))

- La preuve d'un composant sélection ou concaténation est réfutée lorsqu'un des composants fils n'a pu être prouvé. Si c'est le cas, cette possibilité de correspondance entre composants fils est supprimée et est remplacée par la possibilité suivante si elle existe et doit être de nouveau démontrée par ses descendants.
  Si la possibilité suivante n'existe pas, le composant

(concaténation ou sélection) (établi) n'a pu être prouvé (on remonte dans l'arbre avec des descendants n'ayant pu être prouvés.

- Une correspondance au niveau des feuilles est prouvée si dans la PPR, elle existe dans les listes de règles de "traduction" défini par l'utilisateur ou dans la PSPR-SSPR, elle est simplement atteinte (elle est considérée comme correcte).

  Le composant de correspondance suivant à analyser est le premier frère ou si il n'existe pas, le premier frère non encore prouvé de ses ascendants.

  S'il n'y en a plus, l'arbre entier de correpondance formelle a été trouvé.

Partie 2 :

- Pour trouver une correspondance formelle suivante (SSPR), on recherche le "dernier" composant de l'arbre de correspondance précédent ayant des 'types de fils' égal à 2 ou à 3. Ce "dernier" composant est le composant de niveau le plus bas et le plus à droite de son niveau et de type fils égal à 2 ou à 3. On supprime de la structure la possibilité de correspondance de ces fils, ainsi que leurs descendants et on les remplace par la possibilité suivante qui doit être prouvée. Si elle peut être prouvée, on a une nouvelle solution. Si pour un composant de type fils concaténation ou sélection il n'existe plus de solution de correspondance pour ces fils, on recherche le composant de 'type fils' 2 ou 3 précédent de même niveau ou sinon, du niveau supérieur.

3.2.3.2. Règles de déduction de correspondance de deux composants

Ces règles utilisées par le programme sont basées sur les règles de déductions développées dans le paragraphe 2.6. Nous les avons regroupées selon le type de composants sur lesquelles elles s'appliquent. Nous expliquerons brièvement les procédures qui les utilisent et qui établissent les correspondances formelles entre deux composants.

A. Composants d'entrée et de sortie de type fils concaténation :

soient A : A1 . A2 . ... An;
B : B1 . B2 . ... Bm;

deux composants à mettre en correspondance.

Il existera une correspondance formelle entre A et B si on peut prouver une correspondance formelle entre les fils de A : Ai, et les fils de B : Bj. En utilisant les règles de déduction et de transformation suivantes, on peut "établir" un niveau supplémentaire de correspondance entre les Ai et Bi :

Règle 1 : de déduction

```
Bi = O(Ai);
A = A1 . A2 . ... An
B = B1 . B2 . ... Bn
--------------------
     O(A) = B
```

Règle 2 : Transformation d'un arbre :

```
A = A1 . A2 . ... An
--------------------
A = A1 . ... Ai . E . Ai+1 . ... An

        (0 <= i <= n)
        (E = composant vide d'un arbre)
```

C'est sur base de ces deux règles que les procédures "CORCONSEL", "AJ1SOLCON" et "POSAIBI" déterminent les correspondances entre les fils des deux composants A et B (et qu'il faudra eux-mêmes démontrer).

- "POSAIBI" : détermine pour tout sous-composant Ai de A quels sont les sous-composants Bj de B susceptibles d'être mis en correspondance avec Ai, y compris le composant vide "E" (cfr. les types de structures 'avab' et 'vab'). En ne tenant compte que des 'types de fils' des composants Ai et Bj (cette procédure ne tient pas compte des règles ci-dessus).

- "AJ1SOLCON" : détermine une à une toutes les possibilités de correspondance des fils des deux composants A et B sur base de "POSAIBI" et des règles 1 et 2 (cfr. un

élément de la liste de type 'PTTPOSC' : 'TTPOS' défini dans les types du programme).

(rem : - l'ordre des sous-composants-séquence a de l'importance; on en tient compte dans la procédure "AJ1SOLCON".
- Il faut ajouter un minimum de composants vides (E) à la liste des fils de l'un des composants pour que les deux composants A et B aient le même nombre de fils en utilisant la règle 2 pour permettre d'appliquer la règle 1 (n = m).
- La possibilité formelle extrême est la correspondance de tous les fils de A avec des composants vides "E", ainsi que tous les composants fils de B avec des composants vides "E".

- "CORCONSEL" : établit la liste de toutes les possibilités de correspondances définies par "AJ1SOLCON" (cfr. le type de structure 'PTTPOSC' dans la définition des types du programme).

B. Composants d'entrée et de sortie de type fils sélection.

soient A : A1 + A2 + ... + An;
B : B1 + B2 + ... + Bm;

deux composants à mettre en correspondance.

Il existera une correspondance globale entre A et B si on peut prouver une correspondance entre les fils de A : Ai, et les fils de B : Bj. En utilisant les règles de déduction et de transformation suivantes, on peut "établir" un niveau supplémentaire de correspondance entre les Ai et Bi :

Règle 3 : de déduction

Bi = O(Ai);
A = A1 + A2 + ... + An
B = B1 + B2 + ... + Bn

O(A) = B

Règle 4 : Transformation d'un arbre :

```
A = A1 + A2 + ... + An
----------------------
A = A1 + ... + Ai + E + Ai+1 + ... + An

        (0 <= i <= n)
        (E = composant vide d'un arbre)
```

Règle 5 : Transformation d'un arbre :

```
A = A1 + A2 + ... + An
----------------------
A = Ai1 + Ai2 + ... + Ain

  (i1, i2, ..., in permutation de 1, 2, ..., n)
```

C'est sur base de ces trois règles que les procédures "CORCONSEL", "AJ1SOLSEL", et "POSAIBI" déterminent une correspondance formelle entre les fils des deux composants A et B (et qu'il faudra eux-mêmes démontrer).

- "POSAIB" : cfr. point A.
- "AJ1SOLSEL" : détermine une à une toutes les possibilités de correspondance des fils des deux composants A et B sur base de "POSAIBI" et des règles 3, 4 et 5 (cfr. un élément de la liste de type 'PTTPOSC' : 'TTPOSC' défini dans les types du programme).

(rem : - l'ordre des sous-composants-sélection n'a pas d'importance; on en tient compte dans la procédure "AJ1SOLSEL".
- Il faut ajouter un minimum de composants vides (E) à la liste des fils de l'un des composants pour que les deux composants A et B aient le même nombre de fils pour appliquer la règle 3 (n = m).
- La possibilité extrême est la correspondance de tous les fils de A avec des composants vides "E", ainsi que tous les composants fils de B avec des composants vides "E".

- "CORCONSEL" : établit la liste de toutes les possibilités de correspondances définies par "AJ1SOLSEL" (cfr. la structure 'PTTPOSC' dans la définition des types du programme).

C. Composants d'entrée et de sortie de 'types fils' itération (tf = 4).

```
soient  A = * A1;
        B = * B1;
```

Pour "établir" la correspondance entre A1 et B1, on applique la règle suivante :

```
Règle 6 : A = * A1;
          B = * B1
          O(A1) = B1
          ----------
          O(A) = B
```

La procédure "AJFNIT14" n'ajoute qu'un composant de correspondance de niveau inférieur contenant le fils de A : A1, et le fils de B : B1,; si ces fils ont eux-mêmes des 'types de fils' compatibles. La correspondance entre A et B ne sera prouvée que lorsqu'on aura démontré la correspondance entre A1 et B1.

D. Cas particulier d'un composant feuille A en entrée (tf = 5) et d'un composant B de sortie de type fils sélection (tf = 3) dont tous les fils (B1, B2, ..., Bn) sont des composants feuilles (tf = 5).

```
soient  A = [pas de fils];
        B = B1 + B2 + ... + Bn;

                (Bi = [pas de fils])
                (1 <= i <= n)
```

Dans ce cas particulier, on peut mettre en correspondance les composants A et B, en démontrant les règles de traduction suivantes :

```
O(A) = B1
O(A) = B2
...
O(A) = Bn
```

3.3. Description du programme "CONVPROG"

3.3.1. Introduction

Le programme "CONVPROG" transforme une structure d'arbre de correspondance (créée par le programme "CORJACK") en un programme PASCAL correspondant. A son niveau de développement,

ce programme n'est qu'une première étape dans la traduction d'un arbre de correspondance de JACKSON en un programme PASCAL compilable. A ce niveau, il ne permet de transformer en une syntaxe correcte qu'un petit nombre de structures d'arbre. La syntaxe PASCAL est beaucoup plus riche que les règles de traduction de base analysées jusqu'à présent. Nous citerons comme exemple la détermination des conditions d'itération et de sélection où seule l'instruction PASCAL conditionnelle envisagée est le "IF ... THEN ... (ELSE ...)" qui ne permet qu'un ou deux sous-composants sélection dans la structure d'arbre, alors que celle-ci peut en compter plus de deux. Les déterminations de ces conditions sont beaucoup plus complexes que les règles déterminées dans les procédures "TRAITSEL" et "TRAITITE". Ces deux procédures peuvent faire l'objet d'un développement plus approfondi.

Nous décrirons de nouveau ce programme en deux parties. L'une sera consacrée au principe de fonctionnement (mode d'utilisation) du programme et l'autre partie expliquera la technique utilisée pour la transformation d'une structure d'arbre de correspondance en un programme Pascal.

3.3.2. Principe de fonctionnement du programme "CONVPROG"

- La première opération est la demande, à l'utilisateur, du fichier sur lequel se trouve l'arbre de correspondance préalablement établi par le programme "CORJACK" (cfr. écran 21)
- Parmi les différentes feuilles de l'arbre de correspondance (composé d'un composant élémentaire de l'arbre d'entrée et d'un composant élémentaire de l'arbre de sortie), on demande de différencier les variables, les constances et les marques de fin de fichier (cfr. écran 24).
  On demande de déterminer la valeur des constantes et de définir le type des variables. (rem : il n'y aura toujours q'un seul type de variable au niveau des feuilles.)
  Cette différence au niveau des feuilles est nécessaire pour pemettre de définir les déclarations du programme, ainsi que pour déterminer les conditions de sélection et d'itération des composants correspondants.
- On demande également le type des fichiers des données en

entrée et en sortie à l'utilisateur (cfr. écran 22)
(Il serait également possible que le progamme détermine lui-même le type de ces fichiers en fonction du type des variables définit précédemment. Cette remarque fait partie d'une amélioration possible de ce programme).

- Le programme Pascal construit sur base de la structure de correspondance est présenté à l'utilisateur et il pourra le sauver sur un fichier (cfr. écran 23) et le consulté ultérieurement (cfr. le programme TEST de l'annexe 5).

3.3.3. Principe de transformation d'un arbre de correspondance en un programme

3.3.3.1. Détermination des conditions d'itération

Il existe 3 possibilités pour déterminer les conditions d'itération d'un composant 'A' de l'arbre de correspondance. La détermination de la condition se fait par l'analyse de certaines feuilles descendantes (feuilles adéquates) : soit du composant A, soit du premier frère de ce composant A. Lorsqu'on recherche les feuilles (adéquates) à analyser, on part du composant A ou du frère de A en descendant (ceci dépend du type de condition que l'on détermine : (cfr. les trois possibilités ci-dessous)) vers les composants feuilles. Si on rencontre un composant de 'type fils' concaténation on ne tiendra compte que du premier fils de ce composant pour la recherche des feuilles; par contre, si ce descendant est de 'type fils' sélection, on tiendra compte de tous ces fils pour la recherche des feuilles.

Possibilités :

1. Si tous les composant de l'arbre d'entrée des feuilles adéquates (descendantes) 'Fi' de A sont des constantes, alors la condition est définie de la manière suivante :

```
WHILE ((X = F1) OR (X = F2) OR ... OR (X = Fn)) DO
```

2. Sinon, si le composant A à un frère 'F' n'ayant pas de fils et dont l'élément d'entrée est une marque de fin de fichier, alors la condition est définie de la manière suivante :

```
WHILE (NOT FF) DO
```

3. Sinon, on analyse les feuilles adéquates 'Fi' du frère de A, si les composants d'entrée de celles-ci (Fi) sont toutes des constantes, alors la condition est définie de la manière suivante :

   WHILE ((X <> F1) AND ... AND (X <> Fn)) DO

3.3.3.2. Détermination des conditions de sélections

La condition de sélection est déterminée dans la procédure "TRAITSEL". La seule instruction conditionnelle envisagée est du type "IF .. THEN ... (ELSE ...)". Elle permet pour un composant de "type fils" sélection d'avoir au maximum deux fils. Tous les composants-sélection possédant plus de deux fils ne sont pas pris en compte dans cette procédure. La condition de sélection devrait être dans ce cas, en langage PASCAL, du type "CASE".

1. Pour déterminer la condition de sélection, on analyse tout d'abord les descendants feuilles adéquats (cfr. 3.3.3.1.) du premier fils du composant de "type fils" sélection.
   Si ceux-ci (Fi) sont toutes des constantes, alors la condition est définie de la manière suivante :

   IF ((X = F1) OR (X = F2) OR ... OR (X = Fn))

2. Sinon, on analyse les descendants feuilles adéquats (cfr. 3.3.3.1.) du deuxième fils du composant de "type fils" sélection.
   Si ceux-ci (Fi) sont toutes des constantes, alors la condition est définie comme :

   IF ((X <> F1) OR (X <> F2) OR ... OR (X <> Fn))

3.3.3.3. Détermination des opérations de base liées aux composants élémentaires.

Soit A, un composant feuille de l'arbre de correspondance contenant un élément de l'arbre d'entrée AE et un élément de l'arbre de sortie AS qui sont mis en correspondance.

1. Si un composant d'entrée AE ou de sortie AS correspond au composant vide "E", alors aucune opération ne leur sera attribuée dans le programme.
2. Si un composant d'entrée AE (respectivemment de sortie AS)

est une marque de fin de fichier, cet élément AE (resp. AS) équivaut à l'opération de fermeture du fichier d'entrée "CLOSE(Fent)" (resp. du fichier de sortie "CLOSE(Fsort)").

3. Si un composant d'entrée AE correspond à une variable ou à une constante, l'opération correspondante est l'opération de lecture du fichier d'entrée "READ(Fent,Elemf)".
4. Si un composant de sortie AS correspond à une variable ou à une constante l'opération correspondante est l'opération d'écriture dans le fichier de sortie de cette variable ou de cette constante "WRITE(Fsort,Var)" ou "WRITE(Fsort,Const)".

3.3.3.4. Opérations d'initialisation d'un programme PASCAL

Elle sont de trois types :

1. L'en-tête du programme :

   "PROGRAM TEST(INPUT,OUTPUT);"

2. Les déclarations des constantes et des variables, ainsi que les fichiers d'entrée et de sortie.

3. Les opérations d'ouverture des fichiers d'entrée et de sortie, ainsi que d'une lecture préalable du fichier d'entrée.

CONCLUSION

En élaborant ce travail, nous avons pu mettre en évidence les difficultés concernant la compréhension et la formalisation de la méthode J.S.P. Nous pouvons préciser que dans chacune des parties, à titre de conclusion, nous avons identifié et classé ces différents problèmes. Nous insistons dans cette dernière section sur les prolongements possibles de ce travail et notamment du formalisme proposé dans la seconde partie.

Les études formelles réalisées par Hughes et les autres auteurs ne recouvrent qu'une partie de la méthode "intuitive" de Jackson. La formalisation demande encore beaucoup de réflexion et de recherche pour permettre de se rapprocher en grande partie de la méthode J.S.P. Notre travail fut orienté dans cette optique c'est-à-dire, une amélioration, une extension de la formalisation et de l'automatisation de la méthode J.S.P. Nous croyons avoir posé les limites de notre formalisme de manière non ambiguë, sur base desquelles nous avons présenté les premières ébauches d'extension.

Néanmoins, quel que soit le niveau auxquelles ces différentes études peuvent arriver, nous ne croyons pas qu'il est possible de formaliser complètement la méthode "intuitive". Par contre, un des avantages des études formelles est qu'elles peuvent conduire à d'autres styles de programmation (impérative, traditionelle, etc.).

L'idée sous-jacente de la formalisation n'est pas de supprimer complètement la tâche du programmeur, mais d'essayer de lui faciliter le travail. Nous pensons que ce travail de programmation sera toujours nécessaire et important et que le fait d'utiliser une méthode (formelle ou non) mènera parfois à de la programmation différente par rapport à la traditionnelle et qu'elle sera plus rentable. Une seconde idée que l'on retrouve dans la formalisation est de permettre une meilleur compréhension de la méthode étudiée ou utilisée.

Nous espérons, pour terminer, que notre étude, même si elle ne mène pas directement à une amélioration de la méthode de programmation structurée de Jackson, contribue largement à cette seconde idée.

REFERENCES

REFERENCES

[DURIEUX, 86] J.L. Durieux : "Ebauche de Formalisation de la Construction de Jackson-Hughes". Acte du 3° colloque de Génie Logiciel, AFCET, Versaille, MAI 86, p 295-304.

[ENSELME, 86] D. Enselme, G. Benay, F.Y. Villemin : "Automatic Program Synthesis from Data Structures". Laboratoire d'intelligence artificielle, PARIS, p361-369.

[GINSBURG, 65] S. Ginsburg : "Mathematical Theory of Context-free Langage". MCGRAW-HILL, NEW-YORK, 1965, pp 97-102.

[HUGHES, 79] J.W. Hughes : "A Formalization and Explication of the Michael Jackson Method of Program Design". SOFTWARE-PRACTICE and EXPERIENCE, VOL 9, 191-202 (1979).

[JACKSON, 75] M.A. Jackson : "Principle of Program Design". Academic Press Inc. (London) LTD, 1975.

[JAVEY, 87] S. Javey : "The Concept of 'Correspondence' in JSP". Proceeding of Tweentith Annual Hawaii International Conference on System Science, 1987, p 14-22.

[LE CHARLIER, 85] B. Le Charlier : "Reflexions sur le problème de la correction des programmes". Thèse de doctorat, FNDP NAMUR, 1985.

[McCULLOCH, 43] McCulloch-Pitts : "A Logical Calculus of the Ideas immanent in Nervous Activity". Bulletin of Math. Biophysius 5, p 115-133 (1943)

[MATHIEU, 86] I. Mathieu : "Conception et réalisation des applications de gestion : une approche basée sur l'explication des raisonnements". Mémoire présenté en juin 86, F.N.D.P.

[MINSKY, 67] M.L. Minsky : "COMPUTATION : Finite and Infinite Machines".
Prentice-Hall, INC. Englewood CLIFFS, N.J., (1967).

[NIVAT, 68] M. Nivat : "Transactions des Langages de Chomsky".
Annales de l'institut Fourier, 18 (1968).

[SOUQUIERE, 82] J. Souquière : "Construction et transformations d'algorithmes itératifs", Centre de Recherche en Informatique de NANCY.
Thèse de doctorat (MARS 82).

[VAN LANSWEERDE, 87] A. Van Lansweerde : Méthodologie de dévelopemment de logiciels.
Notes de cours 1986.

[VAN 'T DACK, 86] R. Van 't Dack : "Jackson Structured Programming".
C.G.E.R. Bruxelles.

[VIVARES, 88] F. Vivares : " Un outil d'aide à la conception de programmes, supportant la méthode de Jackson".
ONERA-CERT, Département d'Etudes et de Recherches en Informatique, TOULOUSE , 2 Février 1988.

[WHIRTH, 73] Whirth : "Systematic Programming : An Introduction."
Prentice-Hall, 1973.

REFERENCES COMPLEMENTAIRES

[COLEMAN, 77] D. Coleman : "The Systematic Design of File-processing Programs". Software-Practice and Experience, vol 7, p 371-381 (1977).

[FINANCE, 85] J. Souquières et J.-P. Finance : "A Method and a Langage for onstructing Iterative Programs". Science of Computer Programming 5 (1985) 201-218, North-Holland.

[HENDERSON, 72] P. Henderson, R. Snowdon : "An Experiment in Structured Programming". BIT 12 (1972), 38-53.

[JANSEN, 82] H. Jansen : "Jackson Structured Programming in een online omgeving". informatie jaargang 24 nr. 6 pag. 313 t/m 369 Amsterdam, juni 1982.

[SOUQUIERES, 1] J. Souquières, B. Huc, P. Lescanne, C. Pair, M. Quere et J.-L. Remy : "Programmation Déductive et Structures de données". CRIN, Institut Universitaire de Technologie, Département Informatique, 2 bis Boulevard Charlemagne - 54000 Nancy, France.

[SOUQUIERES, 2] J. Souquières : "Méthode pour la Construction d'Algorithmes dans le cas de conflit de structures". CRIN, Institut Universitaire de Technologie, Département Informatique, 2 bis Boulevard Charlemagne - 54000 Nancy, France.

[SOUQUIERES, 85] J. Souquières et J.-P. Finance : "Description and Improvement of Iterative Program Transformations". Science of Computer Programming 5 (1985) 233-264, North-Holland.

[WILSON, 84] A. Wilson : "Programs to Process Trees, Representing Program Structures and Data Structures". Software-Practice and Experience, vol 14 (9), p 807-816, sept. 1984.

ANNEXES

ANNEXE 1 :

Exemple de développement d'un programme selon la méthode de base de JACKSON

1. Enoncé de l'exemple. (Van 't dack, 86)

"Le département magasin d'une fabrique s'occupe des sorties et des entrées d'articles. Chaque entrée et chaque sortie est enregistrée sur une carte perforée. Cette carte contient le numéro de l'article, le code opération ("S" pour sortie, "E" pour entrée) et la quantité. Les cartes sont déjà enregistrées dans un fichier et classées dans l'ordre croissant des numéros des articles. Ce fichier est dénommé FSMT (Fichier Stock Magasin Trié).

Le programme à concevoir doit produire une liste résumée RSM (Rapport Stock Magasin) qui donne le solde après opération sur chaque article. Cet aperçu se présente comme suit :

```
                APERCU STOCK MAGASIN

A 2454      SOLDE OPERATION     + 120
A 4852      SOLDE OPERATION     -  45
  ...
  ...
  ...
X 1964      SOLDE OPERATION     + 842

                FIN DU RAPPORT
```

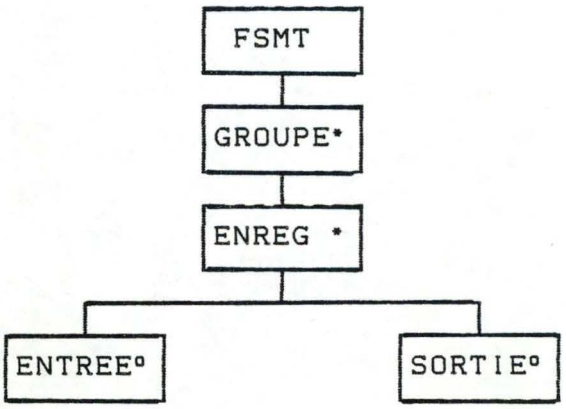
Il n'y a pas lieu de s'intéresser aux sauts de page du rapport RSM. Le rapport est imprimé sous la forme d'une liste continue avec une ligne en-tête, tout au début, et une ligne fin, tout à la fin du rapport.

2.A. Première possibilité de structuration du fichier F.S.M.T. (entrée).

Le fichier FSMT est vu comme une suite de groupes. Un groupe est une suite d'enregistrements (ENREG.) de même numéro d'article.

Un enregistrement représente soit une entrée d'article, soit une sortie d'article. Un enregistrement entrée (ENTREE) contient le numéro de l'article, un code opération égal à "E" et une quantité. Un enregistrement sortie (SORTIE) contient le numéro

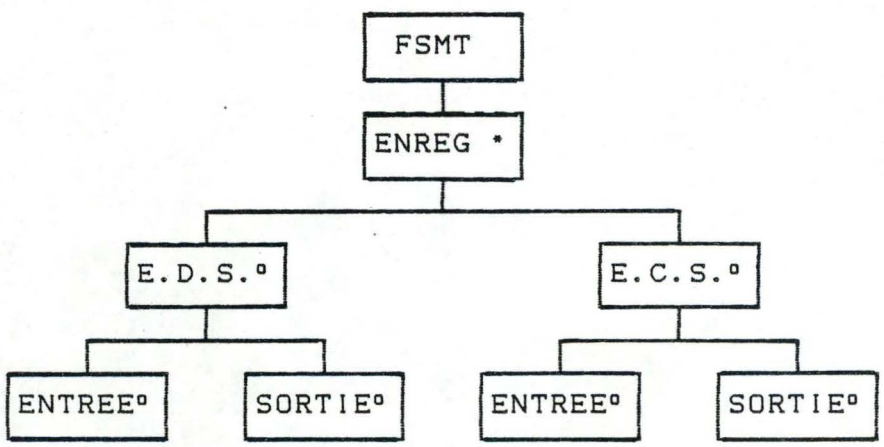
de l'article, un code opération égal à "S" et une quantité.



Remarque : le formalisme proposé ne permet de décrire qu'incomplètement la structure du fichier FSMT : il ne permet pas d'exprimer que celui-ci est trié, ni que tous les enregistrements d'un groupe ont même numéro d'article.

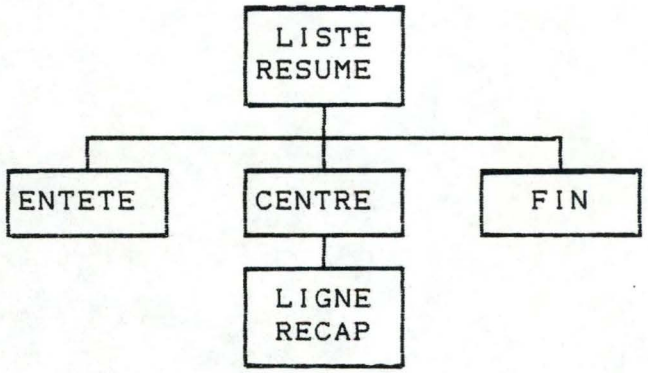
2.B. Deuxième possibilité de structuration du fichier F.S.M.T. (entrée).

Le fichier FSMT est vu comme une suite d'enregistrements. Un enregistrement est soit un enregistrement de début de séquence, (E.D.S.), soit un enregistrement de continuation de séquence (E.C.S.). Un enregistrement de début (continuation) de séquence est un enregistrement qui n'est pas précédé (est précédé) d'un enregistrement de même numéro d'article. Les enregistrements de début ou de continuation de séquence sont soit des enregistrements entrées, soit des enregistrements sorties.
Un enregistrement entrée (sortie) contient le numéro de l'article, un code opération égal à "E" ("S") et une quantité.



On admet, cfr. [Jackson, 75], que cette deuxième façon de structurer le fichier FSMT est beaucoup moins claire (reflète moins la structure du problème) que la précédente. On verra par la suite qu'il est difficile de construire un programme en se basant sur cette deuxième structuration.
On peut déjà tenter de justifier que la première est plus simple en remarquant que, dans ce cas, tout flux constitutif peut être caractérisé à partir de ses composants (par exemple : un groupe est un ensemble d'enregistrements de même numéro) alors que dans le second cas, les composants E.D.S. et E.C.S. sont caractérisés à partir des composants qui les précèdent dans le fichier.
Dans le premier cas, la définition d'un composant se suffit à elle-même (a un sens indépendant du problème général posé) pas dans le second.

3. Structuration du fichier de sortie.



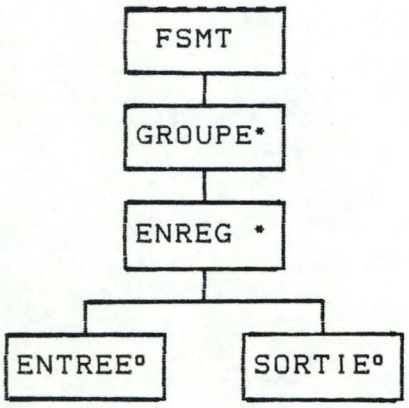
La liste "RESUME" est vue comme la succession d'une ligne en tête de rapport suivie d'un ensemble de lignes récapitulatives suivi d'une ligne de fin de rapport.

Désignons par :

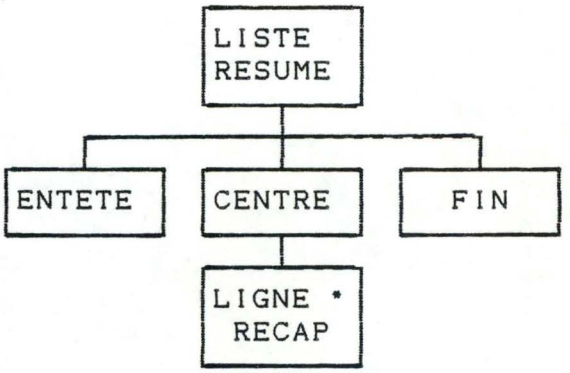
- B l'arbre représentant la structure des sorties;
- A1 l'arbre représentant la première possibilité de structuration des entrées;
- A2 l'arbre représentant la deuxième possibilité de structuration des entrées.

4. Recherche d'une correspondance entre A1 et B.

Rappel A1 :

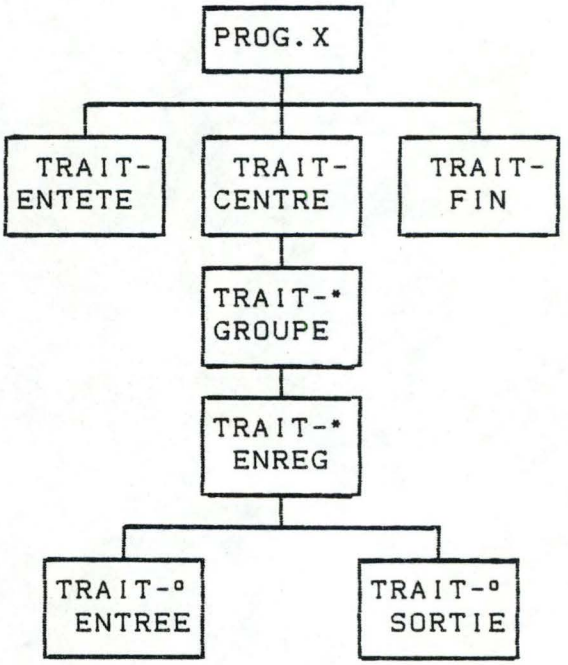


B :



En essayant de mettre les deux arbres A1 et B en correspondance nous obtiendrons un arbre fusion de A1 et B (désigné par C).

C :



En élaborant cette structure intermédiaire C on a bien mis A1 en correspondance avec B.

En effet :

- l'arbre est un élagage de C puisqu'il peut être obtenu à partir de C en supprimant les feuilles TRAIT-ENTETE, TRAIT-FIN et en remplaçant le chemin partant du composant PROG. X et aboutissant au composant TRAIT-GROUPE par un arc.

- l'arbre B est un élagage de C puisqu'il peut être obtenu à partir de C en supprimant les composants TRAIT-ENREG, TRAIT-ENTREE, TRAIT-SORTIE.

D'où on en déduit que A peut-être mis en correspondance formelle avec B. Dans cette correspondance formelle les composants GROUPE, LIGNE RECAP sont en correspondance, ainsi que les composants FSMT, LISTE RESUME.
De plus :

- pour les couples (FSMT, LISTE RESUME), (GROUPE, LIGNE RECAP) de composants de A1 et B en correspondance, il y a toujours autant d'occurrences du composant FSMT (GROUPE) que de LISTE RESUME (LIGNE RECAP)

et si n est ce nombre d'occurrences, $\forall$ i : $1 \leq i \leq n$, la ième occurrence de LISTE RESUME (LIGNE RECAP) peut être générée à partir de la ième occurrence de FSMT (GROUPE).

Il existe donc bien une correspondance entre A1 et B selon la définition donnée au paragraphe 1.2.5.

L'arbre C représente la structure du programme à construire pour résoudre le problème donné. Il reste à ajouter les conditions et actions primitives. Mais avant cela, il est nécessaire d'attacher à chaque composant de l'arbre C une spécification. Cette activité est facilitée, dans le cas présent, par la simplicité des règles de correspondance. Nous le montrerons, ci-dessous, en spécifiant les composants principaux de l'arbre C à savoir TRAIT-CENTRE, TRAIT-GROUPE, TRAIT-ENREG.

Définitions

1. Un groupe est un ensemble d'enregistrements de même numéro d'article, chacun de ces enregistrements représente une opération effectuée sur cet article (opération d'entrée ou de sortie).
2. Une ligne récapitulative est constituée d'un numéro d'article, de la suite de caractères "SOLDE OPERATION" et du montant du solde opération de cet article.
3. Le solde opération d'un article de numéro x est égal à la somme des montants des opérations de sortie effectuées sur le même article.

Spécification de TRAIT-CENTRE

- lit n (≥ 0) groupes et génère n lignes récapitulatives;
- le premier enregistrement du premier groupe a déjà été lu s'il existait sinon la fin de fichier a été déclanchée (précondition).

Spécification de TRAIT-GROUPE

- lit un groupe dont le premier enregistrement a déjà été lu;
- génére la ligne récapitulative correspondant à ce groupe;
- lit le premier enregistrement suivant ce groupe, s'il existe, sinon déclanche la fin de fichier.

Spécification de TRAIT-ENREG.

- reçoit un enregistrement représentant une opération effectuée sur un article de numéro égal à NOART;
- ajoute le montant de cette opération au contenu de la variable SOLDE si c'est une opération entrée, sinon la retire;
- lit l'enregistrement suivant s'il existe, sinon déclanche la fin de fichier.

5. Recherche d'une correspondance entre A2 et B.

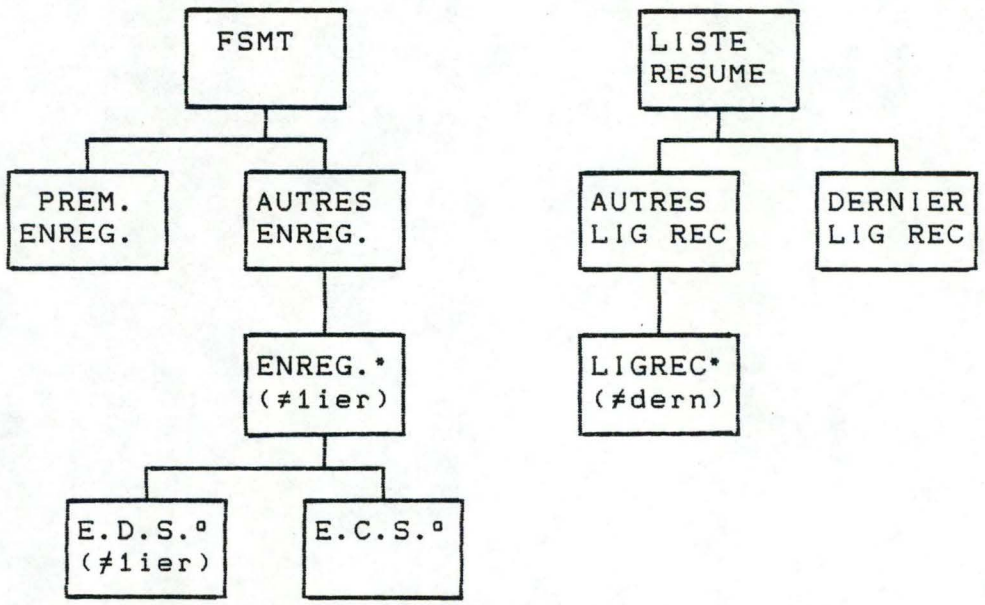
(une compréhension complète de ce paragraphe nécessite la lecture préalable du paragraphe 1.4. et principalement le paragraphe 1.4.3. relatif aux conflits de structure).

Il est imposible de trouver une correspondance entre A2 et B. On dit qu'il y a conflit de structure. On peut essayer de résoudre cette situation en appliquant les solutions habituellement proposées (voir paragraphe 1.4.3.). Mais en suivant cette voie, on se rend compte que la solution reviendrait à créer un flux intermédiaire qui aurait à la fois la "bonne" structure A1 et la "mauvaise" structure A2. Dans ce cas, le programme produisant le flux intermédiaire à partir du flux d'entrée n'aurait d'autre but que de restituer le flux d'entrée tel quel. La solution consiste donc à remplacer la mauvaise structure par la bonne. Pour éliminer le conflit de structure entre A2 et B on pourrait procéder autrement : modifier légèrement les structures en entrée et en sortie en s'arrangeant pour qu'il y ait correspondance. Pour ce faire, on

peut partir d'une idée intuitive de l'exécution d'un programme dont la structure serait proche de A2. Chaque fois que l'on rencontre un enregistrement de début de séquence (E.D.S.) il faut clôturer le groupe précédent en générant la ligne récapitulative lui correspondant et initialiser le nouveau groupe.
Ce traitement ne peut cependant pas être réalisé dans son entièreté lorsqu'on rencontre le premier enregistrement du fichier. En effet, dans ce cas la notion de groupe précédent n'a pas de sens, il est donc nécessaire de créer un traitement spécial pour le premier enregistrement. Ces deux types de traitement ne suffisent pas, car si on s'en contente, la clôture du dernier groupe ne sera jamais réalisée, on créera donc un traitement spécial pour le dernier enregistrement du fichier de sortie.

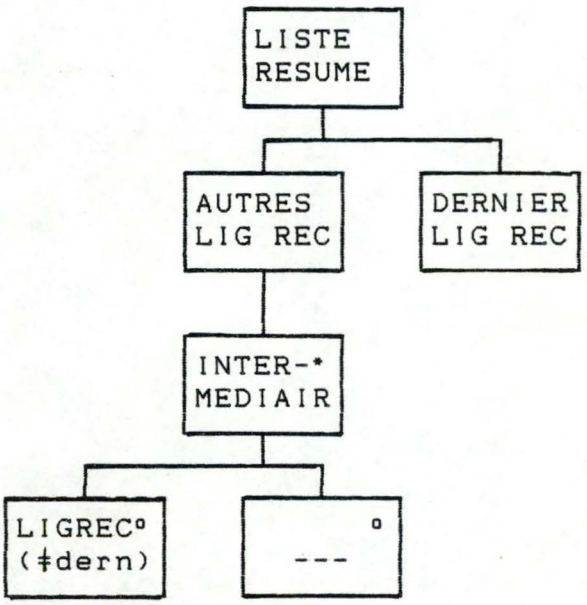
En se basant sur ces idées :
- le fichier d'entrée est vu comme une suite constituée du premier enregistrement suivi des autres enregistrements;
- le fichier de sortie est vu comme une suite constituée de l'ensemble des enregistrements (à l'exception du dernier) suivi de ce dernier enregistrement.

On en déduit les 2 arbres ci-dessous :

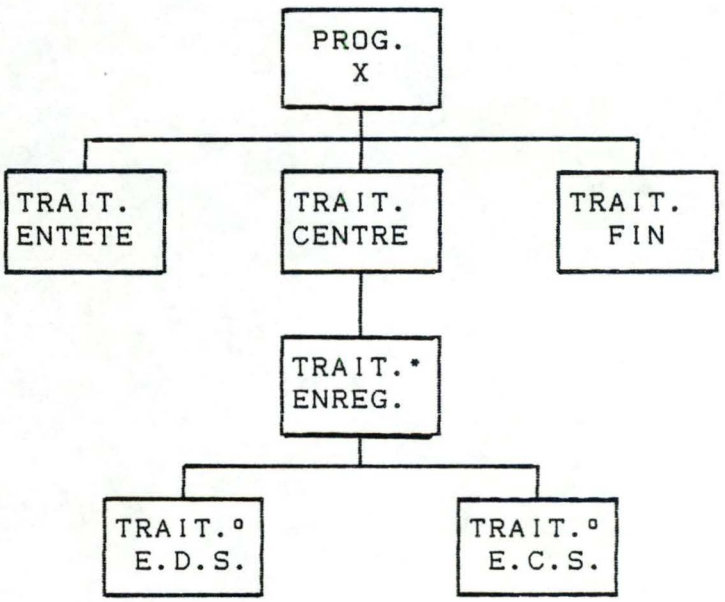


(La définition des flux correspondants aux différents composants se déduit de la discussion précédente).
Ces deux arbres ne peuvent pas encore être mis en correspondance

parce qu'il n'y a pas autant d'occurrences de "lignes récapitulatives" (différentes de la dernière) que d'enregistrements ENREG. (différents du premier) dans le fichier d'entrée. Pour résoudre ce dernier problème il suffit de remplacer le composant "ligne récapitulative" (différente de la dernière) par un composant condition ayant autant d'occurrences que le nombre d'enregistrements du fichier d'entrée, moins un et admettant deux sous-composants dont les occurrences sont soit des lignes récapitulatives soit "vides". On obtient finalement l'arbre de sortie suivant qui peut être mis de manière évidente en correspondance avec l'arbre d'entrée.



En fusionnant les deux arbres on "obtient" la structure suivante pour le programme



Il serait exagéré de dire que cette structure a vraiment été obtenue en appliquant la méthode de Jackson car elle correspond exactement à l'idée intuitive de la page 160.
C'est donc du programme qu'on déduit la structure des fichiers et non l'inverse.
Pour illustrer le fait que cette manière de procéder conduit à un programme plus compliqué et en tout cas moins clair, nous donnons ci-dessous les spécifications des différents composants du programme.

Spécification de TRAIT-ENTETE

(1) ouvre les fichiers d'entrée et de sortie;
(2) génère un enregistrement d'en-tête dans le fichier de sortie;
(3) lit le premier enregistrement du fichier d'entrée et initialise
- une variable NOART avec son n° article,
- une variable SOLDE avec son montant (si c'est une entrée), l'opposé de son montant (si c'est une sortie);
(4) lit le deuxième enregistrement s'il existe (sinon déclenche la fin de fichier).

Spécification de TRAIT-ENREG.

(1) lit les autres enregistrements du fichier d'entrée (jusqu'à déclancher la fin de fichier);
(2) écrit sur le fichier sortie l'ensemble des lignes récapitulatives sauf la dernière;
(3) se termine en ayant plaçé dans NOART et SOLDE le n° d'article et le solde de la dernière ligne récapitulative.

Spécification TRAIT-ENREG.

(1) si l'enregistrement courant (dernier lu) à un n° d'article égal à NOART, son montant est ajouté à la variable SOLDE (si c'est une entrée) et retiré sinon.
Sinon, une ligne récapitulative est écrite sur le fichier de sortie avec le n° d'article NOART et le solde SOLDE; ensuite le montant de l'enregistrement courant est placé dans SOLDE (si c'est une entrée) ou l'opposé de son montant (si c'est une sortie), le numéro d'article de l'enregistrement courant est placé dans NOART.

(2) on lit l'enregistrement suivant du fichier d'entrée s'il existe (sinon la fin de fichier est déclanchée).

Spécification TRAIT-FIN

(1) ecrit une ligne récapitulative avec le numéro d'article NOART et le solde SOLDE;

(2) ecrit une ligne de fin de rapport;

(3) ferme les fichiers.

ANNEXE 2 :

EXEMPLE "ENREGISTREMENT DIRECTEUR"

Enoncé de l'exemple : Le fichier de transactions T contient des enregistrements opérations (type d'enregistrement T1) et des enregistrements directeurs (type d'enregistrement T2 et T3). Les enregistrements directeurs T2 ou T3 sont précédés de 0, 1 ou plusieurs enregistrements opérations T1 (le fichier T est obligatoirement terminé par un des enregistrements T2 et T3). Certains enregistrements du fichier T doivent être transcrits dans le fichier de sortie S, en respectant les règles suivantes :

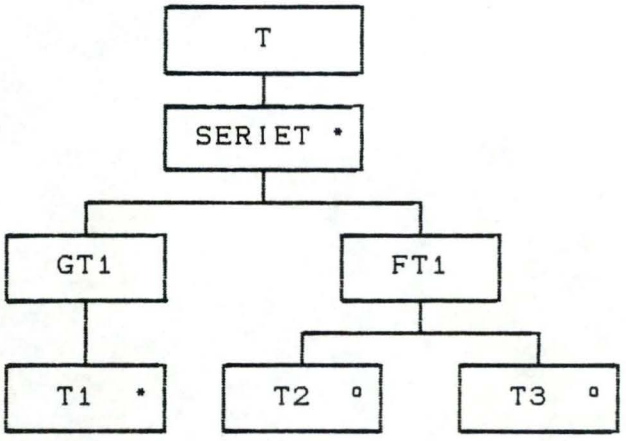
- un enregistrement T1 est transcrit dans le fichier S, si dans le fichier T, le premier enregistrement directeur suivant séquentiellement cet enregistrement T1 est un enregistrement directeur T2;
- un enregistrement T1 n'est pas transcrit dans le fichier S si, dans le fichier T, le premier enregistrement directeur suivant séquentiellement cet enregistrement T1 est un enregistrement directeur T3;
- un enregistrement T2 est transcrit dans le fichier S;
- un enregistrement T3 n'est pas transcrit dans le fichier S;

La description des enregistrements pour le fichier T est la suivante :

```
COMPTE  = RECORD
              TYPE_COMPTE : integer;
                    {1 = T1;
                     2 = T2;
                     3 = T3}
              OPERATION   : array[1..20] of char
          end;
```

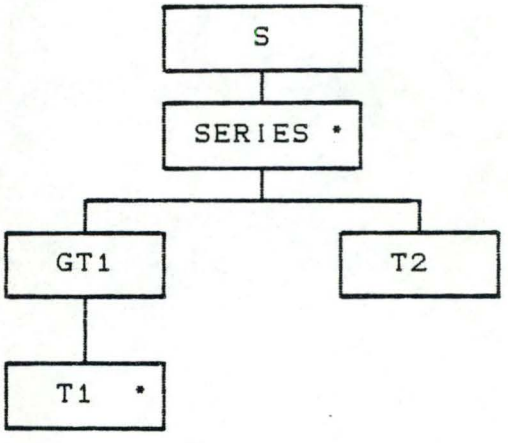
a. Raisonnement ascendant

- La structure logique du fichier T la plus "générale" est :



La propriété d'une série de T1 suivie d'un enregistrement directeur T2 ou T3 est respectée dans cette structure.

- La structure de sortie est vue comme une série de 0, 1 ou plusieurs groupes d'enregistrements constitués d'une suite de 0, 1 ou plusieurs enregistrements T1 suivis d'un enregistrement T2 :



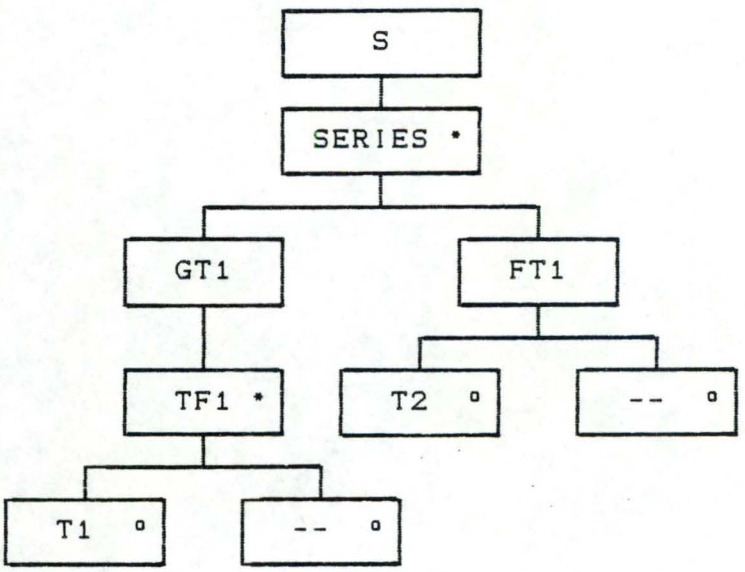
Il n'existe aucune correspondance entre ces deux structures (suivant les règles de correspondance du paragraphe 1.2.5). Les nombres de SERIET et de SERIES ne sont pas identiques (pas de correspondance). On dit qu'il y a conflit de structure . On peut essayer de résoudre ce problème en modifiant légèrement les structures des sorties et des entrées afin d'établir une correspondance.

Pour ce faire, partons de l'idée intuitive d'un progamme dont la structure serait proche de la structure d'entrée (suivant le raisonnement ascendant).

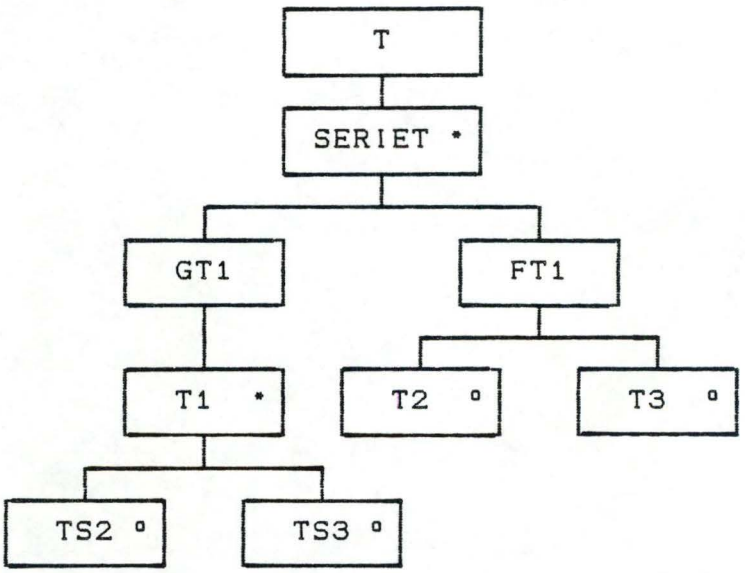
Chaque fois que l'on rencontre un enregistrement T1 faisant partie d'une serie T2, on le recopie dans le fichier S; si il fait partie d'une série T3, il est n'est pas recopié dans S. Si l'on rencontre un enregistrement T2, on le recopie dans S, et si c'est T3 que l'on rencontre, on ne le recopie pas.

En se basant sur ces idées, certaines modifications des structures peuvent être apportées :

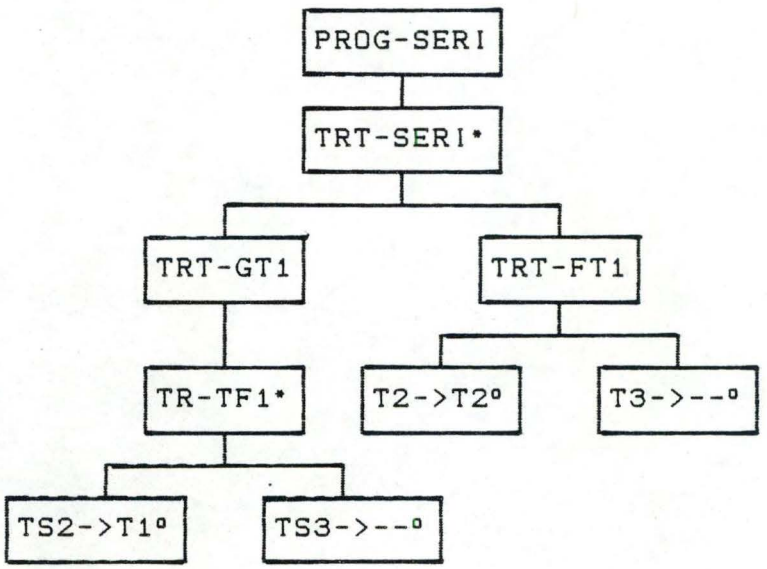
- Les enregistrements T1 du fichier T sont vus soit comme faisant partie d'une série T2, soit de la série T3;
- Le composant T1 du fichier de sortie S est remplacé par un composant-condition fictif TF1 intermédiaire ayant autant d'occurrences que le nombre d'enregistrements T1 du fichier d'entrée. Ce composant fictif admet deux sous-composants dont les occurrences sont soit des enregistrements T1, soit "vide".



La structure du fichier d'entrée, après ces modifications, se présente de la manière suivante :



En intégrant les deux arbres on "obtient" pour le programme la structure suivante :



TRT-GT1 : traitement du groupe de T1;
TRT-FT1 : traitement de la fin de la série T1 (T2 ou T3;
TRT-TF1 : traitement de l'enregistrement T1 fictif;
TS2->T1 : traitement d'un T1 appartenent à une série terminée par T2 {WRITE(S,T1)};
TS3->-- : traitement d'un T1 appartenant à une série terminée par T3 (non-transcrit dans S);
T2->T2 : traitement d'un T2 transcrit dans S;
T3->-- : traitement d'un T3 non-transcrit dans S.

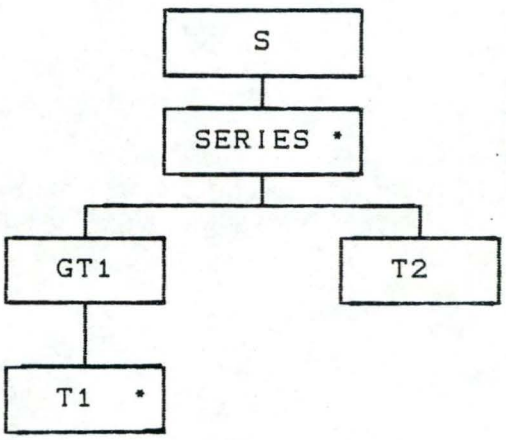
Remarque : nous supposons que lors du traitement d'un enregistrement T1 de T, le programme sait identifier si cet enregistrement fait partie d'une série T2 ou d'une série T3. Si ce n'est pas possible de le faire avec la

technique "d'une lecture préalable", nous verrons dans le paragraphe 1.4.2. une autre technique permettant d'évaluer de telles conditions de sélection basées sur des lectures multiples.

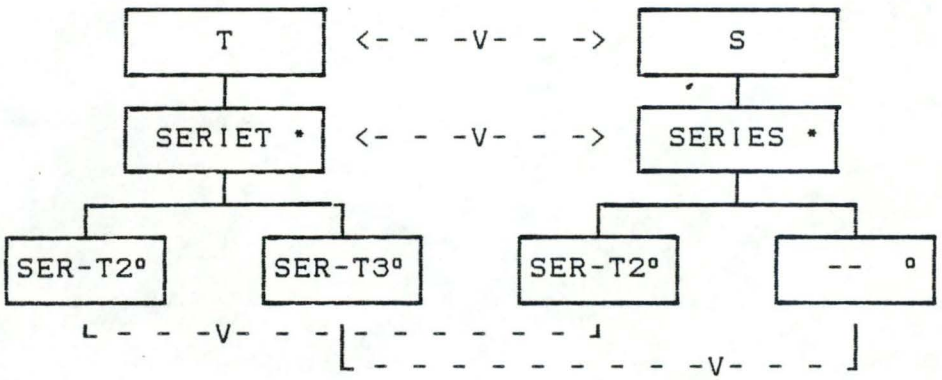
Remarquons à nouveau que les structures des données ne sont pas "optimum" et qu'elles demandent des adaptations pour structurer le programme. Nous pouvons également faire apparaitre dans cette structure de programme un inconvénient supplémentaire lié à la technique d'évaluation des conditions de sélection demandant une lecture de plusieurs enregistrements. A chaque enregistrement T1 de T, le programme devra effectuer une lecture des T1 suivants jusqu'à l'enregistrement directeur T2 ou T3 permettant d'identifier sa série. Les enregistrements T1 seront lus n fois (n étant le nombre d'enregistrement T1 précédant dans sa série cet enregistrement analysé).
Nous verrons que dans la solution suivante, l'inconvénient de lecture multiple d'un même enregistrement sera supprimé.

b. Raisonnement descendant

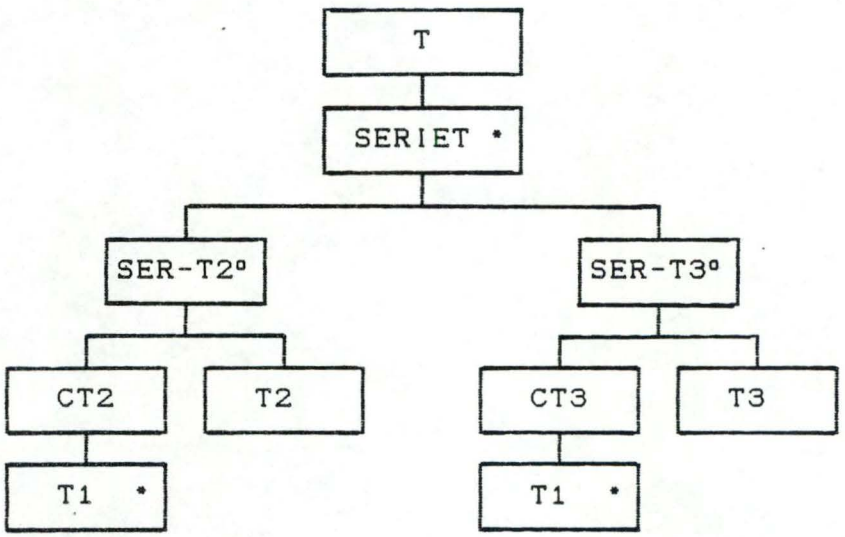
Sur base de la structure de sortie :



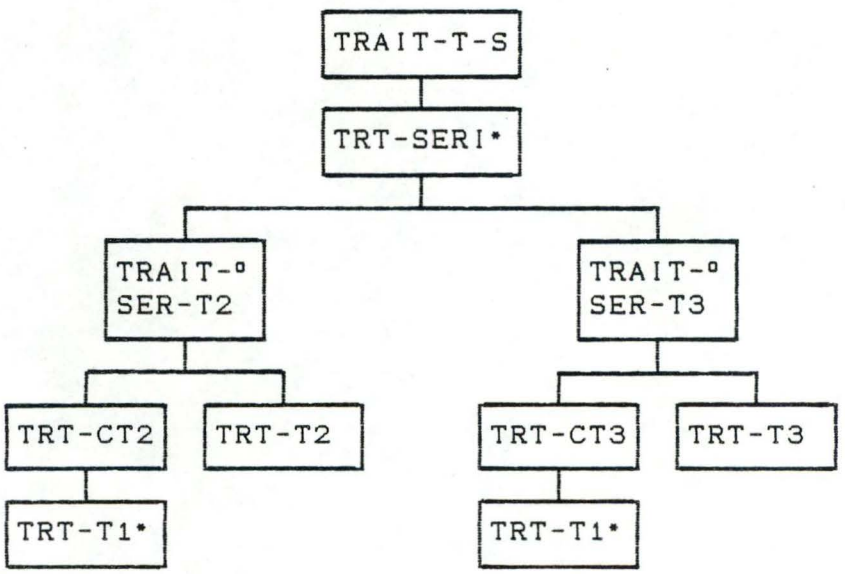
ne possédant que des séries T2, nous pouvons structurer les entrées en faisant la différence entre les séries T2 et les séries T3. Les composants SERIET et SERIES peuvent, dés lors, être mis en correspondance :



Les séries T2 (respectivement T3) sont composées d'une série de T1 suivie de T2 (resp. T3) :



Sur base de la correspondance entre les series T2 et T3 (SER-T2 <--> SER-T2) et (SER-T3 <--> SER-T3) la structure du programme en est "déduite", et se présente de la manière suivante :



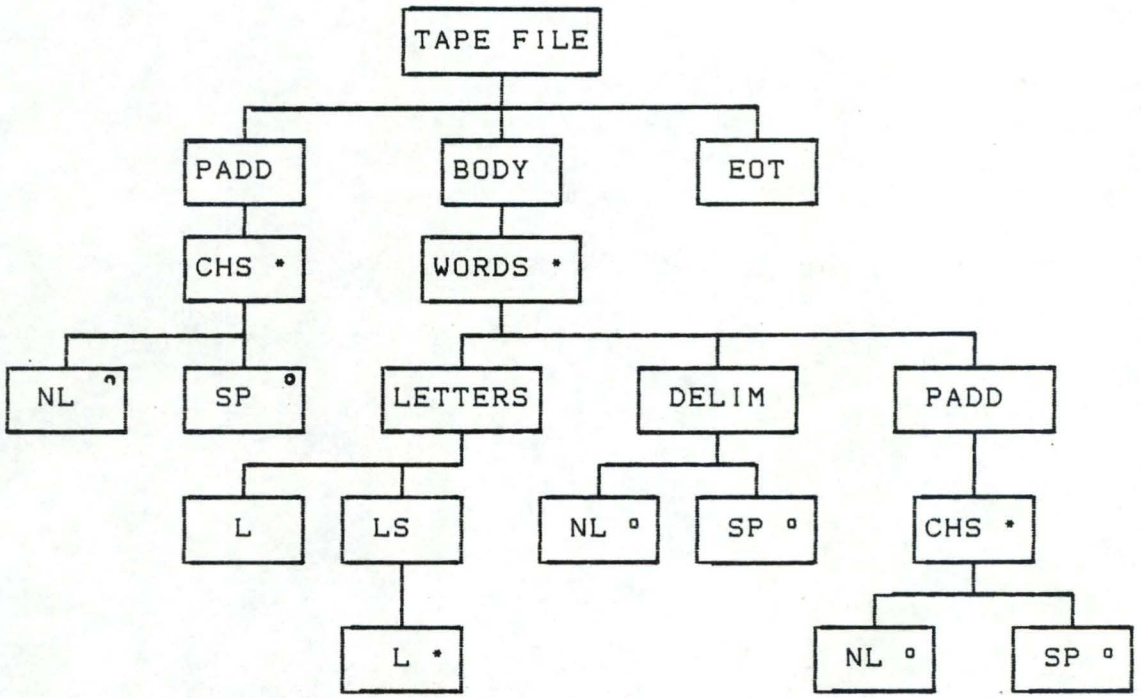
L'avantage de cette solution, par rapport à la première version, est que nous évaluons par une lecture de plusieurs enregistrements les conditions de la sélection d'une série (T2 ou T3). Nous avons une première lecture d'un enregistrement T1 pendant l'évaluation de la série et une seconde lecture de l'enregitrement T1 pendant le traitement proprement dit de cet enregistrement (écriture dans S si c'est une série T2, suivie d'une lecture de l'enregistrement suivant; seulement une lecture de l'enregistrement suivant si c'est une série T3).

ANNEXE 3 :

TRANSFORMATION D'UN ARBRE DE JACKSON EN UNE EXPRESSION REGULIERE :

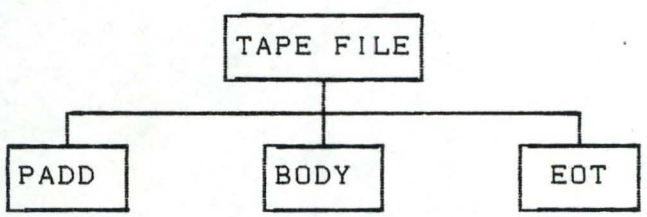
"TRAITEMENT DE TEXTE"

Considérons l'arbre d'entrée "TAPE FILE" suivant :



Suivant les règles de transformation définies dans le paragraphe 2.7., nous pouvons transposer niveau par niveau, l'arbre "TAPE FILE" en une expression régulière correspondante :

1. Point de départ : (racine de l'arbre)



L'expression régulière intermédiaire correspondante au premier niveau est :

(PADD . BODY . EOT)

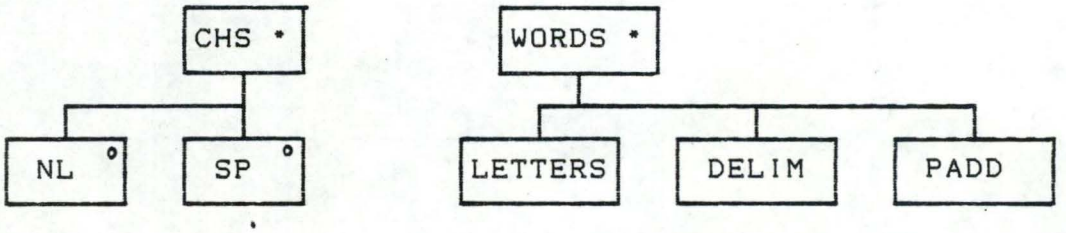
2. Deuxième niveau :

(CHS)* (WORDS)*

L'expression régulière intermédiaire correspondante à l'arbre :

((CHS)* . (WORDS)* . EOT)

3. Troisième niveau :

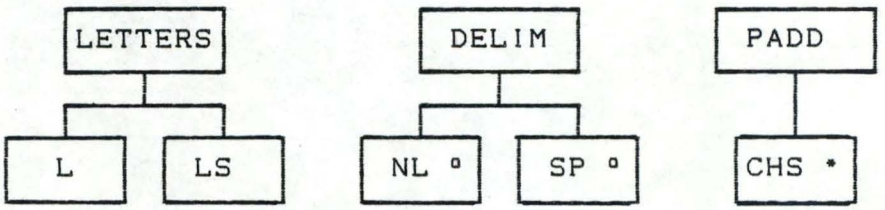


(NL / SP) (LETTERS . DELIM . PADD)

L'expression régulière intermédiaire correspondante à l'arbre devient :

((NL / SP)* . (LETTERS . DELIM . PADD)* . EOT)

4. Quatrième niveau :

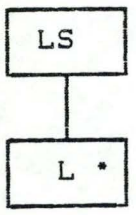


(L . LS) (NL / SP) (CHS)*

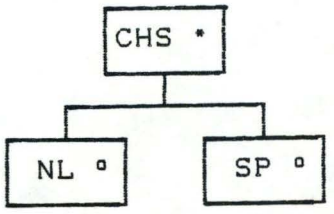
L'expression régulière intermédiaire correspondante à l'arbre devient :

((NL/SP)*.((L.LS).(NL/SP).(CHS)*)*.EOT)

5. Cinquième niveau :



(L)*



(NL / SP)

L'expression régulière correspondante à l'arbre de Jackson "TAPE FILE" est :

((NL/SP)*.((L.(L)*).(NL/SP).(NL/SP)*)*.EOT)

L'expression régulière correspondante à l'arbre de sortie "OUTPUTFILE" de l'énoncé du paragraphe 2.3.2. est déduite de manière similaire :

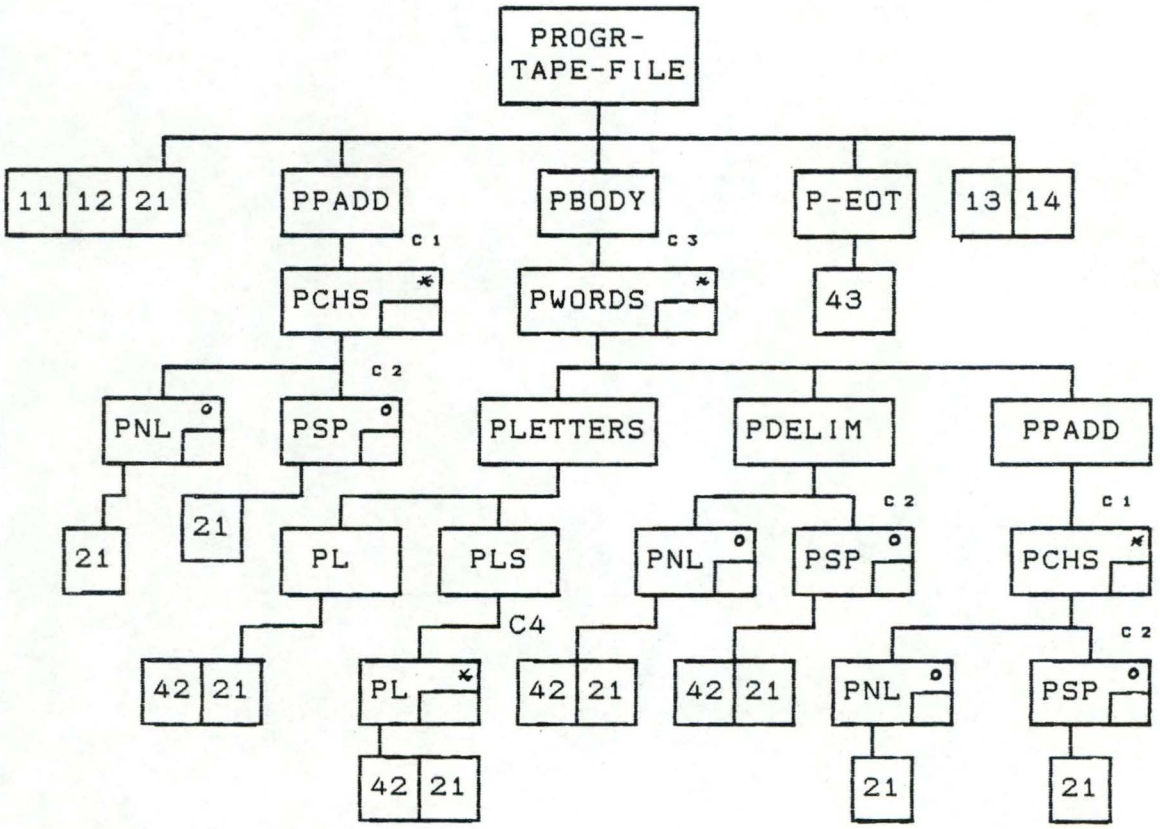
((L.L*).EW)*.EOT

ANNEXE 4 :

EXEMPLE : REGLES DE CONSTRUCTION DE PROGRAMMES

Nous considérons la structure du programme, correspondant à l'énoncé du paragraphe 2.3.2., obtenue après la quatrième étape (c'est-à-dire la structure du programme complétée par les conditions d'itération et de sélection, ainsi que les actions primitives).

a. Structure complète du programme :



b. Opérations de base :

1. 11 rewrite(outputfile)
   12 reset(tapefile)
   13 close(outputfile)
   14 close(tapefile)

2. 21 pread(tapefile,ch)

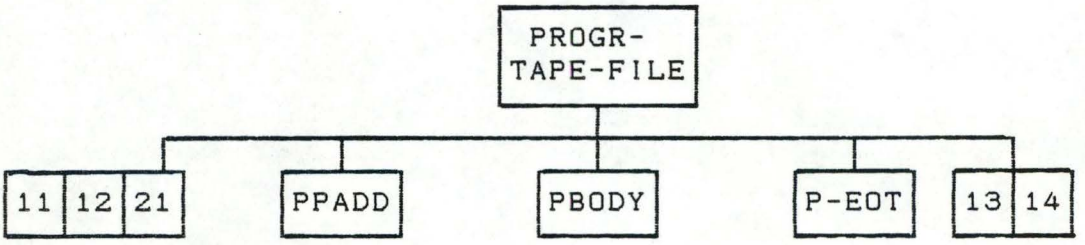
4. 41 write(outputfile,ew)
   42 write(outputfile,ch)
   43 write(outputfile,eof)

c. Conditions d'itération et de sélection :

```
C1 :  while (((ch = nl) or (ch = sp)) and
              (not ff)) do
C2 :  if (ch = nl)
C3 :  while (not ff) do
C4 :  while (ch = l) do

(ff : boolean; {fin de fichier})
```

Le pseudo-code Pascal de cette structure est obtenu en appliquant les règles de base de construction définies dans le paragraphe 2.7. A chaque étape, nous indiquerons la règle utilisée. Nous proposerons une dernière version du programme en appliquant les règles simplificatrices.

(a) Composant de départ : PRG-TAPEFILE (règle 5)



Le pogramme correspondant (règle 1) :

```
Program TAPE_FILE

begin
   proc 11;    }
   proc 12;    } ---> Composants Feuilles
   proc 21;    }

   proc PPADD;  }
   proc PBODY;  } ---> Composants intermédiaires
   proc P-EOT;  }

   proc 13;    }
   proc 14     } ---> Composants Feuilles
end.
```

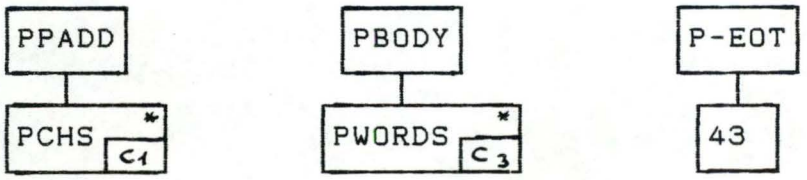
(b) Les composants feuilles deviennent (règle 4) :

```
proc 11 ::= rewrite(output)
proc 12 ::= reset(tapefile)
proc 21 ::= pread(tapefile,ch)
proc 13 ::= close(outputfile)
proc 14 ::= close(tapefile)
```

Le programme correspondant devient :

```
Program TAPE_FILE

begin
   rewrite(outputfile);
   reset(tapefile);
   pread(tapefile,ch);
   proc PPADD;
   proc PBODY;
   proc P-EOT;
   close(outputfile);
   close(tapefile)
end.
```

(c) Développement de "PPADD", "PBODY" et "P-EOT" :

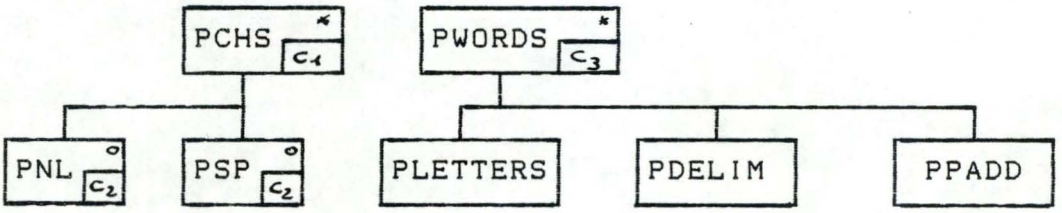



```
proc PPADD ::=

   while (((ch = nl) or (ch = sp)) and
          (not ff)) do
      proc PCHS

      (règle 3)


proc PBODY ::=

  while (ch <> eot) do
     proc PWORDS

     (règle 3)


proc P-EOT ::= proc 43

   (règle 1')
```

Le programme correspondant devient :

```
Program TAPE_FILE

begin
   rewrite(outputfile);
   reset(tapefile);
   pread(tapefile,ch);
   while (((ch = nl) or (ch = sp)) and
          (not ff)) do
      proc PCHS;
   while (ch <> eot) do
      proc PWORDS;
   write(outputfile,eof);
   close(outputfile);
   close(tapefile)
end.
```

(d) Développement de "PCHS" et de "PWORDS" :



- proc PCHS ::=

```
if (ch = nl) then pread(tapefile,ch)
             else pread(tapefile,ch)
```

(règles 2 et 4)

Le "proc PCHS" peut être simplifié par l'instruction (règle 2') :

```
proc PCHS ::= pread(tapefile,ch)
```

- proc PWORDS ::=

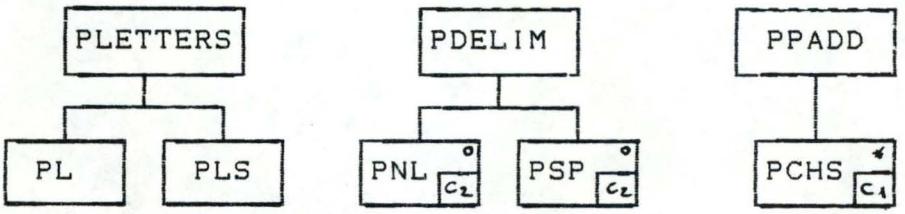
```
begin
   proc PLETTERS;
   proc DELIM;
   proc PPADD
end
```

(règle 1)

Le programme correspondant à l'étape de développement :

```
Program TAPE_FILE

begin
   rewrite(outputfile);
   reset(tapefile);
   pread(tapefile,ch);
   while (((ch = nl) or (ch = sp)) and
          (not ff)) do
      pread(tapefile,ch);
   while (ch <> eot) do
   begin
      proc PLETTERS;
      proc PDELIM;
      proc PPADD
   end;
   write(outputfile,eof);
   close(outputfile);
   close(tapefile)
end.
```

(e) Développement de "PLETTERS", "PDELIM" et "PPADD" :



- proc PLETTERS ::=

```
begin
   proc PL;
   proc PLS
end
```

(règle 1)

- proc PDELIM ::=

```
if (ch = nl) then proc PNL
             else proc PSP
```

(règle 2)

- proc PPADD ::=

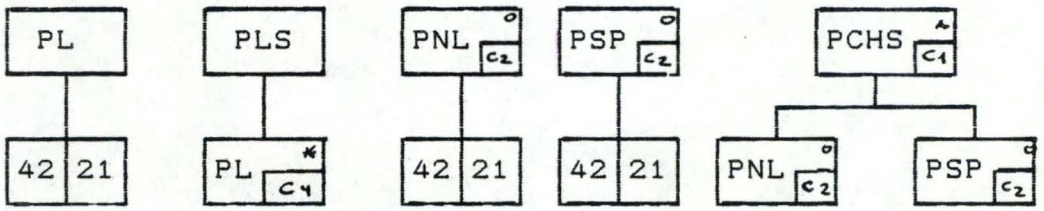
```
while (((ch = nl) or (ch = sp)) and
       (not ff)) do
   proc PCHS
```

(règle 3)

Le programme correspondant à l'étape de développement :

```
Program TAPE_FILE

begin
   rewrite(outputfile);
   reset(tapefile);
   pread(tapefile,ch);
   while (((ch = nl) or (ch = sp)) and
          (not ff)) do
      pread(tapefile,ch);
   while (ch <> eot) do
   begin
      begin
         proc PL;
         proc PLS
      end;
      if (ch=nl) then proc PNL
                 else proc PSP;
      while ((ch=nl)or(ch:sp)) do
         proc PCHS
   end;
   write(outputfile,eof);
   close(outputfile);
   close(tapefile)
end.
```

(f) développement de "proc PL", "proc PLS", "proc PNL", "proc PSP" et "proc PCHS" :



- proc PL ::=

```
begin
   write(outputfile,ch);
   pread(tapefile,ch)
end
```

(règles 1 et 4)

- proc PLS ::=

```
while (ch=1) do
   proc PL
```

(règle 3)

- proc PNL ::=

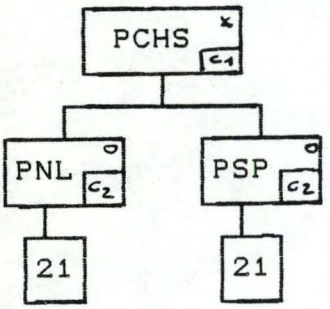
```
begin
   write(outputfile,ew);
   pread(tapefile,ch)
end
```

(règles 1 et 4)

- proc PSP ::=

```
begin
   write(outputfile,ew);
   pread(tapefile,ch)
end
```

(règles 1 et 4)



- proc PCHS ::= pread(tapefile,ch)

(règles 2, 2'et 1')

Le programme complet "TAPE FILE" correspondant est :

```
Program TAPE_FILE

begin
   rewrite(outputfile);
   reset(tapefile);
   pread(tapefile,ch);
   while (((ch = nl) or (ch = sp)) and
          (not ff)) do
      pread(tapefile,ch);
   while (ch <> eot) do
   begin
(1)   begin
         begin
            write(outputfile,ch);
            pread(tapefile,ch)
         end;
         while (ch=1) do
         begin
            write(outputfile,ch);
            pread(tapefile,ch)
         end;
      end;
      if (ch=nl)
        then begin
                write(outputfile,ew);
(2)             pread(tapefile,ch)
             end
        then begin
                write(outputfile,ew);
(2)             pread(tapefile,ch)
             end;
      while ((ch=nl)or(ch:sp)) do
        pread(tapefile,ch)
   end;
   write(outputfile,eof);
   close(outputfile);
   close(tapefile)
end.
```

(g) Le progamme "TAPE FILE" peut être modifié à partir des règles de simplification de la manière suivante :

```
Program TAPE_FILE

begin
   rewrite(outputfile);    }
   reset(tapefile);        }  (1) règle 3'
   pread(tapefile,ch);     }
   while (((ch = nl) or (ch = sp)) and
          (not ff)) do
      pread(tapefile,ch);
   while (ch <> eot) do
   begin
      write(outputfile,ch);
      pread(tapefile,ch);
      while (ch=1) do
      begin
         write(outputfile,ch);
         pread(tapefile,ch)
      end;
      write(outputfile,ew);        }(2) règle 2'
      pread(tapefile,ch);          }
      while (((ch = nl) or (ch = sp)) and
             (not ff)) do
            pread(tapefile,ch)
   end;
   write(outputfile,eof);
   close(outputfile);
   close(tapefile)
end.
```

A N N E X E 5

- Définitions des types des programmes "CORJACK" et "CONVPROG";

- Présentation des écrans du mode de fonctionnement des deux programmes;

- Présentation du résultat du programme "CONVPROG" : "programme test".

DEFINITION DES TYPES DU PROGRAMME "CORJACK"

Partie 1 : Constitution des arbres d'entrée et de sortie :

* COMPOSANT D'UN ARBRE : cmpa

```
pcmpa  =  ^cmpa;

cmpa   =  record                    (*COMPOSANT D'UN ARBRE*)
             niv   : integer;        (*niveau d'un composant d'un arbre *)
             nomco : str10  ;        (*nom du composant                  *)
             idf   : integer;        (*identifiant d'une feuille        *)
             pp    : pcmpa  ;        (*pointeur vers le composant père   *)
             tf    : integer;        (*type des composants fils (1..5)  *)
             pfl   : pcmpa  ;        (*pointeur vers les fils           *)
             pfr   : pcmpa           (*pointeur vers le frère           *)
             end;

(rem : type des composant fils : 1 - Sélection ou Concaténation avec
                                     un seul fils.
                                 2 - CONCATENATION.
                                 3 - SELECTION.
                                 4 - ITERATION.
                                 5 - [pas de fils]  )
```

* LISTE DES FEUILLES D'UN ARBRE : pcfa

```
pcfa   =  ^pfeuil;

pfeuil =  record
             feui  : cmpa;      (*composant feuille d'un arbre      *)
             pfsu  : pcfa       (*pointeur vers le composant feuille*)
                                (*suivant                           *)
          end;
```

* LISTE DES FONCTIONS DE CORRESPONDANCES DES FEUILLES DE L'ARBRE D'ENTREE VERS DES FEUILLES DE L'ARBRE DE SORTIE : pcorab

```
pcorab = ^corab;

corab  = record              (*CORRESPONDANCE DES FEUILLES DE 2 ARBRES*)
            ida   : integer;
               (*identifiant  d'une  feuille de  l'arbre d'entrée*)
            idb   : integer;
               (*                                   de sortie*)
                  cnext : pcorab
         end;
```

* LISTE DES FONCTIONS DE CORRESPONDANCES DES FEUILLES D'UN ARBRE VERS UN COMPOSANT FICTIF DE L'ARBRE OPPOSE : pcorc

```
pcorc  = ^corc;

corc   = record              (*CORRESPONDANCE F-A-& ET F-&-B*)
               idc  : integer;
                     (*identifiant d'une feuille d'un arbre*)
               nex  : pcorc
         end;

              (*******************************)
```

Partie 2 : Constitution des arbres de correspondances formelles :
========

* COMPOSANT D'UN ARBRE DE CORRESPONDANCES : strcor

```
pstrcor = ^strcor;  (*pointeur vers un élément d'une structure de  *)
                    (*correspondance                               *)

strcor  = record  (*structure d'une correspondance entre 2 composants*)
               elema : cmpa;     (*élément de l'arbre d'entrée       *)
               elemb : cmpa;     (*                  de sortie       *)
               pns   : pstrcor; (*ptr vers un élément de niv. sup.   *)
               pni   : pstrcor; (*                            inf.   *)
               pnm   : pstrcor; (*                       de même niv. *)
               pos   : integer  (*posibilité de correspondances des *)
                                (*fils traités                       *)
          end;

pnivcor = ^nivcor;

nivcor  = record   (*niveau de correspondance de l'arbre*)
               ns  : pnivcor;  (*ptr de niveau supérieur          *)
               ni  : pnivcor;  (*              inférieur          *)
               nc  : pstrcor;  (*              correspondant      *)
               niv : integer   (*niveau de correspondance         *)
         end;
```

* LISTE D'UNE POSSIBILITE DE CORRESPONDANCES ENTRE LES COMPOSANTS FILS D'UN COMPOSANT SELECTION OU ITERATION : pposcor

```
pposcor = ^poscor;  (*pointeur vers une possibilité de correspondance*)
                    (*d'un fils de niveau i+1 d'un composant de niv i*)

poscor  = record
               posci : strcor ;     (*possibilité du composant i*)
               nexci : pposcor      (*ptr du composant i+1       *)
         end;
```

* LISTE DES POSSIBILITES DE CORRESPONDANCES ENTRE LES COMPOSANTS FILS D'UN COMPOSANT SELECTION OU ITERATION : pttposc

```
pttposc = ^ttposc;    (* ptr vers une possibilité complète de      *)
                      (* correspondance des fils des composants E-S*)

ttposc  = record
              psol  : pposcor ; (*ptr vers la solution de corresp*)
                                (*des premiers composants A-B    *)
              ppcas : pttposc   (*ptr vers la solution possible  *)
          end;                  (*suivante                       *)
```

* LISTE DES COMPOSANTS FILS D'UN COMPOSANT SELECTION OU CONCATENATION D'UN ARBRE D'ENTREE AUQUELS ON A ASSOCIE UNE LISTE DES COMPOSANTS FILS D'UN COMPOSANT CONCATENATION OU SELECTION DE L'ARBRE DE SORTIE POUVANT ETRE MIS EN CORRESPONDANCE : pavab

```
pavab = ^avab;

pvab  = ^vab ;

avab  = record      (*élément fils d'un composant pouvant être mis en *)
                    (*correspondance avec des composants bi           *)
            c   : integer;
            nbrc: integer;
            ai  : cmpa ;
            pbi : pvab ;  (*ptr vers le 1er composant 'b' mis       *)
                          (*en correspondance                       *)
            paj : pavab;  (* élément fils suivant du composant 'a'  *)
            pah : pavab   (*              précédent                  *)
      end;

vab   = record              (* élément fils d'un composant b mis en   *)
                            (*correspondance avec un composant a     *)
            no  : integer;
            bi  : cmpa   ;
            pbj : pvab
      end;
```

* TYPE DES DIFFERENTS FICHIERS

```
ficha   = file of cmpa;     (*fichier contenant une structure d'arbre *)
fichbnf = text        ;     (*fichier contenant des expressions B.N.F.*)
                            (*d'une structure d'arbre                 *)


fac = file of strcor;       (*fichier contenat une structure de       *)
                            (*correspondance entre deux arbres        *)
```

* **REPRESENTATION D'UN COMPOSANT DE JACKSON SOUS FORME D'EXPRESSION B.N.F.** :

```
pbnf = ^bnf;                    (*définition des fils d'un composant d'un *)
                                (*arbre sous forme B.N.F.                 *)
bnf  = record
          niv     : integer;
          backus  : str100 ;
          pnex    : pbnf
       end;

(*************************************************************************)
(*************************************************************************)
```

DEFINITION DES TYPES DU PROGRAMME "CONVPROG"
==========

Partie 3 : Conversion d'une structure de correspondance de JACKSON en un programme PASCAL :
========

* **LISTE DES LIGNES D'UN PROGRAMME PASCAL** : plib

```
plib = ^lib;

lib = record
         ligne : str100;  (*Ligne de maximum 100 characteres*)
         pls   : plib     (*pointeur vers la ligne suivante *)
      end;
```

* **LISTE DES FEUILLES D'UN ARBRE** : plfu

```
plfu  = ^lfu;

lfu   = record
           fu    : str10  ;   (*nom d'une feuille                 *)
           arbre : integer;   (*arbre = 1 : arbre d'entrée, ou   *)
                              (*arbre = 2 : arbre de sortie      *)
           pfus  : plfu
        end;
```

* **LISTE DES DECLARATIONS D'UN PROGRAMME PASCAL** : pldecl

```
pldecl = ^ldecl;

ldecl  = record
            decl   : str100 ; (*déclaration Pascal                *)
            feu    : str10  ; (*nom de la feuille correspondante *)
            arbre  : integer;
            pdes   : pldecl
         end;
```

* **LISTE DE POINTEURS DE COMPOSANT DE CORRESPONDANCE** : pelpif

```
pelpif = ^elpif;

elpif  = record
               elp : pstrcor;
               pn  : pelpif
         end;
```

* **TYPE DE FICHIER RESULTAT** :

```
fac = file of strcor;   (*fichier d'un arbre de correspondance       *)

(*************************************************************************)
```

POSSIBILITE DE CHOIX POUR LA LECTURE DE L'ARBRE D'ENTREE

1. l'arbre se trouve sur un fichier

2. vous préférez introduire un nouvel arbre

PREFEREZ-VOUS LA PREMIERE SOLUTION (O/N) :

ECRAN 1

ARBRE D'ENTREE

RACINE DE L'ARBRE : TAPEFILE

ECRAN 2

```
                    ARBRE D'ENTREE : TAPEFILE

Demande des fils de : TAPEFILE      de NIVEAU 0

      1. quel est le type des fils : 2
           (1 - 2 - 3 - 4 ou 5)
      2. quel est le fils : /
          si plus de fils tapez : "/"

-------------------------------------------------------------------

 TAPEFILE ::= ( PADD + BODY + EOT )

                                          Type de composants fils

                                             1 : Terminal
                                             2 : Concaténation
                                             3 : Sélection
                                             4 : Itération
                                             5 : pas de fils

                    PRESSEZ UNE CLE
```

ECRAN 3

```
                    ARBRE D'ENTREE : TAPEFILE

Demande des fils de : PADD      de NIVEAU 1

      1. quel est le type des fils : 4
           (1 - 2 - 3 - 4 ou 5)
      2. quel est le fils : CHS

-------------------------------------------------------------------

 PADD ::= ( * CHS )

                                          Type de composants fils

                                             1 : Terminal
                                             2 : Concaténation
                                             3 : Sélection
                                             4 : Itération
                                             5 : pas de fils

                    PRESSEZ UNE CLE
```

ECRAN 4

ARBRE D'ENTREE : TAPEFILE

Demande des fils de : EOT de NIVEAU 1

1. quel est le type des fils : 5
(1 - 2 - 3 - 4 ou 5)

EOT ::= [pas de fils]

Type de composants fils
1 : Terminal
2 : Concaténation
3 : Sélection
4 : Itération
5 : pas de fils

PRESSEZ UNE CLE

ECRAN 5

ARBRE D'ENTREE : TAPEFILE

Demande des fils de : CHS de NIVEAU 2

1. quel est le type des fils : 3
(1 - 2 - 3 - 4 ou 5)
2. quel est le 3ieme fils : /
si plus de fils tapez : "/"

CHS ::= (NL / SP)

Type de composants fils
1 : Terminal
2 : Concaténation
3 : Sélection
4 : Itération
5 : pas de fils

PRESSEZ UNE CLE

ECRAN 6

```
DESIREZ-VOUS SAUVER L'ARBRE TAPEFILE SOUS FORME BNF
DANS UN FICHIER (O/N) :
```

ECRAN 7

```
DESIREZ-VOUS SAUVER L'ARBRE TAPEFILE SOUS FORME BNF
DANS UN FICHIER (O/N) :
QUEL EST LE NOM DU FICHIER : b:TAPEFILE
```

ECRAN 8

```
                ARBRE D'ENTREE  :  TAPEFILE

TAPEFILE ::= (PADD . BODY . EOT)                          niv : 1
PADD ::= ( * CHS)                                         niv : 2
BODY ::= ( * WORDS)                                       niv : 2
CHS ::= (NL / SP)                                         niv : 3
WORDS ::= (LETTERS . DELIM . PADD)                        niv : 3
LETTERS ::= (L . LS)                                      niv : 4
DELIM ::= (NL / SP)                                       niv : 4
PADD ::= ( * CHS)                                         niv : 4
LS ::= ( * L)                                             niv : 5
CHS ::= (NL / SP)                                         niv : 5

                                          PRESSEZ UNE CLE
```

ECRAN 9

```
SUR QUEL FICHIER SE TROUVE L'ARBRE DE SORTIE : b:OUTPUT
```

ECRAN 10

ARBRE DE SORTIE : OUTPUTFILE

```
OUTPUTFILE ::= (WORDS . EOF)                    niv : 1
WORDS ::= ( * WORD)                             niv : 2
WORD ::= (LETTERS . EW)                         niv : 3
LETTERS ::= (L . LS)                            niv : 4
LS ::= ( * L)                                   niv : 5
```

PRESSEZ UNE CLE

ECRAN 11

CHOIX DE LA POSSIBILITE DE RECHERCHE

1. vous connaissez les règles de traduction

2. on vous proposera toute les solutions
   de correspondances des deux arbres

PREFEREZ-VOUS LA PREMIERE SOLUTION (O/N) :

ECRAN 12

```
DEMANDE DES FONCTIONS DE TRADUCTION AU NIVEAU DES FEUILLES
---------------------------------------------------------------------------
COMPOSANT(S) FEUILLE(S) DE L'ARBRE DE SORTIE :

    1.  L          niv 4  |  3.  EW         niv 3  |  4.  EOF       niv 1
    2.  L          niv 5  |                        |
---------------------------------------------------------------------------
COMPOSANT(S) FEUILLE(S) D'ENTREE :
  LE COMPOSANT  "  L "  DE NIVEAU 4 (identifiant : 3)

    Y A-T-IL UNE CORRESPONDANCE DANS L'ARBRE DE SORTIE (O/N) :

    DONNEZ LE NUMERO IDENTIFIANT DU COMPOSANT FEUILLE DE SORTIE : 1
```

ECRAN 13

```
DEMANDE DES FONCTIONS DE TRADUCTION AU NIVEAU DES FEUILLES
---------------------------------------------------------------------------
COMPOSANT(S) FEUILLE(S) DE L'ARBRE DE SORTIE :

    1.  L          niv 4  |  3.  EW         niv 3  |  4.  EOF       niv 1
    2.  L          niv 5  |                        |
---------------------------------------------------------------------------
COMPOSANT(S) FEUILLE(S) D'ENTREE :
  LE COMPOSANT  "  NL "  DE NIVEAU 4 (identifiant : 5)

    Y A-T-IL UNE CORRESPONDANCE DANS L'ARBRE DE SORTIE (O/N) :
```

ECRAN 14

PRESENTATION DES REGLES DE TRANSFORMATION DES COMPOSANTS FEUILLES

REGLE(S) F-A-B :

1 . L	niv 4	- L	niv 4
2 . L	niv 5	- L	niv 5
3 . NL	niv 4	- EW	niv 3
4 . SP	niv 4	- EW	niv 3
5 . EOT	niv 1	- EOF	niv 1

REGLE(S) F-A-& :

1 . NL	niv 3	- &
2 . SP	niv 3	- &
3 . NL	niv 5	- &
4 . SP	niv 5	- &

REGLE(S) F-&-B :

PRESSEZ UNE CLE

ECRAN 15

CORRESPONDANCE FORMELLE ENTRE TAPEFILE et OUTPUTFILE

NIV 0 : TAPEFILE	--->	OUTPUTFILE
NIV 1 : PADD	--->	?
NIV 1 : BODY	--->	WORDS
NIV 1 : EOT	--->	EOF
NIV 2 : CHS	--->	?
NIV 2 : WORDS	--->	WORD
NIV 3 : NL	--->	?
NIV 3 : SP	--->	?
NIV 3 : LETTERS	--->	LETTERS
NIV 3 : DELIM	--->	EW
NIV 3 : PADD	--->	?
NIV 4 : L	--->	L
NIV 4 : LS	--->	LS
NIV 4 : NL	--->	EW
NIV 4 : SP	--->	EW
NIV 4 : CHS	--->	?
NIV 5 : L	--->	L
NIV 5 : NL	--->	?

NIV 5 : SP ---> ?

PRESSEZ UNE CLE

ECRAN 16

CORRESPONDANCE FORMELLE ENTRE TAPEFILE et OUTPUTFILE

NIV 0 : TAPEFILE	---> OUTPUTFILE	NIV 4 : NL	---> ?
NIV 1 : PADD	---> ?	NIV 4 : SP	---> ?
NIV 1 : BODY	---> ?	NIV 4 : CHS	---> ?
NIV 1 : EOT	---> ?	NIV 4 : ?	---> L
NIV 1 : ?	---> WORDS	NIV 4 : ?	---> LS
NIV 1 : ?	---> EOF	NIV 5 : L	---> ?
NIV 2 : CHS	---> ?	NIV 5 : NL	---> ?
NIV 2 : WORDS	---> ?	NIV 5 : SP	---> ?
NIV 2 : ?	---> WORD	NIV 5 : ?	---> L
NIV 3 : NL	---> ?		
NIV 3 : SP	---> ?		
NIV 3 : LETTERS	---> ?		
NIV 3 : DELIM	---> ?		
NIV 3 : PADD	---> ?		
NIV 3 : ?	---> LETTERS		
NIV 3 : ?	---> EW		
NIV 4 : L	---> ?		
NIV 4 : LS	---> ?		

CETTE CORRESPONDANCE VOUS CONVIENT-ELLE (O/N) :

PRESSEZ UNE CLE

ECRAN 19

AUCUNE CORRESPONDANCE ENTRE

CES DEUX ARBRES

PRESSEZ UNE CLE

ECRAN 20

DEMANDE DU FICHIER CONTENANT L'ARBRE DE CORRESPONDANCE

Sur quel fichier se trouve l'arbre de correspondance :

b:ARBRE

ECRAN 21

DEMANDE DU TYPE DES FICHIERS D'ENTREE ET DE SORTIE

Quel est le type des fichiers des données en entrée et en sortie :

Type = TEXT

ECRAN 22